webMethods Broker Messaging Programmer's Guide

Version 10.15

October 2022

WEBMETHODS

This document applies to webMethods Broker 10.15 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: PIF-BROKER-MESSAGING-PROGRAMMERS-GUIDE-1015-20220912

Table of Contents

About this Guide

The *webMethods Broker Messaging Programmer's Guide* is designed for the developer who is responsible for developing and deploying Java Message Service (JMS) and C# .NET messaging applications.

This guide does the following:

- Explains webMethods messaging.
- Shows how to create, configure, and manage connection factories and destination objects.
- Describes the command-line tool (jmsadmin) that is used to work with administered objects.
- Provides commented samples of JMS and C# client code.
- Explains how to use JCA to connect your messaging client application that runs on the application server to webMethods Broker.

This guide assumes that you are familiar with the following:

- Terminology and basic operations of your operating system (OS). If you are not, please refer to the appropriate documentation for your OS.
- General knowledge of programming, the Java programming language, and the JMS API.
- Basic concepts of webMethods architecture and terminology.

Important:
If you have a lower fix level installed, some of the features described in this document might not be available to you. For a cumulative list of fixes and features, see the latest fix readme on the Empower website at https://empower.softwareag.com.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.

Convention	Description
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at https://documentation.softwareag.com.

In addition, you can also access the cloud product documentation via https://www.softwareag.cloud. Navigate to the desired product and then, depending on your solution, go to "Developer Center", "User Center" or "Documentation".

Product Training

You can find helpful product training material on our Learning Portal at https://knowledge.softwareag.com.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at https://techcommunity.softwareag.com. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at https://github.com/softwareag and https://hub.docker.com/publishers/softwareag and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at https://empower.softwareag.com. Many services on this portal require that you have an account.

If you do not yet have one, you can request it at https://empower.softwareag.com/register. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

I Introduction

1 Introduction to webMethods Broker Messaging

Overview

This chapter provides overview information about webMethods Broker, including how the webMethods Broker API for JMS and the webMethods Broker Client C# API work with webMethods Broker. The chapter also describes key features supported by these messaging APIs, such as a common object model, shared administrative tools, Java Connector Architecture (JCA), and the Java Transaction API.

More detailed information about webMethods Broker API for JMS and Broker Client C# API features is described elsewhere in this programming guide; use the available links to jump directly to that information.

webMethods Broker

You can configure webMethods Broker to work with standard Java JMS clients using the webMethods Broker API for JMS, and with Windows .NET clients using the webMethods Broker Client C# API. Both of these messaging platforms support the publish/subscribe and point-to-point messaging styles (for details, see "Messaging Styles" on page 24).

JMS Clients

webMethods Broker used as a JMS provider supports JMS interfaces. It provides a set of Java classes that enable JMS client applications to send and receive messages using the message brokering technology for transport and delivery.

webMethods Broker used as a JMS provider functions as a standalone messaging system. It also supports several of the most commonly used application servers, such as the IBM WebSphere, Oracle WebLogic, and JBoss application servers, by means of a standardized adapter technology, the Java EE Connector Architecture (JCA) protocol.

webMethods Broker used as a JMS provider can send messages to and consume messages from a variety of JMS client sources such as:

- Plain old Java objects (POJOs)
- EntityBeans, SessionBeans, and MessageDrivenBeans
- Java clients built-in Integrated Development Environments (IDEs)

C# Clients

The webMethods Broker Client C# API follows the same type of messaging object model and object hierarchy used by the webMethods Broker API for JMS (for example, the use of connection factories, topics, and queues). However, the clients are written in C# and compiled into a pure .NET assembly that interfaces with webMethods Broker.

Note:

C# clients must be written and compiled for Microsoft .NET Framework version 3.5 or above. For using C# clients that are built for versions of Microsoft .NET Framework above 3.5, refer the Microsoft documentation.

Client Interoperability

There is a high degree of interoperability among JMS and C# messaging clients. To the Broker, clients written for either JMS or C# are indistinguishable, since both utilize the same wire format. The webMethods Broker Client C# API is essentially a C# implementation of the JMS API. You can use the same administrative tools to work with both.

Several differences exist between the implementations:

- The webMethods Broker Client C# API is designed to work solely on a Windows-based .NET platform, therefore, it does not support Java EE applications or distributed (XA) transactions.
- Clients written for the webMethods Broker Client C# API must save their administered objects to an LDAP directory. Clients written for the webMethods Broker Client API for JMS can use any of several supported JNDI providers to store administered objects, including a JNDI LDAP provider.

The webMethods Broker APIs for JMS and webMethods Broker C# messaging APIs are *not* interoperable with the webMethods Broker Java APIs and webMethods Broker C APIs. Interoperability does not exist because the two groups of APIs do not use the same packing rules to access a Broker event.

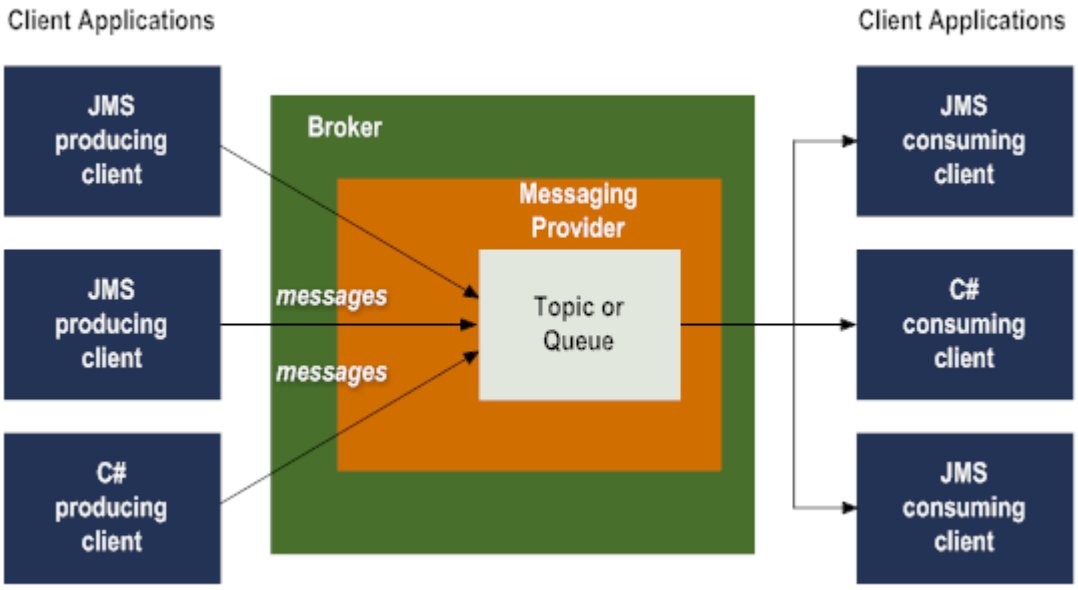
However, the webMethods Broker API for JMS can interoperate with webMethods Broker Java APIs and webMethods C APIs through a marshalling mechanism (see "JMS Marshalling" on page 223). This mechanism allows users to hook in conversion routines to convert data between the JMS and legacy webMethods Broker client formats. No equivalent mechanism currently exists for C# clients.

webMethods Broker Messaging Technology

webMethods Broker messaging technology supplies the underlying routing and distribution facility for transporting messages among JMS and C# clients. Messaging clients connect to each other through one or more Brokers.

The Broker is the messaging backbone of webMethods Broker messaging. It provides the infrastructure for implementing asynchronous, message-based solutions that are built on a publish-subscribe or point-to-point model.

The role of the Broker is to route documents between messaging clients that are information producers and messaging clients that are information consumers.



The Broker maintains a registry of document types that it recognizes. It also maintains a list of subscribers that are interested in receiving those types of documents. When the Broker receives a published document, it queues it for the subscribers of that document type. Subscribers retrieve documents from their queues. This action usually triggers an activity on the subscriber's system that processes the document.

For additional information about configuring and managing webMethods Broker messaging technology, see *Administering webMethods Broker*.

Highlights

The following are features supported by webMethods Broker messaging.

Shared State

Both the webMethods Broker API for JMS and the webMethods Broker Client C# API allow clients in different applications to share the same client state. Sharing client state allows several clients possibly executing on different hosts to handle events in a parallel, first-come, first-serve, basis. Sharing state enables your messaging solution to incorporate parallel processing and load balancing.

For more information about shared state Broker clients, see the *webMethods Broker Client Java API Programmer's Guide*.

Basic Authentication Support

webMethods Broker supports the basic authentication protocol for exchanging information securely across a network.

You can use basic authentication for messaging clients and webMethods Broker in conjunction with Broker Access Control Lists (ACLs).

For more information, see "Configuring Messaging Clients for Basic Authentication" on page 153 and *Administering webMethods Broker*.

SSL Support

webMethods Broker fully supports the Secure Sockets Layer (SSL) protocol for exchanging information securely across a network. You can use SSL as part of your messaging solution for communications between JMS and C# messaging clients and webMethods Broker.

You can use SSL authentication for messaging clients in conjunction with Broker Access Control Lists (ACLs). With ACLs, you control access to Broker Servers, Brokers on those Broker Servers, client groups, territories, territory gateways, clusters, and cluster gateways. To gain access, a messaging client must have an identity (distinguished name and authenticator's distinguished name) that matches the ACLs.

For more information, see "Configuring webMethods Messaging Clients for SSL" on page 147 and *Administering webMethods Broker*.

Java EE Connector Architecture (JCA) Support (JMS Clients Only)

For JMS clients, webMethods Broker used as a JMS provider supports application servers as described in the Java Message Service standard. It provides this support through JCA, a standardized adapter for connecting enterprise applications to JMS providers.

Currently, webMethods Broker used as a JMS provider supports JCA for the following application servers:

- JBoss Application Server
- IBM WebSphere Application Server

> **Note:**webMethods Broker provides support for Oracle WebLogic Application Server through JMS Application Server Facilities (ASF).

For information on the versions of the application servers that are supported by webMethods Broker, see the *System Requirements for Software AG Products*.

Java Transaction API Support (JMS Clients Only)

webMethods Broker used as a JMS provider is a Java Transaction API (JTA) capable provider. webMethods Broker used as a JMS provider supports the JTA XAResource API as described in the Java Message Service (JMS) standard. JTA support enables webMethods Broker to participate in distributed transactions. For more information about JTA support for a webMethods Broker client-side clustering environment, see "JTA /XA Support in a JMS Clustered Environment" on page 279.

webMethods Broker used as a JMS provider is not a distributed transaction manager but relies on the transaction managers that the supported application servers provide. webMethods Broker

supports distributed transactions on supported application servers, although it does not support distributed transactions for standalone JMS messaging solutions.

Note:webMethods Broker supports the *resource side* of the ASF and JTA protocols as described in the Java Message Service standard (specifically, it supports the `javax.transaction.xa.XAResource`, `javax.jms.XAConnection`, and `javax.jms.XASession` interfaces).

webMethods Broker used as a JMS provider supports the transactional connection factories XAConnectionFactory, XAQueueConnectionFactory and XATopicConnectionFactory to create connections to administered objects that will be managed by the application server's distributed transaction coordinator in distributed transactions. Messages created from these objects can be grouped into distributed transactions with other resources.

Local or one-phase-commit transactions can be handled in standalone messaging applications within the context of a single session. These transactions are not based on XA resources nor are they handled at the administrative level. For more information, see "Implementing a Local Transaction" on page 206.

Note:
JCA and JTA support is only available for Java-JMS messaging clients that connect to webMethods Broker that is used as a JMS provider.

Managing Administered Objects

Administered objects occupy a key role in message management; their creation, configuration, and storage form the basis for communication between the messaging APIs and their clients.

You can manage the administered objects of JMS and C# messaging clients through the My webMethods user interface or by using a messaging command-line tool; however, not all options available with the command-line tool are available in My webMethods. You can also use manage administered objects programmatically by writing directly to the APIs. For more information, see "Configuring Administered Objects" on page 37.

Saving Administered Objects

The following summarizes how administered objects are stored by the two webMethods Broker messaging providers:

- **JMS.** Like all JMS providers, webMethods Broker used as a JMS provider stores administered objects in a standardized namespace using a Java Naming and Directory Interface (JNDI) provider. Clients written for webMethods Broker used as a JMS provider can use one of several supported JNDI providers.

  Take a backup of the Broker storage to avoid any data loss. For more information, see *Administering webMethods Broker*.

  You can use webMethods Broker as a JNDI provider in the production environment for storing only the Broker administered objects such as connection factories, queues, and topics. For more

information, see " webMethods Naming Service for JMS: Configuration Settings and Properties" on page 211.

- **C#.** Clients written for the webMethods Broker Client C# API that save their administered objects must bind them to an LDAP directory through JNDI. As such, only the Oracle JNDI LDAP Service Provider supports webMethods Broker C# clients.

Developing Applications with webMethods Messaging

Following are the high-level steps for developing client applications with webMethods Broker:

1. Install and configure the components required to establish your messaging environment. For instructions, see "Configuring Environment Variables" on page 19 and *Installing and Upgrading webMethods Broker*.

2. Configure administered objects for your client application, using either the JMSAdmin command-line tool (see "The JMSAdmin Command-Line Tool" on page 45) or the My webMethods user interface (see *Administering webMethods Broker*).

3. Design and implement the client programs that make up your messaging application. For information and basic examples, see "A Basic JMS Sender-Receiver Client" on page 173, "JMS Request-Reply Application with a Message Selector" on page 181, "C# Application Configuration File" on page 255 and the online API programming documentation.

4. Execute your client programs.

Configuring Environment Variables

For ease of use, you can update several of the environment variables to point to the libraries that the messaging client will use at run time.

Setting the CLASSPATH Variable (JMS Clients)

Add the following files to the CLASSPATH on the client machine (for UNIX, replace the \ directory path indicator with /, except where indicated).

- For webMethods Broker client libraries:

  Software AG_directory \common\lib\wm-jmsclient.jar

- For the JMS library:

  Software AG_directory \common\lib\glassfish\gf.javax.jms.jar

- For the client libraries for your JNDI provider:
  - For the webMethods JMS Naming Service, both:
    - *Software AG_directory* \common\lib\wm-jmsnaming.jar

- *Software AG_directory* \common\lib\wm-brokerclient.jar
- *Software AG_directory* \common\lib\wm-g11nutils.jar

- For other JNDI providers:

  The JAR file for your JNDI provider (see your JNDI provider documentation).

Setting the Windows PATH (C# Clients)

The C# API msg.dll is installed into *webMethods Broker_directory* \bin. Your Windows PATH will point to that location.

Properties Specific to webMethods Broker Used as a JMS Provider

webMethods Broker when used as a JMS provider, supports properties that are not part of the Java Message Service (JMS) standard. In some cases, when you use webMethods Broker as a JMS provider, you may need to set the values of specific properties to values other than the defaults. These properties include internal values such as the timeout value for a connection to the Broker.

When you use webMethods Broker as a JNDI provider, if encryption is not required, set the `com.webmethods.jms.naming.encrypted` property to `false`. The default value of `com.webmethods.jms.naming.encrypted` is `true`.

To specify properties when you use webMethods Broker as a JMS provider

1. Create a Java properties file named `wmjms.properties`.
2. Add the properties that you want to change to that file.
3. Add the directory containing the `wmjms.properties` file to CLASSPATH.
4. when you use webMethods Broker as a JMS provider, the properties are defined in the Javadoc of the `WmJMSConfig` class. Use that information before reconfiguring the property values in your code.

Configuring the C# Application

You may need to configure your C# client so that it is able to locate the LDAP server that will store the administered objects, or for other purposes, such as customizing logging for the application.

You can use the application configuration file (app.config) supplied with your .NET project to customize these kinds of parameters. For example, to specify the LDAP server, you include its URL as a parameter value in the configuration file.

For more information, see "C# Application Configuration File" on page 255 and the C# messaging API documentation, available on the Software AG Documentation website at http://documentation.softwareag.com.

2 Basics of webMethods Messaging

Overview

This chapter provides an overview of concepts common to the webMethods Broker API for JMS client and the webMethods Broker Client C# API. This chapter is for developers who are new to messaging architectures and for those wanting a review of JMS concepts and the JMS API. Experienced messaging developers and those looking for specific information about the webMethods Broker APIs can skip this chapter.

An in-depth treatment of messaging architecture is beyond the scope of this programming guide.

webMethods Messaging

webMethods Broker APIs allow applications to communicate with each other using a common set of interfaces. Although you can write applications in both Java/JMS and C# .NET in webMethods Broker, the programming or object model is virtually the same for both.

- The webMethods Broker API for JMS supports the Java Message Service standard and includes a few additional webMethods Broker-specific enhancements.
- The webMethods Broker Client C# API is a Windows .NET API with an object model patterned after JMS.

JMS Messaging

The Java Message Service (JMS) is a Java API that allows applications to communicate with each other using a common set of interfaces. The JMS API provides messaging interfaces but not the implementations.

A *JMS provider*, such as the webMethods Broker used as a JMS provider, is a messaging system that supports the JMS message interfaces and provides administrative and control features. It supports routing and delivery of messages by means of the JMS API on the client. A goal of JMS is to maximize the portability of JMS applications across different JMS providers.

JMS clients are the programs or components, written in Java, that produce and consume messages.

C# Messaging

The C# messaging API allows Windows messaging clients written as Microsoft .NET assemblies to communicate with the Broker as if they were JMS clients. Because the C# messaging API is based on the JMS programming model, and because the wire format is the same for both clients, .NET developers only need concern themselves with the client code and not the implementation details when connecting to Broker.

Messaging Styles

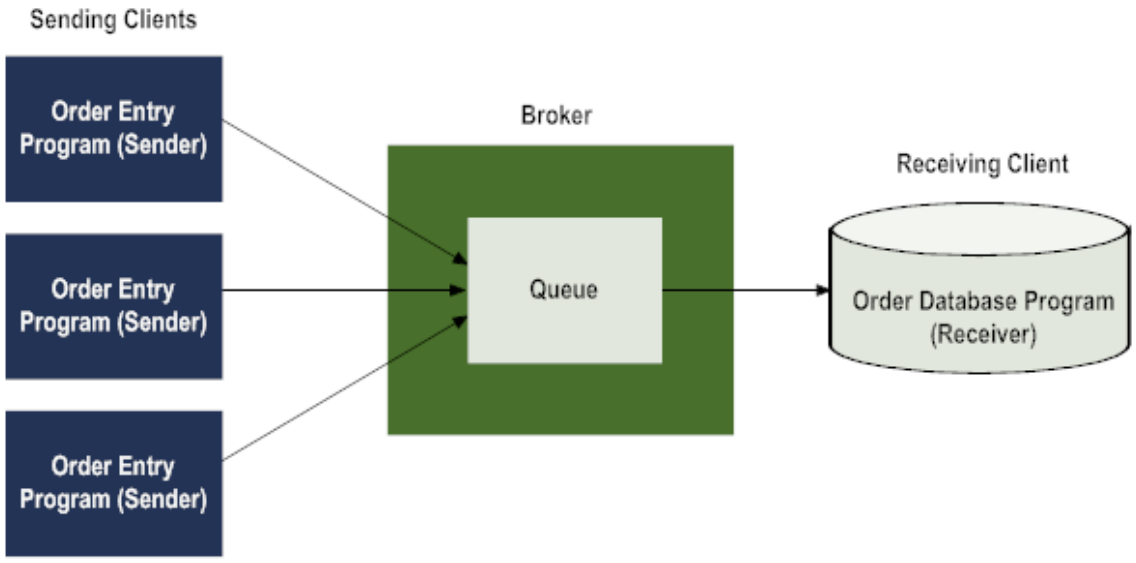
A messaging style refers to how messages are produced and consumed. webMethods Broker supports the publish-subscribe (pub-sub) and point-to-point (PTP) messaging styles.

Point-to-Point (PTP) Messaging

In point-to-point (PTP) messaging, message producers and consumers are known as *senders* and *receivers*.

The central concept in PTP messaging is a destination called a *queue*. A queue represents a single receiver. Message senders submit messages to a receiver's queue and wait for the receiver to respond to the message.

In the PTP model, a queue may receive messages from many different senders and may deliver messages to multiple receivers; however, each message is delivered to only one receiver.

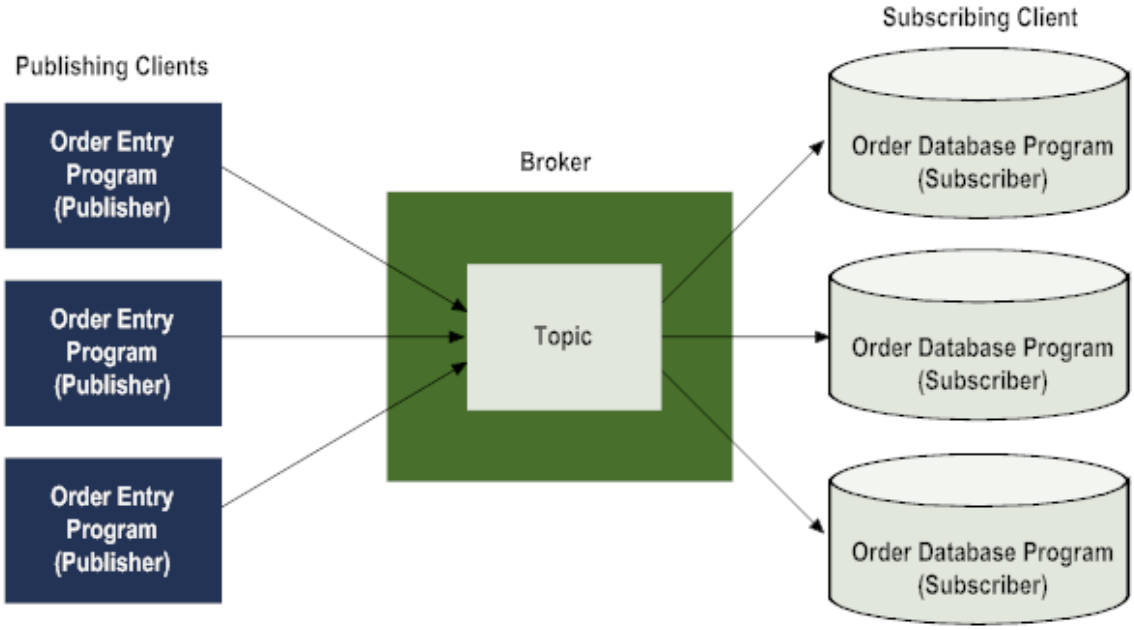


Publish-Subscribe Messaging

In publish-subscribe messaging, message producers and consumers are known as *publishers* and *subscribers*.

The central concept in the publish-subscribe messaging is a destination called a *topic*. Message publishers send messages of specified topics. Clients that want to receive that type of message subscribe to the topic.

The publishers and subscribers never communicate with each other directly. Instead, they communicate by exchanging messages through a message Broker.



Publishers and subscribers have a timing dependency. Clients that subscribe to a topic can consume only messages published after the client has created a subscription. In addition, the subscriber must continue to be active to consume messages.

The messaging APIs relax this dependency by making a distinction between *durable subscriptions* and *non-durable subscriptions*.

Durable Subscriptions

Durable subscriptions allow subscribers to receive all the messages published on a topic, including those published while the subscriber is inactive. When the subscribing applications are not running, the messaging provider holds the messages in nonvolatile storage. It retains the messages until either of the following situations occurs:

- The subscribing application becomes active, identifies itself to the provider, and sends an acknowledgment of receipt of the message.
- The expiration time for the messages is reached.

Non-durable Subscriptions

Non-durable subscriptions allow subscribers to receive messages on their chosen topic, only if the messages are published while the subscriber is active. You generally use this type of subscription for any kind of data that is time sensitive, such as financial information.

JMS API Programming Model

The following section summarizes the most important components of the JMS API. Since the same abstractions are used for webMethods Broker APIs for JMS and C# APIs, the descriptions that follow are valid for both types of clients, except where noted.

The building blocks of a JMS application consist of the following:

- Administered objects (connection factories and destinations)
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

Administered Objects

Administered objects are preconfigured objects that an administrator creates for use with C# and JMS client programs. Administered objects serve as the bridge between the client code and the .NET assembly or JMS provider.

By design, the messaging APIs separate the task of configuring administered objects from the client code. This architecture maximizes portability: the provider-specific work is delegated to the administrator rather than to the client code. However, the implementation must supply its own set of administrative tools (for example, the My webMethods user interface and the jmsadmin command-line utility) to configure the administered objects.

JMS administered objects are stored in a standardized namespace called the Java Naming and Directory Interface (JNDI). JNDI is a Java API that provides naming and directory functionality to Java applications. JNDI provides a way to store and retrieve objects by a user supplied name.

Types of Administered Object

There are two types of administered objects: *connection factories* and *destinations*.

Connection Factories

A *connection factory* is the object a client uses to create a connection with a JMS provider. It encapsulates the set of configuration parameters that a JMS administrator defines for a connection.

The type of connection factory determines whether a connection is made to a topic (in a publish-subscribe application), a queue (in a point-to-point application), or can be made to both (generic connection), and whether messages are managed like elements in a distributed transaction in the client application.

You use XA-based connection factories in JMS applications managed by an application server, in the context of a distributed transaction. You do *not* use XA-based connection factories in:

- Standalone JMS messaging applications
- webMethods C# messaging applications

Following are the types of connection factories:

- *ConnectionFactory*. The preferred connection factory object. This is a generic connection factory object, which can be used as either a `QueueConnectionFactory` or `TopicConnectionFactory` or both at run time.

  Note:
  The webMethods C# API only supports use of a generic *ConnectionFactory*. All other types of connection factory can only be used in JMS clients.

- *QueueConnectionFactory*. Used to make connections in a point-to-point application.
- *TopicConnectionFactory*. Used to make connections in a publish-subscribe application.
- *XAConnectionFactory*. A generic connection factory object, where the object can be used in a transactional context.
- *XAQueueConnectionFactory*. Used to make connections in a point-to-point application, where the object will be used in a transactional context.
- *XATopicConnectionFactory*. Used to make connections in a publish-subscribe application, where the object will be used in a transactional context.

Destinations

Destinations are the objects that a client uses to specify the target of messages it produces and the source of messages it consumes. These objects specify the identity of a destination to a JMS API method. There are four types of destinations; only the first two (queues and topics) are administered objects:

- *Queue*. A queue object covers a provider-specific queue name and is how a client specifies the identity of a queue to JMS methods.
- *Topic*. A topic object covers a provider-specific topic name and is how a client specifies the identity of a topic to JMS methods.
- *TemporaryQueue*. A queue object created for the duration of a particular connection (or `QueueConnection`). It can only be consumed by the connection from which it was created.
- *TemporaryTopic*. A topic object that is created for the duration of a particular connection (or `TopicConnection`). It can only be consumed by the connection from which it was created.

Connections

A *connection* object is an active connection from a client to its JMS provider. In JMS, connections support concurrent use. A connection serves the following purposes:

- A connection encapsulates an open connection with a JMS provider. It typically represents an open TCP/IP socket between a client and the service provider software.
- The creation of a connection object is the point where client authentication takes place.
- A connection object can specify a unique client identifier.
- A connection object supports a user-supplied `ExceptionListener` object.

A connection should always be closed once its use is no longer required.

Sessions

A *session* object is a single-threaded context for producing and consuming messages. If a client uses different threads for different paths of message execution, a session must be created for each of the threads.

A session is used to create message producers, message consumers, temporary topics, and temporary queues; it also supplies provider-optimized message factories.

In JMS, a session provides the context for grouping a set of send and receive messages into a transactional unit.

Message Producer

A *message producer* is an object that a session creates to send messages to a destination (a topic or a queue).

Message Consumer

A *message consumer* is an object that a session creates to receive messages sent to a destination. A message consumer allows a client to register interest in a destination, which manages the delivery of messages to the registered consumers of that destination.

Message Listener

A *message listener* object is an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains the single method `onMessage()`. The implementation of `onMessage()` is a user-defined set of actions taken when a message arrives at its destination.

Message Selector

A client may want to receive subsets of messages. A *message selector* allows a client to filter the messages it wants to receive by use of a SQL92 string expression in the message header. That expression is applied to properties in the message header (not to the message body content) containing the value to be filtered.

If the SQL expression evaluates to true, the message is sent to the client; if the SQL expression evaluates to false, it does not send the message.

Messages

Messages are objects that communicate information between client applications. Following are descriptions of several key concepts related to JMS and C# messages.

Message Structure

Messages are composed of the following parts:

- **Header.** All messages support the same set of header fields. Header fields contain predefined values that allow clients and providers to identify and route messages. Each of the fields supports its own `set` and `get` methods for managing data; some fields are set automatically by the `send` and `publish` methods, whereas others must be set by the client.

  Examples of header fields include:

  - `JMSDestination`, which holds a `destination` object representing the destination to which the message is to be sent.
  - `JMSMessageID`, which holds a unique message identifier value and is set automatically.
  - `JMSCorrelationID`, which is used to link a reply message with its requesting message. This value is set by the client application.
  - `JMSReplyTo`, which is set by the client and takes as a value a `Destination` object representing where the reply is being sent. If no reply is being sent, this field is set to null.

- **Properties (optional).** Properties are used to add optional fields to the message header. There are several types of message property fields.
  - *Application-specific properties* are typically used to hold message selector values. Message selectors are used to filter and route messages.
  - *Standard properties*. The API provides some predefined property names that a provider may support. Support for the `JMSXGroupID` and `JMSXGroupSeq` is required; however, support for all other standard properties is optional.
  - *Provider-specific properties* are unique to the messaging provider and typically refer to internal values.
- **Body (optional).** JMS defines various types of message body formats that are compatible with most messaging styles. Each form is defined by a message interface:
  - `StreamMessage`. A message whose body contains a stream of Java primitive values. It is filled and read sequentially.
  - `MapMessage`. A message whose body contains a set of name-value pairs where names are Strings and values are Java primitive types. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
  - `TextMessage`. A message whose body contains a `java.lang.String`.
  - `ObjectMessage`. A message that contains a Serializable Java object.
  - `BytesMessage`. A message that contains a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format. In many cases, it will be possible to use one of the other, self-defining, message types instead.

    Both `StreamMessage` and `MapMessage` support the same set of primitive data types. Conversions from one data type to another are possible.

Message Delivery

A client can receive messages synchronously or asynchronously.

For synchronous receipt, a client can request the next message from a message consumer using one of its `receive` methods. Several variations of `receive` allow a client to poll or wait for the next message. `receive()` is a blocking method that blocks until the message arrives or times out.

For asynchronous receipt, a client can register a `MessageListener` object with a message consumer. As messages arrive at the message consumer, the client delivers them by calling the message listener's `onMessage()` method. The `onMessage()` method proceeds based on the contents of the message. You register the message listener with a specific message consumer using the `setMessageListener()` method.

Message Acknowledgment

A message is not considered to be successfully consumed until it is acknowledged. Depending on the session acknowledgment mode, the messaging provider may send a message more than once to the same destination. There are several message acknowledgment constants:

Value	Description
`AUTO_ACKNOWLEDGE`	Automatically acknowledges the successful receipt of a message. Message acknowledgements are sent asynchronously to Broker Server. Configure these properties to change the default behavior of message acknowledgements: ■ autoAckBatchSize ■ autoAckBatchSizeRetry ■ autoAckServicePollInterval For information about the APIs available for configuring the asynchronous auto acknowledgements, see the descriptions of the WmJMSConfig class in the Javadoc.
`CLIENT_ACKNOWLEDGE`	Acknowledges the receipt of a message when the client calls the message's `acknowledge()` method.
`DUPS_OK_ACKNOWLEDGE`	Instructs the session to automatically, lazily acknowledge the receipt of messages. This reduces system overhead but may result in duplicate messages being sent.
`SESSION_TRANSACTED`	Used for transacted sessions. The acknowledgment mode is ignored.

II webMethods Messaging Administration

3 Configuring Administered Objects

Overview

This chapter provides an overview of managing webMethods C# and JMS administered objects. It describes different methods of configuring administered objects and recommendations for when to use each.

The chapter also explains how to configure Broker client group permissions, which allow C# and JMS clients to send and receive messages.

Administrator Tools

The primary tasks for administrative tools include:

- Creating and managing connection factories and destinations
- Storing administered objects in a JNDI namespace
- Assigning the Broker permissions for destinations to send and receive messages

Working with Administered Objects

Because the tools used to manage administered objects are implementation dependent, the webMethods product suite supplies its own tools for its JMS and C# administered objects. These tools are described below.

Tool	Description
JMSAdmin command-line tool	This tool is an interpreter that uses command syntax to manage administered objects for both JMS and C# clients. This administrative tool is run from the command line; administrative commands can be executed one at a time or stored in a text file and executed in batch mode. For more information, see "The JMSAdmin Command-Line Tool" on page 45, which explains how to use the tool, and "JMSAdmin Command Reference" on page 57, which lists the JMSAdmin commands, their parameters, and their syntax and provides examples.
My webMethods user interface	This user interface includes selections for creating, viewing, modifying, and assigning attributes and permissions to JMS and C# administered objects, information about their JNDI providers, and the Broker addresses associated with the objects. For information about JNDI naming directories and using My webMethods to configure JMS and C# administered objects, see *Administering webMethods Broker*. The Broker part of the My webMethods user interface is also referred to as the *Broker admin component*.

Tool	Description
JMS factory API (JMS only)	This API allows you to create connection factories and destinations for JMS without using JNDI. The webMethods Java package `com.webmethods.jms` contains the JMS factory class `WmJMSFactory`. If you use this class, you can create the connection factories and destinations at run time (in the JMS client code) rather than administratively. However, the code used to create the administrative objects is not portable and cannot be used with other JMS providers.
Broker Client C# API (C# only)	The `MsgFactory` API allows you to create connection factories and destinations for C# clients.

Note:
For more information about the webMethods API for JMS, see *webMethods Broker API for JMS Reference*. For more information about the C# API, see *webMethods Broker C# Client API Reference*. Both references are available on the Software AG Documentation website at http://documentation.softwareag.com. It is not included in this programming guide.

Choosing an Administrative Tool

Although all of the webMethods administrator tools are, for the most part, able to perform the same functions, each has its own strengths. Your choice of how to configure JMS and C# administered objects for your messaging client depends on several factors.

Standard webMethods Tools for Administered Objects

- If you prefer working in the command line or you want to create batch files to automatically execute sets of commands, use the JMSAdmin command-line tool.

  For example, you can write a batch file of JMS or C# administrative commands that automatically creates numerous connection factories and destinations. If you were to use My webMethods instead, you would have to manually create one object at a time via the user interface, which is more time consuming.

- If you prefer using a graphical user-interface, use My webMethods.

  With My webMethods, you can manage JMS and C# administered objects using standard controls on a web page. This tool provides the best views and the easiest methods for creating, modifying, and assigning permissions to JMS administered objects. In addition, you can display lists of messaging objects and their properties, which you can sort and filter. Many other webMethods applications are also accessible through My webMethods.

Using the webMethods Broker APIs for JMS and C# APIs

If you prefer creating administrative objects directly in your JMS or C# client code, write to the APIs for JMS or C#. For more information, see the *webMethods Broker API for JMS Reference*, available on the Software AG Documentation website at http://documentation.softwareag.com.

Broker Administration API

The webMethods Java package `com.webmethods.jms.admin` contains the JMS administration class `WmJMSAdmin`, which you can use to administer the Broker for JMS-related tasks.

Normally, you will not need to use this class, as it is simpler to administer Broker using the JMSAdmin command-line tool, which is built from `WmJMSAdmin`, or by using My webMethods. It is also possible to use `WmJMSAdmin` to build your own administrative tools, if desired.

Client Groups and Permissions

A *client group* is a named collection of property settings and ACL permissions that constitute the group's membership characteristics. Broker clients that join a client group inherit its property settings and permissions. For a Broker client, membership in a client group is *mandatory*; every Broker client must belong to exactly one client group.

JMS Client Groups

webMethods clients for JMS and C# are also Broker clients. For a JMS or C# client to exchange information with other clients, it needs to be assigned to a client group. For JMS or C# clients, the default client group configured by webMethods tools is named *JMSClient* (you can and should use your own application-specific client groups).

For JMS and C# clients, client group membership is necessary to assign permissions that specify whether clients can publish and subscribe to specific documents (topics and queues).

See *Administering webMethods Broker* for a detailed description of Broker client groups.

Setting Client Group Access Permissions

You use an administrative tool to assign access permissions to a client group.

Note:
Administrator access to a Broker Server is required to use an administrative tool, such as JMSAdmin, with that Broker Server. Administrator access is also required in order to connect to the Admin client group in the Broker that is being administered.

Typically, in the client code, after binding the administered objects to a JNDI namespace and creating the administered objects, you create the JMS and C# client groups and then assign the groups' publish and subscribe permissions for the destinations.

In the following example, JMSAdmin is used to create the client groups `SampleSenders` and `SampleReceivers`, and publish and subscribe permissions are assigned for the topic `Samples::RequestInfo`:

```
create group SampleSenders
permit group SampleSenders to publish Samples::RequestInfo
create group SampleReceivers
permit group SampleReceivers to subscribe Samples::RequestInfo
```

In the example, the JMSAdmin `permit` command adds the topic `Samples::RequestInfo` to the client group's can-publish and can-subscribe lists. These lists are properties of the Broker client group that stores information about which documents are authorized to be published or received.

You can also use My webMethods to create client groups and set permissions for destinations. See *Administering webMethods Broker* for information. This manual also contains a detailed description of the client group can-publish and can-subscribe properties.

For more information about configuring client groups and setting group permissions using JMSAdmin, see "Create Group" on page 96, "Delete Group" on page 101, "Deny" on page 103, and "Permit" on page 141.

Using JNDI Providers (Naming Directories)

webMethods Broker allows you to select from several JNDI providers when configuring a messaging client. The selections that are available depend on whether you are configuring a JMS or C# client, and if you are configuring a JMS client, whether it is a standalone application or an application server.

- If you are configuring administered objects for a C# client, you must use the Oracle LDAP Provider. You must also have an operational LDAP server.
- If you are configuring administered objects for a JMS client to be used as a standalone messaging application, your options are as follows:
  - webMethods Naming Service for JMS

    This JNDI provider is supplied for storing only the Broker administered objects such as connection factories, queues, and topics. For more information, see " webMethods Naming Service for JMS: Configuration Settings and Properties" on page 211.
  - Oracle LDAP Provider

    If you use this JNDI provider, you must also have an operational LDAP server.
- If you are configuring a JMS client to be used in an application server, you can choose from these JNDI providers:
  - WebLogic JNDI Provider
  - WebSphere JNDI Provider

Configuring Time-out for JMS Client Connections

In the previous webMethods Broker versions, it was difficult to identify whether a JMS client connection timed-out due to non-availability of Broker or because of a busy socket connection. The JMS client connections would time-out when the time assigned to the `com.webmethods.jms.brokerTimeout`parameter elapsed.

The three new JMS properties improve the management of JMS client connection time-out. Set the JMS client time-out properties mentioned in the table below in one of the following ways:

- Assign the properties to the `watt.config.systemProperties` property in the **Settings > Extended** page in Integration Server Administrator.
- Set the properties in the `wmjms.properties` file.

Property	Description
`com.webmethods.jms.brokerTimeout`	Specifies the maximum time in seconds a JMS client connection will wait for a reply from Broker, irrespective of the socket being busy or idle. The default value is 60 seconds. Set this property to a high value considering the time required for a huge file transfer. For example, assuming that one of the IS triggers using the connection alias takes about 50 minutes to transfer a large file, set `com.webmethods.jms.brokerTimeout=3600` (1 hour)
`com.webmethods.jms.brokerIdleSocketTimeout`	Specifies the maximum time in seconds a JMS client waits on an idle socket before timing out. The JMS client senses the connection issue when it receives an idle socket time-out exception. The default value is 60 seconds. Set this property to a small value so that an idle socket time-out can be discovered at the earliest. For example, set `com.webmethods.jms.brokerIdleSocketTimeout=30`
`com.webmethods.jms.brokerTimeoutPostConnectionClose`	Specifies the maximum time in seconds a JMS client will wait for any operation to be complete on a connection marked for closure. The default value is 30 seconds. For example, if `com.webmethods.jms.brokerTimeoutPostConnectionClose=30`, the cluster connection waits for 30 seconds to complete any file transfer operation in progress before closing the cluster connection.

Configuring Keep-alive

Configuring the keep-alive properties helps you identify whether the connection failure is at the Broker Server end or at the JMS client end.

Configuring Keep-alive for Broker Server

Property	Description
`com.webmethods.jms.broker.keepAliveInterval`	Specifies the time interval in seconds for sending the keep-alive messages from Broker Server to the JMS client. Sending keep-alive messages in regular intervals helps in identifying dead connections. This feature is very useful when you configure your messaging system in a wide area network. Set the value to `0` (zero) to disable the keep-alive feature. The default value is `60`.
`com.webmethods.jms.broker.keepAliveAttempts`	Specifies the number of keep-alive attempts. Broker Server assumes that the JMS client connection is down if the client fails to respond to all the consecutive keep-alive attempts. The default value is `1`.
`com.webmethods.jms.broker.keepAliveTimeout`	Specifies the time in seconds the Broker Server will wait for the client to reply to a keep-alive message sent. If the client does not respond to Broker Server's keep-live message within the keepAliveTimeout time, Broker Server times out the client connection. The default value is `60`.

Configuring Keep-alive for JMS Clients

Property	Description
`com.webmethods.jms.keepAliveInterval`	Specifies the time interval in seconds for sending the keep-alive messages from the JMS client to the Broker Server. Sending keep-alive messages in regular intervals helps in identifying dead connections. This feature is very useful when you configure your messaging system in a wide area network. Set the value to `0`(zero) to disable the keep-alive feature. The default value is `30`.
`com.webmethods.jms.keepAliveAttempts`	

Property	Description
	Specifies the number of keep-alive attempts. JMS client assumes that the Broker Server connection is down if the Broker Server fails to respond to all the consecutive keep-alive attempts. The default value is 2.

4 The JMSAdmin Command-Line Tool

Overview

This chapter describes the JMSAdmin command-line tool. The command-line tool allows you to:

- Generate and manage administered objects for JMS and C# clients.
- Navigate through the hierarchy of JNDI contexts.
- Create the Broker through which clients exchange messages.
- Manage permissions for topics and queues.
- Create batch files of JMSAdmin commands to automate the management of administered objects.

Modes of Operation

JMSAdmin is a command interpreter that you use with webMethods clients for JMS and C#. When you start JMSAdmin, it opens a session that accepts and executes commands.

You can use the command-line tool in either of two modes, described in the following table. For information on the flags and syntax used for these modes of operation, see "Start-up Options" on page 47.

Mode	Usage
Interactive	Invoke JMSAdmin from a command prompt and execute administrative commands one at a time. After JMSAdmin successfully completes a command (or sequence of commands) and displays any status messages, you can enter another command. If the command was unsuccessful, error information is displayed. When you use JMSAdmin interactively, a session does not terminate until you explicitly end it with an `end` or `quit` command.
Batch	Invoke JMSAdmin against a batch file containing a sequence of JMSAdmin commands. When you use JMSAdmin in batch mode, you do not need to explicitly terminate a session. This mode is useful if you need to enter a large number of commands and plan to use the same command sequence numerous times.

Before Using JMSAdmin

To run JMSAdmin, the following requirements must be met:

- The machine on which you are running JMSAdmin must be equipped with the appropriate classes for your JNDI provider.

Note:
If you are storing C# created administered objects in LDAP, note that LDAP is included in the Java run time that comes with JMSAdmin.

- JMSAdmin must have read and write authorization to the JNDI or LDAP store on the JNDI or LDAP server.
- JMSAdmin must have administrator permissions; that is, the application must have Broker administrator permissions (that is, belong to the admin client group; for more information, see *Administering webMethods Broker*).
- If you are using an LDAP server, that server must have the Java object schema installed or be running with schema validation turned off.
- If you are using webMethods Broker as a JMS provider with a JNDI properties file, that file must exist containing the properties necessary to establish a connection with your JNDI provider.
- If you will be executing commands that create structures on the Broker, the machine running JMSAdmin must have Broker access.
- If your Broker Server is enabled with the basic authentication mechanism, a client identity (username and password for basic authentication) is required to access the Broker Server. In addition, the client identity must match an Access Control List (ACL) that permits administrative access to the Broker Server.
- If your Broker Server is SSL-enabled, the client on which JMSAdmin is installed must have a valid digital certificate in a keystore, trust store, and password. In addition, the client identity (distinguished name and authenticator's distinguished name) must match an Access Control List (ACL) that permits administrative access to the Broker.

Starting JMSAdmin

To start JMSAdmin, run *Software AG_directory* \Broker\bin\jmsadmin.exe (on Windows systems) or *Software AG_directory* /Broker/bin/jmsadmin (on UNIX systems).

Start-up Options

The following table describes start-up options for JMSAdmin. Some options have both a short and long form. The long form is useful in JMSAdmin batch files because it makes the commands more readable.

Option	Description
`-c command(s)`	Executes the specified command(s) when JMSAdmin starts. To specify more than one command, separate the commands with semicolons.

Option	Description
	When you use this option, JMSAdmin executes the commands and when finished leaves you at the JMSAdmin command prompt. If you specify both the `-c`and `-f` options (see below), the `-c` option is executed first. When using variables with `-c` command lists, use the percent sign (%) as the first character in the variable name instead of the dollar sign ($). For example, the variable `%myRemoteHost` could be used to represent the name of a remote host in a script.
`-q`	Enables quiet mode, which suppresses confirmation messages such as those JMSAdmin issues before deleting a Broker or a context that is not empty. This option is most useful when running JMSAdmin in batch mode.
`-f` *`filename`*	Starts JMSAdmin and executes commands from the specified file rather than from the command prompt. Use this option to run JMSAdmin in batch mode.
`-h`	Displays a brief description of the startup options.
`-p` *`propertiesFile`* or `-properties` *`propertiesFile`*	Reads properties from the specified file. If you do not set this option, JMSAdmin searches the directories in the CLASSPATH for a JNDI properties file. The properties file that JMSAdmin looks for is named `jndi.properties`. For a list of properties that you can specify with JMSAdmin, see "JMSAdmin Properties" on page 52.
`-t` *`character-set-name`* or `-encoding` *`character-set-name`*	Specifies the name of the character set encoding used in imported files. If you do not specify this option, the machine's default encoding is used. The name must be a valid Java encoding.

Basic Authentication Enabled Brokers

The following table lists the command-line options for connecting JMSAdmin to a Broker that is basic authentication enabled.

Option	Description
`-u` *`username`* or `-user` *`username`*	Specifies the username that JMSAdmin will use to connect to a basic authentication-enabled Broker.

Option	Description
`-w password` or `-password password`	Specifies the password that JMSAdmin will use to connect to a basic authentication-enabled Broker.

You can set basic authentication parameters by using JMSAdmin and providing the values on the command line. The values you provide by using JMSAdmin override the values you set in the jndi.properties file.

For additional information about basic authentication configuration, see *Administering webMethods Broker*.

SSL-Enabled Brokers

The following table lists command-line options for connecting JMSAdmin to a Broker that is SSL-enabled.

Option	Description

`-s keystore`
or
`-sslKeystore keystore`
	Specifies the file that holds the digital certificate JMSAdmin will use to connect to an SSL-enabled Broker.
	Specifying this option sets the value of the *$sslKeystore* system variable.

`-w password`
or
`-sslPassword password`
	Specifies the password JMSAdmin is to use when connecting to the Broker using SSL.
	Specifying this option sets the value of the *$sslPassword* system variable.

`-r truststore`
or
`-sslTruststore truststore`
	Specifies the file that holds the trusted root, or CA public key that JMSAdmin will use to connect to an SSL-enabled Broker.
	Specifying this option sets the value of the *$sslTruststore* system variable.

`-e`
or
`-sslEncrypted`
	Specifies that communication between JMSAdmin and the Broker is to be encrypted.
	This option does not take an argument. If the option is not used, JMSAdmin will use unencrypted communications.
	Specifying this option sets the value of the *$sslEncrypted* system variable to "true."

You can set SSL parameters by using JMSAdmin and providing the values on the command line. You can also specify the SSL parameter values by setting the SSL-related system variables after JMSAdmin is running, which overrides any SSL options set via the command line.

For additional information about SSL configuration, see *Administering webMethods Broker*.

JMSAdmin Command Syntax

Following are some basic rules of JMSAdmin command syntax:

Category	Rule
Case sensitivity	JMSAdmin commands are *not* case sensitive.
Command forms	Some command names have long and short forms. For example, you can execute the copy command as either `copy` or `cp`.
White space	Spaces and tab characters are considered white space. You can use empty lines in a command sequence to increase readability, such as in a batch file.
Use of quotes	If a parameter value is a keyword used by JMSAdmin, or contains spaces or special characters, enclose the value in single or double quotes (for example, "My Directory", "$$NewVariableVal", and 'QueueConnectionFactory').

Continuation character	You can use the hyphen (-) at the end of a line as a continuation character.
	
	The following code fragment shows the use of a hyphen as a continuation character:
	`bind cf publicCF with -`
	`brokerHost=$bHost -`
	`brokerName=$bName`

Multiple commands	You can place multiple commands on a single line if the commands are separated by semicolons (;). The maximum length of a command line is limited only by the constraints of your operating environment.

Comments	Single-line comments are indicated by the pound-sign character (#), as shown in the following example:
	`# This sequence creates a context at the top level`
	`# and switches to the new context.`
	`create context TestContext # Creates the new context.`

JMSAdmin Variables

JMSAdmin allows you to save values that are needed in several places as variables. The variables you create must begin with the dollar sign character ($) or percent sign (%); for example, the following two variables are the same:

```
$someVariable
%someVariable
```

Variables may contain letters, numbers, and underscores. Variable names that begin with a dollar followed by an underscore ($_) designate read-only variables.

Using Variables to Set Parameters

You can use the `set` command to set variables and then use the variables as parameter values in JMSAdmin commands.

The following example shows how you might use variables to set the parameters in a `bind` command:

```
set $bHost=Aries; set $bName=InvControl
.
.
.
bind tcf publicTCF with brokerHost=$bHost brokerName=$bName
```

JMSAdmin cannot use environment variables that you set in the shell or operating environment prior to launching JMSAdmin. It can only reference local variables; that is, variables set during the JMSAdmin session.

System Variables

JMSAdmin defines a number of system variables for specific uses. These variables are defined in the following table.

Name	Description
$_Broker	Name of the Broker to which there is a current connection. If there is no connection, the value is an empty string. For more information, see the Connect and Disconnect commands.
$_Server	Name of the server to which there is a current connection. If there is no connection, the value is an empty string. For more information, see the Connect and Disconnect commands.
$sslKeystore	Name of the file containing the SSL certificate to use when making Broker connections.
$sslPassword	Password to use when making SSL connections to the Broker.
$sslEncrypted	Boolean flag that turns on encryption when making SSL connections to the Broker.
$sslTruststore	Name of the file containing the trusted root (CA public key) to use when making SSL connections to the Broker.
$user	The user name to use when making the basic authentication connection to the Broker.

Name	Description
$password	Password to use when making the basic authentication connection to the Broker.

Note:
The SSL variables in the above table are used for JMSAdmin SSL connections to the Broker for the creation of Broker objects. They are *not* used for SSL settings for connection factories, topics, and queues.

Managing Contexts

JNDI divides its namespace into hierarchical *contexts*. The JNDI context structure is similar to the directory structure in a file system. A context can contain subcontexts which themselves can contain additional subcontexts.

The root context in the namespace within which JMSAdmin operates is referred to as the *initial context*. You specify the initial context for JNDI by setting the `java.naming.provider.url` property in the properties file. JMSAdmin uses this setting to locate the JNDI server and to position itself at its starting point (its initial context) within the server's namespace.

Within the initial context, you can use JMSAdmin commands to create subcontexts. To navigate among the contexts in the namespace, you execute the JMSAdmin change context command. Like a file system, you can switch contexts using absolute or relative path notation. Here are some examples:

```
change context  /JAZZ/inventoryIn # absolute path
change context receipts/arrivalNotice # relative path (down)
change context  ../../inventoryOut # relative path (up)
```

JMSAdmin Properties

Following is a list of the properties you can specify with JMSAdmin, using the `-p` (properties) flag and a JNDI properties file.

Property	Description
`com.webmethods.jms.admin.namePrefix`	Property used to prefix names with the specified string value when accessing the JNDI provider. This property is especially useful for LDAP servers that require names to have a particular component identifier; for LDAP, the value would typically be "cn=".
`com.webmethods.jms.admin.useDirectory`	Property used to specify an initial directory context rather than an initial context.

Property	Description
	JMSAdmin uses a directory context for JNDI providers where the provider URL starts with LDAP. If this is not the case, but the provider requires an initial directory context, specify this property with a value of `true`.
`java.naming.factory.initial`	JNDI property used to locate the initial context.
`java.naming.provider.url`	JNDI property used to locate the JNDI provider.
`java.naming.security.principal`	JNDI property used to identify the user to JNDI (if required).
`java.naming.security.credentials`	JNDI property used to present the password for the JNDI user (if required).
`com.webmethods.jms.naming.clientgroup`	JMS Naming Service Provider property required for write access to JMS Naming contexts (not required for other JNDI providers). If write access is required, this property should be set to `admin`. Setting the property to the name of another client group will allow you to read but not modify JMS Naming objects.
`com.webmethods.jms.naming.keystore`	JMS Naming Service Provider property used to pass in the SSL keystore filename. (Used only with SSL connections to the Broker.)
`com.webmethods.jms.naming.keystoretype`	JMS Naming Service Provider property used to pass in the SSL keystore's file type. If you must specify, set the value to `PKCS12`.
`com.webmethods.jms.naming.truststore`	JMS Naming Service Provider property used to pass in the SSL truststore that contains the trusted root certificates.
`com.webmethods.jms.naming.truststoretype`	JMS Naming Service Provider property used to pass in the SSL truststore's file type. If you must specify, set the value to JKS.
`com.webmethods.jms.naming.encrypted`	JMS Naming Service Provider property used to turn SSL encryption on or off. (Used only with SSL connections to the Broker.)

Sample JNDI Properties File

Following is an example of a JNDI properties file used for an SSL-enabled Broker connection on Windows. The JNDI provider is LDAP. The numbers in brackets are used for illustrative purposes; they are not included in the file.

```
[1] java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
[2] java.naming.provider.url=ldap://adam.south.acme.com/
            ou=jmsdev,ou=pd,o=wm
[3] java.naming.security.principal="cn=Manager,ou=jmsdev,ou=pd,o=wm"
[4] java.naming.security.credentials=JMSatAxyz
[5] com.webmethods.jms.naming.keystore=C:/BrokerSSL/MyKeystore.p12
[6] com.webmethods.jms.naming.truststore=C:/BrokerSSL/MyTruststore.jks
[7] com.webmethods.jms.naming.encrypted=True
```

The sample properties file specifies:

1. The factory class LdapCtxFactory as the initial context for the JNDI provider.
2. The location of the JNDI provider. In this case, LDAP component identifiers are used.
3. The distinguished name that the JNDI client uses to connect to the Broker. This property must be set if the Broker connection is SSL-enabled.
4. The password that the JNDI client uses to connect to the Broker; here, the password is set to "JMSatAxyz".
5. The fully-qualified name of the file containing the SSL certificate that the JNDI client uses to connect to the Broker.
6. Whether message traffic between the JNDI client and the Broker is encrypted. A value of "true," the setting used here, turns on encryption.

Error Messages

If an error occurs while JMSAdmin is running, JMSAdmin writes the error messages to `stderr` and continues to the next command. An error does not cause JMSAdmin to exit a command sequence or terminate the JMSAdmin session.

Sample JMSAdmin Command Sequence

The following is an example of a JMSAdmin command sequence that binds administered objects for a pub/sub application called JAZZ.

For more examples of using JMSAdmin to configure administered objects, see “Administrative Setup Commands” on page 178.

```
[1] java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
[2] java.naming.provider.url=ldap://adam.south.acme.com/
            ou=jmsdev,ou=pd,o=wm
[3] java.naming.security.principal="cn=Manager,ou=jmsdev,ou=pd,o=wm"
[4] java.naming.security.credentials=JMSatAxyz
```

```
[5] com.webmethods.jms.naming.keystore=C:/Temp/BrokerSSL/NewCert6d.cert
[6] com.webmethods.jms.naming.encrypted=True
```

The preceding JMSAdmin command sequence:

1. Starts JMSAdmin using the properties file `myJNDI.prop`.
2. Sets the variable *$bHost* to a host server named `Aries` and the variable *$bName* to a Broker instance named InvControl.
3. Creates a new context, `JAZZ`, under the initial context.
4. Switches to the new context.
5. Binds a connection factory in JNDI to the lookup name `xInvBroker` for a connection to the Broker named `InvControl` on the host server `Aries` (default port).
6. Binds a topic to lookup name `xNewSuppliers` called `newSuppliers`.
7. Ends the JMSAdmin interactive session.

5 JMSAdmin Command Reference

Overview

This chapter is a command reference for the JMSAdmin command-line tool.

- For information about using this tool to configure and manage administered objects (for both JMS and C# clients), refer to "The JMSAdmin Command-Line Tool" on page 45.
- For information about how to launch JMSAdmin, see "Starting JMSAdmin" on page 47.
- For information about the command-line parameters you use with JMSAdmin, see "Start-up Options" on page 47.

Command Syntax Conventions

Following is a list of the font, style, and symbol conventions used for JMSAdmin command syntax.

Convention	Description
`Typewriter font`	Identifies characters and values that you must type exactly as specified.
Italic typewriter font	Identifies variable information that you must supply or change when using the command.
Bold typewriter font	Identifies command name, parameter name, and parameter value information to be typed exactly as specified.
|	Specifies an "or" logical operator between two options.
[ ]	Specifies optional keywords or values. Do not type the [ ] symbols in your own code.
{ }	Indicates a selection to be made between two or more choices. Do not type the { } symbols in your own code.
()	Indicates group or set of elements. Do not type the ( ) symbols in your own code. Separate the elements in the group or set with a comma (,).
+	Indicates that the last grouped element or set of elements in the command can be repeated. Do not type the + symbol in your own code.

JMSAdmin Command Descriptions

This section describes the commands available for use by the JMSAdmin command-line tool. The commands are organized alphabetically, with syntax and parameter descriptions for each command, along with explanatory material and examples where appropriate.

Commands or parameters available only to JMS clients or C# clients are so noted.

Bind ClusterConnectionFactory

Adds a Cluster ConnectionFactory definition into a JNDI context.

Syntax

```
bind  { cl_cf | ClusterConnectionFactory } lookup-name  [ with

  clusterName=cluster-name clusterBrokers=(broker-name@broker-server:port)
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ weights=(weight-per-broker) ]
  [ multiSendCount=broker-count-for-multisend-policy ]
  [ includeAllBrokers={ true | false } ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
`clusterName`	*Required.* The name of the cluster containing the Brokers. If `includeAllBrokers` property is set to `true`, the name of the cluster and the first Broker in the cluster is used to identify the remaining Brokers in the cluster. When `includeAllBrokers` is set to `true`, set the value of `clusterName` correctly to retrieve all the other Brokers in the cluster.

`clusterBrokers`	*Required.* List of Brokers in the cluster that will be part of this connection factory. A cluster Broker is specified in the form of *broker-name@broker-server:port*. The list of Brokers in the cluster is specified by separating the cluster Brokers with a comma (,).
	
	For example, set the value of `clusterBrokers` as following:
	`clusterBrokers=broker_01@hostname:port, broker_02@hostname:port`
	
	or use variables to specify the list of Brokers:
	`set $brkr_set=broker_01@hostname:port, broker_02@hostname:portclusterBrokers=$brkr_set`

Part	Description
`clusterRefreshInterval`	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of `clusterRefreshInterval` property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `WEIGHTED_ROUND_ROBIN` ■ `MULTISEND_GUARANTEED` ■ `MULTISEND_BEST_EFFORT` **Note:**`MULTISEND_GUARANTEED` cluster policy is applicable only to XA connection factories. `MULTISEND_BEST_EFFORT` policy does not work with XA connection factory.
`weights`	*Optional.* A comma separated list of weights of each Broker in the cluster governed by `WEIGHTED_ROUND_ROBIN` policy. Specify the weight of each Broker in the same order as specified in the `clusterBrokers` property. The weights must be greater than `0`. For example, `weights=10,20`. If you do not specify a weight for a Broker, a weight of `1` is set for that Broker. **Note:** Set this property only when `clusterPolicy` property value is set to `WEIGHTED_ROUND_ROBIN` policy.
`multiSendCount`	*Optional.* The number of Brokers in the cluster governed by `MULTISEND_GUARANTEED` cluster policy. The `multiSendCount` property value must be greater than 0 and less than the number of Brokers in the cluster. **Note:**

Part	Description
	Set this property only when `clusterPolicy` property value is set to `MULTISEND_GUARANTEED` policy.
`includeAllBrokers`	*Optional.* Whether to include all the Brokers in the cluster with this connection factory. ■ If set to `true`, all the Brokers in the cluster are included in this connection factory. The value of `clusterName` property and the first Broker in the cluster is used to retrieve all the other Brokers in the cluster. ■ If set to `false`, all Brokers are not included in this connection factory. The default value is `false`.
`clientID`	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted.

Part	Description
	■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
bind ClusterConnectionFactory TEST_CONNECTION_FACTORY with
  clusterName=cluster_01
  clusterBrokers=broker_01@hostname:port,broker_02@hostname:port
  clusterRefreshInterval=640
  clusterPolicy=WEIGHTED_ROUND_ROBIN
  weights=10,20
show TEST_CONNECTION_FACTORY
TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  clusterRefreshInterval: 640
  clusterPolicy: WEIGHTED_ROUND_ROBIN
  weights: 10,20
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify ClusterConnectionFactory" on page 109

Bind ClusterQueueConnectionFactory

Adds a Cluster Queue ConnectionFactory definition into a JNDI context.

Syntax

```
bind  { cl_qcf | ClusterQueueConnectionFactory } lookup-name  [ with
  clusterName=cluster-name
  clusterBrokers=(broker-name@broker-server:port)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ weights=(weight-per-broker)+ ]
  [ multiSendCount=broker-count-for-multisend-policy ]
  [ includeAllBrokers={ true | false } ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
`clusterName`	*Required.* The name of the cluster containing the Brokers. If `includeAllBrokers` is set to `true`, the name of the cluster and the first Broker in the cluster is used to identify the remaining Brokers in the cluster. When `includeAllBrokers` is set to `true`, set the value of `clusterName` correctly to retrieve all the other Brokers in the cluster.
`clusterBrokers`	*Required.*List of Brokers in the cluster that will be part of this connection factory. A cluster Broker is specified in the form of *broker-name@broker-server:port*. The list of Brokers in the cluster is specified by separating the cluster Brokers with a comma (,). For example, set the value of `clusterBrokers` as following: `clusterBrokers=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>`

Part	Description
	or use variables to specify the list of Brokers: `set $brkr_set=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>clusterBrokers=$brkr_set`
`clusterRefreshInterval`	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `WEIGHTED_ROUND_ROBIN` ■ `MULTISEND_GUARANTEED` ■ `MULTISEND_BEST_EFFORT` **Note:**`MULTISEND_GUARANTEED` cluster policy is applicable only to XA connection factories. `MULTISEND_BEST_EFFORT` policy does not work with XA connection factory.
`weights`	*Optional.* A comma separated list of weights of each Broker in the cluster governed by `WEIGHTED_ROUND_ROBIN` policy. Specify the weight of each Broker in the same order as specified in the `clusterBrokers` property. The weights must be greater than `0`. For example, `weights=10,20.` If you do not specify a weight for a Broker, a weight of `1` is set for that Broker. **Note:** Set this property only when `clusterPolicy` property value is set to `WEIGHTED_ROUND_ROBIN` policy.
`multiSendCount`	*Optional.* The number of Brokers in the cluster governed by `MULTISEND_GUARANTEED` cluster policy. The value of this property

Part	Description
	must be greater than 0 and less than the number of Brokers in the cluster. **Note:** Set this property only when `clusterPolicy` property value is set to `MULTISEND_GUARANTEED` policy.
`includeAllBrokers`	*Optional.* Whether to include all the Brokers in the cluster with this connection factory. ■ If set to `true`, all the Brokers in the cluster are included in this connection factory. The value of `clusterName` property and the first Broker in the cluster is used to retrieve all the other Brokers in the cluster. ■ If set to `false`, all the Brokers are not included in this connection factory. The default value is `false`.
`clientID`	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted.

Part	Description
	■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this cluster queue connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified cluster queue connection factory with an application server, specifying `false` saves some of the overhead associated with support for cluster XA queue connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
bind ClusterQueueConnectionFactory TEST_CONNECTION_FACTORY with
  clusterName=cluster_01
  clusterBrokers=$brkr_set
  clusterPolicy=ROUND_ROBIN
  clusterRefreshInterval=640
  useXA=true

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmXAQueueConnectionFactoryImpl
clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 640
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
```

```
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify ClusterQueueConnectionFactory" on page 113

Bind ClusterTopicConnectionFactory

Adds a Cluster Topic ConnectionFactory definition into a JNDI context.

Syntax

```
bind  { cl_tcf | ClusterTopicConnectionFactory }lookup-name
[ withclusterName=cluster-name clusterBrokers=
(broker-name@broker-server:port)+
  [   clusterRefreshInterval=cluster-refresh-interval ]
  [   clusterPolicy=cluster-policy-name ]
  [   weights=(weight-per-broker)+ ]
  [   multiSendCount=broker-count-for-multisend-policy ]
  [   includeAllBrokers={ true | false }]
  [   clientId=client-id ]
  [   application=application-name ]
  [   group=client-group-name ]
  [   sslKeystore=filename ]
  [   sslPassword=password ]
  [   sslTruststore=filename ]
  [   sslEncrypted={ true | false }]
  [   useXA={ true | false }]
  [   marshallInClassName=class-name ]
  [   marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterName	*Required.* The name of the cluster containing the Brokers. During run time, If includeAllBrokers is set to true, the name of the cluster and the first Broker in the cluster is used to identify the remaining Brokers in the cluster. When includeAllBrokers is set to true, set the value of clusterName correctly to retrieve all the other Brokers in the cluster.
clusterBrokers	*Required.*List of Brokers in the cluster that will be part of this connection factory. A cluster Broker is specified in the form of *broker-name@broker-server:port*. The list of Brokers in the cluster is specified by separating the cluster Brokers with a comma (,).

Part	Description

	For example, set the value of `clusterBrokers` as following:
	`clusterBrokers=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>`
	or use variables to specify the list of Brokers:
	`set $brkr_set=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>clusterBrokers=$brkr_set`

`clusterRefreshInterval`	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the cluster routing policy. Specify any of the following policies: - `ROUND_ROBIN` (the default value) - `STICKY` - `RANDOM` - `WEIGHTED_ROUND_ROBIN` - `MULTISEND_GUARANTEED` - `MULTISEND_BEST_EFFORT` **Note:**`MULTISEND_GUARANTEED` cluster policy is applicable only to XA connection factories. `MULTISEND_BEST_EFFORT` policy does not work with XA connection factory.
`weights`	*Optional.* A comma separated list of weights of each Broker in the cluster governed by `WEIGHTED_ROUND_ROBIN` policy. Specify the weight of each Broker in the same order as specified in the `clusterBrokers` property. The weights must be greater than `0`. For example, `weights=10,20.` If you do not specify a weight for a Broker, a weight of `1` is set for that Broker. **Note:** Set this property only when `clusterPolicy` property value is set to `WEIGHTED_ROUND_ROBIN` policy.

Part	Description
`multiSendCount`	*Optional.* The number of Brokers in the cluster governed by `MULTISEND_GUARANTEED` cluster policy. The value of this property must be greater than 0 and less than the number of Brokers in the cluster. **Note:** Set this property only when `clusterPolicy` property value is set to `MULTISEND_GUARANTEED` policy.
`includeAllBrokers`	*Optional.* Whether to include all the Brokers in the cluster with this connection factory. ■ If set to `true`, all the Brokers in the cluster are included in this connection factory. The value of `clusterName` property and the first Broker in the cluster is used to retrieve all the other Brokers in the cluster. ■ If set to `false`, all the Brokers are not included in this connection factory. The default value is `false`.
`clientID`	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.

Part	Description
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this cluster topic connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified cluster topic connection factory with an application server, specifying `false` saves some of the overhead associated with support for cluster XA topic connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
bind ClusterTopicConnectionFactory TEST_CONNECTION_FACTORY with
  clusterName=cluster_01
  clusterBrokers=$brkr_set
  clusterPolicy=ROUND_ROBIN
  clusterRefreshInterval=640
  useXA=true

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmXATopicConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 640
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
```

```
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify ClusterTopicConnectionFactory" on page 117

Bind CompositeConnectionFactory

Adds a Composite ConnectionFactory definition into a JNDI context.

Syntax

```
bind  { cm_cf | CompositeConnectionFactory } lookup-name  [ with

  clusterConnectionFactories=(connection-factory-name)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterConnectionFactories	*Required.*List of cluster connection factory names that are part of the composite connection factory. These cluster connection factories must be created prior to binding them to a composite connection factory. The list of cluster connection factory names is specified by separating the cluster connection factory names with a comma (,). For example, `clusterConnectionFactories=clusterFactory_01, clusterFactory_02`
clusterRefreshInterval	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker

Part	Description
	connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the composite cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `MULTISEND_BEST_EFFORT` ■ `MULTISEND_GUARANTEED`
`useXA`	*Optional.* Whether XA transactions are allowed with this composite connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified composite connection factory with an application server, specifying `false` saves some of the overhead associated with support for composite XA connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
bind CompositeConnectionFactory TEST_CONNECTION_FACTORY with
  clusterConnectionFactories=clusterFactory_01,clusterFactory_02
  clusterPolicy=ROUND_ROBIN
  clusterRefreshInterval=640
  useXA=false
show TEST_CONNECTION_FACTORY
TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_01,clusterFactory_02
  clusterRefreshInterval: 640
```

```
  clusterPolicy: ROUND_ROBIN
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
Cluster Factories: 2
Cluster Factory [0]: clusterFactory_01
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
Cluster Factory [1]: clusterFactory_02
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify CompositeConnectionFactory" on page 122

Bind CompositeQueueConnectionFactory

Adds a Composite Queue ConnectionFactory definition into a JNDI context.

Syntax

```
bind  { cm_qcf | CompositeQueueConnectionFactory } lookup-name  [ with

  clusterConnectionFactories=(cluster-queue-connection-factory-name)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ useXA={ true | false } ]
```

```
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterConnectionFactories	*Required.*List of cluster queue connection factory names that will be part of the composite queue connection factory. These cluster connection factories must be created prior to binding them to a composite connection factory. The list of cluster queue connection factory names is specified by separating the cluster queue connection factory names with a comma (,). For example, `clusterConnectionFactories=cf_01,cf_02`
clusterRefreshInterval	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
clusterPolicy	*Optional.*The name of the composite cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `MULTISEND_BEST_EFFORT` ■ `MULTISEND_GUARANTEED`
useXA	*Optional.* Whether XA transactions are allowed with this composite queue connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified composite queue connection factory with an application server, specifying `false` saves some of the overhead associated with support for composite XA queue connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.

Part	Description
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
bind CompositeQueueConnectionFactory TEST_CONNECTION_FACTORY with
  clusterConnectionFactories=clusterFactory_01,clusterFactory_02
  clusterPolicy=ROUND_ROBIN
  clusterRefreshInterval=640
  useXA=false

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmQueueConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_01,clusterFactory_02
  clusterRefreshInterval: 640
  clusterPolicy: ROUND_ROBIN
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
Cluster Factories: 2
Cluster Factory [0]: clusterFactory_01
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
Cluster Factory [1]: clusterFactory_02
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
```

```
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify CompositeQueueConnectionFactory" on page 125

Bind CompositeTopicConnectionFactory

Adds a Composite Topic ConnectionFactory definition into a JNDI context.

Syntax

```
bind  { cm_tcf | CompositeTopicConnectionFactory } lookup-name  [ with

  clusterConnectionFactories=(cluster-topic-connection-factory-name)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterConnectionFactories	*Required.*List of cluster topic connection factory names that will be part of the composite topic connection factory. These cluster connection factories must be created prior to binding them to a composite connection factory. The list of cluster topic connection factory names is specified by separating the cluster topic connection factory names with a comma (,). For example, `clusterConnectionFactories=clusterFactory_01, clusterFactory_02`
clusterRefreshInterval	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection

Part	Description
	factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the composite cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `MULTISEND_BEST_EFFORT` ■ `MULTISEND_GUARANTEED`
`useXA`	*Optional.* Whether XA transactions are allowed with this composite topic connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified composite topic connection factory with an application server, specifying `false` saves some of the overhead associated with support for composite XA topic connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

```
bind CompositeTopicConnectionFactory TEST_CONNECTION_FACTORY with
  clusterConnectionFactories=clusterFactory_01,clusterFactory_02
  clusterPolicy=ROUND_ROBIN
  clusterRefreshInterval=640
  useXA=false

show TEST_CONNECTION_FACTORY
```

```
TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmTopicConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_01,clusterFactory_02
  clusterRefreshInterval: 640
  clusterPolicy: ROUND_ROBIN
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
Cluster Factories: 2
Cluster Factory [0]: clusterFactory_01
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
Cluster Factory [1]: clusterFactory_02
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify CompositeTopicConnectionFactory" on page 128

Bind ConnectionFactory

Adds a ConnectionFactory definition into a JNDI context.

```
bind  { cf | ConnectionFactory } lookup-name  [ with

  [ { brokerHost | bh }=host[:port] ]
  [ { brokerName | bn }=broker-name ]
  [ clientId=client-id ]
  [ application=application-name ]
```

```
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
brokerHost	■ *Optional.*The host part is the host name of the Broker Server on which the Broker resides. The default value is localhost. ■ *Optional.*The port part is the port number of the Broker Server on which the Broker resides. The default value is 6849.
brokerName	*Optional.* The name of the Broker. If you do not specify a Broker name, JMSAdmin uses the default Broker on the specified Broker Server.
clientID	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
application	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
group	*Optional.* The client group name that clients created from this connection factory will use. Note that the bind command does not create the client group; you must create a client group explicitly using the create group command. The default *client-group-name* is JMSClient.
sslKeystore	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
sslPassword	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
sslTruststore	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
sslEncrypted	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted.

Part	Description
	■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Related Commands

"Modify ConnectionFactory" on page 131

Bind Queue

Adds a queue definition into a JNDI context.

Syntax

```
bind queue lookup-name with { queueName | qn }=broker-queue-name
  [ priorityOrdering={ true | false } ]
  [ { sharedState | ss }={ true | false } ]
  [ { sharedStateOrdering | sso }={ none | publisher } ]
```

The command syntax has these parts:

Part	Description
`lookup-name`	*Required.* The name used to look up this queue in the JNDI directory.

Part	Description
`queueName`	*Required.* The name to give to the queue, which must meet the following criteria: ■ The first character cannot be a pound sign (#). ■ The name cannot contain the at sign (@), forward slash (/), or colon (:). ■ The name cannot exceed 255 bytes. The `queueName` can be the same as the `lookup-name`. If JMS clients will send and receive messages to the queue from other Brokers in a territory, specify `queueName` using a fully qualified name (for example, `/territoryName/brokerName/queueName`).
`priorityOrdering`	*Optional.*Whether or not the queue orders messages by priority. The default value is `true`.
`sharedState`	*Optional.* Whether to allow multiple connections to share the same queue on the Broker. The default value is `false`. If `sharedState` is set to `true`, webMethods Broker used as a JMS provider creates Broker clients as shared state clients for the queue connection.
`sharedStateOrdering`	*Optional.* The ordering of documents received on a shared state client. The default value is `none`. ■ To receive documents in order from a publisher, set to `publisher`. ■ To receive documents in any order, set to `none`.

Related Commands

"Create Queue" on page 96

"Delete Queue" on page 101

"Modify Queue" on page 133

Bind QueueConnectionFactory

Adds a QueueConnectionFactory definition into a JNDI context (JMS clients only).

Syntax

```
bind  { qcf | QueueConnectionFactory }  lookup-name  [ with ]

  [ { brokerHost | bh }=host[:port] ]
  [ { brokerName | bn }=broker-name ]
  [ clientId=client-id ]
```

```
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
brokerHost	■ *Optional.*The `host` part is the host name of the Broker Server on which the Broker resides. The default value is `localhost`. ■ *Optional.*The `port` part is the port number of the Broker Server on which the Broker resides. The default value is `6849`.
brokerName	*Optional.* The name of the Broker. If you do not specify a Broker name, JMSAdmin uses the default Broker on the specified Broker Server.
clientID	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
application	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
group	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
sslKeystore	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
sslPassword	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
sslTruststore	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
sslEncrypted	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted.

Part	Description
	■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller.
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller.

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Related Commands

"Bind ConnectionFactory" on page 78

"Modify QueueConnectionFactory" on page 134

Bind String

Adds (binds) a specified string into a JNDI context (JMS clients only).

Syntax

```
bind string lookup-name
"stringName"
```

Remarks

- Specify the value of the string, enclosed in quotes, in `stringName`. In the `lookup-name` parameter, specify the key with which to look up the string in JNDI.
- Binding string values to a JNDI context is one convenient way for an application program to store and retrieve values such as configuration parameters.

Related Commands

"Modify String" on page 136

Bind Topic

Adds a topic definition into a JNDI context.

Syntax

```
bind topic lookup-name with  { topicName | tn }=document-type-name

  [ deadLetterOnly={ true | false } ]
  [ localOnly={ true | false } ]
  [ priorityOrdering={ true | false } ]
  [ event type=event-type-name ]
  [ { sharedState | ss }={ true | false } ]
  [ { sharedStateOrdering | sso }={ none | publisher } ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this topic in the JNDI directory.
topicName	*Required.* The name of the topic (destination) through which publishers and subscribers will exchange messages. The name must meet the following criteria: ■ The first character cannot be a pound sign (#). ■ The name cannot contain the at sign (@), forward slash (/), or colon (:). However, topic names can contain colons when using event type scopes (for example, `MyTopics::Topic1` is allowed; `MyTopics:Topic1` is not). ■ Each name segment cannot exceed 255 bytes.
deadLetterOnly	*Optional.*If you set this option to `true`, only dead letter subscriptions will be received. The default value is `false`.
localOnly	*Optional.* If you set this option to `true`, only messages originating (published) from the local machine will be received. The default value is `false`.
priorityOrdering	*Optional.*Whether or not the subscribers' queues order messages by priority. The default value is `true`.
eventType	*Optional.* If you specify an `event-type-name`, the supplied name will be used for the document name on the Broker. The name must meet the following criteria:

Part	Description
	■ The first character cannot be a pound sign (#). ■ The name cannot contain the at sign (@), forward slash (/), or colon (:). ■ Each name segment cannot exceed 255 bytes. The `event-type-name` can be the same as the `lookup-name` and the `topicName`; the default `eventType` is the same as the `topicName`.
`sharedState`	*Optional.* Whether to allow multiple connections to share the same Broker client for the subscription. The default value is `false`. If `sharedState` is set to `true`, webMethods Broker used as a JMS provider creates Broker clients as shared state clients for the connection.
`sharedStateOrdering`	*Optional.* The ordering of documents received on a shared state client. The default value is `none`. ■ To receive documents in order from a publisher, set to `publisher`. ■ To receive documents in any order, set to `none`.

Remarks

You can organize topic names using the scope delimiter (::). The scope can consist of zero or more levels. Following is the syntax:

```
context::subcontext::subcontext...
```

where *subcontext* segments are optional.

The name must begin with an alphabetic character followed by alphanumeric characters or an underscore (_). The following are valid `topicName` and `eventType` values:

```
paris
 stocks::all
 stocks::nasdaq::webm
 stocks::nasdaq::csco
```

A topic subscriber can also use a topic name that includes a single wildcard character, *, at the end of a topic name. For example, to subscribe to all NASDAQ stocks quotes you could use the following topic name:

```
stocks::nasdaq::*
```

If you use wildcards in the topic name, only those topics for which the subscriber has permission to subscribe are included in the subscription.

Related Commands

"Create Durable Subscriber" on page 93

"Create Topic" on page 98

"Delete Durable Subscriber" on page 100

"Delete Topic" on page 102

"Modify Topic" on page 137

Bind TopicConnectionFactory

Adds a TopicConnectionFactory definition into a given context (JMS clients only).

Syntax

```
bind  { tcf | TopicConnectionFactory } lookup-name [ with ]

  [ { brokerHost | bh }=host[:port] ]
  [ { brokerName | bn }=broker-name ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
brokerHost	■ *Optional.*The host part is the host name of the Broker Server on which the Broker resides. The default value is localhost. ■ *Optional.*The port part is the port number of the Broker Server on which the Broker resides. The default value is 6849.
brokerName	*Optional.* The name of the Broker. If you do not specify a Broker name, JMSAdmin uses the default Broker on the specified Broker Server.
clientID	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster.

Part	Description
	■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller.
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller.

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Related Commands

"Bind ConnectionFactory" on page 78

"Bind Topic" on page 84

"Modify Topic" on page 137

"Modify TopicConnectionFactory" on page 138

Change Context

Navigates a JNDI namespace.

Syntax

```
{ change | cd }  [ context | ctx ]  context-name
```

Remarks

To switch contexts, you can specify an absolute or relative path.

Path names are specified using the forward slash character (/) to separate contexts in the path. A path name that begins with the forward slash character represents a context that is relative to the initial context. A name that begins with a context name represents a context that is relative to the current context. You can use double periods (..) to represent the parent of the current context and a single period to represent the current context (.).

Examples

The following command uses an absolute path to change context:

```
\> change context /inventory/hardware/tools
```

The following command uses a relative path to change context:

```
\> change hardware/tools
```

The following command navigates to the parent of the current context and then to the lighting subcontext:

```
\> change ../lighting
```

The following command navigates to the root context:

```
\> cd /
```

Related Commands

"Create Context" on page 92

"Delete Context" on page 100

"List Context" on page 108

Connect

Establishes a connection to a Broker.

Syntax

```
{ connect | conn } [ to ] [ broker broker-name ]
   [ server server[:port]  ]
```

The command syntax has these parts:

Part	Description
`broker`	*Optional.* The name of the Broker on the Broker Server. If you omit this parameter, JMSAdmin will connect to the default broker.
`server`	*Optional.*The name of the Broker Server. If you omit this parameter, JMSAdmin will connect to `localhost`.
`port`	*Optional.* The port number for the default Broker Server. The default port number is `6849`.

Remarks

The following JMSAdmin commands require a Broker connection:

```
create group                          delete queue
create durable subscriber             delete topic
create queue                          deny
create topic                          display
delete durable subscriber             initialize broker
delete group                          permit
```

You can only use one JMSAdmin to Broker connection at a time. Creating a new connection terminates an existing one.

To use a basic authenticated connection, you must set the following system variables using the `set` command or with the proper command-line arguments at JMSAdmin startup:

$user

$password

To use an SSL connection, you must set the following system variables using the `set` command or with the proper command-line arguments at JMSAdmin startup:

$sslKeystore

$sslEncrypted

$sslDN

$sslPassword

For more information, see "System Variables" on page 51.

Examples

Connect to the following Brokers:

The default Broker on `localhost`.

```
\> connect
```

Broker "JMS Test" on `localhost`.

```
\> connect to broker "JMS Test"
```

The default Broker on the `testserver` host system using port `7849`.

```
\> connect server testserver:7849
```

Broker "JMS Test" on the default port (`6849`) of the `testserver` host system.

```
\> connect broker "JMS Test" server testserver
```

Related Commands

"Create Broker " on page 91

"Disconnect" on page 104

"Initialize Broker " on page 107

Copy

Copies an administered object to another context.

Syntax

```
{ copy | cp } { lookup-name | * } to  context-name
```

Remarks

You can specify `lookup-name` (the name of the source object) and `context-name` (the name of the target context) using relative or absolute paths. The specified target context must already exist. If an object with the same name as the source object already exists in `context-name`, JMSAdmin issues an error message.

You can use the wildcard character (*) as a way to copy all webMethods defined objects or strings bound into a context to another context.

Example

Copy the object `nTopic3` to the context `/sub4`:

```
\> copy nTopic3 to /sub4
```

Related Commands

"Move" on page 140

Create Broker

Creates a new Broker on the specified Broker Server. When you create Brokers, the JMS client groups (IS-JMS, JMSClient, and EventMembers) and Broker objects (JMS::Temporary::Queue, JMS::Temporary::Topic, Event::WebM::EventSubscriptionMgmt::SubscriptionChange, and Event::WebM::Deployment) are created by default.

By default, the "can-publish" and "can-subscribe" permissions for Broker objects:

- JMS::Temporary::Queue andJMS::Temporary::Topic are assigned to IS-JMS and JMSClient.
- Event::WebM::EventSubscriptionMgmt::SubscriptionChange is assigned to EventMembers.

Syntax

```
create [ default ] broker broker-name  [ on server[:port] ]
```

The command syntax has these parts:

Part	Description
`default`	*Optional.* If you specify this parameter, the new Broker is designated as the default Broker on the specified `server`.
`broker`	*Required.* The name of the new Broker on the Broker Server. Broker names must agree with the following rules: - The first character cannot be a pound sign (#).

Part	Description
	■ The name cannot contain the at sign (@), forward slash (/), or colon (:). ■ The name cannot exceed 255 bytes. ■ If the name contains a space, enclose `broker-name` in quotes.
`server`	*Optional.* The name of the Broker Server. If you omit this parameter, JMSAdmin will attempt to create a Broker on the Broker Server running on `localhost`.
`port`	*Optional.* The port number for the default Broker Server. If you omit this parameter, JMSAdmin will use port `6849` when creating the new Broker.

Remarks

JMSAdmin automatically connects to the new Broker when this command completes.

Examples

The following command sequence creates a Broker, initializes that Broker, and then ends the connection:

```
> create broker "JMS Broker #6" on testserver:8849
 > initialize broker
 > disconnect
```

The following command creates a Broker called `JMS Broker #6` on the Broker Server running on `localhost` at port `6849`:

```
\> create broker "JMS Broker #6"
```

Related Commands

"Connect" on page 89

"Delete Broker " on page 99

"Disconnect" on page 104

"Initialize Broker " on page 107

Create Context

Creates a new subcontext in the current context.

Syntax

```
create context context-name
```

Remarks

You can specify `context-name` using an absolute or relative path name; however, the parent contexts in the path must already exist.

Examples

The following command creates the context `ctx1`:

```
\> create context ctx1
```

The following example uses an absolute path to create a subcontext in `ctx1` and then lists the contents of the `ctx1` context:

```
> create context /ctx1/ctx2
> ls /ctx1
[0]: ctx2 ==> com.webmethods.jms.naming.WmJMSNamingContext
```

Related Commands

"Delete Context" on page 100

"List Context" on page 108

Create Durable Subscriber

Creates a durable subscriber on a Broker.

Syntax

```
create durable subscriber subscriber-name
  for topicName  using tcf-name

    [ with  [ nolocal ]  [ selector=selector-string ]
    [ user=userName  authenticator=authenticator ] ]

create durable subscriber subscriberName
  topic topicName  client-id client-id

    [ with  [ nolocal ]  [ selector=selectorString ]
    [ group=client-group-name ]
    [ application=application-name ]
    [ sharedState ={ true | false } ]
    [ sharedStateOrdering ={ none | publisher } ]
    [ user=userName ]
    [ authenticator=authenticator ]
    [ localOnly={ true | false } ]
    [ deadLetterOnly={
```

```
true | false } ]
    [ priorityOrdering={ true | false } ]
```

The command syntax has these parts:

Part	Description
subscriber-name	*Required.* The unique name of the durable subscriber. This name must match the name that the JMS client will specify in the name parameter when it calls the `session.createDurableSubscriber()` method.
topicName	*Required.* The name of the topic to which the durable subscriber will subscribe.
tcf-name	*Required* (for the *first* version of the command). The connection factory name found in JNDI (which can be a `ConnectionFactory` or `TopicConnectionFactory`). The factory name that you specify must have a client ID setting.
client-id	*Required* (for the *second* version of the command). The durable subscriber that is created uses the supplied `client-id` as part of the Broker client ID that is created.
`nolocal`	*Optional.* Suppresses the delivery of messages that are published by the durable subscriber's own connection. Include this option if the JMS client will enable the `nolocal` attribute when it calls the `session.createDurableSubscriber()` method.
`selector`	*Optional.* A message selector string. Use if the JMS client includes a message selector string when it calls the `session.createDurableSubscriber()` method.
`user`	*Optional.* If the subscription is going to be accessed from a connection using basic authentication, specify the basic authentication user. If the subscription is going to be accessed from a connection using SSL, specify the distinguished name of the user.
`authenticator`	*Optional.* If the subscription is going to be accessed from a connection using basic authentication, specify the basic authentication alias (authenticator source). This value can be found in the basicauth.cfg file. If the subscription is going to be accessed from a connection using SSL, specify the distinguished name of the certificate issuer. This value can usually be found in the certificate file.
`group`	*Optional.* The name of an existing client group for this subscriber. The default *client-group-name* is JMSClient.

Part	Description
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`sharedState`	*Optional.* Whether to allow multiple durable subscribers to share the same queue of messages. The default value is `false`. If `sharedState` is set to `true`, webMethods Broker used as a JMS provider creates Broker clients as shared state clients for the connection.
`sharedStateOrdering`	*Optional.* The ordering of documents received on a shared state client. The default value is `none`. ■ To receive documents in order from a publisher, set to `publisher`. ■ To receive documents in any order, set to `none`.
`priorityOrdering`	*Optional.*Whether or not the queue orders messages by priority. The default value is `true`.
`localOnly`	*Optional.* If you set this option to true, only messages originating (published) from the local machine will be received. The default value is false.
`deadLetterOnly`	*Optional.* If you set this option to true, only dead letter subscriptions will be received. The default value is false.

Remarks

You must establish a connection to the Broker before using this command.

If the subscriber is going to be accessed from a `Connection` or `TopicConnection` using SSL, the distinguished name that the subscriber will provide when it establishes a connection must be supplied. In addition, the distinguished name of the issuer of the certificate must be supplied.

You must also provide the actual name of the topic to which the durable subscriber subscribes and the lookup name of the connection factory that creates the connection through which the durable subscriber connects.

The second form of the command creates a durable subscriber without using JNDI.

Note:
The `create durable subscriber` commands that use the `sslDn` and `issuerDn` parameters of the previous Broker versions will continue to work. Use the `user` parameter instead of `sslDn`, and the `authenticator` parameter instead of `issuerDn`.

Example

Connect to a durable subscriber named `MySub`, which will receive selected messages from topic `Test::Topic1` through connection factory `MyTCF`.

```
> connect broker "Test JMS Broker" server testserver
 > create durable subscriber MySub for Test::Topic1 using MyTCF -
   with selector = "city='Sunnyvale'"
```

Related Commands

"Delete Durable Subscriber" on page 100

Create Group

Creates client groups on the Broker.

Syntax

```
create { group | groups } ( client-group-name )+
```

Remarks

A client group controls access to topics and queues.

When a client group is created that has no permissions for any topic or queue, the permissions must be added with the `permit` command.

Note that you must create a client group explicitly using the `create group` command (that is, you cannot create a client group implicitly using a `bind` command).

The default *client-group-name* is JMSClient.

Related Commands

"Delete Group" on page 101

"Deny" on page 103

"Permit" on page 141

Create Queue

Creates a queue's supporting client on the Broker.

Syntax

```
create  { queue | queues }  for  {  ( lookup-name )  +  |  *  }

  [ using  qcf-name
    with  user=userName  authenticator=authenticator  ]

create  { queue | queues }  ( queue-names )+ with

  [  group=client-group-name  ]
  [  application=application-name  ]
```

```
[  { sharedState | ss }={ true | false } ]
[  { sharedStateOrdering | sso }={ none | publisher } ]
[  priorityOrdering={ true | false } ]
[  user=userName  ]
[  authenticator=authenticator  ]
```

The command syntax has two forms; following are the parts for both forms:

Part	Description
`queueorqueues`	*Required.*The name used to look up this queue in the JNDI directory.
`qcf-name`	*Optional.*The connection factory name found in JNDI, which can either be a `ConnectionFactory` or `QueueConnectionFactory`.
`user`	*Optional.*If the queue is going to be accessed from a connection using basic authentication, specify the basic authentication user. If the queue is going to be accessed from a connection using SSL, specify the distinguished name of the user.
`authenticator`	*Optional.* If the queue is going to be accessed from a connection using basic authentication, specify the basic authentication alias (authenticator source). This value can be found in the basicauth.cfg file. If the queue is going to be accessed from a connection using SSL, specify the distinguished name of the certificate issuer. This value can usually be found in the certificate file.
`queue-names`	*Required.* The queue names that you want to create.
`group`	*Optional.*The name of an existing client group or a new client group. If you enter a new client group name, that client group is created. The default *client-group-name* is JMSClient.
`application`	*Optional.* The name of the client application on which to create the queue.
`priorityOrdering`	*Optional.*Whether or not the queue orders messages by priority. The default value is `true`.
`sharedState`	*Optional.* Whether to allow multiple queue receivers to concurrently read from the queue. The default value is `false`.
`sharedStateOrdering`	*Optional.*The ordering of documents received on a shared state client. The default value is `none`. ■ To receive documents in order from a publisher, set to `publisher`. ■ To receive documents in any order, set to `none`.

Remarks

You must establish a connection to the Broker before using this command.

The first form of the command creates the clients for queues bound into JNDI as `lookup-name`. You can specify the lookup name using an absolute or relative path.

You can specify an asterisk (*) for `lookup-name` to create supporting Broker clients for all queue objects in the current JNDI context. When you use an asterisk for `lookup-name`, you *must* be in the context of the queues you want to create. You cannot use an asterisk with a relative or absolute path to a context.

If a queue receiver will access the queue through SSL, you must specify the lookup name of the connection factory that the receiver will use to retrieve messages from the queue. You must also supply the distinguished name that the receiver will supply when it connects to the Broker through this connection factory. In addition, you must specify the distinguished name of the issuer of the certificate.

If you specify the lookup names of multiple queues, the SSL credentials in the specified connection factory will be applied to all queues.

The second form of the command creates the Broker client without using JNDI.

Note:
The `create queue` commands that use the `sslDn` and `issuerDn` parameters of the previous Broker versions will continue to work. The `user` parameter replaced the `sslDn` parameter, and the `authenticator` parameter replaced the `issuerDn` parameter.

Example

The following command sequence connects to the Broker and then navigates to the `TestEnvironment` context. Using the `bind` command, it creates the queue object, `EmployeeQueue`, and then binds it to JNDI under the lookup name `EmpQ`. As a last step, the command sequence uses the `create queue` command to create the queue's supporting client on the Broker.

```
> connect broker "Test JMS Broker" server testserver
 > cd TestEnvironment
 > bind queue EmpQ with queueName=EmployeeQueue
 > create queue for EmpQ
```

Related Commands

"Bind Queue" on page 80

"Delete Queue" on page 101

"Modify Queue" on page 133

Create Topic

Creates the supporting Broker document types for topics.

Syntax

```
create  { topic | topics }  ( topic-name )+
create  { topic | topics }  for { ( lookup-name )+ |  *  }
```

Remarks

You must establish a connection to the Broker before using this command.

The first form creates the document type using the `topic-name` that is specified; it should be a document type name and can include document type scopes.

The second form creates the document types for the topic bound to JNDI by `lookup-name`. If you specify an asterisk (*) for `lookup-name`, all topics in the current context have their corresponding document types created.

Examples

The following command sequence connects to the Broker and creates the topics `Acme::JMS::Topics::OrderReceived`and`Acme::JMS::Topics::OrderInquiry`:

```
> connect broker "Test JMS Broker" server testserver
 > create topic Acme::JMS::Topics::OrderReceived::Topics::T2
 > create topic Acme::JMS::Topics::OrderInquiry
```

Related Commands

"Bind Topic" on page 84

"Delete Topic" on page 102

"Modify Topic" on page 137

Delete Broker

Deletes an existing Broker.

Syntax

```
delete broker brokerName [ on server[:port] ]
```

Remarks

When you delete a Broker, all definitions and messages on that Broker are lost.

If you do not specify the `server` and `port`, JMSAdmin attempts to delete the specified Broker on `localhost:6849`.

If you delete the specified Broker on the default server and port, you may want to set a new default Broker using the `set broker` command.

JMSAdmin will prompt you to confirm the deletion, unless you started JMSAdmin with the quiet switch (`-q`). For more information, see "Start-up Options" on page 47.

Related Commands

"Connect" on page 89

"Create Broker " on page 91

"Disconnect" on page 104

"Initialize Broker " on page 107

"Set Broker " on page 143

Delete Context

Deletes a specified context.

Syntax

```
delete { context | ctx }  contextName [ '('recursive')' ]
```

Remarks

You must switch to the context's parent and specify the name of the context using its relative name, not its absolute path name. If the context is not empty, JMSAdmin will prompt you to confirm the deletion (unless you started JMSAdmin with the (`-q`) quiet switch.

To delete a context that contains subcontexts, you must specify the `recursive` option. When that option is set, JMSAdmin deletes all subcontexts in the specified context.

Example

Delete `ctx1` and all of its subcontexts.

```
\> delete context ctx1 (recursive)
```

Related Commands

"Change Context" on page 88

"Create Context" on page 92

"List Context" on page 108

Delete Durable Subscriber

Deletes a durable subscriber's broker client in the currently connected Broker.

Syntax

```
delete durable subscriber subscriber-name  using tcf-name

delete durable subscriber subscriber-name  clientID client-id
```

Remarks

You must establish a connection to the Broker before using this command.

In the first form of the command, to delete the subscriber's client, you specify the durable subscriber's unique name in the `subscriber-name` parameter. You must also supply the lookup name of the topic connection factory (`tcf-name`) that creates the connection through which the durable subscriber connects.

The second form of the command deletes a durable subscriber without using JNDI.

Related Commands

"Create Durable Subscriber" on page 93

Delete Group

Deletes specified client groups from the Broker.

Syntax

```
delete { group | groups }  ( clientGroupName )+
```

Remarks

Use this command with extreme caution, since this command deletes a client group regardless of whether the group has connection factories or queues currently assigned to it.

The default *client-group-name* is JMSClient.

Related Commands

"Create Group" on page 96

"Deny" on page 103

"Permit" on page 141

Delete Queue

Deletes the Broker client user for a queue.

Syntax

```
delete { queue | queues }  for  { ( lookup-name )+ | * }

delete { queue | queues }  ( queue-name )+
```

Remarks

You must establish a connection to the Broker before using this command.

The first form of the command deletes the client for the queue bound into JNDI as `lookup-name`. If the asterisk (*) is specified as the name, all queues in the current context that have Broker clients are deleted. The Broker client name is the same as the `queueName` property for the queue.

The second form of the command deletes the Broker client without using JNDI.

Related Commands

"Bind Queue" on page 80

"Create Queue" on page 96

"Modify Queue" on page 133

Delete Topic

Deletes from the Broker the supporting Broker document types (also referred to as event types) for a topic.

Syntax

```
delete  { topic | topics }  ( topic-name )+

delete  { topic | topics }  using  { ( lookup-name )+ |  *  }
```

Remarks

You must establish a connection to the Broker before using this command.

The first form deletes the document type(s) specified in `topic-name`; each must be a document type name and can include document type scopes.

The second form deletes the document type for the topic bound to JNDI as `lookup-name`. If you specify an asterisk (*) for `lookup-name`, this command will delete the document types for all topic objects in the current context.

Examples

The following example deletes two specified document types from the Broker:

```
\> delete topic Acme::JMS::Topics::OrdReceived Acme::JMS::OrdShipped
```

The following deletes the document type that corresponds to the topic object bound to the JNDI name `nNewEmp`:

```
\> delete topic using nNewEmp
```

Related Commands

"Bind Topic" on page 84

"Create Topic" on page 98

"Modify Topic" on page 137

Deny

Removes subscribe or publish permissions for one or more topics from groups, and send and receive permissions on queues.

Syntax

```
deny  group group-name  [to]  { subscribe | publish } ( topic-names )+

deny  group group-name [to]  send    [ to ]    ( queue-names )+

deny  group group-name [to]  receive [ from ]  ( queue-names )+
```

Remarks

The second and third forms revoke a queue's send or receive permissions for a specified client group by removing the document type associated with the queue from the client's can-publish or can-subscribe lists.

Note: `topic-name` specifies the topic's actual name, not its JNDI lookup name. Queue names are also the Broker names, not the JNDI names.

Examples

```
> connect broker "Test JMS Broker" server testserver
> deny group Staff to publish Test::Topics::T1 Test::Topics::T3
> deny group Managers to subscribe Test::Topics::T2
> deny group Auditors to subscribe Test::Topics::T2
> deny group Staff to send to RequestQueue
> deny group Managers to receive from RequestQueue
```

Related Commands

"Permit" on page 141

Disconnect

Ends a Broker connection.

Syntax

```
{ disconnect | disconn }
```

Related Commands

"Connect" on page 89

"Create Broker " on page 91

"Delete Broker " on page 99

"Initialize Broker " on page 107

Display

Displays information about a specified Broker object.

Syntax

```
display brokers [ [ on ] host[:port] ]
display groups
display group group-name
display queues
display queue queue-name
display topics
display topic topic-name
display durable subscribers
display durable subscriber subscriber-name
display subscribers
display subscriber subscriber-name
```

Remarks

The `display brokers` command displays all the Brokers that exist on the specified server, or localhost, if the `host` is not specified.

The `display groups`, `display queues`, `display topics`, `display durable subscribers`, and `display subscribers` commands display a list of the corresponding objects on the Broker to which you are connected. Use the singular form of each command to show more detailed information about the corresponding object.

End

Ends a JMSAdmin session and exits the application.

Syntax

```
{ end | quit }
```

You can use the `quit` command interchangeably with `end`.

Export

Creates a script file of JMSAdmin `bind` commands that will recreate the entries bound into JNDI.

Syntax

```
export { ( * | lookup-name )+ }  to  { filename | stdout }
  [ ( { append | overwrite } ) ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up the entry in the JNDI directory.
filename or stdout	*Required.* The name of the command file that will be created, or whether the `bind` statements will be directed to the console.
append or overwrite	*Optional.* The `append` parameter to add the results of the command to the end of the named file, if it exists. If the specified file does not exist, a new file will be created. If you use the `overwrite` option and the specified file currently exists, the existing data in the file will be replaced with new data. If the file exists and you do not specify `overwrite`, JMSAdmin will issue an error.

Remarks

You can export all entries within a context or individual entries.

The export file contains a `bind` statement for each webMethods administered object in the specified context (only webMethods administered objects are exported). You can export to a file or to `stdout`. Exporting to `stdout` is most useful when it is redirected at JMSAdmin startup.

Only JNDI `bind` commands can be exported to a script file. If the commands are exported to a file rather than `stdout`, you can edit the script file (for example, to add commands) before executing the file, if desired.

When you specify `filename` in Windows notation, which uses the backslash character (\) as the directory separator, you must enclose the entire pathname in quotes.

Examples

The following example creates `bind` statements for all connection factories, queues, and topics in the context `/Production/HR` and saves them in the file, `HR_Prod_Def.txt`:

```
\> export /Production/HR to myData/exports/HR_Prod_Def.txt    (UNIX)
\> export /Production/HR to "myData\exports\HR_Prod_Def.txt"  (Windows)
```

The first statement below switches to the `/Test/Manufacturing` context, and the second statement exports all webMethods Broker objects in that context and stores them in the file `TestEnvironment.txt`:

```
\> cd /Test/Manufacturing
\> export * to TestEnvironment.txt
```

The following example exports two objects (`NewMachineTopic` and `OldMachineTopic`) in the `Test` context and appends them to `TestEnvironment.txt`:

```
> export /Test/NewMachineTopic /Test/OldMachineTopic to
    TestEnvironment.txt (append)
```

Related Commands

"Import" on page 106

Help

Displays help text for JMSAdmin.

Syntax

```
help [command]
```

Remarks

To display help text for a specific command, use the command name as a parameter. To display help for all JMSAdmin commands, do not specify a parameter.

Import

Executes JMSAdmin commands contained in a file.

Syntax

```
import from  filename [ with  encoding={ charset-name | default } ]
```

Remarks

Use the `import` command to execute a script file or batch file of JMSAdmin commands. The `import` command can work with any JMSAdmin commands.

The `filename` must be specified in the syntax of the operating system in which JMSAdmin is running. You can nest `import` commands arbitrarily deep (depth is limited only by available memory).

When specifying `filename` using Windows-notation (that is, using a backslash character (\) as the directory separator), enclose the entire path in quotes as shown in the first example below.

The `import` command works the same as running JMSAdmin with the `-f` switch. For more information, see "Start-up Options" on page 47.

The `import` command is often used with the `export` command, which creates a script file; however, `export` only produces JMSAdmin `bind` commands, whereas `import` can execute a file of any JMSAdmin commands.

Examples

```
\> import from  "c:\JMS Definitions\TestSystem.dat"
\> import from /jmsDefinitions/TestSystem
```

Related Commands

"Export" on page 105

Initialize Broker

Creates the default JMS definitions on the currently connected Broker.

Syntax

```
initialize broker
```

Remarks

This command creates the following structures on the Broker:

```
JMS::Temporary::Topic
(document type)JMS::Temporary::Queue
(document type)JMSClient
(client group)
```

Can-publish and can-subscribe permissions are also added to the JMSClient group for the two document types created.

Related Commands

"Connect" on page 89

"Create Broker " on page 91

"Delete Broker " on page 99

"Disconnect" on page 104

List Context

Displays the contents of the current context or a specified context.

Syntax

```
{ list | ls }   [ context | ctx ]  [ context-name ]
```

Remarks

Each line in the resulting list displays an entry name and the type of object to which the name is bound, for example:

```
[1]: MyCF ==> com.webmethods.jms.impl.WmConnectionFactoryImpl
[2]: Q1   ==> com.webmethods.jms.impl.WmQueueImpl
[3]: T1   ==> com.webmethods.jms.impl.WmTopicImpl
[4]: T2   ==> com.webmethods.jms.impl.WmTopicImpl
[5]: T3   ==> com.webmethods.jms.impl.WmTopicImpl
[6]: sub6 ==> com.webmethods.jmsadmin.jndi.MemoryNameServiceCtx
```

You can specify `context-name` using a relative or absolute path.

Examples

Any of the following commands would list the entries in the current context:

```
\> list
\> list context
\> ls
\> ls ctx>
\> list .
```

The following command lists the entries in the context whose absolute path is `/sub2`:

```
\> ls /sub2
```

Related Commands

"Change Context" on page 88

"Create Context" on page 92

"Delete Context" on page 100

Modify ClusterConnectionFactory

Changes property values for an existing cluster connection factory definition.

Syntax

```
modify  { cl_cf | ClusterConnectionFactory } lookup-name [ with

  clusterName=cluster-name clusterBrokers=(broker-name@broker-server:port)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ weights=(weight-per-broker)+ ]
  [ multiSendCount=broker-count-for-multisend-policy ]
  [ includeAllBrokers={ true | false } ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
`clusterName`	*Required.* The name of the cluster containing the Brokers. If `includeAllBrokers` is set to `true`, the name of the cluster and the first Broker in the cluster is used to identify the remaining Brokers in the cluster. When `includeAllBrokers` is set to `true`, set the value of `clusterName` correctly to retrieve all the other Brokers in the cluster.
`clusterBrokers`	*Required.*List of Brokers in the cluster that will be part of this connection factory. A cluster Broker is specified in the form of *broker-name@broker-server:port*. The list of Brokers in the cluster is specified by separating the cluster Brokers with a comma (,). For example, set the value of `clusterBrokers` as following: `clusterBrokers=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>` or use variables to specify the list of Brokers:

Part	Description
	`set $brkr_set=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>clusterBrokers=$brkr_set`
`clusterRefreshInterval`	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the cluster routing policy. Specify any of the following policies: - `ROUND_ROBIN` (the default value) - `STICKY` - `RANDOM` - `WEIGHTED_ROUND_ROBIN` - `MULTISEND_GUARANTEED` - `MULTISEND_BEST_EFFORT`
`weights`	*Optional.* A comma separated list of weights of each Broker in the cluster governed by `WEIGHTED_ROUND_ROBIN` policy. Specify the weight of each Broker in the same order as specified in the `clusterBrokers` property. The weights must be greater than `0`. For example, `weights=10,20.` If you do not specify a weight for a Broker, a weight of `1` is set for that Broker. **Note:** Set this property only when `clusterPolicy` property value is set to `WEIGHTED_ROUND_ROBIN` policy.
`multiSendCount`	*Optional.* The number of Brokers in the cluster governed by `MULTISEND_GUARANTEED` cluster policy. The `multiSendCount` property value must be greater than 0 and less than the number of Brokers in the cluster. **Note:** Set this property only when `clusterPolicy` property value is set to `MULTISEND_GUARANTEED` policy.

Part	Description
`includeAllBrokers`	*Optional.* Whether to include all the Brokers in the cluster with this connection factory. ■ If set to `true`, all the Brokers in the cluster are included in this connection factory. The value of `clusterName` property and the first Broker in the cluster is used to retrieve all the other Brokers in the cluster. ■ If set to `false`, only the specified Brokers are included in this connection factory. The default value is `false`.
`clientID`	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.

Part	Description
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

If you are using LDAP as JNDI provider, the changes made in a child cluster connection factory is not refreshed in the composite cluster connection factory. If you modify a child cluster connection factory, rebind the composite cluster connection factory.

You can only use the marshalInClassName and marshalOutClassName parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
show TEST_CONNECTION_FACTORY
TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  clusterRefreshInterval: 6400
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
modify ClusterConnectionFactory TEST_CONNECTION_FACTORY with
  clusterPolicy=STICKY
  clusterRefreshInterval=900
  useXA=true
show TEST_CONNECTION_FACTORY
TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmXAConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  clusterRefreshInterval: 900
  clusterPolicy: STICKY
```

```
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify ClusterConnectionFactory" on page 109

Modify ClusterQueueConnectionFactory

Changes property values for an existing cluster queue connection factory definition.

Syntax

```
modify  { cl_qcf | ClusterQueueConnectionFactory } lookup-name  [ with

  clusterName=cluster-name clusterBrokers=(broker-name@broker-server:port)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ weights=(weight-per-broker)+ ]
  [ multiSendCount=broker-count-for-multisend-policy ]
  [ includeAllBrokers={ true | false } ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterName	*Required.* The name of the cluster containing the Brokers. If includeAllBrokers is set to true, the name of the cluster and the first Broker in the cluster is used to identify the remaining Brokers in the cluster.

Part	Description
	When `includeAllBrokers` is set to `true`, set the value of `clusterName` correctly to retrieve all the other Brokers in the cluster.

`clusterBrokers`	*Required.*List of Brokers in the cluster that will be part of this connection factory. A cluster Broker is specified in the form of *broker-name@broker-server:port*. The list of Brokers in the cluster is specified by separating the cluster Brokers with a comma (,).
	
	For example, set the value of `clusterBrokers` as following:
	`clusterBrokers=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>`
	
	or use variables to specify the list of Brokers:
	`set $brkr_set=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>clusterBrokers=$brkr_set`

`clusterRefreshInterval`	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `WEIGHTED_ROUND_ROBIN` ■ `MULTISEND_GUARANTEED` ■ `MULTISEND_BEST FFORT` **Note:**`MULTISEND_GUARANTEED` cluster policy is applicable only to XA connection factories. `MULTISEND_BEST_EFFORT` policy does not work with XA connection factory.
`weights`	*Optional.* A comma separated list of weights of each Broker in the cluster governed by `WEIGHTED_ROUND_ROBIN` policy. Specify the weight of each Broker in the same order as specified in the `clusterBrokers` property. The weights must be greater than `0`. For example, `weights=10,20.`

Part	Description
	If you do not specify a weight for a Broker, a weight of `1` is set for that Broker. **Note:** Set this property only when `clusterPolicy` property value is set to `WEIGHTED_ROUND_ROBIN` policy.
`multiSendCount`	*Optional.* The number of Brokers in the cluster governed by `MULTISEND_GUARANTEED` cluster policy. The `multiSendCount` property value must be greater than 0 and less than the number of Brokers in the cluster. **Note:** When you set the value of `multiSendCount` property, ensure that you have set the value of `clusterPolicy` property to`MULTISEND_GUARANTEED` policy.
`includeAllBrokers`	*Optional.* Whether to include all the Brokers in the cluster with this connection factory. ■ If set to `true`, all the Brokers in the cluster are included in this connection factory. The value of `clusterName` and the first Broker in the cluster is used to retrieve all the other Brokers in the cluster. ■ If set to `false`, only the specified Brokers are included in this connection factory. The default value is `false`.
`clientID`	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.

Part	Description
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this cluster queue connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified cluster queue connection factory with an application server, specifying `false` saves some of the overhead associated with support for cluster XA queue connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmXAQueueConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 640
```

```
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>

modify ClusterQueueConnectionFactory TEST_CONNECTION_FACTORY with
  clusterPolicy=STICKY
  clusterRefreshInterval=90
  useXA=false

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmQueueConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 90
  clusterPolicy: STICKY
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Bind ClusterQueueConnectionFactory" on page 63

Modify ClusterTopicConnectionFactory

Changes property values for an existing cluster topic connection factory definition.

Syntax

```
modify  { cl_tcf | ClusterTopicConnectionFactory } lookup-name  [ with

  clusterName=cluster-name
  clusterBrokers=(broker-name@broker-server:port)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ weights=(weight-per-broker)+ ]
  [ multiSendCount=broker-count-for-multisend-policy ]
```

```
  [ includeAllBrokers={ true | false } ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterName	*Required.* The name of the cluster containing the Brokers. During run time, if includeAllBrokers is set to true, the name of the cluster and the first Broker in the cluster is used to identify the remaining Brokers in the cluster. When includeAllBrokers is set to true, set the value of clusterName correctly to retrieve all the other Brokers in the cluster.

clusterBrokers	*Required.*List of Brokers in the cluster that will be part of this connection factory. A cluster Broker is specified in the form of *broker-name@broker-server:port*. The list of Brokers in the cluster is specified by separating the cluster Brokers with a comma (,).
	
	For example, set the value of clusterBrokers as following:
	`clusterBrokers=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>`
	
	or use variables to specify the list of Brokers:
	`set $brkr_set=broker_01@<hostname>:<port>, broker_02@<hostname>:<port>clusterBrokers=$brkr_set`

clusterRefreshInterval	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is 1440.
clusterPolicy	*Optional.*The name of the cluster routing policy. Specify any of the following policies: - ROUND_ROBIN (the default value)

Part	Description
	■ STICKY ■ RANDOM ■ WEIGHTED_ROUND_ROBIN ■ MULTISEND_GUARANTEED ■ MULTISEND_BEST_EFFORT **Note:** MULTISEND_GUARANTEED cluster policy is applicable only to XA connection factories. MULTISEND_BEST_EFFORT policy does not work with XA connection factory.
weights	*Optional.* A comma separated list of weights of each Broker in the cluster governed by WEIGHTED_ROUND_ROBIN policy. Specify the weight of each Broker in the same order as specified in the clusterBrokers property. The weights must be greater than 0. For example, weights=10,20. If you do not specify a weight for a Broker, a weight of 1 is set for that Broker. **Note:** Set this property only when clusterPolicy property value is set to WEIGHTED_ROUND_ROBIN policy.
multiSendCount	*Optional.* The number of Brokers in the cluster governed by MULTISEND_GUARANTEED cluster policy. The multiSendCount property value must be greater than 0 and less than the number of Brokers in the cluster. **Note:** Set this property only when clusterPolicy property value is set to MULTISEND_GUARANTEED policy.
includeAllBrokers	*Optional.* Whether all the Brokers in the cluster are included in this connection factory. ■ If set to true, all the Brokers in the cluster are included in this connection factory. The value of clusterName property and the first Broker in the cluster is used to retrieve all the other Brokers in the cluster. ■ If set to false, all the Brokers are not included in this connection factory. The default value is false.

Part	Description
`clientID`	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
`application`	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
`group`	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this cluster topic connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified cluster topic connection factory with an application server, specifying `false` saves some of the overhead associated with support for cluster XA topic connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).

Part	Description
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmTopicConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 55
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslLEncrypted: false
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>

modify cl_tcf TEST_CONNECTION_FACTORY with
  clusterPolicy=STICKY
  clusterRefreshInterval=66
  useXA=true

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmXATopicConnectionFactoryImpl
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 66
  clusterPolicy: STICKY
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Bind CompositeQueueConnectionFactory" on page 73

Modify CompositeConnectionFactory

Changes property values for an existing composite cluster connection factory definition.

Syntax

```
modify  { cm_cf | CompositeConnectionFactory } lookup-name  [ with

  clusterConnectionFactories=(connection-factory-name)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterConnectionFactories	*Required.*List of connection factory names that are part of the composite connection factory. The list of connection factory names is specified by separating the connection factory names with a comma (,).
clusterRefreshInterval	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is 1440.
clusterPolicy	*Optional.*The name of the composite cluster routing policy. Specify any of the following policies: ■ ROUND_ROBIN (the default value) ■ STICKY ■ RANDOM ■ MULTISEND_BEST_EFFORT

Part	Description
	■ MULTISEND_GUARANTEED
useXA	*Optional.* Whether XA transactions are allowed with this composite connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified composite connection factory with an application server, specifying `false` saves some of the overhead associated with support for composite XA connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
marshalInClassName	*Optional.* Class name of the inbound marshaller (JMS clients only).
marshalOutClassName	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

If you are using LDAP as JNDI provider, the changes made in a child cluster connection factory is not refreshed in the composite cluster connection factory. If you modify a child cluster connection factory, rebind the composite cluster connection factory.

You can only use the marshalInClassName and marshalOutClassName parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_03,clusterFactory_04
  clusterRefreshInterval: 640
  clusterPolicy: STICKY
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factories: 2
cluster Factory [0]: clusterFactory_03
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
```

```
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factory [1]: clusterFactory_04
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslLKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>

modify cm_cf TEST_CONNECTION_FACTORY with
  clusterConnectionFactories=clusterFactory_01,clusterFactory_02
  clusterPolicy=RANDOM
  clusterRefreshInterval=240

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_01,clusterFactory_02
  clusterRefreshInterval: 240
  clusterPolicy: RANDOM
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factories: 2
cluster Factory [0]: clusterFactory_01
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factory [1]: clusterFactory_02
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
```

```
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Modify ClusterConnectionFactory" on page 109

Modify CompositeQueueConnectionFactory

Changes property values for an existing composite queue connection factory definition.

Syntax

```
modify  { cm_qcf | CompositeQueueConnectionFactory } lookup-name  [ with

  clusterConnectionFactories=(cluster-queue-connection-factory-name)+
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterConnectionFactories	*Required.*List of cluster queue connection factory names that will be part of the composite queue connection factory. The list of cluster queue connection factory names is specified by separating the cluster queue connection factory names with a comma (,).
clusterRefreshInterval	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker

Part	Description
	connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the composite queue cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `MULTISEND_BEST_EFFORT`
`useXA`	*Optional.* Whether XA transactions are allowed with this composite queue connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified composite queue connection factory with an application server, specifying `false` saves some of the overhead associated with support for composite XA queue connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmQueueConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_01,clusterFactory_02
  clusterRefreshInterval: 640
  clusterPolicy: ROUND_ROBIN
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factories: 2
```

```
cluster Factory [0]: clusterFactory_01
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factory [1]: clusterFactory_02
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>

modify CompositeQueueConnectionFactory TEST_CONNECTION_FACTORY with
  clusterConnectionFactories=clusterFactory_03,clusterFactory_04
  clusterPolicy=STICKY

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY:com.webmethods.jms.impl.WmQueueConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_03,clusterFactory_04
  clusterRefreshInterval: 640
  clusterPolicy: STICKY
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factories: 2
cluster Factory [0]: clusterFactory_03
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
```

```
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factory [1]: clusterFactory_04
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Bind CompositeQueueConnectionFactory" on page 73

Modify CompositeTopicConnectionFactory

Changes property values for an existing composite topic connection factory definition.

Syntax

```
modify  { cm_tcf | CompositeTopicConnectionFactory } lookup-name  [ with

  clusterConnectionFactories=(cluster-topic-connection-factory-name +
  [ clusterRefreshInterval=cluster-refresh-interval ]
  [ clusterPolicy=cluster-policy-name ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
clusterConnectionFactories	*Required.*List of cluster topic connection factory names that will be part of the composite topic connection factory. The list of cluster topic connection factory names is specified by

Part	Description
	separating the cluster topic connection factory names with a comma (,).
`clusterRefreshInterval`	*Optional.*The cluster refresh interval in minutes. JMS client applications can use this interval to refresh the cluster connection factory, and choose to reconnect if the connection factory has changed. The webMethods API for JMS will not automatically refresh the connection factories or Broker connections. The value of this property must be an integer greater than 0. The default value is `1440`.
`clusterPolicy`	*Optional.*The name of the composite topic cluster routing policy. Specify any of the following policies: ■ `ROUND_ROBIN` (the default value) ■ `STICKY` ■ `RANDOM` ■ `MULTISEND_BEST_EFFORT` ■ `MULTISEND_GUARANTEED`
`useXA`	*Optional.* Whether XA transactions are allowed with this composite topic connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified composite topic connection factory with an application server, specifying `false` saves some of the overhead associated with support for composite XA topic connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Remarks

You can only use the `marshalInClassName` and `marshalOutClassName` parameters when implementing the marshalling feature when you use webMethods Broker as a JMS provider. For more information, see "JMS Marshalling" on page 223.

Example

```
show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmTopicConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_03,clusterFactory_04
  clusterRefreshInterval: 640
  clusterPolicy: STICKY
  useXA: false
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factories: 2
cluster Factory [0]: clusterFactory_03
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factory [1]: clusterFactory_04
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>

modify cm_tcf TEST_CONNECTION_FACTORY with
  clusterConnectionFactories=clusterFactory_01,clusterFactory_02
  clusterPolicy=RANDOM
  clusterRefreshInterval=240

show TEST_CONNECTION_FACTORY

TEST_CONNECTION_FACTORY : com.webmethods.jms.impl.WmTopicConnectionFactoryImpl
  clusterConnectionFactories: clusterFactory_01,clusterFactory_02
  clusterRefreshInterval: 240
  clusterPolicy: RANDOM
  useXA: false
```

```
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factories: 2
cluster Factory [0]: clusterFactory_01
  clusterName: cluster_01
  clusterBrokers: broker_01@hostname:port,broker_02@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
cluster Factory [1]: clusterFactory_02
  clusterName: cluster_01
  clusterBrokers: broker_03@hostname:port,broker_04@hostname:port
  includeAllBrokers: false
  clusterRefreshInterval: 144
  clusterPolicy: ROUND_ROBIN
  application: JMS
  clientGroup: JMSClient
  clientId: <not specified>
  sslKeystore: <not specified>
  sslTruststore: <not specified>
  sslKeystoreType: <not specified>
  sslTruststoreType: <not specified>
  sslEncrypted: false
  useXA: true
  marshalInClassName: <not specified>
  marshalOutClassName: <not specified>
```

Related Commands

"Bind CompositeTopicConnectionFactory" on page 76

Modify ConnectionFactory

Changes property values for an existing connection factory definition.

Syntax

```
modify  { cf | ConnectionFactory } lookup-name with
  [ { brokerHost | bh }=host[:port] ]
  [ { brokerName | bn }=broker-name ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
```

```
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
brokerHost	■ *Optional.*The host part is the host name of the Broker Server on which the Broker resides. The default value is localhost. ■ *Optional.*The port part is the port number of the Broker Server on which the Broker resides. The default value is 6849.
brokerName	*Optional.* The name of the Broker. If you do not specify a Broker name, JMSAdmin uses the default Broker on the specified Broker Server.
clientID	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
application	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
group	*Optional.* The client group name that clients created from this connection factory will use. Note that the bind command does not create the client group; you must create a client group explicitly using the create group command. The default *client-group-name* is JMSClient.
sslKeystore	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
sslPassword	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
sslTruststore	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
sslEncrypted	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. The default value is false. ■ If set to true, all traffic to the Broker on this connection will be encrypted.

Part	Description
	■ If set to `false`, the connection will be authenticated but no data will be encrypted.
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories. This parameter is for JMS only. If the connection factory is for a C# client, this parameter is ignored.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller (JMS clients only).
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller (JMS clients only).

Example

In this example, an existing connection factory `MyTCF` has its `BrokerName` property setting changed from the default Broker (`localhost`) to `JMS Broker`, and its `BrokerHost` property changed from `localhost` to `testserver:8849`.

```
/sub4/sub5> show MyCF
MyTCF: com.webmethods.jms.impl.WmConnectionFactoryImpl
   BrokerHost: localhost
   BrokerName: <default broker>
   Application: <not specified>
   ClientGroup: BrokerJMS_Connections
   ClientID: <not specified>
   SSLKeystore: <not specified>
   SSLEncrypted: false
   UseXA: false
/sub4/sub5> modify cf MyCF with brokerName="JMS Broker" bh=testserver:8849
/sub4/sub5> show MyCF
MyTCF: com.webmethods.jms.impl.WmConnectionFactoryImpl
   BrokerHost: testserver:8849
   BrokerName: JMS Broker
   Application: <not specified>
   ClientGroup: BrokerJMS_Connections
   ClientID: <not specified>
   SSLKeystore: <not specified>
   SSLEncrypted: false
   UseXA: false
```

Related Commands

"Bind ConnectionFactory" on page 78

Modify Queue

Changes the properties of an existing queue definition.

Syntax

```
modify queue lookup-name with  { queueName | qn }=broker-queue-name
  [ priorityOrdering={ true | false } ]
  [ { sharedState | ss }={ true | false } ]
  [ { sharedStateOrdering | sso }={ none | publisher } ]
```

The command syntax has these parts:

Part	Description
`queue`	*Required.* The name of the JNDI directory in which to bind this queue.
`queueName`	*Required.* The name to give to the queue, which must meet the following criteria: ■ The first character cannot be a pound sign (#). ■ The names cannot contain the at sign (@), forward slash (/), or colon (:). ■ The names cannot exceed 255 bytes.
`priorityOrdering`	*Optional.*Whether or not the queue orders messages by priority. The default value is `true`.
`sharedState`	*Optional.* Whether to allow multiple queue receivers to concurrently read from the queue. The default value is `false`.
`sharedStateOrdering`	*Optional.* The ordering of documents received on a shared state client. The default value is `none`. ■ To receive documents in order from a publisher, set to `publisher`. ■ To receive documents in any order, set to `none`.

Example

Change the value of the `sharedState` property for the queue bound to NetAdmin:

```
\> modify queue nNetAdmin with qn=NetAdmin ss=true sso=none
```

Related Commands

"Create Queue" on page 96

"Delete Queue" on page 101

Modify QueueConnectionFactory

Changes property values for an existing `QueueConnectionFactory` definition (JMS clients only).

Syntax

```
modify { qcf | QueueConnectionFactory } lookup-name  with

  [ { brokerHost | bh }=host[:port] ]
  [ { brokerName | bn }=broker-name ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
brokerHost	■ *Optional.*The `host` part is the host name of the Broker Server on which the Broker resides. The default value is `localhost`. ■ *Optional.*The `port` part is the port number of the Broker Server on which the Broker resides. The default value is `6849`.
brokerName	*Optional.* The name of the Broker. If you do not specify a Broker name, JMSAdmin uses the default Broker on the specified Broker Server.
clientID	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
application	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
group	*Optional.* The client group name that clients created from this connection factory will use. Note that the `bind` command does not create the client group; you must create a client group explicitly using the `create group` command. The default *client-group-name* is JMSClient.
sslKeystore	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.

Part	Description
`sslKeystore`	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
`sslPassword`	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
`sslTruststore`	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
`sslEncrypted`	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to `true`, all traffic to the Broker on this connection will be encrypted. ■ If set to `false`, the connection will be authenticated but no data will be encrypted. The default value is `false`.
`useXA`	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is `true`, which means XA transactions are allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller.
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller.

Example

The following command changes the values of the `brokerName` and `brokerHost` properties for the QueueConnectionFactory `qcf`:

```
\> modify qcf MyQCF with bn"JMS Broker" bh=testserver:8849
```

Related Commands

"Bind QueueConnectionFactory" on page 81

Modify String

Changes the value of a Java string that has been bound into a JNDI context (JMS clients only).

Syntax

```
modify string lookup-name newStringValue
```

Related Commands

"Bind String" on page 83

Modify Topic

Changes the values of properties in an existing topic definition.

Syntax

```
modify topic lookup-name  with { topicName | tn }=event-type-name

  [ deadLetterOnly={ true | false } ]
  [ localOnly={ true | false } ]
  [ priorityOrdering={ true | false } ]
  [ { sharedState | ss }={ true | false } ]
  [ { sharedStateOrdering | sso }={ none | publisher } ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name of the JNDI directory to which the topic is bound.
topicName	*Required.* The name of the topic (destination) through which publishers and subscribers will exchange messages. The name must meet the following criteria: ■ The first character cannot be a pound sign (#). ■ The name cannot contain the at sign (@), forward slash (/), or colon (:). ■ Each name segment cannot exceed 255 bytes.
deadLetterOnly	*Optional.*If you set this option to `true`, only dead letter subscriptions will be received. The default value is `false`.
localOnly	*Optional.* If you set this option to `true`, only messages originating (published) from the local machine will be received. The default value is `false`.
priorityOrdering	*Optional.*Whether or not the subscribers' queues order messages by priority. The default value is `true`.
sharedState	*Optional.* Whether to allow multiple connections to share the same Broker client for the subscription. The default value is `false`.

Part	Description
	If `sharedState` is set to `true`, webMethods Broker used as a JMS provider creates Broker clients as shared state clients for the connection.
`sharedStateOrdering`	*Optional.* The ordering of documents received on a shared state client. The default value is `none`. ■ To receive documents in order from a publisher, set to `publisher`. ■ To receive documents in any order, set to `none`.

Example

The following changes the name of the topic that is bound to the name `nNewEmp`:

```
\> modify topic nNewEmp with topicName=Acme::JMS::Topics::HR::NewEmp
```

Related Commands

"Bind Topic" on page 84

"Create Topic" on page 98

"Delete Topic" on page 102

Modify TopicConnectionFactory

Changes property values for an existing topic connection factory that is already bound to JNDI (JMS clients only).

Syntax

```
modify  { tcf | TopicConnectionFactory } lookup-name with

  [ { brokerHost | bh }=host[:port] ]
  [ { brokerName | bn }=broker-name ]
  [ clientId=client-id ]
  [ application=application-name ]
  [ group=client-group-name ]
  [ sslKeystore=filename ]
  [ sslPassword=password ]
  [ sslTruststore=filename ]
  [ sslEncrypted={ true | false } ]
  [ useXA={ true | false } ]
  [ marshallInClassName=class-name ]
  [ marshallOutClassName=class-name ] ]
```

The command syntax has these parts:

Part	Description
lookup-name	*Required.* The name used to look up this connection factory in the JNDI directory.
brokerHost	■ *Optional.*The host part is the host name of the Broker Server on which the Broker resides. The default value is localhost. ■ *Optional.*The port part is the port number of the Broker Server on which the Broker resides. The default value is 6849.
brokerName	*Optional.* The name of the Broker. If you do not specify a Broker name, JMSAdmin uses the default Broker on the specified Broker Server.
clientID	*Optional.* The base client ID for connections created by this factory. ■ If you specify a client ID, it must be unique. This ID is assigned to all the connections to the Brokers in the cluster. ■ If you do not specify a client ID, the connection factory assigns a unique ID when a connection is established.
application	*Optional.* The name of the application stored as part of the application's Broker client state. The application is displayed by JMSAdmin.
group	*Optional.* The client group name that clients created from this connection factory will use. Note that the bind command does not create the client group; you must create a client group explicitly using the create group command. The default *client-group-name* is JMSClient.
sslKeystore	*Optional.* The fully qualified path of the file containing the SSL certificate that JMSAdmin will use on SSL connections made from this connection factory.
sslPassword	*Optional.* The password for the keystore file that JMSAdmin is to use when connecting to the Broker using SSL.
sslTruststore	*Optional.* The file that holds the trusted root (CA public key) that JMSAdmin will use to connect to an SSL-enabled Broker.
sslEncrypted	*Optional.* Whether traffic to the Broker on connections made from this connection factory will be encrypted. ■ If set to true, all traffic to the Broker on this connection will be encrypted. ■ If set to false, the connection will be authenticated but no data will be encrypted. The default value is false.
useXA	*Optional.* Whether XA transactions are allowed with this connection factory. The default value is true, which means XA transactions are

Part	Description
	allowed. If you are not using the specified connection factory with an application server, specifying `false` saves some of the overhead associated with support for XA connection factories.
`marshalInClassName`	*Optional.* Class name of the inbound marshaller.
`marshalOutClassName`	*Optional.* Class name of the outbound marshaller.

Example

In this example, an existing topic connection factory named `MyTCF` has its `BrokerName` property setting changed from the default (null) to `JMS` Broker and its `BrokerHost` property changed from `localhost` to `testserver`.

```
/sub4/sub5> show MyTCF
MyTCF : com.webmethods.jms.impl.WmXATopicConnectionFactoryImpl
   BrokerHost: localhost
   BrokerName: <default broker>
   Application: <not specified>
   ClientGroup: BrokerJMS_Connections
   ClientID: <not specified>
   SSLKeystore: <not specified>
   SSLEncrypted: false
   UseXA: false
/sub4/sub5> modify tcf MyTCF with brokerName="JMS Broker" bh=testserver:8849
/sub4/sub5> show MyTCF
MyTCF : com.webmethods.jms.impl.WmXATopicConnectionFactoryImpl
   BrokerHost: testserver:8849
   BrokerName: JMS Broker
   Application: <not specified>
   ClientGroup: BrokerJMS_Connections
   ClientID: <not specified>
   SSLKeystore: <not specified>
   SSLEncrypted: false
   UseXA: false
```

Related Commands

"Bind TopicConnectionFactory" on page 86

"Modify Topic" on page 137

Move

Moves an administered object from one context to another.

Syntax

```
{ move | mv } { lookup-name | * } to  context-name
```

Remarks

When JMSAdmin executes a `move` command, it copies the source object to the specified context and then unbinds and deletes it from the original context.

You can specify `lookup-name` (the name of the source object) and `context-name` (the name of the target context) using relative or absolute paths. The target context must already exist. If an object with the name as the source object already exists in `context-name`, JMSAdmin will issue an error message.

You can use the wildcard character (*) as a way to move all webMethods defined objects or strings bound in a context to another context.

Related Commands

"Copy" on page 90

Permit

Grants publish and subscribe permission for one or more topics to groups, or send and receive permissions for one or more queues to groups.

Syntax

```
permit group groupName [to] { subscribe | publish } (topic-names)+
permit group groupName [to] send [ to ]  (queue-names)+
permit group groupName [to] receive [ from ]  (queue-names)+
```

Remarks

The `topic-names` specify the actual topic names, *not* the JNDI lookup names.

For information about client group can-publish and can-subscribe lists, see *Administering webMethods Broker*.

Examples

The following command sequence connects to the Broker, creates three topics (document types) on the Broker, and then does the following:

- Grants publish permission for `Test::Topics::T1` and `Test::Topics::T3` to the client group `Publishers`.
- Grants subscribe permissions for `Test::Topics::T2` to the client group `Subscribers`.
- Explicitly grants publish permission for `Test::Topics::T2` to the `Auditors` client group.

```
> connect broker "Test JMS Broker" server testserver
> create topics Test::Topics::T1 Test::Topics::T2 Test::Topics::T3
```

```
> permit group Publishers to publish Test::Topics::T1::Topics::T3
> permit group Subscribers to subscribe Test::Topics::T2
> permit group Auditors to subscribe Test::Topics::T2
```

Related Commands

"Deny" on page 103

Set

Creates, sets, and displays variables.

Syntax

```
set  [ { $variableName | %variableName } [ =value ] ]
```

Remarks

You can use variables to represent values for properties in commands. You can also use this command to display the values of all variables.

Names of variables must begin with a dollar sign ($) or percent sign (%). Variable names are not case-sensitive. If a value contains a space, enclose it within a pair of single- or double-quote marks.

You can use `set` to view the values of the system variables `$_Broker` and `$_Server`. You can also use it to modify SSL system variables.

Variable names that begin with "$_" are reserved for read-only system variables. JMSAdmin will not permit you to create a variable whose name starts with this character combination.

Examples

The following commands set the variable `$bHost` to `testserver:8849` and the variable `$bName` to `Broker01_VA`:

```
\> set $bHost = testserver:8849
\> set $bName = Broker01_VA
```

The following command displays the value of the variable `$bName`:

```
\> set $bName
```

The following command lists all current variable settings:

```
\> set
```

The following commands use variables as parameter values for the `connect` and `bind` commands:

```
\> connect broker $bName server $bHost
\> bind TopicConnectionFactory Tcf1 with brokerHost=$bHost  brokerName=$bName
```

Using System Variables

You can use the `set` command with the following system variables:

System Variable	Description
`$_Broker`	*Read-only.* Contains the name of the Broker to which JMSAdmin is currently connected. This variable will contain an empty string if JMSAdmin is not connected to a Broker. For more information, see the "Connect" on page 89 and "Disconnect" on page 104 commands.
`$_Server`	*Read-only.* Contains the name of the Broker Server to which JMSAdmin is currently connected. This variable will contain an empty string if JMSAdmin is not connected to a Broker Server. For more information, see the "Connect" on page 89 and "Disconnect" on page 104 commands.
`$sslKeystore`	Specifies the filename containing the SSL certificate (keystore) that JMSAdmin will use when connecting to the Broker using SSL.
`$sslPassword`	Specifies the password JMSAdmin will use when connecting to the Broker using SSL.
`$sslEncrypted`	Indicates whether JMSAdmin will use encryption when connected to the Broker using SSL. Set the variable to `true` to enable encryption; set to `false` otherwise.
`$sslTruststore`	Specifies the file containing the trusted root (CA public key) to use when making SSL connections to the Broker.

Related Commands

"Display" on page 104

Set Broker

Changes the specified Broker to be the default Broker on the connected server.

Syntax

```
set broker brokerName  [ [ on ] server-name  ] [ as ] default
```

Related Commands

"Connect" on page 89

"Create Broker " on page 91

"Disconnect" on page 104

"Initialize Broker " on page 107

Set Encoding

Specifies the name of a character set encoding.

Syntax

```
set encoding  [ ={ charset-name | default } ]
```

Remarks

Specifies the name of a character set encoding for the machine on which the Broker runs. You can use a value of `default` to specify the machine's default encoding. The `charset-name` must be a valid Java encoding.

The `set encoding` command works the same as running JMSAdmin with the `-t` switch. For more information, see "Start-up Options" on page 47.

Example

The following command sets the encoding to UTF-8:

```
\> set encoding = utf-8
```

Set RecoverMode

Changes the Broker recover mode for recovery of pending transactions.

Syntax

```
set recoverMode [ ={ global | restricted } ]
```

Remarks

Use this command when working with XA transactions and an application server.

The recover mode deals with the recovery of XA transactions that were marked as pending, but did not execute because of a system problem.

By default, `recoverMode` is set to a value of `restricted`. This setting means that a call to the XAResource's `recover` method returns only those pending transactions in which the current Broker client participated.

A setting of `global` means that a call to the XAResource returns *all* the pending transactions within the Broker, not just those transactions in which the client making the request participated.

Issuing this command without a parameter displays the current `recoverMode` setting.

Examples

The following command sets the recover mode to `global`:

```
\> set recoverMode = global
```

The following command displays the value of the current `recoverMode` setting:

```
\> set recoverMode
restricted
```

Show

Displays the properties for a specified administered object in JNDI.

Syntax

```
show lookup-name
```

Remarks

In *lookup-name*, specify the JNDI name to which the object is bound.

Examples

The following `show` command displays the current properties of the queue that is bound to the name `Q1`:

```
/sub4/sub5> show Q1
Q1: com.webmethods.jms.impl.WmQueueImpl
    QueueName: Q1
    SharedState: false
    SharedStateOrdering: none
```

The following `show` command displays the current properties of the topic that is bound to the name `T1`:

```
/sub4/sub5> show T1
T1: com.webmethods.jms.impl.WmTopicImpl
    TopicName: TestAdmin::Topic::T1
    EventType: TestAdmin::Topic::T1
    SharedState: false
    SharedStateOrdering: none
```

Related Commands

"Display" on page 104

Unbind

Removes an administered object (connection factory, topic connection factory, queue connection factory, topic, or queue definition).

Syntax

```
unbind { lookup-name | * }
```

Remarks

In `lookup-name`, specify the JNDI name to which the object is bound. This command does not delete topics or queues from the Broker.

You can use the wildcard character (*) with `unbind`; using the wildcard character unbinds all webMethods objects and strings in the current context.

6 Configuring webMethods Messaging Clients for SSL

Overview

This chapter describes how to configure webMethods Broker clients for JMS and C# messaging clients for SSL authentication. It also explains how SSL encryption is used with messaging clients, and how SSL protection is used by access control lists (ACLs).

For a JMS or C# client to communicate with a Broker through SSL, the messaging client, the Broker Server, *and* the Broker admin component (hosted by the My webMethods Server) must have their own certificates. These certificates contain each component's SSL credentials, or *identity*.

This chapter deals with configuring SSL for Broker messaging clients. For more information, refer to *Administering webMethods Broker*. That manual explains the Broker security model, contains information on configuring SSL for the Broker Server *and* the Broker admin component, and describes the usage of ACLs in detail.

Note:webMethods Broker API for JMS uses JSSE (Java Secure Sockets Extension) to support SSL. You can optionally use Entrust to support SSL.

Securing JMS Clients with SSL

Use either the JMSAdmin command-line tool or My webMethods to configure the properties for SSL on the connection factory that will be used for the connection. These properties include:

- Specifying the keystore (its filename).
- Specifying the trust store (its filename)
- Optionally, selecting encryption (whether or not to enable encryption. Encryption is enabled by default)

For JMS clients, each user certificate and the corresponding private key pair is stored in a separate *keystore*. A keystore is a file that saves a certificate and the corresponding private key in PKCS12 format. These files are password protected. You can save only one keypair in a keystore.

For JMS clients, the public keys for all certificate authorities (CA) that the JMS client accepts are stored in a separate file called a *trust store*. A trust store file is not password protected. Unlike a keystore file, which can only save one client certificate, a trust store can contain certificates for multiple CAs.

The file format of a trust store is typically JKS. If the messaging client is JMS, you need to import the trusted root for the certificate authority (CA) to a trust store file.

Example:

The following example uses a JMSAdmin `bind TopicConnectionFactory` command to configure the SSL properties for the topic connection factory `MyTCF`:

```
bind tcf MyTCF with
   sslKeystore="c:\keystore.p12"
   sslTrustStore="c:\truststore.jks"
   sslEncrypted=true
```

In the JMS client code, set the SSL properties for the connection factory that will carry the SSL connection, as shown in the following example:

```
((WmConnectionFactoryImpl)tcf).setSSLKeystore("c:\keystore.p12");
((WmConnectionFactoryImpl)tcf).setSSLTruststore("c:\truststore.jks");
((WmConnectionFactoryImpl)tcf).setSSLEncrypted(true);
```

Enabling FIPS in JMS Clients

You can enable FIPS mode only if your SSL provider is Entrust. Perform the following steps to enable the FIPS mode in JMS clients. Your JVM must use JCE Unlimited Strength Jurisdiction Policy Files if you want webMethods Broker to run in FIPS mode. For more information about the installed JDKs, see the *Installing and Upgrading webMethods Broker* guide.

To enable FIPS mode in a JMS client

1. Set the SSL provider to Entrust as follows:

   ```
   WmJMSConfig.setSSLProvider(AwSSLUtil.SSL_PROVIDER_ENTRUST);
   ```

2. Enable the FIPS mode as follows.

   ```
   WmJMSConfig.setSSLFIPSModeEnabled(true);
   ```

   You can also enable the FIPS mode by setting `com.webmethods.jms.ssl.fipsmode=true` in the wmjms.properties file.

Upgrading JMS Clients that Use SSL

The webMethods Broker SSL implementation uses a trust store file to store trusted roots. When upgrading JMS clients that are SSL-secured, you must specify the trust store file in these situations:

- When re-binding connection factories using the new SSL setting.
- When creating connection factories at run-time using the factory class `WmJMSFactory`
- When using SSL with `WmJMSAdmin` clients
- When using jmsadmin command-line utility SSL connections

For more information about migrating from pre-version 7.1 keystore files and configuring SSL, see *Administering webMethods Broker*.

Securing C# Clients with SSL

You use Microsoft Management Console (MMC) and the Certificates Snap-In to configure and manage SSL for a C# client. When using MMC to configure SSL:

- Open MMC, select **File > Add/Remove Snap-In**, then **Add** the Certificates.

- Make sure to select the **Client Authentication** property of the Client Certificate.
- Import the trusted root for the C# client certificate into the Trusted Root Certification Authorities folder.
- Check the client and trusted root certificates to make sure that they have not expired.
- Assign a name value to the certificate's friendly name if it is empty. When a connection is created, this alias will be used by the connection factory.

  Important:
  Use the Friendly Name as the client certificate DN in your C# client connection code.

For more information on MMC, and on configuring SSL for Windows objects, refer to your Windows OS documentation.

Securing Other Broker Components with SSL

You also need to configure a keystore and trust store for the Broker Server to which a JMS or C# client will connect, and for the Broker admin component on My webMethods Server. The procedures for creating and managing certificates for a Broker Server and the Broker admin component are described in *Administering webMethods Broker*.

Encrypting Data

For Broker, the SSL encryption setting refers to the optional encrypting of the *data traffic* between a messaging client and the Broker Server. SSL encryption is separate from SSL authentication. You can turn on SSL data encryption with or without configuring SSL authentication.

Encrypting the data traffic provides a high level of security. Typically, you would consider using encryption when:

- The data traffic contains sensitive data.
- The network is susceptible to packet sniffing done for malicious purposes.

If the data stays within your corporate LAN and the network uses switches rather than hubs, the risk of a security breach is low, and encryption may not be necessary.

If the data travels over a WAN or the Internet (for example, Broker clients to a Broker in a DMZ, or Broker-to-Broker over the Internet), encryption may be a worthwhile security precaution. Of course, if the data is extremely sensitive, then encryption should be added to the security model, whatever the environment.

For more information about using SSL encryption with the Broker Server, see *Administering webMethods Broker*.

SSL Authorization and Access Control Lists

An Access Control List (ACL) is a list of SSL distinguished names (DNs) that is attached to a Broker object (client group, Broker Server, territory, territory gateway, cluster, cluster gateway) representing an access point. It grants permission for clients on the list to access the SSL-protected Broker object. To gain access to a Broker, a client must have an identity (distinguished name and authenticator's distinguished name) that matches an entry on the ACL. Clients whose DNs do not match those on the ACL are denied authorization.

There are several different types of ACL. ACLs authorize clients for:

- Administrative access to a Broker Server
- Membership in a client group
- Permission to access data over specified Broker gateways and territories

You can configure ACLs only if you are a Broker administrator. *Administering webMethods Broker* explains how to configure the various types of ACL.

7 Configuring Messaging Clients for Basic Authentication

Overview

JMS clients can connect to Broker as JMS providers over a secure connection established using basic authentication. This connection requires that you enable basic authentication on the Broker Server.

This chapter describes how to configure the connections from JMS clients using the basic authentication capabilities provided by Broker. For more information about configuring basic authentication on Broker Server, see *Administering webMethods Broker*.

The JMS clients can create connections with basic authentication by using any of the following:

- webMethods APIs for JMS
- System properties
- The wmjms.properties file

Enabling Basic Authentication Clients Using the webMethods APIs for JMS

You use the following webMethods API for JMS to pass the basic authentication credentials to Broker Server. This API allows JMS clients to create a connection with Broker by using the user identity you specify. The API throws JMSSecurityException if client authentication fails due to an invalid user name or password.

```
createConnection(String userName, String password)
```

When you pass only a username and password combination, the webMethods API for JMS creates a basic authentication connection.

If you have already specified the keystore through the property file, or through the WmJMSConfig.setSSLKeystore() API, then the value of *username* corresponds to SSL DN and the value of *password* corresponds to the SSL keystore password. However, if keystore is not already specified, then the value of *username* corresponds to the basic authentication user name and the value for *password* corresponds to the password that the basic authentication user will use.

```
WmConnectionFactoryfactory  = new WmConnectionFactoryfactory();
factory.setClientGroup(CLIENT_GROUP);
factory.setClientID("testClient");
factory.setBrokerName("Broker #1");
factory.setBrokerHost("localhost");

String username="sag";
String password="SoftwareAG123";

connection = factory.createConnection(username, password);
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
// continue with publish or subscribe
```

Setting the System Properties to Enable Basic Authentication Clients Using the webMethods APIs for JMS

You can also use the System Properties to configure basic authentication. The following table helps you configure basic authentication by using the system properties on JMS clients.

To set this basic authentication property...	Set this system property...	As shown here...
Username	com.webmethods.jms.username	`System.setProperty("com.webmethods.jms.username","sag");`
Password	com.webmethods.jms.password	`System.setProperty("com.webmethods.jms.password", "SoftwareAG123");`

Using the wmjms.properties File to Secure JMS Clients with Basic Authentication

You can also use the wmjms.properties file to configure basic authentication. Specify the properties for basic authentication in the wmjms.properties file as follows:

Specifying the properties for basic authentication in the wmjms.properties file

1. Create a Java properties file named wmjms.properties.
2. Add the following properties to the wmjms.properties file as shown below:

```
com.webmethods.jms.username="sag"
com.webmethods.jms.password="SoftwareAG123"
```

3. Add the directory containing the wmjms.properties file to your CLASSPATH.

   The webMethods Broker-specific properties are defined in the Javadoc of the WmJMSConfig class; use that information before reconfiguring the property values in your code.

Securing C# Clients with Basic Authentication

You must enable basic authentication on a Broker Server if you want a C# client to connect to the Broker Server by using basic authentication mechanism.

To secure C# clients with basic authentication, create a connection with basicAuthInfo in the C# client code as follows, where `basicAuthInfo` contains the basic authentication username and password:

```
IConnection CreateConnection(BasicAuthInfo basicAuthInfo);
```

For details about enabling basic authentication on Broker Server, see *Administering webMethods Broker*.

III Application Server Support

8 Application Server Support When You Use webMethods Broker as a JMS Provider

Overview

This chapter describes the support for application servers when you use webMethods Broker as a JMS provider. It provides a high-level summary of these facilities, lists the application servers and describes the package installation and environment configuration for some of the application servers.

JMS Application Server Support

webMethods Broker used as a JMS provider provides JMS clients the support for application servers as described in the Java Message Service standard. It provides this support through Java EE Connector Architecture (JCA) and JMS Application Server Facilities (ASF).

When you use webMethods Broker as a JMS provider, configuring the supported application server allows the components hosted by the server, such as message-driven beans (MDBs), to access the services and functions of the webMethods Broker that is used as a JMS provider, thus enhancing their feature set.

webMethods Broker used as a JMS provider supports the `ConnectionConsumer` interface as described by the Java Message Service standard. webMethods Broker used as a JMS provider is a Java Transaction API (JTA)-capable provider that supports XA interfaces. This enables webMethods Broker to participate in distributed transactions in conjunction with the application server, using a transaction manager supplied by the application server.

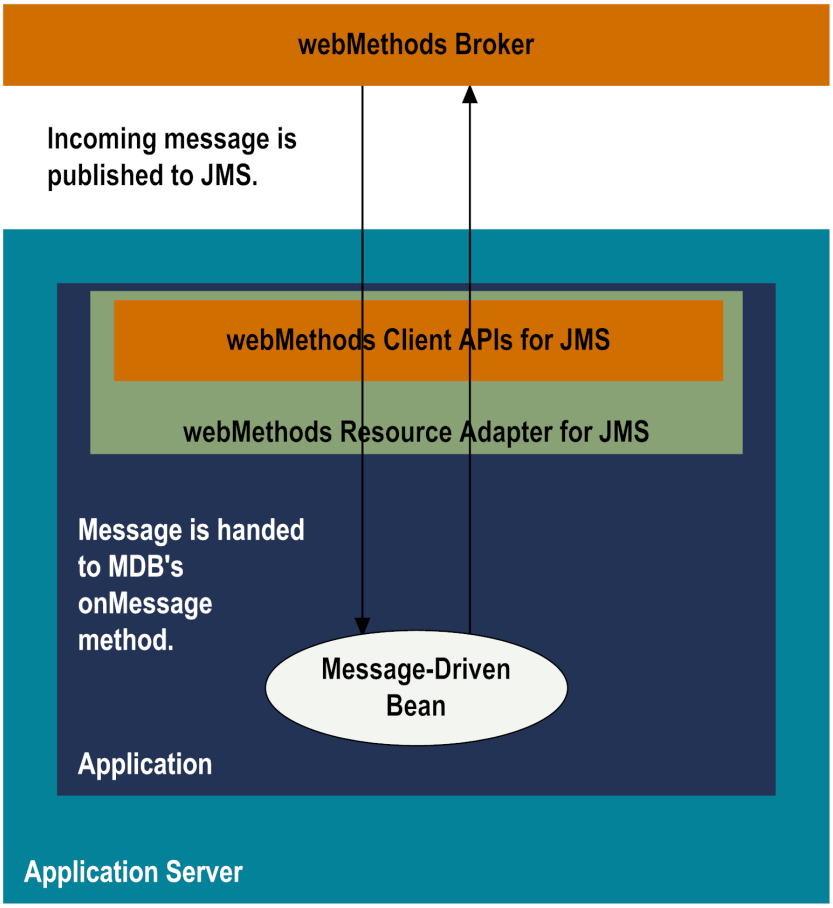
Support through JCA

webMethods Broker as a JMS provider provides JCA support for the following application servers:

- IBM WebSphere Application Server
- JBoss Application Server

The installation of webMethods Broker used as a JMS provider includes a custom JCA resource adapter that uses the JMS client library and integrates webMethods Broker with an application server.

The following diagram illustrates the flow of data between webMethods Broker that is used as a JMS provider, the webMethods Resource Adapter for JMS, the application server, and a message-driven bean.



As shown in the diagram above, incoming messages are published to the webMethods Client for JMS. The webMethods Client for JMS hands the message to the webMethods Resource Adapter for JMS. The webMethods Resource Adapter for JMS hands the message to the `onMessage` method of the MDB.

webMethods Resource Adapter for JMS

The webMethods Resource Adapter for JMS works with the application server to provide the appropriate transaction and connection mechanisms. The webMethods Resource Adapter for JMS is used within the address space of the application server.

webMethods Resource Adapter for JMS supports the specification of Generic Resource Adapter for JMS. For information about the properties of Generic Resource Adapter for JMS, see the Generic Resource Adapter documentation.

The webMethods Resource Adapter for JMS, webm-jmsra.rar, is located in the *webMethods Broker_directory* \lib directory on the machine where you installed webMethods Broker as a JMS provider client software. Use the properties in the ra.xml file to configure the resource adapter. For descriptions of the properties in ra.xml file, see “Resource Adapter Configuration Properties in ra.xml File” on page 169.

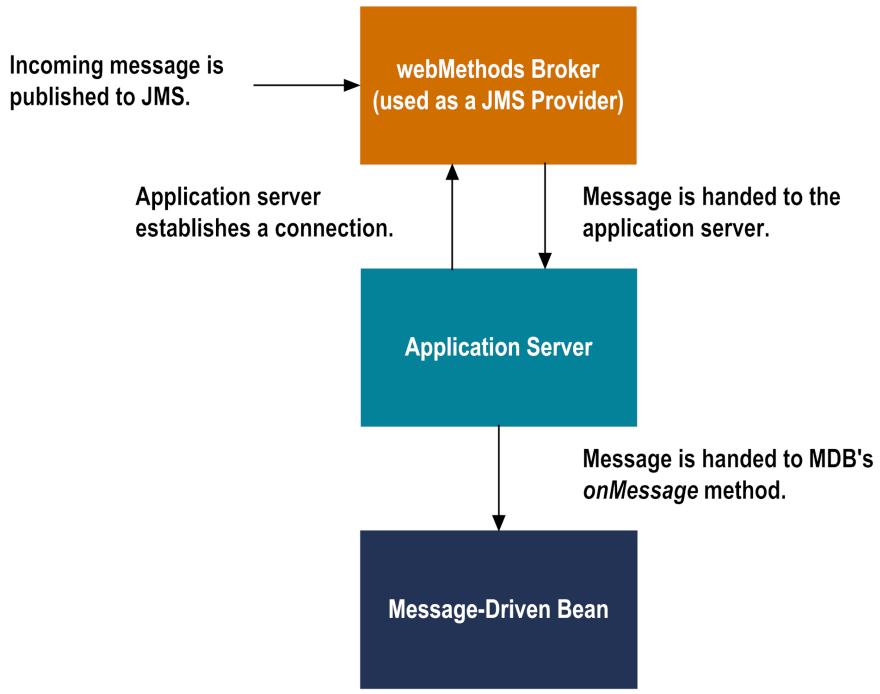
Configuration instructions for the resource adapter are provided in "Sample IBM WebSphere Application Server Configuration through JCA" on page 162 and "Sample JBoss Application Server Configuration through JCA" on page 164.

Support through ASF

When you use webMethods Broker as a JMS provider, it provides ASF support for the following application servers:

- Oracle WebLogic
- IBM WebSphere
- JBoss

The following diagram illustrates the flow of data between webMethods Messaging, the application server, and a message-driven bean.



Sample IBM WebSphere Application Server Configuration through JCA

IBM WebSphere Application Servers are supported when you use webMethods Broker as a JMS provider through JCA and ASF.

Overview

To use webMethods Messaging with WebSphere Application Server through JCA, the process includes:

- Installing webMethods Broker as a JMS provider and the client libraries.

  For information, see *Installing and Upgrading webMethods Broker*.

- Configuring the webMethods Resource Adapter for JMS.
- Optionally, configuring the selected JNDI provider.

  For information about JNDI refer to the documentation for configuring a foreign JMS provider in the WebSphere documentation.

To configure IBM WebSphere Application Server 7.0, see section "Configuring IBM WebSphere Application Server 7.0" on page 163. For information on configuring any other version of WebSphere Application Server, see the WebSphere documentation.

Configuring IBM WebSphere Application Server 7.0

You can configure the WebSphere Application Server 7.0 to use webMethods Broker as a JMS provider, by using the WebSphere console to configure webMethods Broker as a generic JMS provider.

For more information, refer to the topic "Configuring a Generic JMS Provider" in the WebSphere 7.0 documentation.

To configure the IBM WebSphere Application Server 7.0

1. Launch the WebSphere console on the browser and log in to the server.
2. Add the webMethods Resource Adapter for JMS to the WebSphere Application Server. To do this, go to **Resources > Resource Adapters** and click **Install RAR**. In the RAR list, select webm-jmsra.rar.
3. On the Resource Adapters page, select **webMethods Resource Adapter for JMS** from the **Resource Adapter** list.
4. Configure the resource adapter as follows:
   a. If the application server supports XA, open the custom properties page for the resource adapter and set the `supportsXA` property to `true`.
   b. Create a new Activation Specification. Open the **J2C Activation Specification** page and click **New**. Provide a name for the activation specification that will be used by the WebSphere Application Server and click **Apply**.
      a. On the **J2C Activation Specification** page, click **Custom Properties**.
      b. If the application server supports XA, set the `supportsXA` property to `true`.

c. Set the `destination type` property to `javax.jms.Topic`.

d. Enter a `destination` property name.

e. If you use JNDI, specify the following additional settings:

Parameter	Specify...
`providerIntegrationMode`	`JNDI` as the value.
`jndiProperties`	Your JNDI settings. For example: `java.naming.factory.initial=com.webmethods.jms.naming.WmJmsNamingCtxFactory,java.naming.provider.url=wmjmsnaming://Broker#1@localhost:6849/JMSAdminTest,com.webmethods.jms.naming.clientgroup=admin`

c. Set `connectionFactoryJndiName` to reflect the connection factory from which the MDB will receive messages.

d. Set `destinationJndiName` to reflect the destination from which the MDB will receive messages.

Use the above activation specification in MDB to receive and send messages to Broker.

5. Click **Apply** and **OK**.
6. Save your configuration in the console to apply changes to the master configuration.
7. Restart the WebSphere Application Server.

WebSphere Application Server Usage Notes

When modifying a connection factory used by the WebSphere Application Server, the server needs to be stopped and re-started before the changes take effect.

Sample JBoss Application Server Configuration through JCA

JBoss Application Servers are supported when you use webMethods Broker as a JMS provider through JCA.

Overview

To use webMethods Broker as a JMS provider with JBoss Application Server, the process includes:

- Installing webMethods Broker as a JMS provider and client libraries.

For information, see *Installing and Upgrading webMethods Broker*.

- Configuring the webMethods Resource Adapter for JMS.
- Optionally, configuring the selected JNDI provider.

  For information about JNDI refer to the documentation for configuring a foreign JMS provider in the JBoss documentation.

To configure JBoss Application Server 5.0.1, see section "Configuring JBoss Application Server 5.0.1" on page 165. For information on configuring any other version of JBoss Application Server, see the JBoss documentation.

Configuring JBoss Application Server 5.0.1

To install webMethods Broker on a JBoss Application Server, add the webm-jmsra.rar file (the JMS client file) to the JBoss directory, and then configure the JMS client files for the JBoss Application Server.

You can also connect external JNDI providers to the JBoss JNDI provider. JBoss lets you use an external context to make these connections. For configuration information, see the JBoss documentation.

To configure the JBoss Application Server 5.0.1

1. Extract the contents of *Software AG_directory* \Broker\lib\webm-jmsra.rar to a temporary directory.

2. If you use JNDI, specify the following additional settings in the ra.xml file. The ra.xml file is located in webm-jmsra.rar\META-INF.

   - Set the `jndiProperties` value to reflect your JNDI settings. For example:

     ```
     java.naming.factory.initial=com.webmethods.jms.naming.WmJmsNamingCtx
     Factory,java.naming.provider.url=wmjmsnaming://Broker#1@localhost:6849/
     JMSAdminTest,com.webmethods.jms.naming.clientgroup=admin
     ```

   - Set the `connectionFactoryJndiName` value to reflect the connection factory from which the MDB will receive messages.
   - Set the `destinationJndiName` value to reflect the destination from which the MDB will receive messages.

   Save your changes and recompress the webm-jmsra.rar file.

3. Add the webm-jmsra.rar file to the *`JBoss_directory`*\server\default\deploy directory.

   a. Create the webm-service.xml file in the deploy directory of JBoss. The webm-service.xml file must define the external context MBean that allows JBoss to access an external JNDI provider. In this case the external JNDI provider is webMethods.

b. Change the connection factory attributes to use webMethods Broker as a JMS provider. In the deploy\messaging<jboss.home>\server\default\deploy\messaging\jms-ds.xml file.

For example, replace the following line:

```
<attribute name="TopicFactoryRef">java:/XAConnectionFactory
</attribute>
```

with the following:

```
<attribute name="TopicFactoryRef">external/webm/NewConnectionFactory
</attribute>
```

Note that *NewConnectionFactory* is defined in the webm- service.xml file.

4. Update jboss.xml to point to the desired Destination and deploy the MDB.

Sample Oracle WebLogic Application Server Configuration through ASF

Oracle WebLogic Application Servers are supported when you use webMethods Broker as a JMS provider through ASF.

Overview

To use webMethods Broker as a JMS provider with WebLogic application server, the process includes:

- Installing webMethods Broker as a JMS provider and the client libraries.

  For information, see *Installing and Upgrading webMethods Broker*.

- Setting the CLASSPATH and environment variables.
- Configuration related to the selected JNDI provider.
- Configuring XML descriptor files.
- Administrative commands needed to deploy the JMS client. Deploying the JMS client.

To configure WebLogic Application Server 10.3.3, see section "Configuring Oracle WebLogic Application Server 10.3.3" on page 166. For information on configuring any other version of WebLogic Application Server, see the WebLogic documentation.

Configuring Oracle WebLogic Application Server 10.3.3

WebLogic 10.3.3 provides automated scripts to set the environment variables and start the WebLogic 10.3.3 server (using the WebLogic examples Server). The instructions in "Setting the CLASSPATH and Environment Variables" on page 167 show how to modify these scripts to allow the webMethods Broker that is used as a JMS provider to act as a foreign JMS provider.

Setting the CLASSPATH and Environment Variables

You must set the WebLogic CLASSPATH to incorporate webMethods Broker as a JMS provider and the selected JNDI provider.

To set the CLASSPATH and environment variables

1. Set the WebLogic CLASSPATH (WEBLOGIC_CLASSPATH) to incorporate webMethods Broker as a JMS provider and the selected JNDI provider.

   - **Windows:** Change the CLASSPATH in WebLogic_*HomeDirectory*\common\bin\commEnv.cmd to point to the required JAR files.
   - **Unix:** Change the CLASSPATH in *WebLogic_HomeDirectory*/common/bin/commEnv.sh to point to the required JAR files.

2. Modify your WEBLOGIC_CLASSPATH so that the following JAR files precede the WebLogic JAR files:

   - webMethods Broker as a JMS provider: wm-jmsclient.jar
   - webMethods Broker as a JMS provider: wm-jmsadmin.jar
   - webMethods Broker as a JMS provider: wm-g11nutils.jar
   - webMethods Broker as a JMS provider: wm-jmsnaming.jar
   - webMethods Naming Service for JMS: wm-brokerclient.jar
   - selected JNDI provider: <JAR files for the selected JNDI provider>

   Note that when you use webMethods Broker as a JMS provider, the JAR files need to precede the WebLogic JAR files in the CLASSPATH.

Configuring XML Descriptor Files

Configure Weblogic-ejb-jar.xml to point to the webMethods Broker that is used as a JMS provider as shown in the XML sample:

```
<weblogic-ejb-jar>
 <weblogic-enterprise-bean>
  <ejb-name>SampleMessageDrivenBean</ejb-name>
  <message-driven-descriptor>
   <destination-jndi-name>SampleMDBTopic</destination-jndi-name>
   <initial-contextfactory>
   com.webmethods.jms.naming.WmJmsNamingCtxFactory</initial-contextfactory>
   <provider-url>wmjmsnaming://Broker
   #1@localhost:6849/JMSAdminTest</provider-url>
   <connection-factory-jndi-
   name>SampleMDBConnectionFactory</connectionfactory-
   jndi-name>
  </message-driven-descriptor>
 </weblogic-enterprise-bean>
```

```
</weblogic-ejb-jar>
```

Deploying the JMS Client

To deploy the JMS client, copy the MDB ear file to the <*WebLogic_HOME*>\..\user_projects\domains\base_domain\autodeploy directory.

WebLogic Application Server Usage Notes

The following technical considerations apply when you use webMethods Broker as a JMS provider with the WebLogic Application Server:

- When you use webMethods Broker as a JMS provider, the connection factories that generate and bind into the JNDI provider implement the `XATopicConnectionFactory` and `XAQueueConnectionFactory` interfaces. These factories provide support for the Java Transaction API (JTA). The WebLogic Application Server will create actual XA connections and XA sessions from the connection factories only if the trans-attribute is set to "Required" in the ejb-jar.xml deployment descriptor file.
- A WebLogic MDB listener with a message selector does not receive messages if the publisher and subscriber are sharing the same `connectionId`.

  WebLogic Application Server sets the `noLocal` flag to `True` if filters are enabled. Because of this, when you use webMethods Broker as a JMS provider, it does not recognize messages if the publisher and subscriber are sharing the same `connectionId`. To work around this limitation, ensure that the `connectionIds` for publishers and subscribers are different in the objects that you bind to the JNDI provider.

Application Server Usage Notes

These technical considerations apply when you use webMethods Broker as a JMS provider with any of the supported application servers.

- Tuning the configuration properties when you use webMethods Broker as a JMS provider. Adjusting these properties, which are located in the `WmJMSConfig` class, may be useful in the situations described below. For more information, see "Properties Specific to webMethods Broker Used as a JMS Provider" on page 20 and the *webMethods Broker API for JMS Reference* for the `WmJMSConfig` class.
  - If you need multiple durable subscribers to share the same clientID, change the clientIDSharing setting as follows:

    ```
    com.webmethods.jms.clientIDSharing=true
    ```

    Sharing the clientID was allowed by default in releases prior to 6.5; however, in the current release, you must set this property for multiple durable subscribers to share the same clientID.
  - To improve the response time for detecting disconnections between a Broker and a client:

    ```
    com.webmethods.jms.broker.keepAliveTimeout=30
    ```

```
com.webmethods.jms.broker.keepAliveAttempts=100
com.webmethods.jms.broker.keepAliveInterval=30
com.webmethods.jms.keepAliveInterval=30
```

- **Setting the Broker recover mode**. The default setting for the Broker recover mode is `restricted`. When you use webMethods Broker as a JMS provider with application servers, it needs to be changed to `global`.

  Use the JMSAdmin command-line tool (described in "The JMSAdmin Command-Line Tool" on page 45), or the Broker user interface in My webMethods to change the setting.

  ```
  set recoverMode=global
  ```

  For more information about setting the Broker recover mode with the JMSAdmin command-line tool, see "Set RecoverMode" on page 144. For information about setting the Broker recover mode through My webMethods, see *Administering webMethods Broker*.

Resource Adapter Configuration Properties in ra.xml File

Set the properties in the ra.xml file to configure the resource adapter. The ra.xml file is located in the webm-jmsra.rar\META-INF folder.

Property	Description
`ForceDelayedTxn`	Specifies the whether the resource adapter uses InboundXAResourceProxy or SimpleXAResourceProxy. ■ If `ForceDelayedTxn=true`, the resource adapter uses InboundXAResourceProxy. ■ If `ForceDelayedTxn=false` and the `RedeliveryAttempts=0`, then the resource adapter uses SimpleXAResourceProxy. The default value is `true`.
`supportsXA`	Specifies whether the resource adapter supports XA transactions.
`jndiProperties`	Specifies your JNDI settings.
`connectionFactoryJndiName`	Specifies the connection factory from which the MDB will receive messages.
`destinationJndiName`	Specifies the destination from which the MDB will receive messages.

IV Coding Messaging Client Applications

9 A Basic JMS Sender-Receiver Client

Overview

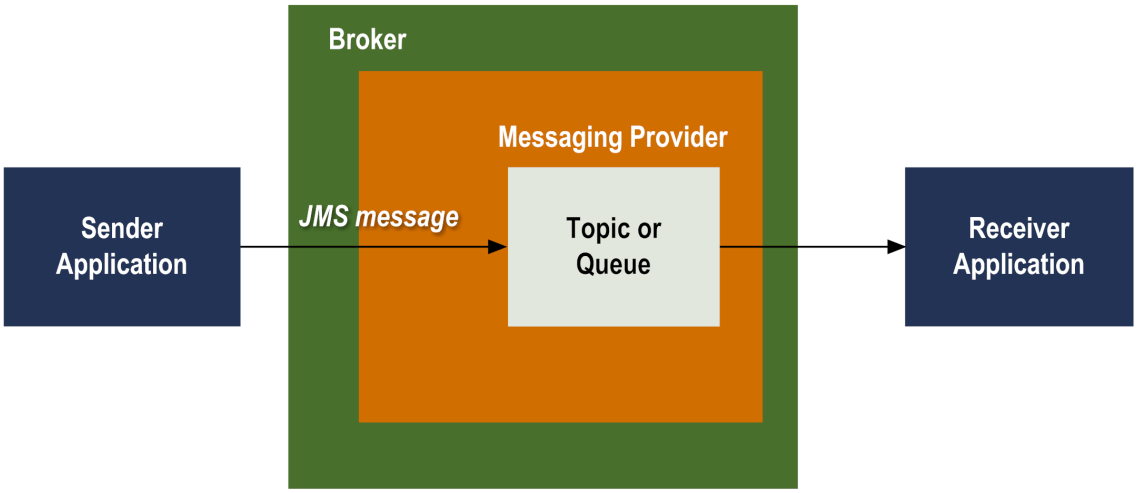
This chapter describes the basics of coding a standalone webMethods Client for JMS, the webMethods Broker used as a JMS provider, using a simple application that sends and receives a JMS message. The chapter shows how to code the principal JMS objects, message production and delivery, and walks you through the administrative commands needed to configure the application.

The sample application in this chapter is the basis for the more complex messaging scenario in the next chapter. It is recommended that you examine the JMS code for this example before continuing on to the following chapter.

Note:
The example in this chapter supports the Java Message Service Version 1.1 standard.

Sender-Receiver Application Components

The example consists of a *sender* application that connects to a webMethods Broker and sends a single JMS text message to a destination. A *receiver* application consumes the message and then shuts itself down.



JNDI Properties File

The client uses the JNDI configuration information contained in the properties file specified in the client's CLASSPATH. The JMS administrative tool accesses this configuration information (see "Administrative Setup Commands" on page 178). Using a JNDI properties file has the advantage of allowing different JNDI providers to be used without having to modify the client code.

The following example shows a properties file for the sender-receiver client:

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://anteater.north.webmethods.com/ou=jmsdept,
```

```
      ou=pd,o=wm
java.naming.security.principal=cn=Manager,ou=jmsdept,ou=pd,o=wm
java.naming.security.credentials=JMSatWEBM
```

JNDI Lookup Code

One of the first steps in creating a JMS client is writing the code for looking up the client's administered objects in JNDI.

In the lookup code for this example, the variable *factory* is defined as an object of type `ConnectionFactory`, which signifies a "generic" connection factory. Although it is possible to specify connection factories as topic connection factories or queue connection factories, coding a connection factory in this manner provides a greater degree of flexibility, allowing you to determine the type of connection factory at run time rather than at design time.

The lookup code also defines a generic *destination* variable, which represents the text message to be sent and received. Consistent with usage of a generic `ConnectionFactory`, usage of a generic destination allows this object to be specified as a queue, topic, temporary queue, or temporary topic at run time.

Both the sender and receiver applications use the `initialize()` method containing the JNDI lookup code (shown following). Its parameters (the names of the connection factory and destination) are passed in at run time as command line arguments.

```
protected ConnectionFactory   factory;
   protected Destination         destination;

   protected SimpleApplication() {}

   public boolean initialize(String factoryName, String destinationName)
   {
       factory     = lookupFactory(factoryName);
       destination = lookupDestination(destinationName);

       return factory != null && destination != null;
   }

   public boolean initialize(String factoryName)
   {
       factory     = lookupFactory(factoryName);
       return factory != null;
   }
```

The code for the two JNDI lookup methods is shown below. These methods get an initial context from the JNDI namespace (an initial context provides access to the JNDI provider and connects to the namespace in which the administered objects are stored). The methods then return the objects that correspond to the names passed in as parameters.

```
   private Destination lookupDestination(String name)
   {
       try {
           Context namingContext = new InitialContext();
           Object  destinationObject = namingContext.lookup(name);
           namingContext.close();
```

```
            if (destinationObject instanceof Destination) {
                return (Destination) destinationObject;   }
. . .
    private ConnectionFactory lookupFactory(String name)
    {
        try {
            Context namingContext = new InitialContext();

            Object  factoryObject = namingContext.lookup(name);
            namingContext.close();

            if (factoryObject instanceof ConnectionFactory) {
                return (ConnectionFactory) factoryObject; }
. . .
```

Coding Messaging Objects

A JMS client typically creates a connection, one or more sessions, and several message producers and consumers. This section shows how to write code for these objects:

- A *connection object* encapsulates an open connection with a JMS provider. When a connection is created, it is in stopped mode, meaning that no messages can be received.

  The JMS client in this example consists of two applications (a message sender and message receiver) that must each connect to the webMethods Broker used as a JMS provider; therefore, two connections are required.

- A *session object* is a single-threaded context for producing and consuming messages, and is used to create the necessary message producer and message consumer objects. A session also provides the context for grouping a set of send and receive messages into a transactional unit.

- A *message producer object*, created by a session object, is used to send messages to a destination (a topic or a queue).

- A *message consumer object*, created by a session object, is used for receiving messages sent to a destination.

Message Sender

Following is the code used for creating the messaging objects in the sender application:

```
conn = factory.createConnection();
Session   session = conn.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
MessageProducer sender = session.createProducer(destination);
```

To create the connection object, call the `createConnection()` method on the factory object. To create the session object, call the `createSession()` method on the connection. The `createSession()` method takes the following two parameters:

- **Transaction mode.** Here, the transaction mode is set to false, indicating that transactions will not be used.
- **Acknowledgment mode**. In this example, the mode is set to `AUTO_ACKNOWLEDGE`, indicating that the session automatically acknowledges a client's receipt of a message.

You create a message producer by executing the `createProducer()` method on a session. `createProducer()` takes a single parameter, the destination to which the message will be sent.

Message Receiver

Following is the code used to create the messaging objects in the receiver application:

```
conn  = factory.createConnection();
Session   session = conn.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
MessageConsumer receiver = session.createConsumer(destination);
```

You define the connection and session objects the same as for the message sender application. However, the message receiver defines a message consumer rather than a message producer. The message consumer is created calling `createConsumer()` on the session.

Managing the Connection

After coding the necessary connection objects in your JMS client, you need to manage their state; that is, when to start and close a connection.

In JMS, when a connection is created, it is in stopped mode, meaning that no messages can be received on that connection. However, messages are able to be sent.

In this example, you need to start the connection on the receiver application so that it can receive messages. This is accomplished by calling the connection's `start()` method:

```
conn.start()
```

When the application will no longer use a connection, invoke the connection's `close()` method:

```
conn.close()
```

You should always close a connection during application cleanup whether or not a previous `start()` method was called on the connection.

Implementing Message Sending and Reception

The following lines of code from the sender application show how to define and send a simple text message.

```
  public void sendAMessage(String messageText)
  {
    ...
```

```
      TextMessage     msg = session.createTextMessage();
      msg.setText(messageText);
      sender.send(msg);
```

The following code block from the receiver application shows an implementation for receiving a simple text message:

```
    Message msg = receiver.receive();

    if (msg instanceof TextMessage) {
       System.out.println("Received message: " +
       ((TextMessage) msg).getText());
       } else {
           System.out.println("Received message: " + msg.toString());
   }
```

The `receive()` method in this sample application receives one message at a time (that is, a single message each time it is called). In a production application, you would typically call `receive()` from within a loop so it is able to receive messages continuously.

Configuring the Administered Objects and Running the Client

Before configuring the administered objects and deploying a client, webMethods Broker must be installed and configured. For information, see *Installing and Upgrading webMethods Broker*.

Administrative Setup Commands

The following sections use the webMethods Broker JMSAdmin command-line tool to set up the client. The commands locate the JNDI properties file and the JNDI context, bind the objects, create a topic and subscription, and set permissions for the sender and receiver applications.

Following is the complete sequence of JMSAdmin commands that you would use to set up the example in this chapter, including any messages JMSAdmin directs to the console. The numbered labels identify lines of code that are described in more detail in the table that follows.

```
[1] jmsadmin -p jndi.properties

    Connected to JNDI Context: ou=jmsdept,ou=pd,o=wm
[2] > bind cf aSenderFactory with group=SampleSenders
    > bind cf aReceiverFactory with group=SampleReceivers
    > bind topic aTopic with topicName=Samples::RequestInfo
[3] > connect
    Connected to Broker #1 on localhost:6849
[4] > create topic for aTopic
[5] > create group SampleSenders
    > permit group SampleSenders to publish Samples::RequestInfo
    > create group SampleReceivers
    > permit group SampleReceivers to subscribe Samples::RequestInfo
[6] > end
```

Line(s)	Description
1	Start JMSAdmin using a properties filename. The tool connects to JNDI with the configuration properties specified in the sample properties file.
2	Bind connection factories into a JNDI context for the sender and receiver applications, and assign them to a Broker group. **Note:** Note that the connection factory names must be different for the two applications. The last bind command binds the topic to which the message will be delivered into a subcontext. For more information, see "Bind TopicConnectionFactory" on page 86 and "Bind Topic" on page 84.
3	Connect to the Broker. Using the connect command without parameters starts a connection to the default Broker on the default server (localhost on port 6849), as indicated by the message shown after the connection is made. For more information, see "Connect" on page 89.
4	Create the topic used by the sender and receiver applications. You can only use create topic after a connection to the Broker has been established. For more information, see "Create Topic" on page 98.
5	Create groups for the sender and receiver applications and assign them the appropriate Broker publish and subscribe access permissions. For more information, see "Create Group" on page 96 and "Permit" on page 141.
6	End the JMSAdmin session using the end command.

10 JMS Request-Reply Application with a Message Selector

Overview

This chapter describes how to code a JMS request-reply application with a message selector.

The example used throughout this chapter builds on the sender-receiver application described in "A Basic JMS Sender-Receiver Client" on page 173. The example demonstrates features of JMS messaging, such as implementing message replies, a message listener, a message selector, and a local transaction. It also shows how to set Broker access permissions and how to code messages on different threads.

Note:
The example in this chapter supports the Java Message Service Version 1.1 standard.

Request-Reply Application: Description

The request-reply application in this example manages user requests for customer information.

Application Components

A *requestor* application connects to a webMethods Broker and sends a request for information about a customer (contained in a JMS message) to a server application. Using information in the request message, the server application builds a query, which it executes against a customer database.

The server retrieves the query results containing the customer information, and maps the data to a JMS reply message. The server then sends the reply to the requestor application. The requestor application processes the reply message, extracts the customer data, and sends the customer data to the console for viewing.

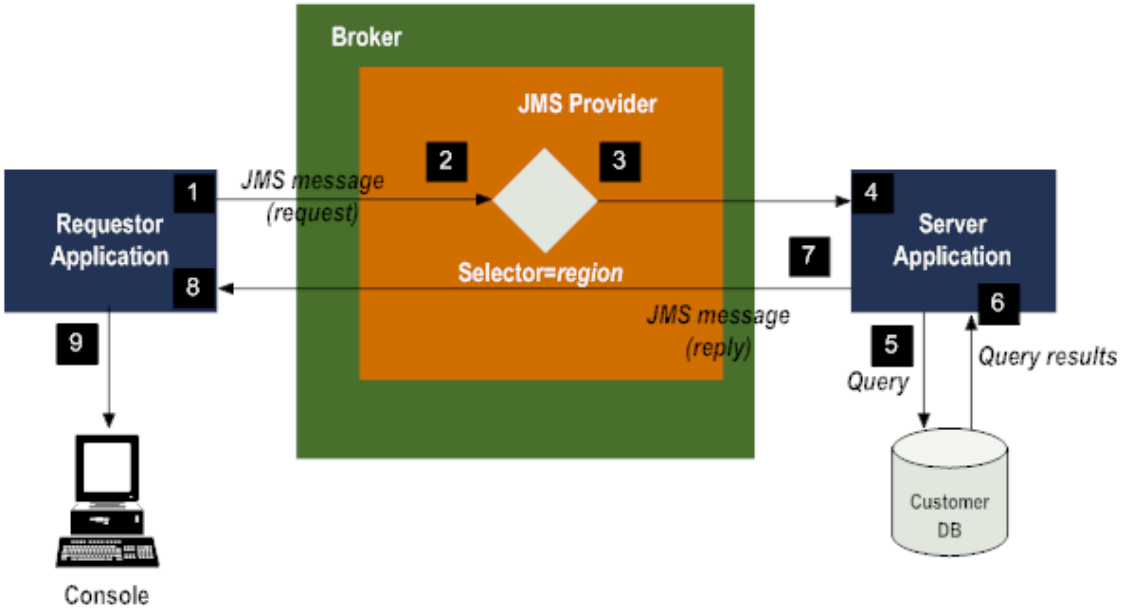
Using a Message Selector

The application uses a *message selector* to filter customer requests; only requests about customers in a selected geographical region (Eastern states or Western states) are received.

For more information, see "Message Consumer" on page 32.

Message and Data Flow

The following figure shows the JMS client's message and data flow.



The numbers in the figure correspond to the sequence of messaging steps described below:

Step	Description
1	The requestor application builds a JMS message containing a lookup request for customer information.
2	The requestor sends a message to the server application. The message contains the information needed to lookup a customer record.
3	The message selector ensures that only messages from the selected geographical region are passed to the server.
4	The server receives the message requesting customer information.
5	The server parses the request message, builds a database query, and executes the database query against a simple in-memory database containing customer records.
6	The server maps the query results to a reply message and links the reply message to the original request.
7	The server sends the reply back to the requestor application.
8	The requestor extracts the customer data from the reply.
9	The requestor sends the customer data to the console.

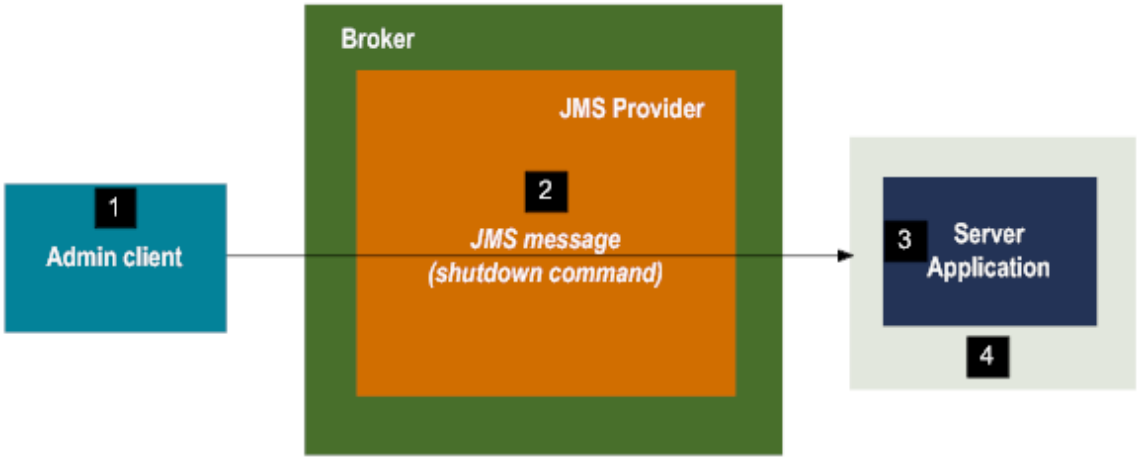
Using a Second Messaging Thread

In JMS, each session supports a single thread. For an application to support the reception of concurrent messages, the application must contain a session (thread context) for each message receiver.

The customer inquiry application constitutes one messaging thread: messages are sent from a requestor application and received asynchronously by a server. Adding a second thread requires that another session be created and message reception handled on that thread.

To demonstrate use of a second messaging thread, the example includes code for sending and receiving a single administrator command. The command is sent by an administrator client and received by the server application. When the server receives the command, it shuts itself down.

The following figure shows the execution flow for the example's second message reception thread.



The numbers in the figure correspond to the steps that occur when the `shutdown` command is issued:

Step	Description
1	The admin client application is started with the server `shutdown` command entered on the command line.
2	The command is copied to a JMS message and sent to the server.
3	The server extracts the command string from the message.
4	The server executes the command and shuts down.

Application Code

This section describes the application code, focusing on the application's JMS messaging functionality.

Topics covered by the basic JMS example, such as managing the connection, setting up the JNDI code for the administered objects, and creating the sender and receiver objects are not repeated here. The code descriptions focus on the following areas:

- Implementing an asynchronous message listener. The asynchronous message listener receives customer inquiry messages and is installed on the server.
- Linking a reply to a request message. The reply message, which contains the results of the customer inquiry, and is linked through message header fields to the original request.
- Implementing a local transaction. The request message for customer information, and its reply message and acknowledgment are grouped into a local transaction.
- Using a temporary queue. Since reply messages are only needed for the duration of a session, they are stored on a temporary queue by the requestor application.
- Using a message selector. The message selector determines whether a customer request will be received, based on the Boolean expression specified by the selector.
- Receiving messages on a second thread. The example includes a second session and a message consumer with a synchronous `receive()` method that manages the reception of an administrator message on a second thread.

Implementing an Asynchronous Message Listener

The ServerApplication class implements a single asynchronous message listener to receive customer inquiries. The message listener is implemented through the JMS `MessageListener` and `MessageConsumer` interfaces.

```
import javax.jms.MessageListener;
import javax.jms.MessageConsumer;


public class ServerApplication extends SimpleApplication implements
          MessageListener

...
   receiver.setMessageListener(this);
```

The `MessageListener` object receives the incoming customer request messages asynchronously from the requestor application. The MessageConsumer's `setMessageListener()` method is necessary to activate the listener so that it is able to receive messages.

In JMS, each message listener supports a single `onMessage()` method. In this example, the method is called whenever the server receives a customer request message. This implementation of `onMessage()` copies, from the request message, the customer information needed to make the database query. It then issues the query and builds the reply message, calling `constructReplyMessage()` to map data from the customer record to the reply message fields. The following code from `onMessage()` shows these steps.

```
public void onMessage(Message message)
    {
        try {
           Message reply;
```

```
String  command =
    message.getStringProperty(AppConstants.CMD_PROP);
if (command.equals(AppConstants.CUSTOMER_QUERY_CMD)) {
   try {
      if (message instanceof MapMessage) {
         MapMessage  queryMsg = (MapMessage) message;

         String firstName =
           queryMsg.getString(AppConstants.FIRST_NAME_FIELD);
         String lastName  =
           queryMsg.getString(AppConstants.LAST_NAME_FIELD);
         String location  =
           queryMsg.getString(AppConstants.LOCATION_PROP);
           CustDB.CustRecord customer = custDB.lookup(firstName,
           lastName, location);

         reply = constructReplyMsg(customer);
```

Linking a Reply to a Request Message

To configure a JMS request-reply message pair, you need to link the two together through application code.

Because each request message expects a reply, the JMSReplyTo header field of the request message must be set to the destination to which the reply will be sent. In this example, you set it to the `replyQueue` (of type `Queue`) declared in the RequestorApplication.java class definition, as shown in the following line of code. The message object `msg` is the request message:

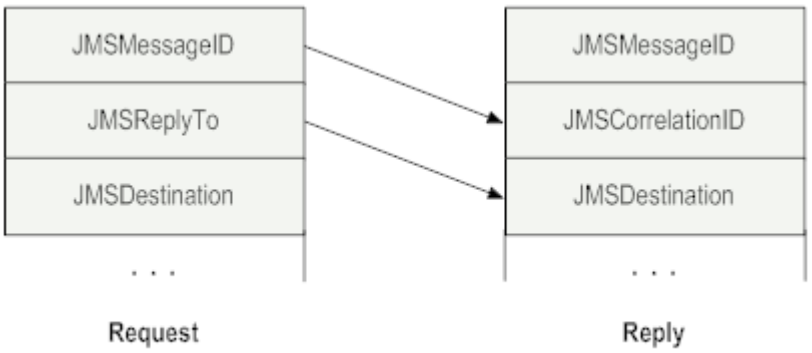
```
msg.setJMSReplyTo(replyQueue);
```

Next, in the server application, in the message listener's onMessage() method, the message ID header information of the request message is copied to a field in the reply message. This is done in a single line of code by:

- Calling the `setJMSCorrelationID()` method on the reply message object.
- Getting the value for the reply message JMSCorrelationID header field through `message.getJMSMessageID();` this sets the value of the field to the request message header ID.

  ```
  reply.setJMSCorrelationID(message.getJMSMessageID());
  ```

The following figure summarizes the relationships that must be coded between the JMS header fields of the request and reply messages:



After the request and reply messages are linked, a `send` is issued for the reply message; however, the message will not be sent until an explicit `commit` is executed. This is because the session to which the reply message belongs is defined as transacted in `ServerApplication.setup()`. The transaction is explained more fully in "Implementing a Local Transaction" on page 206.

```
sender.send(message.getJMSReplyTo(), reply);
```

In the requestor application, in `lookupCustomerInfo()`, the JMSMessageID header field value for the request message is saved for later use as a means of validating the reply.

```
String pendingMsgID = msg.getJMSMessageID();
```

The method then validates that the reply is linked to the proper request message (checks to see which reply it is) and initiates further processing of the reply.

```
while (true)
{
    Message replyMessage = receiver.receive(REPLY_WAIT_TIME);
    ...
    if(!pendingMsgID.equals(replyMessage.getJMSCorrelationID()))
    ...
 }
```

Implementing a Local Transaction

The example contains a local transaction consisting of the following elements:

- Receipt of a customer inquiry message
- Acknowledgment of the customer inquiry message
- Publishing of a reply

The code for setting up the local transaction is in the server application, and is shown below:

```
session    = conn.createSession(true, 0);
sender     = session.createProducer(null);
...
receiver   = session.createConsumer(destination);
```

Here, a session is created transactionally by setting the first parameter in `conn.createSession()` to a value of `true`. Since the message sender and the message receiver are both created from the same session, they share the same transactional context.

The transaction is executed in `onMessage()`. There, a `session.commit()` is issued, which means that if the local transaction is successful, receipt of the original message is acknowledged (because the message consumer is part of the transaction) and the reply is published (because the message producer is part of the transaction).

```
session.commit();
```

When the transaction commits, the inbound (request) message is acknowledged and removed from the JMS message queue, and the outbound (reply) message is published. If the transaction cannot commit and is rolled back, the inbound message is returned to the message queue and no message acknowledgment or publishing takes place.

JMS transactions of a larger scope, such as those that include operations on databases, must be implemented in the context of a distributed transaction, using the JMS XA interfaces and a distributed transaction manager. For more information, see "Java Transaction API Support (JMS Clients Only)" on page 17.

Using a Temporary Queue

JMS applications typically use temporary queues to store messages that are only needed for the duration of a session. As soon as the session ends, messages on the temporary queue are deleted.

The requestor application in this example uses a temporary queue to store reply messages. The following lines of code demonstrate the usage of a temporary queue for reply messages:

```
MessageConsumer receiver;
   Queue           replyQueue;

   ...
   replyQueue = session.createTemporaryQueue();
   receiver   = session.createConsumer(replyQueue);
```

Using a Message Selector

The server application implements a message selector that filters messages based on a customer's geographical region. The message listener receives only messages where the customer region matches that specified by the selector.

Message selector values are specified in property fields in the message header. In this example, a "location" value specified in `the` requestor application represents such a property field in a customer request message:

```
msg.setStringProperty(AppConstants.LOCATION_PROPERTY, location);
```

You enter the message selector value (*currentRegion*) as a command line argument at server startup. If you enter a value of "All," the server receives a message irrespective of region (that is, message selection is disabled). If you enter the value of a valid region ("East" or "West"), the server enables message selection, allowing the listener to receive only messages from the specified region.

```
if (currentRegion.equals(AppConstants.ALL_REGIONS)) {
        receiver = session.createConsumer(destination);
   } else {
        receiver = session.createConsumer(destination,
                   "location = '" + currentRegion + "'", false);
}
```

You specify the valid regions in `determineRegion()` in the server application.

```
private static String determineRegion(String regionName)
    {
        String region;
        if (regionName.equalsIgnoreCase("East")) {
            region = AppConstants.EAST_REGION;
        } else if (regionName.equalsIgnoreCase("West")) {
            region = AppConstants.WEST_REGION;
        } else if (regionName.equalsIgnoreCase("All")) {
            region = AppConstants.ALL_REGIONS;
        ...
        return region;
    }
```

Implementing Messaging on a Second Thread

In addition to receiving customer inquiry messages on an asynchronous messaging thread, the server application also receives administrator shutdown commands on the main thread.

The administrator shutdown command is sent from an admin client to the server. To control which clients can send the shutdown command to the server, you can create a separate client group that has permission to send messages to the admin Queue. You use the JMSAdmin command-line tool to create this group at server application and admin client startup.

The setup code for the shutdown command contains its own set of messaging objects (session, message consumer, and destination). The setup and initialization (factory creation and lookup) code portions are similar to those of the sender-receiver application in "A Basic JMS Sender-Receiver Client" on page 173.

```
Session         adminSession;
   MessageConsumer adminReceiver;
   Destination     adminDestination;
        ...
        adminSession = conn.createSession(false,
                                Session.AUTO_ACKNOWLEDGE);
        adminReceiver = adminSession.createConsumer(adminDestination);
```

A loop in `serverRun()` control server operation and shutdown, as shown below. Message reception for the shutdown command occurs via a synchronous `receive()` method, unlike the asynchronous message listener that receives customer inquiries on the other messaging thread.

```
void serverRun()
   {
      boolean running = true;
      while (running) {
         try {
            Message msg = adminReceiver.receive();
            String  command =
```

```
                msg.getStringProperty(AppConstants.ADMIN_CMD_PROP);

            ...             }
            if (command.equals(AppConstants.ADMIN_CMD_SHUTDOWN))  {
                conn.stop();
                session.setMessageListener(null);
                running = false;
            }
        ...
    }
```

Administrative Setup Commands

Following are the JMSAdmin commands needed (using generalized labels to identify command parameters) to set up the server and requestor applications.

```
jmsadmin -p jndi.properties

  bind cf <serverFactory> with group=<serverGroup>
  bind cf <requestorFactory> with group=<requestorGroup>
  bind topic <requestTopic> with topicName=<requestTopicEventType>
  bind queue <serverAdmin> with queueName=<serverAdminQueue>
  connect

  create group <serverGroup>
  create group <requestorGroup>
  create group <serverAdminGroup>
  create topic for <requestTopic>
  create queue for <serverAdmin> using <serverFactory>
  permit group <serverGroup> to send JMS::Temporary::Queue
  permit group <requestorGroup> to receive queue JMS::Temporary::Queue
  permit group <serverGroup> to subscribe <requestTopicEventType>
  permit group <requestorGroup> to publish <requestTopicEventType>
  permit group <serverGroup> to receive queue <serverAdminQueue>
  permit group <serverAdminGroup> to send to queue <serverAdminQueue>
  end
```

Binding the Administered Objects to JNDI

The JMSAdmin commands for configuring JNDI and binding the connection factories and the destinations to a JNDI context follow the same basic pattern as those used in the basic example in "Administrative Setup Commands" on page 178.

When binding the connection factories for the server and requestor applications, use the optional `with` keyword with the `group` parameter to create their respective Broker client groups. Creating client groups on the Broker is necessary so that access permissions can subsequently be granted to the groups.

```
bind cf myServerFactory with group=myServerGroup
bind cf myRequestorFactory with group=myRequestorGroup
```

Note how the `bind` commands are issued for a generic connection factory object, which corresponds to the use of generic connection factory objects specified in the server-requestor application code (that is, the type of factories are resolved at run time).

The following `bind` commands create a JNDI context for the topics (customer request messages), and one for the queue object on the server, which will receive an administrator shutdown command from the admin client.

```
bind topic customerSample with topicName=JMSSamples::requestMessage
bind queue myAdminQueue with queueName=myServerAdminQueue
```

Creating the Administered Objects

The following commands create server and requestor Broker groups, and the topic and queue objects for which JNDI bindings were previously established.

```
create group myServerGroup
create group myRequestorGroup
create topic for customerSample
create queue for myAdminQueue using myServerFactory
```

For more information, see "Create Group" on page 96, "Create Queue" on page 96, and "Create Topic" on page 98.

Assigning Group Permissions

The following JMSAdmin commands assign Broker permissions for:

- The server application to send reply messages to a JMS temporary queue.

  ```
  permit group myServerGroup to send JMS::Temporary::Queue
  ```

- The requestor application to receive reply messages from the JMS temporary queue.

  ```
  permit group myRequestorGroup to receive queue JMS::Temporary::Queue
  ```

- The server application to subscribe to the customer request topic published by the requestor application.

  ```
  permit group myServerGroup to subscribe JMSSamples::requestMessage
  ```

- The requestor application to publish the customer request topic.

  ```
  permit group myRequestorGroup to publish JMSSamples::requestMessage
  ```

For more information, see "Permit" on page 141.

Client Startup

To start the server, execute a statement such as the following from the command line:

```
java wmjms.ServerApplication East myServerFactory
   customerSample myServerAdmin
```

where the first command line argument (in this case, `East`) is the region to be placed in the message selector, `myServerFactory` represents the name of the server connection factory, `customerSample`

is the name of the customer request topic to be published, and `myServerAdmin` is the JNDI lookup name for the message consumer for server admin commands.

To start the requestor application, execute a statement such as the following from the command line:

```
java wmjms.RequestorApplication myRequestorFactory customerSample
```

where myRequestorFactory is the name of the requestor connection factory, and customerSample is the name of the request topic to be published.

To run the admin client application that shuts down the server application, execute a statement such as the following from the command line:

```
java wmjms.AdminClient shutdown
```

Application Configurations

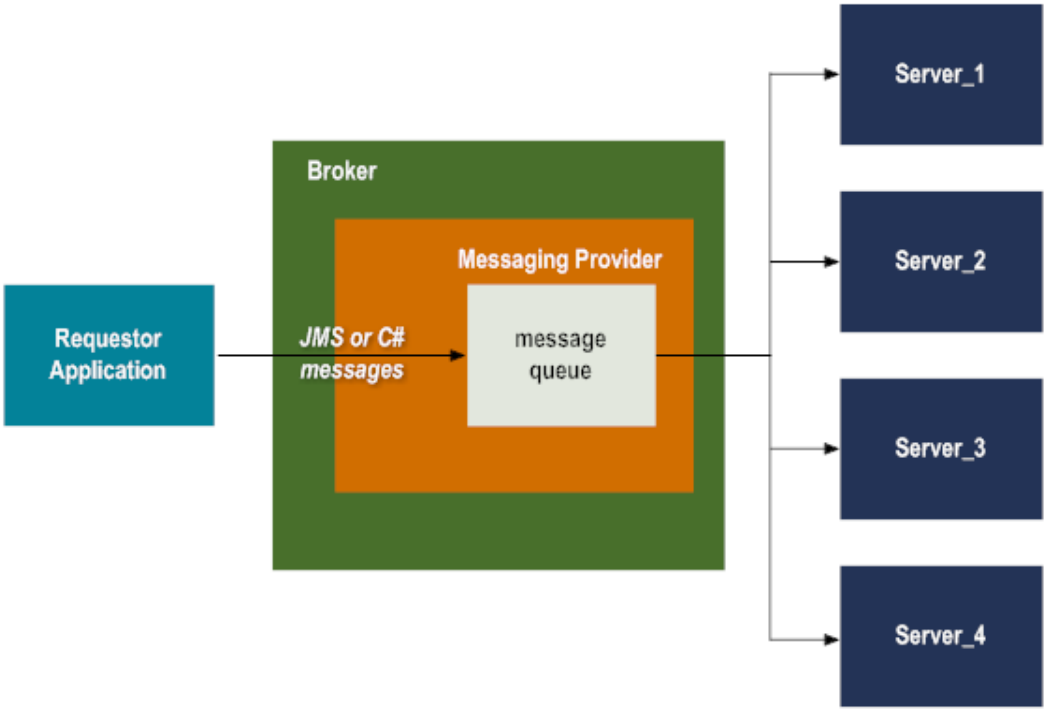
You can deploy JMS (or C#) clients like the one described in this example in a number of different configurations. Following are descriptions of more complex configurations in which you can deploy webMethods messaging clients; they are included to show how your messaging applications can be scaled to different levels.

Shared-State Configuration

The following figure shows a deployment of the server-requestor example with a "shared-state" Broker configuration (several server applications are shown in the figure). In a shared-state configuration, multiple clients share a single Broker message queue, with the Broker managing the cursor of the message queue relative to each of the clients.

In a configuration such as the one shown, a single server must process a complete topic from the requestor application, because a shared state configuration does not support the parts of a multipart message being processed by different servers. However, in this scenario, it is possible to realize the gain in efficiency from processing several different messages in parallel.

For more information about the Broker shared state feature, see *Administering webMethods Broker*.



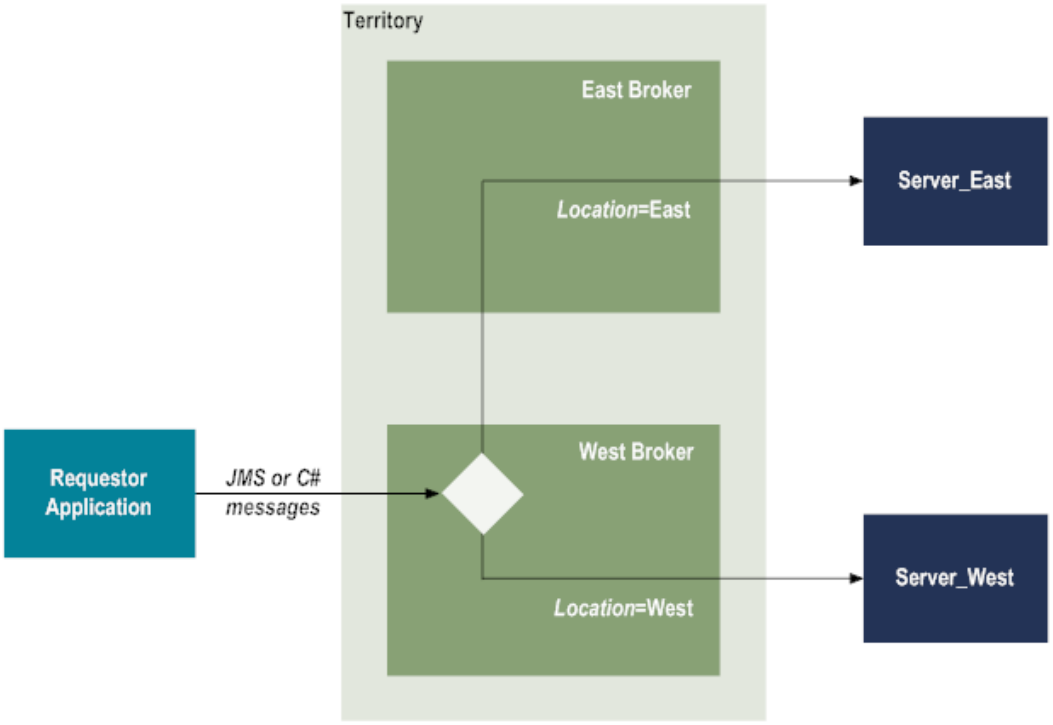
Broker Territory Configuration

The next configuration shows a variation of the server-requestor application with two Brokers joined in a territory, a configuration where multiple Brokers share information.

The requestor application connects to the primary Broker, in the Western region; however, there is a second server connected to a Broker in the Eastern region. Requests for information for customers in the Eastern region are routed via a message selector to the Broke server servicing that region.

For this type of configuration to work with the sample application, you would need to create a more realistic database setup than the in-memory database used in the chapter example. You would need databases for both the Eastern and Western regions, and the databases would share the same schemas.

For more information about Broker territories, see *Administering webMethods Broker*.

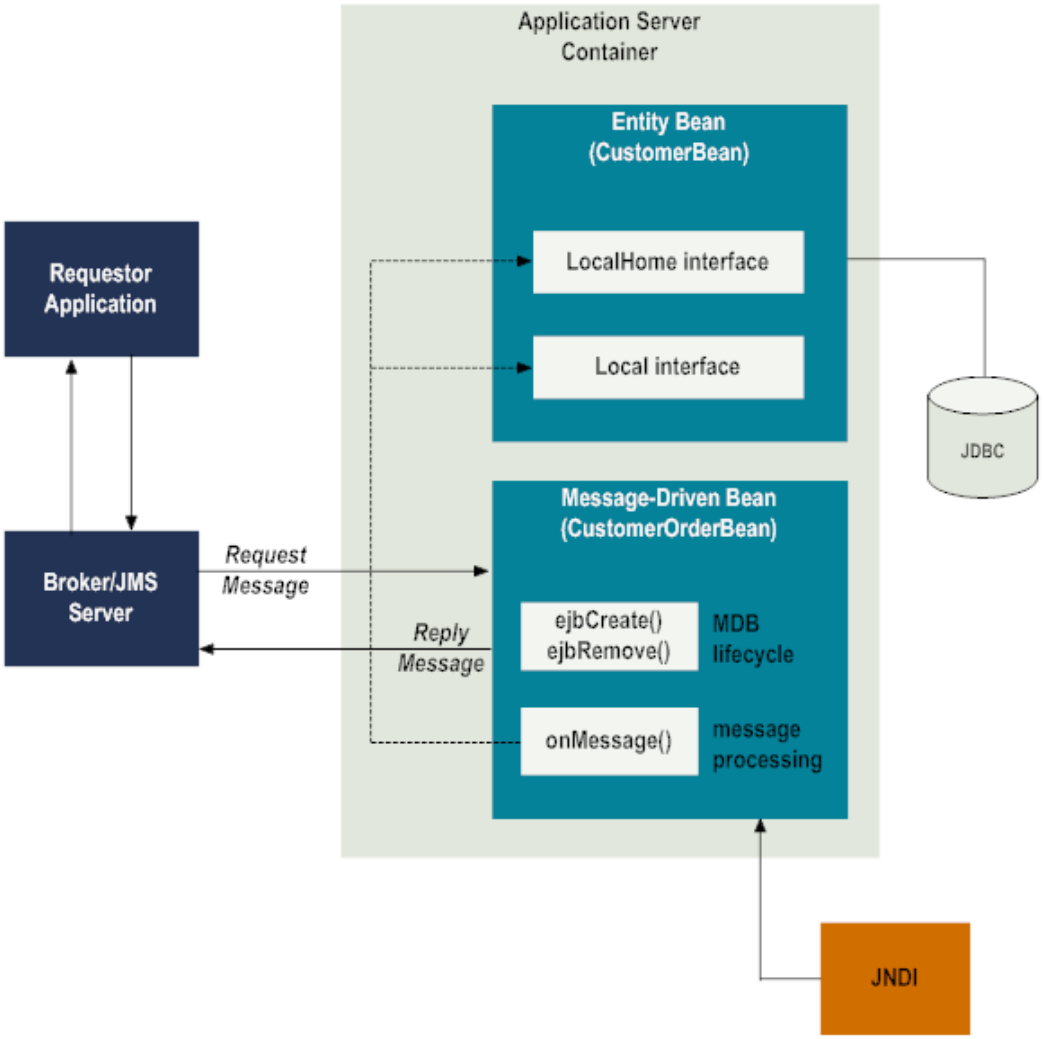


Message-Driven Bean Configuration

For JMS clients, you can use webMethods Broker as a JMS provider as part of your application server messaging solution.

The configuration outlined here contains the same functionality as the customer request part of the standalone messaging example in this chapter; however, the client is configured to run as a container application. Messaging is managed by a message-driven bean (MDB), an entity bean, and the application server container.

The following figure identifies the high-level components of such a client:



- **Message-driven bean (MDB)**. The MDB encapsulates all the actions that are taken as a result of the asynchronous receipt of a message requesting customer information. Once a message is received, the MDB's `onMessage()` method is invoked. The `onMessage()` method, which runs inside the scope of a transaction, removes the incoming message from the message queue, updates the database, and sends a reply to the requesting Broker client. `ejbCreate()` and `ejbRemove()` are methods that control the life cycle of the MDB.
- **Entity bean**. The entity bean controls the mapping and sending of data to and from the application server database. In this client, the entity bean and MDB are called from within the same EJB container system and executed within the same JVM. Therefore, only the local client interfaces `LocalHome` and `Local` are required.
- **Application server container**. The application server container manages the entity bean and MDB components. If container-managed persistence (CMP) is specified for the application server, the container manages many operations automatically that would normally be handled programmatically in a standalone JMS client. For example, the container manages all update, insert, and delete database operations needed to communicate with the application server database, as well as transactional operations such as commits and rollbacks.

- **Database**. A standard database managed by the application server, such as a JDBC database.

For more information about configuring application servers for use with the webMethods Broker that is used as a JMS provider, see " webMethods Messaging Publisher Reconnect" on page 251.

11 C# Messaging Clients

Overview

Using sample applications, this chapter explains the basics of writing C# .NET messaging applications that work with webMethods Broker. The chapter's goal is to familiarize application developers with the key constructs and programming details necessary for building webMethods Broker C# messaging clients.

The webMethods Broker C# Messaging API (henceforth referred to as the C# API) is based on the JMS API; the two object models closely resemble each other. However, some important implementation differences exist. These differences stem from the fact that C# messaging clients are .NET Framework applications that execute exclusively in a Windows environment. For example, there is no built-in support for constructs such as JNDI and Java EE. Besides the many similarities, the chapter highlights the important implementation differences that distinguish between the two APIs.

The chapter examines two sample C# clients. These two samples are similar, in design and functionality, to the JMS samples described in "A Basic JMS Sender-Receiver Client" on page 173 and "JMS Request-Reply Application with a Message Selector" on page 181, and is done for comparison purposes. These similarities demonstrate how the same JMS messaging constructs described in those chapters are coded using C# .NET and the webMethods C# Messaging API, instead of Java/JMS and the JMS API.

Important:
C# Messaging API support Microsoft .NET Framework version 3.5. For using C# Messaging API with later versions of Microsoft .NET Framework, refer the Microsoft documentation.

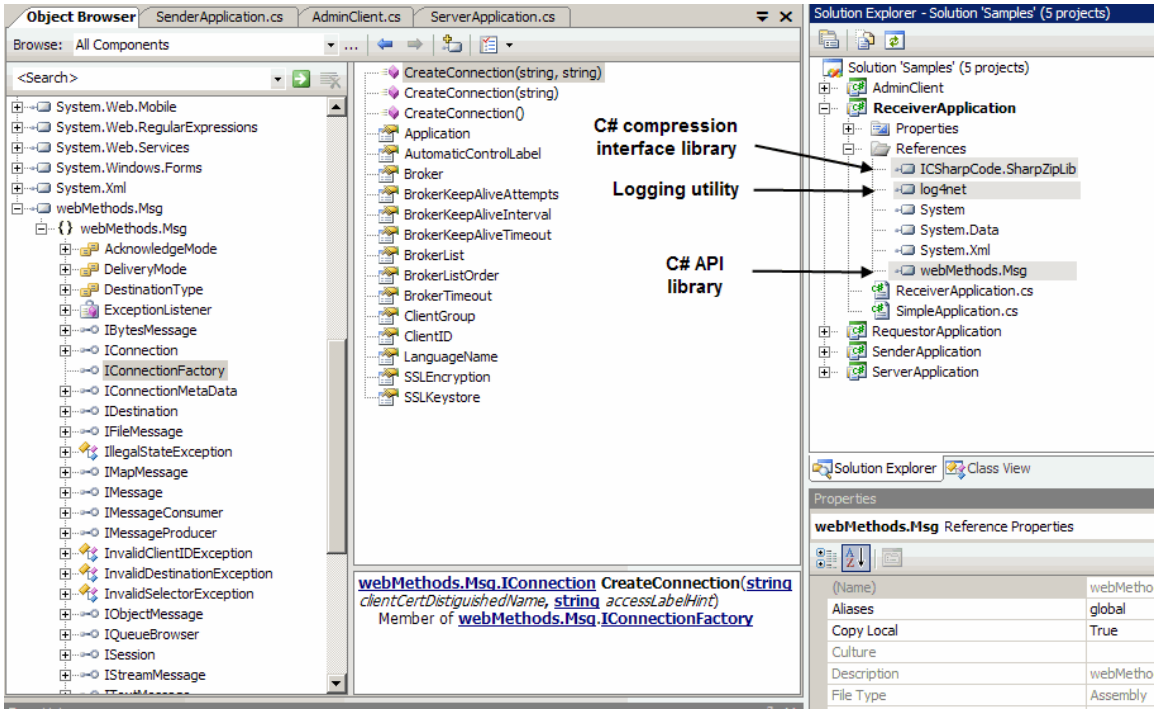
webMethods Broker C# Messaging Library

webMethods C# messaging clients must reference the dynamic-link library (DLL) `webmethods.msg.dll`, which implements the C# API. Make sure that the Samples solution contains that reference; if not, use the menu **Project > Add Reference** to add the C# library to the solution in Visual Studio.

The following figure shows the Visual Studio Solution Explorer with the Samples solution and webMethods C# library installed. The `CreateConnection(string, string)` method of `IConnectionFactory` from the C# library is displayed in the Object Browser. In addition, the following references from the Receiver application are called out and highlighted in gray:

- The C# compression interface library `ICSharpCode.SharpZipLib`
- The standard logging utility `log4net`
- The C# API library `webMethods.msg`

Visual Studio Object Browser View of IConnectionFactory and CreateConnection(string, string)



Sender-Receiver Application

The sender-receiver C# sample works the same as the JMS sample described in "Sender-Receiver Application Components" on page 174. Both are "bare bones" examples designed to illustrate messaging basics. In both samples, a sender application connects to webMethods Broker and sends a single text message to a destination. A receiver application consumes the message and then shuts itself down.

Application Classes

The C# version of the Sender-Receiver example contains three classes, described in the following table.

Class	Description
SimpleApplication.cs	Base class for Sender.cs and Receiver.cs. Contains the code for initializing and looking up administered objects in LDAP.
SenderApplication.cs	Contains the code for creating the connection, session, and message producer objects needed to send the message.
ReceiverApplication.cs	Contains the code for creating the connection, session, and message consumer objects needed to receive the message.

Binding Administered Objects in C#

In JMS, administered objects are bound and created in a JNDI namespace. In most cases, you configure a JNDI provider to work with your JMS provider, then bind the objects administratively in the JNDI namespace.

With webMethods C# messaging, you *must* bind the administered objects in an LDAP namespace by using My webMethods or the jmsadmin command-line tool (version 7.1 and later).

This sample assumes that you have already bound the administered objects to LDAP (you enter the required LDAP information through command-line arguments when starting the sender application). The sample's base class SimpleApplication.cs contains the look-up code needed to return the connection factory bound to LDAP.

```
class SimpleApplication
    {
       protected IConnectionFactory connectionFactory;
       protected IDestination destination;
              protected String ldapURL;
       protected String ldapUserName;
       protected String ldapPassword;
             protected SimpleApplication() { }
       public bool InitializeAdminObject(String _ldapURL,
                                         String _ldapUserName,
                                         String _ldapPassword,
                                         String factoryName)
       {
           this.ldapURL =      _ldapURL;
           this.ldapUserName = _ldapUserName;
           this.ldapPassword = _ldapPassword;
           connectionFactory = LookupConnectionFactory(factoryName);
           return connectionFactory != null;
        }
  ...
```

Sending Messages in C#

C# messaging objects are coded like their JMS counterparts (for more information, see "Coding Messaging Objects" on page 176). Following is the C# code used in the sender application (SenderApplication.cs) to create the connection, session, and message producer objects, and send a message.

```
public void SendAMessage(String messageText)
   {
     IConnection conn = null;

     try
     {
       conn = connectionFactory.CreateConnection();
       ISession session = conn.CreateSession(false, AcknowledgeMode.Auto);
       IMessageProducer sender = session.CreateProducer(destination);

       ITextMessage msg = session.CreateTextMessage();
       msg.Text = messageText;
```

```
        Console.WriteLine("Sending message '" + messageText + "'");
        sender.Send(msg);
        ...
```

Receiving Messages in C#

The following lines of C# code from ReceiverApplication.cs shows how to set up message reception (for more information, see "Coding Messaging Objects" on page 176).

```
public void ReceiveAMessage()
   {
     IConnection conn = null;
     try
     {
      conn = connectionFactory.CreateConnection();
      ISession session = conn.CreateSession(false, AcknowledgeMode.Auto);
      IMessageConsumer receiver = session.CreateConsumer(destination);

      conn.Start();
      IMessage msg = receiver.Receive();

     ...}
}
```

You define the connection and session objects the same as for the message sender application. However, the message receiver defines a message consumer rather than a message producer. The message consumer is created calling `createConsumer()` on the session.

Request-Reply Application: Description

This request-reply application manages user requests for customer information. This C# example is similar to the JMS sample of "JMS Request-Reply Application with a Message Selector" on page 181.

Application Components

A *requestor* application connects to a webMethods Broker and sends a request for information about a customer (contained in a C# message) to a server application. Using information in the request message, the server application builds a query, which it executes against a customer database.

The server retrieves the query results containing the customer information, and maps the data to a C# reply message. The server then sends the reply to the requestor application. The requestor application processes the reply message, extracts the customer data, and sends the customer data to the console for viewing.

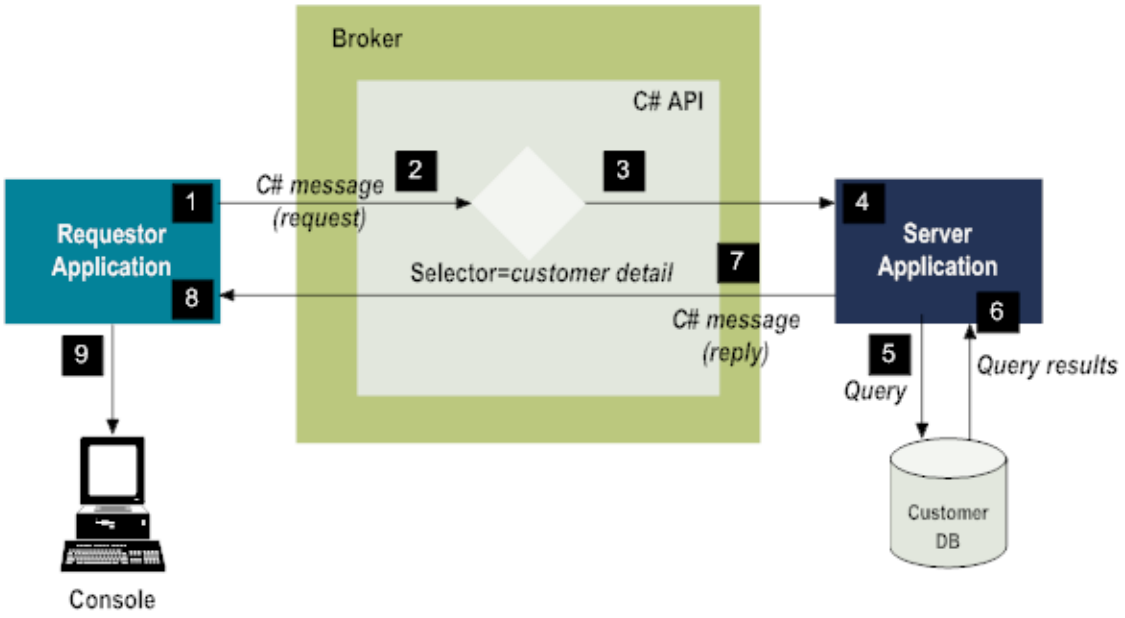
Using a Message Selector

The application uses a *message selector* to query for detailed information about a specific customer.

For more information, see "Message Consumer" on page 32.

Message and Data Flow

The following figure shows the C# client's message and data flow.



The numbers in the figure correspond to the sequence of messaging steps described below:

Step	Description
1	The requestor application builds a C# message containing a lookup request for customer information.
2	The requestor sends a message to the server application. The message contains the information needed to lookup a customer record.
3	The message selector ensures that only messages from the selected record are passed to the server.
4	The server receives the message requesting detailed customer information (the complete customer record).
5	The server parses the request message, builds a database query, and executes the database query against a simple in-memory database containing customer records.
6	The server maps the query results to a reply message and links the reply message to the original request.
7	The server sends the reply back to the requestor application.
8	The requestor extracts the customer data from the reply.

Step	Description
9	The requestor sends the customer data to the console.

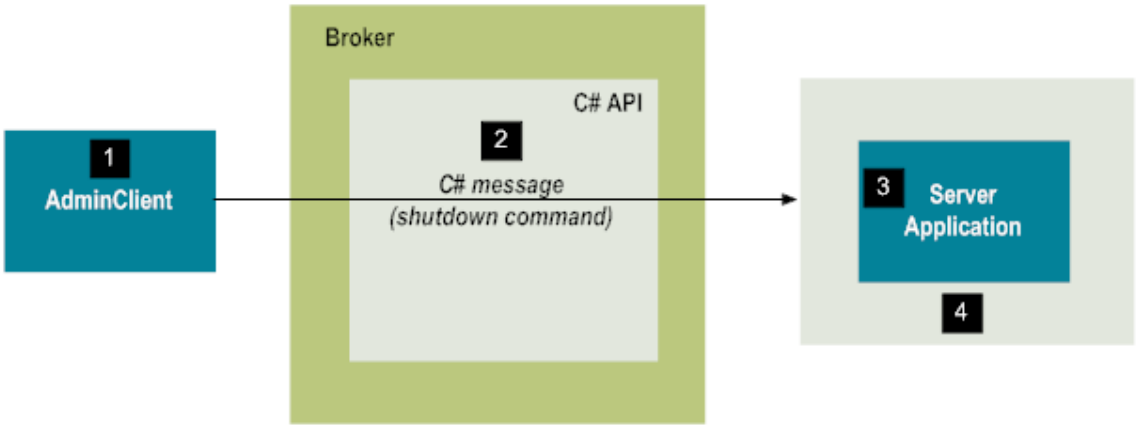
Using a Second Messaging Thread

In the C# messaging API, each session supports a single thread. For an application to support the reception of concurrent messages, the application must contain a session (thread context) for each message receiver.

The customer inquiry application constitutes one messaging thread: messages are sent from a requestor application and received asynchronously by a server. Adding a second thread requires that another session be created and message reception handled on that thread.

To demonstrate use of a second messaging thread, the example includes code for sending and receiving a single administrator command. The command is sent by an administrator client (AdminClient.cs) and received by the server application. When the server receives the command, it shuts itself down.

The following figure shows the execution flow for the example's second message reception thread.



The numbers in the figure correspond to the steps that occur when the `shutdown` command is issued:

Step	Description
1	The AdminClient application is started with the server `shutdown` command entered on the command line.
2	The command is copied to a C# message and sent to the server.
3	The server extracts the command string from the message.
4	The server executes the command and shuts down.

Application Code

This section describes the application code, focusing on the application's C# messaging functionality. The code descriptions focus on the following areas:

- Implementing an asynchronous message listener. The asynchronous message listener receives customer inquiry messages and is installed on the server.
- Linking a reply to a request message. The reply message, which contains the results of the customer inquiry, and is linked through message header fields to the original request.
- Implementing a local transaction. The request message for customer information, and its reply message and acknowledgment are grouped into a local transaction.
- Using a message selector. The message selector determines whether a customer request will be received, based on the Boolean expression specified by the selector.
- Receiving messages on a second thread. The example includes a second session and a message consumer with a synchronous `receive()` method that manages the reception of an administrator message on a second thread.

Implementing an Asynchronous Message Listener

The ServerApplication class implements a single asynchronous message listener to receive customer inquiries. The message listener is implemented through the C# `MessageListener` and `MessageConsumer` interfaces.

```
class ServerApplication : samples.RequestReply.SimpleApplication
    {
        IConnection      conn = null;
        ISession         session;
        IMessageProducer sender;
        IMessageConsumer receiver;
        ISession         adminSession;
        IMessageConsumer adminReceiver;
        . . .

        receiver.MessageListener = new MessageListener(OnMessage);
        conn.Start();
```

The `MessageListener` delegate receives the incoming customer request messages asynchronously from the requestor application.

Building the Query (Request) Message

In the C# API, as in JMS, the `MessageListener` delegate receives messages asynchronously. In this example, the delegate `OnMessage` is called whenever the server receives a customer request message. This implementation of `ServerApplication.OnMessage()` copies, from the request message, the customer information needed to make the database query. It then issues the query and builds the reply message, calling `ConstructReplyMessage()` to map data from the customer record to the reply message fields. The following code from `OnMessage()` shows these steps.

```
{
      try {
          IMessage reply;
         String  command = message.GetStringProperty
                             (ApplicationConstants.COMMAND_PROPERTY);
          Console.WriteLine("OnMessage::Received a command " + command);
          if (command.Equals(ApplicationConstants.CUSTOMER_QUERY_COMMAND,
                       StringComparison.InvariantCultureIgnoreCase)) {
              try {
                  if (message is IMapMessage) {
                      IMapMessage  queryMsg = (IMapMessage) message;

                      String firstName = queryMsg.GetString
                              (ApplicationConstants.FIRST_NAME_FIELD);
                      String lastName = queryMsg.GetString
                              (ApplicationConstants.LAST_NAME_FIELD);
                      String location = queryMsg.GetStringProperty
                              (ApplicationConstants.LOCATION_PROPERTY);
                      CustomerDB.CustomerRecord customer =
                         customerDb.lookup(firstName, lastName, location);

                     Console.WriteLine("OnMessage::QueryCommand with
                              FirstName=" + firstName + ";LastName=" +
                              lastName + ";Location=" + location);
                      reply = ConstructReplyMsg(customer);

               }      ...
```

Linking a Reply to a Request Message

To configure a request-reply message pair, you need to link the two together through application code.

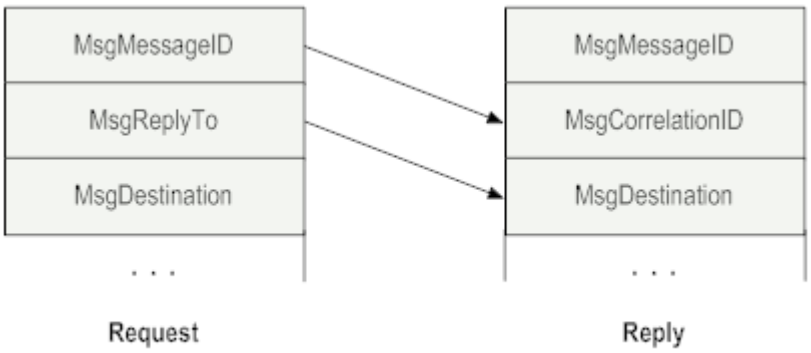
Because each request message expects a reply, the MsgReplyTo field of the request message must be set to the destination to which the reply will be sent. In this example, you set it to the `replyQueue` (of type `Queue`) declared in the RequestorApplication class definition, as shown in the following line of code. The message object `message` is the request message:

```
               reply.MsgCorrelationID = message.MsgMessageID;
               Console.WriteLine("OnMessage: Sending reply");
               sender.Send(message.MsgReplyTo, reply);
               ...
```

Next, in the server application, in the message listener's `OnMessage` delegate, the message ID header information of the request message is copied to a field in the reply message.

```
reply.MsgCorrelationID = message.MsgMessageID;
```

The following figure summarizes the relationships that must be coded between the request and reply messages:



After the request and reply messages are linked, a `send` is issued for the reply message; however, the message will not be sent until an explicit `commit` is executed. This is because the session to which the reply message belongs is defined as transacted in `ServerApplication.OnMessage()`.

```
sender.send(message.MsgReplyTo, reply);
```

In the requestor application, in `LookupCustomerInfo()`, the MsgMessageID header field value for the request message is saved for later use as a means of validating the reply.

```
String pendingMessageID = msg.MsgMessageID;
```

The method then validates that the reply is linked to the proper request message (checks to see which reply it is) and initiates further processing of the reply.

```
while (true)
   {
       IMessage replyMessage = receiver.Receive(REPLY_WAIT_TIME);
       ...
       if (!pendingMessagID.Equals(replyMessage.MsgCorrelationID,
                   StringComparison.InvariantCultureIgnoreCase)) {
           continue;   }
       if (replyMessage is IMapMessage) {
           ProcessDetailedReply((IMapMessage) replyMessage);  }
       else if (replyMessage is ITextMessage) {
           ProcessErrorReply((ITextMessage) replyMessage);
   }
```

Implementing a Local Transaction

The example contains a local transaction consisting of the following elements:

- Receipt of a customer inquiry message.
- Acknowledgment of the customer inquiry message.
- Publishing of a reply.

The code for setting up the local transaction is in the server application, and is shown below:

```
session    = conn.createSession(true, AcknowledgeMode.Auto);
 sender     = session.createProducer(null);
 ...
```

```
  receiver   = session.createConsumer(destination);
```

Here, a session is created transactionally by setting the first parameter in `conn.createSession()` to a value of `true`. Since the message sender and the message receiver are both created from the same session, they share the same transactional context.

The transaction is executed in `OnMessage()`. There, a `session.commit()` is issued, which means that if the local transaction is successful, receipt of the original message is acknowledged (because the message consumer is part of the transaction) and the reply is published (because the message producer is part of the transaction).

```
session.Commit();
```

When the transaction commits, the inbound (request) message is acknowledged and removed from the message queue, and the outbound (reply) message is published. If the transaction cannot commit and is rolled back, the inbound message is returned to the message queue and no message acknowledgment or publishing takes place.

Using a Message Selector

The server application implements a message selector that filters messages based on a customer's geographical region.

You enter the message selector value (*currentRegion*) as a command line argument at server startup. If you enter a value of "All," the server receives a message regardless of region (that is, message selection is disabled). If you enter the value of a valid region ("East" or "West"), the server enables message selection, allowing the listener to receive only messages from the specified region.

```
if (currentRegion.Equals(ApplicationConstants.ALL_REGIONS,
                         StringComparison.InvariantCultureIgnoreCase)) {
     receiver   = session.CreateConsumer(destination);
  } else {
     receiver   = session.CreateConsumer(destination, "location = " +
                                         currentRegion, false);
  }
```

You specify the valid regions in `determineRegion()` in the server application.

```
private static String DetermineRegion(String regionName)
       {
          String region;
          if (regionName.Equals("East",
                         StringComparison.InvariantCultureIgnoreCase)) {
              region = ApplicationConstants.EAST_REGION;  }
          else if (regionName.Equals("West",
                         StringComparison.InvariantCultureIgnoreCase)) {
              region = ApplicationConstants.WEST_REGION;  }
          else if (regionName.Equals("All",
                         StringComparison.InvariantCultureIgnoreCase)) {
              region = ApplicationConstants.ALL_REGIONS;
          ...
          return region;
       }
```

Implementing Messaging on a Second Thread

In addition to receiving customer inquiry messages on an asynchronous messaging thread, the server application can receive administrator shutdown commands on the main thread.

The administrator shutdown command is sent from an admin client to the server. To control which clients can send the shutdown command to the server, you can create a separate client group that has permission to send messages to the admin Queue. You use the JMSAdmin command-line tool to create this group at server application and admin client startup.

```
    ISession         adminSession;
    IMessageConsumer adminReceiver;
    IDestination     adminDestination;
        ...
    adminSession = conn.CreateSession(false, AcknowledgeMode.Auto);
    adminReceiver = adminSession.CreateConsumer(adminDestination);
```

A loop in `serverRun()` controls server operation and shutdown, as shown below. Message reception for the shutdown command occurs via a synchronous `receive()` method, unlike the asynchronous message listener that receives customer inquiries on the other messaging thread.

```
void ServerRun()
    {
      bool running = true;
      while (running) {
         try {
           IMessage msg = adminReceiver.Receive();
           String  command = (msg.GetStringProperty
                              (ApplicationConstants.ADMIN_COMMAND_PROPERTY);
           Console.WriteLine("Received a command: " + command);
           if (command == null) {
               continue;  }

           if (command.Equals(ApplicationConstants.ADMIN_COMMAND_SHUTDOWN,
                 StringComparison.InvariantCultureIgnoreCase))  {
               Console.WriteLine("Shutting down");
               conn.Stop();
               running = false;  }
       } ...
}
```

Administrative Setup Commands

Before running the C# samples, you can bind the administered objects to LDAP and create the objects. Use the JMSAdmin command-line utility to perform these bind and create operations (see "JMSAdmin Command Reference" on page 57). To determine which objects to bind and create, view the sample code and examine the command-line arguments for `Main()`, which include the look-up commands.

When binding the connection factories for the server and requestor applications, use the optional `with` keyword with the `group` parameter to create their respective Broker client groups. Creating

client groups on the Broker is necessary so that access permissions can subsequently be granted to the groups.

```
bind cf myServerFactory with group=myServerGroup
bind cf myRequestorFactory with group=myRequestorGroup
```

Note how the `bind` commands are issued for a generic connection factory object, which corresponds to the use of generic connection factory objects specified in the server-requestor application code (that is, the type of factories are resolved at run time).

Creating the Administered Objects

The following commands create server and requestor Broker groups, and the topic and queue objects for which LDAP bindings were previously established.

```
create group myServerGroup
create group myRequestorGroup
create topic for customerSample
create queue for myAdminQueue using myServerFactory
```

Assigning Group Permissions

The following JMSAdmin commands assign Broker permissions for:

- The server application to subscribe to the customer request topic published by the requestor application.

  ```
  permit group myServerGroup to subscribe requestMessage
  ```

- The requestor application to publish the customer request topic.

  ```
  permit group myRequestorGroup to publish requestMessage
  ```

For more information, see "Permit" on page 141.

Client Startup

Use Visual Studio to build executables for the three client application comprising the solution.

To start the server, execute a statement such as the following:

```
ServerApplication East myServerAdmin
  customerSample myServerFactory
```

where the first command line argument (in this case, `East`) is the region to be placed in the message selector, `myServerAdmin` represents the name of the server connection factory, `customerSample` is the name of the customer request topic to be published, and `myServerFactory` is the LDAP lookup name for the message consumer for server admin commands.

To start the requestor application, execute a statement such as the following:

```
RequestorApplication myRequestorFactory customerSample
```

where `myRequestorFactory` is the name of the requestor connection factory, and `customerSample` is the name of the request topic to be published.

To run the admin client application that shuts down the server application, execute a statement such as the following:

```
AdminClient shutdown
```

A webMethods Naming Service for JMS: Configuration Settings and Properties

Overview

This appendix provides configuration information about the webMethods Naming Service for JMS. The appendix also describes this provider's properties.

You can configure the provider with the webMethods Broker JMSAdmin command-line tool (see "The JMSAdmin Command-Line Tool" on page 45) or through the Broker user interface (for more information, see *Administering webMethods Broker*).

What Is the webMethods Naming Service for JMS?

webMethods Naming Service for JMS is suitable for storing only the Broker JMS administered objects such as connection factories, queues, and topics. The client-side library (wm-jmsnaming.jar) for the Naming Service is installed with the webMethods Client for JMS. The Broker acts as the JNDI naming server and uses native Broker structures to bind and store JMS administrative objects.

Note:
The webMethods Naming Service for JMS is provided as a convenience to enable fast, out-of-the box JNDI support for JMS development, testing, and production purposes. It is not meant for use as a repository for anything other than JMS administered objects.

Configuring the Provider

Configuring the webMethods Naming Service for JMS includes gathering some location information as well as making changes to the CLASSPATH.

Before Configuring the Provider

Before configuring the Naming Service, take the following steps:

1. Locate the wm-jmsnaming.jar file on the machine where you installed the webMethods Messaging client software. This file resides in the *Software AG_directory* \common\lib directory.

2. Obtain the URL of the Broker that will act as the JNDI naming server, and determine the name of the initial context into which you want to bind JMS administrative objects.

3. If you are using SSL, know the location of the SSL credentials (digital certificate, distinguished name, password, and so forth) that are needed to access the Broker that will act as the JNDI naming server.

Configuring the CLASSPATH

To configure the CLASSPATH for the Naming Service, add the following files to the CLASSPATH on the JMS client machine:

- *Software AG_directory* \common\lib\wm-jmsnaming.jar

- *Software AG_directory* \common\lib\wm-brokerclient.jar
- *Software AG_directory* \common\lib\wm-g11nutils.jar

For additional configuration information, see "Configuring Environment Variables" on page 19.

Using a JNDI Properties File

This section shows how to create a properties file for the webMethods Naming Service for JMS and describes the properties you use when configuring the provider.

File Syntax

JNDI Properties are specified as a set of name/value pairs as follows:

```
propertyName=value
propertyName=value
propertyName=value
.
.
.
```

Typical Properties File

A finished properties file for the Naming Service looks similar to the following example:

```
[1] java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
[2] java.naming.provider.url=ldap://adam.south.acme.com/
            ou=jmsdev,ou=pd,o=wm
[3] com.webmethods.jms.naming.clientgroup=admin
[4] java.naming.security.principal="cn=Manager,ou=jmsdev,ou=pd,o=wm"
[5] java.naming.security.credentials=JMSatAxyz
[6] com.webmethods.jms.naming.keystore=C:/BrokerSSL/MyKeystore.p12
[7] com.webmethods.jms.naming.truststore=C:/BrokerSSL/MyTruststore.jks
[8] com.webmethods.jms.naming.encrypted=True
```

At a minimum, you must provide the properties shown on lines 1 and 2. When you use JMSAdmin to perform administrative tasks with Naming Service, you must also specify the `clientgroup` property as shown on line 3. Setting the `clientgroup` property to `admin` gives you write permission for the Naming Service.

Provider Property Descriptions

The following are the properties for the Naming Service.

Note:
When you use My webMethods and go to Naming Directories, default property values are set when you add a JNDI provider and select Naming Service. You can then configure the provider as appropriate. For more information, see *Administering webMethods Broker*.

java.naming.factory.initial

Required. The factory class for the Naming Service. This factory class resides in the wm-jmsnaming.jar library. You must include wm-jmsnaming.jar in the `CLASSPATH` on the machine where the Naming Service client is running. Set this property as follows:

```
java.naming.factory.initial=com.webmethods.jms.naming.WmJmsNamingCtxFactory
```

java.naming.provider.url

Optional. The URL that points to the initial context and to the Broker that will act as the naming server and to the initial naming context. The following is the URL syntax:

```
wmjmsnaming:// [ brokerName ] [ @brokerHost [ : port ] ]  [ /context ]
```

If *brokerName* is omitted, the default Broker is used. If *brokerHost* is omitted, `localhost` is used. If *port* is omitted, 6849 is assumed. Use *context* to specify the root namespace; use the following notation:

```
rootContext::subContext::subContext
```

Each segment in context must begin with an alphabetic character followed by any combination of alphanumeric characters or the underscore character (_). Each segment of the *context* name can be up to 255 characters long. (On the Broker, *context* is represented by a document type name, which means that it must adhere to valid, document-type naming syntax.)

If you do not specify an initial context, Naming Service creates a default initial context with the name `WmJmsNamingContext`.

- Example 1

  The following example points to a Broker called "mars" on Broker host `B01East` at port `9000`. It sets the initial context to `JMS::CustMgmt`.

  ```
  java.naming.provider.url=wmjmsnaming://mars@B01East:9000/JMS::CustMgmt
  ```

- Example 2

  The following URL points to the default Broker on localhost on the default port (6849). It sets the initial context to `JMSObjects`.

  ```
  java.naming.provider.url=wmjmsnaming:::///JMSObjects
  ```

com.webmethods.jms.naming.clientgroup

Required to access webMethods Naming Service for JMS with admin tools. The client group that the Naming Service for uses to connect to the Broker naming server. To access the Naming Service through the administrative tools, set this property to the `admin` client group as shown below:

```
com.webmethods.jms.naming.clientgroup=admin
```

A client that operates as a member of the admin group has permission to write to the Broker's JNDI namespace. For more information, see the "Managing Client Groups" chapter of *Administering webMethods Broker*.

You do not need to set the `clientgroup` property for read-only access to the Naming Service (for example, to simply recall an administered object from within a JMS application). By default, Naming Service clients function as members of the "WmJmsNamingReader" client group, which has read-only access to all objects in the Broker JNDI namespace.

If you are using webMethods JNDI event listener, you must add the "Broker::Activity::EventTypeChange" document type to the client group's can-subscribe list. Otherwise, if the Broker Server restarts, the event listener will not get a remote notification, and the JNDI client will throw the following exception:

```
Cannot subscribe to document type 'Broker::Activity::EventTypeChange'.
Permission is denied.  Check the 'can subscribe' permissions in the client's
client group.
```

com.webmethods.jms.naming.keystore

*Required if the Broker that provides naming services is SSL-enabled.*The fully qualified name of the file containing the SSL certificate that the Naming Service is to use to connect to the Broker.

The following example points to the file myKeystore.p12.

```
com.webmethods.jms.naming.keystore=C:\BrokerSSL\myKeystore.p12
```

com.webmethods.jms.naming.truststore

Required if the Broker that provides naming services is SSL-enabled. The fully qualified name of the file containing the trusted root certificates that the Naming Service is to use to connect to the Broker.

The following example points to the file myTruststore.jks.

```
com.webmethods.jms.naming.truststore=C:\BrokerSSL\myTruststore.jks
```

java.naming.security.principal

Required if the Broker that provides naming services is SSL enabled. The distinguished name that the Naming Service will use to connect to the Broker. The following example sets a distinguished name:

```
java.naming.security.principal=CN=Developer 1,OU=Product Development,
      O=webMethods Inc.,L=Fairfax,ST=VA,C=US
```

java.naming.security.credentials

Required if the Broker that provides naming services is SSL enabled. The passphrase for the keystore that is specified in com.webmethods.jms.naming.keystore. The following example sets the password to `mysecurePassword`.

```
java.naming.security.credentials=mySecurePassword
```

com.webmethods.jms.naming.encrypted

A true/false flag that specifies whether traffic between the Naming Service and the Broker is encrypted.

B Broker to JMS Mappings

Overview

This appendix describes the various relationships between webMethods Broker objects and properties and the webMethods Broker used as a JMS provider. It shows the relationships between the JMS message header, properties, and body fields and their Broker document equivalents. It also lists the relationships between Broker clients, client groups, and Document Types, and how these values are used in JMS.

JMS Message Field Mappings

The following tables list the correspondences (mappings) between the fields of a JMS message and a Broker document.

JMS Message: Header Fields

The header fields of a webMethods Broker used as a JMS provider message map to the envelope fields of a Broker document. There is one exception: the value of the `JMSRedelivered` header field is set by the Broker when it hands off a message for re-delivery.

Both the `JMSDeliveryMode` and `JMSExpiration` header fields are computed values, that is, they are not copies of values but the results of calculations made on their mapped Broker document fields.

JMS Message Header Field	Broker Envelope Field
`JMSDestination`	`jms_destination`
`JMSDeliveryMode` (computed field)	`storage_type`
`JMSExpiration` (computed field)	`ttl` (If `JMSExpiration` is set to 0, which is the default value, then `ttl` is not carried to the envelope.)
`JMSPriority`	`priority` (If `JMSPriority` is set to its default value of 4, then `JMSPriority` is not carried to the envelope.)
`JMSMessageID`	`jms_message_id`(This field can be turned off at the sender side; in that case, it does not exist in the document.)
`JMSTimestamp`	`jms_timestamp`
`JMSCorrelationID`	`jms_correlationID`(If this field is not set in a JMS message, the corresponding Broker envelope field does not exist.)
`JMSReplyTo`	`jms_replyToDestination` *and* `replyTo`
`JMSType`	`jms_type`. If this field is not set in a JMS message, the corresponding Broker envelope field does not exist.
`JMSRedelivered`	Generated by the Broker

JMS Message: Property Fields

All JMS properties are in a Broker document structure named Properties (for example, a property field for use in a message selector).

JMS Message: Body Fields

JMS message body fields (data) are mapped to Broker document body fields. The following table shows the body mappings between an outbound JMS message and a Broker Document:

JMS Message Type	Broker Document Fields	Notes
`Message`	No body fields	
`BytesMessage`	bytes	Sequence of bytes
`MapMessage`	Each field in the map is a separate top-level field	Type of the field corresponds to the map field type
`TextMessage`	text	Unicode string field
`StreamMessage`	Each field in the stream is a separate top-level field. Since stream fields do not have a name, the document field name is a number prefixed by _sf_ (for example, _sf_1).	Type of the field corresponds to the map field type
`ObjectMessage`	object	Sequence of bytes

When non-JMS events are received, they are mapped to a JMS Message type according to the following algorithm:

- If there are no fields, the message type is `Message`.
- If there is a single string field, the message type is `TextMessage`.
- If there is a single byte sequence field named `object`, the message type is `ObjectMessage`.
- If there is a single byte sequence field, the message type is `BytesMessage`.
- If there are no structs or sequences, the message type is `MapMessage`.
- If there are no structs or sequences, then the entire event is encapsulated in a `BytesMessage`. The receiver can use the Broker Java API to decode the contents of the `BytesMessage` by creating an instance of `BrokerEvent` using the contents of `BytesMessage`. For more information, see `BrokerEvent.fromBinData` in the Javadoc.

Broker-to-JMS Mappings

The following tables show the relationships between various Broker objects and properties and how they are used by JMS. The relationships are grouped into tables for Broker Client Group, Broker Client, and Document Type.

For more information about the webMethods Broker objects and properties listed below, see *Administering webMethods Broker* and the *webMethods Broker Administration Java API Programmer's Guide*. For information about the webMethods API for JMS, see the Javadoc available on the Software AG Documentation website at http://documentation.softwareag.com. It is not included in this programming guide.

Broker Client Group

Broker	JMS	Notes
Name	WMConnectionFactory.setClientGroup() WMConnectionFactory.getClientGroup()	Sets permissions for MessageProducers and MessageConsumers
Lifecycle	N/A	Setting is overridden by JMS
Queue Type	N/A	Setting is overridden by JMS
Encryption level	WMConnectionFactory.setSSLEncrypted() WMConnectionFactory.getSSLEncrypted()	Set encryption level when calling ConnectionFactory. createConnection()
Access control	Set through the Broker Administrator	
Can publish	Required for MessageProducer, TopicPublisher, and QueueSender	Topic or Queue Document Type permissions
Can subscribe	Required for MessageConsumer TopicSubscriber and QueueReceiver	Topic or Queue Document Type permissions
Log publish	Set through the Broker Administrator	
Log acknowledge	Set through the Broker Administrator	

Broker Client

Broker	JMS	Notes
ID	■ Connection:<clientID> ■ Queue:<name>	The three different JMS objects listed use the Broker ClientID value; the labels

Broker	JMS	Notes
	■ DurableSubscriber:<clientID>## ■ <subscription name>	within angle brackets are the JMS terms.
Application Name	WMConnectionFactory.setApplicationName() WMConnectionFactory.getApplicationName()	--
Broker	WmConnectionFactory.getBrokerName() WmConnectionFactory.setBrokerName()	
Client Group	WmConnectionFactory.getClientGroup() WmConnectionFactory.setClientGroup()	
User name	ConnectionFactory.createConnection (String username, String password)	
Shared Document Order	WmDestination.getSharedStateOrdering() WmDestination.setSharedStateOrdering()	Applicable only to QueueReceivers and DurableSubscribers
State Sharing	WmDestination.getSharedState() WmDestination.setSharedState()	Applicable only to QueueReceivers and DurableSubscribers
Infoset	Internal use only.	
Forced Reconnect	N/A	

Document Type

Broker	JMS	Notes
Name	Topic:<name>Queue:JMS::Queues::<name>	A Topic must have a Document Type with the same name as the Topic. The Document Type of a Queue is the same as the name of the Queue and is prefixed with "JMS::Queues::". For example, "JMS::Queues::MyQueues".
Storage Type	N/A	Overridden by JMS
Time to live	N/A	Overridden by JMS

Broker	JMS	Notes
Validation	N/A	Must have a value of either `None` or `Open`
Fields	See "JMS Marshalling" on page 223.	

C JMS Marshalling

Overview

This appendix describes the JMS marshalling feature. You can use this feature for making native Broker message formats compatible with a format that the webMethods Broker used as a JMS provider understands.

Marshalling is an advanced feature; only developers with a thorough understanding of JMS and Broker events should consider using it when other message remapping solutions are not feasible.

Introduction

The JMS marshalling feature allows you to control the message format of Broker events. The feature provides significant flexibility in mapping JMS messages to and from Broker events generated by native (non-JMS) Broker applications.

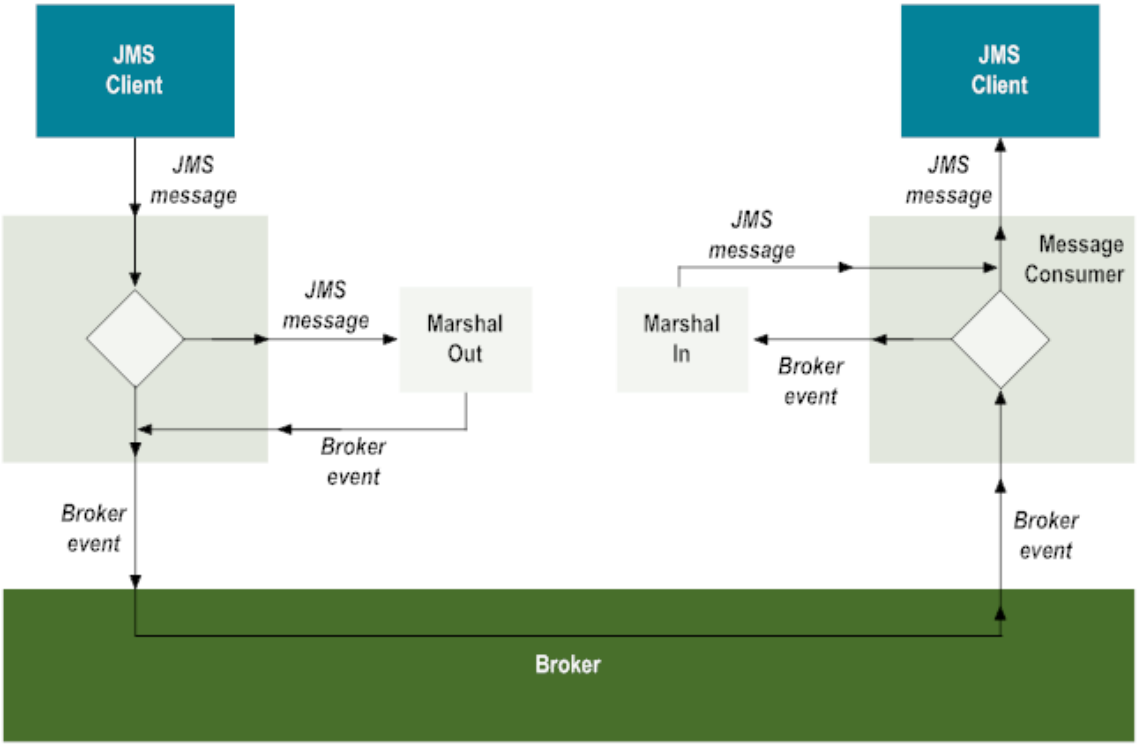
The webMethods Broker used a JMS provider's `WmMessageProducer` and `WmMessageConsumer` classes use the marshalling interfaces to convert JMS messages to and from Broker events. JMS clients using marshalling supply implementations of the interfaces. You can implement marshalling in one direction or in both directions (JMS message to Broker event, or Broker event to JMS message).

The following figure illustrates the message flow in JMS marshalling. For outbound marshalling, the message flow is as follows:

1. A JMS client sends a JMS message to a message producer.
2. The message producer hands off the JMS message to the outbound marshaller.
3. The outbound marshaller maps the JMS message to a Broker event.
4. The Broker event is handed back to the message producer, and is sent to the Broker.

For inbound marshalling, the message flow is as follows:

1. The message consumer receives the Broker event.
2. The message consumer hands off the Broker event to the inbound marshaller.
3. The inbound marshaller maps the Broker event to a JMS message.
4. The inbound marshaller hands back the JMS message to the message consumer.
5. A JMS client receives the JMS message that was marshalled.



Marshalling and Message Producers

Normally, the message producer converts JMS messages to Broker events and sends them to the Broker for transmission to a message consumer.

With outbound marshalling, the intent is to take a fully populated JMS message and produce a Broker event of a form used by a native Broker application.

The normal message flow between the message producer, webMethods Broker, and message consumer is changed by the introduction of `MarshalOut` object. This changes the messaging flow as follows:

1. The message producer sends the JMS message to the `MarshalOut` object.
2. The `MarshalOut` object maps the JMS message to a Broker event.
3. The message producer hands the returned Broker event to the Broker.

Marshalling and Message Consumers

On reception of a Broker event, the message consumer performs the reverse operation from that of the message producer. Normally, the message consumer converts the Broker event into a JMS message of the appropriate type, then passes the message to the JMS client. If a `MarshalIn` object is defined, the message consumer hands the Broker event to the `MarshalIn` object and passes the returned JMS message to the JMS client.

Marshalling and Connection Factories

The set up of marshalling objects on a connection factory is done once per connection. To set up marshalling, you use the webMethods connection factory interface `WmConnectionFactory`. This interface defines the following methods for marshalling:

- Set the outbound marshalling object (`setMarshalOut()`)
- Set the inbound marshalling object (`setMarshalIn()`)
- Set/get the inbound marshaller class name (`setMarshalInClassName()`, `getMarshalInClassName()`)
- Set/get the outbound marshaller class name (`setMarshalOutClassName()`, `getMarshalOutClassName()`)

The first two methods offer the greatest flexibility in using the marshalling feature, allowing you to pass in a user data object containing context information. These methods are useful when writing marshalling code.

The last two methods are employed when setting marshalling properties in a connection factory with the JMSAdmin command-line tool. The methods store class names and bind objects to JNDI. However, if these methods are used, you cannot configure a user data object; a Null value is always returned for a user data object.

Changing the marshalling objects on a `WmConnectionFactory` affects new connections created from that factory. Existing connections continue to use the marshalling objects that were set when they were created.

Following are complete definitions of the marshalling-related methods of the webMethods `WmConnectionFactory` interface:

```
public interface WmConnectionFactory {
    ...
    public void setMarshalOut(WmMarshalOut marshaller, Object userData)
                                throws JMSException;
    public WmMarshalOut getMarshalOut()   throws JMSException;
    public void setMarshalOutClassName(String className)
                                throws JMSException;
    public String getMarshalOutClassName() throws JMSException;
    public void setMarshalIn(WmMarshalIn marshaller, Object userData);
                                throws JMSException
    public WmMarshalIn getMarshalIn()   throws JMSException;
    public void setMarshalInClassName(String className);
                                throws JMSException;
    public String getMarshalInClassName() throws JMSException;
    ...
}
```

Below is some sample code that shows how to implement `WmConnectionFactory`:

```
Counter counter = new Counter();
WmConnectionFactory factory;
Connection conn;
```

```
// Create the connection factory
factory = WmJMSFactory.getConnectionFactory();

// Set up the marshalling methods.
factory.setMarshalOut(new SimpleMarshalOut(), counter);
factory.setMarshalIn(new SimpleMarshalIn(), counter);

// Set up the ConnectionFactory's other properties as needed.
factory.setBrokerHost(brokerHost);
factory.setBrokerName(brokerName);
factory.setClientGroup(jmsClientGroup);

// Set up the JMS connection.
conn = factory.createConnection();
```

Outbound Marshalling

The outbound marshalling object sets the destination and method of delivery for the Broker event, using the `WmBrokerEventWrapper` class.

```
public interface WmMarshalOut
{
    public WmBrokerEventWrapper mapMessageToEvent(Message msg,
                                                  Object userData);
}

public abstract class WmBrokerEventWrapper
{
    public static WmBrokerEventWrapper create(Object brokerEvent)
                throws Exception
    {
        return new WmBrokerEventWrapperImpl(brokerEvent);
    }

    public abstract void setDestinationId(String destId);
    public abstract String getDestinationId();
}
```

If the marshalling object sets the `DestinationId` in the `WmBrokerEventWrapper` that is returned, the webMethods Broker used as a JMS provider delivers the event to that Broker client (the `DestinationId` must be a valid Broker clientID). If the `DestinationId` is not set, the event is published using the event type specified in the `BrokerEvent`. If the event type has not been set for either method of delivery, a `JMSException` will be thrown from the publish or send operation.

The marshalling object can determine the destination of the message from the `JMSDestination` header field in the message. This should be cast to the webMethods-specific `destination` as follows:

```
WmDestination destination =(WmDestination) msg.getJMSDestination();
```

The `getEventType()` method can be used to find out what the webMethods Broker used as a provider would have used for the Broker event type. For queues, the `getName()` method will tell the Broker the client name to which the event would be delivered.

The marshalling object can indicate that it wants to use standard processing by returning a value of `null`. The webMethods Broker used as a JMS provider then publishes or delivers the message as if marshalling was not specified.

Program Control Flow for Outbound Marshalling

The program control flow for an outbound marshalling object is as follows:

1. Consult the `userData` object for any context information that may be needed.
2. Create a Broker event using the constructors for `BrokerEvent`. The `BrokerClient` parameter should be `null`.
3. Iterate over the fields and properties of the JMS message and add them as desired to the newly created `BrokerEvent`. It is possible to use structures and sequences as needed.
4. Create a `WmBrokerEventWrapper` object using the `WmBrokerEventWrapper.create()` method.
5. If the event is to be delivered, use the `setDestinationId()` method of the `WmBrokerEventWrapper` class to specify the Broker client.

Outbound Marshalling Code Example

Code for a simple outbound marshaller is shown in the example below. If the message is a `TextMessage`, the text is changed to uppercase and a counter field is added, using the `Counter` object passed in `UserData`, and the message is published to the same destination (event type) as the message. If the message is not a `TextMessage`, a value of `null` is returned and the webMethods Broker used as a JMS provider will send the message as if no marshaller was present.

```
class SimpleMarshalOut implements WmMarshalOut
{
  public WmBrokerEventWrapper mapMessageToEvent(Message msg, Object
                                    userData) throws Exception
  {
    try {
      if (msg instanceof TextMessage) {
        WmDestination destination=(WmDestination) msg.getJMSDestination();
       BrokerEvent event =
           new BrokerEvent(null,destination.getEventType());

       TextMessage textMsg = (TextMessage) msg;

       event.setUCStringField("myField", textMsg.getText().toUpperCase());
       event.setIntegerField("counter", ((Counter) userData).getValue());
       event.setIntegerField("_env.tag", 10);

       ((Counter) userData).increment();
       return WmBrokerEventWrapper.create(event);
     }

     return null;
   } catch (Exception exc) {
       throw exc;
    }
  }
}
```

Inbound Marshalling

The inbound marshalling interface, `WmMarshalIn`, is called when Broker events arrive at the client. This routine needs to turn the Broker event it gets passed into a JMS message.

The marshal object is passed a `WmMessageFactory` object which implements the create message methods from the JMS `Session` interface.

The marshalling object can indicate that it wants the standard conversion to take place (that is, as if no marshaller was present) by returning a value of `null`.

```
public interface WmMarshalIn
{
  public WmMessage mapMessageFromEvent(Object event, Object userData,
                                       WmMessageFactory messageFactory);
}
```

Program Control Flow for Inbound Marshalling

The program control flow for an inbound marshalling object is as follows:

1. Consult the `userData` object for any context information that may be needed.
2. Use the `WmMessageFactory` object to create a message of the appropriate type.
3. Iterate over the fields of the event and set the appropriate fields or properties in the JMS message.
4. Return the message.

Inbound Marshalling Code Example

A simple inbound marshaller is shown below. The marshaller converts all events into a `TextMessage` with the text field set to a string containing a line for each field with the name, an equals sign, and the value.

```
class SimpleMarshalIn implements WmMarshalIn
{
  public WmMessage mapMessageFromEvent(Object eventIn, Object userData,
                         WmMessageFactory msgFactory)  throws Exception
  {
    BrokerEvent     event = (BrokerEvent) eventIn;
    TextMessage     msg = msgFactory.createTextMessage();
    String[]        fieldNames = event.getFieldNames(null);
    StringBuffer    buf = new StringBuffer();

    for (int i = 0; i < fieldNames.length; i++) {
      BrokerField field = event.getField(fieldNames[i]);

      buf.append(fieldNames[i]);
      buf.append("=");
      buf.append(field.value);
      buf.append("\n");
    }
```

```
    msg.setText(buf.toString());
    return (WmMessage) msg;
  }
}
```

Marshalling and Message Processing

Passing a message through the marshalling routines for processing constitutes the run-time component of your JMS marshalling code. The passing of messages through the routines is done on a per JMS message or Broker event basis.

Following is a simple example of a class that you can use with marshalling, to count the number of messages sent or received (or both).

```
class Counter
{
private int _counter;
public Counter() { _counter = 0; }
public Counter(int start) { _counter = start; }
public synchronized int getValue() { return _counter; }
public synchronized void increment() { _counter++; }
}
```

Marshalling and the Broker Java API

JMS clients that do not use marshalling do not require the Broker Java API; those that do use marshalling will need access to the `BrokerEvent` class. Therefore, JMS clients that use the marshalling interfaces will need the file wm-brokerclient.jar in their CLASSPATH in addition to wm-jmsclient.jar.

Marshalling API Reference

This section contains API reference information for the following:

- `WmMessageFactory` interface
- `WmMarshalIn` interface
- `WmMarshalOut` interface
- `WmBrokerEventWrapper` class

WmMessageFactory

Creates a message of a specified type for use by WmMarshalIn implementors.

Parameters

Name	Description
`object`	(Use only with type ObjectMessage.) The object used to initialize a message containing a serializable object.
`text`	(Use only with type TextMessage). The string used to initialize a text message.

Interface

```
package com.webmethods.jms.marshal;

import java.io.Serializable;

import javax.jms.JMSException;

import com.webmethods.jms.WmBytesMessage;
import com.webmethods.jms.WmMapMessage;
import com.webmethods.jms.WmMessage;
import com.webmethods.jms.WmObjectMessage;
import com.webmethods.jms.WmStreamMessage;
import com.webmethods.jms.WmTextMessage;

public interface WmMessageFactory
{
    /**
     * Returns a new BytesMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmBytesMessage  createBytesMessage() throws JMSException;

    /**
     * Returns a new MapMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmMapMessage    createMapMessage() throws JMSException;

    /**
     * Returns a new Message that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmMessage       createMessage() throws JMSException;

    /**
     * Returns a new ObjectMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmObjectMessage createObjectMessage() throws JMSException;

    /**
     * Returns a new ObjectMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmObjectMessage createObjectMessage(Serializable object) throws
                JMSException;

    /**
```

```
     * Returns a new StreamMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmStreamMessage createStreamMessage() throws JMSException;

    /**
     * Returns a new TextMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmTextMessage   createTextMessage() throws JMSException;

    /**
     * Returns a new TextMessage that can be populated by the
     * marshalling routine from the BrokerEvent.
     */
    public WmTextMessage createTextMessage(String text)
                            throws JMSException;
}
```

Remarks

`WmMessageFactory` implements the same message creation routines as the `javax.jms.session` interface, except that the routines return the webMethods interfaces for each message type. This is to ensure that the marshaller returns webMethods Broker used as a JMS provider messages, and not a message from a different JMS provider session.

WmMarshalIn

Maps a WmBrokerEvent to a JMS message.

Parameters

Name	Description
`event`	The Broker event that was received.
`userData`	An Object set when establishing the marshalling routine.
`msgFactory`	The factory used to create the message.

Interface

```
package com.webmethods.jms.marshal;

import javax.jms.Message;
import javax.jms.Session;

import com.webmethods.jms.WmMessage;

public interface WmMarshalIn
{
   public WmMessage mapMessageFromEvent(Object event,
         Object userData, WmMessageFactory msgFactory) throws Exception;
```

```
}
```

Remarks

The marshaller is established by using the `WmConnectionFactory.setMarshalIn` method. The `Message` that represents the event is returned. If `null` is returned, the message is derived from the event in the standard manner.

WmMarshalOut

Maps a JMS message to a WmBrokerEvent for transport through the Broker.

Parameters

Name	Description
`msg`	The JMS message that is to be sent.
`userData`	An Object set when establishing the marshalling routine. It may contain data in specific contexts, a list of Broker events, or may be a specific event type.
`msgFactory`	The factory used to create the message.

Interface

```
package com.webmethods.jms.marshal;

import javax.jms.Session;
import javax.jms.Message;
import javax.jms.Destination;

import com.webmethods.jms.WmMessage;

public interface WmMarshalOut
{
    public WmBrokerEventWrapper mapMessageToEvent(Message msg,
             Object userData) throws Exception;
}
```

Remarks

This interface does all of the input transformation necessary to convert a fully populated JMS message to a native BrokerEvent. The marshaller is established by using the `WmConnectionFactory.setMarshalOut` method. The WmBrokerEventWrapper class returns a `WmBrokerEventWrapper` object that contains the event to be sent. If `null` is returned, the original message will be sent.

WmBrokerEventWrapper

Creates a wrapper for a BrokerEvent that is used by the outbound marshalling object.

Parameters

Name	Description
`brokerEvent`	The BrokerEvent that is to be wrapped.
`destID`	The destination to which the wrapped event will be delivered.

Class

```
package com.webmethods.jms.marshal;

import com.webmethods.jms.marshal.impl.WmBrokerEventWrapperImpl;

public abstract class WmBrokerEventWrapper
{
    public static WmBrokerEventWrapper create(Object brokerEvent)
                throws Exception
    {
        return new WmBrokerEventWrapperImpl(brokerEvent);
    }

    public abstract void setDestinationId(String destId);
    public abstract String getDestinationId();
}
```

Remarks

`WmBrokerEventWrapper` is an abstract class whose methods create a wrapper for a `brokerEvent`, which is the native Broker object that the marshalling routines have been populating. (The class is defined as abstract so that it is not instantiated directly.) An object containing the wrapped `brokerEvent` is returned. If `null` is returned, the original (non-marshalled) message will be sent.

A JMS message originally set to be published cannot be changed to be delivered without supplying a Broker client name. That is the purpose of using `setDestinationID()`, which allows a Broker client name to be entered as the destination. Do not use `setDestinationID()` if the `brokerEvent` is to be published rather than delivered.

If `getDestinationID()` has been set, it returns the name of the Broker client to which the event will be delivered.

`WmBrokerEventWrapper` throws an exception if the Broker Java API classes cannot be found.

D Message Compression

Introduction

The message compression allows JMS or C# messages to be compressed prior to being sent by a message producer, and automatically decompressed when received by a message consumer. Using message compression reduces the amount of bandwidth required and saves space for messages that the Broker persists.

This feature only compresses the body of a message; the message header, which includes the envelope fields and message properties, are left in their uncompressed state. This allows message selectors (filters) to work for the compressed message.

Classes

webMethods message compression for JMS uses the same classes as the `java.util.zip.Deflator` and `java.util.zip.Inflator` classes provided with the JDK, and used by the tool WinZip.

webMethods message compression for C# uses the following classes:

```
ICSharpCode.SharpZipLib.Zip.Compression.Streams.DeflatorOutputStream
ICSharpCode.SharpZipLib.Zip.Compression.Streams.DeflatorInputStream
```

Enabling Message Compression

You can compress messages on a per-message and per-producer basis.

Compressing a Single Message

To enable message compression on a *per-message basis,* the following methods are provided. Note that in the interface code, the compression ratio for a sent message is calculated as follows:

ratio = ((uncompressed - compressed)/uncompressed) * 100

- For JMS messaging:

```
public interface WmMessage {

   // Enables or disables compression for this message.
   public void setCompression(boolean compress);

   // Get the value for whether compression is enabled or disabled.
   public void getCompression();

   // Get the compression ratio for message that was sent.
   public int getCompressionRatio();
}
```

  The following example shows how to compress a single message for JMS.

```
// Create a topic publisher.
TopicPublisher pub = session.createPublisher(topic);

// Create a text message.
```

```
TextMessage msg = session.createTextMessage(text);

// Set the compression.
msg.setCompression(true);

// Publish the message.
pub.publish(msg);

// Print the compression ratio.
int ratio = ((WmMessage) msg).getCompressionRatio();
System.out.println("compression ratio = " + ratio);
```

- For C# messaging:

```
Boolean Compression ()  }
   {
      set;
      get;
   }
   int CompressionRatio
   {
      get;
   }
}
```

The following example shows how to compress a single message for C#.

```
// Create a topic publisher.
IMessageProducer pub = session.createPublisher(topic);

// Create a text message.
ITextMessage msg = session.createTextMessage(text);

// Set the compression.
msg.Compression = true;

// Publish the message.
pub.publish(msg);

// Print the compression ratio.
int ratio = msg.CompressionRatio;
System.out.println("compression ratio = " + ratio);
```

Compressing All Messages From a Producer

You can enable message compression on a *per-producer basis,* automatically compressing every message sent by the producer whose size exceeds a particular value. Note that a value of -1 (the default) disables message compression. The following methods are provided for JMS and C#:

- For JMS messaging:

```
public interface WmMessageProducer {

   // Set the compression threshold for this message producer.
   public void setCompressionThreshold(int threshold);

   // Get the compression threshold for this message producer.
   public int getCompressionThreshold();
```

```
}
```

- For C# messaging:

```
public int CompressionThreshold();
  {
     set;
     get;
  }
```

The following are some sample codes showing compression on a per-producer basis:

- For JMS messaging:

```
// Create a queue sender.
QueueSender sender = session.createSender(queue);

// Set the sender's compression threshold so that every
//   message greater than 256 bytes will be compressed.
((WmMessageProducer) sender).setCompressionThreshold(256);
```

- For C# messaging:

```
// Create a queue sender.
IMessageProducer sender = session.createProducer(queue);

// Set the sender's compression threshold so that every
//   message greater than 256 bytes will be compressed.
sender.CompressionThreshold = 256;
```

E Message Streaming

Overview

This appendix describes the webMethods message streaming feature, which allows you to stream large amounts of data or a large file from a message producer to a message consumer.

Introduction

The webMethods streaming feature permits the streaming of large amounts of data/large files (greater than 1 Mb). The interface for this mechanism includes stream interfaces; these interfaces do not support transacted sessions.

Supporting Interfaces

Support for message streaming is provided through the following webMethods interfaces:

JMS

- Extensions to `WmMessageProducer`, which has methods to get the output stream object. In the message stream, there is a method to write data to the message producer.
- Extension to `WmMessageConsumer`, which has a method to get the input stream object. In the message stream, there is a method to read data from a message consumer.

Implementation of the streaming API introduces two new classes: `WmJMSInputStream` and `WmJMSOutputStream`. These classes extend the `java.io.InputStream` and `java.io.OutputStream` classes, and are returned by `WmMessageConsumer.getInputStream()` and `WmMessageProducer.getOutputStream()`.

The `WmJMSOutputStream` class streams data written to it by packaging the data into 1 Mb chunks and sending the data to the message producer's destination using a JMS `BytesMessage`. The quality of service (persistent or non-persistent), along with priority and expiration, will be the same as those set for the message producer. The `WmJMSInputStream` class asynchronously receives the `BytesMessages` and makes the data available when the stream is read.

Once the `WmJMSOutputStream` is closed, a final `BytesMessage` is sent with the provider-specific Boolean property `JMS_WM_END_OF_STREAM`set to true. When this is received by the `WmJMSInputStream` class, it will return -1 from `read()` to indicate that the end of the stream has been reached.

C#

- Interfaces to `IMessageProducer`, which has a method returning class `MsgOutputStream`. You can use the class to write data.
- Interfaces to `IMessageConsumer`, which has a method to read data from the class `MsgInputStream`.

Implementation of the streaming API introduces two new classes: `MsgInputStream` and `MsgOutputStream`. These extend the stream class and are returned by `IMessageConsumer.getInputStream()` and `IMessageConsumer.getOutputStream()`.

The `MsgOutputStream` class streams data written to it by packaging the data and sending it to the message producer's destination using a `BytesMessage`. The quality of service (persistent or non-persistent), along with priority and expiration, will be the same as those set for the message producer. The `MsgInputStream` class asynchronously receives the `BytesMessages` and makes the data available when the stream is read.

Once the `MsgOutputStream` is closed, a final `BytesMessage` is sent with the provider-specific Boolean property `JMS_WM_END_OF_STREAM` set to true. When this is received by the `MsgInputStream` class, it will return 0 from `read()` indicating that the end of the stream has been reached.

Support for Multiple Message Producers

The message streaming feature supports the existence of multiple streams. To identify each stream:

- JMS uses a stream identification property to identify the next unique string. You can get or set this string for recovery purposes.

```
public  void setStreamID (String streamID);
public  void getStreamID ()
```

- C# uses the following to identify multiple streams:

```
public  String StreamID ()
{
   set;
   get;
}
```

The message consumer will create a Broker selector set to Broker Server (with the header property `Begin_of_Stream` set to `true`) when initially receiving the message.

Support for Read Timeout Setting

A user can specify the timeout for the `WmJMSInputStream` read function by the following API in the `WmJMSInputStream` class:

```
void setReadTimeout(long millisecond);
```

The `WmReadTimeoutException` will be thrown when a timeout occurs.

Recovery Operations

In the event of a failure on the part of the message producer or consumer, the interface provides several methods and properties to assist in recovery operations.

The following methods are used to support crash recovery. You need to use `setStreamID()` and `getStreamID()` (for JMS), and `StreamID()` (for C#), described earlier, to identify the correct stream in the event of a crash.

JMS

For `WmJMSInputStream`:

```
public  void  markStreamBytePosition (int pos)
public  int   currentReadBytePosition ()
```

For `WmJMSOutputStream`:

```
public  void  markStreamBytePosition (int pos)
public  int   currentWrittenBytePosition ()
```

C#

```
public  Long Position ()
{
   set;
   get;
}
```

On the message consumer side, if the user specified `CLIENT_ACKNOWLEDGE` during session creation, all of the messages will be acknowledged when the input stream is closed.

For additional information about specifying the location in a stream, see "Technical Considerations" on page 243.

Example

Following are sample code fragments for receiving and playing an audio file using `WmJMSInputStream` for JMS, and `MsgInputStream` for C#:

JMS

```
// Create a non-transacted session.
Session sess = connection.createSession(false, Session.AUTO_ACKNOWLEDGE)

// Create a message consumer.
MessageConsumer consumer = sess.createConsumer(topic);

// Wrap the message consumer's input stream in an audio stream.
AudioStream as = new AudioStream(as);

// Start the audio player.
AudioPlayer.player.start(as);
```

C#

```
// Create a non-transacted session.
Session sess = connection.createSession(false, Session.AUTO_ACKNOWLEDGE)

// Create a message consumer.
IMessageConsumer consumer = sess.createConsumer(topic);
```

```
MsgInputStream consumer is = consumer.getInputStream();

// Wrap the message consumer's input stream in an audio stream.
AudioStream as = new AudioStream(as);

// Start the audio player.
AudioPlayer.player.start(as);
```

Size Guidelines

Consider using the webMethods message streaming feature only when message size is 1 MB or greater; otherwise, use regular messaging.

For messages averaging between 1 MB and 1 GB in size, use `CLIENT_ACKNOWLEDGE` when creating a session. For messages averaging over 1 GB in size, use `AUTO_ACKNOWLEDGE` when creating a session.

Technical Considerations

Following is additional information on properties used to specify location in a message stream.

Property	Description
`STREAM_BYTE_POSITION`	Specifies the byte position of the stream that will be sent to the message consumer. The consumer stores this value locally.Whenever the consumer receives the incoming byte position number, and it is less than the local byte position number, it skips the streaming data until the incoming byte position number exceeds that value. Thus, the property provides a means for handling situations where the producer fails during the streaming process and attempts recovery.
`JMS_WM_END_OF_STREAM`	Identifies the end of a stream. When the message consumer receives this value, it will clear the filter on the Broker Server side.

F Using Access Labels with webMethods Messaging

Overview

This appendix describes how to implement access labels with webMethods Messaging. Access labels allow the granting of user rights on a per-message basis. Access labels provide access control for event types that is independent of the event type. For detailed information about access labels and how they work in webMethods Broker, see *Administering webMethods Broker*.

Introduction

webMethods Broker provides several methods for JMS and C# APIs that allow you to set access labels and access label hints for Broker Clients.

Supporting Interfaces

Support for access labels is provided through the following webMethods interfaces:

JMS

Support for access labels is provided through the webMethods message interface `WmMessage`, webMethods connection factory interface `WmConnectionFactory`, and webMethods connection interface `WMConnection`. These interfaces define the following methods for access labels:

- WmMessage

  Set the client access label to the envelope field (`setAccessLabel(short [] label)`)

- WmConnectionFactory
  - Set an access label hint for a Broker client when creating the connection (`createConnection(String userName, String password, String accessLabelHint)`)
  - Set an access label to particular document automatically (`setAutomaticControlLabel(boolean enable)`)
- WmConnection

  Get a client access label (`getAccessLabel ()`)

C#

Support for access labels is provided through the webMethods message interface `IMessage`, webMethods connection factory interface `IConnectionFactory`, and webMethods connection interface `IConnection`. These interfaces define the following properties for access labels:

- IMessage

  Set a client access label (`short [] AccessLabel`)

- IConnectionFactory

- Set an access label hint for a Broker client when creating the connection (`CreateConnection(String serverCertDistiguishedName, String clientCertDistiguishedName, String accessLabelHint)`)
- Set an access label to particular document automatically (`boolean AutomaticControlLabel`)

- IConnection

  Get a client access label (`short [] AccessLabel`)

Setting a Client Access Label

The access label assigned to particular document is called a control label, and it is stored in the envelope field `_env.controlLabel`. A publishing client is responsible for setting this field using an API.

To set a client access label, the following methods are provided for JMS and C#:

- JMS Messaging:

```
public interface WmMessage {
// Set access label to message envelope
  public void setAccessLabel(short [] label) throws JMSException;
  }
```

- C# Messaging:

```
public interface IMessage {
// Set access label to message envelope
  short[] AccessLabel
  {
           set;
  }
```

Setting the Client Access Label Automatically

An option is provided to have the Broker set this `controlLabel` envelope field for the client automatically by using the client's access label. This option is disabled by default. To enable this option, use the following API calls:

- JMS Messaging:

```
public interface WmConnectionFactory {
// Enable or disable Broker to set the control label automatically.
The Broker client’s control label will be automatically inserted into
the controlLabel envelope field of any event the client publishes
or delivers. Set to "true" to enable and "false" to disable.
  public void setAutomaticControlLabel(boolean enable);
}
```

- C# Messaging:

```
public interface IConnectionFactory {
//Enable or disable Broker to set control label automatically.
The Broker client's control label will be automatically inserted
```

```
into the controlLabel envelope field of any event the client
publishes or delivers. Set to "true" to enable and "false" to disable.
  Boolean AutomaticControlLabel
  {
           set;
  }
```

Getting a Client Access Label

The client access label that is returned for the client is inserted in the `publabel` envelope field of every event the client publishes or delivers.

To get an access label for a client, the following methods are provided for JMS and C#:

- JMS Messaging:

```
public interface WmConnection {
// Get the client access label
  public short [] getAccessLabel () throws JMSException;
}
```

- C# Messaging:

```
public interface IConnection {
// Get the client access label
  short [] AccessLabel
  {
            get;
  }
```

Specifying an Access Label Hint for a Broker Client

When using the access label feature, a client can specify a hint string to be used to look up the client's access label. The access label look up occurs only when creating a client. The hint string will be ignored when reconnecting a Broker client.

The connection is created in stopped mode. Messages will not be delivered until the `Connection.start` method is explicitly called.

To assign an access label hint for a Broker client, the following API calls are provided for JMS and C#:

- JMS Messaging:

```
public interface WmConnectionFactory {
// Creates a connection with the specified user identity and
access label hint.
   public Connection createConnection(String userName, String password,
   String accessLabelHint)
}
```

- C# Messaging:

```
public interface IConnectionFactory {
// Creates a connection with the specified user identity and
access label hint.
```

```
IConnection CreateConnection(String clientCertDistiguishedName,
String accessLabelHint);
}
```

G webMethods Messaging Publisher Reconnect

Overview

This appendix describes how to implement the publisher reconnect feature with webMethods Messaging.

Introduction

When a connection is interrupted either by a Broker Server failing to respond or a downed network connection, the transaction between the publishers and non-durable subscribers and the Broker Server is stopped and data may be lost.

The publisher reconnect feature enables publishers and non-durable subscribers to automatically connect to an available Broker Server and continue operations if the connection with its original Broker Server fails.

When the reconnect feature is enabled and a connection is disrupted, the publisher reconnect method restores the connection by connecting the publishers and non-durable subscribers to the next available Broker Server specified in the Broker list. The Broker list is a pool of Broker Servers in your webMethods Broker environment to which the system can connect in the event a connection is disrupted or lost.

When you define the Broker list, the reconnect feature becomes enabled. You can also set the order in which a Broker Server is selected from the Broker list. For information about defining the Broker list and setting the connection order, see "Enabling Publisher Reconnect" on page 252.

Considerations

- This feature supports the publishers and non-durable subscriber with non-temp topic only. The reconnect logic will throw exception if a client attempts to create either a queue receiver (for temp or non-temp queue) or a durable subscriber or non-durable subscriber with a temp topic in a multiple Broker Server deployment.
- If this feature is enabled for non-durable subscribers with a non-temp topic, a reconnection might result in data loss.
- A client can only create a durable subscription in a non-clustered or single Broker Server deployment.

Enabling Publisher Reconnect

Enable the JMS publisher reconnect feature by defining the Broker list. You specify all the Broker Servers that you want to include in the reconnection pool in the Broker list. You can also specify the order in which the Broker Servers are chosen in the reconnection pool; sequential or random.

- For JMS messaging:

  Use the `connectionFactory.setBrokerList` property to define the Broker list.

  ```
  public void setBrokerList(String brokerList) throws JMSException;
  ```

Add each Broker Server to the list using the following syntax:

```
[Broker Name]@<host>[:port]
```

Separate each entry with a comma. If there is only one Broker Server in the list, the client will attempt to reconnect with that server at pre-defined intervals. You can specify the intervals in the Java properties, `com.webmethods.jms.broker.reconnectAttempts` and `com.webmethods.jms.reconnectDelay` .

The following example shows how to create a Broker list containing two Broker Servers for JMS.

```
TopicConnectionFactory tcf1 = new    WmTopicConnectionFactoryImpl();
            ((WmConnectionFactoryImpl)tcf1).setBrokerList("Broker
#1@localhost:6849,Broker #1@Tokyo:2000");
```

- For C# messaging:

  Use the `IconnectionFactory.BrokerList` property to define the Broker list.

```
public interface IConnectionFactory {
   String BrokerList
   {
    get;
    set;
   }
```

  Add each Broker Server to the list using the following syntax:

```
[Broker Name]@<host>[:port]
```

  Separate each entry with a comma. When a connection is lost, the client will attempt to reconnect with the original server at pre-defined intervals. If the connection attempt fails, then it will attempt to connect to the next server in the list. You can specify the connection attempt intervals in the application configuration file (app.config).

  The following example shows how to create a Broker list containing two Broker Servers for C#

```
IConnectionFactory cf = MSGFactory.GetConnnectionFactory(null);
   cf.BrokerList = "Broker #1@localhost:6849,Broker #1@Tokyo:2000");
```

Set the Connection Order

You can specify the failover logic to be either random or sequential. The default ordering is `RANDOM` so the system will randomly choose a Broker Server from the Broker Server list. If you specify `SEQUENTIAL` for the connection factory property, the system will attempt to connect to another Broker Server starting with the first one in the list.

- JMS Messaging

  To set the connection order, use the following method.

```
public void setBrokerListOrder(String brokerListOrder) throws JMSException;
```

The following example shows how to set the Broker list order to `SEQUENTIAL` for JMS.

In this example, the Broker list order is set `SEQUENTIAL`, causing the system to connect to the next Broker Server in the list.

```
((WmConnectionFactoryImpl)tcf1).setBrokerListOrder("SEQUENTIAL");
```

- C# Messaging

To set the connection order, use the following method.

```
public interface IConnectionFactory {
   String BrokerListOrder
   {
   set;
   get;
   }
```

The following example shows how to set the Broker list order to `SEQUENTIAL` for C#.

```
cf.BrokerListOrder = "SEQUENTIAL";
```

H C# Application Configuration File

Overview

This appendix describes how to customize a C# messaging application through its application configuration file.

Using a Custom Application Configuration File

You can change the default property settings for a webMethods messaging client built in C# by reconfiguring its application configuration file. For example, you may want to specify a particular Broker to the messaging client, or change the default timeout value for a connection.

By default, no application configuration file is used with a C# messaging client; the client uses the default property values defined in the webMethods C# Messaging API `MsgConfig` class. However, you can reconfigure the default settings by:

- Generating an application configuration file at compile time, if one does not already exist.
- Modifying some XML elements.
- Adding the property names and their values to the configuration file.

Example

The following code fragment shows an example of an App.config XML file for a C# messaging application. The code concerned with customizing the default property values is explained.

You can use the example to copy and paste the appropriate lines of code to the App.config file of your C# messaging application.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="webMethods.Msg"
             type="System.Configuration.SingleTagSectionHandler" />
    <section name="log4net"
             type="System.Configuration.IgnoreSectionHandler" />
  </configSections>
  <webMethods.Msg Broker="Broker #1@localhost:6849" Timeout="30000"
             ...
/>
```

Inside the `<ConfigSections>` block, add a section element with the `name` attribute equal to `webMethods.Msg` and the `type` attribute equal to `System.Configuration.SingleTagSectionHandler`:

```
<section name="webMethods.Msg"
         type="System.Configuration.SingleTagSectionHandler" />
```

Both of the above lines are required to reconfigure the default values.

After the `<configSections>` block, add an element with the `section name` that you identified previously. Then, add the properties whose defaults you want to reconfigure, and enclose their values in double quotes.

```
<webMethods.Msg Broker="Broker #1@localhost:6849" Timeout="30000".../>
```

Note that you can still override any of these reconfigured default values programmatically.

You can include additional <section> blocks for other applications in the application configuration file. For example, to include log4net, an open source utility designed for .NET logging services, you would use the following lines:

```
 <section name="log4net"
        type="System.Configuration.IgnoreSectionHandler" />
```

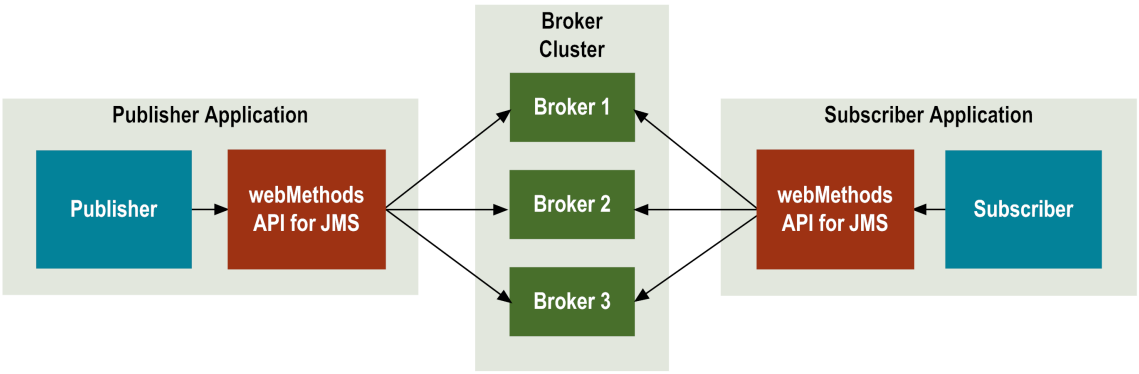
I JMS Policy Based Client-Side Clustering

Overview

The JMS policy based client-side clustering feature in webMethods Broker helps you scale your messaging system. When you implement JMS policy based clustering, you configure a group of two or more Brokers in a cluster to maximize message distribution from the publishing applications to the subscribing applications.

When a Broker comes online or goes offline in a cluster, the webMethods API for JMS throws a connection exception. The JMS client application must catch the connection exception and reconnect. Failover is achieved when the webMethods API for JMS switches the publishing task to another Broker in the cluster based on the policy.

The chapter explains the underlying clustering architecture that supports load balancing for webMethods clients for JMS, lists the most common usage scenarios for these features, and details the steps required for their configuration.



What is Client-Side Clustering for JMS Client Applications?

The webMethods client-side clustering for JMS feature includes a capability that allows you to group multiple Brokers into a single unit of functioning called a Broker *cluster*. All the Brokers in a cluster share the same metadata. The Broker metadata consists of document-type (Broker events) and client-group information. If you change the metadata of one Broker in the cluster, the metadata of all the Brokers in the cluster is updated and synchronized.

You can use a Broker cluster:

- To *scale* your messaging system horizontally, allowing it to handle larger messaging loads (load-balancing).

  Note:
  Delivery of messages in the published order is not guaranteed for all the supported load balancing policy. Use the webMethods client-side clustering for JMS feature when the message ordering is not important.

- To increase the *availability* of your messaging system. In the event of an outage, when the JMS client application reconnects after receiving the connection exception from the webMethods API for JMS, the webMethods API for JMS switches the publishing task to another Broker in the cluster based on the policy.

JMS Policy Based Clustering

webMethods client-side clustering for JMS is made possible through a webMethods extension to the JMS API. The extension allows you to define one or more policy based *cluster connection factories*. A cluster connection factory is similar to a normal JMS connection factory, except that it defines a template for creating connections to a group of Brokers, rather than to a single Broker.

The cluster connection factory is associated with a cluster and a load balancing policy that determines the message distribution logic during the publish operation. Based on the governing policy in the cluster connection factory, the message is published to one or more Brokers in the cluster. For more information about load balancing policies, see "Cluster Load Balancing Policies" on page 264.

webMethods clustering for JMS offers only client-side failover. When a Broker in a cluster is not available for processing the messages, the messages get stuck in a queue and cannot reach a subscriber until that Broker restarts. The remaining Brokers in the cluster continue to share the message distribution load. Conventional HA solution (server-side failover) is still required to back up each server on the list.

Once-and-only once delivery is not guaranteed for the multisend best effort and multisend guaranteed policies if any of the Broker goes down during publishing or subscription. For more information, see "Cluster Load Balancing Policies" on page 264.

C# API clients will continue to use the publisher reconnect feature if they want a rudimentary publisher-side failover support.

JMS priority messaging is supported in client-side clustering for JMS. The individual Broker in a cluster will conform to the message priority, and the queue will contain messages ordered by priority, but, the subscriber could receive messages randomly from any Broker. The Brokers in the cluster will not coordinate with each other to ensure that the subscriber receives the message from a queue that has the highest priority message. Message priority will be strictly enforced on a single Broker, but would only be loosely followed across Brokers in a cluster.

Important:
You can use the webMethods client-side clustering for JMS features only with JMS client applications.

Indications for Using Load Balancing

You should consider load balancing if the message flow rate requirements between publishers and subscribers are greater than the throughput that a single Broker, or Brokers belonging to a single Broker Server, is able to manage.

A typical situation for implementing a load-balancing solution is for *high-volume guaranteed messages*. This category refers to messaging environments with high-volume flow of guaranteed messages between publishers and subscribers. Since a Broker Server's throughput is constrained by its local disk performance, a single Broker Server may be unable to handle the message flow throughput.

Supported Messaging Features

The load balancing feature supports the following:

- JMS-based application clients.
- Both durable and non-durable subscribers.

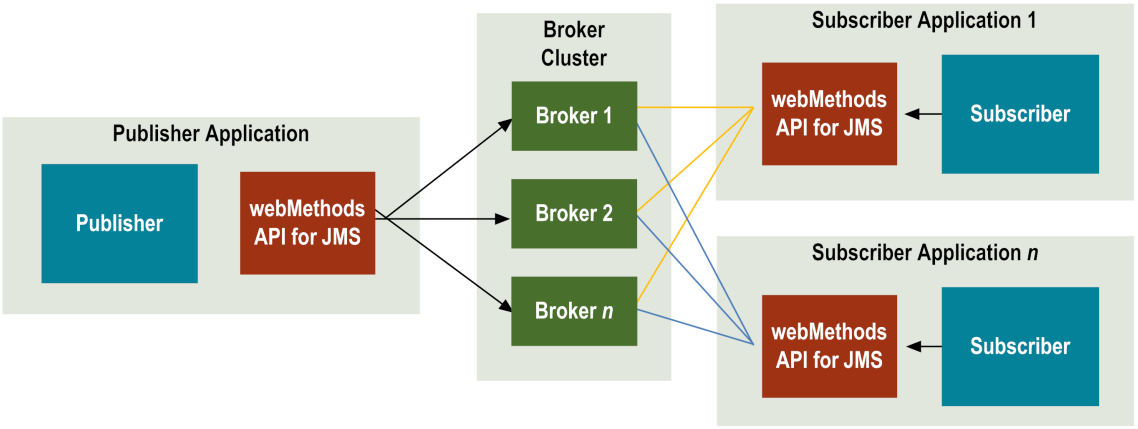
The Broker load-balancing feature does *not* support by-publisher-order delivery (that is, there is no guarantee that messages will arrive at a subscriber in the same order in which they were sent by the publisher). Once-and-only once delivery is not guaranteed for the multisend best effort and multisend guaranteed policies if any of the Broker goes down during publishing or subscription. For more information about multisend best effort and multisend guaranteed policies, see "Cluster Load Balancing Policies" on page 264.

You need to add your own custom application code to guarantee once-and-only-once message delivery.

Distributing Message Volume (Load Balancing)

By distributing the message flow across several Brokers, the messaging system becomes less vulnerable to bottlenecks caused by factors such as a high rate of messages from a publisher, or by a single Broker processing the messages at too slow a rate, which would cause excessive message queuing.

The following figure shows, in a generic sense, how JMS messages flow from a publishing application to subscribing applications through a Broker cluster. The figure uses a simplified messaging scenario consisting of one publishing application, one destination, and two subscribing applications.



As shown in the figure:

- The webMethods API for JMS on the publisher side use the asynchronous publishing interface for initiating multiple publishing operations concurrently to all the available Brokers in the cluster.

- During a publish operation, the webMethods API for JMS determines the Broker to which a published message is routed. The load balancing policy configured in the cluster connection factory defines the routing of the published messages to the Brokers in the cluster.
- A subscriber is connected to all the Brokers in the cluster as shown in the figure and will receive a message from any Broker in the cluster.

  Subscriber applications connect to *all* the Brokers in a cluster, receiving messages from the queues. Messages at the subscriber end are not distributed according to a load balancing policy, as they are at the publisher end. Rather, the messages are simply aggregated from the individual message threads on each clustered Broker and retrieved on a first-come-first-serve basis. The messages are not received by the subscribers in any guaranteed order.

You will need to add custom code to your applications to maintain order of message delivery.

Failover Mechanism in Clusters

In a cluster, if any Broker goes offline or comes online, the webMethods API for JMS throws a connection exception. Configure your JMS client application to catch the connection exception and reconnect to the cluster.

When the webMethods API for JMS receives the request to reconnect from the JMS client application, the webMethods API for JMS switches the publishing task to an available Broker in the cluster, based on the policy.

For example, consider a Broker cluster consisting of Broker A, Broker B, and Broker C. At any point of time, if Broker A goes offline or comes online, failover is achieved as follows:

1. Broker A in the cluster goes offline.
2. webMethods API for JMS throws a connection exception.
3. The JMS client application catches the connection exception and reconnects.
4. webMethods API for JMS switches the publishing task to Broker B and Broker C.
5. The JMS client application continues publishing using Broker B and Broker C.
6. If Broker A comes online, the webMethods API for JMS again throws a connection exception.
7. The JMS client application catches the connection exception and reconnects.
8. webMethods API for JMS includes Broker A for publishing messages.
9. The JMS client application continues publishing using Broker A, Broker B, and Broker C.

Client-Side Failover vs. High-Availability (HA) Clustering

Broker High-Availability (HA) clustering is another failover solution available through Software AG. Compared with Broker client-side failover, Broker High-Availability (HA) clustering is a "true" *server-side* clustering solution requiring the deployment of shared-storage hardware and clustering software. By contrast, Broker client-side failover is a *software* solution: no specialized hardware is required for its deployment.

If a clustered Broker Server (or machine hosting a clustered Broker Server) experiences an outage, the failover software directs the "backup" clustered machine, also connected to the same storage as the primary machine, to take control. Because the clustered failover machine is typically a backup, HA clustering is an *active-passive* clustering solution. On the other hand, JMS client-side failover is an *active-active* failover.

Cluster Load Balancing Policies

JMS messages are published to the specific Brokers in a cluster based on the specified load balancing policy. Select one of the following load balancing policies for the cluster when you define the cluster connection factory.

Note:
The failover mechanism for each of these policies is as explained in "Failover Mechanism in Clusters" on page 263.

- **Round Robin**. A policy where the first publish operation is routed to the first Broker in the cluster, the second publish operation is routed to the second Broker and so on until all Brokers are used. Then the sequence repeats itself. You use this policy if you want to equally distribute the publish operations across all the Brokers in the cluster.
- **Sticky**. A policy where clients route all publish operations to the first Broker in the cluster. Only when the first Broker fails, the webMethods API for JMS begins to route all publish operations to the next available Broker in the cluster.
- **Random**. A policy where publish operations are routed randomly to any Broker in the cluster. It is possible that one of the Brokers might receive too many messages to process, while the other Brokers are idle.
- **Weighted Round Robin**. A policy where each Broker in the cluster handles a different message processing load. You can specify a "weight" to each Broker, which signifies how the Broker must share the message load relative to the other Brokers in the cluster. The weight determines how many more (or fewer) messages a Broker must receive, compared to the other Brokers in the cluster. You must carefully determine the relative weights to assign to each Broker.

  The following examples will help you better understand the weighted round robin policy:

  - If all Brokers in the cluster have the same weight, they will each share an equal proportion of the load.
  - If one Broker has weight 1 and all other Broker have weight 2, the Broker with weight 1 will bear half of the load compared to the other Brokers.
  - If you have three Brokers in a cluster that must be governed by a weighted round robin policy, assign weight 4 for Broker 1, weight 12 for Broker 2, and weight 1 for Broker 3. In this example, for every 17 messages, the webMethods API for JMS routes 4 messages to Broker 1, 12 messages to Broker 2, and 1 message to Broker 3.
- **Multisend Best Effort**. A multisend policy where a client publishes a message to all, or as many Brokers as possible in the cluster. The publish operation is considered successful if even one of the Brokers receives the message. The message that the client publishes to the Brokers

is non-transactional in nature. Multisend best effort policy does not work with XA connection factory.

For more information about preventing duplicate message delivery by the webMethods API for JMS to the subscriber, see "Message Pre-Acknowledgement" on page 268.

- **Multisend Guaranteed**. A multisend policy where a client publishes a message to *n* out of *m* Brokers in a cluster by using XA transactions. Each Broker in the cluster acts as an individual XA resource and the publish operation is considered to be successful if the message is received by *n* out of *m* Brokers in the cluster. For example, if there are 10 Broker in cluster, you want to guaranty publishing to 2 Brokers, then you use 2 out of 10 Brokers. Multisend guaranteed policy works only with XA connection type.

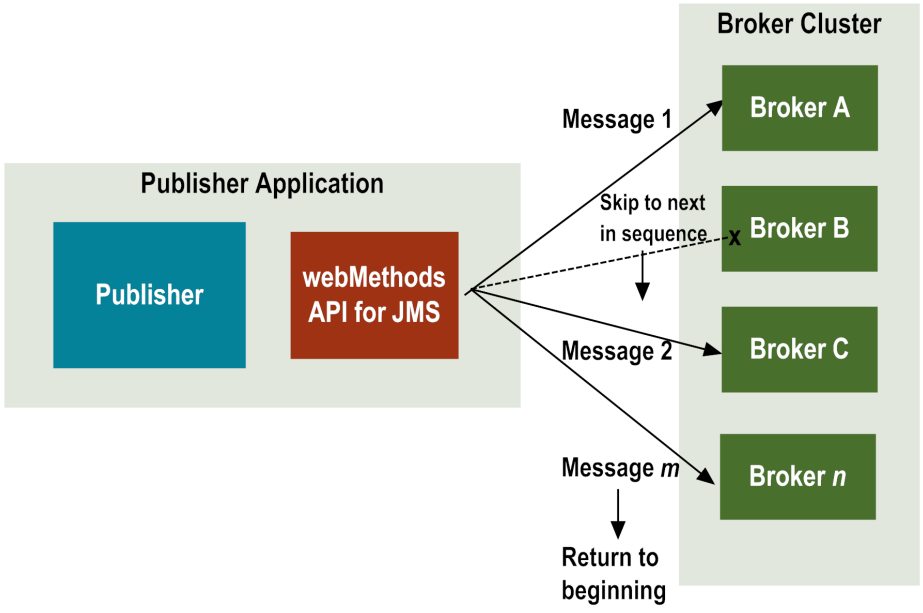
  The client must use the correct webMethods APIs for JMS to retrieve the XA resources, enlist the resources with the transaction manager, and perform the XA operations. The webMethods APIs for JMS do not automatically perform these transaction management operations.

  For more information about preventing duplicate message delivery by the webMethods API for JMS to the subscriber, see "Message Pre-Acknowledgement" on page 268.

How Does the Round Robin Cluster Policy Work?

The following figure gives an example of how round robin load balancing policy will work with a cluster of four Brokers.

Round Robin Selection Logic Used in Load Balancing



For load balancing, published messages are directed sequentially, that is the first message is sent to the first Broker and the next message to the second Broker. After the last Broker in cluster receives the messages, next message is sent to the first Broker, and the sequence repeats. One message at a time is sent to each Broker in the load-balancing cluster.

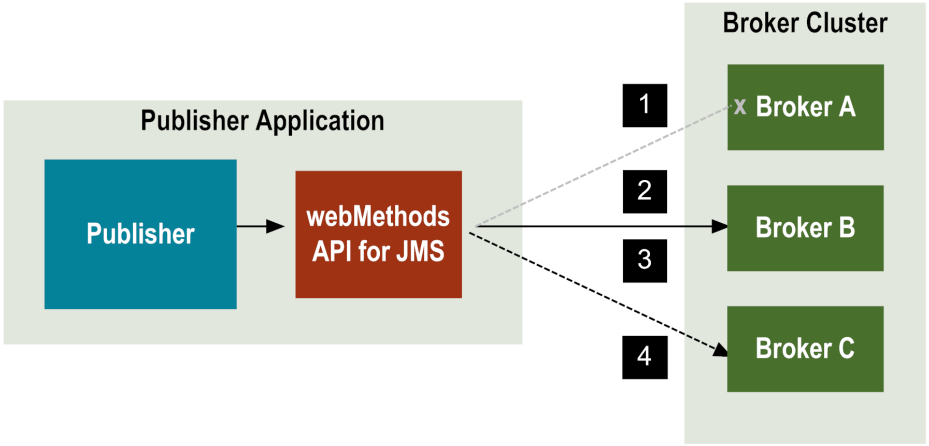
If a Broker in the cluster goes offline (indicated by the cross for Broker B), when it is the turn of Broker B to receive a message, the message is re-directed to the next Broker in the cluster, that is Broker C. Broker B will be skipped for load distribution until it is available again.

Note:
In an XA transaction, all the published messages are routed to a single Broker determined by the cluster policy in the cluster. The cluster policy applies to a transaction instead of a message.

How Does the Sticky Policy Work?

If the cluster connection factory is configured using a sticky policy, when the Broker configured for distributing the messages goes offline, the next available Broker listed in the sequence of clustered Brokers will receive all subsequent messages. The following figure illustrates this process, using an example with a failover cluster consisting of three Brokers.

Selection Logic Used in Client-Side Failover



The numbered labels in the figure list the sequence of events occurring when a Broker becomes non-operational.

Label	Description
1	Broker A goes offline. webMethods API for JMS throws a connection exception and JMS client application catches the connection exception and reconnects.
2	The failover mechanism searches for the next available Broker in the cluster to which messages can be routed; this is Broker B.
3	Subsequent messages are routed to Broker B. Even if the Broker A comes back online, the messages will continue to be sent to Broker B. When Broker A comes back online, Broker A will be included in the list of available Brokers for message distribution. If Broker B experiences an outage or if the publisher client re-establishes the connection to the cluster, the messages are routed back to Broker A.

Label	Description
4	No messages are routed to Broker C until Broker B experiences an outage and Broker A is also not available.

How Does the Multisend Best Effort Policy Work?

In a cluster, if you have defined a multisend best effort policy, the client publishes a message to all, or to the maximum number of Brokers specified in the cluster. For more information about defining the maximum number of receiving Brokers in the cluster, see "Defining the Maximum Number of Receiving Brokers in a Cluster" on page 267.

The publish operation is considered successful if even one of the Brokers receives the message. The message that the client publishes to the Brokers is non-transactional in nature.

Note:
Multisend best effort policy does not work with XA connection factory.

For more information about preventing duplicate message delivery by the webMethods API for JMS to the subscriber, see "Message Pre-Acknowledgement" on page 268.

Defining the Maximum Number of Receiving Brokers in a Cluster

You can define the maximum number of recipient Brokers while defining the multisend best effort cluster connection factory. If you set the maximum limit to *n*, the client sends the message to exactly *n* Brokers. If the number of available Brokers is less than *n*, the message will be sent to all the available Brokers. Message sending will fail only when the message cannot be sent to any Broker in the cluster. For more information about defining the maximum number of receiving Brokers, see "Managing Cluster Connection Factories" in the guide *Administering webMethods Broker*.

Note:
The option to select the maximum limit is optional. If you do not configure the maximum limit, the multisend best-effort policy will continue to send messages to as many Brokers in the cluster connection factory as possible.

How Does the Multisend Guaranteed Policy Work?

When you define a multisend guaranteed policy in a cluster, the webMethods API for JMS publishes a message to *m* out of *n* Brokers in a cluster by using XA transactions. Each Broker in the cluster acts as an individual XA resource and the publish operation is considered to be successful if the message is received by *m* out of *n* Brokers in the cluster.

Note:
Multisend guaranteed policy works only with XA connection type.

For more information about preventing duplicate message delivery by the webMethods API for JMS to the subscriber, see "Message Pre-Acknowledgement" on page 268.

Message Pre-Acknowledgement

The published message is sent to multiple Brokers in the cluster if the cluster is governed by either multisend guaranteed or multisend best effort policy. You require a mechanism to prevent duplicate message delivery to the subscriber. The message pre-acknowledgement functionality prevents duplicate message delivery.

In a cluster configured with either multisend guaranteed or multisend best effort policy, the publisher's webMethods API for JMS marks one of the Brokers in the cluster as the primary Broker. The webMethods API for JMS performs the following steps to prevent duplicate delivery of messages and ensure that the subscriber has received at least one copy of the message.

1. The webMethods API for JMS sends the published message to multiple Brokers in the cluster based on the policy setting. In case a cluster is configured with multisend guaranteed policy, the webMethods API for JMS sends the message to *m* out of *n* Brokers in the cluster. One of the Brokers in the cluster marked as the primary Broker receives the published message. The other Brokers participating in the message distribution in the cluster receive a tentatively pre-acknowledged message.
2. The primary Broker sends the message to the subscriber. The remaining Brokers in the cluster will hold the published message until they receive the confirmed pre-acknowledgement for the message from the subscriber's webMethods API for JMS.
3. The subscriber receives the message from the primary Broker and acknowledges receiving the message.
4. For every message acknowledgement from the subscriber client, the webMethods API for JMS sends:
    a. an acknowledgement to the primary Broker in the cluster. The primary Broker deletes the message from the client queue.
    b. a confirmed pre-acknowledgement to the other Brokers in the cluster. The secondary Brokers delete the tentatively pre-acknowledged messages from the client queues.
5. If any of the Brokers in the cluster does not receive the message pre-acknowledgement before the message pre-acknowledgement timeout interval, that Broker will forward the published message to the subscriber.

Message Pre-Acknowledgement Time Interval Setting

For each Broker in the cluster, edit the Broker Server configuration file, awbroker.cfg file, to set the values of message pre-acknowledgement parameters.

It is recommended to set different values of `tentative-pre-acknowledgement-timeout` parameter for different Brokers in the cluster to prevent simultaneous duplicate message delivery from the secondary Brokers. Appropriate values for pre-acknowledgement parameters optimizes the use of cluster and ensures that the subscriber receives the message only once. For more information, see the details in the following table:

Parameter	Specifies...	Recommended Setting
tentative-pre-acknowledgment-timeout	The minimum time the Brokers in the cluster will hold the published message before sending it to the subscriber. ■ If a Broker does not receive confirmed message pre-acknowledgement before the specified tentative-pre-acknowledgement-timeout time interval, the Broker forwards the published message to the subscriber. ■ If a Broker receives the confirmed pre-acknowledgement before the specified tentative-pre-acknowledgement-timeout time interval, the Broker deletes the published message from the client queue. Default value is 20 seconds.	For each Broker in the cluster, set different values of tentative-pre-acknowledgement-timeout parameter. ■ Specify a small value to enable quick message publishing using secondary Brokers in case the primary Broker goes offline before sending the message to the subscriber. ■ Specify a large value to minimize duplicate message delivery. If the primary Broker does not send the message to the subscriber within the tentative-pre-acknowledgement-timeout time, a duplicate message is sent to the cluster. The message delivery by the primary Broker may be delayed because of the following reasons: ■ Primary Broker has a temporary failure. ■ Slow response either due to high latency networks or the number of documents in the primary Broker is large (that is, the queue builds up).
confirmed-pre-acknowledgment-timeout	The maximum time interval a cluster Broker holds a confirmed pre-acknowledgement before discarding it. This parameter setting is required to enable cleanup of accumulated confirmed pre-acknowledgements in Brokers that never receive the published messages. Default value is 10 minutes.	

Parameter	Specifies...	Recommended Setting
pre-acknowledgement-timer-interval	The time interval in seconds for checking the pre-acknowledgement timeout status in a cluster Broker. The checks are done in the time interval specified by pre-acknowledgement-timer-interval parameter until the value specified by tentative-pre-acknowledgement-timeout parameter is reached. Default value is 10 seconds. **Note:** The timer works only for active sessions. So, if a client is disconnected from the durable queue, the timer does not work. The timeout of tentative pre-acknowledgement works in two phases to account for slow subscribers.	Specify a value less than half the value of tentative-pre-acknowledgement-timeout parameter. ■ If you specify a very small value, Broker checks the timeout status frequently. ■ If you specify a very large value, Broker may not detect the timeout effectively.

Example

For example, consider a cluster is governed by multisend guaranteed policy. The cluster has four Brokers: Broker 1, Broker 2, Broker 3, and Broker 4. Broker 1 is the primary Broker. Only Broker 1, Broker 2, and Broker 3 are used to publish the messages. The pre-acknowledgement parameters have the following settings:

For Broker 1, `pre-acknowledgement-timer-interval=10`, `tentative-pre-acknowledgement-timeout=20`, and `confirmed-pre-acknowledgement-timeout=7200`.

For Broker 2, `pre-acknowledgement-timer-interval=5`, `tentative-pre-acknowledgement-timeout=30`, and `confirmed-pre-acknowledgement-timeout=7200`.

For Broker 3, `pre-acknowledgement-timer-interval=5, tentative-pre-acknowledgement-timeout=40`, and `confirmed-pre-acknowledgement-timeout=7200`.

The webMethods API for JMS performs the following steps:

1. The webMethods API for JMS sends the published message to all the Brokers in the cluster.
2. The subscriber receives the message from the primary Broker, Broker 1, and acknowledges receiving the message.
3. The Broker 2, and Broker 3 will hold the published message until it receives the pre-acknowledgement message from the subscriber.
   a. Broker 2 will hold the message for the time period specified by the `tentative-pre-acknowledgement-timeout` parameter, that is `30` seconds. At every time

interval specified by the `pre-acknowledgement-timer-interval` parameter, that is 5 seconds, the Broker 2 will update the timer status for the message. Broker will perform this step every 5 seconds until the message is not removed from the queue.

b. Broker 3 will hold the message for the time period specified by the `tentative-pre-acknowledgement-timeout` parameter, that is `40` seconds. At every time interval specified by the pre-acknowledgement-timer-interval parameter, that is 5 seconds, the Broker 3 will update the timer status for the message. Broker will perform this step every 5 seconds until the message is not removed from the queue.

4. The webMethods API for JMS sends an acknowledgement to all the Brokers in the cluster.

5. If the other Brokers in the cluster receive the pre-acknowledgement message from the webMethods API for JMS before the specified message pre-acknowledgement timeout interval, these Brokers remove the published document from its queue.

6. If the Brokers in the cluster do not receive the message pre-acknowledgement before the message pre-acknowledgement timeout interval, the other Brokers in the cluster will forward the published message to the subscriber.

   a. If Broker 2 does not receive the pre-acknowledge message before 30 seconds, it will send the message to the subscriber.

   b. If Broker 3 does not receive the pre-acknowledge message before 40 seconds, it will send the message to the subscriber. For more information about recommended pre-acknowledgement setting, see "Message Pre-Acknowledgement Time Interval Setting" on page 268.

7. The webMethods API for JMS will never publish the message to Broker 4. But, Broker 4 also receives a confirmed pre-acknowledgement from the webMethods API for JMS. Broker 4 deletes the confirmed pre-acknowledgement after the `confirmed-pre-acknowledgement-timeout` interval, that is after 7200 seconds in this example.

Overriding Cluster Policy

While sending messages to a cluster, you can override a cluster policy and specify the Brokers to which the messages must be sent in a cluster. This functionality is supported for all cluster polices, including the MultisendGuaranteed policy. This feature supports cluster and composite cluster connection factories.

There are two steps to override the cluster policy and specify the Brokers to which the messages must be sent in a cluster.

1. **At design time**, specify **Cluster Policy Override** while defining the cluster connection factory or the composite cluster connection factory.

2. **At runtime**, override the cluster policy for the subsequent messages by setting the WM_JMS_CLUSTER_NODES message property to the value obtained from the first message. TheWM_JMS_CLUSTER_NODES message property can contain reference to a single Broker or a list of Brokers.

Specifying Cluster Policy Override At Design Time

During design time, use My webMethods or the JMSAdmin command line utilities to specify "Cluster Policy Override" in a cluster connection factory or composite cluster connection factory.

- In My webMethods, select the **Cluster Policy Override** option in the Add or Edit Cluster Connection Factory pages.
- Using the bind or modify JMSAdmin command line utilities, set the `clusterPolicyOverride` parameter to `true` when you define the composite or cluster connection factories.

Specifying Cluster Policy Override At Runtime

If you specify "Cluster Policy Override" in a cluster connection factory, the webMethods API for JMS sets the value of the WM_JMS_CLUSTER_NODES property of the first message to the Broker (or Brokers) to which the message was sent.

When a message is sent to a cluster governed by a cluster policy, the WM_JMS_CLUSTER_NODES property is not set for the first message. The webMethods API for JMS sends this message to the Broker (or Brokers) determined by the cluster policy. The webMethods API for JMS assigns this list of Broker(s) to the value of the WM_JMS_CLUSTER_NODES property of the message.

For the subsequent messages, you override the cluster policy by setting the WM_JMS_CLUSTER_NODES message property to the value obtained from the first message. Thereafter, the JMS client application sends the messages only to the Broker (or Brokers) to the Brokers specified in the WM_JMS_CLUSTER_NODES property.

For example, in case of Random policy, the webMethods API for JMS assigns the value of the WM_JMS_CLUSTER_NODES property of the first message to the Broker to which the message was sent. In case of Multisend Guaranteed policy or Multisend Best Effort policy, the first message is sent to multiple Brokers in the cluster; the webMethods API for JMS assigns the list of Brokers (to which the first message was sent) to the WM_JMS_CLUSTER_NODES property.

webMethods API for JMS throws an exception in the following cases:

- If only one Broker is specified in the WM_JMS_CLUSTER_NODES property, and the specified Broker is not available.
- If the cluster policy is Multisend Best Effort, and all the Brokers specified in the WM_JMS_CLUSTER_NODES property are not available.
- If the cluster policy is Multisend Guaranteed, and any one of the Brokers specified in the WM_JMS_CLUSTER_NODES property is not available.

Examples

- Set the WM_JMS_CLUSTER_NODES property to "Broker #2@localhost:6849" as follows:

```
pMessage.setStringProperty(WmMessage.WM_JMS_CLUSTER_NODES, "Broker
#2@localhost:6849");
```

- Set the cluster override flag in a cluster connection factory as follows:

```
WmQueueConnectionFactoryImpl queueCF = null;
queueCF = (WmQueueConnectionFactoryImpl) WmJMSFactory.
getQueueConnectionFactory();  queueCF.setClientGroup(clientGroup);
queueCF.setBrokerCluster(new String[]
{"Broker #1@localhost:6849","Broker2@localhost:6949",
"Broker3@localhost:7010"});
queueCF.setClusterPolicy(WmClusterPolicyManager.ROUND_ROBIN);
queueCF.setClusterPolicyOverride(true);
```

JMS Local Transactions in Clusters

In a JMS client application, you can use local transactions to group the messages sent and received in the cluster. A transaction commit means that all the produced messages are sent to a Broker in the cluster, and all the consumed messages are acknowledged. A transaction rollback means that all the produced messages are destroyed, and all the consumed messages are recovered from the cluster. For JMS local transactions, each Broker in the cluster on the consumer side is considered as a single XA Resource.

Internally, the webMethods API for JMS adopts the two-phase commit protocol for these local transactions. The two-phase commit ensures that each Broker in the transaction agrees on whether the transaction should be committed or not. Only when all the transacting Brokers agree for a transaction commit, is the commit() executed on the Brokers.

Transaction Logging

The webMethods API for JMS records the transaction information in the transaction log for each transacting Broker in the cluster. To verify the state of local transactions of each Broker in the cluster, view the transaction logging information. The transaction logs help you manually recover transactions that might have failed during the prepare or the commit phase.

Enabling Transaction Logging

By default, the logging for one-phase commit (1PC) local transactions is disabled. To enable the transaction logging, in the wmjms.properties file, set the provider-specific property `com.webmethods.jms.cluster.log1PC` to specify the log file.

If the `com.webmethods.jms.cluster.log1PC`property is not set, the transaction details are not logged.

For more information on setting the properties, see “Properties Specific to webMethods Broker Used as a JMS Provider” on page 20 in “Introduction to webMethods Broker Messaging” on page 13.

The format of the transaction logs is as follows:

```
2009-09-16 14:59:14
PREPARING 202  Broker2@localhost:6949
PREPARED 202 Broker2@localhost:6949
PREPARING 13007 Broker #1@localhost:6849
PREPARED 13007 Broker #1@localhost:6849
COMMITING 202 Broker2@localhost:6949
```

```
COMMITED 202 Broker2@localhost:6949
COMMITING 13007 Broker #1@localhost:6849
COMMITED 13007 Broker #1@localhost:6849
2009-09-16 14:59:31
PREPARING 13011 Broker #1@localhost:6849
PREPARED 13011 Broker #1@localhost:6849
PREPARING 206 Broker2@localhost:6949
PREPARED 206 Broker2@localhost:6949
COMMITING 13011 Broker #1@localhost:6849
COMMITED 13011 Broker #1@localhost:6849
COMMITING 206 Broker2@localhost:6949
COMMITED 206 Broker2@localhost:6949
```

Composite Cluster Connection

A *composite cluster connection* is a collection of cluster connections. Composite cluster connections enable you to use two levels of load balancing cluster policies. Each child cluster connection of the composite cluster connection can have a different load balancing policy, and the composite cluster connection has a separate load balancing policy.

When the publishers send the messages to the composite cluster connection, based on the policy governing the composite cluster connection, the published message load is routed to the child cluster connections. Each child cluster connection processes the messages based on the policy governing that child cluster connection. In a composite cluster connection, the child clusters connections work like a single Broker connection.

Composite Cluster Connection Load Balancing Policies

The JMS messages are published to the composite cluster connections according to the load balancing policy specified in the composite cluster connection factory.

When you define a composite cluster connection factory, select one of the following load balancing policies for the composite cluster connection:

- Round Robin
- Sticky
- Random
- Multisend Best Effort
- Multisend Guaranteed

Note:
A composite cluster factory does not support weighted round robin policies. The multisend best effort policy is not supported in XA mode.

The child cluster connection factories that are associated with a composite cluster factory supports the following load balancing policies.

- Round Robin

- Sticky
- Random
- Multisend Best Effort
- Multisend Guaranteed
- Weighted Round Robin

For more information about cluster load balancing policies, see "Cluster Load Balancing Policies" on page 264.

Usage of Composite Cluster Connections

Composite cluster connections increase the reliability of your messaging system. Using multisend best effort policy in a composite cluster connection factory, you can set parallel distribution of messages through the child cluster Brokers in different data centers.

- **Increase the availability of your messaging system**. In an event of failure of all the Brokers in a child cluster connection, the webMethods API for JMS re-directs the message load to the next available child cluster connection.
- **Increase the reliability of your messaging system**. Using multisend best effort policy in a composite cluster connection factory, you can set parallel distribution of messages through the child cluster Brokers in different data centers.

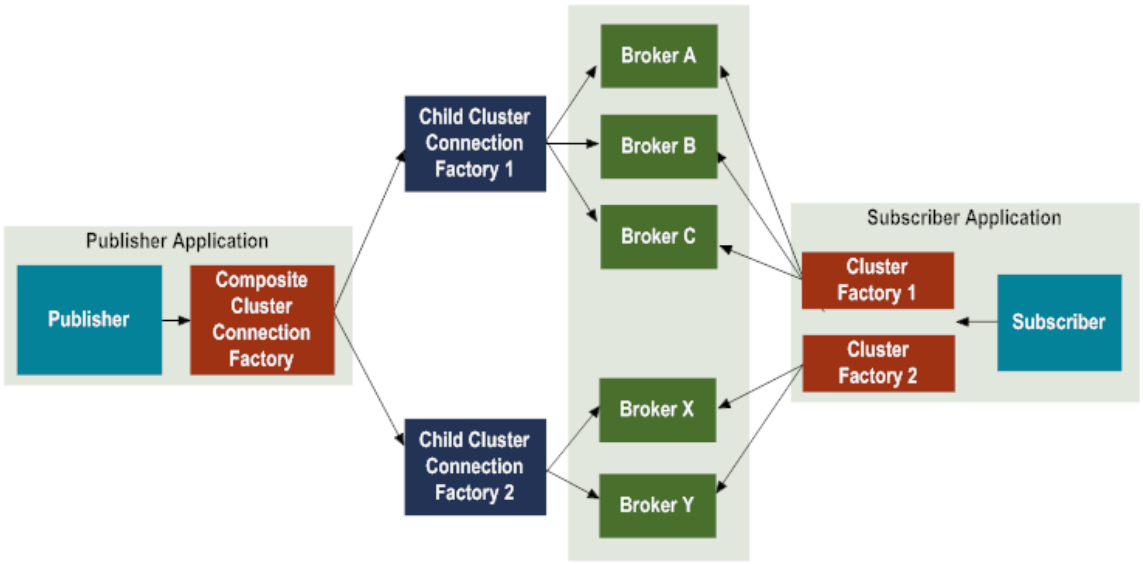
Message Flow in a Composite Cluster Connection

According to the governing policy, a composite cluster connection distributes the messages to the child cluster connections. The child cluster connections then distribute these messages using its load balancing policy to the respective Brokers.

For example, consider the following composite cluster connection factory configuration:

- A composite cluster connection factory contains two child cluster connection factories: Child cluster connection factory 1 and Child cluster connection factory 2.
- The composite cluster connection factory is configured for failover using the round robin load balancing policy.
- Child cluster connection factory 1 is configured to use random load balancing policy.
- Child cluster connection factory 2 is configured to use sticky load balancing policy

The following figure illustrates this process.



1. The first message is published to child cluster connection factory 1 based on the round robin load balancing policy.
2. The child cluster connection factory 1 routes the message to Broker A, or Broker B, or Broker C based on the random load balancing policy.
3. The second message from the publisher will then be published to child cluster connection factory 2.
4. The child cluster connection factory 2 sends the message to Broker X based on the sticky load balancing policy.
5. Third message is published to child cluster connection factory 1 and so on.
6. If all the Brokers A, B, and C of child cluster connection factory 1 goes offline, all subsequent messages are routed to the child cluster connection factory 2.

JMS Client-Side Clustering Requirements

Before you begin, make sure the following requirements are met:

- The domain name service (DNS) used by the host machines that participate in the cluster must be capable of *bi-directional name resolution*. This means that your DNS must be able to resolve the name of the host machine to the correct IP address and also be able to resolve the machine's IP address back to the correct name.

  When a host machine uses the Dynamic Host Configuration Protocol (DHCP) to dynamically obtain its IP address, many DNS servers will not return the correct name for a given IP address. In this case, you must contact your IT department or your network administrator and ask that a static IP address be assigned to the host machine.

- Each Broker must have a name that is unique within the cluster.
- Each cluster must have a unique name.

- A Broker must not belong to another cluster. To become part of the new cluster, it must first leave its current cluster. (For similar reasons, you cannot merge clusters. To create a single cluster where two existed before, the Brokers in one cluster must leave it and then join the second cluster.)

All Broker Servers that participate in the cluster must have the same security settings. That is, they must all be basic authentication-enabled, or SSL-enabled, or without any security settings.

Configuring webMethods Client-Side Clustering for JMS

To use the webMethods client-side clustering for JMS feature

1. Identify Broker Servers that will be a part of the cluster.

2. Create webMethods Broker cluster. For more information about using the My webMethods user interface, see *Administering webMethods Broker*.

3. Configure the cluster connection factory as follows:

   a. Associate the Broker cluster with a JMS connection factory. For more information about using the My webMethods user interface, see *Administering webMethods Broker*. For more information about configuring a cluster connection factory using the JMSAdmin command-line tool, see the bind connection factory and modify connection factory commands in the "JMSAdmin Command Reference" on page 57.

   b. Associate the JMS connection factory with a load balancing policy. For more information about using the My webMethods user interface, see *Administering webMethods Broker*.

   Important:
   When you create a cluster connection factory using the webMethods APIs for JMS clients, the webMethods API for JMS does not validate if the specified Brokers belong to the same cluster. Make sure you specify the correct Brokers while creating a cluster connection factory.

4. You can build a Broker cluster connection factory for use with load balancing using any of the following:

   - The My webMethods user interface

     For information, see *Administering webMethods Broker*.

   - Application code using the webMethods API for JMS

     For information about the webMethods API for JMS, see the *webMethods Broker API for JMS Reference* available on the Software AG Documentation website at http://documentation.softwareag.com.

   - The JMSAdmin command-line tool

For information, see "The JMSAdmin Command-Line Tool" on page 45.

Note:
You can use the JMSAdmin command-line tool to only create the connection factories. Use My webMethods user interface or webMethods API for JMS to create clusters.

Application Server and webMethods Client-Side Clustering for JMS

The webMethods clustering API for JMS provides support for Message Driven Beans (MDB). The webMethods API for JMS enables asynchronous communication between loosely coupled components.

Setting an MDB as a listener on the Broker cluster will result in an MDB listening on the specified destination of all the Brokers in the cluster. When a message arrives on any one of the destinations in the cluster, the MDB listener will be invoked and the messages will be passed to the consumer on a first-come-first-serve basis.

The webMethods Broker client-side clustering for JMS client applications relies on the JCA for communicating with the application server. As a JMS provider, we do not provide a JCA compliant resource adapter for communicating with the application servers. You must use a JCA Resource Adapter that implements the JCA 1.5 API standard for inbound communication and outbound communication.

As per the current technology trends, it is recommended that you use JCA (resource adapters) to communicate between an application server and the webMethods Broker used as a JMS provider.

The webMethods Broker used as a JMS provider installation includes the Generic Resource Adapter for JMS that uses the JMS client library and integrates the webMethods Broker used as a JMS provider (Broker) with an application server. The Generic Resource Adapter for JMS uses ASF functionality for inbound connections.

Message Logging

For message logging, set the following properties in the wmjms.properties file.

```
com.webmethods.jms.log.level=DEBUG
com.webmethods.jms.log.filename=filename
```

Make sure you set the classpath of the client applications to the directory containing the `wmjms.properties` file.

The messages are logged when:

- A message is published or republished
- A message is received
- An error occurs
- Connectivity changes

Enabling Load Balancing and Failover in Your JMS Applications

You need not change the code in your JMS applications to enable load balancing feature if JNDI provider is used to store the connection factory data. The webMethods API for JMS will enable or disable load balancing based on the existence of a Broker cluster and load balancing policy definition when a JMS connection is made.

- If the connection factory is defined with only one Broker or without a load balancing policy, all message publishing and receiving will be ordinary JMS messaging.
- If the connection factory is associated with a Broker cluster (multiple Brokers) and a load balancing policy is specified, all the Brokers in cluster will work as a single Broker and distribute the messages according to the load balancing policy.
- If your existing JMS application is using Broker List feature, it is recommended that you update your code to use the load balancing clustering feature. For example, if your current code is similar to:

```
WmConnectionFactoryImpl cf = new WmConnectionFactoryImpl();
cf.setBrokerName("Broker #1");
cf.setBrokerHost("localhost:6849");
cf.setBrokerList("Broker #1@localhost:6849, Broker 2@localhost:6949");
cf.createConnection();
```

  Update your code to use the JMS Policy based clustering feature by replacing `setBrokerList()` with `setClusterPolicy()` & `setBrokerCluster()`:

```
WmConnectionFactoryImpl cf = new WmConnectionFactoryImpl();
cf.setClusterPolicy(WmClusterPolicyManager.STICKY);
cf.setBrokerCluster(new String[] {"Broker #1@localhost:6849",
"B2@localhost:6949", "B3@localhost:7010"});
cf.createConnection();
```

If you create a cluster using the user interface in My webMethods, the cluster validation process ensures that the specified Brokers are in the same cluster and synchronizes the document types and client groups details in all the Brokers in the cluster.

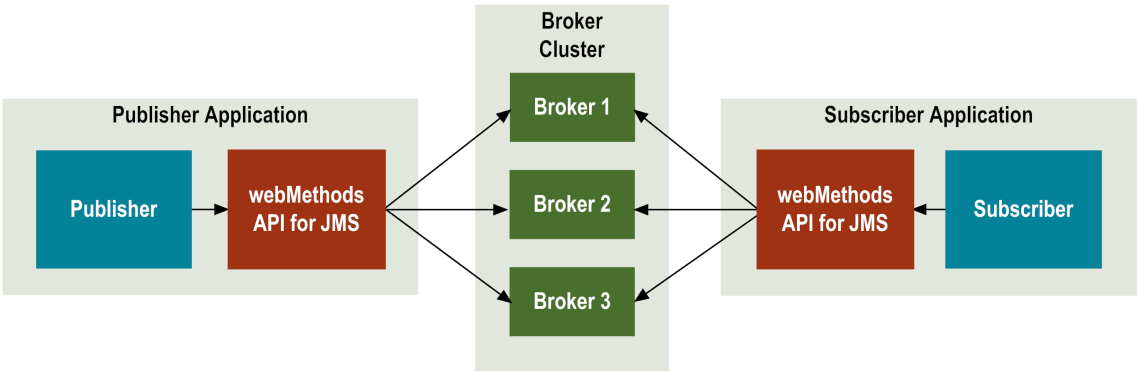
Note:webMethods Broker client-side clustering for JMS client applications does not validate at the webMethods API for JMS level to check if the specified Brokers are in the same cluster or not. Make sure that all the Brokers used for publishing and subscription are defined in the same cluster. All the Brokers in the cluster must have identical document type and client group information.

JTA /XA Support in a JMS Clustered Environment

webMethods Broker used as a JMS provider is a Java Transaction API (JTA) capable provider. This section describes the JTA support to participate in distributed transactions in a webMethods client-side clustering environment for JMS client applications.

Managing Transactions in Clusters

During transactional processing in a webMethods client-side clustering environment for JMS, all the messages published in a transaction are routed to a single Broker in the cluster determined by the cluster policy. In a transactional publish operation, the cluster policy applies to a transaction instead of a message. Whereas in a non-transactional publish operation, each message is distributed to a different Broker based on the cluster policy.



For example, consider that you have configured an XA cluster connection factory with a round robin load balancing policy. The following steps lists the sequence of transaction processing operations in a webMethods clustering environment for JMS client applications.

1. webMethods API for JMS starts the transactions on each of the Brokers in the cluster using the same transaction xid in all the Brokers.

2. Publisher application publishes the messages to the cluster. According to the round robin policy, all the messages of the first transaction are routed to Broker 1. Broker 1 to which all the messages of the transactions are published becomes the participating Broker of this transaction. The messages of the next transaction are routed to the next available Broker in the cluster.

   Subscriber application connects to all the Brokers in the cluster to receive the transaction messages. The transaction messages are gathered from the individual message threads on each cluster Broker and retrieved on a first-come-first-serve basis. Note that if the subscriber receives the messages from Broker 1 and Broker 2, Broker 1 and Broker 2 will become the participating Brokers at the subscriber end. If any of the participating Brokers at the subscriber end goes offline, the commit operation for that transaction will be affected. If any of the non-participating Brokers goes offline, the transaction is not affected.

   Once all the transaction messages are received, the Broker client ends the transaction.

3. webMethods API for JMS prepares the actively participating Brokers in the cluster.

   The prepare operation returns a value indicating the result of the transaction or returns an appropriate XAException for rolling back the transaction.

4. webMethods API for JMS commits the transaction on the participating Brokers. The transactions started in the remaining non-participating Brokers will be rolled back automatically.

The commit operation does not report the errors occurred during a rollback of non-participating Brokers as an XAException.

5. In case of a rollback, webMethods API for JMS rolls back the transactions on each of the participating Brokers.

   The transactions started in the remaining non-participating Brokers will also be rolled back. The rollback operation does not report the errors occurred during rollback of non-participating Brokers as an XAException.

6. The recover operation obtains the list of transaction branches that are currently in the prepared or heuristically completed state on each of the Brokers in the cluster. The transaction manager executes the recover operation during recovery to obtain the list of transaction branches that are currently in prepared or heuristically completed state.

   The recover operation throws an appropriate XAException if an error occurs on any of the participating Brokers in the cluster during recovery.

7. The forget operation forgets all the heuristically completed transaction branches on each of the Brokers in the cluster.

Samples

Sample 1: Defining Weighted Round Robin Policy in a Cluster

This sample demonstrates how you can configure your cluster to use the weighted round robin policy using the webMethods API for JMS.

To configure a cluster to use the weighted round robin policy

1. Create the connection factory using the webMethods API for JMS.

2. Set the load balancing policy as weighted round robin for the connection factory.

3. Define the Brokers in the cluster.

4. Set the weights for each Broker in the cluster and create the weights list in the connection factory.

5. Create the JMS connection, JMS session, and the message producer.

6. Publish messages using the message producer.

```
// create the connection factory
WmConnectionFactory factory = new WmConnectionFactoryImpl();
factory.setClientGroup(clientGroup);
factory.setBrokerCluster( new String[] {"Broker #1@localhost:6849",
                                        "Broker2@localhost:6949",
                                        "Broker3@localhost:7010"} );
```

```
// set the load balancing policy as Weighted Round Robin
factory.setClusterPolicy(WmClusterPolicyManager.WEIGHTED_ROUND_ROBIN);

// Set the weight for each Broker in the cluster
CopyOnWriteArrayList<WeightedRoundRobin> weightList = new
CopyOnWriteArrayList<WeightedRoundRobin>();

WeightedRoundRobin wrr1 = new WeightedRoundRobin();
wrr1.setBrokerName("Broker #1@localhost:6849");
wrr1.setWeight(Integer.parseInt("1"));

WeightedRoundRobin wrr2 = new WeightedRoundRobin();
wrr2.setBrokerName("Broker2@localhost:6949");
wrr2.setWeight(Integer.parseInt("2"));

WeightedRoundRobin wrr3 = new WeightedRoundRobin();
wrr3.setBrokerName("Broker3@localhost:7010");
wrr3.setWeight(Integer.parseInt("2"));

weightList.add(wrr1);
weightList.add(wrr2);
weightList.add(wrr3);

// Set the weights for the factory
factory.setWeightedRoundRobinList(weightList);

// Create the JMS Connection
connection = (Connection) factory.createConnection();
connection.start();

// Create the JMS session
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(queue);

// publish JMS Messages
for ( int i =0; i< 10000; i++) {
     producer.send(session.createTextMessage());
}

// For the above weights set on the Brokers,
// BrokerA should get 2000 documents
// BrokerB should get 4000 documents
// BrokerC should get 4000 documents
```

Sample 2: Defining Multisend Best Effort Policy in a Cluster

This sample demonstrates how you can configure your cluster to use the multisend best effort policy using the webMethods API for JMS.

To configure the cluster to use the multisend best effort policy

1. Create the connection factory using the webMethods API for JMS.

2. Set the load balancing policy as multisend best effort for the connection factory.

3. Set the Brokers in the cluster.

4. Set the maximum number of receiving Brokers in the connection factory.

5. Create the JMS connection, JMS session and the message producer.

6. Publish messages using the message producer.

```
WmQueueConnectionFactoryImpl queueCF = null;

queueCF = (WmQueueConnectionFactoryImpl)
WmJMSFactory.getQueueConnectionFactory();

// set the JMS Client group
queueCF.setClientGroup(clientGroup);

// set the brokers in the cluster
factory.setBrokerCluster( new String[] {"Broker1@localhost:6849",
                                        "Broker2@localhost:6949",
                                        "Broker3@localhost:7010"} );

// set the load balancing policy as Multi Send Best effort
queueCF.setClusterPolicy(WmClusterPolicyManager.MULTISEND_BEST_EFFORT);

// set the maximum number of brokers count
queueCF.setMultiSendCount(2);

QueueConnection queueConnection = (QueueConnection)
queueCF.createQueueConnection();
queueConnection.start();

QueueSession queueSession = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);

MessageProducer producer = queueSession.createProducer(queue);

TextMessage message = null;

   for (int i = 0; i < 10; ++i)
{
message = queueSession.createTextMessage();
message.setText("QueueMessage " + i);
producer.send(message);
}

producer.close();
queueSession.close();
queueConnection.close();
```

Sample 3: Creating Composite Connection Factory

This sample demonstrates how you can programmatically configure a composite connection factory using the webMethods API for JMS for client applications.

A composite cluster connection factory contains two child cluster connection factories: child cluster connection factory 1 and child cluster connection factory 2.

To configure a composite factory

1. Create the a first child cluster connection factory using the webMethods API for JMS.

2. Set the Brokers in the first child cluster connection factory, and configure the first child cluster connection factory to use round robin load balancing policy.

3. Create the second child cluster connection factory.

4. Set the Brokers in the second child cluster connection factory, and configure the second child cluster connection factory to use sticky load balancing policy.

5. Assign the child cluster connection factories created in step 1 and 2 to the composite factory.

6. Configure the composite cluster connection factory to use the multisend guaranteed load balancing policy.

```
//Create the composite child Cluster Connection Factories using the API
   WmConnectionFactory childFactory1 = WmJMSFactory.getXAConnectionFactory();
   String[] brokerList = new String[2];
   brokerList[0] = "Broker1@localhost:9100";
   brokerList[1] = "Broker12@localhost:9200";
   childFactory1.setClientGroup("JMSClient");
   childFactory1.setBroker1Cluster(brokerList); //Set the Broker1 list
   childFactory1.setClusterPolicy(WmClusterPolicyManager.ROUND_ROBIN);
//Cluster Policy
   childFactory1.setClusterName("C1"); //Name of the cluster
   childFactory1.setIncludeAllBroker1s(true); //Include all Broker1s in
   the cluster with the connection factory.

//Create the composite child Cluster Connection Factories using the API
   WmConnectionFactory childFactory2 = WmJMSFactory.getXAConnectionFactory();
   String[] brokerList2 = new String[2];
   brokerList[0] = "Broker1A@localhost:9300";
   brokerList[1] = "Broker1B@localhost:9400";
   childFactory2.setClientGroup("JMSClient");
   childFactory2.setBroker1Cluster(brokerList2); //Set the Broker1 list
   childFactory2.setClusterPolicy(WmClusterPolicyManager.STICKY);
//Cluster Policy
   childFactory2.setClusterName("C2"); //Name of the cluster
   childFactory2.setIncludeAllBroker1s(true); //Include all Broker1s
   in the cluster with the connection factory.

   WmConnectionFactory compositeFactory =
WmJMSFactory.getXAConnectionFactory();
   CopyOnWriteArrayList<WmConnectionFactory> childConnFactories =
   new CopyOnWriteArrayList<WmConnectionFactory>();
   childConnFactories.add(childFactory1);
   childConnFactories.add(childFactory2);

//Set the composite child factory list
```

```
   compositeFactory.setCompositeChildConnFactories(childConnFactories);

//Cluster Policy at composite factory level
   compositeFactory.setClusterPolicy(WmClusterPolicyManager.
   MULTISEND_GUARANTEED);
   compositeFactory.setMultiSendCount(2);
```

J Glossary

administered objects
Administered objects are preconfigured JMS (or C#) objects that a Broker administrator creates for use with client programs. Administered objects are stored in a standardized namespace called the Java Naming and Directory Interface (JNDI), and can be managed with administrative tools or programmatically. There are two types of administered objects: connection factories and destinations.

authentication
The process by which the system validates a user's logon information. The credentials (user name and password) are compared to an authorization list. If a match occurs, authorization is granted to the extent available for the user in the permissions list.

Broker
A part of the Broker Server process, providing services such as receiving, queueing, and delivering events. One or more Brokers can exist on a Broker Server. Each Broker can have any number of document types, client groups, and clients associated with it; they also share process and storage space with other Brokers. Brokers can be added to or leave territories. See also territory.

Broker Server
The core product of the webMethods Broker communication system that runs Brokers. Broker Servers are the delivery and administration hubs of document-based computing. Broker Servers can have multiple Brokers that share the same process and storage space.

certificate
The public key of a private/public key pair that has been signed by a Certification Authority. See also Certification Authority.

Certification Authority
An entity that issues certificates, usually with an authentication server. The webMethods Broker uses trusted root certificates so that a live connection to a Certification Authority to verify certificates is unnecessary.

CLASSPATH
The environment variable that tells the Java compiler where to look for the classes it needs. `-classpath` is an option to the Java interpreter and the Java compiler that tells them (while compiling or running) where to look for a class. `-classpath` overrides `CLASSPATH`.

client group

A Broker client group defines certain properties for all clients belonging to that group, including a client's publish and subscribe permissions to specific document types.

cluster
A cluster is a group of Brokers functioning as a single logical Broker. All Brokers in a cluster maintain the same set of document types and client groups.

connection
An active connection from a JMS or C# client to webMethods Broker.

connection factory
An administered object that a JMS or C# client uses to create a connection with webMethods Broker. A connection factory encapsulates the set of configuration parameters defined for a connection.

The type of connection factory determines whether a connection is made to a topic (in a publish-subscribe application) or a queue (in a point-to-point application), and whether messages are available for management by a distributed transaction coordinator on an application server (XATopicConnectionFactory and XAQueueConnectionFactory).

C# clients can only use a generic connection factory (ConnectionFactory) to establish a connection.

container
An entity that provides services such as security, life cycle management, and run-time services to components. Types of containers include servlets, applets, and application clients.

destination
A destination is an administered object that a client uses to specify the target of messages it produces and the source of messages it consumes. Destinations specify the identity of a destination to a JMS or C# API method. There are four types of destinations: queue, topic, temporaryQueue, and temporaryTopic.

distinguished name
A certificate needs a distinguished name to identify the issuer of the certificate. A distinguished name consists of one or more of the following components:

CN, OU, O, EM, L, ST, C

where CN is the common name, OU the organizational unit, O the organization, EM the e-mail address, L the locality, ST the state or province, and C the country from where the certificate was issued.

distributed transaction
In the messaging domain, a distributed transaction is a transaction where resources from other applications are modified as a result of one or more messages being consumed, as part of a single commit operation. For JMS applications managed by an application server, you use XA-based connection factories in the context of a distributed transaction.

DSA
Digital Signature Algorithm (DSA) is part of the Digital Signature Standard (DSS), selected to be the digital authentication standard of the U.S. Government. DSA is intended for authentication only.

durable subscription

A durable subscription allows subscribers to receive all the messages published on a topic, including those published while the subscriber is inactive. Making a subscription durable provides the reliability of queues to the publish/subscribe style of messaging.

event
A generic message exchanged by resources. The generic nature of events makes possible universal tools that operate on all kinds of events.

initial context
An object on the client that provides access to the JNDI provider and connects to the namespace in which the administered objects are stored.

keystore
A storage location for an SSL certificate. A keystore does not store trusted roots; these are kept in a trust store.

Lightweight Directory Access Protocol (LDAP)
A standard protocol for accessing information in a directory. LDAP defines processes by which clients can connect to an X.500-compliant or LDAP-compliant directory service. Provided the clients have sufficient access rights, they can add, delete, modify, or search for information.

local transaction
A transaction that is handled within the context of a session, and includes any sends and receives within the session. This type of transaction cannot include updates to external resources such as databases. A local transaction does not use the XA-based connection factories

message consumer
A message consumer receives messages from a topic or queue to which it has subscribed. Programmatically, a message consumer is an object (created by a session) that is used for receiving messages sent to a destination.

message listener
An asynchronous event handler for messages.

message producer
A JMS or C# client that produces messages. Programmatically, a message producer is an object (created by a session) that is used to send messages to a destination (a topic or a queue).

message selector
Allows a JMS or C# client to filter the messages it wants to receive by use of a SQL expression in the message header. The expression is applied to a property in the message header containing the value to be filtered Only messages whose property values match the selector (that is, evaluate to a Boolean value of true) are received by the message consumer.

nonpersistent message
A message that has no guarantee of being saved if a failure occurs and the session ends.

persistent message
A message that will be saved if the connection is lost.

Point-to-Point (PTP) messaging

One-to-one delivery of messages. Messages are exchanged between clients through named message queues. A message producer sends a message to a named queue. A message receiver, which subscribes to the queue, receives the message.

publish/subscribe messaging
One-to-many delivery of messages. Messages are exchanged between clients through publishing and subscribing to named topics. A message publisher sends messages to a named topic, and every subscriber to the topic can retrieve the messages.

rollback
A way of ending a transaction whereby all the messages composing the transaction are discarded and updates to any resources included in the transaction are reversed. Typically, any messages included in the transaction are sent again after a rollback.

session
A session object is a single-threaded context for producing and consuming messages. A session can create and service multiple message producers and consumers.

shared state
The process of distributing the processing of topics to multiple clients, possibly executing on different hosts. This process provides a basic form of load balancing. Shared state is enabled when multiple JMS client connections to the webMethods Broker used as a JMS provider share the same connection client ID.

subscriber
In the publish-subscribe messaging model, a subscriber is a client that registers interest in receiving a message. There are two types of subscribers:

- Durable. These subscribers receive all the messages published on a topic, including those published while the subscriber is inactive
- Non-durable. These subscribers receive messages on their chosen topic only if the messages are published while the subscriber is active.

territory
A set of Brokers that share information about their document type definitions and client groups. Brokers within the same territory have knowledge of one another's document type definitions and client groups. Events can travel from clients on one Broker to clients on another Broker in the same territory.

time to live (TTL)
Length of time in milliseconds that the message system retains a produced message.

topic
A specific type of message that is published by a message producer and subscribed to by an interested message subscriber.

transaction
An atomic unit of work that modifies data. Transactions can be local or distributed. Distributed transactions require use of the XA-based interfaces.

trust store

A storage location for the SSL certificates, or trusted roots, of one or more Certification Authorities (CAs). The webMethods Broker uses trusted roots to verify certificates

trusted root

A special certificate issued by a well known and trusted Certification Authority. Trusted roots are used to validate the authenticity of certificates received by a client or server application, and are saved in a special file called a trust store.