Fuse Message Broker

Connectivity Guide

Version 5.5
Febuary 2012

Connectivity Guide

Version 5.5

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux <mwp1@cornell.edu>

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE

OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE

OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Protocol Summary

Fuse Message Broker supports a wide variety of protocols for client-to-broker, broker-to-broker, and client-to-client connections. The intention is that the variety of protocols will make it easier to connect to a range of client types. Different network topologies can also be supported with the help of special protocols, such as discovery and peer-to-peer.

Simple Connections

Overview

The following protocols can be used either for straightforward client-to-broker connections (transport connector) or broker-to-broker connections (network connector). For each wire protocol (that is, on-the-wire message encoding), Fuse Message Broker supports one or more associated transport protocols. Hence, you can configure connections with a wide variety of wire protocol/transport protocol combinations.

Protocols for simple connections

Table 1.1 on page 12 shows the protocol combinations that messaging clients can use to connect directly to the message broker.

Table 1.1. Protocols for Simple Connections

Wire Protocol	Transport Protocol	Sample URL	Description
OpenWire	TCP	`tcp://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over TCP protocol. This URL is also used to configure the transport connector in a broker.
OpenWire	TCP	`nio://Host:Port`	Same as `tcp`, except that the New I/O (NIO)[1] Java API is used, which provides better performance in some scenarios.
OpenWire	SSL	`ssl://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over SSL protocol.

[1] http://en.wikipedia.org/wiki/New_I/O

Wire Protocol	Transport Protocol	Sample URL	Description
			This URL is also used to configure the transport connector in a broker.
OpenWire	HTTP	`http://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over HTTP protocol (HTTP tunneling). You can use this protocol to navigate through firewalls. This URL is also used to configure the transport connector in a broker.
OpenWire	HTTPS	`https://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over HTTPS protocol This URL is also used to configure the transport connector in a broker.
Stomp	TCP	`stomp://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the Stomp over TCP protocol. This URL is also used to configure the transport connector in a broker.
Stomp	SSL	`stomp+ssl://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the Stomp over SSL protocol. This URL is also used to configure the transport connector in a broker.

Wire Protocol	Transport Protocol	Sample URL	Description
REST	HTTP	`http://Host:Port/demo/message/FOO/BAR?timeout=10000&type=queue`	Connect to the message broker endpoint at `Host:Port` using the REST protocol. The REST endpoint is implemented as a servlet deployed in a servlet engine. For example, the sample URL is built up from a Web context name, `demo`, followed by the servlet name, `message`, followed by a destination name, `FOO/BAR`, and some query options. This URL is *not* used to configure the REST transport connector in a broker. Use the `<jetty>` tag to configure the REST endpoint in the broker.
RESTs	HTTPS	`http://Host:Port/demo/message/FOO/BAR?timeout=10000&type=queue`	
XMPP	TCP	`xmpp://Host:Port`	Configure the transport connector in a message broker to accept XMPP connections on `Host:Port` (for example, from an Instant Messaging client).
VM	*N/A*	`vm://BrokerName`	Configure clients to connect to a broker embedded within the same Java Virtual Machine (JVM). The `BrokerName` is the broker name of the embedded broker.

Discovery Protocols

Overview

A discovery protocol builds a connection to a message broker in two steps:

1. Obtain a list of available broker endpoints.
2. Connect to one or more endpoints from the discovered list, according to some selection algorithm.

Discovery protocols are particularly useful for clients that connect to a cluster of message brokers.

Summary of discovery protocols

Table 1.2 on page 15 describes the discovery protocols that clients can use.

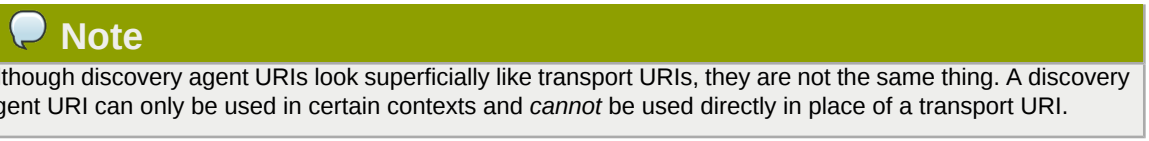
Table 1.2. Summary of Discovery Protocols

Protocol	Sample URL	Description
Failover	`failover://(uri1,...,uriN)?TransportOptions`	Configure clients to connect to one of the broker endpoints from the URI list, `uri1,...,uriN`. The transport options, `?TransportOptions`, are specified in the form of a query list. If no transport options are required, you can omit the parentheses and the question mark, `?`. For more information see "Failover Protocol" in *Fault Tolerant Messaging*.
Discovery	`discovery://(DiscoveryAgentUri)?TransportOptions`	Configure clients to connect to one of the broker endpoints from a URI list that is dynamically discovered at runtime, using a discovery agent. The discovery agent URI, `DiscoveryAgentUri`, is normally a multicast discovery agent—for example, `multicast://default`. For more information see "Dynamic Discovery Protocol" in *Using Networks of Brokers*.
Fanout	`fanout://(DiscoveryAgentUri)?TransportOptions`	Configure clients to connect to *all* of the broker endpoints from a

Protocol	Sample URL	Description
		dynamically discovered URI list. For more information see "Fanout Protocol" in *Using Networks of Brokers*.

Discovery agents

The discovery protocol supports a number of discovery agents, which are also specified in the form of a URI. For details of the supported discovery agents, see "Discovery Agents" in *Using Networks of Brokers*.



Peer-to-Peer Protocols

Overview

Peer-to-peer protocols enable messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker.

Summary of peer-to-peer protocols

Table 1.3 on page 17 describes the peer-to-peer protocols that clients can use.

Table 1.3. Summary of Peer-to-Peer Protocols

Protocol	Sample URL	Description
Peer	`peer://PeerGroup/BrokerName?BrokerOptions`	Configure clients to connect to their peers in the group with the group name, *`PeerGroup`*. The *`BrokerName`* specifies the broker name for the embedded broker. The broker options, *`BrokerOptions`*, are specified in the form of a query list (for example, `?persistent=true`).

Broker options

The `peer` protocol supports a variety of broker options. For details, see the broker options listed in Table 6.1 on page 75 .

Chapter 2. OpenWire Protocol

The OpenWire protocol is the default on-the-wire protocol for Fuse Message Broker. This chapter provides a brief introduction to the protocol, illustrating how to use OpenWire with a variety of transport protocols.

Introduction to the OpenWire Protocol

Overview

The OpenWire protocol is a JMS compliant wire protocol (defining message types and message encodings) that is native to the Fuse Message Broker. The protocol is designed to be fully-featured, JMS-compliant, and highly performant. It is the default protocol of the Fuse Message Broker.

Transport protocols

Table 2.1 on page 20 shows the transport protocols supported by the OpenWire wire protocol:

Table 2.1. Transport Protocols Supported by OpenWire

Transport Protocol	URL	Description
TCP	`tcp://Host:Port`	Endpoint URL for OpenWire over TCP/IP. The broker listens for TCP connections on the host machine, *`Host`*, and IP port, *`Port`*.
TCP	`nio://Host:Port`	Same as `tcp`, except that the New I/O (NIO)[1] Java API is used, which provides better performance in some scenarios.
SSL	`ssl://Host:Port`	*(Java clients only)* Endpoint URL for OpenWire over TCP/IP, where the socket layer is secured using SSL (or TLS). For details of how to configure an SSL connection, see *Security Guide*.
HTTP	`http://Host:Port`	*(Java clients only)* Endpoint URL for OpenWire over HTTP.
HTTPS	`https://Host:Port`	*(Java clients only)* Endpoint URL for OpenWire over HTTP, where the socket layer is secured by SSL (or TLS).

[1] http://en.wikipedia.org/wiki/New_I/O

Transport Protocol	URL	Description
		For details of how to configure a HTTPS connection, see *Security Guide*.

Transport options

OpenWire supports a number of transport options, which can be set as query options on the transport URL. For example, to specify that error messages should omit the stack trace, use a URL like the following:

```
tcp://localhost:61616?wireformat.stackTraceEnabled=false
```

Where the `wireformat.stackTraceEnabled` property is set to `false` to disable the inclusion of stack traces in error messages. Table 2.2 on page 21 gives the complete list of transport options for OpenWire.

Table 2.2. Transport Options Supported by OpenWire Protocol

Property	Default	Description	Negotiation policy
`wireformat .stackTraceEnabled`	`true`	Should the stack trace of an exception occuring on the broker be sent to the client?	Set to `false` if either side is `false`.
`wireformat .tcpNoDelayEnabled`	`false`	Provides a hint to the peer that TCP `nodelay` should be enabled on the communications Socket.	Set to `false` if either side is `false`.
`wireformat .cacheEnabled`	`true`	Should commonly repeated values be cached so that less marshalling occurs?	Set to `false` if either side is `false`.
`wireformat .cacheSize`	`1024`	If `cacheEnabled` is `true`, this property specifies the maximum number of values to cache.	Use the smaller of the two values.
`wireformat .tightEncodingEnabled`	`true`	Should wire size be optimized over CPU usage?	Set to `false` if either side is `false`.
`wireformat .prefixPacketSize`	`true`	Should the size of the packet be prefixed before	Set to `true` if both sides are `true`.

Property	Default	Description	Negotiation policy
		each packet is marshalled?	
`wireformat.maxInactivityDuration`	`30000`	The maximum inactivity duration (before which the socket is considered dead) in milliseconds. On some platforms it can take a long time for a socket to appear to die, so we allow the broker to kill connections if they are inactive for a period of time. Set to a value `<= 0` to disable inactivity monitoring.	Use the smaller of the two values.
`wireformat.maxInactivityDurationInitalDelay`			

Supported clients

Fuse Message Broker currently supports the following client types for the OpenWire protocol:

- *Java clients*—the Java API conforms fully to the JMS specification.

If you want to develop an OpenWire client using other programming languages, try one of the following client types from the Apache ActiveMQ[2] project:

- *C++ clients*—for C++ clients, Apache ActiveMQ provides the CMS (C++ Messaging Service) API, which is closely modelled on the JMS specification. Only the TCP transport is supported for C++ clients.

[2] http://activemq.apache.org/

OpenWire Example

Overview

It is relatively straightforward to try out the various OpenWire+transport combinations using the sample code provided. After configuring the broker with the requisite transport connectors, you can use the sample *producer tool* and the *consumer tool* to propagate messages through the broker using the following protocols: OpenWire over TCP or OpenWire over HTTP.

Note

The secure socket protocols—OpenWire over SSL, and OpenWire over HTTPS—are discussed in *Security Guide*.

Example prerequisites

To run these examples, you require:

- Apache Ant[3] build tool, version 1.6 or later.

Location of sample clients

The OpenWire examples depend on the sample producer and consumer clients located in the following directory:

```
ActiveMQInstallDir/fuse-message-broker-Version/example
```

Example steps

To try out the OpenWire protocol, perform the following steps:

1. "Configure the broker" .
2. "Run the broker" .
3. "Run the consumer" .

[3] http://ant.apache.org/

4. "Run the producer with the TCP protocol" .

5. "Run the producer with the HTTP protocol" .

Configure the broker

Add a HTTP transport connector by editing the broker configuration file (in *InstallDir*/conf/activemq.xml) as follows:

```
<beans>
  ...
  <transportConnectors>
    ...
    <transportConnector name="openwire" uri="tcp://localhost:61616"/>
    <transportConnector name="http"   uri="http://localhost:61620"/>
  </transportConnectors>
  ...
</beans>
```

Run the broker

Run the default broker by entering the following at a command line:

```
activemq
```

The default broker automatically takes its configuration from the default configuration file.

Note

The activemq script automatically sets the ACTIVEMQ_HOME and ACTIVEMQ_BASE environment variables to *FuseInstallDir*/fuse-message-broker-*Version* by default. If you want the activemq script to pick up its configuration from a non-default conf directory, you can set ACTIVEMQ_BASE explicitly in your environment. The configuration files will then be taken from $ACTIVEMQ_BASE/conf.

Run the consumer

To connect the consumer tool to the tcp://localhost:61616 endpoint (OpenWire over TCP), change directory to *ActiveMQInstallDir*/example and enter the following command:

```
ant consumer -Durl=tcp://localhost:61616 -Dmax=100
```

You should see some output like the following:

```
Buildfile: build.xml
init:
compile:
consumer:
     [echo] Running consumer against server at $url = tcp://localhost:61616
for subject $subject = TEST.FOO
     [java] Connecting to URL: tcp://localhost:61616
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] We are about to wait until we consume: 100 message(s) then
we will shutdown
```

Run the producer with the TCP protocol

To connect the producer tool to the `tcp://localhost:61616` endpoint (OpenWire over TCP), open a new command prompt, change directory to *`ActiveMQInstallDir`*`/example` and enter the following command:

```
ant producer -Durl=tcp://localhost:61616
```

In the window where the *consumer tool* is running, you should see some output like the following:

```
[java] Received: Message: 0 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 1 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 2 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 3 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 4 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 5 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 6 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 7 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 8 sent at: Wed Sep 19 14:38:06 BST
2007  ...
     [java] Received: Message: 9 sent at: Wed Sep 19 14:38:06 BST
2007  ...
```

Run the producer with the HTTP protocol

To connect the producer tool to the `http://localhost:61620` endpoint (OpenWire over HTTP), enter the following command from the `example` directory:

```
ant producer -Durl=http://localhost:61620
```

This command sends ten new messages to the consumer client.

Note

The JAR files for the HTTP protocol are currently located in the `lib/optional` subdirectory. If you construct the CLASSPATH manually, you must be sure to include the JAR files from this subdirectory.

Chapter 3. Stomp Protocol

The Stomp protocol is a simplified messaging protocol that is specially designed for implementing clients using scripting languages. This chapter provides a brief introduction to the protocol.

Introduction to the Stomp Protocol

Overview

The Stomp protocol is a simplified messaging protocol that is being developed as an open source project (http://stomp.codehaus.org/). The advantage of the stomp protocol is that you can easily improvise a messaging client—even when a specific client API is not available—because the protocol is so simple.

Transport protocols

Table 3.1 on page 28 shows the transport protocols supported by the Stomp wire protocol:

Table 3.1. Transport Protocols Supported by Stomp

Transport Protocol	URL	Description
TCP	`stomp://Host:Port`	Endpoint URL for Stomp over TCP/IP. The broker listens for TCP connections on the host machine, *`Host`*, and IP port, *`Port`*.
SSL	`stomp+ssl://Host:Port`	Endpoint URL for secure Stomp over SSL. The broker listens for TCP connections on the host machine, *`Host`*, and IP port, *`Port`*.

Supported clients

Stomp currently supports the following client types:

- *C clients*.
- *C++ clients*.
- *C# and .NET clients*.
- *.NET clients*.
- *Delphi clients*.
- *Flash clients*.
- *Perl clients*.

- *PHP clients*.
- *Pike clients*.
- *Python clients*.

Protocol Details

Overview

This section describes the format of Stomp data packets , as well as the semantics of the data packet exchanges. Stomp is a relatively simple wire protocol—it is even possible to communicate manually with a Stomp broker using a `telnet` client (see "Stomp Tutorial" on page 38).

Transport protocols

In principal, Stomp can be combined with any transport protocol, including connection-oriented and non-connection-oriented transports. In practice, though, Stomp is usually implemented over TCP and this is the only transport currently supported by Fuse Message Broker.

Licence

The Stomp specification is licensed under the Creative Commons Attribution v2.5[1]

Stomp frame format

The Stomp specification defines the term *frame* to refer to the data packets transmitted over a Stomp connection. A Stomp frame has the following general format:

```
<StompCommand>
<HeaderName_1>:<HeaderValue_1>
<HeaderName_2>:<HeaderValue_2>

<FrameBody>
^@
```

A Stomp frame always starts with a Stomp command (for example, `SEND`) on a line by itself. The Stomp command may then be followed by zero or more header lines: each header is in a `<key>:<value>` format and terminated by a newline. A blank line indicates the end of the headers and the beginning of the body, `<FrameBody>`, (which is empty for many of the commands). The frame is terminated by the null character, which is represented as `^@` above (Ctrl-@ in ASCII)).

Oneway and RPC commands

Most Stomp commands have oneway semantics (that is, after sending a frame, the sender does not expect any reply). The only exceptions are:

[1] http://creativecommons.org/licenses/by/2.5/

- `CONNECT` *command*—after a client sends a `CONNECT` frame, it expects the server to reply with a `CONNECTED` frame.
- *Commands with a* `receipt` *header*—a client can force the server to acknowledge receipt of a command by adding a `receipt` header to the outgoing frame.
- *Erroneous commands*—if a client sends a frame that is malformed, or otherwise in error, the server may reply with an `ERROR` frame. Note, however, that the `ERROR` frame is not formally correlated with the original frame that caused the error (Stomp frames are not required to include a unique identifier).

Receipt header

Any client frame, other than `CONNECT`, may specify a `receipt` header with an arbitrary value. This causes the server to acknowledge receipt of the frame with a `RECEIPT` frame, which contains the value of this header as the value of the `receipt-id` header in the `RECEIPT` frame. For example, the following frame shows a `SEND` command that includes a receipt header:

```
SEND
destination:/queue/a
receipt:message-12345

Hello a!^@
```

Client commands

Table 3.2 on page 31 lists the client commands for the Stomp protocol. The Reply column indicates whether or not the server sends a reply frame by default.

Table 3.2. Client Commands for the Stomp Protocol

Command	Reply?	Role	Description
"CONNECT"	Yes	Producer, Consumer	Open a connection to a Stomp broker (server).
"SEND"	No	Producer	Send a message to a particular queue or topic on the server.
"SUBSCRIBE"	No	Consumer	Subscribe to a particular queue or topic on the server.
"UNSUBSCRIBE"	No	Consumer	Cancel a subscription to a particular queue or topic.

Command	Reply?	Role	Description
"ACK"	No	Consumer	Acknowledge receipt of one message.
"BEGIN"	No	Producer, Consumer	Start a transaction (applies to `SEND` or `ACK` commands).
"COMMIT"	No	Producer, Consumer	Commit a transaction.
"ABORT"	No	Producer, Consumer	Roll back a transaction.
"DISCONNECT"	No	Producer, Consumer	Shut down the existing connection gracefully.

CONNECT

After opening a socket to connect to the remote server, the client sends a `CONNECT` command to initiate a Stomp session. For example, the following frame shows a typical `CONNECT` command, including a `login` header and a `passcode` header:

```
CONNECT
login: <username>
passcode:<passcode>

^@
```

After the client sends the `CONNECT` frame, the server always acknowledges the connection by sending a frame, as follows:

```
CONNECTED
session: <session-id>

^@
```

The `session-id` header is a unique identifier for this session (currently unused).

SEND

The `SEND` command sends a message to a *destination*—for example, a queue or a topic—in the messaging system. It has one required header, `destination`, which indicates where to send the message. The body of the `SEND` command is the message to be sent. For example, the following frame sends a message to the `/queue/a` destination:

```
SEND
destination:/queue/a
```

```
hello queue a

^@
```

From the client's perspective, the destination name, `/queue/a`, is an arbitrary string. Despite seeming to indicate that the destination is a queue it does not, in fact, specify any such thing. Destination names are simply strings that are mapped to some form of destination on the server; how the server translates these strings is up to the implementation.

The `SEND` command also supports the following optional headers:

- `transaction`—specifies the transaction ID. Include this header, if the `SEND` command partakes in a transaction (see "BEGIN" on page 35).
- `content-length`—specifies the byte count for the length of the message body. If a `content-length` header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

SUBSCRIBE

The `SUBSCRIBE` command registers a client's interest in listening to a specific destination. Like the `SEND` command, the `SUBSCRIBE` command requires a `destination` header. Henceforth, any messages received on the subscription are delivered as `MESSAGE` frames, from the server to the client. For example, the following frame shows a client subscribing to the destination, `/queue/a`:

```
SUBSCRIBE
destination: /queue/foo
ack: client

^@
```

In this case the `ack` header is set to `client`, which means that messages are considered delivered only after the client specifically acknowledges them with an `ACK` frame. The body of the `SUBSCRIBE` command is ignored.

The `SUBSCRIBE` command supports the following optional headers:

- `ack`—specify the acknowledgement mode for this subscription. The following modes are recognized:
  - `auto`—messages are considered delivered as soon as the server delivers them to the client (in the form of a `MESSAGE` command). The server does *not* expect to receive any `ACK` commands from the client for this subscription.
  - `client`—messages are considered delivered only after the client specifically acknowledges them with an `ACK` frame.

- `selector`—specifies a SQL 92 selector on the message headers, which acts as a filter for content based routing.
- `id`—specify an ID to identify this subscription. Later, you can use the ID to `UNSUBSCRIBE` from this subscription (you may end up with overlapping subscriptions, if multiple selectors match the same destination).

  When an `id` header is supplied, the server should append a `subscription` header to any `MESSAGE` commands sent to the client. When using wildcards and selectors, this enables clients to figure out which subscription triggered the message.

UNSUBSCRIBE

The `UNSUBSCRIBE` command removes an existing subscription, so that the client no longer receives messages from that destination. It requires either a `destination` header or an `id` header (if the previous `SUBSCRIBE` operation passed an `id` value). For example, the following frame cancels the subscription to the `/queue/a` destination:

```
UNSUBSCRIBE
destination: /queue/a

^@
```

ACK

The `ACK` command acknowledges the consumption of a message from a subscription. If the client issued a `SUBSCRIBE` frame with an `ack` header set to `client`, any messages received from that destination are not considered to have been consumed until the message is acknowledged by an `ACK` frame.

The `ACK` command has one required header, `message-id`, which must contain a value matching the `message-id` for the `MESSAGE` being acknowledged. Optionally, a `transaction` header may be included, if the acknowledgment participates in a transaction. For example, the following frame acknowledges a message in the context of a transaction:

```
ACK
message-id: <message-identifier>
transaction: <transaction-identifier>

^@
```

BEGIN

The `BEGIN` command initiates a transaction. Transactions can be applied to `SEND` and `ACK` commands. Any messages sent or acknowledged during a transaction can either be commited or rolled back at the end of the transaction.

```
BEGIN
transaction: <transaction-identifier>

^@
```

The `transaction` header is required and the `<transaction-identifier>` can be included in `SEND`, `COMMIT`, `ABORT`, and `ACK` frames to bind them to the named transaction.

COMMIT

The `COMMIT` command commits a specific transaction.

```
COMMIT
transaction: <transaction-identifier>

^@
```

The `transaction` header is required and specifies the transaction, `<transaction-identifier>`, to commit.

ABORT

The `ABORT` command rolls back a specific transaction.

```
ABORT
transaction: <transaction-identifier>

^@
```

The `transaction` header is required and specifies the transaction, `<transaction-identifier>`, to roll back.

DISCONNECT

The `DISCONNECT` command disconnects gracefully from the server.

```
DISCONNECT

^@
```

Server commands

Table 3.3 on page 36 lists the commands that the server can send to a Stomp client. These commands all have oneway semantics.

Table 3.3. Server Commands for the Stomp Protocol

Command	Description
"MESSAGE"	Send a message to the client, where the client has previously registered a subscription with the server.
"RECEIPT"	Acknowledges receipt of a client command, if the client requested a receipt by included a `receipt-id` header.
"ERROR"	Error message sent from the server to the client.

MESSAGE

The `MESSAGE` command conveys messages from a subscription to the client. The `MESSAGE` frame must include a `destination` header, which identifies the destination from which the message is taken. The `MESSAGE` frame also contains a `message-id` header with a unique message identifier. The frame body contains the message contents. For example, the following frame shows a typical `MESSAGE` command with `destination` and `message-id` headers:

```
MESSAGE
destination:/queue/a
message-id: <message-identifier>

hello queue a^@
```

The `MESSAGE` command supports the following optional headers:

- `content-length`—specifies the byte count for the length of the message body. If a `content-length` header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

RECEIPT

A `RECEIPT` frame is issued from the server whenever the client requests a receipt for a given command. The `RECEIPT` frame includes a `receipt-id`, containing the value of the `receipt-id` from the original client request. For example, the following frame shows a typical `RECEIPT` command with `receipt-id` header:

```
RECEIPT
receipt-id:message-12345

^@
```

The receipt body is always empty.

ERROR

The server may send `ERROR` frames if something goes wrong. The error frame should contain a `message` header with a short description of the error. The body may contain more detailed information (or may be empty). For example, the following frame shows an `ERROR` command with a non-empty body:

```
ERROR
message: malformed packet received

The message:
-----
MESSAGE
destined:/queue/a
Hello queue a!
-----
Did not contain a destination header, which is required for message
propagation.
^@
```

The `ERROR` command supports the following optional headers:

- `content-length`—specifies the byte count for the length of the message body. If a `content-length` header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

Stomp Tutorial

Telnet client

Because Stomp frames consist of plain text, it is possible to improvise a Stomp client by starting up a `telnet` session and entering Stomp frames directly at the keyboard. This can be a useful diagnostic tool and is also a good way to learn about the Stomp protocol.

Typing the null character

While most characters in a Stomp frame are just plain text, there is one required character, null, that you might have difficulty typing at the keyboard. On some keyboards, you can type null as Ctrl-@. Other keyboards might require you to do a bit of research, however.

For example, to type a null character on the 101-key keyboard that is commonly used with a Windows PC, proceed as follows:

1. Enable **NumLock** on the numeric keypad (this is essential).
2. While holding down the **Alt** key, type zero, `0`, four times in succession *on the numeric keypad*.

Tutorial steps

To send and receive messages over the Stomp protocol using telnet clients, perform the following steps:

1. "Start the broker" .
2. "Start a telnet session for the producer" .
3. "Start a Stomp session for the producer" .
4. "Send a message to a queue" .
5. "Check the queue status using JMX" .
6. "Start a telnet session for the consumer" .
7. "Start a Stomp session for the consumer" .
8. "Subscribe to a queue" .
9. "Acknowledge a message" .
10. "Unsubscribe from the queue" .

11. "Disconnect the clients" .

Start the broker

Start the default broker by entering the following at a command prompt:

```
activemq
```

Normally, the default broker is configured to initialize a Stomp connector that listens on port, 61613. Look for a line like the following in the broker's log:

```
INFO  TransportServerThreadSupport   - Listening for connections at:
stomp://localhost:61613
```

If the Stomp connector is not present in the broker, you will have to configure it—see "Configure the broker" on page 46 for details.

Start a telnet session for the producer

Open a new command prompt and start a new `telnet` session for the producer client, by entering the following command:

```
telnet
```

This command starts `telnet` in interactive mode. Now enter the following `telnet` commands (the `telnet` prompt that begins each line is implementation dependent):

```
telnet> set localecho
Local echo on
telnet> open localhost 61613
```

After entering the `open` command, `telnet` should connect to the Stomp socket on your local ActiveMQ broker (where the Stomp port is presumed to be 61613 here). You should now see a blank screen, where you can directly type the contents of the Stomp frames you want to send over TCP.

Start a Stomp session for the producer

Start a Stomp session for the producer by entering the following Stomp frame in the `telnet` window:

```
CONNECT
login:foo
passcode:bar

^@
```

The `login` and `passcode` headers are currently ignored by the ActiveMQ broker, so you can enter any values you like for these headers. *Don't forget to insert a blank line after the headers.* Finally, you must terminate the frame by typing the null character, `^@` (for notes on how to type the null character at your keyboard, see "Typing the null character" on page 38).

If all goes well, you will see a response similar to the following:

```
CONNECTED
session:ID:fboltond820-2290-1190810591249-3:0
```

Send a message to a queue

Send a message to the `FOO.BAR` queue by entering the following frame:

```
SEND
destination:/queue/FOO.BAR
receipt:

Hello, queue FOO.BAR
^@
```

As soon as you have finished typing the null character, `^@`, you should receive the following `RECEIPT` frame from the server:

```
RECEIPT
receipt-id:
```

It is a good idea to include a `receipt` header in the frames you send from a telnet client. It enables you to confirm that the connection is working normally.

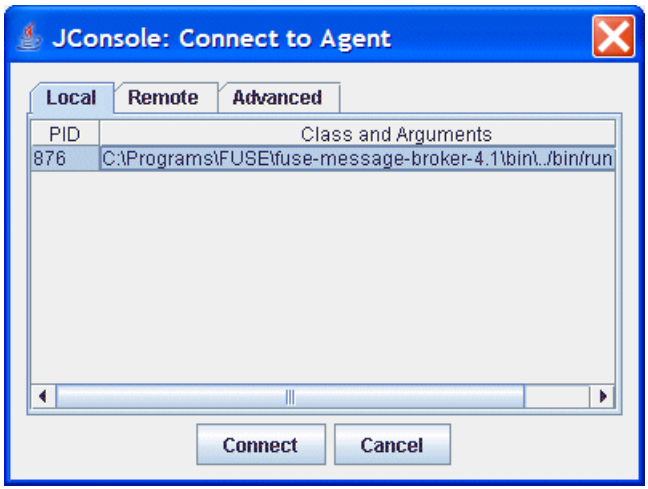
Check the queue status using JMX

The status of the ActiveMQ broker can be monitored through a JMX port. To monitor the broker, start a new command prompt and enter the following command:

```
jconsole
```

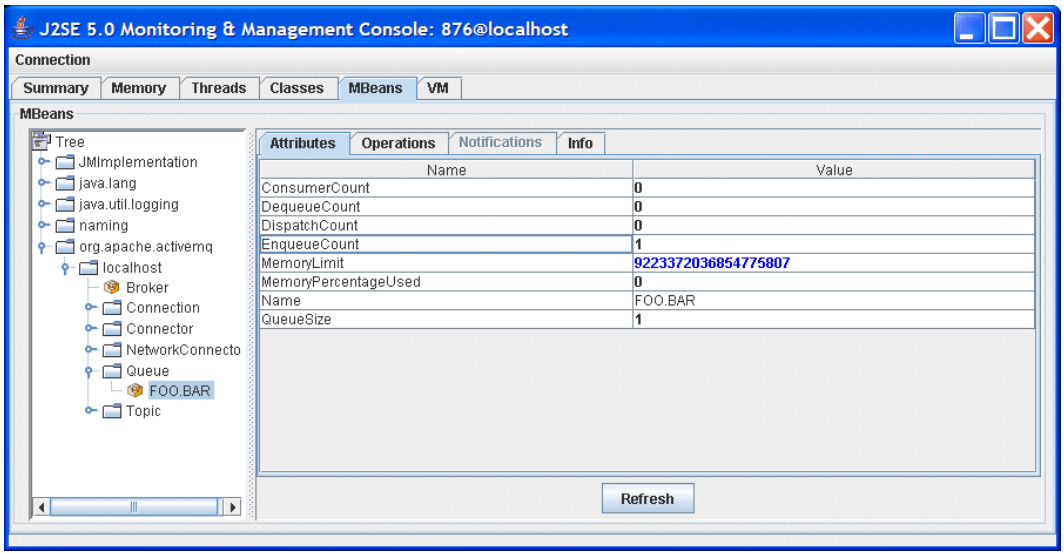
The `jconsole` utility is a standard JMX client that is included with Sun's Java Development Kit (JDK). When you start the `jconsole` utility, a dialog appears and prompts you to connect to a JMX process, as shown in Figure 3.1 on page 41.

Figure 3.1. Connecting to the ActiveMQ JMX Port



Select the ActiveMQ broker process and click **Connect**. The main `jconsole` window opens. To view the current status of the `FOO.BAR` message queue, click on the **MBeans** tab and use the tree on the left hand side to drill down to `org.apache.activemq/localhost/Queue/FOO.BAR`. Click on the `FOO.BAR` icon to view the current status, as shown in Figure 3.2 on page 42.

Figure 3.2. Monitoring the Status of the FOO.BAR Queue



The status shows an **EnqueueCount** of 1, which tells you that the producer has successfully enqueued one message in the `FOO.BAR` queue.

Start a telnet session for the consumer

Open a new command prompt and start a new `telnet` session for the consumer client, by entering the following command:

```
telnet
```

Enter the following `telnet` commands to connect to the Stomp socket on the broker:

```
telnet> set localecho
Local echo on
telnet> open localhost 61613
```

Start a Stomp session for the consumer

Start a Stomp session for the consumer by entering the following Stomp frame in the consumer's `telnet` window:

```
CONNECT
login:foo
```

```
passcode:bar

^@
```

If all goes well, you will see a response similar to the following:

```
CONNECTED
session:ID:fboltond820-2290-1190810591249-3:1
```

Subscribe to a queue

Subscribe to the `FOO.BAR` queue by entering the following Stomp frame in the consumer's `telnet` window:

```
SUBSCRIBE
destination:/queue/FOO.BAR
ack:client

^@
```

The `ack` header is set to the value `client`, which implies that the consumer client is expected to acknowledge each message it receives from the broker. After typing the terminating null character, `^@`, the broker dispatches the sole message on the `FOO.BAR` queue by sending a `MESSAGE` frame, as follows:

```
MESSAGE
destination:/queue/FOO.BAR
receipt:
timestamp:1190811984837
priority:0
expires:0
message-id:ID:fboltond820-2290-1190810591249-3:0:-1:1:1

Hello, queue FOO.BAR
```

To see what effect this has on the queue status, go to the `jconsole` window and click **Refresh** on the MBeans tab. The **DispatchCount** attribute is now equal to 1, indicating that the broker has dispatched the message to the consumer. The **DequeueCount** is equal to 0, however; this is because the message is not considered to be dequeued until the consumer client sends an acknowledgement.

Acknowledge a message

Acknowledge the received message by entering the following Stomp frame in the consumer's `telnet` window:

```
ACK
message-id:ID:fboltond820-2290-1190810591249-3:0:-1:1:1

^@
```

Where the message ID must match the value from the `message-id` header in the received `MESSAGE` frame. To check that the acknowledgement has been effective, go back to the `jconsole` window and click **Refresh** on the **MBeans** tab. You should now find that the **DequeueCount** has increased to 1.

Unsubscribe from the queue

Unsubscribe from the `FOO.BAR` queue by entering the following Stomp frame in the consumer's `telnet` window:

```
UNSUBSCRIBE
destination:/queue/FOO.BAR
receipt:

^@
```

Disconnect the clients

To shut down both the producer and consumer gracefully, enter the following `DISCONNECT` frame in each of their respective `telnet` windows:

```
DISCONNECT

^@
```

Stomp Example

Overview

Fuse Message Broker provides some sample code in *ActiveMQInstallDir*/example/ruby that enables you to experiment with the Stomp protocol in the Ruby programming language.

Example prerequisites

You must download and install the requisite packages to support the Ruby programming language before you can run the Stomp example. Install the following packages:

- *Ruby programming language*—download and install the Ruby programming language from http://www.ruby-lang.org/en/downloads. Add the Ruby /bin directory to your PATH.
- *RubyGems package manager*—RubyGems (http://www.rubygems.org) is a utility for installing and managing add-ons to the Ruby language. Download and install RubyGems as follows:

  1. Download a RubyGems archive file (.tgz, .zip, or .gem) from the RubyForge (http://rubyforge.org/frs/?group_id=126).
  2. Unzip the RubyGems archive.
  3. Initialize RubyGems by entering the following command:

     ```
     ruby GemsInstallDir/setup.rb
     ```

  4. Add *GemsInstallDir*/bin to your PATH.

- *Stomp package for Ruby*—install the Stomp package for Ruby by running the following command:

  ```
  gem install stomp
  ```

  RubyGems downloads and installs the requisite package to support the Ruby Stomp client API.

Example steps

To try out the Stomp protocol, perform the following steps:

1. "Configure the broker" .
2. "Run the broker" .

3. "Run the Ruby listener" .

4. "Run the Ruby publisher" .

Configure the broker

Check that the the Stomp connector is present in the broker configuration file (in *InstallDir*/conf/activemq.xml) as follows:

```
<beans>
  ...
  <transportConnectors>
    ...
    <transportConnector name="stomp"   uri="stomp://localhost:61613"/>
  </transportConnectors>
  ...
</beans>
```

Run the broker

Run the default broker by entering the following at a command line:

```
activemq
```

The default broker automatically takes its configuration from the default configuration file.

Note

The `activemq` script automatically sets the `ACTIVEMQ_HOME` and `ACTIVEMQ_BASE` environment variables to *FuseInstallDir*/fuse-message-broker-*Version* by default. If you want the `activemq` script to pick up its configuration from a non-default `conf` directory, you can set `ACTIVEMQ_BASE` explicitly in your environment. The configuration files will then be taken from `$ACTIVEMQ_BASE/conf`.

Run the Ruby listener

To connect the listener tool to the `stomp://localhost:61613` endpoint (Stomp over TCP), change directory to *ActiveMQInstallDir*/example/ruby and enter the following command:

```
ruby listener.rb
```

They Ruby listener connects to the endpoint, `stomp://localhost:61613`, by default. You could change this endpoint address by editing the `listener.rb` script.

Run the Ruby publisher

To connect the publisher tool to the `stomp://localhost:61613` endpoint (Stomp over TCP), change directory to *`ActiveMQInstallDir`*`/example/ruby` and enter the following command:

```
ruby publisher.rb
```

You should see some output like the following:

```
Sent 1000 messages
Sent 2000 messages
Sent 3000 messages
Sent 4000 messages
Sent 5000 messages
Sent 6000 messages
Sent 7000 messages
Sent 8000 messages
Sent 9000 messages
Sent 10000 messages
Received report: Received 10000 in 4.567 seconds, remaining: 9
```

Chapter 4. REST Protocols

The REST protocol is a simple HTTP-based protocol that enables you to interact with the message broker using HTML forms and DHTML scripts. This chapter provides a brief introduction to the protocol, illustrating how to contact the message broker from a Web browser.

Introduction to the REST Protocol

Overview

The REST protocol is a simple HTTP-based protocol that enables you to contact the message broker through a Web browser. You can contact the message broker by navigating to appropriately formatted URLs or by posting HTML forms.

Transport protocols

The Fuse Message Broker's REST protocol is based on a subset of the HTTP protocol. Hence, HTTP is the only supported transport.

Supported clients

REST supports the following client types:

- *Web forms*—use conventional HTML forms to `POST` a message to a destination (queue or topic) or to `GET` a message from a destination—see "Example of posting a message" on page 54 .
- *Ajax clients*—an Asynchronous JavaScript And Xml (Ajax) library that enables you to communicate with a REST endpoint using JavaScript in a DHTML Web page.

REST servlets

The REST protocol is implemented by the following servlets running in a Web container:

- `message` *servlet*—supports the sending and consuming of messages.
- `queueBrowse` *servlet*—enables you to view the current status of a particular queue.

Protocol Details

What is REST?

Representational State Transfer (REST) is a software architecture designed for distributed systems, like the World Wide Web. For details of the REST architecture and the philosophy underlying it, see the REST Wikipedia[1] article.

One of the key concepts of a RESTful architecture is that the interaction between different network nodes should take on a very simple form. In particular, the number of operations in a RESTful protocol must be kept small: for example, the REST protocol in Fuse requires just three operations.

Outline of a REST interaction

In general, a REST interaction consists of the following elements:

- *Operation*—belongs to a restricted, well-known set of operations—for example, in the HTTP protocol, the main operations are `GET`, `POST`, `PUT`, and `DELETE`. The advantage of this approach is that, in contrast to RPC architectures, there is no need to define interfaces for a RESTful protocol. The operations are all known in advance.
- *URI*—identifies the resource that the operation acts on. For example, a HTTP `GET` operation acts on the URI by fetching data from the resource identified by the URI.
- *Data (if required)*—needed for operations that send data to the remote resource.

HTTP as a RESTful protocol

HTTP is a good example of a protocol demonstrating RESTful design principles. In fact, proponents of REST argue that it is precisely the RESTful qualities of HTTP that enabled the rapid expansion of the World Wide Web. In keeping with REST principles, HTTP has a restricted operation set, consisting of only eight operations: `GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `HEAD`, `TRACE`, and `CONNECT`.

For the purpose of implementing a RESTful protocol, the first four HTTP operations—`GET`, `POST`, `PUT`, and `DELETE`—are the most important. The semantics of these operations are described briefly in Table 4.1 on page 51.

Table 4.1. HTTP RESTful Operations

Operation	Description
`GET`	Fetch the remote resource identified by the URI.

[1] http://en.wikipedia.org/wiki/Representational_State_Transfer#REST.27s_Central_Principle:_Resources

Operation	Description
`POST`	Add/append/insert data to the remote resource identified by the URI.
`PUT`	Replace the remote resource identified by the URI with the data from this operation.
`DELETE`	Delete the remote resource identified by the URI.

This simple set of operations—analogous to the classic CRUD (Create, Replace, Update, and Delete) operations for a database—turns out to be remarkably powerful and flexible.

REST protocol servlets

The following servlets—which are automatically deployed in the message broker Web console—implement RESTful access to the Fuse message queues:

- "message servlet"
- "queueBrowse servlet"

message servlet

The RESTful Web service implemented by the Fuse `message` servlet enables you to enqueue and dequeue messages over HTTP. You can, therefore, use the message servlet to implement message producers and message consumers as Web forms.

To interact with the Fuse `message` servlet, construct a URL of the following form:

```
http://Host:Port/WebContext/message/DestinationPath?Opt1=Val1&Opt2=Val2...
```

Where the URL is constructed from the following parts:

- *`Host:Port`*—the host and port of the servlet engine. For example, in the default message broker configuration, a HTTP port is opened on `localhost:8161`.
- *`WebContext`*—in a Web application, it is usual to group related components (servlets and so on) under a particular Web context, *`WebContext`*. For example, for the REST demonstration servlets, the Web context is `demo` by default.
- `message`—routes this URL to the `message` servlet.
- *`DestinationPath`*—specifies the compound name of a queue or topic in the message broker. For example, the `FOO.BAR` queue has the destination path, `FOO/BAR`.

- *?Opt1=Val1&Opt2=Val2*—you can add some options in order to qualify how the URL is processed.

For example, the following URL can be used to fetch a message from the FOO.BAR queue, where the Web console has the default configuration:

```
http://localhost:8161/demo/message/FOO/BAR?type=queue&timeout=5000
```

Table 4.2 on page 53 shows the URL options recognized by the message servlet:

Table 4.2. URL Options Recognized by the Message Servlet

URL Option	Description
type	Can be either queue or topic.
timeout	When consuming a message from a queue, specifies the length of time (in units of milliseconds) the client is prepared to wait.

Three HTTP operations—GET, POST, and DELETE—are recognized by the message servlet. The semantics of these operations are described briefly in Table 4.3 on page 53 .

Table 4.3. Message Servlet RESTful HTTP Operations

Operation	Description
GET	Consume a single message from the destination (queue or topic) specified by the URL.
POST	Send a single message to the destination (queue or topic) specified by the URL.
DELETE	Consume a single message from the destination (queue or topic) specified by the URL. This operation has the same effect as GET.

For details of the form properties recognized by the message servlet (for POSTing a message), see "Example of posting a message" on page 54 .

queueBrowse servlet

The RESTful Web service implemented by the queueBrowse servlet enables you to monitor the contents and status of any queue or topic in the Web console. Effectively, the queueBrowse servlet is a simple management tool.

To interact with the Fuse queueBrowse servlet, construct a URL of the following form:

```
http://Host:Port/WebContext/queueBrowse/DestinationPath?Opt1=Val1&Opt2=Val2...
```

The `queueBrowse` URL has a similar structure to the `message` URL (see "message servlet" on page 52), except that the `queueBrowse` URL is built from *`WebContext`*`/queueBrowse` instead of *`WebContext`*`/message`.

For example, the following URL can be used to browse the `FOO.BAR` queue, where the Web console has the default configuration:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR
```

Table 4.4 on page 54 shows the URL options recognized by the `queueBrowse` servlet:

Table 4.4. URL Options Recognized by the QueueBrowse Servlet

URL Option	Description
`view`	Specifies the format for viewing the queue/topic. The following views are supported: • `simple`—(*default*) displays a compact summary of the queue in XML format, where each message is shown as a `message` element with ID. • `xml`—displays a detailed summary of the queue in XML format, where each message is shown in full. • `rss`—displays a compact summary of the queue in the form of an RSS 1.0, 2.0 or Atom 0.3 feed. You can configure the type of feed using `feedType`.
`feedType`	In combination with the setting, `view=rss`, you can use this option to specify one of the following feeds: • `rss_1.0` • `rss_2.0` • `atom_0.3`
`contentType`	Override the MIME content type of the view.
`maxMessages`	The maximum number of messages to render.

Example of posting a message

Example 4.1 on page 55 shows an example of the Web form used to send a message to the `FOO.BAR` queue in the Web console, as demonstrated in "Send a message" on page 60.

Example 4.1. Web Form for Sending a Message to a Queue or Topic

```
<html>
<head>
  <title>Send a JMS Message</title>
  <link rel="stylesheet" href="style.css" type="text/css">
</head>
<body>
<h1>Send a JMS Message</h1>
<form action="message/FOO/BAR" method="post">
  <p>
    <label for="destination">Destination name</label>
    <input type="text" name="destination"/>
  </p>
  <p>
    <label for="type">Destination Type: </label>
    <select name="type">
      <option selected value="queue">Queue</option>
      <option type" value="topic">Topic</option>
   </select>
  </p>
  <p>
    <textarea name="body" rows="30" cols="80">
Enter some text here for the message body...
    </textarea>
  </p>
  <p>
    <input type="submit" value="Send"/>
    <input type="reset"/>
  </p>
</form>
</body>
</html>
```

Table 4.5 on page 55 describes the form properties that are recognized by the message servlet.

Table 4.5. Form Properties Recognized by Message Servlet

Form Property	Description
Form action	The `action` attribute of the `<form>` tag has the format, `message/`*`DestinationPath`*, where *`DestinationPath`* is the compound name of the queue or topic, using forward slash, `/`, as the delimiter (for example, `FOO/BAR`).
`destination`	The compound name of the destination queue or topic, using a period, `.`, as the delimiter (for example,

Form Property	Description
	FOO.BAR). If this property is specified in the form, it overrides the value of the *DestinationPath* in the form action.
type	Destination type, equals queue or topic.
body	Message body.

Example of getting a message

To consume a message from a topic or queue, send a HTTP GET operation (for example, by following a hypertext link) using the URL format described in "message servlet" on page 52 . For example, to consume a message from the FOO.BAR queue, navigate to the following URL:

```
http://localhost:8161/demo/message/FOO/BAR?timeout=10000&type=queue
```

Examples of browsing a queue

To browse a queue using the queueBrowse servlet, simply navigate to an URL of the appropriate form, as described in "queueBrowse servlet" on page 53 .

For example, to browse the FOO.BAR queue in XML format:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=xml
```

To browse the FOO.BAR queue as an Atom 1.0 feed:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=rss&feedType=atom_1.0
```

To browse the FOO.BAR queue as an RSS 1.0 feed:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=rss&feedType=rss_1.0
```

REST Example

Overview

This section describes how to run the REST example, which consists of a servlet engine integral to the message broker binary, and some demonstration servlets that run as a Web application. To connect to the Web applications, you can use your favorite Web browser.

Example prerequisites

You must ensure that the message broker is configured to instantiate an embedded servlet engine. In your broker configuration file, `conf/activemq.xml`, check that there is a `jetty` element configured as shown in Example 4.2 on page 57 .

Example 4.2. Configuration of an Embedded Servlet Engine

```
<!-- Embedded servlet engine for serving up the Admin console
-->
<jetty xmlns="http://mortbay.com/schemas/jetty/1.0">
  <connectors>
    <nioConnector port="8161" />
  </connectors>
  <handlers>
    <webAppContext contextPath="/admin"
      resourceBase="${activemq.base}/webapps/admin"
      logUrlOnStart="true" />
    <webAppContext contextPath="/demo"
      resourceBase="${activemq.base}/webapps/demo"
      logUrlOnStart="true" />
  </handlers>
</jetty>
```

With the configuration shown in Example 4.2 on page 57 , the servlet engine opens up a HTTP port on IP port, 8161. The following Web applications are loaded:

- Demonstration application (from `webapps/demo`),
- REST protocol servlets (from `webapps/demo`).
- Web console servlet (from `webapps/admin`),

Example steps

To run the REST Web example, perform the following steps:

1. "Run the servlet engine" .
2. "Open a Web browser" .
3. "Send a message" .
4. "Browse the message queue" .
5. "Receive a message from the queue" .

Run the servlet engine

To run the embedded servlet engine, open a new command window and enter the following command to start the default message broker:

```
activemq
```

This step assumes that your broker is configured as described in "Example prerequisites" on page 57.

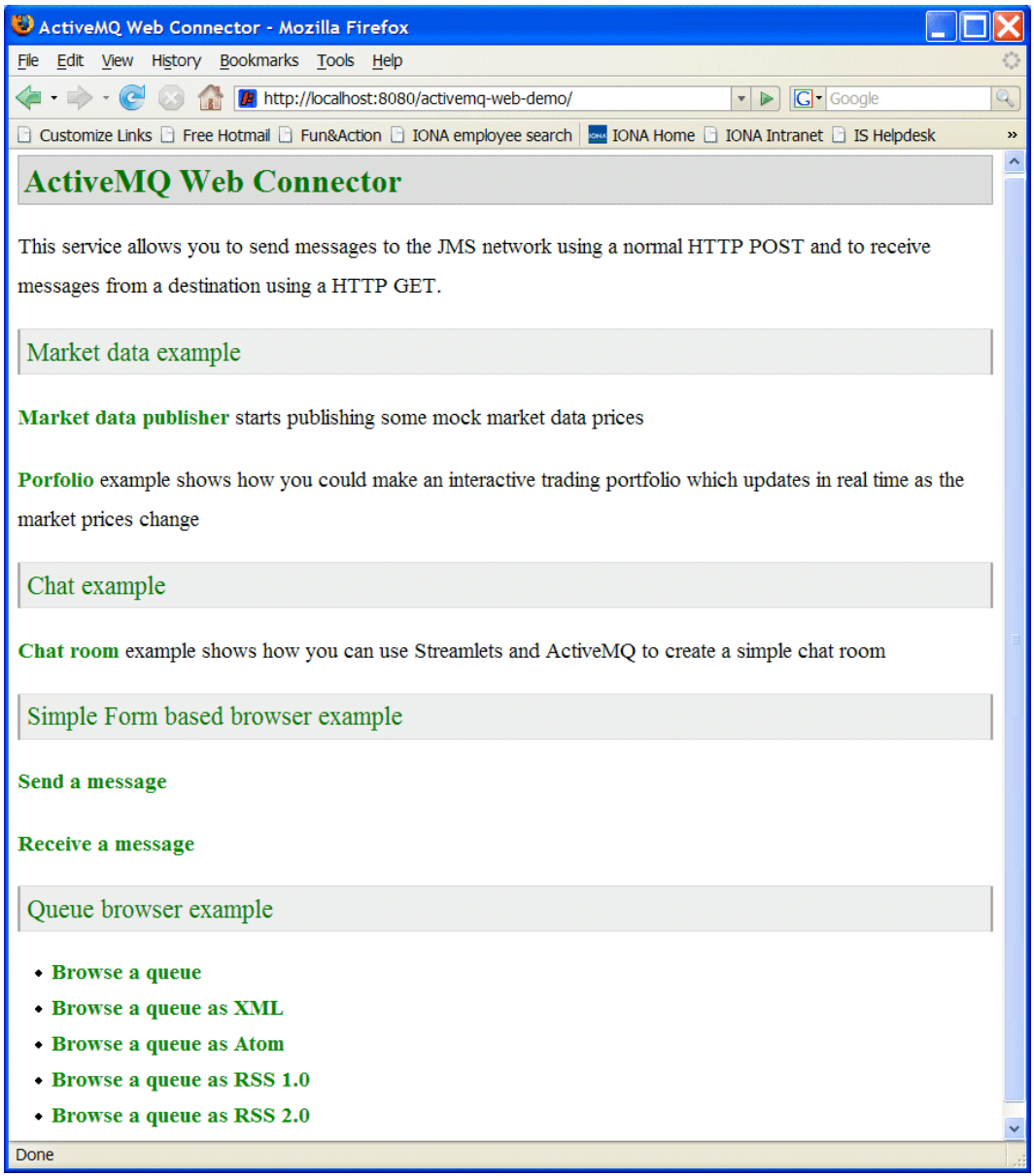
Open a Web browser

Open your favorite Web browser (for example, Firefox or Internet Explorer) and navigate to the following URL:

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

Your browser should now show the welcome page for the Web examples, as shown in Figure 4.1 on page 59.

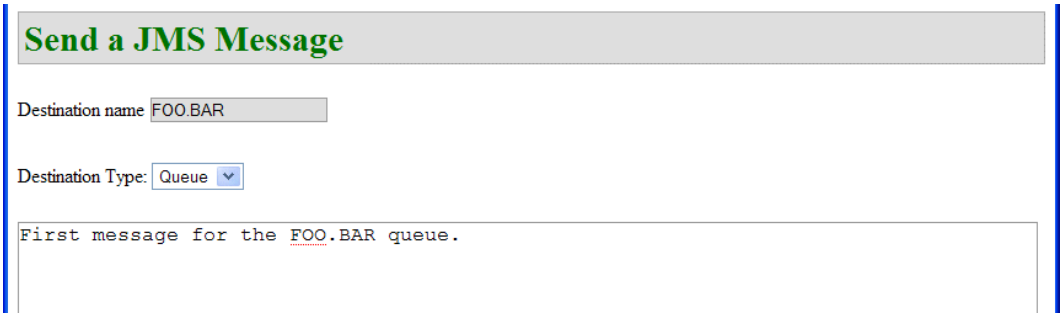
Figure 4.1. Welcome Page for Web Examples



Send a message

To view the form for publishing messages, click the link, Send a message[2]. The **Send a JMS Message** form now appears in your browser, as shown in Figure 4.2 on page 60.

Figure 4.2. The Send a JMS Message Form



In the **Destination name** text field, enter `FOO.BAR` to send a message to the `FOO.BAR` queue. Leave the **Destination Type** as `Queue`. Then enter an arbitrary text message in the large message text box. Click the **Send** button at the bottom of the form to send the message.

Browse the message queue

Using the history feature of your browser, navigate back to the example welcome page (see Figure 4.1 on page 59). The `queueBrowse` servlet supports a variety of ways to browse the contents of a queue and these are listed at the bottom of the welcome page. The following browsing options are listed:

- Browse a queue.[3]
- Browse a queue as XML[4].
- Browse a queue as Atom[5].
- Browse a queue as RSS 1.0[6].
- Browse a queue as RSS 2.0[7].

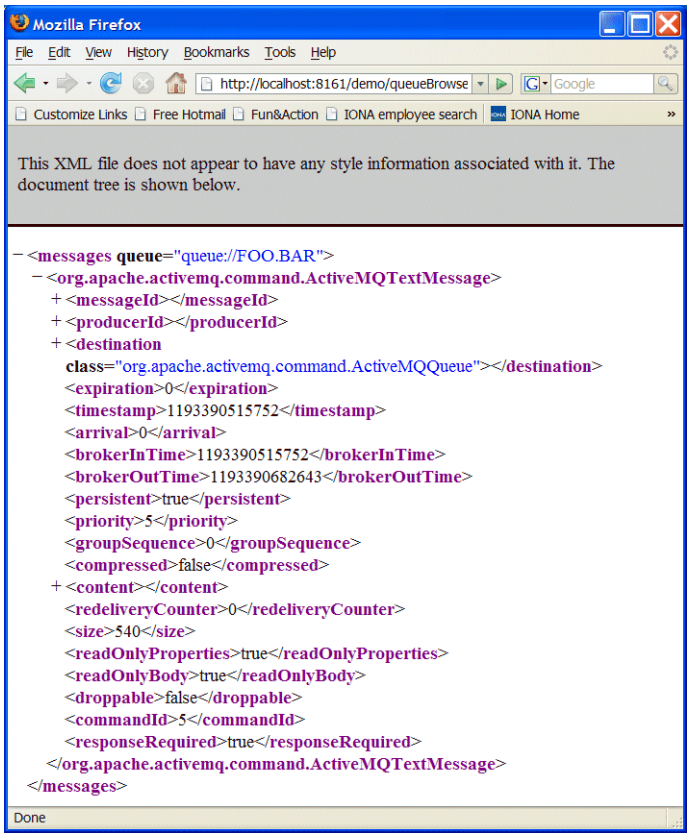
[2] http://localhost:8080/activemq-web-demo/send.html
[3] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR
[4] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=xml
[5] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=atom_1.0
[6] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=rss_1.0
[7] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=rss_2.0

If you click on **Browse a queue**, you should see a page like Figure 4.3 on page 61 .

Figure 4.3. Default Option to Browse a Queue

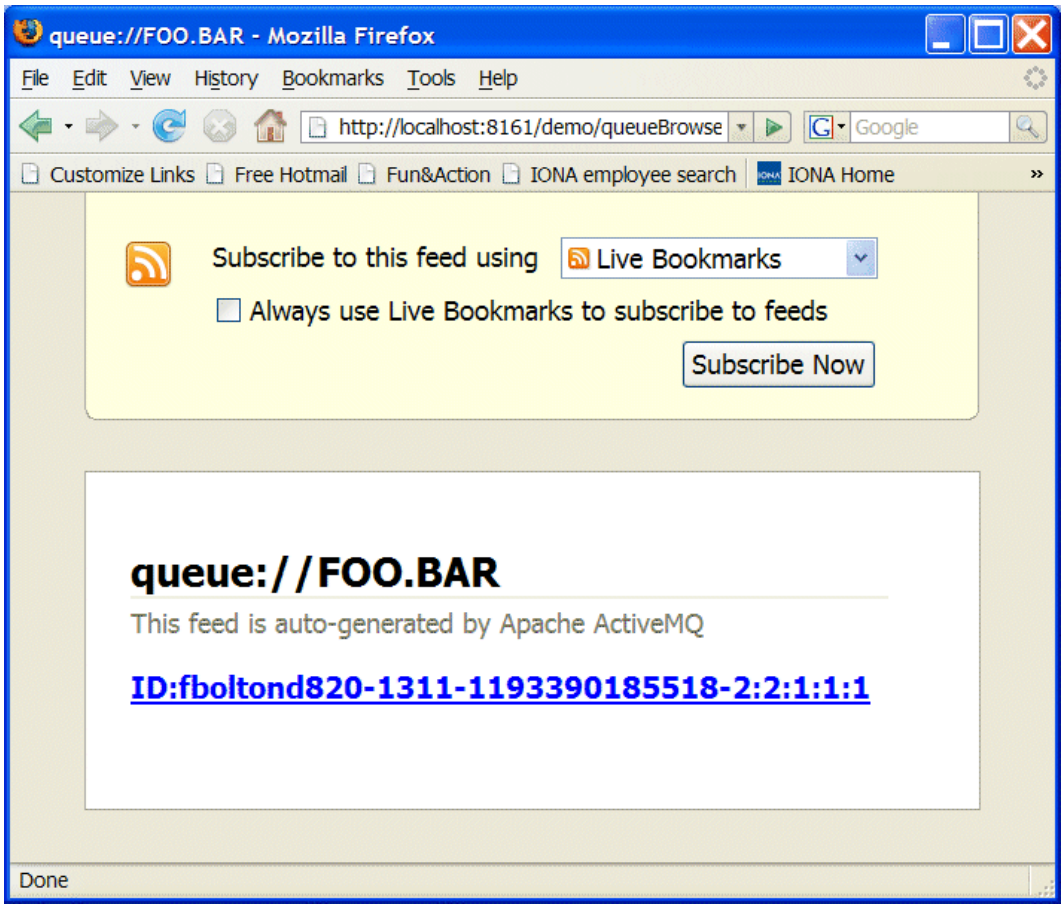
If you click on **Browse a queue as XML**, you should see a page like Figure 4.4 on page 62 .

Figure 4.4. Option to Browse a Queue as XML



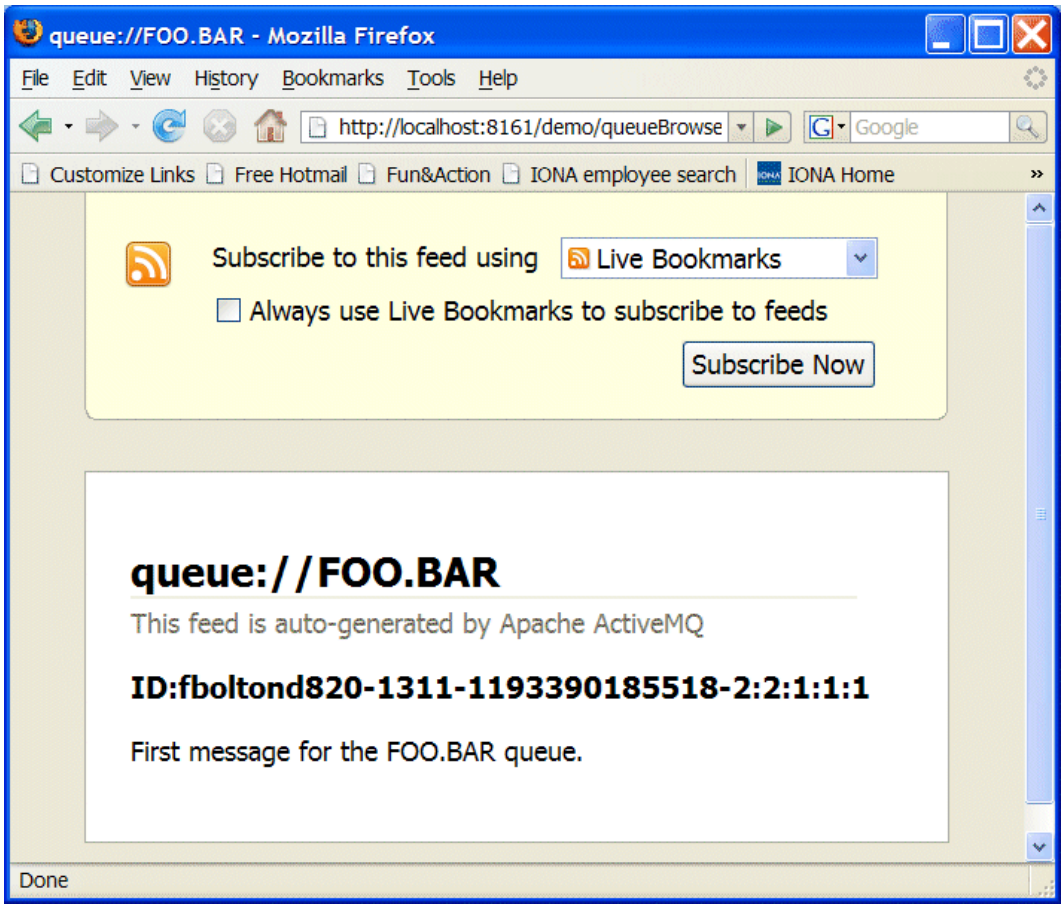
If you click on **Browse a queue as Atom**, you should see a page like Figure 4.5 on page 63 .

Figure 4.5. Option to Browse a Queue as Atom



If you click on **Browse a queue as RSS 1.0** or **Browse a queue as RSS 2.0**, you should see a page like Figure 4.6 on page 64 .

Figure 4.6. Option to Browse a Queue as RSS 1.0



Receive a message from the queue

To receive a message from the `FOO.BAR` queue, open the example welcome page in your browser, http://localhost:8161/demo[8], and click the link, Receive a message[9].

[8] http://localhost:8080/activemq-web-demo
[9] http://localhost:8080/activemq-web-demo/message/FOO/BAR?timeout=10000&type=queue

You should now see the text of the message that you sent earlier. You will probably also receive an error from your browser, if the message is not formatted as HTML or XML (which the browser expects).

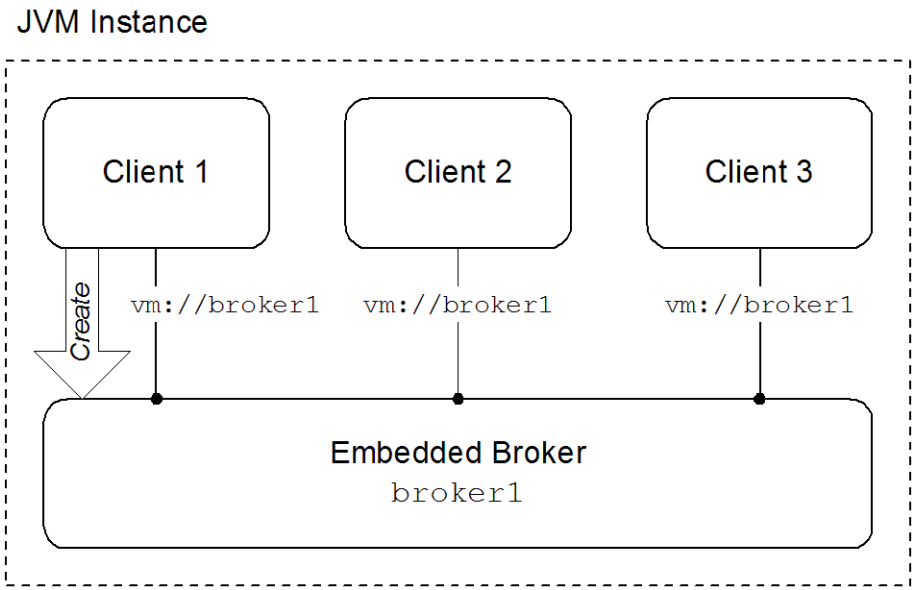
Chapter 5. VM Transport

The VM transport allows clients to connect to each other inside the Java Virtual Machine (JVM) without the overhead of network communication.

Overview

The VM transport enables Java clients running inside the same JVM to communicate with each other inside the JVM, without having to resort to a using a network connection. Because the clients require a broker to communicate, the VM transport can implicitly create an embedded broker the first time it is accessed. Figure 5.1 on page 67 shows the basic architecture of the VM protocol.

Figure 5.1. Clients Connected through the VM Transport



The URI used to specify the VM transport comes in two flavors to provide maximum control over how the embedded broker is configured:

- simple—specifies the name of the embedded broker to which the client connects and allows for some basic broker configuration
- advanced—uses a broker URI to configure the embedded broker

Embedded brokers

The VM transport uses a broker embedded in the same JVM as the clients to facilitate communication between the clients. The embedded broker can be created in several ways:

- explicitly defining the broker in the application's configuration
- explicitly creating the broker using the Java APIs
- automatically when the first client attempts to connect to it using the VM transport

The VM transport uses the broker name specified in either the simple URI or in the configuration provided by the advanced URI's broker URI to determine if an embedded broker needs to be created. When a client uses the VM transport to connect to a broker, the transport checks to see if an embedded broker by that name already exists. If it does exist, the client is connected to the broker. If it does not exist, the broker is created and then the broker is connected to it.

When using explicitly created brokers there is a danger that your clients will attempt to connect to the embedded broker before it is started. If this happens, the VM transport will auto-create an instance of the broker for you. To avoid this conflict you can set the `waitForStart` option or the `create=false` option to manage how the VM transport determines when to create a new embedded broker.

Simple URI syntax

The simple VM URI is used in most situations. It allows you to specify the name of the embedded broker to which the client will connect. It also allows for some basic broker configuration.

Example 5.1 on page 68 shows the syntax for a simple VM URI.

Example 5.1. Simple VM URI Syntax

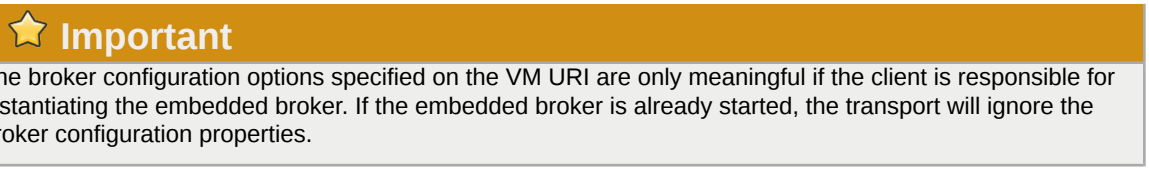
```
vm://BrokerName?TransportOptions
```

- *`BrokerName`* specifies the name of the embedded broker to which the client connects.
- *`TransportOptions`* specifies the configuration for the transport. They are specified in the form of a query list. Table 5.2 on page 70 lists the available options.

In addition to the transport options listed in Table 5.2 on page 70, the simple VM URI can use the options described in Table 5.1 on page 69 to configure the embedded broker.

Table 5.1. VM Transport Broker Configuration Options

Option	Description
`broker.*`	Properties prefixed by `broker.` configure the embedded broker. You can specify any of the standard broker options described in Table 5.3 on page 70.
`brokerConfig`	Specifies an external broker configuration file. For example, to pick up the broker configuration file, `activemq.xml`, you would set `brokerConfig` as follows: `brokerConfig=xbean:activemq.xml`.



Example 5.2 on page 69 shows a basic VM URI that connects to an embedded broker named `broker1`.

Example 5.2. Basic VM URI

```
vm://broker1
```

Example 5.3 on page 69 creates and connects to an embedded broker that uses a non-persistent message store.

Example 5.3. Simple URI with broker options

```
vm://broker1?broker.persistent=false
```

Advanced URI syntax

The advance VM URI provides you full control over how the embedded broker is configured. It uses a broker configuration URI similar to the one used by the administration tool to configure the embedded broker.

Example 5.4 on page 69 shows the syntax for a simple VM URI.

Example 5.4. Advanced VM URI Syntax

```
vm://(BrokerConfigURI)?TransportOptions
```

- *`BrokerConfigURI`* is a broker configuration URI. See "Broker URI" in *Managing and Monitoring a Broker* for information about broker configuration URIs.
- *`TransportOptions`* specifies the configuration for the transport. They are specified in the form of a query list. Table 5.2 on page 70 lists the available options.

Example 5.5 on page 70 creates and connects to an embedded broker configured using a broker configuration URI.

Example 5.5. Advanced VM URI

```
vm:(broker:(tcp://localhost:6000)?persistent=false)?marshal=false
```

Transport options

Table 5.2 on page 70 shows options for configuring the VM transport.

Table 5.2. VM Transport Options

Option	Description
`marshal`	If `true`, forces each command sent over the transport to be marshalled and unmarshalled using the specified wire format. Default is `false`.
`wireFormat`	The name of the wire format to use.
`wireFormat.*`	Properties prefixed by `wireFormat.` configure the specified wire format.
`create`	Specifies if the VM transport will create an embedded broker if one does not exist. The default is `true`.
`waitForStart`	Specifies the time, in milliseconds, the VM transport will wait for an embedded broker to start before creating one. The default is `-1` which specifies that the transport will not wait.

Broker options

Table 5.3 on page 70 shows the options used to configure the broker using the simple VM URI.

Table 5.3. Broker Options

Option	Description
`useJmx`	Specifies if JMX is enabled. Default is `true`.

Option	Description
`persistent`	Specifies if the broker uses persistent storage. Default is `true`.
`populateJMSXUserID`	Specifies if the broker populates the `JMSXUserID` message property with the sender's authenticated username. Default is `false`.
`useShutdownHook`	Specifies if the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill. Default is `true`.
`brokerName`	Specifies the broker name. Default is `localhost`.
`deleteAllMessagesOnStartup`	Specifies if all the messages in the persistent store are deleted when the broker starts up. Default is `false`.
`enableStatistics`	Specifies if statistics gathering is enabled in the broker. Default is `true`.

Chapter 6. Peer-to-Peer Protocols

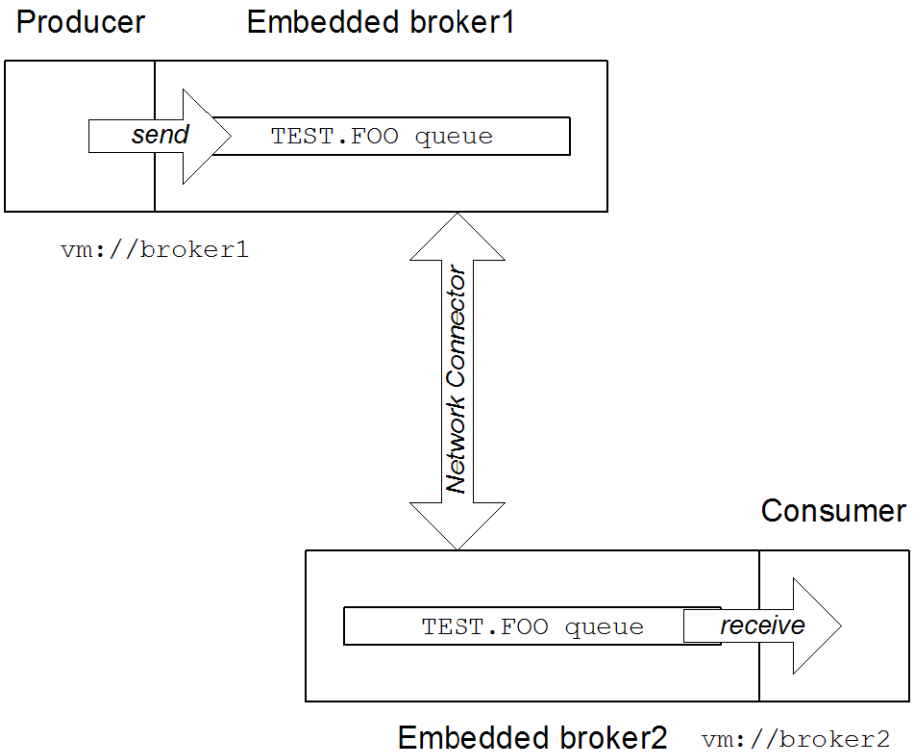
Peer-to-peer protocols enable messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker.

Peer Protocol

Overview

The peer protocol enables you to set up a peer-to-peer network by creating an embedded broker inside each peer endpoint. Figure 6.1 on page 74 illustrates the peer-to-peer network topology for a simple two-peer network.

Figure 6.1. Peer Protocol Endpoints with Embedded Brokers



In this topology, a standalone broker is *not* required, because each peer instantiates its own embedded broker. As shown in Figure 6.1 on page 74 , the producer sends messages to its embedded broker, `broker1`, by connecting to the local VM endpoint, `vm://broker1`—see "VM Transport" on page 67 . The embedded brokers, `broker1` and `broker2`, are linked together using a network connector, which allows messages to flow in either direction between the brokers. When the producer sends a message to the queue, `TEST.FOO`, the first

embedded broker, `broker1`, automatically pushes the message across the network connector and on to the remote embedded broker, `broker2`. The consumer can then receive the message from its embedded broker, `broker2`.

Discovering peer endpoints

Implicitly, the peer protocol uses multicast discovery to locate active peers on the network. In order for this to work, you must ensure that the IP multicast protocol is enabled on your operating system. See "Discovery Agents" in *Using Networks of Brokers* for details.

URI syntax

A `peer` URI must conform to the following syntax:

```
peer://PeerGroup/BrokerName?BrokerOptions
```

Where the group name, *`PeerGroup`*, identifies the set of peers that can communicate with each other. That is, a given peer can connect only to the set of peers that specify the *same* *`PeerGroup`* name in their URLs. The *`BrokerName`* specifies the broker name for the embedded broker. The broker options, *`BrokerOptions`*, are specified in the form of a query list (for example, `?persistent=true`).

Broker options

The peer URL supports the broker options described in Table 6.1 on page 75.

Table 6.1. Broker Options

Option	Description
`useJmx`	If `true`, enables JMX. Default is `true`.
`persistent`	If `true`, the broker uses persistent storage. Default is `true`.
`populateJMSXUserID`	If `true`, the broker populates the `JMSXUserID` message property with the sender's authenticated username. Default is `false`.
`useShutdownHook`	If `true`, the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill. Default is `true`.
`brokerName`	Specifies the broker name. Default is `localhost`.
`deleteAllMessagesOnStartup`	If `true`, deletes all the messages in the persistent store as the broker starts up. Default is `false`.
`enableStatistics`	If `true`, enables statistics gathering in the broker. Default is `true`.

Sample URI

The following is an example of a peer URL that belongs to the peer group, `groupA`, and creates an embedded broker with broker name, `broker1`:

```
peer://groupA/broker1?persistent=false
```

Peer Example

Overview

To try out the peer protocol, perform the following steps:

1. "Start up consumer with embedded broker" .

2. "Start up producer with embedded broker" .

Start up consumer with embedded broker

Start a consumer that consumes messages from the `TEST.FOO` queue belonging to the `group` peer group. To start the consumer, run the consumer tool with a peer group URL as follows:

```
cd InstallDir/example
ant consumer -Durl="peer://group/broker1?persistent=false" -Dmax=100
```

Where the first component of the URL path, `group`, specifies that this peer belongs to the `group` peer group. The second component, `broker1`, specifies the name of the embedded broker and the setting, `persistent=false`, sets a broker option. When the consumer starts up, you should see output like the following in the command window:

```
consumer:
     [echo] Running consumer against server at $url = peer://group/broker1?persistent=false
 for subject $subject = TEST.FOO
     [java] Connecting to URL: peer://group/broker1?persistent=false
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] 15:43:10 INFO  ActiveMQ null JMS Message Broker (broker1) is starting
     [java] 15:43:10 INFO  For help or more information please see:
http://activemq.apache.org/
     [java] 15:43:10 INFO  Using Persistence Adapter: MemoryPersistenceAdapter
     [java] 15:43:10 INFO  Listening for connections at: tcp://fboltond820:2399
     [java] 15:43:10 INFO  Connector tcp://fboltond820:2399 Started
    [java] 15:43:10 INFO  Network Connector org.apache.activemq.transport.discovery.multic
ast.MulticastDiscoveryAgent@da4b71 Started
     [java] 15:43:10 INFO  ActiveMQ JMS Message Broker (broker1, ID:fboltond820-2398-
1192200190327-2:0) started
     [java] 15:43:10 INFO  Connector vm://broker1 Started
     [java] 15:43:10 INFO  JMX consoles can connect to service:jmx:rmi:///jndi/rmi://local
host:1099/jmxrmi
     [java] We are about to wait until we consume: 100 message(s) then we will shutdown
```

While the consumer is starting up, it activates an embedded broker with broker name, `broker1`, and attempts to connect to its peers using a multicast discovery agent.

Start up producer with embedded broker

Start a producer that sends messages to the `TEST.FOO` queue on the `group` peer group. To start the producer, run the producer tool with a peer group URL as follows:

```
cd InstallDir/example
ant producer -Durl="peer://group/broker2?persistent=false" -DsleepTime=1000
```

Where the name of the embedded broker is set to `broker2` and the sleep time (time between successive messages) is set to 1000 ms. When the producer starts up, the output log should include some lines like the following:

```
     [java] 15:43:27 INFO  Establishing network connection between from vm://broker2 to
tcp://fboltond820:2399
     [java] 15:43:28 INFO  Network connection between vm://broker2#2 and tcp://local
host/127.0.0.1:2399(broker1) has been established.
```

These lines indicate that a peer-to-peer connection was successfully established between the embedded brokers, `broker1` and `broker2`. The consumer should now be able to receive the messages sent by the producer.

Appendix A. Transport Options

This appendix is a reference guide for the TCP, NIO, SSL, UDP, and wire format URI options.

TCP and NIO Transports

Overview

The TCP transport is used for creating OpenWire/TCP endpoints. The NIO transport is similar to the TCP transport, except that it uses the Java New I/O (NIO)[1] socket library, which can provide better scalability when used on the server side. TCP and NIO have the same transport options.

Syntax

A TCP URI has the following syntax:

```
tcp://Host[:Port]?transportOptions
```

An NIO URI has the following syntax:

```
nio://Host[:Port]?transportOptions
```

Where the transport options, *`transportOptions`*, are specified as follows:

```
?option=value&option=value&...
```

In XML configuration, you must escape the `&` symbol, replacing it with `&`. For example:

```
?option=value&amp;option=value&amp;...
```

Setting client-side options

When setting a client-side option, the name of the options is exactly as given in Table A.1 on page 81. For example, to enable tracing on a client TCP endpoint, set the `trace` option as follows:

```
tcp://fusesource.com:61616?trace=true
```

Setting server-side options

When setting a server-side option, there are two alternative option syntaxes as follows:

TCP listener socket options
: To configure options on the TCP listener socket, add the `transport.` prefix to the option names shown in Table A.1 on page 81. For example, to enable tracing on a TCP listener socket, set the `trace` option as follows:

[1] http://en.wikipedia.org/wiki/New_I/O

```
tcp://fusesource.com:61616?transport.trace=true
```

TCP connection socket options

To configure options on a TCP connection socket (which is spawned from the listener socket whenever the server accepts a new TCP connection), use the option name exactly as given in Table A.1 on page 81. For example, to enable tracing on a TCP connection socket, set the `trace` option as follows:

```
tcp://fusesource.com:61616?trace=true
```

Options

Table A.1 on page 81 shows the options supported by the TCP and the NIO URIs.

Table A.1. TCP and NIO Transport Options

Option	Default	Description
`minmumWireFormatVersion`	`0`	The minimum wire format version that is allowed.
`trace`	`false`	Causes all commands sent over the transport to be logged.
`daemon`	`false`	Specifies whether the transport thread runs as a daemon or not. Useful to enable when embedding in a Spring container or in a web container, to allow the container to shut down properly.
`useLocalHost`	`true`	When `true`, causes the local machine's name to resolve to `localhost`.
`socketBufferSize`	`64*1024`	Sets the socket buffer size in bytes.
`keepAlive`	`false`	When `true`, enables TCP KeepAlive[2] on the broker connection. Useful to ensure that inactive consumers do not time out.
`soTimeout`	`0`	Sets the socket timeout in milliseconds
`connectionTimeout`	`30000`	A non-zero value specifies the connection timeout in milliseconds. A zero value means wait forever for

[2] http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/overview.html

Option	Default	Description
		the connection to be established. Negative values are ignored.
`closeAsync`	`true`	The `false` value causes all sockets to be closed synchronously.
`soLinger`	`MIN_INTEGER`	When > -1, enables the `SoLinger` socket option with this value. When equal to -1, disables `SoLinger`. (from 5.6.0).
`maximumConnections`	`MAX_VALUE`	The maximum number of sockets the broker is allowed to create.
`diffServ`	`0`	*(Client only)* The preferred Differentiated Services traffic class to be set on outgoing packets, as described in RFC 2475. Valid integer values are `[0,64)`. Valid string values are `EF`, `AF[1-3][1-4]` or `CS[0-7]`. With JDK 6, only works when the Java Runtime uses the IPv4 stack, which can be done by setting the `java.net.preferIPv4Stack` system property to `true`. Cannot be used at the same time as the `typeOfService` option.
`typeOfService`	`0`	*(Client only)* The preferred *type of service* value to be set on outgoing packets. Valid integer values are `[0,256)`. With JDK 6, only works when the Java Runtime uses the IPv4 stack, which can be done by setting the `java.net.preferIPv4Stack` system property to `true`. Cannot be used at the same time as the `diffServ` option.
`wireFormat`		The name of the wire format to use.
`wireFormat.*`		All the properties with this prefix are used to configure the wireFormat.

Option	Default	Description
		See Table A.4 on page 88 for more information.

SSL Transport

Overview

The SSL transport is used for creating OpenWire/TCP endpoints with SSL/TLS enabled.

> **Note**
>
> The URI transport options described here are *not* sufficient to configure an SSL endpoint completely. You must also associate X.509 certificates with the endpoint. For more details, see ????.

Syntax

An SSL URI has the following syntax:

```
ssl://Host[:Port]?transportOptions
```

Where the transport options, *`transportOptions`*, are specified as follows:

```
?option=value&option=value&...
```

In XML configuration, you must escape the `&` symbol, replacing it with `&`. For example:

```
?option=value&amp;option=value&amp;...
```

TCP transport options

The SSL transport inherits all of the options supported by the TCP transport URI. See Table A.1 on page 81.

Options

Table A.2 on page 84 shows the options supported by the SSL URI.

Table A.2. SSL Transport Options

Option	Default	Description
`transport.enabledCipherSuites`		Specifies the cipher suites accepted by this endpoint, in the form of a comma-separated list.

Option	Default	Description
`transport.enabledProtocols`		Specifies the secure socket protocols accepted by this endpoint, in the form of a comma-separated list. If using Sun's JSSE provider, possible values are: `SSL`, `SSLv2`, `SSLv3`, `TLS`, or `TLSv1`.
`transport.wantClientAuth`		*(Server only)* If `true`, the server requests (but does not require) the client to send a certificate.
`transport.needClientAuth`	`false`	*(Server only)* If `true`, the server *requires* the client to send its certificate. If the client fails to send a certificate, the server will throw an error and close the session.
`transport.enableSessionCreation`	`true`	*(Server only)* If `true`, the server socket creates a new SSL session every time it accepts a connection and spawns a new socket. If `false`, an existing SSL session must be resumed when the server socket accepts a connection.

UDP Transport

Overview

The UDP transport enables you to send datagrams using the unreliable UDP/IP protocol.

Warning

Because UDP does not keep track of IP packets, *you can lose messages* sent over a raw UDP connection. It is up to you to provide a reliability layer to avoid message loss over UDP.

Syntax

A UDP URI has the following syntax:

```
udp://Host[:Port]?transportOptions
```

Where the transport options, *`transportOptions`*, are specified as follows:

```
?option=value&option=value&...
```

In XML configuration, you must escape the `&` symbol, replacing it with `&`. For example:

```
?option=value&amp;option=value&amp;...
```

Options

Table A.3 on page 86 shows the options supported by the UDP URI.

Table A.3. SSL Transport Options

Option	Default	Description
`minmumWireFormatVersion`	`0`	The minimum version wire format that is allowed.
`trace`	`false`	Causes all commands sent over the transport to be logged.
`useLocalHost`	`true`	When `true`, causes the local machine's name to resolve to `localhost`.

Option	Default	Description
`datagramSize`	`4*1024`	Specifies the size of a datagram.
`wireFormat`		The name of the wire format to use.
`wireFormat.*`		All options with this prefix are used to configure the wire format. See Table A.4 on page 88 for more information.

Wire Format Options

Overview

The wire format options configure the OpenWire message layer and can be specified as transport options with any of the preceding transports.

Options

Table A.4 on page 88 shows the wire format options supported by the OpenWire protocol.

Table A.4. Transport Options Supported by OpenWire Protocol

Option	Default	Description	Negotiation policy
`wireformat .stackTraceEnabled`	`true`	Should the stack trace of an exception occuring on the broker be sent to the client?	Set to `false` if either side is `false`.
`wireformat .tcpNoDelayEnabled`	`false`	Provides a hint to the peer that TCP `nodelay` should be enabled on the communications Socket.	Set to `false` if either side is `false`.
`wireformat .cacheEnabled`	`true`	Should commonly repeated values be cached so that less marshalling occurs?	Set to `false` if either side is `false`.
`wireformat .cacheSize`	`1024`	If `cacheEnabled` is `true`, this property specifies the maximum number of values to cache.	Use the smaller of the two values.
`wireformat .tightEncodingEnabled`	`true`	Should wire size be optimized over CPU usage?	Set to `false` if either side is `false`.
`wireformat .prefixPacketSize`	`true`	Should the size of the packet be prefixed before each packet is marshalled?	Set to `true` if both sides are `true`.
`wireformat .maxInactivityDuration`	`30000`	The maximum inactivity duration (before which the socket is considered dead)	Use the smaller of the two values.

Option	Default	Description	Negotiation policy
		in milliseconds. On some platforms it can take a long time for a socket to appear to die, so we allow the broker to kill connections if they are inactive for a period of time. Set to a value `<= 0` to disable inactivity monitoring.	
`wireformat.maxInactivityDurationInitalDelay`	`10000`	The initial delay in starting the maximum inactivity checks. **Note:** The mis-spelling, `Inital`, is a typographic error in the source code.	

Index

E

V