BEA WebLogic Server®

Programming WebLogic JMS .NET Client Applications

Version: 10.3 Tech Preview
Document Revised: October 2007

Contents

Programming Considerations

JMS .NET Client Sample Application

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—Programming WebLogic JMS .NET Client Applications.

Document Scope and Audience

This document is a resource for software developers who want to develop .NET applications, written in C#, that access WebLogic Server Java Message Service (JMS) resources.

It assumes that the reader is familiar with the Microsoft® .NET Framework and with programming .NET applications in C#.

This document does not provide detailed information about WebLogic Server JMS. For links to WebLogic Server documentation and resources for these topics, see "Related Documentation" on page 1-2.

Guide to this Document

- This chapter, Chapter 1, "Introduction and Roadmap," introduces the organization of this guide.

- Chapter 2, "Overview of the WebLogic JMS .NET Client," provides an overview of the JMS .NET client, describes how it works, and provides a high-level description of the API.
- Chapter 3, "Installing and Copying the WebLogic JMS .NET Client Libraries," provides details about installing the client, the location of the installed components, and information for copying the library to the .NET machine.
- Chapter 4, "Developing a Basic JMS Application Using the WebLogic JMS .NET API," steps you through the basic steps you use to create a JMS .NET client application and provides a basic example.
- Chapter 5, "Programming Considerations," provides important information and best practices to use when creating a JMS .NET client application.
- Appendix A, "JMS .NET Client Sample Application," provides a detailed example that illustrates how to use some of the advanced JMS concepts in an application.

Related Documentation

This document contains JMS .NET client-specific application development information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server JMS applications, see the Messaging for BEA WebLogic Server® web page on the edocs Web site.

For details about the WebLogic JMS .NET API, see the WebLogic Messaging API Reference documentation.

CHAPTER 2

Overview of the WebLogic JMS .NET Client

These sections provide an overview of the WebLogic JMS .NET client, illustrate how a JMS .NET client application accesses WebLogic JMS resources, and provide a brief summary of the WebLogic JMS .NET API.

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

What is the WebLogic JMS .NET Client?

The WebLogic JMS .NET client is a fully-managed .NET runtime library and application programming interface (API) that enables programmers to create .NET client applications, written in C#, that can access WebLogic Java Message Service (JMS) applications and resources.

WebLogic JMS is an enterprise-level messaging system that fully supports the JMS 1.1 Specification and also provides numerous WebLogic JMS Extensions to the standard JMS APIs. To familiarize yourself with the features of WebLogic JMS, see the Messaging for BEA WebLogic Server® web page on the edocs Web site. For a summary of the WebLogic Server value-added JMS features, see WebLogic Server Value-Added JMS Features.

For complete details about all the classes and interfaces in the JMS .NET API, see the WebLogic Messaging API Reference for .NET Clients documentation.

The WebLogic JMS .NET client, which is bundled with WebLogic Server 10.3 and higher, is supported on Microsoft .NET Framework versions 2.0 and 3.0. Installation details are provided in Chapter 3, "Installing and Copying the WebLogic JMS .NET Client Libraries."

Supported JMS Features

For this release, the WebLogic JMS .NET client supports the major standard features of the JMS Version 1.1 Specification. For a list of the JMS 1.1 standard features that are not supported, see "Limitations of Using the WebLogic JMS .NET Client" on page 5-6.

In addition to the standard JMS 1.1 Specification support, the WebLogic JMS .NET client also supports several of the WebLogic JMS extensions. For more information about the features supported and how they can be used with the JMS .NET client, see "Using the WebLogic JMS Extensions" on page 5-2.

Messaging Models

The WebLogic JMS .NET client supports the following messaging models:

- The point-to-point (PTP) messaging model, which enables one application to send a message to exactly one recipient.
- The publish/subscribe (pub/sub) messaging model, which enables an application to send a message to multiple recipients.

Messages can be specified as persistent or non-persistent:

- Persistent messages are guaranteed to be delivered *once-and-only-once*. The message will not be lost due to JMS server failure and it will not be redelivered once it is acknowledged by an application. It is not considered sent until it has been safely written to a file or database.
- Non-persistent messages are not stored. They are guaranteed to be delivered *at-most-once,* unless there is a JMS provider failure, in which case messages may be lost, and will not be redelivered.

For more information, see "Understanding the Messaging Models" in *Programming WebLogic JMS*.

Message Types

The WebLogic JMS .NET client supports the following message types, as defined in the JMS 1.1 Specification:

- Message
- BytesMessage
- MapMessage
- ObjectMessage (between producers and consumers written in the same language only)
- StreamMessage
- TextMessage

The XMLMessage type extension provided by WebLogic JMS is not supported in this release. Such messages are automatically converted to a TextMessage type when received by a .NET client.

For more information about using the supported message types, see "Exchanging Messages Between Different Language Environments" on page 5-8.

How the WebLogic JMS .NET Client Works

The following figure illustrates how a JMS .NET client application running in a .NET Framework CLR can access JMS resources deployed on WebLogic Server.

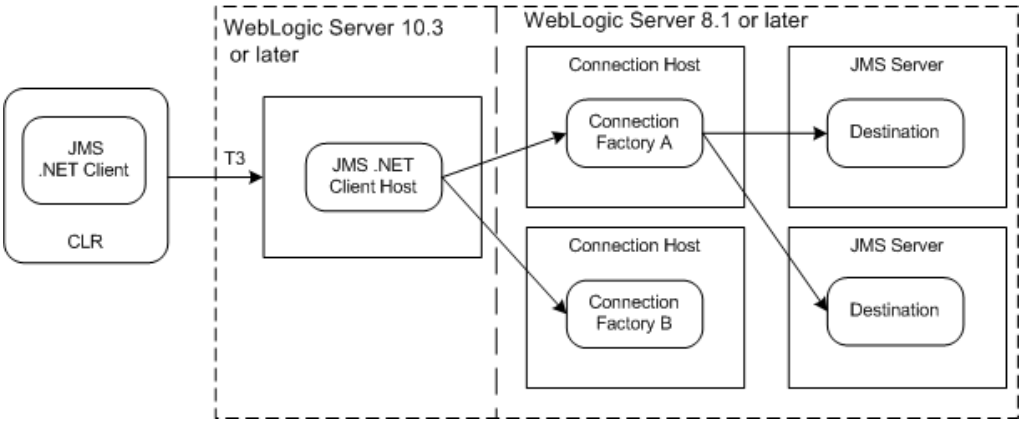


Figure 2-1 JMS .NET Client Architecture

The major components depicted in the illustration consist of the following:

- JMS .NET client written in C#, running in a .NET Framework CLR, that either produces messages to destinations or consumes messages from destinations.
- JMS .NET client host running on WebLogic Server 10.3 or later that provides the interface between the JMS .NET client and the JMS servers.
- One or more connection hosts.
- One or more JMS servers that define a set of JMS destinations.

Traffic to the JMS servers is always routed from the .NET client through the JMS .NET client host to the connection host to the JMS servers. Traffic to the JMS .NET client is always routed from the JMS servers to the connection host and through the JMS .NET client host to the .NET client.

Note: All of the WebLogic components shown in the figure can be hosted on a single instance of WebLogic Server 10.3 or later. In a multi-server or cluster configuration, each of the WebLogic Server components can be running on a separate instance of WebLogic Server; however, the connection host and the JMS server must be in the same cluster and the JMS .NET client host must be running on WebLogic Server 10.3 or later.

A brief summary of the process used to exchange messages between the JMS .NET client and a JMS server, as illustrated in Figure 2-1, is summarized in the following steps

1. The JMS .NET client establishes an initial T3 network connection with the JMS .NET client host running on WebLogic Server 10.3 or later.
2. The JMS .NET client obtains a connection factory from the JMS .NET client host.
3. The JMS .NET client host, in turn, obtains the connection factory from JNDI.
4. The JMS .NET client creates a connection using the connection factory, which will establish a connection from the JMS .NET client host to one of the connection hosts where the connection factory resides.
5. When the JMS .NET client sends (produces) a message, the JMS .NET client host sends it to the connection host, which in turn routes it to the JMS server hosting the destination. Alternatively, when the JMS .NET client receives (consumes) a message, the connection host routes it from the JMS server hosting the destination to the JMS .NET client host, which passes the message to the JMS .NET client.

Instructions and examples for creating a JMS .NET client application are provided in Chapter 4, "Developing a Basic JMS Application Using the WebLogic JMS .NET API."

Interoperating with Previous WebLogic Server Releases

The JMS .NET client can communicate directly only with WebLogic Server 10.3 and later. To access destinations on WebLogic Server 8.1 or later that are not in the same cluster as the .NET client host running on 10.3 or later, you must configure the remote instance of WebLogic Server as a Foreign Server. For more information, see "Configuring Foreign Server Resources to Access Third-Party JMS Providers" in *Configuring and Managing WebLogic JMS*.

Note: Although you can also use Foreign Servers to connect to third-party JMS providers using WebLogic JMS, this feature is not supported in the WebLogic JMS .NET client.

Understanding the WebLogic JMS .NET API

The following table lists the primary JMS .NET API classes and interfaces used to create a JMS .NET client application. For complete details about all the classes and interfaces in the JMS .NET API, see the WebLogic Messaging API Reference documentation.

Table 2-1 WebLogic JMS .NET Classes and Interfaces

Interface	Description
`Constants`	The `Constants` class is used to define commonly used constants/enumerations for the API.
`ContextFactory`	A `ContextFactory` is used to create contexts, which are network connections from the .NET client to the client host.
`IContext`	An `IContext` object represents a network connection from the .NET client to the client host. It is used to lookup destinations and connection factories, and to close the network connection when it is no longer needed.
`IConnectionFactory`	An `IConnectionFactory` object encapsulates JMS connection configuration information. A JMS .NET client looks up a connection factory using an `IContext` object, and then uses it to create an `IConnection` with a JMS server.
`IConnection`	An `IConnection` object is the active connection between the JMS .NET client host and the JMS connection host. Authentication optionally takes place during the creation of the connection. A connection is used to create sessions.

Table 2-1 WebLogic JMS .NET Classes and Interfaces

Interface	Description
`ISession`	An `ISession` object is a single-threaded entity for producing and consuming messages. A session can create and service multiple message producers and consumers.
`IDestination`	An `IDestination` object identifies a queue or topic. Queue and topic destinations manage the messages delivered from the point-to-point and pub/sub messaging models, respectively.

`ITopic`	An `ITopic` object is pub/sub `IDestination` that encapsulates a provider-specific topic name. It is the way a client specifies the identity of a topic to JMS API methods. For those methods that use an `IDestination` as a parameter, an `ITopic` object may be used as an argument. For example, an `ITopic` can be used to create an `IMessageConsumer` and an `IMessageProducer` by calling:
	`ISession.CreateConsumer(IDestination destination)`
	`ISession.CreateProducer(IDestination destination)`

`IQueue`	An `IQueue` object is a point-to-point `IDestination` that encapsulates a provider-specific queue name. It is the way a client specifies the identity of a queue to JMS API methods.
`IMessageConsumer`	A JMS .NET client uses an `IMessageConsumer` object to receive messages from a destination. An `IMessageConsumer` object is created by passing an `IDestination` object to a message-consumer creation method supplied by a session.
`IMessageProducer`	A JMS .NET client uses an `IMessageProducer` object to send messages to a destination. An `IMessageProducer` object is created by passing an `IDestination` object to a message-producer creation method supplied by a session.

Table 2-1 WebLogic JMS .NET Classes and Interfaces

Interface	Description
`IMessage`	The `IMessage` interface is the root interface of all JMS messages. It defines the message header and the Acknowledge method used for all messages. JMS messages are composed of the following parts: Header - All messages support the same set of header fields. Header fields contain values used by both clients and providers to identify and route messages. Properties - Each message contains a built-in facility for supporting application-defined property values. Properties provide an efficient mechanism for supporting application-defined message filtering. Body - The JMS API defines several types of message body, which cover the majority of messaging styles currently in use.
`IMapMessage`	An `IMapMessage` object is used to send a set of name-value pairs. The names are String objects, and the values are primitive data types in the Java and C# programming languages. The names must have a value that is not null, and not an empty string. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined. `IMapMessage` inherits from the `IMessage` interface and adds a message body that contains a map.
`IObjectMessage`	An `IObjectMessage` object is used to send a message that contains a serializable object in the Java and C# programming languages. It inherits from the `IMessage` interface and adds a body containing a single reference to an object. Only Serializable Java objects can be used. C# objects cannot be read by Java programs, and vice versa. For more information, see "Exchanging Messages Between Different Language Environments" on page 5-8.
`IStreamMessage`	An `IStreamMessage` object is used to send a stream of primitive types in the Java programming language. It is filled and read sequentially. It inherits from the `IMessage` interface and adds a stream message body. Its methods are based largely on those found in `java.io.DataInputStream` and `java.io.DataOutputStream`.
`ITextMessage`	An `ITextMessage` object is used to send a message containing a `java.lang.String`. It inherits from the `IMessage` interface and adds a text message body.
`IBytesMessage`	An `IBytesMessage` object is used to send a message containing a stream of uninterpreted bytes. It inherits from the `IMessage` interface and adds a bytes message body. The receiver of the message supplies the interpretation of the bytes.

CHAPTER 3

Installing and Copying the WebLogic JMS .NET Client Libraries

These sections describe the JMS .NET client components installed on a WebLogic Server platform, the location to which they are installed, and how to copy them to a .NET Framework machine.

Installing the WebLogic JMS .NET Client

The WebLogic JMS .NET client is bundled with WebLogic Server 10.3 and later and is installed automatically when you install WebLogic Server on a supported platform, including non-Windows platforms. For a list of supported platforms for WebLogic Server, see *Supported Configurations*.

For details about installing WebLogic Server, see the *Installation Guide*.

Note: A valid JMS license is required on the WebLogic Server platform hosting the JMS .NET client host. No license is required on the client itself.

Location of Installed Components

The WebLogic JMS .NET client is installed in the following directory on the WebLogic Server platform:

BEA_HOME/modules/com.bea.weblogic.jms.dotnetclient_1.0.0.0

where *BEA_HOME* is the top-level installation directory for BEA products that you selected during the installation process.

The JMS .NET client installation consists of the following components:

- `WebLogic.Messaging.dll`—The fully-managed JMS .NET client library used by the client for the JMS client application.
- `WebLogic.Messaging.pdb`—The debug version of the JMS .NET client library that can be used by the client, together with the `WebLogic.Messaging.dll`, to debug the JMS .NET client application.
- `jms.dotnet.api.zip`—HTML and Windows help-style documentation for the WebLogic JMS .NET API

Copying the Library to the Client Machine

After installing WebLogic Server on a supported platform, you need to copy the `WebLogic.Messaging.dll` library from the installation directory specified in "Location of Installed Components" on page 3-1 to your development directory on a supported .NET client machine, and you need to ensure that your .NET application references the library. The JMS .NET client is a fully-managed runtime library that is supported on the following Windows platforms running version 2.0 or 3.0 of the .NET Framework:

- Windows 2003
- Windows XP
- Windows Vista

If you are using Visual Studio®, you can add the `WebLogic.Messaging.dll` as a reference assembly in your project as follows:

1. Select *Project*→References
2. Select Add Reference and specify the `WebLogic.Messaging.dll` from the directory into which you copied it on the .NET machine

Optionally, you can also copy the debug version of the JMS .NET client library, `WebLogic.Messaging.pdb`, and the API documentation to your client machine, but it is not required.

CHAPTER 4

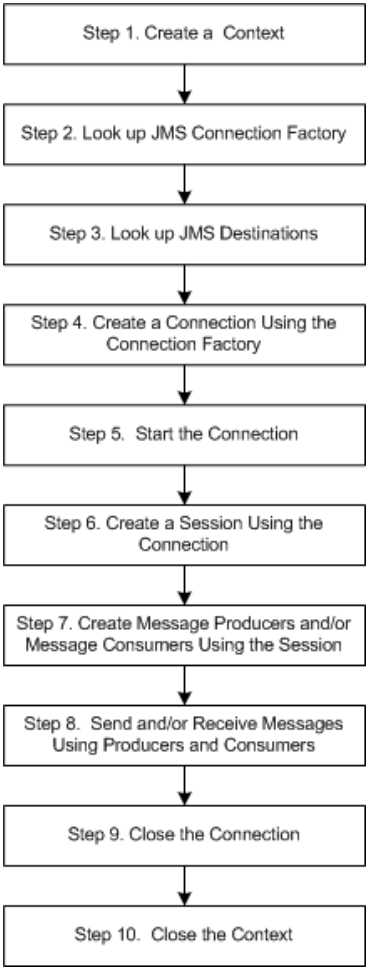
Developing a Basic JMS Application Using the WebLogic JMS .NET API

The process for developing a JMS application using the WebLogic JMS .NET client is very similar to the process used to develop a Java client. These sections provide information on the steps required to develop a basic JMS application in C# using the JMS .NET API.

Creating a JMS .NET Client Application

The following flowchart illustrates the steps in a basic JMS .NET application.

Figure 4-1 Basic Steps in a JMS .NET Client Application



Note: Creating and closing resources has relatively higher overhead in comparison to sending and receiving messages. BEA recommends that contexts be shared between threads, and that other resources be cached for reuse. For more information, see "Best Practices" on page 5-13.

Example: Writing a Basic PTP JMS .NET Client Application

The following example shows how to create a basic PTP JMS .NET client application, written in C#. It uses synchronous receive on a queue configured using auto acknowledge mode. A complete copy of the example is provided in Appendix A, "JMS .NET Client Sample Application."

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

For more information about the .NET API classes and methods used in this example, see "Understanding the WebLogic JMS .NET API" on page 2-5, or the WebLogic Messaging API Reference documentation.

At the beginning of your program, be sure to define the required variables, including the WebLogic Server host, the connection factory, and the queue and topic names:

```
using System;
using System.Collections;
using System.Collections.Generic;

using WebLogic.Messaging;

public class MessagingSample
{
  private string host      = "localhost";
  private int    port      = 7001;
  private string cfName    = "weblogic.jms.ConnectionFactory";
  private string queueName = "jms.queue.TestQueue1";
```

Step 1

Create a context to establish a network connection to the WebLogic Server host and optionally login.

```
IDictionary<string, Object> paramMap = new Dictionary<string, Object>();

paramMap[Constants.Context.PROVIDER_URL] =
 "t3://" + this.host + ":" + this.port;
```

```
IContext context = ContextFactory.CreateContext(paramMap);
```

Notes: The Provider_URL may contain multiple addresses, separated by commas, using the following format:

```
t3://host:port,host:port
```

In this case, the context will try each address in turn until one succeeds or they all fail.

You also have the option of supplying a username and password with the initial context, as follows:

```
paramMap[Constants.Context.SECURITY_PRINCIPAL] = username;
paramMap[Constants.Context.SECURITY_CREDENTIALS] = password;
```

Step 2

Look up the JMS connection factory.

```
IConnectionFactory cf = context.LookupConnectionFactory(this.cfName);
```

Step 3

Look up JMS destination resources in the context using their configured JNDI names.

```
IQueue queue = (IQueue)context.LookupDestination(this.queueName);
```

Step 4

Create a connection using the connection factory. This establishes a JMS connection from the .NET client host to the JMS connection host. The connection host will be one of the servers that is in the configured target list for the connection factory, and which can be the same as the .NET client host.

```
IConnection connection = cf.CreateConnection();
```

Step 5

Start the connection to allow the consumers to get messages.

```
connection.Start();
```

Step 6

Create a session using the `AUTO_ACKNOWLEDGE` acknowledge mode.

Note: Sessions are not thread safe. Use multiple sessions if you need to run producers and/or consumers concurrently. For an example using multiple sessions, see the asynchronous example in Appendix A, "JMS .NET Client Sample Application."

```
ISession session = connection.CreateSession(
/* transacted= */ false, Constants.SessionMode.AUTO_ACKNOWLEDGE);
```

Step 7

Create a message producer and send a persistent message.

```
IMessageProducer producer = session.CreateProducer(queue);

producer.DeliveryMode = Constants.DeliveryMode.PERSISTENT;

ITextMessage sendMessage = session.CreateTextMessage("My q message");

producer.Send(sendMessage);

PrintMessage("Sent Message:", sendMessage);
```

Step 8

Create a message consumer and receive a message. Note that the message is automatically deleted from the server because the session was created in `AUTO_ACKNOWLEDGE` mode, as shown in Step 6.

```
IMessageConsumer consumer = session.CreateConsumer(queue);

IMessage recvMessage = consumer.Receive(500);

PrintMessage("Received Message:", recvMessage);
```

Step 9

Close the connection. Note that closing a connection also closes its child sessions, consumers, and producers.

```
connection.Close();
```

Step 10

Close the context.

```
context.CloseAll();
```

Note: `context.Close()` does not terminate the network connection until all the IConnections have been closed.
`context.CloseAll()` closes the network connection and all open IConnections.

Using Advanced Concepts in JMS .NET Client Applications

Appendix A, "JMS .NET Client Sample Application," provides a complete example of a JMS .NET client application, written in C#, that demonstrates some of the following advanced concepts:

- The use of local transactions instead of acknowledge modes.
- Message persistence. For more information, see "Persistent vs. Non-Persistent Messages" in *Programming WebLogic JMS*.
- Acknowledge modes. For more information, see "Non-Transacted Session" in *Programming WebLogic JMS*.
- Exception listeners. For more information, see "Best Practices" on page 5-13.
- Durable Subscriptions. For more information, see "Setting Up Durable Subscriptions" in *Programming WebLogic JMS*.

For guidelines in the use of other advanced concepts in the JMS .NET client such as interoperability, security, and best practices, see Chapter 5, "Programming Considerations."

CHAPTER 5

Programming Considerations

These sections provide programming considerations and best practices to use when creating a JMS .NET client application:

Using the WebLogic JMS Extensions

Table 5-1 lists the WebLogic JMS extensions that are supported in this release of the JMS .NET client. There are several ways that messaging can be configured:

- On the connection factory—This method defines the default configuration settings.
- Programmatically in the application using the API—Certain settings may override the connection factory configuration.
- On the server—Certain settings may override both the connection factory and programmatic configuration.

In some cases, there are differences in the way that an extension is configured, or in the behavior, between a JMS .NET client and a Java client. For example, some extensions cannot be enabled programmatically using the JMS .NET API, and can only be enabled via configuration. The following table summarizes the differences. Additional details, if required, are provided in the subsequent sections.

Table 5-1 WebLogic JMS Extensions Supported in the JMS .NET Client

Feature	Configurable on Connection Factory	Configurable on the Server	Java API	JMS .NET API	Comments
Distributed Destinations (Uniform and Weighted) For more information, see: • "Using Distributed Destinations" in *Programming WebLogic JMS* • "Configuring Distributed Destination Resources" in *Configuring and Managing WebLogic JMS*	Yes	Yes	No	No	
Flow Control For more information, see: "Controlling the Flow of Messages on JMS Servers and Destinations" in *WebLogic Server Performance and Tuning*	Yes	Yes	No	No	

Table 5-1 WebLogic JMS Extensions Supported in the JMS .NET Client (Continued)

Feature	Configurable on Connection Factory	Configurable on the Server	Java API	JMS .NET API	Comments
Blocking producers during quota conditions For more information, see "Defining a Send Timeout on Connection Factories" in *WebLogic Server Performance and Tuning*	Yes	Yes	No	No	
Foreign destinations for remote instances of WebLogic Server For more information, see "Configuring Foreign Server Resources to Access Third-Party JMS Providers" in *Configuring and Managing WebLogic JMS*	No	Yes	No	No	See "Interoperating with Previous WebLogic Server Releases" on page 2-5.
Imported store-and-forward (SAF) destinations For more information, see "Imported SAF Destinations" in *Configuring and Managing WebLogic Store-and-Forward*	No	Yes	No	No	
Redelivery limit For more information, see "Setting a Redelivery Limit for Messages" in *Programming WebLogic JMS*	No	Yes	Yes	No	
Redelivery delay For more information, see "Setting a Redelivery Delay for Messages" in *Programming WebLogic JMS*	Yes	No	Yes	No	
Error destinations For more information, see "Configuring an Error Destination for Undelivered Messages" in *Programming WebLogic JMS*	No	Yes	No	No	

Table 5-1 WebLogic JMS Extensions Supported in the JMS .NET Client (Continued)

Feature	Configurable on Connection Factory	Configurable on the Server	Java API	JMS .NET API	Comments
WLDestination.getCreateDestination Argument	No	No	Yes	Yes	
No Acknowledge Mode For more information, see "Using NO_ACKNOWLEDGE" in *Programming WebLogic JMS*	No	No	Yes	Yes	
Unit of order For more information, see: • "Using Message Unit-of-Order" in *Programming WebLogic JMS* • "Tuning Applications Using Unit-of-Order" in *WebLogic Server Performance and Tuning*	Yes	Yes	Yes	Yes	See "Unit-of-Order" on page 5-5
Scheduled message delivery For more information, see "Setting Message Delivery Times" in *Programming WebLogic JMS*	Yes	Yes	Yes	Yes	See "Message Delivery Time" on page 5-6.
Messages Maximum pipeline For more information, see "Asynchronous Message Pipeline" in *Programming WebLogic JMS*	Yes	No	Yes	No	
Message Compression For more information, see "Message Compression" in *Programming WebLogic JMS*	Yes	No	Yes	No	See "Message Compression" on page 5-5.
Quotas For more information, see "Defining Quota" in *WebLogic Server Performance and Tuning*	No	Yes	No	No	

Table 5-1 WebLogic JMS Extensions Supported in the JMS .NET Client (Continued)

Feature	Configurable on Connection Factory	Configurable on the Server	Java API	JMS .NET API	Comments
One-way message sends For more information, see "Using One-Way Message Sends For Improved Non-Persistent Messaging Performance" in *WebLogic Server Performance and Tuning*	Yes	No	No	No	See "One-Way Message Sends" on page 5-6.
Acknowledge policy	Yes	No	No	No	
`JMSXUserID`	Yes	Yes	No	No	These properties are set by the server. You can retrieve the value set by the server through the API.
`JMSXDeliveryCount`	No	No	No	No	These properties are set by the server. You can retrieve the value set by the server through the API.

Message Compression

In this release, automatic message compression is not supported for client sends between the JMS .NET client and the JMS .NET client host running on WebLogic Server. However, if the compression settings are set on the connection factory, message compression behavior between the .NET client host and the destination is the same as that of the Java client. Compressed messages are decompressed by the JMS .NET client host on the server side when they are received by the .NET client.

For more information, see "Message Compression" in *Programming WebLogic JMS*

Unit-of-Order

The method used to specify Unit-of-Order (UOO) in the JMS .NET API differs from the Java API. To set Unit-of-Order in the JMS .NET API, add a string property named

`Constants.MessagePropertyNames.UNIT_OF_ORDER_PROPERTY_NAME` to the message with the desired UOO.

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

Unit-of-Work

The WebLogic JMS Unit-of-Work (UOW) extension is not supported in the JMS .NET client in this release. If a .NET client attempts to set a UOW property on a message, a `System.NotImplementedException` is generated. In addition, a .NET consumer cannot receive UOW messages with deserializable content that are sent by a Java client. In this case, the consumer gets a `MessageFormatException` when it calls the `ObjectMessage.getObject()` method on the `ObjectMessage`.

Message Delivery Time

The method used to specify message delivery times in the JMS .NET API differs from the Java API. To set message delivery times in the JMS .NET API, add a string property named `Constants.MessagePropertyNames.DELIVERY_TIME_PROPERTY_NAME` to the message, where the value is the number of milliseconds in the future in which the message will be delivered.

One-Way Message Sends

Although you can configure one-way message sends on the connection factory, this behavior is not fully supported in the JMS .NET client. Messages sent as one-way sends will actually be two-way sends between the .NET client and the .NET client host, and one-way sends between the .NET client host and the JMS connection host.

Limitations of Using the WebLogic JMS .NET Client

The following sections describe the JMS features that are not supported in the JMS .NET client.

Unsupported JMS 1.1 Standard Features

In this release, the following JMS 1.1 standard features are not supported:

- Creating and closing temporary destinations (`javax.jms.TemporaryQueue` and `javax.jms.TemporaryTopic`). The JMS .NET client can still produce messages to temporary destinations created by Java clients.

- `javax.jms.QueueRequester` and `javax.jms.TopicRequester`. (These helper classes are related to temporary destinations.)
- Queue browsers: `javax.jms.QueueBrowser`.
- Queue and Topic interfaces (`QueueConnectionFactory`, `TopicConnectionFactory`, `QueueConnection`, `TopicConnection`, `QueueSession`, `TopicSession`). These queue and topic interfaces are legacy JMS 1.0.2 interfaces that have been superseded by the JMS 1.1 common interfaces.

Unsupported JMS 1.1 Optional Features

In this release, the following JMS 1.1 optional features are not supported:

- XA interfaces (`XAConnectionFactory`, `XAConnection`, and `XASession`).
- Participation in global XA transactions (See "Transactions" on page 5-7).
- Connection Consumer and Server session pools (`javax.jms.ConnectionConsumer`, `ServerSessionPool`, and `ServerSession`). These are optional capabilities that have been superseded by Jave EE MDBs, and are not supported by the WebLogic Java JMS client.
- `MessageProducer.setDisableMessageTimestamp` method. Note that the WebLogic JMS Java client ignores this method.

Unsupported WebLogic JMS Extensions

In this release, the following WebLogic JMS extensions are not supported:

- SSL
- HTTP tunneling
- SAF Client—See "Reliably Sending Messages Using the JMS SAF Client"
- Multicast Subscribers—See "Using Multicasting with WebLogic JMS"
- Automatic Reconnect—See "Automatic JMS Client Failover"

Transactions

In this release, the JMS .NET client supports transacted sessions as defined in the JMS Specification only. Transacted sessions provide a standard local transaction capability. As with

the Java client, one or more WebLogic JMS destinations from within the same cluster may participate in a transacted session local transaction, but no other resources may participate (such as JMS servers in other clusters, databases, or foreign JMS providers).

Global XA transactions are not supported, therefore JMS cannot participate in a .NET transaction. The XA setting of the connection factory is ignored by the .NET client. The JMS NET client operations cannot participant in any .NET transactions.

Exchanging Messages Between Different Language Environments

The following Java JMS message types can be exchanged between a .NET producer and a Java or C consumer, and vice versa:

- `Message`
- `BytesMessage`
- `StreamMessage`
- `MapMessage`
- `TextMessage`

An `ObjectMessage` type, however, can be sent from one language and received by another, but the message cannot be interpreted unless it is written in the same language. The producer and consumer of an `OBJECTMESSAGE` type must be written in the same language, either C# or Java. If a mismatch occurs; that is, if a .NET `ObjectMessage` is received by a Java consumer, or a Java `ObjectMessage` is received by a .NET consumer, then `message.getObject()` throws a `MessageFormatException`.

Implementing Security With the JMS .NET Client

You need to be aware of the following security considerations when creating a JMS .NET client:

- To access *secure* JNDI and JMS resources on the server, the JMS .NET client application can supply a username and password as follows:
  - When establishing the initial context to the server using `ContextFactory.CreateContext()`. The credentials supplied when creating the initial context are used for authentication to gain access to secure JNDI and JMS resources on the server.

  - When creating a connection using the `IConnectionFactory.createConnection()` method. In this case, the credentials supplied when creating a connection override the credentials supplied during the initial context. That is, if user `Fred` is supplied during initial context, and user `Tony` is supplied when the connection is created, the user `Tony` credential is used for authentication to gain access to secure JMS resources.

  Note: In both instances, the password is encrypted. If the resources are not secured, a username and password is optional.

- Authentication for the .NET client is associated with the JMS object that invokes the secured resource. That is, the credential for a JMS object is inherited from the parent JMS context, or from the connection override if credentials are supplied when creating the connection. This differs from Java client security where credentials are associated with the current thread.
- SSL is not supported for the JMS .NET client in this release. Therefore, it is important that you secure the networking services that the operating system provides, as well as any networking connections. For more information, see "Securing Network Connections" in *Securing a Production Environment.*
- Although usernames and passwords are protected, and passwords are encrypted, a sophisticated user or intruder might be able to defeat the protection mechanisms. Be sure to secure any network connections when usernames and passwords are provided.
- Similar to the Java client, the JMS.NET client does not support message level encryption.

Understanding Socket and Threading Behavior

WebLogic JMS .NET clients share the same WebLogic Server T3 port as other types of WebLogic clients. When an `IContext` initial context is created by a .NET client using the `ContextFactory` class, the client specifies a URL that references a T3 capable port on the server, and a socket pair is implicitly created to service the requested network connection. The socket pair consists of one socket on the client and another socket on the WebLogic Server JMS .NET client host. All JMS operations on JMS objects obtained from the .NET context route through the implicit network connection of the context.

If two concurrent `IContext` initial context instances on the same .NET CLR connect to the same WebLogic Server JMS .NET client host, then two network connections are created. Each network connection has its own pair of sockets: a server-side socket and a client-side socket. Therefore, when two network connections are created, two sockets are created on the CLR client and two sockets are created on the WebLogic Server acting as the JMS .NET client host. This contrasts with WebLogic Java clients, which automatically detect and close duplicate network connections

to a remote JVM and, instead, implicitly multiplex all traffic to and from a particular remote JVM over a single network connection.

A server-side socket for a JMS .NET client is serviced by the same WebLogic Server socket-reader muxer thread pool as other types of WebLogic clients. When working on behalf of JMS .NET client requests, the socket-reader muxer thread pool reads the incoming requests from the socket and dispatches work into the WebLogic Server default thread pool which, in turn, processes the requests and sends the responses back to the client.

On a JMS .NET client, a new internal thread is automatically created for each network connection (that is, per `IContext` initial context instance). This dedicated thread reads all incoming data on the client socket and dispatches the related work into the CLR thread pool. This means that .NET client application asynchronous message event handlers run in the CLR thread pool.

Note: The CLR thread pool is supplied by the .NET Framework `System.Threading.ThreadPool` class. There is one thread pool per process. The thread pool has a default size of 25 threads per available processor, however, you can change the number of threads in the thread pool using the `ThreadPool.SetMaxThreads` method. Each thread in the thread pool uses the default stack size and runs at the default priority. For more information, refer to the Microsoft .NET Framework documentation for the `System.Threading.ThreadPool` class.

For JMS .NET applications that create many concurrent initial contexts that all connect to the same WebLogic Server .NET client host, you may obtain performance improvements by modifying the application so that it uses a single shared initial context. A shared context ensures that the client only creates a single network connection.

Data Type Conversions

Endian Conversion

Java and .NET use different byte order formats for storing primitive types:

- Microsoft Windows .NET uses the Little-Endian (low-order) format
- Java uses the Big-Endian (high-order) format

To support interoperability between Java and .NET, data is transferred over the network using the Big-Endian format. When a .NET application uses the JMS .NET API to read and write primitives, data is automatically converted between Big-Endian and Little-Endian, as needed. For example, if you use `BytesMessage.WriteInt` in the JMS .NET API, the data is always stored as Big Endian and can be read using both the Java API and the JMS .NET API bytes message read integer methods.

For specialized applications that do not use the JMS .NET API to pass primitives, but instead transfer primitive data using raw byte arrays, you need to manually convert the byte format to Big Endian when communicating with Java. If you need to perform a manual Endian conversion in your application, you can use the following helper methods from the utility class `WebLogic.Messaging.Transport.Util.EndianConvertor` provided in the JMS .NET client library:

```
public static char SwitchEndian(char x)
public static short SwitchEndian(short x)
public static int SwitchEndian(int x)
public static long SwitchEndian(long x)
public static ushort SwitchEndian(ushort x)
public static uint SwitchEndian(uint x)
public static ulong SwitchEndian(ulong x)
public static double SwitchEndian(double x)
public static float SwitchEndian(float x)
public static byte[] SwitchEndian(byte[] x)
```

For example, the standard .NET classes *`System.IO.BinaryReader`* and *`System.IO.BinaryWriter`* for reading and writing primitives to raw byte arrays use Little Endian. The following code snippet illustrates how to store and retrieve an integer to/from a .NET byte array:

```
binaryWriter.WriteInt(EndianConverter.SwitchEndian(i))
i=EndianConverter.SwitchEndian(binaryReader.ReadInt())
```

Signed and Unsigned Byte Conversions

With the exception of the `byte` data type, there is an equivalent C# data type, with the same name, for every Java primitive data type. The following table lists the different names used for signed and unsigned bytes in C# and Java.

Table 5-2 Byte Primitive Data Type in C# and Java

C#	Java	Description
`byte`	N/A	Unsigned byte
`sbyte`	`byte`	Signed byte

As shown in Table 5-2, Microsoft .NET supports both `byte` (unsigned byte) and `sbyte` (signed byte) as primitive data types, but Java supports only `byte` (signed byte) as a direct primitive type. The standard convention in both languages is to use the `byte` data type; however, in .NET this represents an unsigned byte and in Java this represents a signed byte.

For interoperability between .NET and Java, the JMS .NET client allows only the use of the signed byte for reading and writing bytes. There is no difference between signed bytes and unsigned bytes when the byte value is 127 or less. An unsigned byte with a value of 127 or less is stored as an `sbyte`. However, if a .NET client needs to store an unsigned byte with a value greater than 127 in a signed byte, it needs to be converted from a signed byte to an unsigned byte. The following samples illustrate conversion methods that you can use to read and write an unsigned byte as a signed byte:

- Byte Conversion in C#

  An unsigned byte value of 255 can be passed as a signed byte as follows:

  ```
  – byte unsignedByteValue = 255;
    sbyte signedByteValue = unchecked ( (sbyte)unsignedByteValue ); //
    converted signed value=-1
  ```

  Similarly, you can use the following method to convert a signed byte value to an unsigned byte value:

  ```
  – sbyte signedByteValue = -1;
    byte unsignedByteValue = unchecked ( (byte)signedByteValue ); //
    converted unsigned value=255
  ```

- Byte Conversion in Java

  The unsigned value can be read as a signed byte and converted to an unsigned byte value as follows:

  ```
  – byte signedByteValue = -1;
    int unsignedByteValue = 0xFF &  signedByteValue; //converted signed
    value = 255
  ```

An unsigned value can be written as follows:

```
– Int unsignedByteValue = 255;
  byte signedByteValue = 0xFF & unsignedByteValue; // converted
  signed value=-1
```

The JMS .NET API only allows for storing single bytes as signed bytes. When the JMS .NET API is used to retrieve `sbyte` values as `short`, `int`, `long`, or `string`, the value is treated as an `sbyte`, not an unsigned byte. For example, if the unsigned byte value 255 is stored using `message.SetByteProperty("myvalue", unchecked( (sbyte)((byte)255) ))`, a call to `message.GetByteProperty("myvalue")` or `message.GetShortProperty("myvalue")` returns "-1".

Byte Array Transfers

When transferring byte arrays from the JMS .NET client to WebLogic JMS, all byte arrays `(byte[])` are passed as is (that is, there is no conversion from unsigned to signed.) Therefore, no data is lost in the translation.

Best Practices

The following list identifies best practices to use when creating a JMS .NET client application:

- Always register a connection exception listener using an `IConnection` if the application needs to take action when an idle connection fails. The connection exception listener is asynchronously notified when there is a communications failure between the .NET client and the .NET client WebLogic host, or between the WebLogic host and the JMS connection host. Applications may choose to implement the connection exceptions listener callback to close all open resources and then periodically attempt a reconnect.
- To obtain performance improvements, have multiple .NET client threads share a single context to ensure that they use a single socket. For more information, see "Understanding Socket and Threading Behavior" on page 5-9. It is important to note that a context creates a socket and that closing the context closes the socket.
- Cache and reuse frequently accessed JMS resources, such as contexts, connections, sessions, producers, destinations, and connection factories. Creating and closing these resources consumes significant CPU and network bandwidth.
- With the exception of `close()` methods, JMS sessions and their child resources are not thread safe. For example, do not call a producer `send()` in one thread, and a consumer `receive()` in parallel in another thread, if the producer and consumer were created using the same session. As another example, do not call any method other than `close()` in an

arbitrary thread for sessions that have asynchronous consumers because a message may arrive and invoke the callback at the same time.

- Use DNS aliases or comma separated addresses for load balancing JMS .NET clients across multiple JMS .NET client host servers in a cluster. In this release, the JMS .NET client does not support automatic cluster load balancing as it is implicitly supplied with the Java client.

APPENDIX A

JMS .NET Client Sample Application

MessagingSample.cs

The following .NET client sample program, written in C#, provides an overview of the basic features of the WebLogic JMS .NET API. For details about the API, see the WebLogic Messaging API Reference documentation.

To make a copy of this sample and maintain the formatting, display the MessagingSample.cs file in a supported browser, and copy and paste the text into the editor of your choice.

Listing A-1 MessagingSample.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Threading;

using WebLogic.Messaging;


/// <summary> Demonstrate the WebLogic JMS .NET API.
/// <para>
/// Copyright (c) 2007 BEA Systems, Inc.
/// </para>
/// <para>
```

```
/// This command line program connects to WebLogic JMS and performs
/// queue and topic messaging operations.  It is supported with
/// versions ?10.3? and later.  To compile the program,
/// link it with "WebLogic.Messaging.dll".   For usage information,
/// run the program with "-help" as a parameter.
/// </para>
/// </summary>

public class MessagingSample
{
  private static string NL = Environment.NewLine;

  private string host      = "localhost";
  private int    port      = 7001;

  private string cfName    = "weblogic.jms.ConnectionFactory";
  private string queueName = "jms.queue.TestQueue1";
  private string topicName = "jms.topic.TestTopic1";

  private static string USAGE =
    "Usage: " + Environment.GetCommandLineArgs()[0] + NL +
    "           [-host <hostname>] [-port <portnum>] " + NL +
    "           [-cf <connection factory JNDI name>] " + NL +
    "           [-queue <queue JNDI name>] [-topic <topic JNDI name>]";


  public static void Main(string[] args)
  {
    try {
      MessagingSample ms = new MessagingSample();

      // override defaults with command line arguments
      if (!ms.ParseCommandLine(args)) return;

      ms.DemoSyncQueueReceiveWithAutoAcknowledge();

      ms.DemoAsyncNondurableTopicConsumerAutoAcknowledge();
```

```
    ms.DemoSyncTopicDurableSubscriberClientAcknowledge();

  } catch (Exception e) {
    Console.WriteLine(e);
  }
}


private void DemoSyncQueueReceiveWithAutoAcknowledge()
{
  Console.WriteLine(
    NL + "-- DemoSyncQueueReceiveWithAutoAcknowledge -- " + NL);

  // ------------------------------------------------
  // Make a network connection to WebLogic and login:
  // ------------------------------------------------

 IDictionary<string, Object> paramMap = new Dictionary<string, Object>();

  paramMap[Constants.Context.PROVIDER_URL] =
  "t3://" + this.host + ":" + this.port;

  IContext context = ContextFactory.CreateContext(paramMap);

  try {
    // -------------------------------------
    // Look up our resources in the context:
    // -------------------------------------

    IConnectionFactory cf = context.LookupConnectionFactory(this.cfName);

    IQueue queue = (IQueue)context.LookupDestination(this.queueName);

    // ------------------------------------------------
    // Create a connection using the connection factory:
    // ------------------------------------------------

    IConnection connection = cf.CreateConnection();
```

```
// ------------------------------------------------------------------
// Start the connection in order to allow receivers to get messages:
// ------------------------------------------------------------------

connection.Start();

// -----------------
// Create a session:
// -----------------
// IMPORTANT:  Sessions are not thread-safe.   Use multiple sessions
// if you need to run producers and/or consumers concurrently. For
// more information, see the asynchronous consumer example below.
//

ISession session = connection.CreateSession(
   /* transacted= */ false, Constants.SessionMode.AUTO_ACKNOWLEDGE);

// ------------------------------------------------
// Create a producer and send a persistent message:
// ------------------------------------------------

IMessageProducer producer = session.CreateProducer(queue);

producer.DeliveryMode = Constants.DeliveryMode.PERSISTENT;

ITextMessage sendMessage = session.CreateTextMessage("My q message");

producer.Send(sendMessage);

PrintMessage("Sent Message:", sendMessage);

// ---------------------------------------
// Create a consumer and receive a message:
// ---------------------------------------
// The message will automatically be deleted from the server as the
// consumer's session was created in AUTO_ACKNOWLEDGE mode.
//
```

```
    IMessageConsumer consumer = session.CreateConsumer(queue);

    IMessage recvMessage = consumer.Receive(500);

    PrintMessage("Received Message:", recvMessage);

   // --------------------------------------------------------------------
    // Close the connection.   Note that closing a connection also closes
    // its child sessions, consumers, and producers.
   // --------------------------------------------------------------------

    connection.Close();

  } finally {

   // --------------------------------------------------------------------
    // Close the context.  The CloseAll method closes the network
    // connection and all related open connections, sessions, producers,
    // and consumers.
   // --------------------------------------------------------------------

    context.CloseAll();
  }
}

// Implement a MessageEventHandler delegate.  It will receive
// asynchronously delivered messages.

public void OnMessage(IMessageConsumer sender, MessageEventArgs args) {
  PrintMessage("Received Message Asynchronously:", args.Message);
}

private void DemoAsyncNondurableTopicConsumerAutoAcknowledge()
{
  Console.WriteLine(
    NL + "-- DemoAsyncNondurableTopicConsumerAutoAcknowledge -- " + NL);
```

```
// ------------------------------------------------
// Make a network connection to WebLogic and login:
// ------------------------------------------------

IDictionary<string, Object> paramMap = new Dictionary<string, Object>();

paramMap[Constants.Context.PROVIDER_URL] =
 "t3://" + this.host + ":" + this.port;

IContext context = ContextFactory.CreateContext(paramMap);

try {
  // -------------------------------------
  // Look up our resources in the context:
  // -------------------------------------

  IConnectionFactory cf = context.LookupConnectionFactory(this.cfName);

  ITopic topic = (ITopic)context.LookupDestination(this.topicName);

  // ------------------------------------------------------------
  // Create a connection using the connection factory:
  // ------------------------------------------------------------

  IConnection connection = cf.CreateConnection();

  // ------------------------------------------------------------------
  // Start the connection in order to allow receivers to get messages:
  // ------------------------------------------------------------------

  connection.Start();

  // ----------------------------------------
  // Create the asynchronous consumer delegate.
  // ----------------------------------------
  // Create a session and a consumer; also designate a delegate
  // that listens for messages that arrive asynchronously.
  //
```

```
// Unlike queue consumers, topic consumers must be created
// *before* a message is sent in order to receive the message!
//
// IMPORTANT:  Sessions are not thread-safe.  We use multiple sessions
// in order to run the producer and async consumer concurrently.  The
// consumer session and any of its producers and consumers
// can no longer be used outside of the OnMessage
// callback once OnMessage is designated as its event handler, as
// messages for the event handler may arrive in another thread.
//

ISession consumerSession = connection.CreateSession(
   /* transacted= */ false, Constants.SessionMode.AUTO_ACKNOWLEDGE);

IMessageConsumer consumer = consumerSession.CreateConsumer(topic);

consumer.Message += new MessageEventHandler(this.OnMessage);

// -------------
// Send Message:
// -------------
// Create a producer and send a non-persistent message.  Note
// that even if the message were sent as persistent, it would be
// automatically downgraded to non-persistent, as there are only
// non-durable consumers subscribing to the topic.
//

ISession producerSession = connection.CreateSession(
   /* transacted= */ false, Constants.SessionMode.AUTO_ACKNOWLEDGE);

IMessageProducer producer = producerSession.CreateProducer(topic);

producer.DeliveryMode = Constants.DeliveryMode.NON_PERSISTENT;

ITextMessage sendMessage = producerSession.CreateTextMessage(
                              "My topic message");

producer.Send(sendMessage);
```

```
        PrintMessage("Sent Message:", sendMessage);

        // -----------------
        // Wait for Message:
        // -----------------
        // Sleep for one second to allow the delegate time to receive and
        // automatically acknowledge the message.  The delegate will print
        // to the console when it receives the message.
        //

        Thread.Sleep(1000);


        // ---------
        // Clean Up:
        // ---------
        // We could just call connection.Close(), which would close
        // the connection's sessions, etc, or we could even just
        // call context.CloseAll(), but we want to demonstrate closing each
        // individual resource.
        //

        producer.Close();
        consumer.Close();
        producerSession.Close();
        consumerSession.Close();
        connection.Close();

    } finally {

        // ------------------------------------------------------------
        // Close the context.  The CloseAll method closes the network
        // connection and any open JMS connections, sessions, producers,
        // or consumers.
        // ------------------------------------------------------------

        context.CloseAll();
```

```
  }
}

private void DemoSyncTopicDurableSubscriberClientAcknowledge() {

  Console.WriteLine(
    NL + "-- DemoSyncTopicDurableSubscriberClientAcknowledge -- " + NL);

  // ------------------------------------------------
  // Make a network connection to WebLogic and login:
  // ------------------------------------------------

 IDictionary<string, Object> paramMap = new Dictionary<string, Object>();

  paramMap[Constants.Context.PROVIDER_URL] =
  "t3://" + this.host + ":" + this.port;

  IContext context = ContextFactory.CreateContext(paramMap);

  try {
    // --------------------------------------
    // Look up our resources in the context:
    // --------------------------------------

    IConnectionFactory cf = context.LookupConnectionFactory(this.cfName);

    ITopic topic = (ITopic)context.LookupDestination(this.topicName);

    // ------------------------------------------------
    // Create a connection using the connection factory:
    // ------------------------------------------------

    IConnection connection = cf.CreateConnection();

    // -------------------------------------------
    // Assign a unique client-id to the connection:
    // -------------------------------------------
    // Durable subscribers must use a connection with an assigned
```

```
// client-id.   Only one connection with a given client-id
// can exist in a cluster at the same time.  An alternative
// to using the API is to configure a client-id via connection
// factory configuration.

connection.ClientID = "MyConnectionID";

// ------------------------------------------------------------------
// Start the connection in order to allow consumers to get messages:
// ------------------------------------------------------------------

connection.Start();

// -----------------
// Create a session:
// -----------------
// IMPORTANT:  Sessions are not thread-safe.   Use multiple sessions
// if you need to run producers and/or consumers concurrently. For
// more information, see the asynchronous consumer example above.
//

ISession session = connection.CreateSession(
  /* transacted= */ false, Constants.SessionMode.CLIENT_ACKNOWLEDGE);

// ------------------------------------------------
// Create a durable subscription and its consumer.
// ------------------------------------------------
// Only one consumer at a time can attach to the durable
// subscription for connection ID "MyConnectionID" and
// subscription ID "MySubscriberID.
//
// Unlike queue consumers, topic consumers must be created
// *before* a message is sent in order to receive the message!
//

IMessageConsumer consumer = session.CreateDurableSubscriber(
  topic, "MySubscriberID");
```

```
        // ------------------------------------------------
        // Create a producer and send a persistent message:
        // ------------------------------------------------

        IMessageProducer producer = session.CreateProducer(topic);

        producer.DeliveryMode = Constants.DeliveryMode.PERSISTENT;

        ITextMessage sendMessage = session.CreateTextMessage("My durable
message");

        producer.Send(sendMessage);

        PrintMessage("Sent Message To Durable Subscriber:", sendMessage);

        // ------------------------------------------------------
        // Demonstrate closing and re-creating the consumer.
        //
        // The new consumer will implicitly connect to the durable
        // subscription created above, as we specify the same
        // connection id and subscription id.
        //
        // A durable subscription continues to exist and accumulate
        // new messages when it has no consumer, and even keeps
        // its persistent messages in the event of a client or server
        // crash and restart.
        //
        // Non-durable subscriptions and their messages cease to
        // exist when they are closed, or when their host server
        // shuts down or crashes.
        // ------------------------------------------------------

        consumer.Close();

        consumer = session.CreateDurableSubscriber(
          topic, "MySubscriberID");

        //
```

```
-----------------------------------------------------------------
      // Demonstrate client acknowledge.  Get the message, force
     // it to redeliver, get it again, and then finally delete the message.
      //
-----------------------------------------------------------------
      // In client ack mode "recover()" forces message redelivery, while
      // "acknowledge()" deletes the message.  If the client application
      // crashes or closes without acknowledging a message, it will be
      // redelivered.

      ITextMessage recvMessage = (ITextMessage)consumer.Receive(500);

      PrintMessage("Durable Subscriber Received Message:", recvMessage);

      session.Recover();

      recvMessage = (ITextMessage)consumer.Receive(500);

      PrintMessage("Durable Subscriber Received Message Again:",
recvMessage);

      recvMessage.Acknowledge();

      // -------------------------------------------------------
      // Delete the durable subscription, otherwise it would continue
      // to exist after the demo exits.
      // -------------------------------------------------------
      //

     consumer.Close();  // closes consumer, but doesn't delete subscription

      session.Unsubscribe("MySubscriberID"); // deletes subscription

     // -----------------------------------------------------------------
      // Close the connection.   Note that closing a connection also closes
      // its child sessions, consumers, and producers.
     // -----------------------------------------------------------------
```

```
    connection.Close();

  } finally {

   // --------------------------------------------------------------------
    // Close the context.  The CloseAll method closes the network
    // connection and all related open connections, sessions, producers,
    // and consumers.
   // --------------------------------------------------------------------

    context.CloseAll();
  }
}

private void PrintMessage(String header, IMessage msg) {
  string msgtext;

  if (msg is ITextMessage)
    msgtext = " Text=" + ((ITextMessage)msg).Text + NL;
  else
    msgtext = " The message is not an ITextMessage";

  System.Console.WriteLine(
    header + NL +
    " JMSMessageID=" + msg.JMSMessageID + NL +
    " JMSRedelivered=" + msg.JMSRedelivered + NL +
    msgtext);
}

private bool ParseCommandLine(string[] args)
{
  int i = 0;
  try {
    for(i = 0; i < args.Length; i++) {
      if (args[i].Equals("-host")) {
        host = args[++i];
        continue;
      }
```

```
      if (args[i].Equals("-port")) {
        port = Convert.ToInt32(args[++i]);
        continue;
      }
      if (args[i].Equals("-cf")) {
        cfName = args[++i];
        continue;
      }
      if (args[i].Equals("-queue")) {
        queueName = args[++i];
        continue;
      }
      if (args[i].Equals("-topic")) {
        topicName = args[++i];
        continue;
      }
      if (args[i].Equals("-help") || args[i].Equals("-?")) {
        Console.WriteLine(USAGE);
        return false;
      }
      Console.WriteLine("Unrecognized parameter '" + args[i] + "'.");
      Console.WriteLine(USAGE);
      return false;
    }
  } catch (System.IndexOutOfRangeException) {
    Console.WriteLine(
      "Missing argument for " + args[i - 1] + "."
    );
    Console.WriteLine(USAGE);
    return false;
  } catch (FormatException) {
    Console.WriteLine(
      "Invalid argument '" + args[i] + "' for " + args[i - 1] + "."
    );
    Console.WriteLine(USAGE);
    return false;
  }
  Console.WriteLine(
```

```
      "WebLogic JMS .NET Client Demo " + NL +
      NL +
      "Settings: " + NL +
      "  host     = " + host + NL +
      "  port     = " + port + NL +
      "  cf       = " + cfName + NL +
      "  queue    = " + queueName + NL +
      "  topic    = " + topicName + NL
    );
    return true;
  }
}
```