Oracle® Fusion Middleware

Developing JMS .NET Client Applications for Oracle WebLogic Server

15c (15.1.1.0.0)
G31902-01
October 2025

ORACLE

Oracle Fusion Middleware Developing JMS .NET Client Applications for Oracle WebLogic Server, 15c (15.1.1.0.0)

G31902-01

Contents

4 Programming Considerations

Preface

This document is written for application developers who want to develop JMS .NET client applications that access WebLogic JMS resources.

Audience

This document is a resource for software developers who want to develop and configure applications that include WebLogic Server JMS. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server or considering the use of WebLogic Server JMS for a particular application

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

This document does not address production phase administration, monitoring, or performance tuning JMS topics. For links to WebLogic Server documentation and resources for these topics, see Related Documentation.

It is assumed that the reader is familiar with Jakarta EE and JMS concepts. This document emphasizes the value-added features provided by WebLogic Server JMS and key information about how to use WebLogic Server features and facilities to get a JMS application up and running.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://support.oracle.com/portal/ or visit Oracle Accessibility Learning and Support if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

This document contains JMS-specific design and development information.

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

- *Administering JMS Resources for Oracle WebLogic Server* for information about configuring and managing JMS resources.
- *Developing JMS Applications for Oracle WebLogic Server* is a guide to JMS API programming with WebLogic Server.
- *Administering the Store-and-Forward Service for Oracle WebLogic Server* for information about the benefits and usage of the Store-and-Forward service with WebLogic JMS.
- *Administering the WebLogic Persistent Store* for information about the benefits and usage of the system-wide WebLogic Persistent Store.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.

Samples and Tutorials

Oracle provides a variety of code examples and tutorials that show WebLogic Server configuration and API use, and provide practical instructions on how to perform key development tasks. For more information, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Oracle recommends that you run some or all of the JMS examples before developing your own JMS applications.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Overview of the WebLogic JMS .NET Client

This chapter provides an overview of the WebLogic JMS .NET client, illustrates how a JMS .NET client application accesses WebLogic JMS resources, and provides 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.

This chapter includes the following sections:

What is the WebLogic JMS .NET Client?

The WebLogic JMS .NET client is a fully-managed .NET runtime library and application programming interface (API). It enables programmers to create client applications using .NET C# or any other supported .NET programming languages to access WebLogic JMS applications and resources.

WebLogic JMS is an enterprise-level messaging system that fully supports the JMS 3.0 Specification (see https://jakarta.ee/specifications/messaging/3.0/) and also provides numerous WebLogic JMS Extensions to the standard JMS APIs. For a summary of the WebLogic Server value-added JMS features, see WebLogic Server Value-Added JMS Features in *Administering JMS Resources for Oracle WebLogic Server*.

For complete details about all the classes and interfaces in the JMS .NET API, see the *Microsoft .NET Messaging API for Oracle WebLogic Server* documentation.

The WebLogic JMS .NET client, which is bundled with Oracle WebLogic Server, is supported on the Microsoft .NET Framework and the .Net Core environment using the V1 version and V2 version of the libraries, respectively. Installation details are provided in 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 3.0 Specification (see https://jakarta.ee/specifications/messaging/3.0/). For a list of the JMS standard features that are not supported, see Limitations of Using the WebLogic JMS .NET Client .

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

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*. Messages may be lost when there is a JMS provider failure and will not be redelivered.

For more information, see Understanding the Messaging Models in *Developing JMS Applications for Oracle WebLogic Server*.

Message Types

The WebLogic JMS .NET client supports the following message types, as defined in the JMS Specification (see https://jakarta.ee/specifications/messaging/3.0/):

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

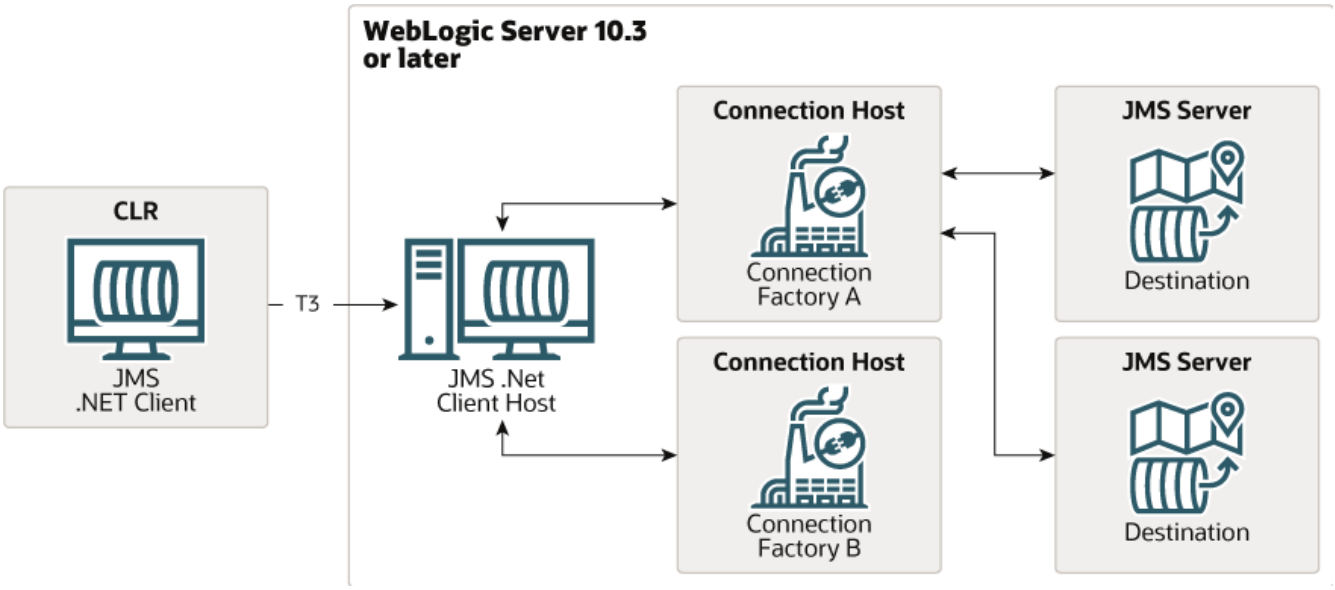
The XMLMessage type extension provided by WebLogic JMS is not supported in this release. Such messages are automatically converted to a TextMessage type when received by a .NET client.

For more information about using the supported message types, see Exchanging Messages Between Different Language Environments.

How the WebLogic JMS .NET Client Works

The following graphic illustrates how a JMS .NET client application running in a .NET environment can access JMS resources deployed on Oracle WebLogic Server.

Figure 1-1 JMS .NET Client Architecture



> **Note**
>
> All of the WebLogic components shown in Figure 1-1 are hosted on a single instance of WebLogic Server 10g Release 3 or later. In a multi-server or cluster configuration, each of the WebLogic Server components can run on a separate instance of WebLogic Server. However, the JMS .NET client host must run on WebLogic Server 10g Release 3 or later, and the connection host and the JMS server must run in the same WebLogic Server 9.x or later cluster.

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

- A JMS .NET client written in C# or any supported .NET programming language, running in a .NET environment, that either produces messages to destinations or consumes messages from destinations.
- A JMS .NET client host running on WebLogic Server 10g Release 3 or later that provides the interface between the JMS .NET client and WebLogic JMS.
- A standard T3 protocol listen port configured on the .NET client host.
- One or more connection hosts (i.e., connection factories).
- 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.

A brief summary of the process used to exchange messages between the JMS .NET client and a JMS server, as illustrated in Figure 1-1, 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 10g Release 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 Developing a Basic JMS Application Using the WebLogic JMS .NET API.

Configuring WebLogic Server

The following sections describe the configuration that must occur before a JMS .NET client application can access JMS resources.

Configuring the Listen Port

The JMS .NET client requires that a listen port configured for T3 protocol is enabled on the WebLogic Server instance hosting the JMS .NET client host. When you install WebLogic Server, a default port is configured for use with T3 protocol. Because the default port configuration can be changed or disabled, the system administrator needs to ensure that the T3 protocol is enabled on the server's default port, or add a network channel that supports the T3 protocol. For configuration information, see the following topics:

- Configure Network Connections in the *Oracle WebLogic Remote Console Online Help*
- Understanding Network Channels in *Administering Server Environments for Oracle WebLogic Server*

Configuring JMS Resources for the JMS .NET Client

Before a JMS .NET client application can access JMS resources deployed on WebLogic Server, the WebLogic Server system administrator must configure the required JMS resources, including the connection factories, JMS servers, and destinations. For instructions for configuring JMS resources, see:

- *Administering JMS Resources for Oracle WebLogic Server*
- Messaging in the *Oracle WebLogic Remote Console Online Help*

Interoperating with Pre-12.1.3 JMS .NET Clients

To enable JMS .NET clients developed prior to WebLogic Server 12.1.3 to interoperate with WebLogic Server 12.1.3 and later, set the following system property on your WebLogic Server 12.1.3 and later instances:

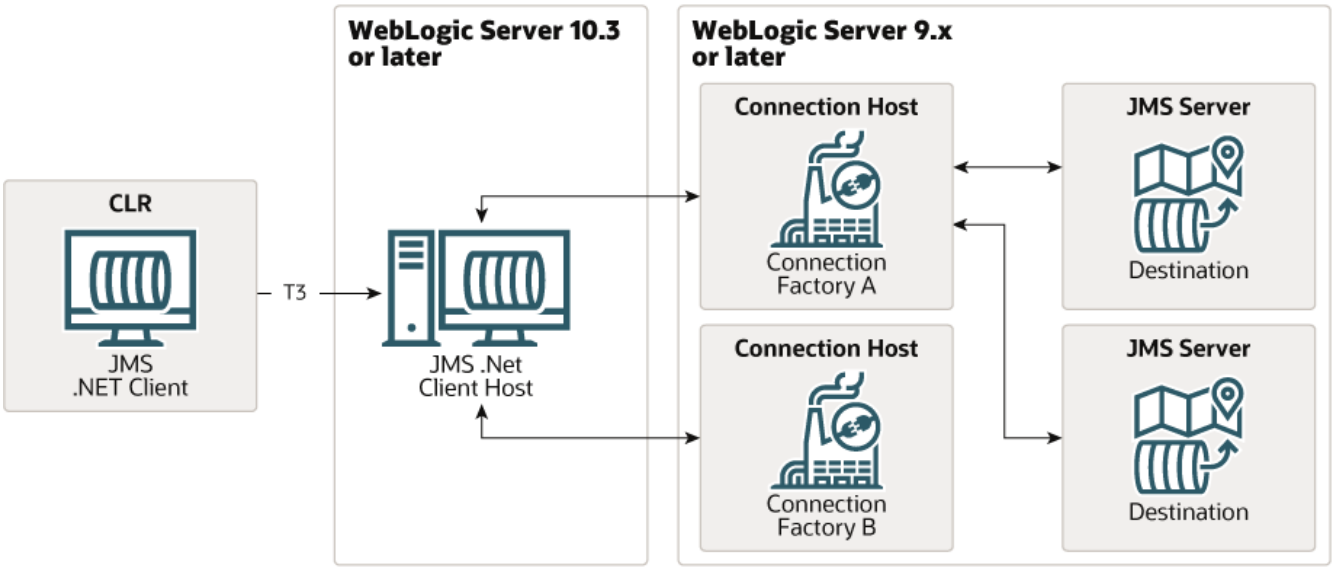
```
-Dweblogic.protocol.t3.login.replyWithRel10Content=true
```

The default value is `false` for interoperability with existing JMS .NET clients developed prior to WebLogic Server 12.1.3.

Interoperating with Previous WebLogic Server Releases

The JMS .NET client can communicate directly only with WebLogic Server 10g Release 3 and later. As shown in Figure 1-2, the JMS .NET client host must run on WebLogic Server 10g Release 3 or later, however, the connection host and the JMS server can run on WebLogic Server 9.x or later. Both the connection host and the JMS server must be in the same cluster.

Figure 1-2 JMS .NET Client Interoperability



To access destinations on WebLogic Server 9.x or later that are not in the same cluster as the .NET client host running on 10g Release 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 *Administering JMS Resources for Oracle WebLogic Server.*

> **Note**
>
> Although you can also use Foreign Servers to connect to third-party JMS providers using JMS Java clients, 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 *Microsoft .NET Messaging API for Oracle WebLogic Server*documentation.

Table 1-1 WebLogic JMS .NET Classes and Interfaces

Interface/Class	Description
`Constants`	The `Constants` family of classes 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.
`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. Since `IQueue` and `ITopic` both inherit from `IDestination`, for those methods that use an `IDestination` as a parameter, an `IQueue` object can be used as the argument. For example, an `IQueue` can be used to create an `IMessageConsumer` and an `IMessageProducer` by calling: `ISession.CreateConsumer(IQueue queue) ISession.CreateProducer(IQueue queue)`
`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.
`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.

Table 1-1 (Cont.) WebLogic JMS .NET Classes and Interfaces

Interface/Class	Description
`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. C# objects cannot be read by Java programs, and vice versa. For more information, see Exchanging Messages Between Different Language Environments.
`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 `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.

2

Installing and Copying the WebLogic JMS .NET Client Libraries

This chapter describes 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.
This chapter includes the following sections:

Installing the WebLogic JMS .NET Client

The WebLogic JMS .NET Client is bundled with WebLogic Server. When you perform a Complete installation of WebLogic Server on a supported platform, including non-Windows platforms, the WebLogic JMS .NET Client is installed by default. If you choose the Custom installation option, ensure that the WebLogic Server Clients component of WebLogic Server is selected. If you deselect this component, the WebLogic JMS .NET Client is not installed.

For a list of supported platforms for WebLogic Server, see Oracle Fusion Middleware Supported System Configurations.

For details about installing WebLogic Server, see *Installing and Configuring Oracle WebLogic Server and Coherence*.

Location of Installed Components

The WebLogic JMS .NET client is included in a Oracle WebLogic Server installation in the following two directories, which contain the Version 1 and Version 2 of the libraries, respectively:

```
ORACLE_HOME/wlserver/modules/com.bea.weblogic.jms.dotnetclient
```

```
ORACLE_HOME/wlserver/modules/com.bea.weblogic.jms.dotnetclient_v2
```

Here, *ORACLE_HOME* is the top-level installation directory that you selected during the installation process. For information about the difference between the Version 1 and Version 2 directory, see Choosing an Installation Version.

A DLL and a PDB file are included in each of the two directories:

- `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.

In addition, the Version 1 directory also contains a `jms.dotnet.api.zip` file. This file contains HTML and Windows help-style documentation for the WebLogic JMS .NET API that is applicable to both Version 1 and Version 2. See *Microsoft .NET Messaging API for Oracle WebLogic Server*.

Choosing an Installation Version

The following table illustrates the applications, .Net environments, and OS platforms that are supported for each .Net JMS Client version.

Library Version	Libraries Location	Applications	.Net Versions	OS Platforms
Version 1	*ORACLE_HOME*/wlserver/modules/com.bea.weblogic.jms.dotnetclient	.Net Framework	.Net Framework 2.0 to .Net Framework 3.5 .Net framework 4.8	Windows
Version 1*	*ORACLE_HOME*/wlserver/modules/com.bea.weblogic.jms.dotnetclient	.Net Core	.Net Core 3.1 and .Net 5.0	Windows and Linux
Version 2**	*ORACLE_HOME*/wlserver/modules/com.bea.weblogic.jms.dotnetclient_v2	.Net Core	.Net Core 3.1 and .Net 5.0	Windows and Linux

Note

- A WebLogic JMS .Net Client DLL's version number is embedded in the DLL as a file property and has a value format of 'N.N.N.N'. '1.N.N.N' corresponds with Version 1 and '2.N.N.N' corresponds with Version 2 in the above table. One way to obtain a DLL's file properties is to right-click on the file from within the Windows File Explorer.
- (*) Although you can use the Version 1 libraries in a .Net Core application, Oracle recommends that .Net Core applications use the Version 2 libraries. But, if for any reason you need to use Version 1 for a .Net Core application, you should note the following:
  - The client side logging and debugging settings in `app.config` are not honored because the `<system.diagnostics>` part of the application config file is not supported in .Net Core. See Configuring Logging and Debugging.
  - You may encounter runtime errors about `System.Configuration.ConfigurationManager`. For example:

    ```
    WebLogic.Messaging.MessageException: Problem creating context

    System.IO.FileNotFoundException: Could not load file or assembly
    'System.Configuration.ConfigurationManager, Version=4.0.3.0,
    Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51'. The system cannot
    find the file specified.
    ```

    Such errors can be resolved by installing the `System.Configuration.ConfigurationManager` package in the project using Visual Studio Manage NuGet packages or by directly adding the package to the `.csproj` file, as shown in the following example:

    ```
    <ItemGroup>

                <PackageReference
    Include="System.Configuration.ConfigurationManager" Version="6.0.0-
    preview.5.21301.5" />

           </ItemGroup>
    ```
- (**) When using the Version 2 libraries, your applications may need to install additional packages such as:
  - `System.Configuration.ConfigurationManager`
  - `Microsoft.Extensions.Configuration`
  - `Microsoft.Extensions.Configuration.Binder`
  - `Microsoft.Extensions.Configuration.Json`

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 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 Windows platforms running the Microsoft .NET Framework:

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** , then select **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.

3

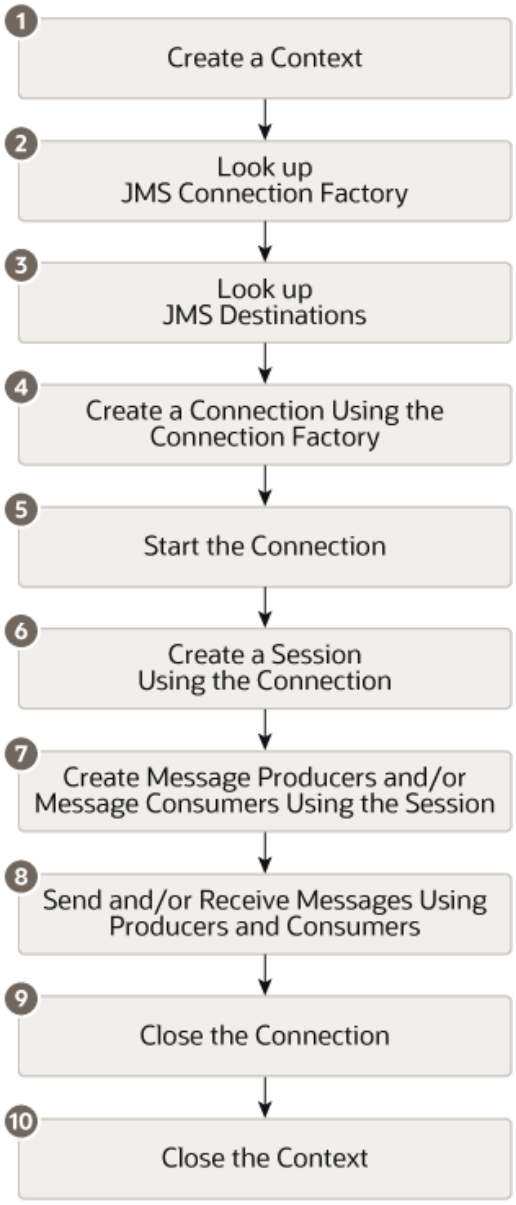
Developing a Basic JMS Application Using the WebLogic JMS .NET API

This chapter describes the steps required to develop a basic JMS application in C# using the 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.
This chapter includes the following sections:

Creating a JMS .NET Client Application

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

Figure 3-1 Basic Steps in a JMS .NET Client Application



> **Note**
>
> Creating and closing resources has relatively higher overhead in comparison to sending and receiving messages. Oracle recommends that contexts be shared between threads, and that other resources be cached for reuse. For more information, see Best Practices.

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 JMS .NET Client Sample Application.

For more information about the .NET API classes and methods used in this example, see Understanding the WebLogic JMS .NET API, or the WebLogic Messaging API Reference for .NET Clients documentation.

Prerequisites

Before proceeding, ensure that the system administrator responsible for configuring WebLogic Server has configured the following:

- Listen port configured for T3 protocol on the server hosting the JMS .NET client host. For more information, see Configuring the Listen Port.
- The required JMS resources, including the connection factories, JMS servers, and destinations. For more information, see Configuring JMS Resources for the JMS .NET Client.

Basic Steps

The following steps assume you have defined the required variables, including the WebLogic Server host, the connection factory, and the queue and topic names at the beginning of your program.

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


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 create a context to establish a network connection to the WebLogic Server host and optionally log in:

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

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

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

> **Note**
>
> The Provider_URL may contain multiple addresses, separated by commas. For details about specifying multiple addresses, see Specifying the URL Format.
>
> When multiple addresses are specified, the context tries each address in turn until one succeeds or they all fail, starting at a random location within the list of addresses, and rotating through all addresses. Starting at a random location facilitates load balancing of multiple clients, as different client contexts will randomly load balance their network connection to different .NET client host servers.

> **Note**
>
> 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 JMS connection factory

To look up the JMS connection factory:

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

Step 3. Look up JMS destinations

To 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.

To create a connection using the connection factory:

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

Step 5. Start the connection

To start the connection to allow consumers to get messages:

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

Step 6. Create a session using the connection

> **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 JMS .NET Client Sample Application.

To create a session using the `AUTO_ACKNOWLEDGE` acknowledge mode:

```
ISession session = connection.CreateSession(
Constants.SessionMode.AUTO_ACKNOWLEDGE);
```

Step 7. Create a message producer and send a message

To 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);
```

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. Create a session using the connection.

To create a message consumer and receive a message:

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

IMessage recvMessage = consumer.Receive(500);
```

Step 9. Close the connection

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

To close the connection:

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

Step 10. Close the context

To 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

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 *Developing JMS Applications for Oracle WebLogic Server.*
- Acknowledge modes. For more information, see Non-Transacted Session in *Developing JMS Applications for Oracle WebLogic Server*.
- Exception listeners. For more information, see Best Practices.
- Durable Subscriptions. For more information, see Setting Up Durable Subscriptions in *Developing JMS Applications for Oracle WebLogic Server*.

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

4

Programming Considerations

This chapter provides programming considerations and best practices to use when creating a JMS .NET client application.
This chapter includes the following sections:

Using WebLogic JMS Extensions

Table 4-1 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 often defines default configuration settings.
- Programmatically in the application using the API—Certain programming constructs may override the connection factory configuration.
- On the server—Certain settings may override both the connection factory and programmatic constructs.

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 4-1 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 *Developing JMS Applications for Oracle WebLogic Server* • Configuring Distributed Destination Resources in *Administering JMS Resources for Oracle WebLogic Server*	Yes	Yes	No	No	
Flow Control Producers For more information, see: Controlling the Flow of Messages on JMS Servers and Destinations in *Tuning Performance of Oracle WebLogic Server*	Yes	Yes	No	No	

Table 4-1 (Cont.) WebLogic JMS Extensions Supported in the JMS .NET Client

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 *Tuning Performance of Oracle WebLogic Server*	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 *Administering JMS Resources for Oracle WebLogic Server*	No	Yes	No	No	See Interoperating with Previous WebLogic Server Releases.
Imported store-and-forward (SAF) destinations For more information, see Imported SAF Destinations in *Administering the Store-and-Forward Service for Oracle WebLogic Server*	No	Yes	No	No	
Redelivery limit For more information, see Setting a Redelivery Limit for Messages in *Developing JMS Applications for Oracle WebLogic Server*	No	Yes	Yes	No	
Redelivery delay For more information, see Setting a Redelivery Delay for Messages in *Developing JMS Applications for Oracle WebLogic Server*	Yes	No	Yes	No	
Error destinations For more information, see Configuring an Error Destination for Undelivered Messages in *Developing JMS Applications for Oracle WebLogic Server*	No	Yes	No	No	
WLDestination.getCreateDestinationArgument	No	No	Yes	Yes	
No Acknowledge Mode For more information, see Using NO_ACKNOWLEDGE in *Developing JMS Applications for Oracle WebLogic Server*	No	No	Yes	Yes	

Table 4-1 (Cont.) WebLogic JMS Extensions Supported in the JMS .NET Client

Feature	Configurable on Connection Factory	Configurable on the Server	Java API	JMS .NET API	Comments
Unit-of-Order For more information, see: • Using Message Unit-of-Order in *Developing JMS Applications for Oracle WebLogic Server* • Tuning Applications Using Unit-of-Order in *Tuning Performance of Oracle WebLogic Server*	Yes	Yes	Yes	Yes	See Unit-of-Order.
Scheduled message delivery For more information, see Setting Message Delivery Times in *Developing JMS Applications for Oracle WebLogic Server*	Yes	Yes	Yes	Yes	See Message Delivery Time.
Asynchronous consumer messages maximum pipeline • For more information, see: Asynchronous Message Pipeline in *Developing JMS Applications for Oracle WebLogic Server* • Tuning MessageMaximum in *Tuning Performance of Oracle WebLogic Server*	Yes	No	Yes	No	
Message Compression For more information, see Message Compression in *Developing JMS Applications for Oracle WebLogic Server*	Yes	No	Yes	No	See Message Compression.
Quotas For more information, see Defining Quota in *Tuning Performance of Oracle WebLogic Server*	No	Yes	No	No	
One-way message sends For more information, see Using One-Way Message Sends in *Tuning Performance of Oracle WebLogic Server*	Yes	No	No	No	See One-Way Message Sends.
Acknowledge policy	Yes	No	No	No	
Automatically include user-id as message property `JMSXUserID`	Yes	Yes	No	No	See Include user-id as JMSXUserId.
Get number of delivery attempts as message property `JMSXDeliveryCount`	No	No	No	No	See Message Delivery Attempts.

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. The behavior is as follows:

- If the client host and destination run on different instances of WebLogic Server, then a sent message is automatically compressed on the client host.
- If the client host and destination run on the same instance of WebLogic Server, then no sent message compression will occur.

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 *Developing JMS Applications for Oracle WebLogic Server*

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 *Developing JMS Applications for Oracle WebLogic Server*

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 property of type `long` 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.

Include user-id as JMSXUserId

The optional `JMSXUserId` system-generated message property on received messages specifies the credential of the original sender. To enable this property, configure the Attach Sender Credential attribute on destinations, distributed destinations, or templates, and configure the Attach JMSXUserId attribute on connection factories. To retrieve, call `msg.GetStringProperty(Constants.MessagePropertyNames.USER_ID_PROPERTY_NAME)`.

Message Delivery Attempts

The `JMSXDeliveryCount` system-generated message property on received messages specifies the number of message delivery attempts. The first attempt is 1. To retrieve the value, call `msg.GetIntProperty(Constants.MessagePropertyNames.DELIVERY_COUNT_PROPERTY_NAME`.

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 2.0 Standard Features

In this release, the JMS 2.0 standard feature 'shared topic subscriptions' is not supported. The workaround is to use the shared subscription feature in Oracle JMS by configuring a custom connection factory and setting the Subscription Sharing Policy attribute on the connection factory as `Sharable` and Client ID Policy as `Unrestricted`.

For more information, see Shared Subscriptions and Client ID Policy in *Developing JMS Applications for Oracle WebLogic Server*.

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 a Java client if the destination objects are obtained from the JMSReplyTo header of received messages.
- `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).
- Connection Consumer and Server session pools (`javax.jms.ConnectionConsumer`, `ServerSessionPool`, and `ServerSession`). These are optional capabilities that have been superseded by Java EE MDBs, and are not supported by the WebLogic Java JMS client.
- `MessageProducer.setDisableMessageTimestamp` method. Note that the WebLogic JMS 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 in *Developing Standalone Clients for Oracle WebLogic Server*
- Multicast Subscribers—See Using Multicasting with WebLogic JMS in *Developing JMS Applications for Oracle WebLogic Server*
- Automatic Reconnect—See Automatic JMS Client Failoverin *Developing JMS Applications for Oracle WebLogic Server*
- Unit-of-Work—If a .NET client attempts to set a UOW property on a message, a `Weblogic.Messaging.MessageException` is generated. In addition, a .NET consumer cannot receive UOW messages with deserializable content that are sent by a Jakarta client. In this case, the consumer gets a `MessageFormatException` if it calls the `ObjectMessage.getObject()` method on the `ObjectMessage`. Note that while Unit-of-Work is not supported, the more commonly used Unit-of-Order extension is fully supported. For more information about Unit-of-Order, see Unit-of-Order.

> **Note**
>
> The JMS .NET API does not provide extensions for programmatically configuring JMS resources (for example, topics and queues). In Java, programmatic configuration is accomplished using JMX MBeans or the `weblogic.jms.extensions.JMSModuleHelper` helper class. Alternative ways to configure JMS include WLST scripting and the WebLogic Remote Console.

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 Jakarta 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`.

Specifying the URL Format

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

```
t3://address [,address]...
```

where a particular address may specify multiple port ranges.

The syntax for specifying multiple addresses is as follows:

```
address = hostlist : portlist
```

where

```
hostlist = hostname [, hostname]...
portlist = portrange [+ portrange]...
portrange = port [- port]
```

Use *port -port* to indicate a port range, and + (plus sign) to separate multiple port ranges.

Table 4-2 provides sample URL formats.

Table 4-2 URL Format Examples

This format . . .	Can also be specified as . . .
`t3://hostA:7001 t3://hostA,hostB:7001,hostC:7002`	`t3://hostA:7001,hostB:7001,hostC:7002`
`t3://hostA:7001+7005+7007,hostB:7001`	`t3://hostA:7001,hostA:7005,hostA:7007,hostB:7001`
`t3://hostA:7001-7003+7005+7007,hostB:8001`	`t3://hostA:7001,hostA:7002,hostA:7003,hostA:7005,hostA:7007,hostB:8001`

Using DNS Alias Host Names

You can also specify DNS alias host names, which are expanded into multiple hosts. For example, if a DNS alias `mycluster` resolves to `host1,host2`, then the URL `t3://mycluster:7001` expands into the address list: `t3://host1:7001,host2:7001`. Contexts that are created with the URL will always retry with `host2` if `host1` is unreachable. DNS aliases are typically configured by network administrators.

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 user name 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 user name and password is optional. But if your application encounters an exception with the error message `The Microsoft .NET client is required to provide credential info`, you need to either provide a user name and password (see Example 4-1), or enable anonymous access using `-Dweblogic.security.remoteAnonymousRMIT3Enabled=true` on the server startup command. See Disable Remote Anonymous RMI T3 and IIOP Requests in *Securing a Production Environment for Oracle WebLogic Server*.
> * Although user names 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 user names and passwords are provided.

- 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 Secure the Network in *Securing a Production Environment for Oracle WebLogic Server*.
- Similar to the Java client, the JMS.NET client does not support message level encryption.
- Due to the use of non-encrypted communication, sniffing of application traffic (see http://www.owasp.org/index.php/Sniffing_application_traffic_attack) is possible. You need to either accept these risks, or take remediation such as using a firewall to protect against these attacks.
- The administration port, if configured, accepts only SSL traffic, and all connections via the port require administrator privileges. In addition, once an administration port is configured, all other ports will refuse connections that have administrator privileges. Because SSL is not supported for the JMS .NET client in this release, it cannot support users with administrative privileges if an administration port is configured.

Example 4-1 Create a Context With a User Name and a Password

```
IDictionary<string, Object> paramMap = new Dictionary<string, Object>();
paramMap[Constants.Context.PROVIDER_URL] = "t3://host:port";
```

```
paramMap[Constants.Context.SECURITY_PRINCIPAL] = "fred";
paramMap[Constants.Context.SECURITY_CREDENTIALS] = "fredpassword";
IContext context =  ContextFactory.CreateContext(paramMap);
```

Configuring Logging and Debugging

Basic logging and debugging is available for the server-side transport and .NET client host running on WebLogic Server.

Server Side

To enable debugging on the server side, use the following commands:

```
-Dweblogic.debug.DebugJMSDotNetT3Server=true
-Dweblogic.debug.Debug.JMSDotNetProxy=true
```

Client Side

Client-side logging and debugging are enabled and controlled by various configuration settings in two categories:

Message Output

You should specify whether log messages are output to the console or saved to a file as shown in Table 4-3.

Table 4-3 Message Output Settings

Key	Value	Setting
`weblogic.debug.JMSDotNet.config.LogFileName`	String	Indicates the full path and file name for the log file. For example `c:\test\MyLogFile.log`. **Note**: The default log file size limit is 500KB. Each time the log file reaches this size, the server renames the log file and creates a new `MyLogFile.log` to store new messages. By default, the rotated log files are numbered in the order of creation. For example `MyLogFile.log.0`, `MyLogFile.log.1`, `MyLogFile.log.2`, `...`, with `MyLogFile.log.0` containing the latest log messages.
`weblogic.debug.JMSDotNet.config.IsLogToConsole`	Boolean	• True - Displays log messages to the console. • False - Does not display log messages to the console.

Log Categories and Levels

Client-side logging is grouped into the following categories:

- `Socket`
- `T3`
- `Transport`
- `PhysicalMsg`
- `LogicalMsg`

- `All` (represents all individual categories listed above)

For each of the categories, you can specify any of these logging levels:

- `Off(0)`
- `Error(1)`
- `Warning(2)`
- `Info(3)`
- `Verbose(4)`

> **Note**
>
> The severity level on the `All` category overrides the setting on each individual category.

When using Version 1 libraries with a .Net Framework application

The client side logging and debugging are configured and controlled using an application configuration file. For generated build files, the application configuration file is named `yourapplicationname.exe.config`, while `yourapplicationname` is the name of the application that runs the messaging client.

Example 4-2 provides the XML content that needs to be added to your application configuration file to configure logging and debugging. The subsequent sections provide additional details about each of the different settings. If you have an existing `yourapplicationname.exe.config` file, add the XML content shown in the following listing to the file. Otherwise, you can create one and locate it in the same directory that contains the `yourapplicationname.exe` file.

> **Note**
>
> If you are using Visual Studio, the logging and debugging settings shown in Example 4-2 need to be added to the `App.config` file. For instructions to add an `App.config` file to your C# project inside a Visual Studio environment, see the Microsoft website.

Example 4-2 XML File Content for `yourapplicationname.exe.config` File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- To forward log output to a file, please uncomment the following
line, and replace the file name with the desired one -->
    <!--- <add key="weblogic.debug.JMSDotNet.config.LogFileName"
value="c:\test\MyLogFile.log" /> -->

    <!-- To prevent log messages from displaying to the console, use the
value 'false' -->
    <!-- <add key="weblogic.debug.JMSDotNet.config.IsLogToConsole"
value="false" /> -->
  </appSettings>
  <system.diagnostics>
```

```
    <switches>
      <!-- Please set the switch value as desired for logging to each
Category -->
      <!-- value for Off=0, Error=1, Warning=2, Info=3, Verbose=4    -->

      <!-- if "AllLogger" is enabled (no zero for the value), every
individual category is set to the same level as the AllLogger,
            no matter how individual category's value is set -->
      <add name="weblogic.debug.JMSDotNet.All" value="0" />
      <add name="weblogic.debug.JMSDotNet.Socket" value="0" />
      <add name="weblogic.debug.JMSDotNet.T3" value="0" />
      <add name="weblogic.debug.JMSDotNet.Transport" value="0" />
      <add name="weblogic.debug.JMSDotNet.PhysicalMsg" value="0" />
      <add name="weblogic.debug.JMSDotNet.LogicalMsg" value="0" />
    </switches>
  </system.diagnostics>
</configuration>
```

> **Note**
>
> In a .Net Core environment, the XML application config file settings are not honored anymore; instead, consider using the latest version of the libraries with the `appsettings.json` file. See Example 4-3.

When using Version 2 libraries with a .Net Core application

The client side logging and debugging is configured and controlled in the application's `appsettings.json` file in a .Net Core (2.1 and later) environment.

Example 4-3 JSON File Content for the `appsettings.json` File

```
{
  "weblogic.debug.JMSDotNet.ALL": {
    "Level": "0"
  },
  "weblogic.debug.JMSDotNet.Socket": {
    "Level": "0"
  },
  "weblogic.debug.JMSDotNet.T3": {
    "Level": "0"
  },
  "weblogic.debug.JMSDotNet.Transport": {
    "Level": "0"
  },
  "weblogic.debug.JMSDotNet.PhysicalMsg": {
    "Level": "0"
  },
  "weblogic.debug.JMSDotNet.LogicalMsg": {
    "Level": "0"
  },
  "weblogic.debug.JMSDotNet.config.LogFileName": "c:\\test\\MyLogFile.log",
  "weblogic.debug.JMSDotNet.config.IsLogToConsole": "false"
}
```

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 environment 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 asynchronous message event handlers in the .NET client application 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 Conversion Between Java and .NET

See the following sections:

Endian Conversions

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 and definition, for every Java primitive data type. The following table lists the different names used for signed and unsigned bytes in C# and Java.

Table 4-4 Byte Primitive Data Type in C# and Java

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

As shown in Table 4-4, 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.

Time Conversions

The WebLogic JMS .NET API represents dates and times using Java rather than .NET conventions. The JMSTimestamp and JMSExpiration attributes of the

`WebLogic.Messaging.IMessage` message interface are type `long` and contain a millisecond absolute time value as specified in the Java programming language. The Java millisecond absolute time value is the difference, measured in milliseconds, between a given time and midnight, January 1, 1970 UTC.

The following examples demonstrate how to convert between .NET times and Java millisecond absolute time values.

Example 4-4 Example C# Code for Converting the Current .NET Time to Java Millisecond Time

```
// Example:  C# code for converting the current .NET time to Java millisecond time
DateTime baseTime = new DateTime(1970, 1, 1, 0, 0, 0);
DateTime utcNow = DateTime.UtcNow;
long timeInMillis = (utcNow.Ticks - baseTime.Ticks)/10000;
Console.WriteLine(timeInMillis);
```

Example 4-5 Example C# Code for Converting Java Millisecond Time to .NET Time

```
// Example:  C# code for converting Java millisecond time to .NET time
DateTime baseTime = new DateTime(1970, 1, 1, 0, 0, 0);
long utcTimeTicks = (timeInMillis * 10000) + baseTime.Ticks;
DateTime utcTime = new DateTime(utcTimeTicks, DateTimeKind.Utc);
Console.WriteLine(utcTime);
Console.WriteLine(utcTime.ToLocalTime());
```

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. See Understanding Socket and Threading Behavior. 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 is implicitly supplied with the Java client.

A

JMS .NET Client Sample Application

This chapter provides a .NET client sample program written in C# which includes basic features of the WebLogic JMS .NET API.

For details about the API, see *Microsoft .NET Messaging API for Oracle WebLogic Server*.

Example 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>
/// This command line program connects to WebLogic JMS and performs
/// queue and topic messaging operations.  It is supported with
/// versions 10g Release 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>
/// <para>
/// Copyright 1996,2008, Oracle and/or its affiliates.
/// </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(
         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 consumer, MessageEventArgs args) {
  PrintMessage("Received Message Asynchronously:", args.Message);

  // ------------------------------------------------------------
  // If the consumer's session is CLIENT_ACKNOWLEDGE, remember to
  // call args.Message.Acknowledge() to prevent the message from
  // getting redelivered, or consumer.Session.Recover() to force redelivery.
  // Similarly, if the consumer's session is TRANSACTED, remember to
  // call consumer.Session.Commit() to prevent the message from
  // getting redeliverd, or consumer.Session.Rollback() to force redeivery.
}

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 and start it:
    // ------------------------------------------------------------

    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(
       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(
       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(
     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";

  string dcProp =
     Constants.MessagePropertyNames.DELIVERY_COUNT_PROPERTY_NAME;

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

Index