Fuse Message Broker

Managing and Monitoring a Broker

Version 5.5
Febuary 2012

Managing and Monitoring a Broker

Version 5.5

Updated: 27 Mar 2014

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 Tables

List of Examples

Chapter 1. Introduction

Once a messaging solution is deployed it needs to be monitored to ensure it performs at peak performance. When problems do arise, many of them can be solved using the broker's administrative tools. The broker's administrative tools can also be used to provide important debugging information when troubleshooting problems.

Overview

Message brokers are long lived and usually form the backbone of the applications of which they are a part. Over the course of a broker's life span, there are a number of management tasks that you may need to do to keep the broker running at peak performance. This includes monitoring the health of the broker, adding destinations, and security certificates.

If applications run into trouble one of the first places to look for clues is the broker. The broker is unlikely to be the root cause of the problem, but its logs and metrics will provide clues as to what is the root cause. You may also be able to resolve the problem using the broker's administrative interface.

Routine tasks

While Fuse Message Broker is designed to require a light touch for management, there are a few routine management tasks that need to be performed:

- installing SSL certificates
- starting the broker
- creating destinations
- stopping the broker
- maintaining the advisory topics
- monitoring the health of the broker
- monitoring the health of the destinations

Troubleshooting

If an application runs into issues the broker will usually be able to provide clues to what is going wrong. Because the broker is central to the operation of any application that relies on messaging, it will be able to provide clues even if the broker is functioning properly. You may also be able to solve the problem by making adjustments to the broker's configuration.

Common things to check for clues as to the nature of a problem include:

- the broker's log file
- the advisory topics
- the broker's overall memory footprint
- the size of individual destination
- the total number of messages in the broker
- the size of the broker's persistent store
- a thread dump of the broker

One or more of these items can provide information about the problem. For example, if a destination grows to a very large size it could indicate that one of its consumers is having trouble keeping up with the messages. If the broker's log also shows that the consumer is repeatedly connecting and disconnecting from the destination, that could indicate a networking problem or a problem with the machine hosting the consumer.

Tools

There are a number of tools that you can use to monitor and administer Fuse Message Broker.

The following tools are included with Fuse Message Broker:

- administration tool on page 13—a command line tool that can be used to manage a broker and do rudimentary metric reporting
- Web console on page 19—a browser based console that provides metric reporting, destination browsing, and other administrative functions

For more advanced monitoring and management capabilities FuseSource offers Fuse HQ[1].

In addition to the FuseSource supplied tools there are a number of third party tools that can be used to administer and monitor a broker including:

- jconsole—a JMX tool that is shipped with the JDK
- VisualVM[2]—a visual tool integrating several command line JDK tools and lightweight profiling capabilities

[1] http://fusesource.com/docs/hq
[2] http://visualvm.java.net/

Chapter 2. Setting up the Administration Tool

Fuse Message Broker's administrative tool requires the shell's environment be set up properly. This involves setting a number of environment variables and can typically be accomplished by creating a simple shell script.

Important

The administration tool requires that brokers have JMX enabled. By default, JMX is enabled. For more information see "Using JMX" on page 53.

Setting up the Windows Environment

Overview

The Windows Fuse Message Broker administrative tool uses four environment variables to determine where to locate configuration files, runtime libraries, and data files. In addition, it requires that your Java environment is properly set up.

Optionally, you may also want to add the administrative tool to the shell's path so that you can use the tool from anywhere.

Environment variables

Table 2.1 on page 14 describes the environment variables used by the administration tool on Windows platforms.

Table 2.1. Windows Administration Tool Environment Variables

Variable	Description
`ACTIVEMQ_HOME`	Specifies the directory where Fuse Message Broker is installed.
`ACTIVEMQ_BASE`	Specifies the directory containing files associated with the current broker instance. Default value is the value of `ACTIVEMQ_HOME`.
`ACTIVEMQ_CLASSPATH`	Specifies the classpath used by the current broker instance.
`SSL_OPTS`	Specifies Java properties required to support SSL/TLS protocols. See *Security Guide* for details.

Java environment

You must set the `JAVA_HOME` environment variable to the location of the Java installation you want to use for running Fuse Message Broker.

Setup script

Example 2.1 on page 14 shows a sample script for setting up the administration tool's environment.

Example 2.1. Sample Windows Environment Script

```
@echo off
set ACTIVEMQ_HOME=ActiveMQInstallDir
```

```
set ACTIVEMQ_BASE=BrokerInstanceDir
set PATH=%PATH%;%ACTIVEMQ_HOME%\bin

REM Configure Java
set JAVA_HOME=JavaInstallDir
set PATH=%JAVA_HOME%\bin;%PATH%

set ACTIVEMQ_CLASSPATH=%ACTIVEMQ_HOME%\lib\optional\activemq-optional-5.5.1-fuse-00-xx.jar

echo Setting Apache ActiveMQ 5.5 environment
```

This sample script sets `JAVA_HOME` as well as the Fuse Message Broker specific environment variables. In addition, it adds the `activemq-optional-5.5.1-fuse-00-xx.jar` JAR file to your Fuse Message Broker classpath so that the optional feature of the broker are available.

Setting up the Unix/Linux/OS X Environment

Overview

The Fuse Message Broker administrative tool on Unix and Unix-like platforms uses a number of environment variables. The environment variables are used to configure the broker's runtime environment and configure the behavior of the administration tool. The tool can also call one or more start up scripts to effect its execution.

In addition, it requires that your Java environment is properly set up.

Optionally, you may also want to add the administrative tool to the shell's path so that you can use the tool from anywhere.

Environment variables

Table 2.2 on page 16 describes the environment variables used by the administration tool on Unix and Unix-like platforms.

Table 2.2. Unix Administration Tool Environment Variables

Variable	Description
`ACTIVEMQ_HOME`	Specifies the directory where Fuse Message Broker is installed.
`ACTIVEMQ_CLASSPATH`	Specifies the classpath used by the current broker instance.
`ACTIVEMQ_SSL_OPTS`	Specifies Java properties required to support SSL/TLS protocols. See *Security Guide* for details.
`ACTIVEMQ_CONFIG_DIR`	Specifies the location of the broker's configuration directory. Default is `$ACTIVEMQ_HOME/conf`.
`ACTIVEMQ_DATA_DIR`	Specifies the location of the broker's data directory. Default is `$ACTIVEMQ_HOME/data`.
`ACTIVEMQ_PIDFILE`	Specifies the location of the file that holds the broker's process ID(PID).
`ACTIVEMQ_USER`	Specifies the user as which the broker daemon runs. The user should be a user with non-root privileges. Default value is a blank string, which implies that the broker does *not* change user.
`ACTIVEMQ_OPTS_MEMORY`	Specifies the JVM memory configuration. Default value is `-Xms256M -Xmx256M`.

Variable	Description
`ACTIVEMQ_QUEUEMANAGERURL`	Specifies the default connection URL that is used with the **browse** task. Default value is `--amqurl tcp://localhost:61616`.
`ACTIVEMQ_KILL_MAXSECONDS`	Specifies how many seconds the **stop** task will wait for an orderly shutdown to complete before killing the broker using `SIGKILL`.

Tip

The default setting for `ACTIVEMQ_OPTS_MEMORY` is very low for systems that run under a heavy load. You will want to change this value to provide the broker with more memory.

Startup scripts

On Unix, the administration tool can find and source a *startup script* immediately before it executes the specified task. The startup script is a shell script, which is normally used to set some environment variables.

The administration tool looks for startup scripts in the following locations:

1. `/etc/default/activemq`
2. `/home/User/.activemqrc`

If both of these scripts exist, they will *both* be sourced by the administration tool, in the order shown.

Java environment

You must set the `JAVA_HOME` environment variable to the location of the Java installation you want to use for running Fuse Message Broker.

Chapter 3. Setting up and Accessing the Fuse Message Broker Web Console

Fuse Message Broker's Web console is an easy way to query the broker for JMX statistics, destination statuses, and information on any routes deployed in the broker.

The Fuse Message Broker Web console is an embedded console for administering your broker, its destination, and the routes deployed in the broker. Using the embedded console is an easy way to manage your broker with minimal configuration.

If you want more security, more reliability, or the ability to monitor master/slave clusters, you can deploy the Web console as a standalone application. It is easy to deploy into Tomcat or any standard Web container.

Using the Embedded Console

Overview

The Web console is loaded along with the broker and deployed into an embedded Jetty container. By default, the console is insecure and accessed on port `8161`.

The default configuration can easily be modified to add security and change the port number.

Accessing the console

The Fuse Message Broker Web console has two main areas:

- broker administration—`http://`*`hostName`*`:`*`portNum`*`/admin`
- route administration—`http://`*`hostName`*`:`*`portNum`*`/camel`

For example, to access the default broker administration console on your local machine, you would point your Web browser at `http://localhost:8161/admin`.

Changing the port

To change the port at which the embedded Web console is accessed, you need to edit the broker's configuration file to as follows:

1. Open the broker's configuration file in an XML, or text, editor.

   The configuration file is typically called `activemq.xml` and is located in the `conf` folder.

2. Determine if the configuration imports the Jetty configuration used by the Web consoles.

   The default configuration includes an `import` element that imports the Jetty configuration.

3. - If the broker configuration imports the Jetty configuration, open the Jetty configuration file in your text editor.
   - If the Jetty configuration is included in the broker's configuration file, then you can locate the required configuration in the broker's configuration.

4. Locate the `property` element whose `name` attribute is set to `connectors`.

   It will look similar to Example 3.1 on page 21.

Example 3.1. Jetty Connector Configuration

```
...
<property name="connectors">
  <list>
    <bean id="Connector" class="org.eclipse.jetty.server.nio.SelectChannelConnector">
      <property name="port" value="8161" />
    </bean>
  </list>
</property>
```

5. In the `property` element with the `name` attribute set to `port`, set the `value` attribute to the number of port at which you want to access the Web console.

The configuration shown in Example 3.2 on page 21 configures the Web console such that the routing console can be accessed from `http://localhost:8563/camel`.

Example 3.2. Configuring the Web Console's Port

```
...
<property name="connectors">
  <list>
    <bean id="Connector" class="org.eclipse.jetty.server.nio.SelectChannelConnector">
      <property name="port" value="8536" />
    </bean>
  </list>
</property>
```

Securing the console

The security for the Web console is provided by the Web container in which it is deployed. For the embedded instance of the Web console, you need to configure the embedded Jetty container's security by editing `conf/jetty.xml`.

See "Web Console Security" in *Security Guide* for details.

Deploying a Standalone Console

Overview

For more security or reliability reasons you can deploy the Web console as a standalone application in Tomcat or another Web container. When running as a standalone application, the Web console can be set up to monitor Master/Slave clusters.

Disabling the embedded console

There is no need to disable the embedded console when using a standalone console. However, If you are using a standalone Web console, there is no reason to use the resources required by the embedded console. Nor is there a reason to leave an extra administrative access point open.

To disable the embedded Web console, you simply need to comment out, or remove, the `import` element that imports the Jetty configuration into your broker's configuration file as shown in Example 3.3 on page 22.

Example 3.3. Disabling the Embedded Web Console

```
<beans  ... >

  <broker ... >
     ...
  </broker>

  <!-- <import resource="jetty.xml"/>  -->

</beans>
```

Configuring Tomcat

To deploy the Web console in Tomcat 5.x:

1. Download the Web console's WAR, `activemq-web-console-5.5.1-fuse-00-xx.war`, from http://repo.fusesource.com/nexus/content/groups/m2-release-proxy/org/apache/activemq/activemq-web-console/5.5.1-fuse-00-xx/.
2. Copy the Web console's WAR to the *`TOMCAT_HOME`*`/webapps` folder.
3. Download `activemq-all-5.5.1-fuse-00-xx.jar` from http://repo.fusesource.com/nexus/content/groups/m2-release-proxy/org/apache/activemq/activemq-all/5.5.1-fuse-00-xx/.
4. Copy `activemq-all-5.5.1-fuse-00-xx.jar` to the *`TOMCAT_HOME`*`/common/lib` folder.

5. Modify *TOMCAT_HOME*/bin/catalina.sh(.bat) to include the configuration in Example 3.4 on page 23.

Example 3.4. Configuration for Deploying the Web Console in Tomcat

```
JAVA_OPTS="-Dwebconsole.type=properties \
-Dwebconsole.jms.url=brokerURL \
-Dwebconsole.jmx.url=brokerJMXURL \
-Dwebconsole.jmx.user=JMXUserName \
-Dwebconsole.jmx.password=JMXPassword"
```

6. Restart Tomcat.

The Web console will be available at *tomcatURI*/activemq-web-console-5.5.1-fuse-00-xx.

Monitoring clusters

It's possible to configure the Web console to monitor a master/slave cluster. To do so:

- Specify the JMS URL, `webconsole.jms.url`, with a `failover:` URI specifying the brokers in the cluster.
- Specify the JMX URL, `webconsole.jmx.url` as a comma separated list that contains the JMX URL for each of the brokers in the cluster.

Example 3.5 on page 23 shows the properties for monitoring a cluster using the Web console.

Example 3.5. Configuration for Monitoring a Cluster with the Web Console

```
-Dwebconsole.jms.url=failover:(tcp://serverA:61616,tcp://serverB:61616)
-Dwebconsole.jmx.url=service:jmx:rmi:///jndi/rmi://serverA:1099/jmxrmi,ser
vice:jmx:rmi:///jndi/rmi://serverB:1099/jmxrmi
```

For more information about master/slave clusters see "Master/Slave" in *Fault Tolerant Messaging*.

Chapter 4. Installing Fuse Message Broker as a Windows Service

Fuse Message Broker ships with wrapper that simplifies the process of installing it as a service on a machine running Windows.

Fuse Message Broker uses a wrapper, based on the Java service wrapper[1] from Tanuki Software Ltd., that provides a solution to the problem of launching a Java application as a Windows service. The wrapper provides a mechanism for specifying all of the required JVM and application specific parameters to the Windows' service interface.

The wrapper is located in the `bin/win32` folder of a Windows Fuse Message Broker installation. This folder contains the following:

- `activemq.bat`—script that starts the broker in the foreground
- `InstallService.bat`—script that installs the broker as a Windows service
- `UninstallService.bat`—script that uninstalls the broker from the service registry
- `wrapper.conf`—file that configures the wrapper service
- `wrapper.dll`
- `wrapper.exe`—wrapper service executable

Note

FuseSource does support the 64 bit version of the service wrapper. However, due to licensing concerns, you must obtain it directly from Tanuki Software Ltd.[2].

Before using the wrapper to install Fuse Message Broker as a Windows service, you will want to check the wrappers configuration to ensure it is set up properly.

[1] http://www.tanukisoftware.com/en/wrapper.php
[2] http://www.tanukisoftware.com

Configuring the Wrapper

Overview

The wrapper is configured by the `wrapper.conf` file, which is located under the *`InstallDir`*`/bin/win32/` directory.

Specifying the Fuse Message Broker's environment

A broker's environment is controlled by two environment variables:

- `ACTIVEMQ_HOME`—the location of the Fuse Message Broker install directory.
- `ACTIVEMQ_BASE`—the root directory containing the configuration, logging, and persistence data specific to the broker instance.

  The configuration for the broker instance is stored in the *`ACTIVEMQ_BASE`*`/conf` directory. The logging and persistence data is stored in the *`ACTIVEMQ_BASE`*`/data` directory.

It is likely that you will need to customize the values of the `ACTIVEMQ_HOME` and `ACTIVEMQ_BASE` properties in the `wrapper.conf` file. Example 4.1 on page 26 shows the default values.

Example 4.1. Default Environment Settings

```
set.default.ACTIVEMQ_HOME=../..
set.default.ACTIVEMQ_BASE=../..
```

Important

When specifying file and directory paths in `wrapper.conf`, the forward slash, `/`, is used.

Passing parameters to the JVM

If you want to pass parameters to the JVM, you do so by setting wrapper properties using the form `wrapper.java.additional.`*`<n>`*. *`<n>`* is a sequence number that must be distinct for each parameter.

Setting Java system properties

One of the most useful things you can do by passing additional parameters to the JVM is to set Java system properties. The syntax for setting a Java system property is `wrapper.java.additional.`*`<n>`*`=-D`*`PropName=PropValue`*.

Example 4.2 on page 27 shows the default Java properties.

Example 4.2. Default Java System Properties

```
# Java Additional Parameters
# note that n is the parameter number starting from 1.
wrapper.java.additional.1=-Dactivemq.home="%ACTIVEMQ_HOME%"
wrapper.java.additional.2=-Dactivemq.base="%ACTIVEMQ_BASE%"
wrapper.java.additional.3=-Djavax.net.ssl.keyStorePassword=password
wrapper.java.additional.4=-Djavax.net.ssl.trustStorePassword=password
wrapper.java.additional.5=-Djavax.net.ssl.keyStore="%ACTIVEMQ_BASE%/conf/broker.ks"
wrapper.java.additional.6=-Djavax.net.ssl.trustStore="%ACTIVEMQ_BASE%/conf/broker.ts"
wrapper.java.additional.7=-Dcom.sun.management.jmxremote
wrapper.java.additional.8=-Dorg.apache.activemq.UseDedicatedTaskRunner=true
wrapper.java.additional.9=-Djava.util.logging.config.file=logging.properties
```

Setting application parameters

You specify application parameters using the syntax `wrapper.app.parameter.<n>`. <*n*> is a sequence number that must follow the pattern:

- `1`—specifies the name of the Java class executed by the wrapper
- `>=2`—specify properties passed to the application

Example 4.3 on page 27 shows the default settings.

Example 4.3. Default Wrapper Application Settings

```
# Application parameters.  Add parameters as needed starting from 1
wrapper.app.parameter.1=org.apache.activemq.console.Main
wrapper.app.parameter.2=start
```

Adding classpath entries

You add classpath entries using the syntax `wrapper.java.classpath.<n>`. <*n*> is a sequence number that must be distinct for each classpath entry.

Example 4.4 on page 27 shows the default classpath entries.

Example 4.4. Default Wrapper Classpath

```
# Java Classpath (include wrapper.jar)  Add class path elements as
#  needed starting from 1
wrapper.java.classpath.1=%ACTIVEMQ_HOME%/bin/wrapper.jar
wrapper.java.classpath.2=%ACTIVEMQ_HOME%/bin/run.jar
```

Reference

For a complete description of all the properties you can set in the `wrapper.conf` configuration file, see Configuration Property Overview[3] at the Tanuki Web site.

[3] http://wrapper.tanukisoftware.com/doc/english/properties.html

Installing and Starting the Windows Service

Overview

Fuse Message Broker provides the `InstallService.bat` and `UninstallService.bat` scripts to install and uninstall the broker as a Windows service.

Installing the broker as a service

After customizing the `wrapper.conf` configuration file as described in "Configuring the Wrapper" on page 26, change directory to *`InstallDir`*`\bin\win32`, and install the broker as a Windows service, using `InstallService.bat`, as follows:

```
C:\apache-activemq5.5.1-fuse-00-xx\bin\win32> InstallService
wrapper  | ActiveMQ installed.
```

Once installed the Fuse Message Broker service will automatically start up when Windows is launched.

Starting the broker manually

You can manually start the installed Fuse Message Broker service using the Windows **net start** tool. The service is registered under the name `ActiveMQ`.

You start the service as follows:

```
c:\> net start ActiveMQ
The ActiveMQ service is starting.........
The ActiveMQ service was started successfully.
```

Stopping the broker manually

You can manually stop the installed Fuse Message Broker service using the Windows **net stop** tool. The service is registered under the name `ActiveMQ`.

You stop the service as follows:

```
c:\> net stop ActiveMQ
The ActiveMQ service is stopping...
The ActiveMQ service was stopped successfully.
```

Uninstalling the broker

To remove the broker from the Windows service registry, run **UninstallService.bat** as follows:

```
C:\apache-activemq5.5.1-fuse-00-xx\bin\win32> UninstallService
wrapper  | ActiveMQ removed.
```

Chapter 5. Starting a Broker

You can start a broker from the command line using either the administration tool or Maven. You specify the configuration for the started broker using a configuration URI.

Specifying the Broker's Configuration

Overview

There are three ways to pass configuration to a broker instance at start-up:

- XBean URI—specifies the location a Spring XML configuration file
- Broker URI—specifies the broker configuration as part of the URI
- Properties URI—specifies the location of properties file containing Fuse Message Broker configuration

XBean configuration

XBean configuration uses a Fuse Message Broker XML configuration file to configure the broker. You pass the location of the configuration file to the administrative tool using a URI prefixed with `xbean:`.

There are three ways to specify the location of the XML configuration file using an XBean URI:

- on the Java classpath using the syntax `xbean:`*`fileName`*

  For example the XBean URI `xbean:activemq.xml` specifies the file `activemq.xml` that is located somewhere on the Java classpath.

- on the file system using the syntax `xbean:file:`*`filePath`*

  For example the XBean URI `xbean:file:activemq.xml` specifies the file `activemq.xml` that is located in the current directory.

Tip

You can specify both relative and absolute paths to the file.

- on a resource path using the syntax `xbean:`*`resourcePath`*

For more information on the Fuse Message Broker XML configuration see the XML Configuration Reference[1].

[1] https://access.redhat.com/site/documentation/en-US/Fuse_Message_Broker/5.5/pdf/xmlref/index.html

Broker URI

You can specify a broker's configuration entirely on the command line using a broker URI of the form shown in Example 5.1 on page 33.

Example 5.1. Broker URI Syntax

```
broker:(transportURI,[network:networkURI])/[brokerName]?brokerOptions
```

- *`transportURI`*—a comma separated list of URIs at which the broker listens for client connections

  See *Connectivity Guide* for more information.

- *`networkURI`*—a comma separated list of URIs at which the broker listens for connections from other brokers

  See "Discovering Brokers" in *Using Networks of Brokers* for more information.

- *`brokerName`*—the broker's name

- *`brokerOptions`*—an ampersand(*&*) separated list configuration properties

  See Table C.1 on page 93 for a description of the properties.

For example, the URI shown in Example 5.2 on page 33 starts up a broker that accepts connections on port 61616, establishes a network connection to `remotehost:61616`, and disables persistence.

Example 5.2. Broker URI

```
broker:(tcp://localhost:61616,network:static:tcp://remotehost:61616)?persistent=false&use
Jmx=true
```

Property configuration

Property configuration uses a Java properties file to configure the broker. You pass the location of the configuration file to the administrative tool using a URI prefixed with `properties:`. As with the XBean URI, you can specify a location on the classpath, on the local file system, or using a resource path. For example the URI `properties:file:activemq.properties` specifies the file `activemq.properties` that is located in the current directory.

The properties file shown in Example 5.3 on page 34 starts up a broker that accepts connections on port 61616, is named Cheese, and disables persistence and JMX.

Example 5.3. Broker Properties File

```
useJmx = false
persistent = false
brokerName = Cheese
```

See Appendix C on page 93 for a description of all of the available properties.

Starting a Broker on Windows

Overview

On Windows the command to start a Fuse Message Broker broker is simple. It allows you to start a broker instance as a foreground process using the specified configuration.

The Unix start command provides more options. See "Starting a Broker on Unix/Linux/OS X" on page 36.

Starting a foreground broker

On Windows, the **activemq** tool to starts an instance of Fuse Message Broker in the foreground. The syntax for the command is shown in Example 5.4 on page 35.

Example 5.4. Syntax for Starting a Broker on Windows

`activemq` {[xbean:file:*`confURI`* [?validate= {[true] | [false]}]] | [broker:*`brokerURI`*] | [properties:*`propURI`*]}

Tip

For a full description of the parameters see activemq on page 72.

For example, to start a broker using the default broker configuration, enter:

```
c:\fmq> activemq xbean:file:conf/activemq.xml
```

You can optionally disable schema validation of the configuration file using the `validate` flag, as follows:

```
c:\fmq> activemq xbean:file:conf/activemq.xml?validate=false
```

Starting a background broker

On Windows the **activemq** command does not provide a mechanism for launching a broker instance as a background process. To start a broker in the background you can either:

- install the broker instance as a Windows service. See "Installing Fuse Message Broker as a Windows Service" on page 25.
- Use the Windows **start** command to spawn the broker in a new command window.

Starting a Broker on Unix/Linux/OS X

Overview

The command to start a broker on Unix and Unix-like platforms provides more flexibility than the Windows start-up command. On Unix, you can start a broker in either the foreground or in the background as a daemon process. In addition, the Unix start-up command can run pre-start scripts and associate an effective user to the broker process.

Starting a background broker

On Unix, the **activemq start** command to starts an instance of Fuse Message Broker in the background. The syntax for the command is shown in Example 5.5 on page 36.

Example 5.5. Syntax for Starting a Daemon Broker on Unix

`activemq start` {[xbean:file:*`confURI`* [?validate= {[true] | [false]}]] | [broker:*`brokerURI`*] | [properties:*`propURI`*]}

Tip

For a full description of the parameters see activemq start on page 74.

For example, to start a daemon broker using the default broker configuration, enter:

```
% activemq start xbean:file:conf/activemq.xml
```

You can optionally disable schema validation of the configuration file using the `validate` flag, as follows:

```
% activemq start xbean:file:conf/activemq.xml?validate=false
```

By default, the daemon broker runs as the current user. It is good practice to specify a special effective user for the broker. This special effective user should be given just enough privileges to run the broker. This provides added security by ensuring that the broker process has limited ability to corrupt sensitive part of your system.

The `ACTIVEMQ_USER` environment variable specifies the effective user for the broker instances. By default, the broker will run as the currently logged in user. See "Setting up the Unix/Linux/OS X Environment" on page 16.

Starting a broker in the foreground

On Unix, the **activemq console** command starts an instance of Fuse Message Broker as a foreground process. The syntax for the command is shown in Example 5.6 on page 37.

Example 5.6. Syntax for Starting a Foreground Broker on Unix

`activemq console` {[xbean:file:*`confURI`* [?validate= {[true] | [false]}]] | [broker:*`brokerURI`*] | [properties:*`propURI`*]}

Tip

For a full description of the parameters see activemq console on page 75.

For example, you can start a broker in the foreground using the default broker configuration, `conf/activemq.xml`, as follows:

```
% activemq console xbean:conf/activemq.xml
```

Starting a Broker with Maven

Overview

When developing messaging applications it can be helpful to have a test broker start up automatically when running the application. This can be accomplished using Fuse Message Broker's Apache Maven plug-in. The plug-in can be used directly from the command line or it can be included as part of a project's POM.

The Maven plug-in treats the broker as a part of the project's build and test environment. Maven will download all of the Fuse Message Broker libraries from the configured Maven repositories. The plug-in will store all of the broker's data in the project's `target` folder.

Note

Starting broker in a production environment from Maven is not recommended.

Using the command line

If a basic broker will suffice for testing an application, the Fuse Message Broker Maven plug-in can be loaded from the command line as shown in Example 5.7 on page 38.

Example 5.7. Using the Maven Plug-in from the Command Line

```
mvn org.apache.activemq.tooling:maven-activemq-plugin:5.5.1-fuse-00-xx:run
```

This will create a broker that:

- listens for client connections on `tcp://localhost:61616`
- does not use JMX
- does not persist messages

Using the POM

If the basic broker provided by adding the plug-in on the command line is insufficient for test an application the plug-in can be included in the project's POM. When included in a POM, the Fuse Message Broker plug-in uses a configuration URI to configure the broker. The plug-in can also set system properties when starting the broker.

Table 5.1 on page 39 describes the configuration properties used by the Fuse Message Broker Maven plug-in.

Table 5.1. Maven Configuration Properties

Property	Default	Description
`configUri`	`broker:(tcp://localhost:61616)?useJmx=false&persistent=false`	Specifies the configuration URI used to configure the broker. See "Specifying the Broker's Configuration" on page 32 for more information.
`fork`	false	Specifies whether the broker is started in a separate thread. Having the broker start in a separate thread can be useful for integration testing.
`systemProperties`		Specifies a collection of system properties that will be set.

To start the configured broker when running the project use **mvn activemq:run**.

Example 5.8 on page 39 shows a POM fragment that configures the Fuse Message Broker plug-in to start a broker using the `activemq.xml` configuration file on the project's class path.

Example 5.8. Using the Maven Plug-in from a POM

```
<project ... >

  ...

  <repositories>
  <!-- FuseSource maven repositories -->
    <repository>
      <id>fusesource.releases</id>
      <name>FuseSoure releases repository</name>
      <url>http://repo.fusesource.com/maven2/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
    <repository>
      <id>fusesource.snapshots</id>
      <name>FuseSource Snapshot Repository</name>
      <url>http://repo.fusesource.com/maven2-snapshot</url>
      <snapshots>
```

```
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>

<pluginRepositories>
<!-- FuseSource maven repositories -->
  <pluginRepository>
    <id>fusesource.releases</id>
    <name>FuseSoure releases repository</name>
    <url>http://repo.fusesource.com/maven2/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>fusesource.snapshots</id>
    <name>FuseSource Snapshot Repository</name>
    <url>http://repo.fusesource.com/maven2-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>

...

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.activemq.tooling</groupId>
      <artifactId>maven-activemq-plugin</artifactId>
      <version>5.5</version>
      <configuration>
        <configUri>xbean:activemq.xml</configUri>
        <fork>false</fork>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>org.springframework</groupId>
```

```
            <artifactId>spring</artifactId>
            <version>2.5.5</version>
          </dependency>
          <dependency>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>jetty-xbean</artifactId>
            <version>6.1.11</version>
          </dependency>
          <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-activemq</artifactId>
            <version>1.1.0</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </build>
</project>
```

Chapter 6. Shutting Down a Broker

The Fuse Message Broker's shutdown tool uses JMX to locate and gracefully shutdown brokers running as daemon processes.

Important

The administration tool requires that brokers have JMX enabled. By default, JMX is enabled. For more information see "Using JMX" on page 53.

Shutting Down a Broker on Windows

Overview

On Windows platforms brokers are run as either foreground processes or as system services. You can shutdown running brokers using operating system actions. For example, typing **CTRL**+**C** in a window in which a foreground broker is running will shutdown the broker. Closing the window will also shutdown a foreground broker.

However, if you want to properly shutdown a broker use the administration tool's **stop** task. It ensures that the broker is gracefully shutdown. It also allows you to shutdown brokers remotely.

The stop task

On Windows, the command to shutdown a broker is **activemq-admin stop**. The syntax for the command is shown in Example 6.1 on page 44.

Example 6.1. Syntax for Stopping a Broker on Windows

`activemq-admin stop` {[*`brokerName`*] | [--all]} [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*]

Tip

For a full description of the parameters see activemq-admin stop on page 81.

If you do not use the *`--jmxurl`* parameter, the default JMX url, `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi`, is used. This is the JMX url specified in the default Fuse Message Broker configuration.

If you configured a broker to use a different JMX url, you must use the *`--jmxurl`* parameter and provide the JMX url for connecting to the broker.

Tip

If you have a broker running as a Windows service and have JMX turned on, you can use **activemq-admin stop** to shut it down. However, the recommended way to shut down a broker running as a Windows service is documented in "Stopping the broker manually" on page 29.

If you have secured JMX access to the broker, you will need to use the *-jmxuser* and *-jmxpassword* parameters. They specify the user name and password required to access the broker's JMX context. For information about using security with JMX see "JMX Security" in *Security Guide*.

Examples

The command shown in Example 6.2 on page 45 shuts down a broker named `gatewayEast` that is running in the default JMX context.

Example 6.2. Shutting Down a Broker on Windows

```
c:\activemq-admin stop gatewayEast
```

The command shown in Example 6.3 on page 45 shuts down all of the brokers running in the default JMX context.

Example 6.3. Shutting Down All Broker on Windows

```
c:\activemq-admin stop --all
```

The command shown in Example 6.4 on page 45 shuts down a broker named `gatewayEast` that is running in the JMX context `service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi`.

Example 6.4. Shutting Down a Broker on Windows in a Non-default JMX Context

```
c:\activemq-admin stop gatewayEast --jmxurl service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi
```

Shutting Down a Broker on Unix/Linux/OS X

Overview

The administration tool's **stop** task uses JMX to locate and shutdown brokers on Unix and Unix-like platforms. This saves you from remembering a broker's PID each time it starts up.

The Unix **stop** task can also be configured to force a shutdown if an orderly shutdown hangs. It does this by waiting a predetermined amount of time and then killing the broker process.

The stop task

On Unix and Unix-like systems, the command to shutdown a broker is **activemq stop**. The syntax for the command is shown in Example 6.5 on page 46.

Example 6.5. Syntax for Stopping a Broker on Unix/Linux/OS X

`activemq stop` {[*brokerName*] | [--all]} [--jmxurl *JMXUrl*] [-jmxuser *userName*] [-jmxpassword *password*]

Tip

For a full description of the parameters see activemq stop on page 78.

If you do not use the `--jmxurl` parameter, the default JMX url, `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi`, is used. This is the JMX url specified in the default Fuse Message Broker configuration.

If you configured a broker to use a different JMX url, you must use the `--jmxurl` parameter and provide the JMX url for connecting to the broker.

If you have secured JMX access to the broker, you will need to use the `-jmxuser` and `-jmxpassword` parameters. They specify the user name and password required to access the broker's JMX context. For information about using security with JMX see "JMX Security" in *Security Guide*.

Shutdown timeout

When you execute the **stop** task, the administration tool initiates an orderly shutdown of the broker (or brokers) by sending a shutdown request to the JMX port. If the broker fails to shut down within a specified timeout, the administration tool sends a `SIGKILL` signal to the broker process.

The duration of the kill timeout is set by the `ACTIVEMQ_KILL_MAXSECONDS` environment variable. For more information about setting up your environment for the administration tool see "Setting up the Unix/Linux/OS X Environment" on page 16.

Examples

The command shown in Example 6.6 on page 47 shuts down a broker named `gatewayEast` that is running in the default JMX context.

Example 6.6. Shutting Down a Broker on Unix/Linux/OS X

```
>activemq stop gatewayEast
```

The command shown in Example 6.7 on page 47 shuts down all of the brokers running in the default JMX context.

Example 6.7. Shutting Down All Broker on Unix/Linux/OS X

```
>activemq stop --all
```

The command shown in Example 6.8 on page 47 shuts down a broker named `gatewayEast` that is running in the JMX context `service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi`.

Example 6.8. Shutting Down a Broker in a Non-default JMX Context on Unix/Linux/OS X

```
>activemq stop gatewayEast --jmxurl service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi
```

Chapter 7. Using the Advisory Topics

Fuse Message Broker generates messages that advise the administrator of events that occur in the broker. These advisories are sent to special topics that you should monitor.

Chapter 8. Using the Log

The broker's log contains information about all of the critical events that occur in the broker. You can configure the granularity of the logged messages to provide the level of detail that you require.

Chapter 9. Using JMX

Fuse Message Broker is fully instrumented to provide statistics about its performance using JMX. You can monitor a broker using any JMX aware monitoring tool.

By default Fuse Message Broker creates MBeans, loads them into the MBean server created by the JVM, and creates a dedicated JMX connector that provides a Fuse Message Broker-specific view of the MBean server. The default settings are sufficient for simple deployments and make it easy to access the statistics and management operations provided by a broker. For more complex deployments you easily configure many aspects of how a broker configures itself for access through JMX. For example, you can change the JMX URI of the JMX connector created by the broker or force the broker to use the generic JMX connector created by the JVM.

By connecting a JMX aware management and monitoring tool to a broker's JMX connector, you can view detailed information about the broker. This information provides a good indication of broker health and potential problem areas. In addition to the collected statistics, Fuse Message Broker's JMX interface provides a number of operations that make it easy to manage a broker instance. These include stopping a broker, starting and stopping network connectors, and managing destinations.

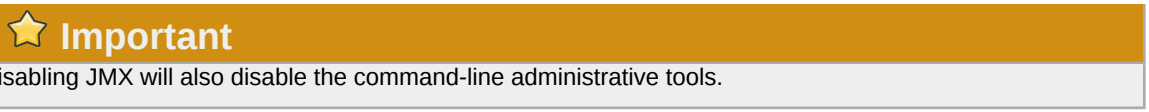
Configuring JMX

Overview

By default a broker is set up to allow for JMX management. It uses the JVM's MBean server and creates its own JMX connector at `service:jmx:rmi:///jndi/rmi://hostname:1099/jmxrmi`. If the default configuration does not meet the needs of the deployment environment, the broker provides configuration properties for customizing most aspects of its JMX behavior. For instance, you can completely disable JMX for a broker. You can also force the broker to create its own MBean server.

Enabling and disabling

By default JMX is enabled for a Fuse Message Broker broker. To disable JMX entirely you simply set the `broker` element's `useJmx` attribute to `false`. This will stop the broker from exposing itself via JMX.



Securing access to JMX

In a production environment it is advisable to secure the access to your brokers' management interfaces. The steps to secure access to a broker's JMX interface depends on the JMX connector the broker uses.

If the broker creates its own JMX connector you configure the security options directly in the broker's configuration. If the broker uses the JMX connector provided by the JVM, you need to modify the scripts used to start the broker to pass the security configuration to the JVM.

For a detailed description of the steps required see "JMX Security" in *Security Guide*.

Advanced configuration

If the default JMX behavior is not appropriate for your deployment environment, you can customize how the broker exposes its MBeans. To customize a broker's JMX configuration, you add a `managementContext` child element to the broker's `broker` element. The `managementContext` element uses a `managementContext` child to configure the broker. The attributes of the inner `managementContext` element specify the broker's JMX configuration.

Table 9.1 on page 55 describes the configuration properties for controlling a broker's JMX behavior.

Table 9.1. Broker JMX Configuration Properties

Property	Default Value	Description
`useMBeanServer`	`true`	Specifies whether the broker will use the MBean server created by the JVM. When set to `false`, the broker will create an MBean server.
`jmxDomainName`	`org.apache.activemq`	Specifies the JMX domain used by the broker's MBeans.
`createMBeanServer`	`true`	Specifies whether the broker creates an MBean server if none is found.
`createConnector`	`true`	Specifies whether the broker creates a JMX connector for the MBean server. If this is set to `false` the broker will only be accessible using the JMX connector created by the JVM.
`connectorPort`	`1099`	Specifies the port number used by the JMX connector created by the broker.
`connectorHost`	`localhost`	Specifies the host used by the JMX connector and the RMI server.
`rmiServerPort`	`0`	Specifies the RMI server port. This setting is useful if port usage needs to be restricted behind a firewall.
`connectorPath`	`/jmxrmi`	Specifies the path under which the JMX connector will be registered.

Example 9.1 on page 55 shows configuration for a broker that will only use the JVM's MBean server and will not create its own JMX connector.

Example 9.1. Configuring a Broker's JMX Connection

```
<broker ... >
  ...
  <managementContext>
    <managementContext createMBeanServer="false"
                       createConnector="false" />
  </managementContext>
  ...
</broker>
```

Statistics Collected by JMX

Broker statistics

Table 9.2 on page 56 describes the statistics collected for a broker.

Table 9.2. Broker JMX Statistics

Name	Description
BrokerId	Specifies the broker's unique ID.
BrokerName	Specifies the broker's name.
BrokerVersion	Specifies the version of the broker.
DataDirectory	Specifies the pathname of the broker's data directory.
TotalEnqueueCount	Specifies the total number of messages that have been sent to the broker.
TotalDequeueCount	Specifies the number of messages that have been acknowledged on the broker.
TotalConsumerCount	Specifies the number of message consumers subscribed to destinations on the broker.
TotalProducerCount	Specifies the number of message producers active on destinations on the broker.
TotalMessageCount	Specifies the number of unacknowledged messages on the broker.
MemoryLimit	Specifies the memory limit, in bytes, used for holding undelivered messages before paging to temporary storage.
MemoryPercentageUsed	Specifies the percentage of available memory in use.
StoreLimit	Specifies the disk space limit, in bytes, used for persistent messages before producers are blocked.
StorePercentageUsed	Specifies the percentage of the store space in use.
TempLimit	Specifies the disk space limit, in bytes, used for non-persistent messages and temporary data before producers are blocked.
TempPercentageUsed	Specifies the percentage of available temp space in use.

Destination statistics

Table 9.3 on page 57 describes the statistics collected for a destination.

Table 9.3. Destination JMX Statistics

Name	Description
BlockedProducerWarningInterval	Specifies, in milliseconds, the interval between warnings issued when a producer is blocked from adding messages to the destination.
MemoryLimit	Specifies the memory limit, in bytes, used for holding undelivered messages before paging to temporary storage.
MemoryPercentageUsed	Specifies the percentage of available memory in use.
MaxPageSize	Specifies the maximum number of messages that can be paged into the destination.
CursorFull	Specifies if the cursor has reached its memory limit for paged messages.
CursorMemoryUsage	Specifies, in bytes, the amount of memory the cursor is using.
CursorPercentUsage	Specifies the percentage of the cursor's available memory is in use.
EnqueueCount	Specifies the number of messages that have been sent to the destination.
DequeueCount	Specifies the number of messages that have been acknowledged and removed from the destination.
DispatchCount	Specifies the number of messages that have been delivered to consumers, but not necessarily acknowledged by the consumer.
InFlightCount	Specifies the number of dispatched to, but not acknowledged by, consumers.
ExpiredCount	Specifies the number of messages that have expired in the destination.
ConsumerCount	Specifies the number of consumers that are subscribed to the destination.
QueueSize	Specifies the number of messages in the destination that are waiting to be consumed.

Name	Description
AverageEnqueueTime	Specifies the average amount of time, in milliseconds, that messages sat in the destination before being consumed.
MaxEnqueueTime	Specifies the longest amount of time, in milliseconds, that a message sat in the destination before being consumed.
MinEnqueueTime	Specifies the shortest amount of time, in milliseconds, that a message sat in the destination before being consumed.
MemoryUsagePortion	Specifies the portion of the broker's memory limit used by the destination.
ProducerCount	Specifies the number of producers connected to the destination.

Subscription statistics

Table 9.4 on page 58 describes the statistics collected for a subscription.

Table 9.4. Connection JMX Statistics

Name	Description
EnqueueCounter	Counts the number of messages that matched the subscription.
DequeueCounter	Counts the number of messages were sent to and acknowledge by the client.
DispatchedQueueSize	Specifies the number of messages dispatched to the client and are awaiting acknowledgement.
DispatchedCounter	Counts the number of messages that have been sent to the client.
MessageCountAwaitingAcknowledge	Specifies the number of messages dispatched to the client and are awaiting acknowledgement.
Active	Specifies if the subscription is active.
PendingQueueSize	Specifies the number of messages pending delivery.
PrefetchSize	Specifies the number of messages to pre-fetch and dispatch to the client.

Name	Description
MaximumPendingMessageLimit	Specifies the maximum number of pending messages allowed.

Managing the Broker with JMX

Overview

The MBeans exposed by Fuse Message Broker provide a number of operations for monitoring and managing a broker instance. You can access these operations through any tool that supports JMX.

Broker actions

Table 9.5 on page 60 describes the operations exposed by the MBean for a broker.

Table 9.5. Broker MBean Operations

Operation	Description
`void start();`	Starts the broker. In reality this operation is not useful because you cannot access the MBeans if the broker is stopped.
`void stop();`	Forces a broker to shut down. There is no guarantee that all messages will be properly recorded in the persistent store.
`void stopGracefully(String queueName);`	Checks that all listed queues are empty before shutting down the broker.
`void enableStatistics();`	Activates the broker's statistics plug-in.
`void resetStatistics();`	Resets the data collected by the statistics plug-in.
`void disableStatistics();`	Deactivates the broker's statistics plug-in.
`String addConnector(String URI);`	Adds a transport connector to the broker and starts it listening for incoming client connections and returns the name of the connector.
`boolean removeConnector(String connectorName);`	Deactivates the specified transport connector and removes it from the broker.
`String addNetworkConnector(String URI);`	Adds a network connector to the specified broker and returns the name of the connector.
`boolean removeNetworkConnector(String connectorName);`	Deactivates the specified connector and removes it from the broker.
`void addTopic(String name);`	Adds a topic destination to the broker.
`void addQueue(String name);`	Adds a queue destination to the broker.

Operation	Description
`void removeTopic(String name);`	Removes the specified topic destination from the broker.
`void removeQueue(String name);`	Removes the specified queue destination from the broker.
`ObjectName createDurableSubscriber(String clientId, String subscriberId, String topicName, String selector);`	Creates a new durable subscriber.
`void destroyDurableSubscriber(String clientId, String subscriberId);`	Destroys a durable subscriber.
`void gc();`	Runs the JVM garbage cleaner.
`void terminateJVM(int exitCode);`	Shuts down the JVM.
`void reloadLog4jProperties();`	Reloads the logging configuration from `log4j.properties`.

Connector actions

Table 9.6 on page 61 describes the operations exposed by the MBean for a transport connector.

Table 9.6. Connector MBean Operations

Operation	Description
`void start();`	Starts the transport connector so that it is ready to receive connections from clients.
`void stop();`	Closes the transport connection and disconnects all connected clients.
`int connectionCount();`	Returns the number of open connections using the connector.
`void enableStatistics();`	Enables statistics collection for the connector.
`void resetStatistics();`	Resets the statistics collected for the connector.
`void disableStatistics();`	Deactivates the collection of statistics for the connector.

Network connector actions

Table 9.7 on page 62 describes the operations exposed by the MBean for a network connector.

Table 9.7. Network Connector MBean Operations

Operation	Description
`void start();`	Starts the network connector so that it is ready to communicate with other brokers in a network of brokers.
`void stop();`	Closes the network connection and disconnects the broker from any brokers that used the network connector to form a network of brokers.

Queue actions

Table 9.8 on page 62 describes the operations exposed by the MBean for a queue destination.

Table 9.8. Queue MBean Operations

Operation	Description
`CompositeData getMessage(String messageId);`	Returns the specified message from the queue without moving the message cursor.
`void purge();`	Deletes all of the messages from the queue.
`boolean removeMessage(String messageId);`	Deletes the specified message from the queue.
`int removeMatchingMessages(String selector);`	Deletes the messages matching the selector from the queue and returns the number of messages deleted.
`int removeMatchingMessages(String selector, int maxMessages);`	Deletes up to the maximum number of messages that match the selector and returns the number of messages deleted.
`boolean copyMessageTo(String messageId, String destination);`	Copies the specified message to a new destination.
`int copyMatchingMessagesTo(String selector, String destination);`	Copies the messages matching the selector and returns the number of messages copied.

Operation	Description
`int copyMatchingMessagesTo(String selector, String destination, int maxMessages);`	Copies up to the maximum number of messages that match the selector and returns the number of messages copied.
`boolean moveMessageTo(String messageId, String destination);`	Moves the specified message to a new destination.
`int moveMatchingMessagesTo(String selector, String destination);`	Moves the messages matching the selector and returns the number of messages moved.
`int moveMatchingMessagesTo(String selector, String destination, int maxMessages);`	Moves up to the maximum number of messages that match the selector and returns the number of messages moved.
`boolean retryMessage(String messageId);`	Moves the specified message back to its original destination.
`int cursorSize();`	Returns the number of messages available to be paged in by the cursor.
`boolean doesCursorHaveMessagesBuffered();`	Returns `true` if the cursor has buffered messages to be delivered.
`boolean doesCursorHaveSpace();`	Returns `true` if the cursor has memory space available.
`CompositeData[] browse();`	Returns all messages in the queue, without changing the cursor, as an array.
`CompositeData[] browse(String selector);`	Returns all messages in the queue that match the selector, without changing the cursor, as an array.
`TabularData browseAsTable(String selector);`	Returns all messages in the queue that match the selector, without changing the cursor, as a table.
`TabularData browseAsTable();`	Returns all messages in the queue, without changing the cursor, as a table.
`void resetStatistics();`	Resets the statistics collected for the queue.
`java.util.List browseMessages(String selector);`	Returns all messages in the queue that match the selector, without changing the cursor, as a list.
`java.util.List browseMessages();`	Returns all messages in the queue, without changing the cursor, as a list.
`String sendTextMessage(String body, String username, String password);`	Send a text message to a secure queue.

Operation	Description
`String sendTextMessage(String body);`	Send a text message to a queue.

Topic actions

Table 9.9 on page 64 describes the operations exposed by the MBean for a topic destination.

Table 9.9. Topic MBean Operations

Operation	Description
`CompositeData[] browse();`	Returns all messages in the topic as an array.
`CompositeData[] browse(String selector);`	Returns all messages in the topic that match the selector as an array.
`TabularData browseAsTable(String selector);`	Returns all messages in the topic that match the selector as a table.
`TabularData browseAsTable();`	Returns all messages in the topic as a table.
`void resetStatistics();`	Resets the statistics collected for the queue.
`java.util.List browseMessages(String selector);`	Returns all messages in the topic that match the selector as a list.
`java.util.List browseMessages();`	Returns all messages in the topic as a list.
`String sendTextMessage(String body, String username, String password);`	Send a text message to a secure topic.
`String sendTextMessage(String body);`	Send a text message to a topic.

Subscription actions

Table 9.10 on page 64 describes the operations exposed by the MBean for a durable subscription.

Table 9.10. Subscription MBean Operations

Operation	Description
`void destroy();`	Destroys the subscription.

Operation	Description
`CompositeData[] browse();`	Returns all messages waiting for the subscriber.
`TabularData browseAsTable();`	Returns all messages waiting for the subscriber.
`int cursorSize();`	Returns the number of messages available to be paged in by the cursor.
`boolean doesCursorHaveMessagesBuffered();`	Returns `true` if the cursor has buffered messages to be delivered.
`boolean doesCursorHaveSpace();`	Returns `true` if the cursor has memory space available.
`boolean isMatchingQueue(String queueName);`	Returns `true` if this subscription matches the given queue name.
`boolean isMatchingTopic(String topicName);`	Returns `true` if this subscription matches the given topic name.

Chapter 10. Using the Statistics Plug-in

Fuse Message Broker is fully instrumented to provide statistics about its performance using JMX. You can monitor a broker using any JMX aware monitoring tool.

Appendix A. Common Problems

Appendix B. Administration Tool Commands

Important

The administration tool requires that brokers have JMX enabled. By default, JMX is enabled. For more information see "Using JMX" on page 53.

Name

activemq — starts a foreground broker on Windows

Synopsis

`activemq` [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]] {[xbean:file:*confURI* [?validate= {[true] | [false]}]] | [broker:*brokerURI*] | [properties:*propURI*]}

Arguments

Argument	Description
-D*name*=*value*	Specifies a system property to be interpreted by the VM.
--version	Displays the version of the tool.
--help, -?, -h	Displays the help for the command.
xbean:file:*confURI*	Specifies the path of the broker configuration file to use.
broker:*brokerURI*	Specifies the broker URI to use.
properties:*propURI*	Specifies the path to a properties file to configure the broker.

Examples

The following command starts a broker with the default configuration using an XBean URI:

```
c:>activemq xbean:activemq.xml
```

The following command starts a broker using a broker URI:

```
c:>activemq broker:(tcp://localhost:61616,network:static:tcp://remotehost:61616)?persistent=false&useJmx=true
```

Name

activemq setup — creates default configurations for the **activemq** command on Unix, and Unix-like, platforms

Synopsis

`activemq setup` [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]] {*refFile*}

Arguments

Argument	Description
-D*name*=*value*	Specifies a system property to be interpreted by the VM.
--version	Displays the version of the tool.
--help, -?, -h	Displays the help for the command.
confFile	Specifies the path of the broker configuration file to use.

Examples

The following command puts the default **activemq** configuration in `~/.activemqrc`:

```
%activemq setup ~/.atcivemqrc
```

Name

activemq start — starts a daemon broker on Unix, and Unix-like, platforms

Synopsis

`activemq start` [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]] {[xbean:file:*confURI* [?validate= {[true] | [false]}]] | [broker:*brokerURI*] | [properties:*propURI*]}

Arguments

Argument	Description
-D*name*=*value*	Specifies a system property to be interpreted by the VM.
--version	Displays the version of the tool.
--help, -?, -h	Displays the help for the command.
xbean:file:*confURI*	Specifies the path of the broker configuration file to use.
broker:*brokerURI*	Specifies the broker URI to use.
properties:*propURI*	Specifies the path to a properties file to configure the broker.

Examples

The following command starts a broker with the default configuration using an XBean URI:

```
%activemq start xbean:activemq.xml
```

The following command starts a broker using a broker URI:

```
%activemq start broker:(tcp://localhost:61616,network:static:tcp://remotehost:61616)?persist
ent=false&useJmx=true
```

Name

activemq console — starts a foreground broker on Unix, and Unix-like, platforms

Synopsis

`activemq console` [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]] {[xbean:file:*confURI*] | [broker:*brokerURI*] | [properties:*propURI*]}

Arguments

Argument	Description
-D*name*=*value*	Specifies a system property to be interpreted by the VM.
--version	Displays the version of the tool.
--help, -?, -h	Displays the help for the command.
xbean:file:*confURI*	Specifies the path of the broker configuration file to use.
broker:*brokerURI*	Specifies the broker URI to use.
properties:*propURI*	Specifies the path to a properties file to configure the broker.

Examples

The following command starts a broker with the default configuration using an XBean URI:

```
%activemq console xbean:activemq.xml
```

The following command starts a broker using a broker URI:

```
%activemq console broker:(tcp://localhost:61616,network:static:tcp://remotehost:61616)?persist
ent=false&useJmx=true
```

Name

activemq restart — restarts a running broker on Unix, and Unix-like, platforms

Synopsis

`activemq restart` {[*brokerName*] | [--all] | [--jmxurl *JMXUrl*]} [-jmxuser *userName*] [-jmxpassword *password*] [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
brokerName	Specifies the name of the broker to be restarted.
`--all`	Restart all brokers running in the default JMX context.
`--jmxurl` *URL*	Sets the JMX URL to connect to.
`--jmxuser` *user*	Sets the JMX user, used for authentication.
`--jmxpassword` *password*	Sets the JMX password, used for authentication.
`-D`*prop*`=`*value*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command restarts the broker `myBroker`:

```
%activemq restart myBroker
```

Name

activemq status — checks the status of a running broker on Unix, and Unix-like, platforms

Synopsis

`activemq status` [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`-D`*`prop`*`=`*`value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command returns zero if the default broker is running:

```
%activemq status
```

Name

activemq stop — shuts down a running broker on Unix, and Unix-like, platforms

Synopsis

`activemq stop` {[*brokerName*] | [--all]} [--jmxurl *JMXUrl*] [-jmxuser *userName*] [-jmxpassword *password*] [-D*name*=*value*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
brokerName	Specifies the name of the broker to be stopped.
`--all`	Stop all brokers running in the default JMX context.
`--jmxurl` *URL*	Sets the JMX URL used to locate the broker.
`--jmxuser` *user*	Sets the JMX user, used for authentication.
`--jmxpassword` *password*	Sets the JMX password, used for authentication.
`-D`*prop*`=`*value*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command shuts down all brokers in the default JMX context:

```
%activemq stop --all
```

The following command shuts down a broker running in a specific JMX context:

```
%activemq stop myBroker --jmxurl service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi
```

Name

activemq-admin start — creates and starts a broker using a configuration file or a broker URI

Synopsis

`activemq-admin start` [--version] [[--help] | [-h] | [-?]] {*`URI`*}

Arguments

Argument	Description
--version	Displays the version of the tool.
--help, -?, -h	Displays the help for the command.
`URI`	Specifies the XBean URI of the configuration file to use or the full broker URI to use.

Examples

The following command starts a broker using the `activemq.xml` configuration file located on your classpath:

```
activemq-admin start xbean:activemq.xml
```

The following command starts a broker with one transport connector, one network connector and persistence disabled:

```
activemq-admin start broker:(tcp://localhost:61616, network:tcp://localhost:5000)?persist
ent=false
```

Name

activemq-admin create — creates a new instance of the broker's file structure

Synopsis

`activemq-admin create` [--amqconf *`confFile`*] [--version] [[--help] | [-h] | [-?]] {*`brokerPath`*}

Arguments

Argument	Description
--amqconf *`confFile`*	Specifies the configuration file to be copied to the new broker instance.
--version	Displays the version of the tool.
--help, -?, -h	Displays the help for the command.
`brokerPath`	Specifies the path of the new broker instance.

Example

The following command creates a broker1 directory under your home directory:

```
%./activemq-admin create ~/broker1
```

Name

activemq-admin stop — shuts down a running broker

Synopsis

`activemq-admin stop` {[*`brokerName`*] | [--all]} [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*] [-D*`name`*=*`value`*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`brokerName`	Specifies the name of the broker to be stopped.
`--all`	Stop all brokers running in the default JMX context.
`--jmxurl` *`URL`*	Sets the JMX URL used to locate the broker.
`--jmxuser` *`user`*	Sets the JMX user, used for authentication.
`--jmxpassword` *`password`*	Sets the JMX password, used for authentication.
`-D`*`prop`*`=`*`value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command shuts down all brokers in the default JMX context:

```
c:\>activemq-admin stop --all
```

The following command shuts down a broker running in a specific JMX context:

```
c:\>activemq-admin stop myBroker --jmxurl service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi
```

Name

activemq-admin list — lists the available brokers

Synopsis

`activemq-admin list` [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*] [-D*`name`*=*`value`*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`--jmxurl` *`URL`*	Sets the JMX URL used to locate the brokers.
`--jmxuser` *`user`*	Sets the JMX user, used for authentication.
`--jmxpassword` *`password`*	Sets the JMX password, used for authentication.
`-D`*`prop`*`=`*`value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command lists all of the brokers running in the specified JMX context:

```
%activemq-admin list --jmxurl service:jmx:rmi:///jndi/rmi://localhost:8050/jmxrmi
```

Name

activemq-admin query — queries the for broker information on specific objects

Synopsis

`activemq-admin query` [-Q*MBeanType=name*] [-xQ*MBeanType=name*] [--objname *query*] [--xobjname *query*] [--view {*attr*...}] [--jmxurl *JMXUrl*] [-jmxuser *userName*] [-jmxpassword *password*] [-D*name=value*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`-Q` *type=name*	Adds to the search list the specific object type matched by the defined object identifier.
`-xQ` *type=name*	Removes from the search list the specific object type matched by the object identifier.
`--objname` *query*	Adds to the search list objects matched by the query similar.
`--xobjname` *query*	Removes from the search list objects matched by the query.
`--view` *attr1, attr2,...*	Selects the specific attribute of the object to view. By default, all attributes are displayed.
`--jmxurl` *URL*	Sets the JMX URL to connect to.
`--jmxuser` *user*	Sets the JMX user, used for authentication.
`--jmxpassword` *password*	Sets the JMX password, used for authentication.
`--jmxlocal`	Use the local JMX server instead of a remote server.
`-D`*prop=value*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command displays all attributes and object name information for all registered MBeans in the default JMX context:

```
%activemq-admin query
```

The following command displays all attributes and object name information of the destination topic `TEST.FOO`:

```
%activemq-admin query -QTopic=TEST.FOO
```

The following command displays all the brokers in a context whose name ends in `host`:

```
%activemq-admin query -QBroker=*host
```

the Following command displays all attributes and object name information for all registered queues:

```
%activemq-admin query -QQueue=*
```

The following command displays all attributes and object name information for all topics ending with `.FOO` except those that also begin with `ActiveMQ.Advisory.`:

```
%activemq-admin query -QTopic=*.FOO -xQTopic=ActiveMQ.Advisory.*
```

Name

activemq-admin bstat — summarizes the statistics for a broker

Synopsis

`activemq-admin bstat` [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*] [-D*`name`*=*`value`*...] [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`--jmxurl` *`URL`*	Sets the JMX URL used to locate brokers.
`--jmxuser` *`user`*	Sets the JMX user, used for authentication.
`--jmxpassword` *`password`*	Sets the JMX password, used for authentication.
`--jmxlocal`	Use the local JMX server instead of a remote server.
`-D`*`prop`*`=`*`value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command obtains the statistics for the broker named `BrokerName`:

```
%activemq-admin bstat BrokerName
```

The following command obtains statistics for all brokers running in the default JMX context:

```
c:\activemq-admin bstat
```

Name

activemq-admin browse — browse the contents of a destination

Synopsis

`activemq-admin browse` {--amqurl *`brokerURL`*} [--msgsel {msgsel...}] [--view {*`attr`*...}] [[-Vheader] | [-Vcustom] | [-Vbody]] [--version] [[--help] | [-h] | [-?]] *`destName`*

Arguments

Argument	Description
`--amqurl` *`brokerURL`*	Specifies the URL of the broker to which you are connecting.
`--msgsel` *`msgsel1,msgsel2,...`*	Displays messages matched by the message selector. See Message filters on page 86.
`-Vheader`	Shows all the standard JMS message headers.
`-Vcustom`	Shows all the custom fields added to each JMS message.
`-Vbody`	Shows the body of the message.
`--view` *`attr1,attr1,...`*	Selects the specific attribute of the message to view.
`-D`*`prop=value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Message filters

Message filters specified using the `--msgsel` option take the form *`header=value`*. Table B.1 on page 86 lists the headers you can use to filter messages.

Table B.1. Message Headers for Filtering

Name	Type
`JMSCorrelationID`	`String`
`JMSDeliveryMode`	`1`-Non-Persistent, `2`-Persistent
`JMSDestination`	`javax.jms.Destination`

Name	Type
JMSExpiration	long
JMSMessageID	String
JMSPriority	int
JMSRedelivered	boolean
JMSReplyTo	javax.jms.Destination
JMSTimestamp	long
JMSType	String

Examples

The following command prints the JMS message header, custom message header, and message body of all the messages in the queue `TEST.FOO` on a broker:

```
c:\>activemq-admin browse --amqurl tcp://localhost:61616 TEST.FOO
```

The following command displays the attributes from the body of the messages in the `TEST.FOO` queue:

```
c:\>activemq-admin browse --amqurl tcp://localhost:61616 -Vbody TEST.FOO
```

The following command displays any messages with an ID ending in `10`:

```
c:\>activemq-admin browse --amqurl tcp://localhost:61616 --msgsel JMSMessaageID='*:10' TEST.FOO
```

The following command displays messages with a priority of `3`, enter:

```
c:\>activemq-admin browse --amqurl tcp://localhost:61616 --msgsel JMSPriority=3 TEST.FOO
```

The message selectors from the preceding two examples can be combined as follows:

```
c:\>activemq-admin browse --amqurl tcp://localhost:61616 --msgsel JMSMessaageID='*:10',JMSPriority=3
 TEST.FOO
```

Name

activemq-admin journal-audit — audits the log files of the AMQ persistence store. **It does not work with the KahaDB persistence store.**

Synopsis

`activemq-admin journal-audit` [--message-format=*`VelocityTemplate`*]
[--topic-ack-format=*`VelocityTemplate`*] [--queue-ack-format=*`VelocityTemplate`*]
[--transaction-format=*`VelocityTemplate`*] [--trace-format=*`VelocityTemplate`*] [--where=*`JoSQLExp`*]
[-D*`name`*=*`value`*...] [--version] [[--help] | [-h] | [-?]] journalDir

Arguments

Argument	Description
`--message-format=`*`VelocityTemplate`*	Specifies the Apache Velocity template used to format the displayed messages.
`--topic-ack-format=`*`VelocityTemplate`*	Specifies the Apache Velocity template used to display topic ack messages.
`--queue-ack-format=`*`VelocityTemplate`*	Specifies the Apache Velocity template used to display queue ack messages.
`--transaction-format=`*`VelocityTemplate`*	Specifies the Apache Velocity template used to display transaction records.
`--trace-format=`*`VelocityTemplate`*	Specifies the Apache Velocity template used to display trace records.
`--where=`*`JoSQLExp`*	Select the records to display, using a SQL-like syntax implemented by JoSQL[1].
`-D`*`prop`*`=`*`value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`-h, -?, --help`	Displays the online help for this command.

Default Templates

The default Velocity expression for messages is:

```
${location.dataFileId},${location.offset}|${type}|${record.destination}|${record.mes
sageId}|${record.properties}|${body}
```

[1] http://josql.sourceforge.net/

The default Velocity expression for topic acks is:

```
${location.dataFileId},${location.offset}|${type}|${record.destination}|${record.client
Id}|${record.subscriptionName}|${record.messageId}
```

The default Velocity expression for queue acks is:

```
${location.dataFileId},${location.offset}|${type}|${record.destination}|${record.mes
sageAck.lastMessageId}
```

The default Velocity expression for transaction records is:

```
${location.dataFileId},${location.offset}|${type}|${record.transactionId}
```

The default Velocity expression for trace records is:

```
${location.dataFileId},${location.offset}|${type}|${record.message}
```

Name

activemq-admin purge — purges messages from a destination

Synopsis

`activemq-admin purge` [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*] [--msgsel {msgsel...}] [-D*`name`*=*`value`*...] [--version] [[--help] | [-h] | [-?]] destName

Arguments

Argument	Description
`--jmxurl` *`URL`*	Sets the JMX URL used to locate the broker.
`--jmxuser` *`user`*	Sets the JMX user, used for authentication.
`--jmxpassword` *`password`*	Sets the JMX password, used for authentication.
`--msgsel` *`msgsel1,msgsel2,...`*	Purges messages matched by the message selector. See Message filters on page 86.
`-D`*`prop=value`*	Sets a Java system property. For example, `-Dactivemq.home=C:/ActiveMQ`.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Examples

The following command purges all the messages in the queue `TEST.FOO` on a broker:

```
c:\activemq-admin purge TEST.FOO
```

The following command purges any messages with an ID ending in `10`:

```
c:\activemq-admin purge --msgsel JMSMessaageID='*:10' TEST.FOO
```

The following command purges messages with a priority of `3`, enter:

```
c:\activemq-admin purge --msgsel JMSPriority=3 TEST.FOO
```

The message selectors from the preceding two examples can be combined as follows:

```
c:\activemq-admin purge --msgsel JMSMessaageID='*:10',JMSPriority=3 TEST.FOO
```

Name

activemq-admin encrypt — encrypts the specified text

Synopsis

`activemq-admin encrypt` [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*] {--password *`password`*} {--input *`text`*} [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`--jmxurl` *`URL`*	Sets the JMX URL used to locate the broker.
`--jmxuser` *`user`*	Sets the JMX user, used for authentication.
`--jmxpassword` *`password`*	Sets the JMX password, used for authentication.
`--password` *`password`*	Specifies the password used to encrypt the text.
`--input` *`text`*	Specifies the text to encrypt.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Name

activemq-admin decrypt — decrypts an encrypted message

Synopsis

`activemq-admin decrypt` [--jmxurl *`JMXUrl`*] [-jmxuser *`userName`*] [-jmxpassword *`password`*] {--password *`password`*} {--input *`text`*} [--version] [[--help] | [-h] | [-?]]

Arguments

Argument	Description
`--jmxurl` *`URL`*	Sets the JMX URL used to locate the broker.
`--jmxuser` *`user`*	Sets the JMX user, used for authentication.
`--jmxpassword` *`password`*	Sets the JMX password, used for authentication.
`--password` *`password`*	Specifies the password used to encrypt the text.
`--input` *`text`*	Specifies the text to decrypt.
`--version`	Displays the version information.
`-h, -?, --help`	Displays the online help for this command.

Appendix C. Broker Properties

Global properties

Table C.1 on page 93 describes the broker properties you can set as options on a broker URI or in a broker properties file.

Table C.1. Broker URI Properties

Property	Default	Description
`useJmx`	true	Specifies if the broker will connect to a JMX server.
`persistent`	true	Specifies whether the broker uses persistent storage.
`populateJMSXUserID`	false	Specifies whether the broker populates the `JMSXUserID` property of messages to indicate the authenticated sender username of the person who sent the message.
`useShutdownHook`	true	Specifies if the broker installs a shutdown hook so that it can properly shut itself down on a JVM kill.
`brokerName`	localhost	Specifies the name of the broker.
`brokerId`	*`hostname`*	Specifies the ID of the broker.
`deleteAllMessagesOnStartup`	false	Specifies if all the messages in the persistent store will be deleted on broker startup.
`enableStatistics`	true	Specifies if statistics gathering is enabled.

Property file specific

Table C.2 on page 94 describes additional properties that you can set in a broker properties file.

Table C.2. Property File Properties

Property	Default	Description
`transportConnectors`	`tcp:\\localhost:61616`	Specifies a comma separated list of transport URIs on which the broker will listen for client connections. See *Connectivity Guide* for more information.
`networkConnectors`		Specifies a comma separated list of URIs on which the broker will listen for connections with other brokers. See *Using Networks of Brokers* for more information.

Index

D

E

F

I

J

K

M

N

P

Q

R

S

T

U

V

W

X