BEA WebLogic Server®

Configuring and Managing WebLogic JMS

Version 9.0 BETA
Revised: December 15, 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

1. Introduction and Roadmap

2. Understanding JMS Resource Configuration

3. Configuring JMS System Resources

4. Configuring JMS Application Modules for Deployment

5. Configuring Clustered WebLogic JMS Resources

6. Tuning WebLogic JMS

7. Monitoring WebLogic JMS

CHAPTER 1

Introduction and Roadmap

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

Document Scope and Audience

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

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

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

Guide to This Document

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

Related Documentation

This document contains JMS-specific configuration and maintenance information.

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

- *Programming WebLogic JMS* is a guide to JMS API programming with WebLogic Server.
- *Configuring WebLogic Servers and Clusters* contains instructions on administering WebLogic Server clusters.
- *Deploying WebLogic Server Applications* 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* provides information about the benefits and usage of the system-wide WebLogic Persistent Store.
- *Configuring and Managing WebLogic Store-and-Forward Service* contains information about the benefits and usage of the Store-and-Forward service with clustered release 9.0 WebLogic JMS implementations.
- *Configuring and Managing WebLogic Message Bridge* for instructions on configuring 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.
- *WebLogic Server Performance and Tuning* contains information on monitoring and improving the performance of WebLogic Server applications.

JMS Samples and Tutorials for the JMS Administrator

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

Avitek Medical Records Application (MedRec) and Tutorials

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

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

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

As companion documentation to the MedRec application, BEA provides tutorials that provide step-by-step procedures for key development tasks, including JMS-specific tasks, such as:

JMS Examples in the WebLogic Server Distribution

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

Additional JMS Examples Available for Download

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

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

New and Changed JMS Features In This Release

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

- "JMS 1.1 Specification Support" on page 1-5
- Configuration and Administration Enhancements:
  - "Modular Configuration and Deployment of JMS Resources" on page 1-5
  - "Store-and-Forward for Highly Available Message Production" on page 1-5
  - A default, system-wide WebLogic Persistent Store
  - "Pause and Resume Message Operations on Destinations" on page 1-6
  - "Message Bridge Enhancements" on page 1-10
  - "Flexible and Simplified Destination Quotas" on page 1-9
- Message Management, Monitoring, and Diagnostics:
  - "Enhanced Run-time Message Management" on page 1-6
  - "More Transparency with Message Life Cycle Logging" on page 1-7
  - "Debug and Diagnostic Information More Readily Available" on page 1-7

JMS 1.1 Specification Support

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

Modular Configuration and Deployment of JMS Resources

JMS configurations in WebLogic Server 9.0 are stored as modules, defined by an XML file that conforms to the `weblogic-jmsmd.xsd schema`, similar to standard J2EE modules. An administrator can create and manage JMS modules as global system resources (similar to the way they were managed prior to WebLogic Server 9.0), as global standalone modules, or as modules packaged with an Enterprise Application. With modular deployment of JMS resources, you 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 EAR file and without extensive manual JMS reconfiguration.

See Understanding JMS Resource Configuration in *Configuring and Managing WebLogic JMS*.

Store-and-Forward for Highly Available Message Production

The JMS Store-and-Forward feature is built on the WebLogic Store-and-Forward (SAF) service to provide highly available JMS message production. For example, a JMS message producer connected to a local server instance can reliably forward messages to a remote JMS destination,

even though that remote destination may be temporarily unavailable when the message was sent. Persistent JMS messages are always forwarded with Exactly-once quality of service provided by the SAF. For non-persistent JMS messages, applications can also use the At-least-once and At-most-once qualities of service. JMS store-and-forward works with all WebLogic JMS features, including new features introduced in release 9.0.

JMS Store-and-forward is transparent to JMS applications. Existing JMS applications can take advantage of this feature without any code changes. In fact, an administrator need only configure imported JMS destinations within JMS module files, which then associate remote JMS destinations to local JNDI names. JMS client code still uses the existing JMS APIs to access the imported destinations. JMS store-and-forward is only for message production; therefore, JMS clients still need to consume messages directly from imported destinations.

For more information, see Configuring and Managing WebLogic Store-and-Forward Service.

Enhanced Run-time Message Management

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

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

Pause and Resume Message Operations on Destinations

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

More Transparency with Message Life Cycle Logging

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

When a JMS destination hosting the subject message is configured with message logging enabled, then each of the basic message life cycle events will generate a message log event in the JMS message log file. The content of the log always includes message ID and correlation ID, but you can also configure information like message type and user properties. 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.

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

`*USER_DOMAIN*\servers\*servername*\logs\jmsServers\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\weblogic90`.

Debug and Diagnostic Information More Readily Available

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

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

Strict Message Ordering with Unit-of-Order

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

be created programmatically with new JMS API extensions to the JMSMessageProducer interface, or administratively by specifying a Unit-of-Order on a connection factory.

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

Uniform Configuration of Distributed Destinations

WebLogic Server 9.0 introduces a new type of distributed destination, termed *Uniform Distributed Destination*, that greatly simplifies the management and development of distributed destination applications. In prior releases, in order to create a distributed destination, an administrator often needed to manually configure physical destinations to function as members of a distributed destination. This method provided the flexibility to create members that were intended to carry extra message load or have extra capacity; however, such differences often led to administrative and application problems because such a *weighted distributed destination* was not deployed consistently across a cluster. This type of distributed destination is now officially referred to as a *Weighted Distributed Destination*.

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

Access to JMS Applications from "C" Clients

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

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

Message-Driven Bean (MDB) Enhancements for JMS

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

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

Document Object Model (DOM) Support for XML Messages

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

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

Flexible and Simplified Destination Quotas

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

Improved Message Paging

The *message paging* feature for freeing up virtual memory during peak message load periods is now enabled by default on JMS servers. The threshold for triggering message paging is controlled by the Message Buffer Size parameter, which controls the amount of memory that is used to store message bodies in memory before they are paged out to disk. Once this threshold is crossed, the JMS server will write message bodies to a specified paging directory in an effort to reduce memory usage below this threshold. Additionally, administrators no longer need to create a dedicated message paging store since paged out messages can be stored in a directory on your file system. However, for the best performance you should specify that messages be paged to a directory other than the one used by the JMS server's persistent store.

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

Message ID Propagation Security Enhancement

This enhancement introduces the JMSXUserID property, which provides the ability to identify the user who produced a message when the message is received. With this property, the recipient has the option of querying the message property to find out who sent the message, and then could

possibly run different logic in the application, depending on who the initiator was. The message sender requests the user identity by using a connection factory with the AttachJMSXUserID attribute enabled. The delivery of the user identity is controlled by the AttachSender attribute on destinations, which can also be managed for multiple destinations by using a JMS template.

Message Bridge Enhancements

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

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

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

Automatic Compression of Large Messages

This feature enables the compression of messages that exceed a specified threshold size to improve the performance of sending large messages travelling across JVM boundaries. A threshold can be set programmatically using a new JMS API extension to the JMSMessageProducer interface, or administratively by either specifying a Compression-Threshold value on a connection factory or a SAF-Compression-Threshold value on a SAF remote context.

Once configured, message compression is triggered on producers for client sends, on connection factories for message receives and message browsing, or through SAF forwarding. Messages are compressed using GZIP an only occurs when message producers and consumers are not collocated on the same server instance. Decompression automatically occurs on the client side, and only when the message content is accessed. On the server side, messages always remains compressed, even when they are written to disk. Compressed messages may actually

inadvertently affect destination quotas since some message types actually grow larger when compressed.

Deprecated JMS Features, Methods, Interfaces, and Methods

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

Legacy JMS Resource Configuration Interfaces

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

- JMSDestCommonMBean
- JMSTopicMBean
- JMSQueueMBean
- JMSConnectionFactoryMBean
- JMSTemplateMBean
- JMSDestinationKeyMBean
- ForeignJMSServerMBean
- ForeignJMSDestinationMBean
- ForeignJMSConnectionFactoryMBean
- JMSDistributedTopicMBean
- JMSDistributedTopicMemberMBean
- JMSDistributedTopicMBean
- JMSDistributedQueueMemberMBean

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

For more information on configuring JMS resource modules, see Configuring JMS System Resources in *Configuring and Managing WebLogic JMS*.

JMS Helper APIs

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

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

For more information on using the JMS resource modules, see the JMSModuleHelper Javadoc.

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

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

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

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

JMS Session Pool and JMS Connection Consumer Interfaces

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

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

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

JMS Store Interfaces

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

- JMSStoreMBean
- JMSFileStoreMBean
- JMSJDBCStoreMBean

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

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

Pause and Resume Methods On the JMS Destination Runtime Interface

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

Message Purge Method on the JMS Durable Subscriber Runtime Interface

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

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

JMS Extensions: WLMessage Interface

The following methods in the WLMessage class have been deprecated:

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

These methods have been replaced by the following properties:

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

These new properties includes the corresponding setter methods.

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

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 objects and WebLogic Server.

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

Overview of JMS and WebLogic Server

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

What Is the Java Message Service?

An enterprise messaging system enables applications to asynchronously communicate with one another through the exchange of messages. A message is a request, report, and/or event that contains information needed to coordinate communication between different applications. A message provides a level of abstraction, allowing you to separate the details about the destination system from the application code.

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

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

The following figure illustrates WebLogic JMS messaging.

Figure 2-1 WebLogic JMS Messaging

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

WebLogic JMS Anatomy and Environment

The major components of the WebLogic JMS architecture include:

- JMS servers host a defined set of destinations (queues for point-to-point communication or topics for publish-and-subscribe) in a JMS module, with which client applications can interact, and any associated persistent storage that resides on a WebLogic Server instance.
- JMS modules contains configuration resources (destinations, connections factories, etc.), and are defined by XML documents that conform to the `weblogic-jmsmd.xsd` schema. [missing from this diagram!]
- 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 (file store or JDBC-accessible) for storing persistent message data.

Domain Configuration Resources: The Big Picture

In general, the WebLogic Server configuration file (`config.xml`) contains the configuration information required for a domain. This configuration information can be further classified into environment-related (or *system resource* definitions) and application-related information. Some examples of environment-related definitions are the identification and definition of JMS Servers, JDBC Datasources, 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 your organization's system administrator or MIS department. To accomplish these administrative tasks, the WebLogic administrator can use the WebLogic Administration Console, various command-line tools, such as WebLogic Scripting Tool (WLST), or JMX APIs for programmatic administration.

Some examples of application-related definitions that are highly independent of the domain environment, are the various J2EE application components configurations, such as EAR, WAR, JAR, RAR files, and which now includes JMS and JDBC modules. The application components are originally developed and packaged by an application development team, and may contain optional programs (compiled Java code) and respective configuration information (also called descriptors, which are mostly stored as XML files). In the case of JMS modules (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 domain. WebLogic Server provides tools for deploying applications, such as the Administration Console, the `weblogic.Deployer` command-line utility, and WebLogic Workshop (not available fo Beta). The process of deploying an application links the application components to the environment-specific resource definitions, such as which server instances should host a given application component (targeting), and the WebLogic persistent store to use for persisting JMS messages.

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

What Are JMS Configuration Resources?

In prior releases, all JMS configuration information was stored in a domain's configuration file. In WebLogic Server 9.0, JMS configurations (destinations, connections factories, templates, etc.) are stored as module descriptor files, defined by XML documents that conform to the `weblogic-jmsmd.xsd` schema. JMS modules do not include JMS server definitions, which are stored in the domain configuration file.

You create and manage JMS resources either as *system modules*, similar to the way they were managed prior to version 9.0, or as *application modules*. JMS application modules are a WebLogic-specific extension of J2EE modules and can be deployed either with a J2EE application (as a packaged resource) or as standalone modules that are globally available.

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

JMS System Modules

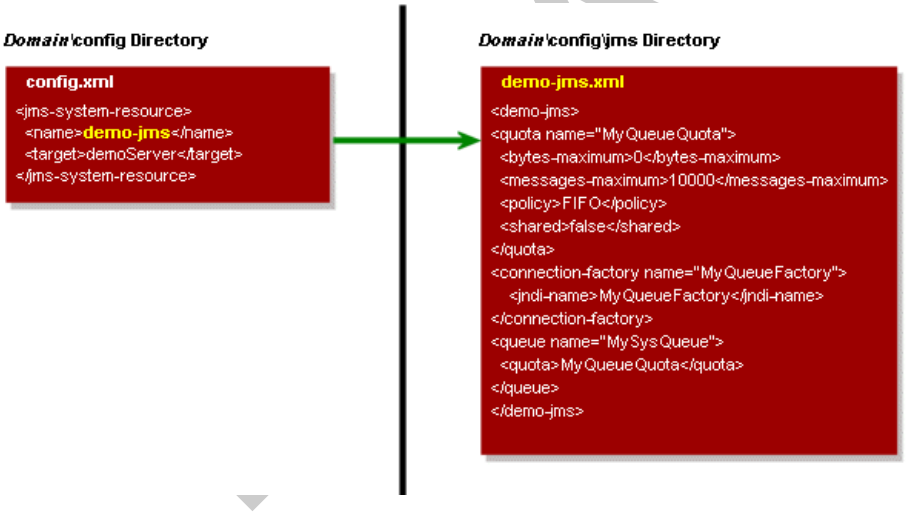
When you create a JMS module using the Administration Console or using the WebLogic Scripting Tool (WLST), WebLogic Server creates a JMS module in the `config\jms` subdirectory of the domain directory, and adds a reference to the module in the domain's `config.xml` file as a `JMSSystemResource` element. This reference includes the path to the JMS module file and a list of target servers and clusters on which the module is deployed. The JMS

module conforms to the `weblogic-jms.xsd` schema, as described in "JMS Schema" on page 4-2.

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



The following configuration elements are defined as part of a system or application JMS module:

- Queue and topic destinations, as described in [TBD to Console Help].
- Connection factories, as described in [TBD to Console Help].
- JMS templates, as described in [TBD to Console Help].
- Destination keys, as described in [TBD to Console Help].
- Quota, as described in [TBD to Console Help].

- Uniform distributed destinations (system-defined), as described in [TBD to Console Help].
- Weighted distributed destinations (user-defined), as described in [TBD to Console Help].
- Foreign servers, destinations, and connection factories, as described in [TBD to Console Help].
- JMS Store-and-Forward configuration items, as described in [TBD to Console Help].

All other JMS resources must configured by the administrator in the same manner as in prior releases. This includes:

- JMS servers, as described in "JMS Servers" on page 2-9
- Persistent stores, as described in "Persistent Stores" on page 2-10
- Messaging bridges, as described in "Messaging Bridges" on page 2-10.

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

JMS Application Modules

JMS configuration resources can also be managed as deployable application modules, similar to standard J2EE descriptor-based modules. JMS Application modules can be deployed either with a J2EE application as a *packaged module*, which is available only to the enclosing application, or as a *standalone module* that provides global access to the resources defined in that module.

Application modules are generally created and packaged by an application developer and are then deployed, managed, and tuned by a WebLogic administrator. As discussed in "Domain Configuration Resources: The Big Picture" on page 2-3, JMS application modules do not contain compiled Java programs as part of the package, enabling administrators or application developers to create and manage JMS resources on demand.

For more information about JMS application modules, see Chapter 4, "Configuring JMS Application Modules for Deployment."

JMS Standalone Modules

A JMS application module can be deployed by itself as a *standalone module*, in which case the module is available to the server or cluster targeted during the deployment process. JMS resources deployed in this manner can be reconfigured using the `weblogic.Deployer` utility or the Administration Console, but are not available through JMX or WLST. For more information

about preparing standalone JMS modules for deployment, see "Deploying Standalone JMS Modules" on page 4-9.

JMS Packaged Modules

Application modules can also be included as part of an J2EE Enterprise Application Archive (EAR), as a *packaged module*. Packaged modules are bundled with an EAR file or in an exploded EAR directory, and are referenced in the `weblogic-application.xml` deployment descriptor. The resource module is deployed along with the Enterprise Application, and is available only to the enclosing application. Using packaged modules ensures that an application always has access to required resources and simplifies the process of moving the application into new environments. For more information about preparing packaged JMS modules for deployment, see "Deploying JMS Modules That Are Packaged In an Enterprise Application" on page 4-2.

JMS Schema

In support of the new modular deployment model for JMS resources in WebLogic Server 9.0, BEA now provides a schema for WebLogic JMS objects: `weblogic-jmsmd.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 the schema.

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

Ownership of Configured JMS Resources

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

- WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JMS modules. These JMS modules are considered system modules. See "JMS System Modules" on page 2-4 for more details.
- Application developers create modules in WebLogic Workshop (not available at Beta) or another development tool that supports creating an XML descriptor file, then package the JMS modules with an application and pass the application to a WebLogic Administrator to deploy. These JMS modules are considered application modules. See "JMS Application Modules" on page 2-6 for more details.

Comparing System and Application Module Capabilities

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

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

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

Module Type	Created with	Dynamically Add/Remove Modules	Modify with JMX Remotely	Modify with JSR-88 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	WebLogic Workshop (not for Beta), another IDE, or an 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 4-1and Deploying Applications in *Deploying Applications to WebLogic Server*.

JMS Interop Modules

During the 9.0 upgrade process, a prior JMS configuration is converted to a JMS system module named `interop-jms.xml` (with the exception of JMS servers and legacy JMS stores, which continue to be stored in the domain's `config.xml` file).

JMS Servers

JMS servers are environment-related configuration entities that act as management containers for JMS queue and topic destinations in a JMS module that are targeted to them. A JMS server's primary responsibility for its 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. For example, JMS servers can

For descriptions of all the configuration and runtime options available for JMS servers, see JMSServerBean in the *WebLogic Server MBean Reference*.

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

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

- Because destinations are now encapsulated in JMS modules, they are no longer nested under JMS servers in the configuration file. However, the prior "parent/child" relationship between JMS servers and destinations is still maintained since destination elements within a JMS module are targeted to JMS servers. This way, JMS servers still manage persistent messages, durable subscribers, message paging, and, optionally, message and/or byte thresholds for destinations targeted to them. Multiple JMS modules can be targeted to each JMS server in a domain.
- JMS servers support the new WebLogic Persistent Store that is available to multiple subsystems and services within a server instance, as described in "Persistent Stores" on page 2-10.
  - JMS servers can store both persistent and non-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 (File and JDBC), JMS servers now support user-defined WebLogic File Stores or JDBC stores, which provide better performance and more capabilities than the legacy JMS Stores. (The legacy JMS Stores are supported in this release for backward compatibility.)

- JMS servers support an improved message paging mechanism. For more information on message paging, see "Paging Out Messages To Free Up Memory" on page 6-2.
  - 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.
  - The temporary paging of non-persistent messages is enabled by default and is controlled by the value set on the Message Buffer Size option. When the total size of non-pending, un-paged messages reaches this setting, a JMS server will attempt to reduce its memory usage by paging out non-persistent 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.
- JMS servers can be undeployed and redeployed without having reboot WebLogic Server.

For more information on configuring JMS servers, see "JMS Server Tasks" on page 3-8.

Persistent Stores

The WebLogic Persistent Store provides a built-in, high-performance storage solution for all subsystems and services that require persistence, especially subsytems like JMS that require the creation and deletion of short-lived, data objects, such as transactional messages for JMS servers. Each 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 *Designing and Configuring WebLogic Server Environments*.

Messaging Bridges

The WebLogic 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 "Configuring a Messaging Bridge" in *Configuring and Managing WebLogic Messaging Bridge*.

Ways to Configure WebLogic JMS Resources

JMS resources can be configured in a number of ways:

Administrators

WebLogic Administrators typically use the Administration Console or the WebLogic Scripting Tool (WLST) to create and deploy (target) JMS system modules.

- The WebLogic Server Administration Console enables you to configure, modify, and target JMS-related resources:
  - JMS servers, as described in [TBD for Console Help]
  - JMS module resources, as described in [TBD for Console Help]
  - Persistent stores, as described in *Using the WebLogic Persistent Store* in *Designing and Configuring WebLogic Server Environments*.
  - Store-and-Forward services for JMS, as described in *Configuring and Managing WebLogic Store-and-Forward Service*.
- The WebLogic Scripting Tool (WLST) is a command-line scripting interface that allows system administrators and operators to initiate, manage, and persist WebLogic Server configuration changes interactively or by using an executable script. For more information, see "WebLogic Scripting Tool".
- WebLogic Java Management Extensions (JMX) is the J2EE solution for monitoring and managing resources on a network. For more information see, "Programming WebLogic Management Services with JMX".

Application Developers

Developers create application modules in WebLogic Workshop (not available at Beta) or another development tool that supports creating an XML descriptor file, then package the JMS modules with an application and pass the application to a WebLogic Administrator to deploy.

- Deploying JMS application modules, as described in "Deploying JDBC and JMS Application Modules."
- The JMSModuleHelper extension interface contains methods to create and manage JMS module configuration resources in a given module. For more information, see the JMSModuleHelper Javadoc.

WebLogic Server Value-Added JMS Features

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

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

Note: This section lists only the value-added features for release 8.1 and earlier. For a comprehensive listing of the new WebLogic JMS feature introduced in release 9.0, see "New and Changed JMS Features In This Release" on page 1-4.

Enterprise-Grade Reliability

- *Out-of-the-box transaction support*:
  - Fully supports transactions, including distributed transactions, between JMS applications and other transaction-capable resources using the Java Transaction API (JTA), as described in "Using Transaction 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), as described in "Using the WebLogic Persistent Store" in *Designing and Configuring WebLogic Server Environments*.
- *Supports connection clustering* using multiple connection factories on multiple WebLogic Servers, as described in "Configuring WebLogic JMS Clustering" on page 5-1.
- *Distributed destinations* that provide higher destination availability, load balancing, and failover support in a cluster, as described in "Using Distributed Destinations" in *Programming WebLogic JMS*.
- *Failed server migration* for manually restarting JMS servers on another WebLogic Server instance in a cluster, as described in "Configuration Steps for JMS Migration" on page 5-5.

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

Enterprise-Level Features

- *Message paging* automatically kicks in during peak load periods to free up virtual memory.
- *Message flow control* during peak load periods, including blocking overactive senders, as described in "Controlling the Flow of Messages on JMS Servers and Destinations" on page 6-3.
- *Timer services* available for scheduled message delivery, as described in "Setting Message Delivery Times" in *Programming WebLogic JMS*.
- *Multicasting of messages* for simultaneous delivery to many clients using IP multicast, as described in "Using Multicasting with WebLogic Server" in *Programming WebLogic JMS*.
- *Supports messages containing XML* (Extensible Markup Language), as described in "Defining XML Message Selectors Using the XML Selector Method" in *Programming WebLogic JMS*.
- *Thin application client* `.JAR` *that provides full WebLogic Server J2EE functionality*, including JMS, yet greatly reduces the client-side WebLogic footprint, as described in "WebLogic JMS Thin Client" in *Programming Stand Alone Clients*.
- *Automatic pooling of JMS client resources in server-side applications* via JMS resource-reference pooling. Server-side applications use standard JMS APIs, but get automatic resource pooling, as described in see "Enhanced 2EE Support for Using WebLogic JMS With EJBs and Servlets" in *Programming WebLogic JMS*.

Tight Integration With WebLogic Server

- *JMS can be accessed locally by server-side applications without a network call* because the destinations can exist on the same server as the application.
- *Uses same ports, protocols, and user identities as WebLogic Server* (T3, IIOP, and HTTP tunnelling protocols, optionally with SSL).

- *Web Services, EJBs, and servlets* supplied by WebLogic Server can work in close concert with JMS.
- *Can be configured and monitored by using the same Administration Console*, or by using the JMS API.
- *Complete JMX administrative and monitoring APIs*, as described in *Programming WebLogic Management Services 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 *Introduction to WebLogic Security* and "JMS (Java Message Service) Resources" in *Securing WebLogic Resources*.

Interoperability With Other Messaging Services

- *Messages forwarded transactionally by the WebLogic Messaging Bridge* to other JMS providers — as well as to other instances and versions of WebLogic JMS, as described see *Configuring and Managing WebLogic Messaging Bridge*.
- *Supports mapping of other JMS providers* so their objects appear in the WebLogic JNDI tree as local JMS objects. Can also reference remote instances of WebLogic Server in another cluster or domain in the local JNDI tree. For more information, see "Accessing Remote or Foreign Providers" on page 3-12.
- *Uses MDBs to transactionally receive messages* from multiple JMS providers, as described in "Message-Driven EJBs" in *Using WebLogic EJB*.
- *Reliable Web Services integration* with JMS as a transport.
- *Automatic transaction enlistment of non-WebLogic JMS client resources* in server-side applications via JMS resource-reference pooling. Server-side applications use standard JMS APIs, but get automatic transaction enlistment, as described in see "Enhanced 2EE Support for Using WebLogic JMS With EJBs and Servlets" in *Programming WebLogic JMS*.
- *Seamless integration with BEA Tuxedo messaging provided by WebLogic Tuxedo Connector* as described in "How to Configure the Tuxedo Queuing Bridge" in the *WebLogic Tuxedo Connector Administration Guide*.

Clustered WebLogic JMS

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

For more information about configuring clustering for WebLogic JMS, see "Configuring WebLogic JMS Clustering" on page 5-1. For detailed information about WebLogic Server clustering, see *Using WebLogic Server Clusters*.

The advantages of clustering include the following:

- *Load balancing of destinations across multiple servers in the cluster*
  - An administrator can establish load balancing of destinations across multiple servers in the cluster by configuring multiple JMS servers and 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.
  - An administrator can configure multiple physical destinations as members of a single distributed destination set within a cluster. Producers and consumers are able to send and receive to the distributed destination. In the event of a single server failure within the cluster, WebLogic JMS then distributes the load across all available members within the distributed destination set. For more information on distributed destinations, see "JMS Distributed Destination Tasks" on page 5-7.
- *Cluster-wide, transparent access to destinations from any server in the cluster*

  An administrator can establish cluster-wide, transparent access to destinations from any server in the cluster by either enabling 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. This way, each connection factory can be deployed on multiple WebLogic Server instances.

  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.

Connection factories are described in more detail in "Understanding JMS Resource Configuration" on page 2-1.

- *Scalability* is provided by:
  - Load balancing of destinations across multiple servers in the cluster, as described previously.
  - Distribution of application load across multiple JMS servers 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*

  As an "exactly-once" service, WebLogic JMS takes advantage of the migration framework implemented in WebLogic Server for clustered environments. This allows WebLogic JMS to respond properly to migration requests and to bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure. For more information, see "Configuring JMS Migratable Targets" on page 5-4.
- *Server affinity for JMS Clients*

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

Note: Automatic failover is not supported by WebLogic JMS for this release. For information about performing a manual failover, refer to "Recovering from a WebLogic Server Failure" in *Programming WebLogic JMS*.

CHAPTER 3

Configuring JMS System Resources

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

Using the Administration Console to Configure JMS Resources

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

Using the Administration Console, you define configuration options to:

- Configure JMS servers and target a server instances on which the JMS servers are deployed, as described in "JMS Server Tasks" on page 3-8.
- Configure JMS system modules, including queue and topic destinations, connection factories, JMS templates, destination sort keys, destination quota, distributed destinations, foreign JMS servers, and store-and-forward parameters.
- Optionally, create a custom persistent store for JMS, either file-based or a JDBC-accessible table, as described in "Using the WebLogic Persistent Store" in the *Designing and Configuring WebLogic Server Environments*.
- Enable any desired WebLogic JMS features, such as:
  - Server clustering using multiple connection factories, as described in "Configuring Clustered WebLogic JMS Resources" on page 5-1.
  - Higher availability and load balancing for queues and topics across a cluster by using *distributed* destinations, as described in "Using Distributed Destinations" in *Programming WebLogic JMS*.
  - Persistent messages and durable subscribers, as described in "Setting Up Durable Subscriptions" in *Programming WebLogic JMS*.
  - Multicasting of messages for delivery to a select group of hosts using an IP multicast address, as described in "Using Multicasting with WebLogic Server" in *Programming WebLogic JMS*.
  - Controlling message flow during peak load periods, including blocking message producers, as described in "Controlling the Flow of Messages on JMS Servers and Destinations" on page 6-3.
- Configure a WebLogic Messaging Bridge to forward messages (including transactional messages) between any two messaging products, including separate implementations of WebLogic JMS, as described in *Configuring and Managing WebLogic Messaging Bridge*.

Modifying Default Values for Configuration Options

WebLogic JMS provides default values for some configuration options; you must provide values for all others. If you specify an invalid value for any configuration option, or if you fail to specify a value for an option for which a default does not exist, WebLogic Server will not boot JMS when you restart it.

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

To configure WebLogic JMS options, follow the procedures outlined in this section to configure and manage JMS resources. Once WebLogic JMS is configured, applications can send and receive messages using the JMS API. For more information about developing basis WebLogic JMS applications, refer to "Developing a Basic JMS Application" in *Programming WebLogic JMS*.

When migrating from a previous release of Weblogic Server, the configuration information is converted automatically, as described in "Porting WebLogic JMS Applications" in *Programming WebLogic JMS*.

Starting WebLogic Server and Configuring JMS

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

Starting the Default WebLogic Server

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

For complete information about starting the Administration Server, see "Overview of Starting and Stopping Servers" in the *Managing Server Startup and Shutdown*.

Starting the Administration Console

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

manage a WebLogic Server domain, see "The WebLogic Server Administration Console" in *Administration Console Online Help*.

Main Steps for Configuring a Basic JMS System Module

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

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

1. For storing persistent messages in a file-based store, you can simply use the server's default Persistent Store, which requires no configuration on your part. However, if you would rather create a dedicated file store for JMS, follow these steps:

   a. Create a directory on the file system where the file store will be kept.

   b. Click the Services →Persistent Stores node to open the Persistent Stores page.

   c. Click the New button and select Create File Store on the pop-up menu.

   d. On the Create a New File Store page, provide a name, select a valid target WebLogic Server instance, and specify the pathname to the file store directory you created.

   e. Click Finish.

   For more information on configuring a custom file store, see "Creating a Custom File Store Tasks" in the *Designing and Configuring WebLogic Server Environments*.

2. For storing persistent messages in a JDBC-accessible database, you must create a JDBC store. (If you need to create a JDBC data source or multi data source, then first complete steps A and B; otherwise, you can skip to step C.)

   a. Create a Expand the Services →JDBC →Data Source or Multi Data Source node.

   b. Create a JDBC data source or multi data source, as explained in "Configuring and Deploying JDBC Resources" in *Configuring and Managing WebLogic JDBC*.

   c. Expand the Services →Persistent Stores node, then click the New button and select Create JDBC Store on the pop-up menu.

   d. On the Create a New JDBC Store page, provide a name, select a valid target WebLogic Server instance, select a configured data source, and enter a prefix name.

e. Click Finish.

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

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

   a. Click the Services →JMS →Servers node to open the JMS Servers page.

   b. Click the New button to open the Create a New JMS Server page.

   c. Provide a name and, optionally, select a persistent store if one was configured for JMS. Otherwise, leave this property set to None when using the default Persistent Store.

   d. Click Next.

   e. On the Select Targets page, select a server instance or migratable target where you want to deploy the JMS server.

   f. Click Finish.

   For more information, see "JMS Servers" on page 2-9.

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

   a. Click the Services →JMS →Modules node to open the Modules page.

   b. Click the New button to open the Create a New JMS Module page.

   c. Provide a name, a descriptor name, and optionally, specify where in the domain the system module will reside. Otherwise, it will default to the domain's `config/jms` directory.

   d. Click Next.

   e. On the Select Targets page, select a server instance or cluster where you want to deploy the JMS system module.

   f. Click Finish.

   For more information, see "JMS System Modules" on page 2-4.

5. Before creating any queue or topic resources in your 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:

– Click the Services →JMS →Quota node to define to configure quota objects and then have destinations refer to them. Destinations can be assigned their own quotas; multiple destinations can share a quota; or destinations can share the JMS server's quota.

– Click the Services →JMS →Templates node to create JMS templates, which allow you to define multiple destinations with similar option settings. You also need a JMS template to create temporary queues. For more information, see "JMS Template Tasks" on page 3-11.

– Click the Services →JMS →Destination Keys node to create custom sort orders of messages as they arrive on a destination. By default, messages are sorted in FIFO (first-in, first-out), which sorts ascending based on each message's unique JMSMessageID. However, if you want to configure a different sorting scheme, use the Destination Keys node to define another sort order (like LIFO, last-in, first-out). For more information, see "Destination Keys Tasks" on page 3-11.

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

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

   a. Click the Services →JMS →Topics node to open the Topics page.

   b. If you have more than one JMS module configured, select the desired module from the Topics In Module drop box.

   c. Click the New button to open the Create a JMS Topic page.

   d. Name the topic, provide a JNDI name, and optionally, select a Template if you already created one in the current module.

   e. Click Next.

   f. On the Select Targets page, select one or more JMS servers where you want to deploy the topic.

   g. Click Finish.

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

   a. Click the Services →JMS →Queues node to open the Queues page.

   b. If you have more than one JMS module configured, select the desired module from the Queues In Module drop box.

   c. Click the New button to open the Create a JMS Queue page.

d. Name the queue, provide a JNDI name, and optionally, select a Template if you already created one in the current module.

e. Click Next.

f. On the Select Targets page, select one or more JMS servers where you want to deploy the queue.

g. Click Finish.

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

   a. Click the Services →JMS →Connection Factory node to open the Connection Factory page.

   b. If you have more than one JMS module configured, select the desired module from the Connection Factories In Module drop box.

   c. Click the New button to open the Create a New Connection Factory page.

   d. Name the connection factory and provide a JNDI name.

   e. Click Next.

   f. On the Select Targets page, select one or more server instances, cluster, or JMS servers on which to deploy the connection factory.

   g. Click Finish.

   For more information about using the default connection factories, see "Using a Default Connection Factory" on page 3-9.

Guidelines for Configuring Advanced JMS System Module Resources

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

- Use the Distributed Destinations node to make your physical destinations part of a single distributed destination set within a server cluster. For more information, see "JMS Distributed Destination Tasks" on page 5-7.
- Use the Store-and-Forward node to reliably forward messages to remote destinations, even if a destination is unavailble at the time a messsage is sent, as described in *Configuring and Managing WebLogic Store-and-Forward Service*.
- Create JMS Session Pools, which enable your applications to process messages concurrently, and Connection Consumers (queues or topics) that retrieve server sessions and process messages. For more information, see "Session Pools Tasks" on page 3-16 and "Connection Consumers Tasks" on page 3-16.

  Note: JMS session pool and connection consumer configuration objects are deprecated in this release of WebLogic Server. They are not a required part of the J2EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are a required part of J2EE.

JMS Configuration Naming Rules

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

The one exception to this unique naming rule, however, is for JMS queue and topic destinations on different JMS servers in a domain, as follows:

- Queue destinations *can* use the same name as other queues on different JMS servers; topic destinations can also use the same name as other topics on different JMS servers.
- Queue destinations *cannot* use the same name with topic destinations, nor can queues nor topics use the same name as any other configurable objects.

JMS Server Tasks

A JMS server manages connections and message requests on behalf of clients. Use the Services →JMS →Server node to configure a JMS server and assign it to either an independent WebLogic Server instance or a migratable server target where it will be deployed.

JMS Connection Factory Tasks

Connection factories are objects that enable JMS clients to create JMS connections. A connection factory supports concurrent use, enabling multiple threads to access the object simultaneously. WebLogic JMS provides preconfigured "default connection factories" that can be enabled or disabled on a per-server basis, as described in "Using a Default Connection Factory" on page 3-9. Otherwise, you can configure one or more connection factories to create connections with predefined options that better suit your application — as long as each connection factory is uniquely named. WebLogic Server adds them to the JNDI space during startup, and the application then retrieves a connection factory using WebLogic JNDI.

You can establish cluster-wide, transparent access to JMS destinations from any server in the cluster, either by using the default connection factories for each server instance, or by configuring one or more connection factories and targeting them to one or more server instances in the cluster. This way, each connection factory can be deployed on multiple WebLogic Servers. For more information on configuring JMS clustering, see "Configuring WebLogic JMS Clustering" on page 5-1.

Using a Default Connection Factory

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

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

You only need to configure a new connection factory if the pre-configured settings of the default factories are not suitable for your application. 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, as shown in the following table:

Table 3-1 Default Connection Factory Settings for Transacted Sessions (XA)

Default Connection Factory. . .	*XAConnectionFactoryEnabled setting is. . .*
weblogic.jms.ConnectionFactory	False
weblogic.jms.XAConnectionFactory	True

An XA factory is required for JMS applications to use JTA user-transactions, but is not required for transacted sessions. For more information about using transactions with WebLogic JMS, see "Using Transactions with WebLogic JMS" in *Programming WebLogic JMS*.

All other default factory configuration options are set to the same default values as a user-defined factory. For more information about the XA Connection Factory Enabled option, and to see the default values for the other connection factory options, see TBD.

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

Note: Some connection factory options are dynamically configurable. When dynamic options are modified at run time, the new values become effective for new connections only, and do not affect the behavior of existing connections.

JMS Queue and Topic Destination Tasks

A JMS destination identifies a queue (point-to-point) or topic (publish/subscribe) for a JMS server. After configuring a JMS server, configure one or more queue or topic destinations for each JMS server.

You configure destinations explicitly or by configuring a destination template that can be used to define multiple destinations with similar option settings, as described in "JMS Template Tasks" on page 3-11.

You can configure multiple physical destinations as members of a single distributed destination set within a cluster. Therefore, if one WebLogic Server instance within the cluster fails, then the other instances hosting the same distributed destination will continue to provide service to JMS producers and consumers. For more information, see "JMS Distributed Destination Tasks" on page 5-7.

Note: 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 must be a destination that is configured on the local JMS server. For more information, see "Configuring an Error Destination for Undelivered Messages" in *Programming WebLogic JMS*.

Some destination options are dynamically configurable. When options are modified at run time, only incoming messages are affected; stored messages are not affected.

JMS Template Tasks

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

- You do not need to re-enter every option setting each time you define a new destination; you can use the JMS template and override any setting to which you want to assign a new value.
- You can modify shared option settings dynamically simply by modifying the template.

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

- If the destination that is using a JMS template specifies an override value for an option, the override value is used.
- If the destination that is using a JMS template specifies a message redelivery value for an option, that redelivery value is used.
- The Name option is not inherited by the destination. This name is valid for the JMS template only. You must explicitly define a unique name for all destinations. For more information, see "JMS Configuration Naming Rules" on page 3-8.
- The JNDI Name, Enable Store, and Template options are not defined for JMS templates.
- The Multicast options are not defined for JMS templates because they apply only to topic destinations.

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. If you do not do so, the configuration information remains incomplete, the WebLogic JMS configuration fails, and the WebLogic JMS does not boot.

Destination Keys Tasks

Use destination keys to define the sort order for messages that arrive on a specific destination.

Accessing Remote or Foreign Providers

WebLogic JMS enables you to reference foreign (that is, third-party) JMS providers within a local WebLogic Server JNDI tree. Using the Foreign Server node, 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 configuration can also be used to reference remote instances of WebLogic Server in another cluster or domain in the local WebLogic JNDI tree.

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

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

The following sections provide more information on how the Foreign Server node works, configuration instructions, 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 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 a Foreign Server

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

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.

Continue by configuring a connection factory and destination objects. You can configure one or more connection factories and destinations (queues or topics) for each foreign server.

Creating a Foreign Connection Factory

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

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

Continue by configuring the destination objects. You can configure one or more destinations (queues or topics) for each foreign JMS server.

Creating a Foreign Destination

A *Foreign Destination* represents either a queue or a topic. It contains the destination JNDI name that is looked up on the foreign JNDI provider and the JNDI name that the destination is mapped

to on the local WebLogic Server. When the foreign destination is looked up on the local server, a lookup is performed on the remote JNDI directory, and the destination object is returned from that directory.

Sample Configuration for MQSeries JNDI

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

Table 3-2 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

For detailed information about foreign server, connection factory, and options, and the valid and default values for them, refer to the following sections:

- TBD
- TBD
- TBD

Session Pools Tasks

Note: Session pool configuration objects are deprecated in this release of WebLogic Server. They are not a required part of the J2EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are a required part of J2EE.

Server session pools enable an application to process messages concurrently. After you define a JMS server, optionally, 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 session pools are restarted.

For more information about creating session pools, see Defining Server Session Pools in *Programming WebLogic JMS*

Connection Consumers Tasks

Note: Connection consumer configuration objects are deprecated in this release of WebLogic Server. They are not a required part of the J2EE specification, do not support JTA user transactions, and are largely superseded by message-driven beans (MDBs), which are a required part of J2EE.

Connection consumers are queues (Point-To-Point) or topics (Pub/Sub) that retrieve server sessions and process messages. After you define a session pool, configure one or more connection consumers for each session pool.

For more information about creating connection consumers, see "Defining Server Session Pools in *Programming WebLogic JMS*

CHAPTER 4

Configuring JMS Application Modules for Deployment

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

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

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

JMS Schema

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

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

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

Deploying JMS Modules That Are Packaged In an Enterprise Application

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

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

Creating Packaged JMS Modules

You create packaged JMS modules using WebLogic Workshop (not available for Beta), or any another development tool that supports creating an XML descriptor file. You then deploy and manage standalone modules using JSR 88-based tools, such as the `weblogic.Deployer` utility, the Administration Console, or WebLogic Workshop (not available for Beta).

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

JMS Packaged Module Requirements

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

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

Main Steps for Creating Packaged JMS Modules

1. Create a JMS system module using the Administration console, as described in "Main Steps for Configuring a Basic JMS System Module" on page 3-4.
2. Configure the necessary resources for the module, such as connection factory and queues or topics, and save your changes.
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 *application-scoped* to only the application, delete the `JNDI-Name` attribute of the module.
5. Add references to the JMS resources in the module to all applicable J2EE application component's descriptor files, as described in "Referencing a Packaged JMS Module In Deployment Descriptor Files" on page 4-4.
6. Package all application modules in an EAR, as described in "Packaging an Enterprise Application With a JMS Module" on page 4-8.
7. Deploy the EAR, as described in "Deploying a Packaged JMS Module" on page 4-8.

Referencing a Packaged JMS Module In Deployment Descriptor Files

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

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

Referencing JMS Modules In a weblogic-application.xml Descriptor

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

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


Referencing JMS Modules In a WebLogic Application

Within any `weblogic-`*`foo`* descriptor file, such as EJB (`ejb-jar.xml`) or WebApp (`web.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>
  <res-ref-name>jms/OrderQueue</res-ref-name>
  <resource-link>Workflows#OrderQueue</resource-link>
</resource-env-description>
```

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

Referencing JMS Resources In a J2EE Application

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

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

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

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

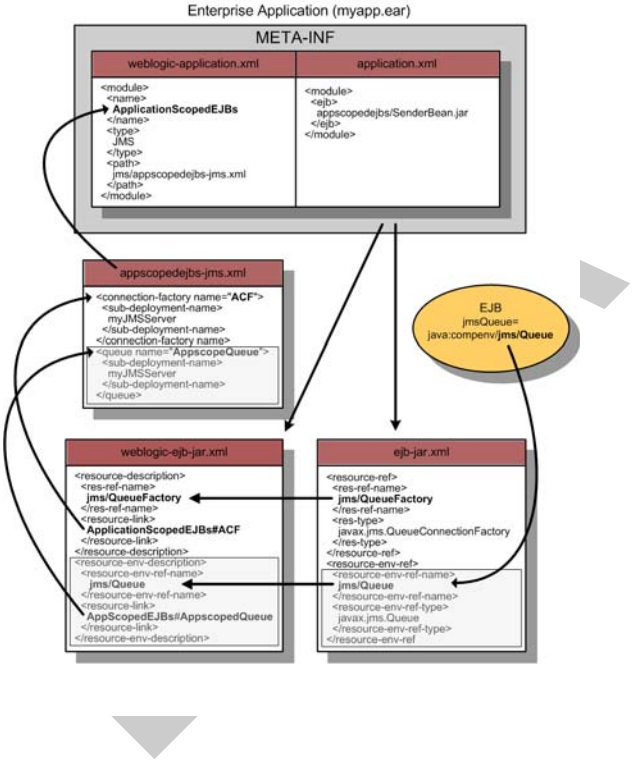
Sample of a Packaged JMS Module In an EJB Application

The following code snippet is an example of the packaged JMS module, appscopedejbs-jms.xml, referenced by the descriptor files in Figure 4-1 below.

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

Figure 4-1 illustrates how a JMS connection factory and queue resources in a packaged JMS module are referenced in an EJB EAR file.

Figure 4-1 Relationship Between a JMS Module and Descriptors In an EJB Application



Packaged JMS Module References In weblogic-application.xml

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

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

Packaged JMS Module References In ejb-jar.xml

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

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

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

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

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

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

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

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

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

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

the JMS module *AppScopedEJBs* containing the connection factory *ACF*, would have a name *AppScopedEJBs#ACF*.

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

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

Packaging an Enterprise Application With a JMS Module

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

Deploying a Packaged JMS Module

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

A recommended best practice for other application components is to use the `java:comp/env` JNDI environment in order to retrieve references to JMS entities, as described in "Referencing JMS Resources In a J2EE Application" on page 4-5. (However, this practice is not required.)

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

Deploying Standalone JMS Modules

A JMS application module can be deployed as a standalone resource using the `weblogic.Deployer` utility or the Administration Console (not available at Beta), in which case the module is typically available to the server or cluster targeted during the deployment process. JMS modules deployed in this manner are called *standalone modules*. Depending on how they are targeted, the resources inside standalone JMS modules are globally available in a cluster or locally on a server instance.

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

Creating Standalone JMS Modules

You can create JMS standalone modules using WebLogic Workshop (not available for Beta), or any another development tool that supports creating an XML descriptor file (such as XML Spy). You then deploy and manage standalone modules using JSR 88-based tools, such as the `weblogic.Deployer` utility, the Administration Console, or WebLogic Workshop (not available for Beta).

Note: As a workaround for Beta, 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 delete or change the `Name` and `JNDI-Name` elements of the module before deploying it with your application to avoid a naming conflict in the namespace.

JMS Standalone Module Requirements

A standalone JMS module must meet the following criteria:

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

Main Steps for Creating Standalone JMS Modules

1. Create a JMS system module using the Administration console, as described in "Main Steps for Configuring a Basic JMS System Module" on page 3-4.
2. Configure the necessary resources for the module, such as a Connection Factory and Queues or Topics, and save your changes.
3. The system module is saved in `config\jms` subdirectory of the domain directory, with a "`-jms.xml`" suffix.
4. Copy the system module to a new location and then:
   a. Give the module a unique name within the domain namespace.
   b. To make the module *globally available*, uniquely rename the `JNDI-Name` attributes of the resources in the module.
   c. If necessary, modify any other tunable values, such as destination thresholds or connection factory flow control parameters.
5. Deploy the module, as described in "Deploying Standalone JMS Modules" on page 4-10.

Sample of a Simple Standalone JMS Module

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

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

Deploying Standalone JMS Modules

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

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

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

Tuning Standalone JMS Modules

JMS resources deployed within *standalone modules* can be reconfigured using 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 unavailable through JMX or the WebLogic Scripting Tool (WLST). Additionally, standalone resources cannot be dynamically added or deleted with any WebLogic utility and must be redeployed.

CHAPTER 5

Configuring Clustered WebLogic JMS Resources

The WebLogic Server *Administration Console* provides an interface for easily enabling, configuring, and monitoring the features of the WebLogic Server, including JMS.

The following sections provide an overview of configuring WebLogic JMS 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. A cluster provides four key features above a single server:

- Scalability—servers can be added to the cluster dynamically to increase capacity.
- High Availability—redundancy of multiple servers insulates applications from failures. Redundancy of multiple destinations (queues and topics) as members of a single *distributed destination* set within a cluster ensures redistribution of the messaging load to other available members in the set when one member becomes unavailable.

- Migratability—respond to migration requests and bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure.
- Server affinity—when configured for the cluster, affinity load balancing algorithms (round-robin-affinity, weight-based-affinity, or random-affinity), provide server affinity for JMS client connections. If an JMS application has a connection to a given server instance, JMS will attempt to establish new JMS connections to the same server instance.

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

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

Obtain a Clustered JMS Licence

In order to implement JMS clustering, you must have a valid clustered JMS license, which allows a connection factory and a destination to be on different WebLogic Server instances. A clustered JMS license is also required to use the Foreign JMS Providers feature. If you do not have a valid clustered JMS license, contact your BEA sales representative.

How JMS Clustering Works

An administrator can establish cluster-wide, transparent access to JMS destinations from any server in the cluster, either by enabling 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. This way, each connection factory can be deployed on multiple WebLogic Servers. However, each user-defined connection factory must be uniquely named to be successfully deployed on multiple WebLogic Servers.

The administrator can also configure multiple JMS servers on the various nodes in the cluster—as long as the JMS servers are uniquely named—and can then assign JMS destinations to the various JMS servers. The application uses the Java Naming and Directory Interface (JNDI) to look up a connection factory and create a connection to establish communication with a JMS server. Each JMS server handles requests for a set of destinations. Requests for destinations not handled by a JMS server are forwarded to the appropriate WebLogic Server instance.

JMS Clustering Naming Requirements

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

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

JMS Distributed Destination within a Cluster

The WebLogic JMS administrator can also configure multiple destinations as part of a single distributed destination set within a cluster. Producers and consumers are able to send and receive to the distributed destination. In the event of a single server failure within the cluster, WebLogic JMS then distributes the load across all available physical destinations within the distributed destination set.

JMS as a Migratable Service within a Cluster

WebLogic JMS takes advantage of the migration framework implemented in the WebLogic Server core for clustered environments. This allows WebLogic JMS to properly respond to migration requests and bring a JMS server online and offline in an orderly fashion. This includes both scheduled migrations as well as migrations in response to a WebLogic Server failure. For more information, see "Configuring JMS Migratable Targets" on page 5-4.

Configuration Steps

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

1. Administer WebLogic clusters as described in "Configuring WebLogic Servers and Clusters" in *Using WebLogic Server Clusters*.
2. Identify *server targets* for JMS servers and for connection factories using the *Administration Console*:
   - For JMS servers, you can identify either a single-server target or a migratable target, which is a set of WebLogic Server instances in a cluster that can host an "exactly-once" service like JMS in case of a single server failure. For more information on migratable targets, see "Configuring JMS Migratable Targets" on page 5-4.

– For connection factories, you can identify either a single-server target or a cluster target, which are WebLogic Server instances that are associated with a connection factory to support clustering.

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

3. Optionally, you can configure your physical destinations as part of a single distributed destination set within a cluster.

What About Failover?

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

However, automatic failover is not currently supported by WebLogic JMS. For information about performing a manual failover, refer to "Recovering From a WebLogic Server Failure" in *Programming WebLogic JMS*.

Configuring JMS Migratable Targets

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

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

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

For more information about the migration of pinned services, see "Migration for Pinned Services" in *Using WebLogic Server Clusters*.

Configuration Steps for JMS Migration

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

1. Administer WebLogic clusters as described in "Configuring WebLogic Servers and Clusters" in the *Using WebLogic Server Clusters*.
2. Configure a candidate migratable target server for the cluster that can potentially host a JMS server.
3. Identify a migratable target server instance on which to deploy a JMS server.

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

   Note: A JMS server and all of its distributed destination members can migrate to another WebLogic Server within a cluster—even when the target WebLogic Server is already hosting a JMS server with all of its distributed destination members. Although this can lead to situations where the same WebLogic server hosts two physical destinations for a single distributed destination, this is permissible in the short term, since the WebLogic Server can host multiple physical destinations for that distributed destination. For more information about JMS distributed destinations, see "Using Distributed Destinations" in *Programming WebLogic JMS*.
4. For implementations that use persistent messaging, make sure that the persistent JMS store is configured such that all the candidate servers in a migratable target share access to the store. For more information about migrating persistent JMS stores, see "Persistent Store Migration" on page 5-5.
5. You can manually migrate services before performing server maintenance or to a healthy server if the host server fails. For more information, see "Migrating a Pinned Service To a Target Service Instance" in *Using WebLogic Server Clusters*.

Persistent Store Migration

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

- Implement a hardware solution, such as a dual-ported SCSI disk or Storage Area Network (SAN) to make your JMS persistent store available from other machines.

- Use JDBC to access your JMS JDBC store, which can be on yet another server. Applications can then take advantage of any high-availability or failover solutions offered by your database vendor.

Migration Failover

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

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 to a messaging resource such as a member of a distributed destination or a store-and-forward agent. It provides a generic way to pin messages to a member of a cluster hosting servlets, distributed queue members, or distributed Store and Forward nodes.

Configuring a Path Service

Configuring a Path Service for a route consists of:

- "Creating a Path Service Instance" on page 5-6
- "Modify an Existing Path Service Instance" on page 5-7

Creating a Path Service Instance

Use the following steps to configure a messaging bridge instance:

1. Click to expand the Services →Path Service node.
2. Click New.
3. Modify the configuration attributes for your bridge.
4. Click Next.
5. Select the servers and/or clusters on which you want to deploy your Path Service. You can only assign one Path Service to a cluster. Click Finish.

Modify an Existing Path Service Instance

The following sections provide information on how to modify the configuration of individual Path Service instance:

- "Changing the Persistent Store" on page 5-7
- "Changing the Target Cluster" on page 5-7

Changing the Persistent Store

Use the following steps to assign your Path Service instance to a different persistent store:

1. Click to expand the Services—Path Service node.
2. Select the Path Service instance to modify.
3. In the Configuration tab, click the Drop-down arrow to view a list of persistent stores.
4. Select the persistent stores for your Path Service instance.
5. Click Save.

Changing the Target Cluster

Use the following steps to assign your Path Service instance to a different cluster:

1. Click to expand the Services—Path Service node.
2. Select the Path Service instance to modify.
3. In the Targets tab, select the cluster for your Path Service instance.
4. Click Save.

JMS Distributed Destination Tasks

A *distributed* destination is a set of physical destinations (queues or topics) that are called under a single JNDI name so they appear to be a single, logical destination to a client, when the members of the set are actually distributed across multiple servers within a cluster, with each destination member belonging to a separate JMS server.

By enabling you to configure multiple physical queues and topics as members of a distributed destination, WebLogic JMS supports high availability and load balancing of the JMS destinations

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

Guidelines for Configuring Distributed Destinations

You configure distributed JMS destinations through the Services →JMS →Distributed Destinations node. To facilitate the configuration process, these instructions are divided into procedures that address the following scenarios:

- New implementations of WebLogic JMS with no physical destinations *or* existing configurations of WebLogic JMS that do not require previously configured destinations to be part of a distributed destination:
- Existing implementations of WebLogic JMS that require previously configured destinations to be members of a distributed destination set:

Note: The default Load Balancing Enabled and Server Affinity Enabled attributes for tuning a distributed destination configuration can be modified on the JMS connection factory through the *Administration Console*. For more programmatic information, see "Load Balancing Messages Across a Distributed Destination" in *Programming WebLogic JMS*. Form more configuration information, see "Configuring Message Load Balancing Across a Distributed Destination" on page 5-8 and "Configuring Server Affinity For a Distributed Destination" on page 5-10.

When a distributed topic or queue destination is created, a corresponding JMS template is automatically created with default attribute values for the distributed destination members. The new template will appear under the JMS Templates node with the same name as the distributed destination. The thresholds, quotas, and other attributes for the distributed destination members can be reset using this template.

Configuring Message Load Balancing Across a Distributed Destination

The Load Balancing Enabled attribute on the JMS Connection Factory →Configuration → General tab defines whether non-anonymous producers created through a connection factory are load balanced within a distributed destination on a per-call basis. 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 turn off the Load Balancing Enabled attribute.

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 configure load balancing on a connection factory:

1. Expand the JMS node.
2. Click the Connection Factories node. The JMS Connection Factories table displays all the connection factories defined in your domain.
3. Click the connection factory on which you want to establish message load balancing. A dialog displays in the right pane showing the tabs associated with modifying a connection factory.
4. Define the setting of the Load Balancing Enabled attribute on the General tab 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.
5. Click Apply to save your changes.

   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 Using the Server Affinity Enabled Attribute" in *Programming WebLogic JMS*.

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).

For more information about how message load balancing takes place among the members of a distributed destination, see "Load Balancing Messages Across a Distributed Destination" in *Programming WebLogic JMS*.

Configuring Server Affinity For a Distributed Destination

The Server Affinity Enabled attribute on the JMS Connection Factory →Configuration →General tab defines whether 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.

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. Expand the JMS node.
2. Click the Connection Factories node. The JMS Connection Factories table displays in the right pane showing all the connection factories defined in your domain.
3. Click the connection factory on which you want to disable server affinity. A dialog displays in the right pane showing the tabs associated with modifying a connection factory.
4. Define the Server Affinity Enabled attribute in the General tab.
   - 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.
5. Click Apply to save your changes.

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 Using the Server Affinity Enabled Attribute" in *Programming WebLogic JMS*.

CHAPTER 6

Tuning WebLogic JMS

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

Overview

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

Paging Out Messages To Free Up Memory

With the *message paging* feature, JMS servers automatically attempt to free up virtual memory during peak message load periods. This feature can greatly benefit applications with large message spaces.

JMS message paging saves memory for both persistent and non-persistent messages, as even persistent messages cache their data in memory. Paged persistent messages continue to be written to the regular backing store (file or database); and paged non-persistent messages are written to the JMS server's message paging store, which is configured separately.

A paged-out message does not free all of the memory that it consumes. The message header and message properties remain in memory for use with searching, sorting, and filtering.

Note: Messages sent in a transacted session are only eligible for paging *after* the session is committed. Prior to that, the message will only be held in memory; therefore, the heap size of the Java Virtual Machine (JVM) should be appropriately tuned to accommodate the projected peak amount of client load from all active sessions until they are committed. For more information on tuning your heap size, see "Tuning Java Virtual Machines (JVMs)" in *WebLogic Performance and Tuning*.

Specifying a Message Paging Directory

Message paging is enabled by default on JMS servers, and so a message paging directory is automatically created when paging is enabled without having to pre-configure one. You can, however, specify a directory using Paging Directory option, then paged-out messages are written to files in this directory.

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

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

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

Tuning the Message Buffer Size Option

The Message Buffer Size option specifies the amount of memory that will be used to store message bodies in memory before they are paged out to disk. The default value of Message Buffer Size is 64 megabytes, which is one-third of the maximum memory size supported by the Java Virtual Machine. The larger this parameter is set, the more memory JMS will consume when many messages are waiting on queues or topics. Once this threshold is crossed, JMS may write message bodies to the directory specified by the Paging Directory option in an effort to reduce memory usage below this threshold.

It is important to note that this parameter is not a quota. If the number of messages on the server passes the threshold, the server will write messages to disk and evict them from memory as fast as it can to reduce memory usage, but it will not stop accepting new messages. It is still possible to run out of memory if messages are arriving faster than they can be paged out. Users with high messaging loads who wish to support the highest possible availability should consider setting a quota, or setting a threshold and enabling flow control to reduce memory usage on the server.

Related Topics

For more information on tuning Weblogic JMS performance, refer to the following help topics:

- "Controlling the Flow of Messages on JMS Servers and Destinations" on page 6-3

Controlling the Flow of Messages on JMS Servers and Destinations

With the Flow Control feature, you can direct a JMS server or destination to slow down message producers when it determines that it is becoming overloaded.

The following sections describe how flow control feature works and how to configure flow control on a connection factory.

- "How Flow Control Works" on page 6-4
- "Configuring Flow Control" on page 6-4
- "Flow Control Thresholds" on page 6-6

How Flow Control Works

Specifically, when either a JMS server or it's destinations exceeds its specified byte or message threshold, it becomes *armed* and instructs producers to limit their message flow (messages per second).

Producers will limit their production rate based on a set of flow control attributes configured for producers via the JMS connection factory. Starting at a specified *flow maximum* number of messages, a producer evaluates whether the server/destination is still armed at prescribed intervals (for example, every 10 seconds for 60 seconds). If at each interval, the server/destination is still armed, then the producer continues to move its rate down to its prescribed *flow minimum* amount.

As producers slow themselves down, the threshold condition gradually corrects itself until the server/destination is *unarmed*. At this point, a producer is allowed to increase its production rate, but not necessarily to the maximum possible rate. In fact, its message flow continues to be controlled (even though the server/destination is no longer armed) until it reaches its prescribed *flow maximum*, at which point it is no longer flow controlled.

Configuring Flow Control

Producers receive a set of flow control attributes from their session, which receives the attributes from the connection, and which receives the attributes from the connection factory. These attributes allow the producer to adjust its message flow.

Specifically, the producer receives attributes that limit its flow within a minimum and maximum range. As conditions worsen, the producer moves toward the minimum; as conditions improve; the producer moves toward the maximum. Movement toward the minimum and maximum are defined by two additional attributes that specify the rate of movement toward the minimum and maximum. Also, the need for movement toward the minimum and maximum is evaluated at a configured interval.

Flow Control options are described in following table:

Table 6-1 Flow Control Attributes

Attribute	Description
Flow Control Enabled	Determines whether a producer can be flow controlled by the JMS server.
Flow Maximum	The maximum number of messages per second for a producer that is experiencing a threshold condition. If a producer is not currently limiting its flow when a threshold condition is reached, the initial flow limit for that producer is set to Flow Maximum. If a producer is already limiting its flow when a threshold condition is reached (the flow limit is less than Flow Maximum), then the producer will continue at its current flow limit until the next time the flow is evaluated. Once a threshold condition has subsided, the producer is not permitted to ignore its flow limit. If its flow limit is less than the Flow Maximum, then the producer must gradually increase its flow to the Flow Maximum each time the flow is evaluated. When the producer finally reaches the Flow Maximum, it can then ignore its flow limit and send without limiting its flow.
Flow Minimum	The minimum number of messages per second for a producer that is experiencing a threshold condition. This is the lower boundary of a producer's flow limit. That is, WebLogic JMS will not further slow down a producer whose message flow limit is at its Flow Minimum.

Table 6-1 Flow Control Attributes

Attribute	Description
Flow Interval	An adjustment period of time, defined in seconds, when a producer adjusts its flow from the Flow Maximum number of messages to the Flow Minimum amount, or vice versa.
Flow Steps	The number of steps used when a producer is adjusting its flow from the Flow Minimum amount of messages to the Flow Maximum amount, or vice versa. Specifically, the Flow Interval adjustment period is divided into the number of Flow Steps (for example, 60 seconds divided by 6 steps is 10 seconds per step). Also, the movement (that is, the rate of adjustment) is calculated by dividing the difference between the Flow Maximum and the Flow Minimum into steps. At each Flow Step, the flow is adjusted upward or downward, as necessary, based on the current conditions, as follows: • The downward movement (the decay) is geometric over the specified period of time (Flow Interval) and according to the specified number of Flow Steps. (For example, 100, 50, 25, 12.5). • The movement upward is linear. The difference is simply divided by the number of Flow Steps.

For detailed information about other connection factory attributes, and the valid and default values for them, see TBD.

Flow Control Thresholds

The attributes used for configuring bytes/messages thresholds are defined as part of the JMS server and/or its destination. Table 6-2 defines how the upper and lower thresholds start and stop flow control on a JMS server and/or JMS destination.

Table 6-2 Flow Control Threshold Attributes

Attribute	Description
Bytes/Messages Threshold High	When the number of bytes/messages exceeds this threshold, the JMS server/destination becomes armed and instructs producers to limit their message flow.
Bytes/Messages Threshold Low	When the number of bytes/messages falls below this threshold, the JMS server/destination becomes unarmed and instructs producers to begin increasing their message flow. Flow control is still in effect for producers that are below their message flow maximum. Producers can move their rate upward until they reach their flow maximum, at which point they are no longer flow controlled.

For detailed information about other JMS server and JMS destination attributes, and the valid and default values for them, see TBD.

-

CHAPTER 7

Monitoring WebLogic JMS

Comprehensive message administration improvements greatly enhance the ability of a WebLogic JMS administrator to view and manipulate the state of the messages in a running JMS Server, using either the Administration Console or through new public runtime APIs. These message management enhancements include, message browsing, message manipulation (such as move, delete, and produce), transaction management, message import/export, durable subscriber management, and connection management.

The following sections explain how to monitor the run-time statistics for your JMS objects from the Administration Console:

Monitoring WebLogic JMS

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

You can monitor statistics for the following JMS objects: JMS servers, connections, pooled connections, destinations, message producers, message consumers, JMS server session pools, and durable subscribers for JMS topics.

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

Note: For instructions on monitoring JMS connections to WebLogic Server, refer to TBD.

Troubleshooting WebLogic JMS

Not available for Beta.

Index

T