

FUSE™ Message Broker

Getting Started with FUSE™ Message Broker

Version 5.2
December 2008

Getting Started with FUSE[™] Message Broker

Progress Software

Version 5.2

Published 02 Dec 2008

Copyright © 2008 IONA Technologies PLC , a wholly-owned subsidiary of Progress Software Corporation.

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 IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC 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

Preface	9
The FUSE Message Broker Library	10
Open Source Project Resources	11
Document Conventions	12
Introducing FUSE Message Broker	15
What is FUSE Message Broker?	16
Supported Standards	17
Supported Wire Protocols and Clients	18
High Availability	19
Scalability	20
Persistence	21
Security	22
Performance	23
Key Concepts	25
JMS Basics	26
Broker Deployment Options	29
Configuring FUSE Message Broker	30
Basic Tasks	33
Starting a Broker	34
Testing the Installation	36
Monitoring FUSE Message Broker	37
Stopping a Broker	38
Running the Java Example	39
Before you begin	40
Running the broker	41
Running the Consumer	42
Running the Producer	44
Monitoring the Broker	45
Configuring the Example	47
Index	49

List of Figures

1. Point-to-point Messaging	27
2. Publish-Subscribe Messaging	28
3. A Network of Brokers	29
4. Transport Connectors	31
5. Network Connectors	31
6. The Advanced Tab in JConsole	45
7. Queue Attributes in JConsole	46

List of Examples

1. Defining Transport and Network Connectors	30
2. Starting an Embedded Broker	34
3. Starting a Named Embedded Broker	34
4. Connecting to a Broker	42
5. Creating a Queue or Topic	42
6. Attaching to a Queue or Topic	44

Preface

The FUSE Message Broker Library	10
Open Source Project Resources	11
Document Conventions	12

The FUSE Message Broker Library

The FUSE Message Broker documentation library consists of the following books:

- [Installing FUSE™ Message Broker](#) discusses the requirements and procedures for installing FUSE Message Broker
- [Getting Started with FUSE™ Message Broker on page 1](#) provides an overview of the central concepts behind FUSE Message Broker and walks you through a simple example.
- [Connectivity Guide](#) explains the different wire protocols and transports that FUSE Message Broker supports.
- [Using FUSE™ Message Broker's Persistence Features](#) describes how to enable message persistence using the AMQ Message Store or a relational database in FUSE Message Broker.

Open Source Project Resources

Apache CXF

Web site: <http://cxf.apache.org/>

User's list: <user@cxf.apache.org>

Apache Tomcat

Web site: <http://tomcat.apache.org/>

User's list: <users@tomcat.apache.org>

Apache ActiveMQ

Web site: <http://activemq.apache.org/>

User's list: <users@activemq.apache.org>

Apache Camel

Web site:
<http://activemq.apache.org/camel/enterprise-integration-patterns.html>

User's list: <camel-user@activemq.apache.org>

Apache ServiceMix

Web site: <http://servicemix.apache.org>

User's list: <users@servicemix.apache.org>

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

<code>fixed width</code>	<p>Fixed width (Courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the <code>javax.xml.ws.Endpoint</code> class.</p> <p>Constant width paragraphs represent code examples or information a system displays on the screen. For example:</p> <pre>import java.util.logging.Logger;</pre>
<i>Fixed width italic</i>	<p>Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:</p> <pre>% cd /users/YourUserName</pre>
<i>Italic</i>	<p>Italic words in normal text represent <i>emphasis</i> and introduce <i>new terms</i>.</p>
Bold	<p>Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the User Preferences dialog.</p>

Keying conventions






This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
...	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.

	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in {} (braces).
--	---

Admonition conventions

This book uses the following conventions for admonitions:

	Notes display information that may be useful, but not critical.
	Tips provide hints about completing a task or using a tool. They may also provide information about workarounds to possible problems.
	Important notes display information that is critical to the task at hand.
	Cautions display information about likely errors that can be encountered. These errors are unlikely to cause damage to your data or your systems.
	Warnings display information about errors that may cause damage to your systems. Possible damage from these errors include system failures and loss of data.

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?	16
Supported Standards	17
Supported Wire Protocols and Clients	18
High Availability	19
Scalability	20
Persistence	21
Security	22
Performance	23

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 FUSE™ Message Broker's Persistence Features](#) 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.

Key Concepts

This chapter introduces some concepts that are key to understanding the Java Message Service (JMS) API and FUSE Message Broker.

JMS Basics	26
Broker Deployment Options	29
Configuring FUSE Message Broker	30

JMS Basics

JMS components

Java Message Service (JMS) is a Java Message Oriented Middleware (MOM) API for sending messages between two or more 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](http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html) [<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html>]

A JMS system is comprised of the following components:

- Providers
- Messages
- Clients
- Destinations

Providers

The JMS provider is a messaging system that implements the JMS interfaces. Message broker such as Apache ActiveMQ and FUSE Message Broker are examples of providers.

Messages

A messages is an object that contains the data being transferred between JMS clients.

Clients

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

Producer

Producers create messages and send them to the broker for delivery to a particular destination.

Consumer

Consumers retrieve messages from a destination.

Destinations

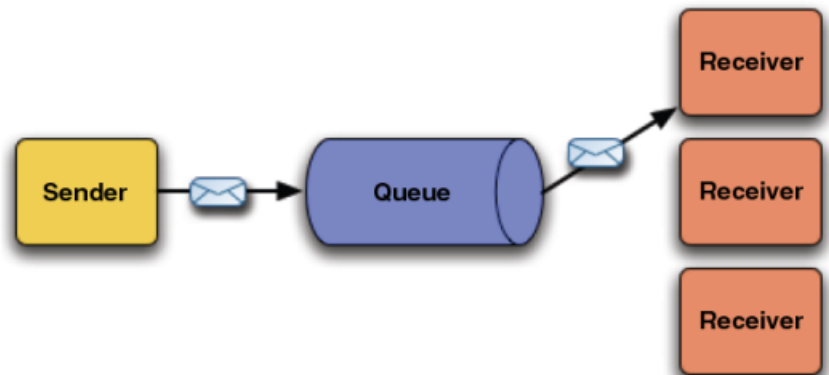
Destinations are maintained by the message broker. They can be either queues or topics.

Queues

A queue is a destination that contains messages that have been sent and are waiting to be read. Messages are delivered in the order sent. A message is removed from the queue once it has been read.

Queues are used for point-to-point or one-to-one messaging.

Figure 1. Point-to-point Messaging

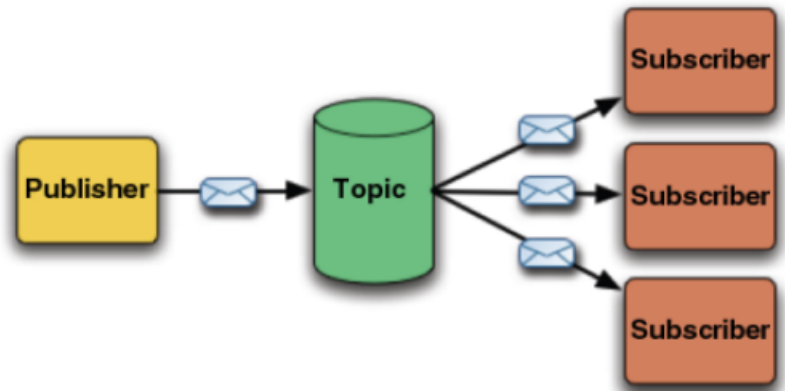


Topics

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. Publish-Subscribe Messaging



Broker Deployment Options

Broker functions

The message broker is responsible for routing messages to the correct topic or queue. It is also responsible for providing quality of service features, such as reliability, persistence, security, and high availability.

You can deploy FUSE Message Broker in either standalone or embedded mode. You can also deploy a network of brokers.

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. See [Starting an embedded broker on page 34](#)

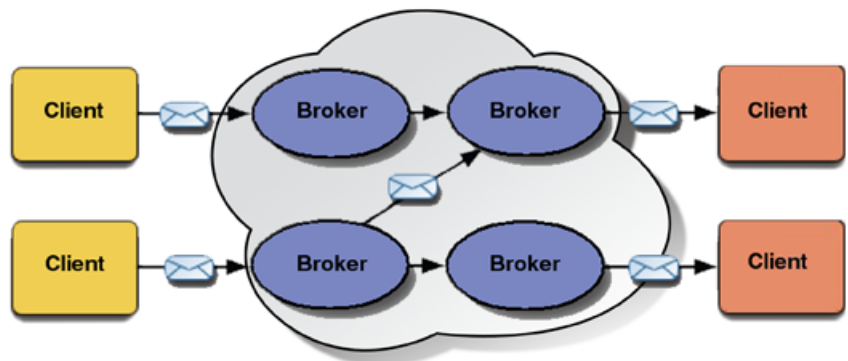
Standalone broker

A standalone broker does not share its JVM process with clients and instead communicates with clients using a network-based transport connector. See [Starting a standalone broker on page 34](#) for details.

Network of brokers

Often a number of brokers can be linked together in 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 3. 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 1. 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 just auto discover the 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 4. Transport Connectors



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

Figure 5. Network Connectors



Note

For more details on transport and network connectors, see the [Connectivity Guide](#) guide.

Basic Tasks

This chapter covers the basics of stopping, starting, testing, and monitoring FUSE Message Broker.

Starting a Broker	34
Testing the Installation	36
Monitoring FUSE Message Broker	37
Stopping a Broker	38

Starting a Broker

Starting a 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
```

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 on page 34](#)

Example 2. 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 the broker name, as shown in [Example 3 on page 34](#):

Example 3. 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 use the URI `vm://broker1`.

Testing the Installation

Overview

Once you have run the start script, a number of messages are written to the command window, including the following:

```
INFO BrokerService - ActiveMQ JMS Message Broker (localhost, ID:hostname-3074-1194432866307-0:0) started
```

Since FUSE Message Broker runs on port 61616, you can check whether that port is in use by running the **netstat** command.

Testing on Windows

To test your installation on Windows, run the following from a command prompt:

```
netstat -an|find "61616"
```

Testing on UNIX

On UNIX, run the following from a terminal window:

```
netstat -an|grep 61616
```

Running the Web demos

FUSE Message Broker includes a number of Web demos and examples that you can use to become familiar with the product.

To run the Web demos, ensure that the broker is running and then open the following URL in your browser:

```
http://localhost:8161/demo
```

Monitoring FUSE Message Broker

Overview

You can monitor FUSE Message Broker using one of the following:

- The ActiveMQ Web Console
 - JMX
-

Using the Web Console

Once FUSE Message Broker is running, you can access the Web Console by opening the following URL in your browser:

```
http://localhost:8161/admin
```

From the home page of the Web Console you can click the **Send** link in the top navigation bar to send a test message to an existing queue or topic.

Click the **Queues** link to see the available queues in your installation. A queue called Example.A is available by default.

Click the **Topics** link to see the available topics in your installation.

Using JMX

To enable JMX support for FUSE Message Broker:

1. Open the following Spring XML configuration file:

```
InstallDir/conf/activemq.xml
```

2. Add a `useJmx` attribute with a value of `true` to the `broker` element, as follows:

```
<broker useJmx="true" brokerName="MyBroker">  
...  
</broker>
```

3. Run a JMX console, by running `jconsole` from `JAVA_HOME/bin`.
4. Select FUSE Message Broker in the list of connections and click **Connect**.

Stopping a Broker

Terminating the broker process

You can stop a running broker; on both Windows and UNIX by pressing **Ctrl+C** in the command window or terminal in which the process is running.

Stopping using the Admin tool

To stop a broker using the Admin tool.

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-admin stop Hostname
```

- UNIX:

```
./activemq-admin stop Hostname
```

Killing a background process on UNIX

If FUSE Message Broker was started in the background on UNIX, first identify the process ID by typing the following in a terminal window:

```
ps -ef|grep activemq
```

Then kill the process by typing the following:

```
kill PID
```

Running the Java Example

This chapter walks you through the Java example that ships with FUSE Message Broker .

Before you begin	40
Running the broker	41
Running the Consumer	42
Running the Producer	44
Monitoring the Broker	45
Configuring the Example	47

Before you begin

Prerequisites

Before you run the Java example, ensure that you have the required version of Apache Ant and of the Java Development Kit installed. See [Java and Compiler Requirements](#) in the *Installing FUSE™ Message Broker* for details.

Example location

You can use Ant to run the example code from the root of the `example` folder in the FUSE Message Broker installation directory. Each Ant target is explained below.

The Java source code is contained in the `src` folder.

What happens

By default, the example allows you to do the following:

1. Run an embedded instance of FUSE Message Broker.
2. Run a consumer that waits to consume a predefined number of messages from a queue and then closes the connection.
3. Run a producer that sends the messages to the queue.

Running the broker

Running an embedded broker

Embedding a broker involves deploying an instance of FUSE Message Broker inside a JVM.



Note

You can also simply run a standalone broker by running the **activemq** script from the `bin` directory under the FUSE Message Broker installation directory.

To run the embedded broker:

1. In a new command window, change directory to the `example` directory in the FUSE Message Broker installation directory.
2. Type the following:

```
ant embedBroker
```

The code explained

The **embedBroker** Ant target points to the `EmbeddedBroker.java` class, which creates a new broker as shown in [Example 2 on page 34](#).

Running the Consumer

Using Ant

To run the consumer, type **ant consumer** at the command prompt. By default, the consumer consumes the published messages from a queue.

The code explained

The **consumer** Ant target points to the `consumerTool.java` class, which uses the `ActiveMQConnectionFactory` class to connect to the broker, as shown:

Example 4. Connecting to a Broker

```
public class ConsumerTool {  
    ...  
    private String user = ActiveMQConnection.DEFAULT_USER;  
    private String password = ActiveMQConnection.DEFAULT_PASSWORD;  
    private String url = ActiveMQConnection.DEFAULT_BROKER_URL;  
    ...  
    // Create the connection.  
    ActiveMQConnectionFactory connectionFactory = new ActiveMQCon  
    nectionFactory(user, password, url);  
    connection = connectionFactory.createConnection();  
    connection.start();  
    ...  
}
```

The consumer uses `createQueue()` to create a queue or `createTopic()` depending on which arguments you pass in at runtime. The queue or topic name is defined as `TEST.FOO` in the `build.xml`.

Example 5. Creating a Queue or Topic

```
...  
private Session session;  
private String subject = "TOOL.DEFAULT";  
private boolean transacted;  
...  
// Create the session  
Session session = connection.createSession(transacted, Ses  
sion.AUTO_ACKNOWLEDGE);  
if (topic) {  
    destination = session.createTopic(subject);  
}
```

```
} else {  
    destination = session.createQueue(subject);  
}
```

Running the Producer

Using Ant

Use **ant producer** to send a given number of messages to a queue or topic.

The code explained

The **producer** Ant target points to the `producerTool.java` class, which again uses `ActiveMQConnectionFactory` to connect to the broker, as shown in [Example 4 on page 42](#).

The producer uses `createQueue()` or `createTopic()` to publish to a destination.



Note

If the destination already exists, the `createQueue()` and `createTopic()` are simply used to attach to the destination.

Example 6. Attaching to a Queue or Topic

```
// Create the session
Session session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
if (topic) {
    destination = session.createTopic(subject);
} else {
    destination = session.createQueue(subject);
}
```

Monitoring the Broker

Overview

You can use any Java Management Extensions (JMX)-compliant console, such as JConsole or MC4J to monitor the messages passed between the producer, the broker, and the consumer.

Using JConsole

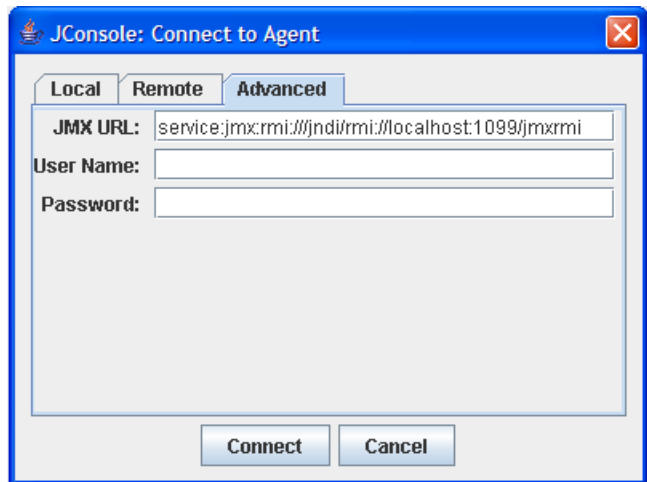
To monitor the messages using JConsole:

1. Ensure that the `JAVA_HOME` environment is set.
2. From a command prompt, run **jconsole**.
3. In the JConsole window, click the **Advanced** tab.
4. Enter the following into the **JMX URL** field.

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
```

5. Click **Connect**.

Figure 6. The Advanced Tab in JConsole



6. Click the **MBeans** tab.

7. Under the **Tree** node in the MBeans panel, expand **org.apache.activemq** **localhost Queue**.
8. Open **TEST.FOO** and in the **Attributes** tab, check that the messages have been sent to the queue.

Figure 7. Queue Attributes in JConsole

Attributes	Operations	Notifications	Info
Name	Value		
AverageEnqueueTime	0.9905		
ConsumerCount	0		
DequeueCount	2000		
DispatchCount	2000		
EnqueueCount	4010		
MaxAuditDepth	2048		
MaxEnqueueTime	280		
MaxProducersToAudit	1024		
MemoryLimit	30198988		
MemoryPercentUsage	13		
MemoryUsagePortion	1.0		
MinEnqueueTime	10		
Name	TEST.FOO		
ProducerCount	0		
ProducerFlowControl	true		
QueueSize	2010		

Configuring the Example

Overview

You can configure the producer and the consumer in the Java example in a number of different ways by passing `-D` plus option flags to the clients at runtime, as shown:

ant consumer -Durl=tcp://hostname:1234 -Dtopic=true

Consumer and producer configuration options

You can configure both the consumer and the producer by passing them the following properties.

`url`

Used to specify a custom URL for the broker. For example,

`tcp://hostname:1234.`

`topic`

A boolean to determine whether to use topics or queues. Defaults to `false`.

`subject`

Used to specify a custom destination. For example, `MyQueue` or `MyTopic`.

`durable`

A boolean to specify that you want to create a durable topic. Defaults to `false`.

`max`

The maximum number of messages to be produced or consumed before the client shuts down.

`transacted`

A boolean to specify whether transactions should be used. Defaults to `false`.

`sleepTime`

The time to wait between message consumptions.

`verbose`

Used to print out more information. Defaults to `true`.

Consumer-only configuration options

When running the consumer, you can also pass in the following arguments:

`clientId`

A string used to identify the client.

`ack-mode`

Sets the type of acknowledgement to use. See the [JavaDoc](http://java.sun.com/products/jms/javadoc-102a/javax/jms/Session.html) [http://java.sun.com/products/jms/javadoc-102a/javax/jms/Session.html] for `java.jmx.Session` for details.

`receive-time-out`

An integer that specifies the time to wait until the message is consumed.

Index

A

AJAX, 17
Apache ActiveMQ, 16
Asynchronous JavaScript and XML (see AJAX)
authentication, 22
authorization, 22

B

broker
 deployment options, 29
 embedded, 29, 41
 monitoring, 45
 standalone, 29
 starting, 34
 stopping, 38

C

clustering, 19, 20
configuration, 30

D

databases, 21

E

encryption, 22
example
 configuring, 47
 running, 39

F

failover, 19

H

high availability, 19
high performance journal, 23

J

J2EE, 17
Java Message Service (see JMS)
Java Naming and Directory Interface (see JNDI)
JConsole, 45
JMS, 17, 26
 clients, 26
 consumers, 27
 messages, 26
 producers, 26
 provider, 26
 queues, 27
 streams, 20
 topics, 27
JMX, 37, 45
JNDI, 17

M

message compression, 20
monitoring, 37

N

network connectors, 30

P

performance, 23
persistence, 21

R

Representational State Transfer (see REST)
REST, 17

S

scalability, 20
security, 22
Spring Framework, 30

T

testing, 36
transport connectors, 30

X

XBean, 30