Fuse MQ Enterprise

Connectivity Guide

Version 7.1
December 2012

Integration Everywhere

Connectivity Guide

Version 7.1

Copyright © 2012 Red Hat, Inc. and/or its affiliates.

Trademark Disclaimer

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Fuse, Red Hat, Fuse ESB, Fuse ESB Enterprise, Fuse MQ Enterprise, Fuse Mediation Router, Fuse Message Broker, Fuse Services Framework, Fuse IDE, Fuse HQ, Fuse Management Console, and Integration Everywhere are trademarks or registered trademarks of Red Hat Corp. or its parent corporation, Progress Software Corporation, or one of their subsidiaries or affiliates in the United States. Apache, ServiceMix, Camel, CXF, and ActiveMQ are trademarks of Apache Software Foundation. Any other names contained herein may be trademarks of their respective owners.

Third Party Acknowledgements

One or more products in the Fuse MQ Enterprise 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 MQ Enterprise 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 MQ Enterprise 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. This URL is also used to configure the transport connector in a broker.

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

Wire Protocol	Transport Protocol	Sample URL	Description
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.
REST	HTTP	`http://Host:Port/demo/message/FOO/BAR`	Connect to the message broker endpoint at `Host:Port` using the REST

Wire Protocol	Transport Protocol	Sample URL	Description
		`?timeout=10000` `&type=queue`	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" on page 62.
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" on page 64.

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" on page 54.

Note

Although discovery agent URIs look superficially like transport URIs, they are not the same thing. A discovery agent URI can only be used in certain contexts and *cannot* be used directly in place of a transport URI.

Chapter 2. OpenWire Protocol

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

Overview

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

Transport protocols

Table 2.1 on page 17 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 18 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 each packet is marshalled?	Set to `true` if both sides are `true`.

Property	Default	Description	Negotiation policy
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 MQ Enterprise 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/

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 22 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`*.

Transport options

The Stomp protocol supports the following transport options:

Table 3.2. Transport Options Supported by Stomp Protocol

Property	Default	Description
`transport .defaultHeartBeat`	`0,0`	Simulated client heartbeat policy of legacy Stomp 1.0 clients. This option is set in the `uri` attribute of a broker's `transportConnector` element to enable backward compatibility with Stomp 1.0 clients.

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*.

Stomp Heartbeats

Stomp 1.1 heartbeats

Stomp 1.1 adds support for heartbeats (keepalive messages) on Stomp connections. Negotiation of a heartbeat policy is normally initiated by the client (Stomp 1.1 clients only) and the client must be configured to enable heartbeats. No broker settings are required to enable support for heartbeats, however.

At the level of the Stomp wire protocol, heartbeats are negotiated when the client establishes the Stomp connection and the following messages are exchanged between client and server:

```
CONNECT
heart-beat:CltSend,CltRecv

CONNECTED:
heart-beat:SrvSend,SrvRecv
```

The *CltSend*, *CltRecv*, *SrvSend*, and *SrvRecv* fields are interpreted as follows:

CltSend

Indicates the minimum frequency of messages *sent from the client*, expressed as the maximum time between messages in units of milliseconds. If the client does not send a regular Stomp message within this time limit, it must send a special heartbeat message, in order to keep the connection alive.

A value of zero indicates that the client does not send heartbeats.

CltRecv

Indicates how often the client expects to *receive* message from the server, expressed as the maximum time between messages in units of milliseconds. If the client does not receive any messages from the server within this time limit, it would time out the connection.

A value of zero indicates that the client does not expect heartbeats and will not time out the connection.

SrvSend

Indicates the minimum frequency of messages *sent from the server*, expressed as the maximum time between messages in units of milliseconds. If the server does not send a regular Stomp message within

this time limit, it must send a special heartbeat message, in order to keep the connection alive.

A value of zero indicates that the server does not send heartbeats.

SrvRecv

Indicates how often the server expects to *receive* message from the client, expressed as the maximum time between messages in units of milliseconds. If the server does not receive any messages from the client within this time limit, it would time out the connection.

A value of zero indicates that the server does not expect heartbeats and will not time out the connection.

In order to ensure that the rates of sending and receiving required by the client and the server are mutually compatible, the client and the server negotiate the heartbeat policy, adjusting their sending and receiving rates as needed.

Stomp 1.0 heartbeat compatibility

A difficulty arises, if you want to support an inactivity timeout on your Stomp connections when legacy Stomp 1.0 clients are connected to your broker. The Stomp 1.0 protocol does *not* support heartbeats, so Stomp 1.0 clients are not capable of negotiating a heartbeat policy.

To get around this limitation, you can specify the `transport.defaultHeartBeat` option in the broker's `transportConnector` element, as follows:

```
<transportConnector name="stomp"
uri="stomp://0.0.0.0:0?transport.defaultHeartBeat=5000,0" />
```

The effect of this setting is that the broker now behaves *as if* the Stomp 1.0 client had sent the following Stomp frame when it connected:

```
CONNECT
heart-beat:5000,0
```

This means that the broker will expect the client to send a message at least once every 5000 milliseconds (5 seconds). The second integer value, `0`, indicates that the client does not expect to receive any heartbeats from the server (which makes sense, because Stomp 1.0 clients do not understand heartbeats).

Now, if the Stomp 1.0 client does not send a regular message after 5 seconds, the connection will time out, because the Stomp 1.0 client is not capable of sending out a heartbeat message to keep the connection alive. Hence, you

should choose the value of the timeout in `transport.defaultHeartBeat` such that the connection will stay alive, as long as the Stomp 1.0 clients are sending messages at their normal rate.

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 MQ Enterprise'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 34 .
- *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 30.

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

Table 4.1. HTTP RESTful Operations

Operation	Description
`GET`	Fetch the remote resource identified by the URI.
`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/mes
sage/FOO/BAR?type=queue&timeout=5000
```

Table 4.2 on page 31 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 32 .

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 `POST`ing a message), see "Example of posting a message" on page 34 .

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 30), 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 33 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.

URL Option	Description
maxMessages	The maximum number of messages to render.

Example of posting a message

Example 4.1 on page 34 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 39 .

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 35 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, `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 30 . For example, to consume a message from the `FOO.BAR` queue, navigate to the following URL:

```
http://localhost:8161/demo/mes
sage/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 32 .

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&feed
Type=atom_1.0
```

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

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=rss&feed
Type=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 37 .

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 37 , 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 37.

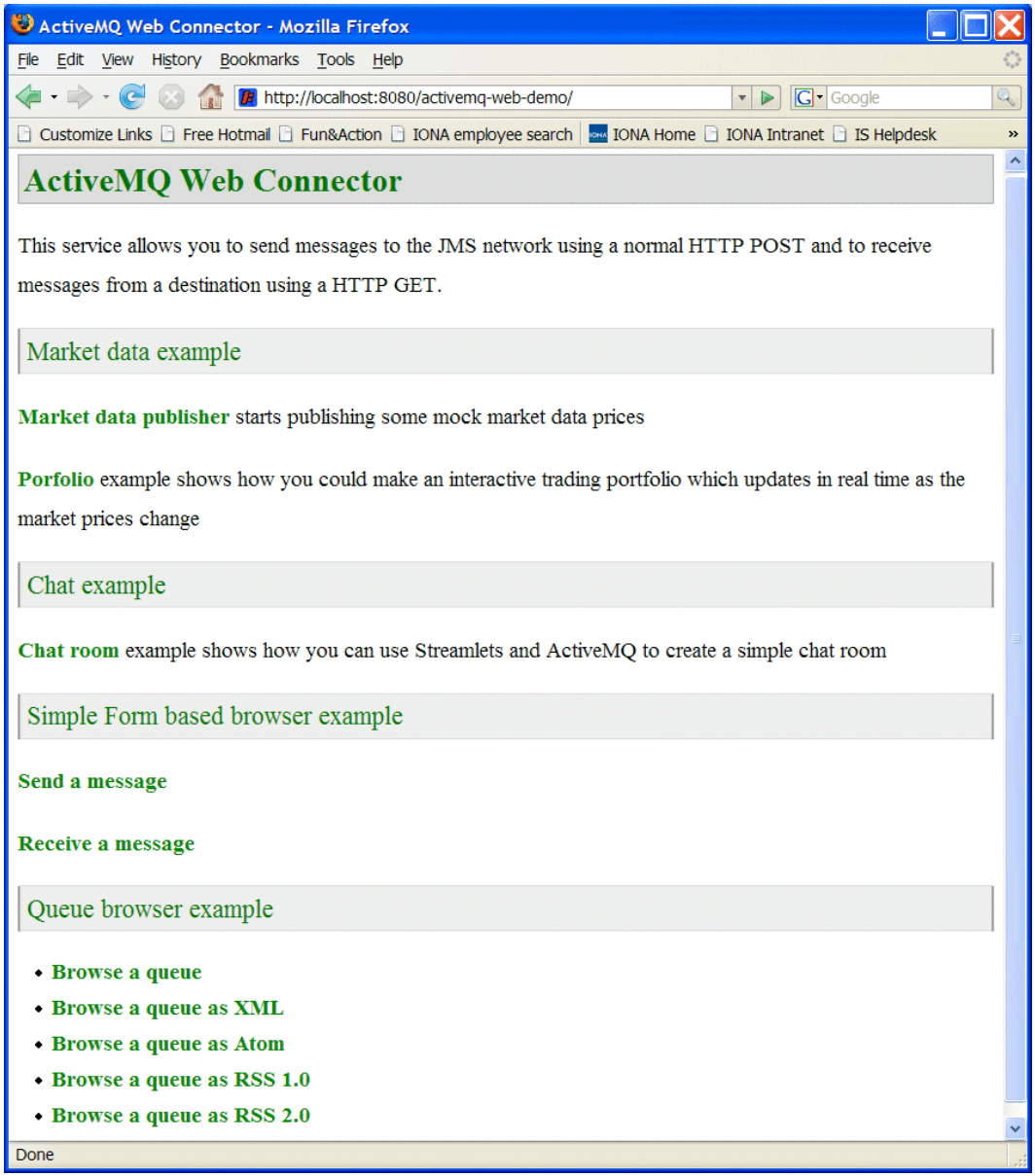
Open a Web browser

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 39.

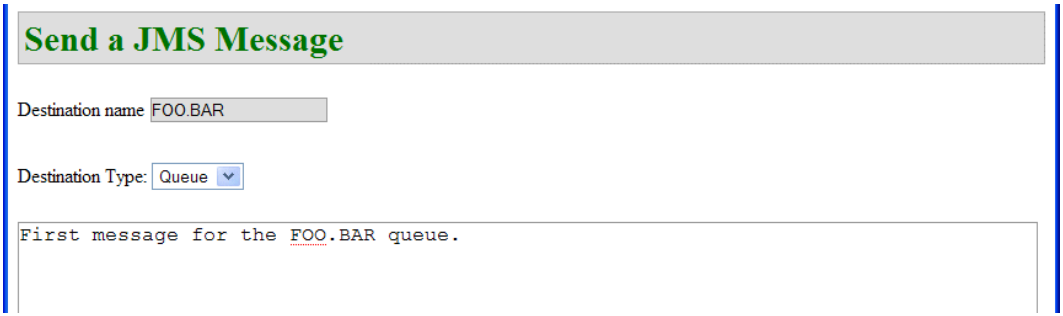
Figure 4.1. Welcome Page for Web Examples



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 40.

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 39). 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].

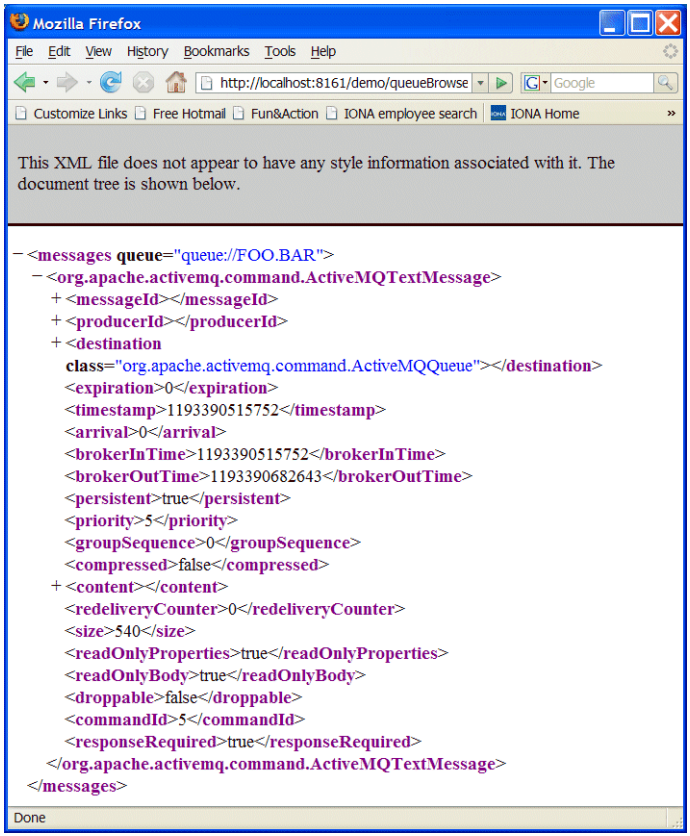
[2] http://localhost:8080/activemq-web-demo/send.html
[3] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR
[4] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=xml
[5] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=atom_1.0
[6] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=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 41 .

Figure 4.3. Default Option to Browse a Queue

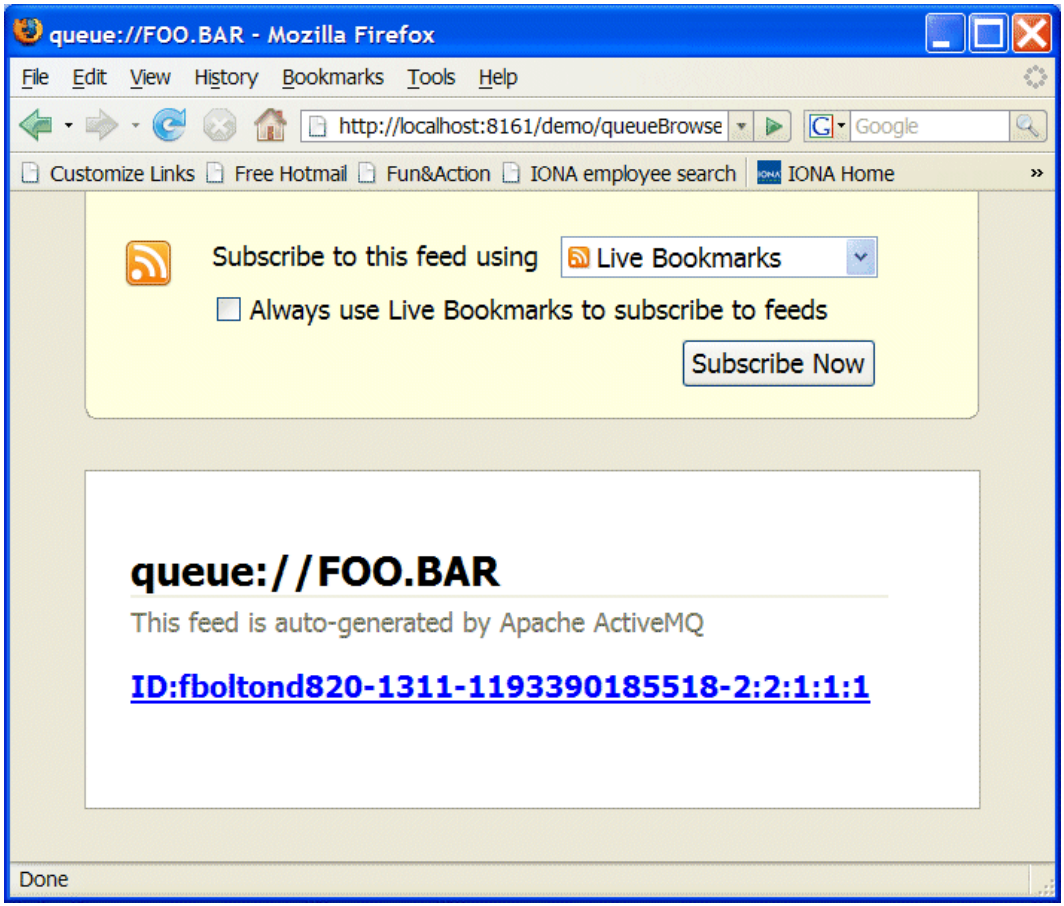
If you click on **Browse a queue as XML**, you should see a page like Figure 4.4 on page 42 .

Figure 4.4. Option to Browse a Queue as XML



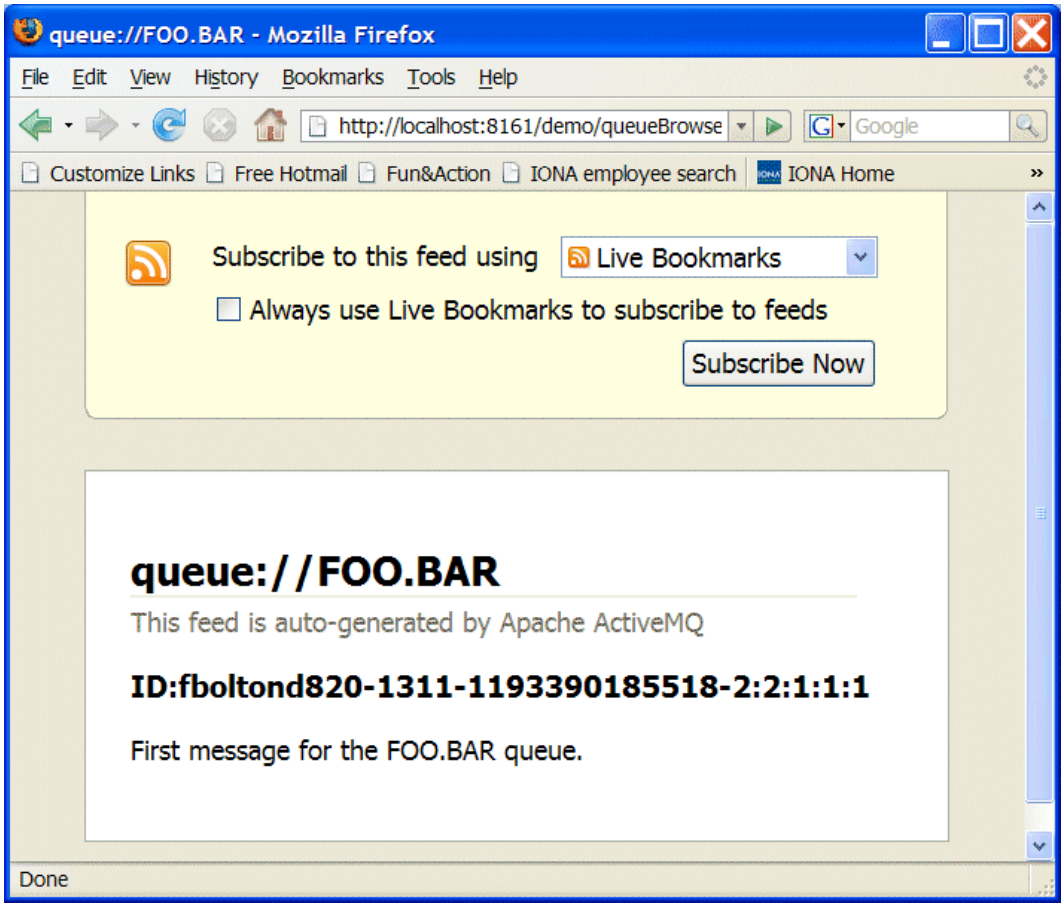
If you click on **Browse a queue as Atom**, you should see a page like Figure 4.5 on page 43 .

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 44 .

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).

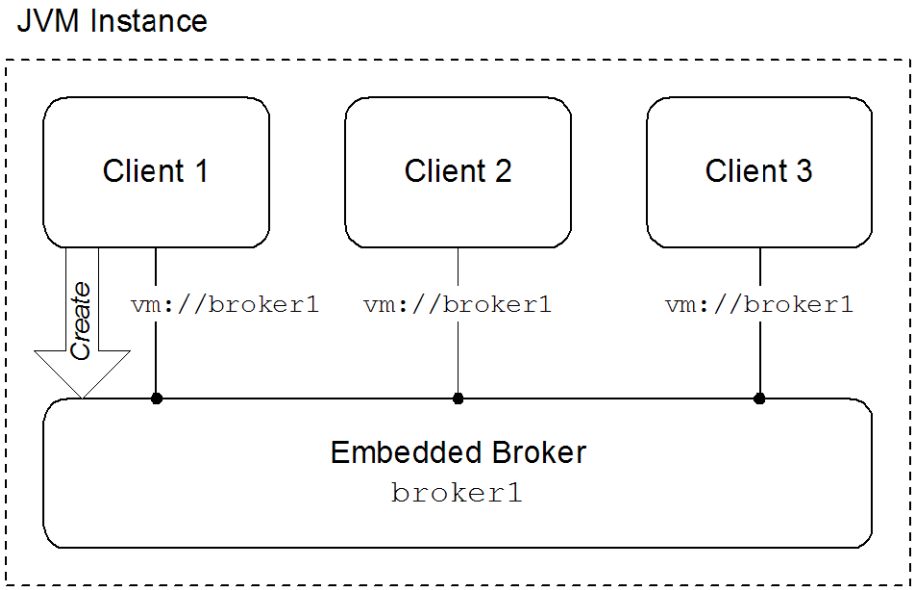
Chapter 5. VM Transport

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 47 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 48 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 50 lists the available options.

In addition to the transport options listed in Table 5.2 on page 50, the simple VM URI can use the options described in Table 5.1 on page 49 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 51.
`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`.

Important

The broker configuration options specified on the VM URI are only meaningful if the client is responsible for instantiating the embedded broker. If the embedded broker is already started, the transport will ignore the broker configuration properties.

Example 5.2 on page 49 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 50 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 advanced 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 50 shows the syntax for an advanced VM URI.

Example 5.4. Advanced VM URI Syntax

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

- *BrokerConfigURI* is a broker configuration URI.
- *TransportOptions* specifies the configuration for the transport. They are specified in the form of a query list. Table 5.2 on page 50 lists the available options.

Example 5.5 on page 50 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)?mar
shal=false
```

Transport options

Table 5.2 on page 50 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.

Option	Description
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 51 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.
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. Discovering Brokers

One of the main strengths of Fuse MQ Enterprise is that brokers can be located dynamically through out your infrastructure. In order for clients and other brokers to be able to interact with a broker, they need some way of discovering that the broker exists. Fuse MQ Enterprise does this using a combination of discovery agents and special URI schemes.

In order for location transparency to work, the members of a messaging application need a way for discovering each other. In Fuse MQ Enterprise this is accomplished using two pieces:

- *discovery agents*—components that advertise the brokers available to other members of a messaging applicaiton
- *discovery URI*—a URI that looks up all of the discoverable brokers and presents them as a list of actual URIs for use by the client or network connector

Discovery Agents

A discovery agent is a mechanism that advertises available brokers to clients and other brokers. When a client, or broker, using a discovery URI starts up it will look for any brokers that are available using the specified discovery agent. The clients will update their lists periodically using the same mechanism.

How a discovery agent learns about the available brokers varies between agents. Some agents use a static list, some use a third party registry, and some rely on the brokers to provide the information. For discovery agents that rely on the brokers for information, it is necessary to enable the discovery agent in the message broker configuration. For example, to enable the multicast discovery agent on an Openwire endpoint, you edit the relevant `transportConnector` element as shown in Example 6.1 on page 54.

Example 6.1. Enabling a Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://default" />
</transportConnectors>
```

Where the `discoveryUri` attribute on the `transportConnector` element is initialized to `multicast://default`.

Tip

If a broker uses multiple transport connectors, you need to configure each transport connector to use a discovery agent individually. This means that different connectors can use different discovery mechanisms or that one or more of the connectors can be indiscoverable.

Fuse MQ Enterprise currently supports the following discovery agents:

- Fuse Fabric Discovery Agent

- Static Discovery Agent
- Multicast Discovery Agent
- Zeroconf Discovery Agent

Fuse Fabric Discovery Agent

Overview

The *Fuse Fabric discovery agent* uses Fuse Fabric to discover the brokers in a specified group. The discovery agent requires that all of the discoverable brokers be deployed into a single fabric. When the client attempts to connect to a broker the agent looks up all of the available brokers in the fabric's registry and returns the ones in the specified group.

URI

The Fuse Fabric discovery agent URI conforms to the syntax in Example 6.2 on page 56.

Example 6.2. Fuse Fabric Discovery Agent URI Format

```
fabric://GID
```

Where *GID* is the ID of the broker group from which the client discovers the available brokers.

Configuring a broker

The Fuse Fabric discovery agent requires that the discoverable brokers are deployed into a single fabric.

The best way to deploy brokers into a fabric is using Fuse Management Console. For information on using Fuse Management Console see Fuse Management Console Documentation[1].

You can also use the console to deploy brokers into a fabric. See "Fabric Console Commands" in *Console Reference*.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a Fuse Fabric agent URI as shown in Example 6.3 on page 56.

Example 6.3. Client Connection URL using Fuse Fabric Discovery

```
discovery://(fabric://nwBrokers)
```

A client using the URL in Example 6.3 on page 56 will discover all the brokers in the `nwBrokers` broker group and generate a list of brokers to which it can connect.

[1] http://fusesource.com/docs/fmc

Static Discovery Agent

Overview

The *static discovery agent* does not truly discover the available brokers. It uses an explicit list of broker URLs to specify the available brokers. Brokers are not involved with the static discovery agent. The client only knows about the brokers that are hard coded into the agent's URI.

Using the agent

The static discovery agent is a client-side only agent. It does not require any configuration on the brokers that will be discovered.

To use the agent, you simply configure the client to connect to a broker using a discovery protocol that uses a static agent URI.

The static discovery agent URI conforms to the syntax in Example 6.4 on page 57.

Example 6.4. Static Discovery Agent URI Format

```
static://(URI1,URI2,URI3,...)
```

Example

Example 6.5 on page 57 shows a discovery URI that configures a client to use the static discovery agent to connect to one member of a broker pair.

Example 6.5. Discovery URI using the Static Discovery Agent

```
discovery://(static://(tcp://localhost:61716,tcp://local
host:61816))
```

Multicast Discovery Agent

Overview

The *multicast discovery agent* uses the IP multicast protocol to find any message brokers currently active on the local network. The agent requires that *each* broker you want to advertise is configured to use the multicast agent to publish its details to a multicast group. Clients using the multicast agent as part of the discovery URI they use for connecting to a broker will use the agent to receive the list of available brokers advertising in the specified multicast group.

Important

Your local network (LAN) must be configured appropriately for the IP/multicast protocol to work.

URI

The multicast discovery agent URI conforms to the syntax in Example 6.6 on page 58.

Example 6.6. Multicast Discovery Agent URI Format

```
multicast://GroupID
```

Where *GroupID* is an alphanumeric identifier. All participants in the same discovery group must use the same *GroupID*.

Configuring a broker

For a broker to be discoverable using the multicast discovery agent, you must enable the discovery agent in the broker's configuration. To enable the multicast discovery agent you set the `transportConnector` element's `discoveryUri` attribute to a mulitcast discovery agent URI as shown in Example 6.7 on page 58.

Example 6.7. Enabling a Multicast Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://default" />
</transportConnectors>
```

The broker configured in Example 6.7 on page 58 is discoverable as part of the multicast group `default`.

Configuring a client

To use the multicast agent a client must be configured to connect to a broker using a discovery URI that uses a multicast agent URI as shown in Example 6.8 on page 59.

Example 6.8. Client Connection URL using Multicast Discovery

```
discovery://(multicast://default)
```

A client using the URI in Example 6.8 on page 59 will discover all the brokers advertised in the `default` multicast group and generate a list of brokers to which it can connect.

Zeroconf Discovery Agent

Overview

The *zeroconf discovery agent* is derived from Apple's Bonjour Networking[2] technology, which defines the zeroconf protocol as a mechanism for discovering services on a network. Fuse MQ Enterprise bases its implementation of the zeroconf discovery agent on JmDSN[3], which is a service discovery protocol that is layered over IP/multicast and is compatible with Apple Bonjour.

The agent requires that *each* broker you want to advertise is configured to use a multicast discovery agent to publish its details to a multicast group. Clients using the zeroconf agent as part of the discovery URI they use for connecting to a broker will use the agent to receive the list of available brokers advertising in the specified multicast group.

Important

Your local network (LAN) must be configured to use IP/multicast for the zeroconf agent to work.

URI

The zeroconf discovery agent URI conforms to the syntax in Example 6.9 on page 60.

Example 6.9. Zeroconf Discovery Agent URI Format

```
zeroconf://GroupID
```

Where the *GroupID* is an alphanumeric identifier. All participants in the same discovery group must use the same *GroupID*.

Configuring a broker

For a broker to be discoverable using the zeroconf discovery agent, you must enable a multicast discovery agent in the broker's configuration. To enable the multicast discovery agent you set the `transportConnector` element's `discoveryUri` attribute to a mulitcast discovery agent URI as shown in Example 6.10 on page 61.

[2] http://developer.apple.com/networking/bonjour/
[3] http://sourceforge.net/projects/jmdns/

Example 6.10. Enabling a Multicast Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://NEGroup" />
</transportConnectors>
```

The broker configured in Example 6.10 on page 61 is discoverable as part of the multicast group `NEGroup`.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a zeroconf agent URI as shown in Example 6.11 on page 61.

Example 6.11. Client Connection URL using Zeroconf Discovery

```
discovery://(zeroconf://NEGroup)
```

A client using the URL in Example 6.11 on page 61 will discover all the brokers advertised in the `NEGroup` multicast group and generate a list of brokers to which it can connect.

Dynamic Discovery Protocol

Overview

The *dynamic discovery protocol* combines reconnect logic with a discovery agent to dynamically create a list of brokers to which the client can connect. The discovery protocol invokes a discovery agent in order to build up a list of broker URIs. The protocol then randomly chooses a URI from the list and attempts to establish a connection to it. If it does not succeed, or if the connection subsequently fails, a new connection is established to one of the other URIs in the list.

URI syntax

Example 6.12 on page 62 shows the syntax for a discovery URI.

Example 6.12. Dynamic Discovery URI

```
discovery://(DiscoveryAgentUri)?Options
```

DiscoveryAgentUri is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in "Discovery Agents" on page 54.

The options, ?*Options*, are specified in the form of a query list. The discovery options are described in Table 6.1 on page 62. You can also inject transport options as described in "Setting options on the discovered transports" on page 63.

Tip

If no options are required, you can drop the parentheses from the URI. The resulting URI would take the form `discovery://DiscoveryAgentUri`

Transport options

The discovery protocol supports the options described in Table 6.1 on page 62.

Table 6.1. Dynamic Discovery Protocol Options

Option	Default	Description
`initialReconnectDelay`	`10`	Specifies, in milliseconds, how long to wait before the first reconnect attempt.

Option	Default	Description
maxReconnectDelay	30000	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
useExponentialBackOff	true	Specifies if an exponential back-off is used between reconnect attempts.
backOffMultiplier	2	Specifies the exponent used in the exponential back-off algorithm.
maxReconnectAttempts	0	Specifies the maximum number of reconnect attempts before an error is sent back to the client. 0 specifies unlimited attempts.

Sample URI

Example 6.13 on page 63 shows a discovery URI that uses a multicast discovery agent.

Example 6.13. Discovery Protocol URI

```
discovery://(multicast://default)?initialReconnectDelay=100
```

Setting options on the discovered transports

The list of transport options, *Options*, in the discovery URI can also be used to set options on the *discovered* transports. If you set an option *not* listed in "Setting options on the discovered transports" on page 63, the URI parser attempts to inject the option setting into every one of the discovered endpoints.

Example 6.14 on page 63 shows a discovery URI that sets the TCP connectionTimeout option to 10 seconds.

Example 6.14. Injecting Transport Options into a Discovered Transport

```
discovery://(multicast://default)?connectionTimeout=10000
```

The 10 second timeout setting is injected into every discovered TCP endpoint.

Fanout Protocol

Overview

The *fanout protocol* enables a producer to auto-discover broker endpoints and broadcast topic messages to *all* of the discovered brokers. The fanout protocol gives producers a convenient mechanism for broadcasting messages to multiple brokers that are not part of a network of brokers.

The fanout protocol relies on a discovery agent to build up the list of broker URIs to which it connects.

URI syntax

Example 6.15 on page 64 shows the syntax for a fanout URI.

Example 6.15. Fanout URI Syntax

```
fanout://(DiscoveryAgentUri)?Options
```

DiscoveryAgentUri is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in "Discovery Agents" on page 54.

The options, *?Options*, are specified in the form of a query list. The discovery options are described in Table 6.2 on page 64. You can also inject transport options as described in "Setting options on the discovered transports" on page 63.

Tip

If no options are required, you can drop the parentheses from the URI. The resulting URI would take the form `fanout://DiscoveryAgentUri`

Transport options

The fanout protocol supports the transport options described in Table 6.2 on page 64.

Table 6.2. Fanout Protocol Options

Option Name	Default	Description
initialReconnectDelay	10	Specifies, in milliseconds, how long the transport will wait before the first reconnect attempt.

Option Name	Default	Description
maxReconnectDelay	30000	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
useExponentialBackOff	true	Specifies if an exponential back-off is used between reconnect attempts.
backOffMultiplier	2	Specifies the exponent used in the exponential back-off algorithm.
maxReconnectAttempts	0	Specifies the maximum number of reconnect attempts before an error is sent back to the client. 0 specifies unlimited attempts.
fanOutQueues	false	Specifies whether queue messages are replicated to every connected broker. For more information see "Applying fanout to queue messages" on page 65.
minAckCount	2	Specifies the minimum number of brokers to which the client must connect before it sends out messages. For more informaiton see "Minimum number of brokers" on page 66.

Sample URI

Example 6.16 on page 65 shows a discovery URI that uses a multicast discovery agent.

Example 6.16. Fanout Protocol URI

```
fanout://(multicast://default)?initialReconnectDelay=100
```

Applying fanout to queue messages

The fanout protocol replicates topic messages by sending each topic message to all of the connected brokers. By default, however, the fanout protocol does *not* replicate queue messages.

For queue messages, the fanout protocol picks one of the brokers at random and sends all of the queue messages to that broker. This is a sensible default, because under normal circumstances, you would not want to create more than one copy of a queue message.

It is possible to change the default behavior by setting the `fanOutQueues` option to `true`. This configures the protocol so that it also replicates queue messages.

Minimum number of brokers

By default, the fanout protocol does not start sending messages until the producer has connected to a *minimum of two brokers*. You can customize this minimum value using the `minAckCount` option.

Setting minimum number of brokers equal to the expected number of discovered brokers ensures that all of the available brokers start receiving messages at the same time. This ensures that no messages are missed if a broker starts up after the producer has started sending messages.

Using fanout with a broker network

You have to be careful when using the fanout protocol with brokers that are joined in a network of brokers.

The combination of the fanout protocol's broadcasting behavior and the nature of how messages are propagated through a network of brokers makes it likely that consumers will receive duplicate messages. If, for example, you joined four brokers into a network of brokers and connected a consumer listening for messages on topic `hello.jason` to broker A and connected a producer to broker B to send messages to topic `hello.jason`, the consumer would get one copy of the messages. If, on the other hand, the producer connects to the network using the fanout protocol, the producer will connect to every broker in the network simultaneously and start sending messages. Each of the four brokers will receive a copy of every message and deliver its copy to the consumer. So, for each message, the consumer will get four copies.

Chapter 7. Peer-to-Peer Protocols

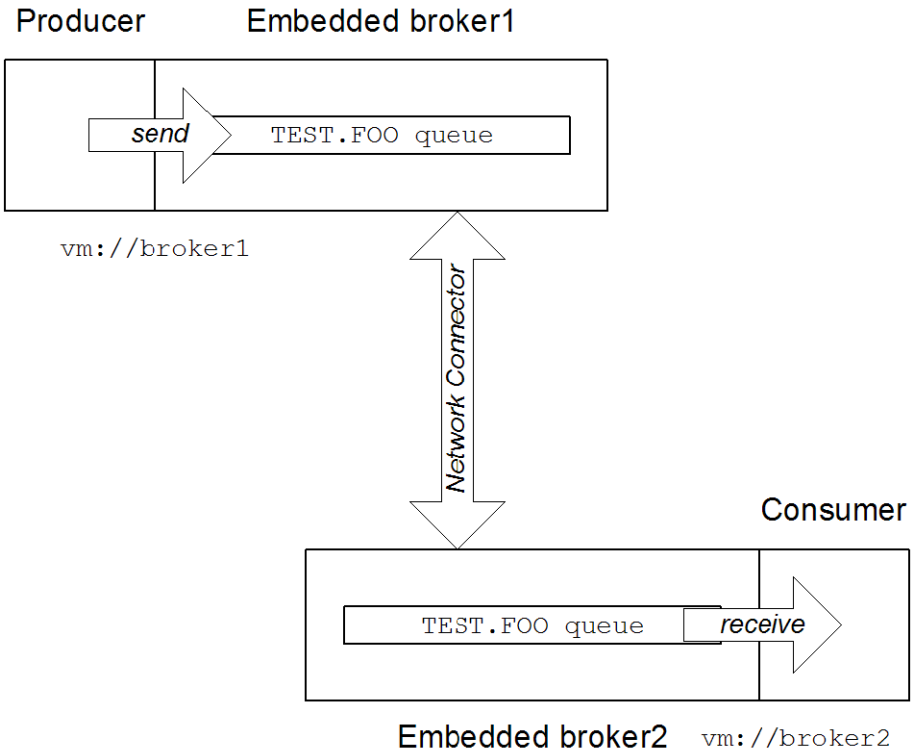
Peer-to-peer protocols enable messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker.

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 7.1 on page 68 illustrates the peer-to-peer network topology for a simple two-peer network.

Figure 7.1. 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 7.1 on page 68 , the producer sends messages to its embedded broker, `broker1`, by connecting to the local VM endpoint, `vm://broker1`—see "VM Transport" on page 47 . 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" on page 54 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 7.1 on page 69.

Table 7.1. 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`.

Option	Description
`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 75. 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 75. 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 75. 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 75 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.

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

Option	Default	Description
soTimeout	0	Sets the socket timeout in milliseconds
soWriteTimeout	0	Sets the socket write timeout in milliseconds
connectionTimeout	30000	A non-zero value specifies the connection timeout in milliseconds. A zero value means wait forever for 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

Option	Default	Description
		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. See Table A.4 on page 82 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 "SSL/TLS Security" in *Security Guide*.

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 75.

Options

Table A.2 on page 78 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.
`transport.enabledProtocols`		Specifies the secure socket protocols accepted by this endpoint, in the form of a comma-separated list. If using

Option	Default	Description
		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 80 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`.
`datagramSize`	`4*1024`	Specifies the size of a datagram.

Option	Default	Description
`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 82 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 82 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) 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	Use the smaller of the two values.

Option	Default	Description	Negotiation policy
		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