Progress FUSE

FUSE™ Message Broker

Getting Started

Version 5.3
Febuary 2009

PROGRESS SOFTWARE

Getting Started

Version 5.3

Publication date 23 Jul 2009
Copyright © 2001-2009 Progress Software Corporation and/or its subsidiaries or affiliates.

Legal Notices

Progress Software Corporation and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from Progress Software Corporation, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

Progress, IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix;, FUSE, and Making Software Work Together are trademarks or registered trademarks of Progress Software Corporation and/or its subsidiaries in the US and other countries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the US and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate Progress Software Corporation makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Progress Software Corporation shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of Progress Software Corporation. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. Progress Software Corporation assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

Table of Contents

1. Introducing FUSE Message Broker 11
What is FUSE Message Broker? 12
Supported Standards 13
Supported Wire Protocols and Clients 14
High Availability 15
Scalability 16
Persistence 17
Security 18
Performance 19
2. Key Concepts 21
JMS Broker Deployment Topologies 22
Configuring FUSE Message Broker 25
JMS Basics 27
3. Exploring JMS 31
Setting Up the Guided Tour of JMS 32
Running JMS Sample Applications 36
About the Exploring JMS Samples 37
Taking the Exploring JMS Guided Tour 39
Running Publish and Subscribe Messaging Samples 40
Running Point-to-Point Messaging Samples 47
Running Request and Reply Samples 53
Running the Queue Test Loop Sample 56
Changing Parameters and Modifying Source Code 57
Revising Parameters in the Build File 58
Analyzing and Modifying the Java Source Files 61
Index 63

List of Figures

2.1. A Network of Brokers 24
2.2. Transport Connectors 26
2.3. Network Connectors 26
2.4. Point-to-point Messaging 28
2.5. Publish-Subscribe Messaging 29

List of Tables

3.1. Command for Running the Installer in GUI Mode 32

List of Examples

2.1. Starting an Embedded Broker 23
2.2. Starting a Named Embedded Broker 23
2.3. Defining Transport and Network Connectors 25

Chapter 1. Introducing FUSE Message Broker

This chapter provides an overview of the supported standards and available features in FUSE Message Broker.

What is FUSE Message Broker? 12
Supported Standards 13
Supported Wire Protocols and Clients 14
High Availability 15
Scalability 16
Persistence 17
Security 18
Performance 19

What is FUSE Message Broker?

ActiveMQ

FUSE Message Broker is Progress Software's distribution of Apache ActiveMQ, the open source message-oriented middleware (MOM) system.

Pure Java

FUSE Message Broker is written in Java and fully implements the Java Message Service (JMS) 1.1 specification. It also supports J2EE integration features such as Java Database Connectivity (JDBC), J2EE Connector Architecture (JCA), and Enterprise JavaBeans (EJB).

Supported Standards

JMS 1.1

JMS 1.1 allows J2EE application components to create, send, receive, and read messages for reliable, loosely coupled communication across distributed systems.

FUSE Message Broker supports the following JMS features:

- Queue- and topic-based messaging
- Persistent and non-persistent messaging
- JMS transactions
- XA transactions

J2EE 1.4

FUSE Message Broker can be used with your organization's existing J2EE platform architecture. It supports any J2EE application server, such as Geronimo 1.x, JBoss 4.x, WebSphere 6.x or WebLogic 9.x.

The JCA Resource Adapter allows a J2EE application server to efficiently pool connections, control transactions, and manage security for FUSE Message Broker.

JNDI

Java Naming and Directory Interface (JNDI) enables applications to locate and connect with services, for seamless connectivity to heterogeneous enterprise naming and directory services. Developers rely on the JNDI standard to build directory-enabled applications.

You can set up JNDI in FUSE Message Broker simply by adding a `jndi.properties` file to your classpath.

AJAX and REST

FUSE Message Broker facilitates integration of existing Internet applications and wireless devices that depend on HTTP. It includes a Representational State Transfer (REST) API that allows you to integrate Asynchronous JavaScript and XML (AJAX) applications into your organization's messaging backbone.

Supported Wire Protocols and Clients

Encoding formats

While FUSE Message Broker is written in Java, it can also supports connections with a host of different clients thanks to its support for the OpenWire and STOMP encoding formats.

OpenWire

The default wire protocol used by native Java FUSE Message Broker clients is the OpenWire binary format. There are also OpenWire client libraries available for C, C++ and .NET.

STOMP

Streaming Text Oriented Messaging Protocol (STOMP) is used to support FUSE Message Broker clients written in languages such as Ruby, Perl, Python, and PHP.

High Availability

Clustering

FUSE Message Broker supports reliable high performance load balancing of messages on a queue across consumers. If a consumer dies, any unacknowledged messages are redelivered to other consumers on the queue. If one consumer is faster than the others it receives more messages.

Failover

A client can connect to one broker node in a cluster and automatically fail over to a new node in the cluster if there is a failure. On the broker side, FUSE Message Broker uses a store-and-forward method to distribute messages over a cluster.

Scalability

High capacity brokers	Each broker supports thousands of persistent messages per second with minimal latency, and can handle a vast number of connections and destinations.
Clustering	Messaging loads can be shared among brokers in a cluster.
JMS streams for large messages	When sending messages of 1GB or larger, JMS streams eliminate the bottleneck that would occur as the JMS client tries to keep such large messages in memory.
Message compression	GZIP compression allows highly verbose messages to be compressed.

Persistence

Persistence options

You can enable or disable persistence depending on your business requirements. When persistence is enabled, you can configure FUSE Message Broker to write messages directly to a database, or to the high performance journal for increased throughput.

See *Using Persistent Messages* for details.

Supported databases

You can use any JDBC-compliant database to store long-term persisted messages. Supported databases include:

- Apache Derby
- Oracle
- Sybase
- DB2
- Microsoft SQL Server
- Postgresql
- MySQL
- Axion
- HSQL

Security

Encryption	FUSE Message Broker supports Secure Sockets Layer (SSL) encryption for transport over HTTPS.
Authentication and authorization	FUSE Message Broker provides plug-in points to support custom authentication and authorization, and supports third-party authentication providers, firewalls, proxy servers, HTTP(s) tunneling and DMZ products.

Performance

Optimized for performance

Although message oriented middleware is primarily focused on reliability over performance, FUSE Message Broker is optimized for high performance through its use of staged event-driven architecture (SEDA), straight through processing (STP), reactive scalable flow control, and high-performance journaling.

Performance options

You can optimize FUSE Message Broker by adjusting the following messaging parameters:

- Message compression
- Message fragmentation
- Asynchronous message sends
- Disable time stamps
- Customizable message pre-fetching
- Disable message copying
- Optimized message dispatch

High performance journal

The FUSE Message Broker high performance journal, which is enabled by default, reduces latency by capturing messages, transaction commits/rollbacks, and message acknowledgements faster than any database can. These are then written to a JDBC database at regular intervals.

Chapter 2. Key Concepts

This chapter introduces some concepts that are key to understanding broker topologies and the Java Message Service (JMS).

JMS Broker Deployment Topologies 22
Configuring FUSE Message Broker 25
JMS Basics 27

JMS Broker Deployment Topologies

Broker functions

The message broker is responsible for managing JMS clients and their messages. The message broker is also responsible for providing quality of service features, such as reliability, persistence, security, and availability.

You can deploy FUSE Message Broker as a standalone broker that hosts clients running in other processes and other locations, or as an embedded broker that can run client functions and broker functions concurrently in one process, yet still enable connections by clients running in other processes and other locations. Standalone brokers and embedded brokers can be configured to work together to provide a more resilient network, or cluster, of brokers.

Standalone broker

To start a standalone instance of FUSE Message Broker:

1. In a command prompt or terminal window, change directory to the FUSE Message Broker installation directory.
2. Change directory to the `bin` directory.
3. Type the following:
   - Windows:

     ```
     activemq.bat
     ```

   - UNIX:

     ```
     ./activemq
     ```

Embedded broker

An embedded broker executes within the same JVM process as the clients that are using its services. So rather than communicating across the network, clients can communicate with the broker more efficiently using direct method invocation.

In addition, if the network fails, clients can continue to send messages to the broker, which will hold the messages until the network is restarted.

Starting an embedded broker

An embedded broker executes within the same JVM process as the clients that are using its services. There are a number of ways to embed a broker. The simplest is shown in Example 2.1 on page 23

Example 2.1. Starting an Embedded Broker

```
BrokerService broker = new BrokerService();
broker.addConnector("tcp://localhost:61616");
broker.start();
```

Clients running in the same VM can connect to the embedded broker using the VM transport connector. External clients connect using the TCP transport connector. If you have more than one broker running in the same VM, you need to set a broker name, as follows:

Example 2.2. Starting a Named Embedded Broker

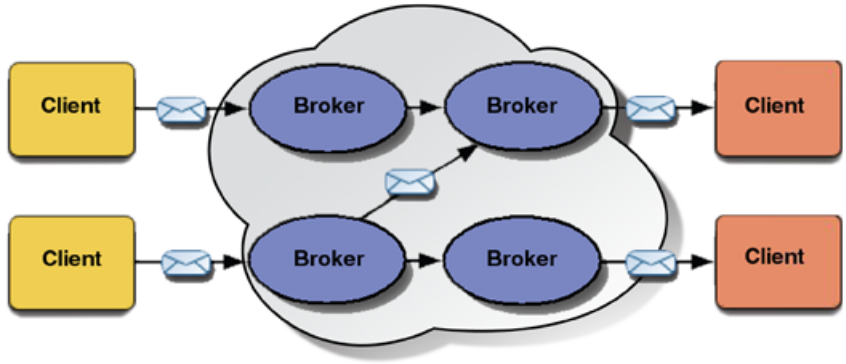
```
BrokerService broker = new BrokerService();
broker.setBrokerName("broker1");
broker.addConnector("tcp://localhost:61616");
broker.start();
```

Clients or other brokers connecting from within the same VM can then connect using the virtual machine protocol on the named broker `vm://broker1`.

Network of brokers

Brokers can be linked together to form a network or cluster of brokers. A network of brokers can use various network topologies, such as hub-and-spoke, daisy chain, or mesh.

Figure 2.1. A Network of Brokers



Configuring FUSE Message Broker

XML configuration

FUSE Message Broker is configured using XBean XML. XBean is an extension of the Spring Framework that has allowed the developers of Apache ActiveMQ to develop a syntax that is less verbose and yet more expressive than basic Spring configuration.

Configuration is stored in the `activemq.xml` file in the *`InstallDir`*`/conf` directory.

Connectors

The `activemq.xml` file allows you to configure transport and network connectors for FUSE Message Broker, as shown below:

Example 2.3. Defining Transport and Network Connectors

```
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://local
host:61616" discoveryUri="multicast://default"/>
  <transportConnector name="ssl" uri="ssl://localhost:61617"/>

  <transportConnector name="stomp" uri="stomp://local
host:61613"/>
  <transportConnector name="xmpp" uri="xmpp://local
host:61222"/>
</transportConnectors>

<networkConnectors>
  <!-- by default auto discover other brokers
  <networkConnector name="default-nc" uri="multicast://de
fault"/>-->
  <!--
      <networkConnector name="host1 and host2" uri="stat
ic://(tcp://host1:61616,tcp://host2:61616)"/>
      -->
</networkConnectors>
```

Transport connectors are used for communication between clients and brokers.

Figure 2.2. Transport Connectors

A broker uses a network connector to communicate with another broker.

Figure 2.3. Network Connectors

Note

For more details on transport and network connectors, see the *Connectivity Guide* guide.

JMS Basics

JMS components

Java Message Service (JMS) is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. A JMS Provider is the software that implements the Java Message Service (JMS) specification for a messaging product's brokers and clients.

If you are unfamiliar with JMS, you may want to read the Java Message Service API section of Sun Microsystems' J2EE 1.4 Tutorial[1]

A JMS messaging product is comprised of the following components:

- Brokers
- Messages
- Destinations
- Clients
- Connections
- Sessions

JMS broker

A JMS broker provides clients with connectivity, and message storage/delivery functions.

Messages

A messages is an object that contains the required heading fields, optional properties, and data payload being transferred between JMS clients.

Destinations

Destinations are maintained by the message broker. They can be either queues or topics.

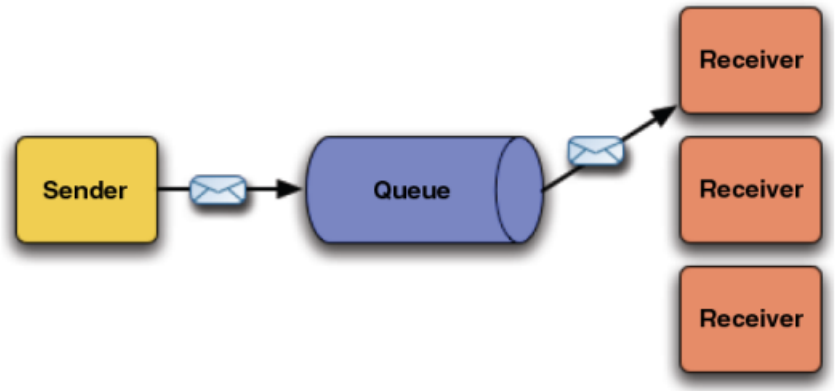
Queues

A queue is a destination that contains messages sent from a producer that await delivery to one consumer. Messages are delivered in the order sent. A message is removed from the queue once it has been acknowledged as received by the consumer.

[1] http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

In this one-to-one messaging model, producers are *senders* and consumers are *receivers*.

Figure 2.4. Point-to-point Messaging

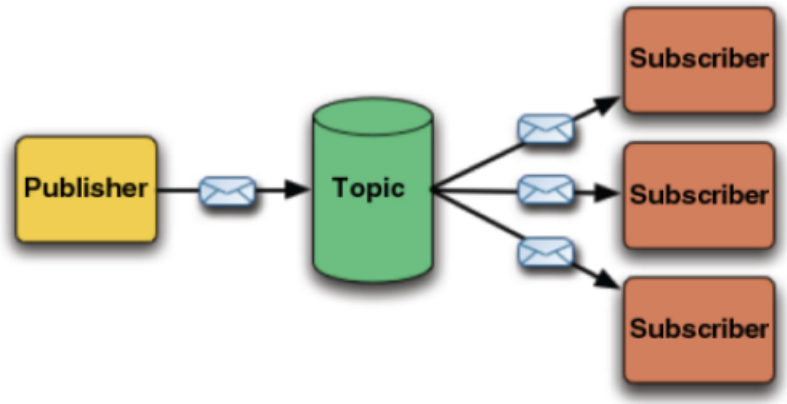


Topics

Topics are used to send messages to one or more consumers. Producers publish messages to a topic and one or more consumers subscribe to the topic

In this one-to-many messaging scenario, producers are also referred to as *publishers* and consumers as *subscribers*.

Figure 2.5. Publish-Subscribe Messaging



JMS Clients

A JMS client is an application that uses the services of the message broker. There are two client types in a JMS system:

Producer

Producers create messages and send or publish them to the broker for delivery to a specified destination.

Consumer

Consumers retrieve messages from a destination.

Connections

Connections are the technique used by clients to specify a protocol and credentials for a sustained client interaction with a broker.

Sessions

Sessions are defined by a client on a conection established with a broker. Each session defines whether the messages will form transactions, and -- if not -- the acknowledgement mode for messages.

Chapter 3. Exploring JMS

This chapter guides you through setting up an environment and running the samples packaged with FUSE™ Message Broker, and then modifying the runtime parameters and the Java source code.

Setting Up the Guided Tour of JMS 32
Running JMS Sample Applications 36
About the Exploring JMS Samples 37
Taking the Exploring JMS Guided Tour 39
Running Publish and Subscribe Messaging Samples 40
Running Point-to-Point Messaging Samples 47
Running Request and Reply Samples 53
Running the Queue Test Loop Sample 56
Changing Parameters and Modifying Source Code 57
Revising Parameters in the Build File 58
Analyzing and Modifying the Java Source Files 61

Setting Up the Guided Tour of JMS

Overview

You need to install FUSE Message Broker, Apache Ant, and a Java JDK to run the samples. You then need to identify those locations in the environment settings for the scripted samples.

You avoid some common problems when you put all the software on the same drive, and ensure that no paths contain spaces (such as `Program Files`).

Installing a Java JDK

The exploring JMS samples require a Java JDK version 1.5.0_11 or higher.

Installing Ant

Apache Ant is used to build and run the samples.

Access the Ant 1.6.5 (or higher) distribution for your platform at http://ant.apache.org.

Installing FUSE Message Broker

To install do the following:

1. Download the FUSE Message Broker 5.3 package for your platform from http://fusesource.com/downloads/.
2. Launch the installer.

Table 3.1. Command for Running the Installer in GUI Mode

Platform	Command
Windows	**`fuse-message-broker-5.3.0.0-windows.exe`**
UNIX	**`sh fuse-message-broker-5.3.0.0-unix.bin`**

Warning

On UNIX/Linux/Macintosh, you might need to set or explicitly specify the Java installation `/bin` to run the installer. On Linux, you might also need to run the `download.bin` file using the syntax:

```
sh  download.bin LAX_VM jvm_install/bin/java
```

3. Follow the wizard prompts for a default installation.

For more information on installing FUSE Message Broker see *Installation Guide*.

Adjusting the broker configuration

The network connector feature in the default broker configuration attempts to automatically discover and connect to other brokers. This default behavior can be distracting to observing the behaviors in these samples. To shut off network connections, navigate to your FUSE Message Broker's installation directory, edit the file `/conf/activemq.xml`, and delete the `networkConnectors` section. Save the edited file to enable the change when you restart FUSE Message Broker.

Accessing the samples

The sample applications used in this book are located in the `exploring-jms` folder of the FUSE Message Broker installation.

Setting up the environment and sample windows

The following procedures describe the steps on various platforms to set the required `HOME`, `PATH`, and `CLASSPATH` values.

On Windows

To set the environment and set up sample windows on Windows:

1. Open a console window, and then enter:

```
set ANT_HOME=ant_install
set JAVA_HOME=jvm_install
set PATH=%JAVA_HOME%\bin;%ANT_HOME%\bin;%PATH%
```

2. Spawn the windows that will run the sample applications:

```
start "Window 1" cmd
start "Window 2" cmd
start "Window 3" cmd
```

3. Launch the broker:

```
set FUSE_MB_ROOT=InstallDir
%FUSE_MB_ROOT%\bin\activemq
```

You are ready to take a tour of the samples on your Windows system!

On Linux

To set the environment and set up sample windows on Linux:

1. Open a terminal window, and then enter:

```
ANT_HOME=ant_install
export ANT_HOME
JAVA_HOME=jvm_install
export JAVA_HOME
PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$PATH
export PATH
FUSE_MB_ROOT=InstallDir
export FUSE_MB_ROOT
```

2. Spawn the windows that will run the sample applications:

```
bg
xterm -title "Window 1" &
xterm -title "Window 2" &
xterm -title "Window 3" &
```

3. Launch the broker:

```
cd $FUSE_MB_ROOT/bin
./activemq
```

You are ready to take a tour of the samples on your Linux system!

On Macintosh

Tip

If you installed Apple's X11 for Mac OS X, you can use the **xterm** commands described for Linux to spawn windows with the environment preset, and then launch the broker.

To set the environment and set up sample windows on Macintosh:

1. Open a terminal window, and then enter:

```
ANT_HOME=ant_install
JAVA_HOME=jvm_install
PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$PATH
FUSE_MB_ROOT=InstallDir
```

2. Launch the broker:

```
cd $FUSE_MB_ROOT/bin
./activemq
```

3. Open three terminal windows to run the sample applications. Set the same environment variables as you set for the broker in each of these windows.

You are ready to take a tour of the samples on your Macintosh system!

On UNIX

To set the environment and set up sample windows on UNIX:

1. Open a terminal window, and then enter:

```
ANT_HOME=ant_install
JAVA_HOME=jvm_install
PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$PATH
FUSE_MB_ROOT=InstallDir
```

2. Spawn the windows that will run the sample applications:

```
bg
xterm -title "Window 1" &
xterm -title "Window 2" &
xterm -title "Window 3" &
```

3. Launch the broker:

```
cd $FUSE_MB_ROOT/bin
./activemq
```

You are ready to take a tour of the samples on your UNIX system!

Running JMS Sample Applications

About the Exploring JMS Samples 37
Taking the Exploring JMS Guided Tour 39
Running Publish and Subscribe Messaging Samples 40
Running Point-to-Point Messaging Samples 47
Running Request and Reply Samples 53
Running the Queue Test Loop Sample 56

About the Exploring JMS Samples

Overview

In these samples, the standard input and standard output displayed in the console represents data flows to and from applications and Internet-enabled devices such as:

- **Application software** for accounting, auditing, reservations, online ordering, credit verification, medical records, and supply chains
- **Real-time devices** with embedded controls such as monitor cameras, cell phones, medical delivery systems, and climate control systems, and machinery
- **Distributed knowledge bases** such as collaborative designs, service histories, medical histories, and workflow monitors

Note

The samples assume that you are using the default FUSE Message Broker setup, which does not enable security; as such, user names in the samples are arbitrary and not authenticated.

What is demonstrated

The samples demonstrate the basic JMS features, as follows:

- **Publish and Subscribe Messaging Samples**— Basic messaging behaviors of the Pub/Sub messaging model are demonstrated:`Chat, DurableChat, HierarchicalChat, MessageMonitor, MessageMonitor, SelectorChat, TransactedChat`
- **Point-to-point Messaging Samples** — Similar basic messaging behaviors of the PTP messaging model are demonstrated:`Talk, QueueMonitor, SelectorTalk, TransactedTalk`
- **Request and Reply** — These transacted examples show the mechanisms for the producer requesting a reply and the consumer fulfilling that request:
  - **Originator's Request** — `Requestor` (PTP, Pub/Sub)
  - **Receiver's Response** — `Replier` (PTP, Pub/Sub)

- **Test Loop** — This sample shows how quickly messages can be sent and received in a test loop: `QueueRoundTrip` (PTP)

Taking the Exploring JMS Guided Tour

Running Publish and Subscribe Messaging Samples 40
Running Point-to-Point Messaging Samples 47
Running Request and Reply Samples 53
Running the Queue Test Loop Sample 56

Running Publish and Subscribe Messaging Samples

The tour starts with features of Publish and Subscribe messaging.

This series of samples explores basic subscriptions, durable subscriptions, wildcards in topic hierarchies, filtered subscriptions, and batching in transacted sessions.

Chat Application

What the sample does

In the `Chat` application, whenever anyone sends a text message to a given topic, all active applications running `Chat` receive that message as subscribers to that topic. This is the most basic form of publish and subscribe activity.

Running the sample

To run the chat sample do the following:

1. In window 1, enter: **`ant chat1`**, then type **`Hello`**, and press **Enter**.

   Window 1 displays:

   ```
   Chatter_1: Hello
   ```

2. In window 2, enter: **`ant chat2`**, then type **`Pronto`**, and press **Enter**.

   Both subscribers get the message so both windows display:

   ```
   Chatter_2: Pronto
   ```

3. In window 3, enter: **`ant chat3`**, then type **`Bonjour`**, and press **Enter**.

   All three subscribers get the message, so each window displays:

   ```
   Chatter_3: Bonjour
   ```

4. In window 3, stop `chat3` by pressing **Ctrl**+**C**.

5. Send some messages in the `chat1` and `chat2` windows.

6. In window 3, run: **`ant chat3`** again.

7. Send some messages in the `chat1` and `chat2` windows.

   All three subscribers get the message. But `Chatter_3` gets only the messages since it reconnected, and gets none of the messages that were sent while it was disconnected.

   If subscribers miss some of the messages, they pick up just the latest messages whenever they reconnect to the broker. Nothing is retained and nothing is guaranteed to be delivered, so throughput is fast.

Stopping the sample

To stop the Chat sessions press **Ctrl**+**C** in each of the windows.

DurableChat Application

What the sample does

In Pub/Sub messaging, when messages are produced, they are sent to all active consumers who subscribe to a topic. Some subscribers register an enduring interest in receiving messages that were sent while they were inactive. These **durable subscriptions** are permanent records in the broker's persistent storage mechanism.

Running the sample

To run the DurableChatting sample do the following:

1. In window 1, enter **`ant durable1`**.
2. In window 2, enter **`ant durable2`**.
3. Type text in each window and press **Enter**. Each window displays all the messages. Connected durable subscribers are the same as connected nondurable subscribers.
4. In window 2, press **Ctrl**+**C**.
5. In window 1, type text, and press **Enter**.
6. In window 2, enter **`ant durable2`** again.

   When the window opens, it first displays all the messages that were stored for its subscription. (If you waited a while, messages sent more than 30 minutes ago were dropped.)
7. In window 3, enter **`ant durable3`**.

In the console window that opens, no messages are displayed. While it is a durable subscriber, it had not yet established its durable subscription on this broker topic.

Important

If this sample has run against this broker before this run, the durable interest has been established so you **do** get the stored messages. You could unsubscribe the user from the durable subscription, but in the scope of this JMS exploration, you might find it as easy to close all the open windows, delete the /data directory, and then restart the exploration.

Important

While some applications tolerate multiple instances of the same user producing or consuming on the same destination, running a second instance of one of these `DurableChatter` scripts fails (as it should) with the error (in the broker's console window):

```
InvalidClientIDException:
Broker:localhost, Client:DurableChatter_n already connected
```

Stopping the sample

To stop the DurableChat sessions press **Ctrl**+**C** in each of the windows.

HierarchicalChat Application

What the sample does

FUSE Message Broker supports a hierarchical topic structure that allows wildcard subscriptions.

Each application instance specifies a publish topic and a subscribe topic, as follows:

- *Chat*:
  - Publish to `jms.samples.chat`
  - Subscribe to `jms.samples.chat`
- *DurableChat*:

  - Publish to `jms.samples.durablechat`
  - Subscribe to `jms.samples.durablechat`
- *HierarchicalChat*:
  - Publish to `jms.samples.hierarchicalchat`
  - Subscribe to `jms.samples.*`

You can see that each of the applications is publishing to a different topic. However, the `HierarchicalChat` application is subscribing to a topic that ends in asterisk (**`*`**), a wildcard that accepts any topic with the root `jms.samples`. It is important to note that this is referred to a hierarchical wildcard, as it must be adjacent to dot delimiters.

Running the sample

To run the HierarchicalChatting sample do the following:

1. In window 1, enter **`ant chat1`**
2. In window 2, enter **`ant durable1`**
3. In window 3, enter **`ant wildcard`**
4. In the **wildcard** window, enter some text and then press **Enter**.

   The message is displayed in only that window.
5. In the **chat1** window, enter some text and then press **Enter**.

   The message is displayed in that window and in the **wildcard** window.
6. In the **durable1** window, enter some text and then press **Enter**.

   The message is displayed in that window and in the **wildcard** window.

Stopping the sample

To stop the wildcard session press **Ctrl**+**C** in window 3—the **wildcard** window.

MessageMonitor Application

What the sample does

The `MessageMonitor` sample application provides an example of a supervisory application with a graphical interface. By subscribing with the wildcard syntax used in the `HierarchicalChat`, the monitor gets messages on all topics in the topic hierarchy. The application listens for any message activity, and then displays each message in its window.

Running the sample

To start the MessageMonitor sample, enter **`ant tmonitor`** in window 3.

The **MessageMonitor** Java window opens.

To send messages to the MessageMonitor do the following:

1. In window 1, type **`Hello`**, and press **Enter**.

   The message displays in the **MessageMonitor** window, noting that the message was received on the `jms.samples.chat` topic

2. In window 2, type **`Hello`**, and press **Enter**.

   The message displays in the **MessageMonitor** window, noting that the message was received on the `jms.samples.durablechat` topic.

   Because the *MessageMonitor* subscribes to the topic `jms.samples.*`, messages are received from both publishers. The `Chat` and `DurableChat` applications subscribe to only their respective topics.

3. Click the **Clear** button to empty the listed messages.

Stopping the sample

To stop the applications, press **Ctrl**+**C** in each of the windows.

SelectorChat Application

What the sample shows

While specific queues and topics provide focused content nodes for messages that are of interest to an application, there are circumstances where the application developer might want to qualify the scope of interest a consumer has in messages using a syntax similar to an SQL `WHERE` clause.

In the `SelectorChat` samples, each of the `SelectorChat` samples publishes to the `jms.samples.chat` topic with messages that have a property set to

a different value, and then sets the subscriber to select only messages set to the appropriate property value.

Running the sample

To run the SelectorChatting samples do the following:

1. In window 1, enter **`ant filterchat1`**.
2. In window 2, enter **`ant filterchat2`**.
3. In window 3, enter **`ant chat1`**.
4. In the **SelectiveChatter_1** window, enter some text and then press **Enter**.

   The message is displayed in that window and the **Chatter_1** window.
5. In the **SelectiveChatter_2** window, enter some text and then press **Enter**.

   The message is displayed in that window and the **Chatter_1** window.
6. In the **Chatter_1** window, enter some text and then press **Enter**.

   The message is displayed only in that window.

Stopping the samples

To stop the **SelectorChat** applications, press **Ctrl**+**C** in window 1 and in window 2.

TransactedChat Application

What that sample shows

Transacted messages are a group of messages that form a single unit of work. Much like an accounting transaction made up of a set of balancing entries, a messaging example might be a set of financial statistics where each entry is a completely formed message and the full set of data comprises the update.

A session is declared as *transacted* when the session is created. While producers—PTP Senders and Pub/Sub Publishers—produce messages as usual, the messages are stored at the broker until the broker is notified to act on the transaction by delivering or deleting the messages. To determine when the transaction is complete, the programmer must:

- Call the method to *commit* the set of messages. The session's `commit( )` method tells the broker to sequentially release each of the messages that

have been cached since the last transaction. In this sample, the commit case is set for the string `COMMIT`.

- Call the method to *roll back* the set of messages. The session's `rollback( )` method tells the broker to flush all the messages that have been cached since the last transaction ended. In this sample, the rollback case is set for the string `CANCEL`.

Running the sample

To run the Pub/Sub TransactedChat samples do the following:

1. In window 1, enter **`ant xnchat`**.
2. In the console window that opens, type text in the window and press **Enter**.
3. Type **`COMMIT`** in the window and press **Enter**.

   The message is delivered.
4. Type text in the window and press **Enter**.
5. Repeat to produce a few messages.
6. Type **`COMMIT`** in the window and press **Enter**.

   The batch of messages is delivered as a series of individual messages.
7. Type text in the window and press **Enter**.
8. Repeat to produce a few messages.
9. Type **`CANCEL`** in the window and press **Enter**.

   The batch of messages is dropped.

Subsequent entries will form a new transaction to either commit or rollback.

Stopping the sample

To stop all the applications, press **Ctrl**+**C** in each of the windows.

Running Point-to-Point Messaging Samples

The following samples demonstrates how Point-to-point messaging differs from Publish and Subscribe messaging.

Talk Application

What the sample does

In the `Talk` application, whenever a text message is sent to a given queue, all active `Talk` applications that are waiting to receive messages on that queue take turns receiving the message at the front of the queue.

Running the sample

To run the Talking sample do the following:

1. In window 1, enter **`ant talk1`**.
2. In window 2, enter **`ant talk2`**.
3. In window 3, enter **`ant talk3`**.
4. In the **Talker1** window, type **`1`**, and then press **Enter**.

   The text is displayed in only one of the other `Q1` receiver windows. A point-to-point message has only one receiver.
5. Again, in the **Talker1** window, type **`2`**, and then press **Enter**.

   The text is displayed in the other `Q1` receiver window. Multiple receivers on a queue take turns receiving messages.
6. In the **Talker1** window, create several messages, such as **`3`** ,**`4`**, ..., **`9`**.

   Important

   Be sure to press **Enter** between each message.

   One of the `Q1` receivers gets messages `1, 3, 5, 7, 9` while the other gets `2, 4, 6, 8`.

If you opened another **Talker_2** or **Talker_3** window, the distribution to the `Q1` receivers would be `1, 4, 7` for the first, `2, 5, 8` for the next and `3, 6, 9` for the third.

Stopping the sample

To stop the Talk sessions, press **Ctrl+C** in the **Talk2** and the **Talk3** window.

The QueueMonitor Application

What the sample does

The QueueMonitor moves through a specified set of queues, listing the active messages it finds as it examines each queue. In the examples to this point, the Talk samples left no messages in any queue.

Comparing MessageMonitor and QueueMonitor

The monitor samples each open GUI windows that provide a scrolling array of its contents. The nature of the two monitors underscores fundamental differences between the Publish and Subscribe messaging model and the Point-to-point messaging model. The differences between MessageMonitor and QueueMonitor are as follows:

What messages are displayed?

- *MessageMonitor:* Delivered.
- *QueueMonitor:* Undelivered.

When does the display update?

- *MessageMonitor:* When a message is published to a subscribed topic, it is added to the displayed list.
- *QueueMonitor:* When you click the ***Browse Queues*** button, the list is refreshed.

When does the message go away?

- *MessageMonitor:* When the display is cleared for any reason.
- *QueueMonitor:* When the message is delivered (or when it expires).

What happens when the broker and monitor are restarted?

- *MessageMonitor:* As messages are listed at the moment they are delivered, there are no messages in the **`MessageMonitor`** until new deliveries occur.

- *QueueMonitor:* Listed messages marked `PERSISTENT` are stored in the broker persistent storage mechanism. They are redisplayed when the broker and the ***QueueMonitor*** restart and then choose to browse queues.

Running the sample

To run the QueueMonitor sample do the following:

1. In window 3, enter **`ant qmonitor`**.

   The `QueueMonitor`'s console window lists the queues that have been specified for it to browse.

   The **QueueMonitor** Java window opens.

2. Click **Browse Queues**.

   The messages in the queue at the moment it is browsed are listed. If you are following along carefully, there should be no messages in any queue.

To put messages into a queue do the following:

1. In window 1 (where `Talker1` is still running), type **`1`** and then press **Enter**.
2. Repeat step 1 to create a few messages, such as **`2`** **Enter**, **`3`** **Enter**, **`4`** **Enter**.
3. In the **QueueBrowser** window, click **Browse Queues**.

   The messages are in the queue. They will continue to be there until a receiver receives them on that queue, or they expire (set to 30 minutes by the sender).

The messages that are waiting on the queue will get delivered to the next receiver that chooses to receive from that queue.

To receive the queued messages do the following:

1. In window 2, enter **`ant talk2`**.
2. When the **Talker_2** window opens, it shows that it consumes the messages in the queue is sequence.

3. In the **QueueBrowser** window, click **Browse Queues**.

   The queues are all empty. As long as you have receivers on the sample queues, no messages will display in the **QueueMonitor** window.

Stopping the sample

To stop all the applications, press **Ctrl**+**C** in each of the windows.

SelectorTalk Application

What the sample does

The `SelectorTalk` sample applications are similar yet consistent with the behavior of the messaging model. The **`SelectiveTalkers`** both send and receive on **`Q1`**. The **`Talkers`** do not specify selection parameters.

Running the sample

To run the SelectorTalk sample do the following:

1. In window 1, enter **`ant filtertalk1`**.

2. In window 2, enter **`ant filtertalk2`**.

3. In window 3, enter **`ant talk2`**.

   `Talker_2` sends to `Q2` and receives on `Q1`.

4. In the **SelectiveTalker_1** window, enter some text and then press **Enter**. Send a few messages in this window.

   The messages are displayed in *either* that window or the **Talker_2** window (usually alternately.)

5. In the **SelectiveTalker_2** window, enter some text and then press **Enter**. Send a few messages in this window.

   The messages are displayed in *either* that window or the **Talker_2** window (usually alternately.)

6. In the **Talker_2** window, enter some text and then press **Enter**.

   None of the windows receives the message. **Talker_2** is receiving on `Q2`. The selective talkers are receiving on `Q1`, but they are qualifying their selection as only messages that have the specified property, and, at that, set to their preferred value. So the message will be stored in `Q1` even though there are receivers on `Q1`.

7. In window 3, press **Ctrl**+**C** to stop `Talker_2`.

8. In window 3, enter **`ant talk1`**.

   `Talker_1` sends to `Q2` and receives on `Q1`. When it starts, it immediately receives the messages stored on `Q1` (unless they have expired.)

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

TransactedTalk Application

What the sample does

The transaction samples show that the transaction scope is between the client in the JMS session and the broker. When the broker receives commitment, the messages are placed onto queues or topics in the order in which they were buffered as standard messages. The following message delivery is normal:

- **Pub/Sub Messages** — Messages are delivered in the order entered in the transaction yet influenced by the priority setting of these and other messages, the use of additional receiving sessions, and the use of additional or alternate topics. The messages are not delivered as a group.
- **PTP Messages** — The order of messages in the queue is maintained with adjustments for priority differences but there is no guarantee that—when multiple consumers are active on the queue—a `MessageConsumer` will receive one or more of the `MessageProducer`'s transacted messages.

Running the sample

To run PTP TransactedTalk sample do the following:

1. In window 1, enter **`ant xntalk`**.

2. In the console window that opens, type text in the window and press **Enter**.

3. Repeat to produce a few messages.

4. Type **`COMMIT`** in the window and press **Enter**.

5. In window 2, enter **`ant qmonitor`**.

6. In the **QueueBrowser** window, click **Browse Queues**.

   The messages are listed.

7. In window 3, enter **`ant talk2`**.

   The window opens with the series of messages in `Q1` from the **`TransactedTalker`**.

8. In window 1, type some text, and press **Enter**.

9. Repeat to produce a few messages.

10. Type **`CANCEL`** in the window and press **Enter**.

    The batch of messages is dropped:

    - In window 3—the **talk2** window, no messages are received on `Q1`.
    - In the **QueueBrowser** window, **Browse Queues** lists no messages in `Q1`.

11. In window 1, do the following:

    a. Type some text.

    b. Press **Enter**.

    c. Type **`COMMIT`**.

    d. Press **Enter.**

    The newly-committed transaction batch is received in window 3.

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

Running Request and Reply Samples

Loosely coupled applications require special techniques when it is important for the publisher to certify that a message was delivered in either messaging domain:

- **Publish and Subscribe** — While the publisher can send long-lived messages to durable receivers and get acknowledgement from the broker, neither of these techniques confirms that a message was actually delivered or how many, if any, subscribers received the message.
- **Point-to-point** — While a sender can see if a message was removed from a queue, implying that it was delivered, there is no indication where it went.

A message producer can request a reply when a message is sent. A common way to do this is to set up a **temporary destination** and header information that the consumer can use to create a reply to the sender of the original message.

In both request and reply samples, the replier's task is a simple data processing exercise: standardize the case of the text sent—receive text and send back the same text as either all uppercase characters or all lowercase characters—then publish the modified message to the temporary destination that was set up for the reply.

While request-and-reply provides proof of delivery, it is a blocking transaction—the requestor waits until the reply arrives. While this situation might be appropriate for a system that, for example, issues lottery tickets, it might be preferable in other situations to have a formally established return destination that echoes the original message and a **correlation identifier**—a designated identifier that certifies that each reply is referred to its original requestor.

The sample applications use JMS sample classes `TopicRequestor` and `QueueRequestor`. You should create the Request/Reply helper classes that are appropriate for your application.

These request and reply samples show that request/reply mechanisms are very similar across messaging models, and that, while there might be zero or many subscriber replies, there will be, at most, one PTP reply.

Request and Reply (Pub/Sub)

What the sample does

In this example in the Pub/Sub domain, the replier application must be started before the requestor so that the Pub/Sub replier's message listener can receive the message and release the blocked requestor.

Running the sample

To run the Pub/Sub Request and Reply sample do the following:

1. In window 1, enter **`ant trequest`**.
2. In window 2, enter **`ant treply`**.
3. In the **Requestor** window, type **`AaBbCc`** then press **Enter**.

   The **Replier** window reflects the activity, displaying:

   ```
   [Request] RequestingChatter:
                               AaBbCc
   ```

   The replier completes its operation (converts text to uppercase) and sends the result in a message to the requestor. The requestor gets the reply from the replier:

   ```
   [Reply] Transformed RequestingChatter: AaBbCc to all uppercase:
    REQUESTINGCHATTER: AABBCC
   ```

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

Request and Reply (PTP)

Overview

In the PTP domain, the requestor application can be started and even send a message before the replier application is started. The queue holds the message until the replier is available. The requestor is still blocked, but when the replier's message listener receives the message, it releases the blocked requestor. The sample code includes an option (`-m`) to switch the mode between uppercase and lowercase.

Running the sample

To run the PTP Request and Reply sessions do the following:

1. In window 1, enter: **`ant qrequest`**.
2. In window 2, enter: **`ant qreply`**.
3. In the `Requestor` window, type **`AaBbCc`** then press **Enter**.

   The **Replier** window reflects the activity, displaying:

   ```
   [Request] RequestingTalker:
                              AaBbCc
   ```

   The replier completes its operation (converts text to uppercase) and sends the result in a message to the requestor. The requestor gets the reply from the replier:

   ```
   [Reply] Transformed RequestingTalker: AaBbCc to all uppercase:
   REQUESTINGTALKER: AABBCC
   ```

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

Running the Queue Test Loop Sample

Overview

A simple loop test lets you experiment with messaging performance.

The `RoundTrip` sample application sends a brief message to a sample queue and then uses a temporary queue to receive the message back. A counter is incremented and the message is sent for another trip. After completing the number of cycles you entered when you started the test, the run completes by displaying summary and average statistics.

Note

This sample is not intended as a performance tool.

Running the sample

To run the QueueRoundTrip sample enter **`ant roundtrip`** in Window 1.

The `QueueRoundTrip` window produces and consumes queue messages in a loop, producing the next message after the prior one has been consumed. When it has completed the specified number of cycles (set to 10,000 for the sample run), it reports how long it took to complete all the cycles.

Changing Parameters and Modifying Source Code

Revising Parameters in the Build File 58
Analyzing and Modifying the Java Source Files 61

While the exploration of the JMS samples ran the compiled applications, it used an Ant build file provide optimal cross-platform porting and to preset some parameter values to keep the samples on a defined track.

- **Modify and extend application parameters in the Ant build file**—You will change a few of the parameters and reset some that were allowed to use their default values. Then, you will run the same samples to observe the changed behaviors.

- **Analyze and modify the JMS methods and patterns in the Java source files** —You will examine some of the sample application source files to see the JMS methods in the samples. The you will change some sample applications to revise the quality of service and the functionality in the scope of the original application. Then, compiling and running the application will let us test out the changes.

Revising Parameters in the Build File

Basics

The Ant `build.xml` file was revised to add the FUSE Message Broker version and installation location. Now you will change and add parameters in the `target` section for each sample run. Each parameter's name-value pair is presented as two sequential arguments. For example:

```
<target name="chat1">
          <java classname="Chat" ...>
          ...
          <arg value="-u"/>
          <arg value="Chatter1"/>
          </java>
          </target>
```

That listing shows that the `-u` parameter, the user name, is assigned the value `"Chatter1"`

Changing the User Name

Change the user name value from `"Chatter1"` to `"Fred"` as shown:

```
<target name="chat1">
          <java classname="Chat" ...>
          ...
          <arg value="-u"/>
          <arg value="Fred"/>
          </java>
          </target>
```

Save the file, and then enter `ant chat1` in one of the sample windows, type `Hello`, and then press `Enter`. The response line is:

```
[java] Fred: Hello
```

Note

The *`password`* parameter requires implementation of security. For these samples, that would require all the user names and their respective passwords be defined in the security store. That goes beyond the scope of exploring JMS, and will be discussed in the configuration guide. There, you will learn about the simple authentication plugin (so you can specify credentials directly in the configuration file), and how to access and set up the Java

Authentication and Authorization Service (JAAS) authentication plugin for a more powerful and customizable solution.

Changing destination names

The queue and topic names used in the sample are arbitrary and can be created dynamically. Some TopicPubSub application applications let you specify the publish topic and the subscribe topic as parameters. For `wildcard`, the `HierarchicalChat` exposes the topic to which the application publishes as the `-t` parameter, and the topic to which it subscribes as the `-s` parameter. You could change these topics to demonstrate a FUSE Message Broker feature by extending the publish topic to a fourth level, and changing the wildcard on the subscriber topic from `*` to `>`. That expands the wildcard from all topics at the current hierarchical level to all topics at that level or deeper.

```
<target name="wildcard">
          <java classname="HierarchicalChat" fork="true">
          ...
          <arg value="-u"/>
          <arg value="HierarchicalChatter"/>
          </java>
          </target>
```

Other topic and queues could be changed in the build file so you can observe how they interact with other sample applications. For the **MessageMonitor** application, change its depth of topic hierarchies in its properties file, **MessageMonitor.properties**, to **subscriptionTopics jms.samples.>**

Changing the queue round trips

The QueueRoundTrip application is set in the build file to iterate 10,000 times through sending a message to a queue, receiving it off the queue, and taking that as the cue to send another message. You can change that number to a larger or smaller value to see start and stop operations are impactful. Try 1000 (one thousand), save the build file, and run **ant roundtrip**. Try 100000 (one hundred thousand), save the build file, and run **ant roundtrip**. For example:

```
<target name="chat1">
          <java classname="QueueRoundTrip" ...>
          ...
          <arg value="-n"/>
          <arg value="1000"/>
```

```
          </java>
          </target>
```

Distributing the client and the broker

At this point in exploring JMS, the broker used by the samples is always on the computer where the sample applications run. While you might use a local or an embedded broker, JMS messaging is designed so that the sample applications can run on a computer that has the appropriate libraries, yet can connect to a broker on a different system to produce and consume messages. The standalone broker system would typically by in a location where it can be monitored and provided resources that ensure optimal availability to any applications that use it.

The sample applications provide a `-b` parameter to specify the protocol, host, and port of the preferred broker. As the default broker configuration specifies the TCP protocol on `localhost`, listening on port 61616, you can use another installation of the Java, FUSE Message Broker, Ant, and the Exploring JMS files (as described in the previous chapter) on another computer to experience distributed connection. On one host set up and start the broker. On the other host (the remote host), do the same.

Stop the broker on the remote host, then modify the `build.xml` file on the remote host to specify add the connection parameter and specify the host name where the broker is running. For example:

```
<target name="chat1">
          <java classname="Chat" ...>
          ...
          <arg value="-u"/>
          <arg value="Fred"/>
          <arg value="-b"/>
          <arg value="tcp://remoteHostName:61616"/>
          </java>
          </target>
```

Save the build file and then run `chat1` in a sample window on each of the computers. When you enter messages in either `Chatter_1` window, the subscribers both get the message from the same broker.

Analyzing and Modifying the Java Source Files

Overview

Now that you are familiar with JMS behavior, let's look inside the application source files to examine some of the patterns that are used. You will see how you can make and compile some changes to the source files that expose FUSE Message Broker client's JMS features you can then test. (There are features that require setting broker configurations; those will be discussed in a forthcoming chapter.)

Basics

The application source files and the class files that they create are located in application-specific folders that are subfolders of the two messaging models, `QueuePTPSamples`, and `TopicPubSubSamples`. The build file enabled you to enter ant commands at the root of the samples such that **ant chat1** did the same as navigating to the `TopicPubSubSamples/Chat` directory to run the `Chat.class` file at that location as:

```
java Chat -u Chatter1
```

The `Chat` folder contains the source file `Chat.java`.

When you make changes to that or any other `.java` file, compile the source file to an updated class file and then run the modified file. To do this, you need to have a Java Development Kit (JDK) specified as `JAVA_HOME`, and the `CLASSPATH` needs to specify the JDK's tools JAR, the FUSE Message Broker's activemq core JAR, and the Geronimo JMS JAR. For example, on a Windows system, you might do the following:

```
set JAVA_HOME=C:\jdk1.5.0_11
set FUSE_MB_=C:\progress\fuse-message-broker-5.3.0.0
set CLASSPATH=.;%JAVA_HOME%\lib\tools.jar;   \
                %FUSE_MB%\lib\activemq-core-5.3.0.0-fuse.jar;   \
                %FUSE_MB%\lib\geronimo-jms_1.1_spec-1.1.1.jar
```

Then, with the command line located at the `Chat` directory, enter:

```
javac Chat.java
```

Setting noLocal in chat

The TopicPubSub samples show that the message is received by every subscriber to the topic and its hierarchy -- including the publisher's connection. A feature of JMS is the ability to set a `noLocal` Boolean on the subscription that inhibits echoing messages sent in the publisher's connection. In `Chat.java` you can set the `noLocal` value to `true` but there is a catch: the

method signature that sets `noLocal` requires a message selector string also. On line 82 of `Chat.java`, the message consumer is listed as:

```
javax.jms.MessageConsumer subscriber = subSession.createConsumer(topic);
```

Edit the line to set the message selector to a zero-length String to satisfy the method signature, and then add the boolean value `true`, as follows:

```
javax.jms.MessageConsumer subscriber = subSession.createConsumer(topic, "", true);
```

When you save and compile this sample, see the effect in your samples. Start `chat1` and `chat2`. Send some messages from each of the chatters. The messages are not echoed in that sender's window.

Steal this code!

Use the patterns in these sample applications to meld them and transform them from user interactive examples to applications that pass your data to the message producer, and that take messages received by message consumers to move them into your data stores and application logic. As always, provide proper copyrights and licenses in your source files and application packages.

Index

A
AJAX, 13
Apache ActiveMQ, 12
Asynchronous JavaScript and XML (see AJAX)
authentication, 18
authorization, 18

B
broker
  embedded, 22
  standalone, 22
  topologies, 22

C
clustering, 15, 16
configuration, 25

D
databases, 17

E
encryption, 18

F
failover, 15

H
high availability, 15
high performance journal, 19

J
J2EE, 13
Java Message Service (see JMS)
Java Naming and Directory Interface (see JNDI)
JMS, 13, 27
  broker, 27
  clients, 29
  consumers, 29
  messages, 27
  producers, 29
  queues, 27
  streams, 16
  topics, 28
JNDI, 13

M
message compression, 16

N
network connectors, 25

P
performance, 19
persistence, 17

R
Representational State Transfer (see REST)
REST, 13

S
scalability, 16
security, 18
Spring Framework, 25

T
transport connectors, 25

X
XBean, 25