**Using Apache ActiveMQ**

To help you get started using Apache ActiveMQ version 5 you may wish to start off with the [Version 5 Getting Started](http://activemq.apache.org/version-5-getting-started.html) guide or the [Configuring version 5 Transports](http://activemq.apache.org/configuring-version-5-transports.html). Otherwise here is a complete list of the guides.

* [Configure version 5 Brokers](http://activemq.apache.org/configure-version-5-brokers.html)
* [Configuring Version 5 Transports](http://activemq.apache.org/configuring-version-5-transports.html)
* [LDAP Broker Discovery Mechanism](http://activemq.apache.org/ldap-broker-discovery-mechanism.html)
* [Version 5 Examples](http://activemq.apache.org/version-5-examples.html)
* [Version 5 Getting Started](http://activemq.apache.org/version-5-getting-started.html)
* [Version 5 Hello World](http://activemq.apache.org/version-5-hello-world.html)
* [Version 5 Initial Configuration](http://activemq.apache.org/version-5-initial-configuration.html)
* [Version 5 Installation](http://activemq.apache.org/version-5-installation.html)
* [Version 5 Performance Tuning](http://activemq.apache.org/version-5-performance-tuning.html)
* [Version 5 Run Broker](http://activemq.apache.org/version-5-run-broker.html)
* [Version 5 Topologies](http://activemq.apache.org/version-5-topologies.html)
* [Version 5 Web Samples](http://activemq.apache.org/version-5-web-samples.html)
* [Version 5 XML Configuration](http://activemq.apache.org/version-5-xml-configuration.html)

**Commercial Documentation**

The Commercial Providers listed on the [Support](http://activemq.apache.org/support.html) page may also have additional documentation, examples, tutorials, etc... that can help you get started using ActiveMQ.

# Configure version 5 Brokers

### Overview

There is an updates XML syntax for configuring message brokers - [see here](http://activemq.apache.org/schema/core/activemq-core-5.0-SNAPSHOT.xsd)

So we decided that using XML would make this configuration much easier. we use [XBean](http://xbean.org/) to perform the XML configuration.

For details of the XML see the [Xml Reference](http://activemq.apache.org/xml-reference.html)

**Be careful with broker names and URIs**

Make sure you do not use any strange characters in the names of brokers as they are converted to URIs which [do not allow things like underscores](http://java.sun.com/j2se/1.4.2/docs/api/java/net/URI.html) in them etc.

## Examples

The default ActiveMQ configuration: [current default config](http://svn.apache.org/repos/asf/activemq/trunk/assembly/src/release/conf/activemq.xml).

|  |
| --- |
| <beans    xmlns="<http://www.springframework.org/schema/beans>"    xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"    xsi:schemaLocation="http://www.springframework.org/schema/beans <http://www.springframework.org/schema/beans/spring-beans.xsd>    http://activemq.apache.org/schema/core <http://activemq.apache.org/schema/core/activemq-core.xsd>">      <!-- Allows us to use system properties as variables in this configuration file -->      <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">          <property name="locations">              <value>file:${activemq.conf}/credentials.properties</value>          </property>      </bean>     <!-- Allows accessing the server log -->      <bean id="logQuery" class="io.fabric8.insight.log.log4j.Log4jLogQuery"            lazy-init="false" scope="singleton"            init-method="start" destroy-method="stop">      </bean>      <!--          The <broker> element is used to configure the ActiveMQ broker.      -->      <broker xmlns="<http://activemq.apache.org/schema/core>" brokerName="localhost" dataDirectory="${activemq.data}">          <destinationPolicy>              <policyMap>                <policyEntries>                  <policyEntry topic=">" >                      <!-- The constantPendingMessageLimitStrategy is used to prevent                           slow topic consumers to block producers and affect other consumers                           by limiting the number of messages that are retained                           For more information, see:                           http://activemq.apache.org/slow-consumer-handling.html                      -->                    <pendingMessageLimitStrategy>                      <constantPendingMessageLimitStrategy limit="1000"/>                    </pendingMessageLimitStrategy>                  </policyEntry>                </policyEntries>              </policyMap>          </destinationPolicy>            <!--              The managementContext is used to configure how ActiveMQ is exposed in              JMX. By default, ActiveMQ uses the MBean server that is started by              the JVM. For more information, see:              http://activemq.apache.org/jmx.html          -->          <managementContext>              <managementContext createConnector="false"/>          </managementContext>          <!--              Configure message persistence for the broker. The default persistence              mechanism is the KahaDB store (identified by the kahaDB tag).              For more information, see:              http://activemq.apache.org/persistence.html          -->          <persistenceAdapter>              <kahaDB directory="${activemq.data}/kahadb"/>          </persistenceAdapter>              <!--              The systemUsage controls the maximum amount of space the broker will              use before disabling caching and/or slowing down producers. For more information, see:              http://activemq.apache.org/producer-flow-control.html            -->            <systemUsage>              <systemUsage>                  <memoryUsage>                      <memoryUsage percentOfJvmHeap="70" />                  </memoryUsage>                  <storeUsage>                      <storeUsage limit="100 gb"/>                  </storeUsage>                  <tempUsage>                      <tempUsage limit="50 gb"/>                  </tempUsage>              </systemUsage>          </systemUsage>          <!--              The transport connectors expose ActiveMQ over a given protocol to              clients and other brokers. For more information, see:              http://activemq.apache.org/configuring-transports.html          -->          <transportConnectors>              <!-- DOS protection, limit concurrent connections to 1000 and frame size to 100MB -->              <transportConnector name="openwire" uri="<tcp://0.0.0.0:61616?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600>"/>              <transportConnector name="amqp" uri="<amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600>"/>              <transportConnector name="stomp" uri="<stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600>"/>              <transportConnector name="mqtt" uri="<mqtt://0.0.0.0:1883?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600>"/>              <transportConnector name="ws" uri="[ws://0.0.0.0:61614?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600](ws://0.0.0.0:61614/?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600)"/>          </transportConnectors>          <!-- destroy the spring context on shutdown to stop jetty -->          <shutdownHooks>              <bean xmlns="<http://www.springframework.org/schema/beans>" class="org.apache.activemq.hooks.SpringContextHook" />          </shutdownHooks>      </broker>      <!--          Enable web consoles, REST and Ajax APIs and demos          The web consoles requires by default login, you can disable this in the jetty.xml file          Take a look at ${ACTIVEMQ\_HOME}/conf/jetty.xml for more details      -->      <import resource="jetty.xml"/>  </beans> |

From the binary distributionthere is an activemq script allowing you to run a Message Broker as a stand alone process from the command line easily providing the $ACTIVEMQ\_HOME/bin directory is on your PATH.

# Configuring Version 5 Transports

**Transport configuration options**

One of the first kinds of URI you are likely to use is a transport URI to connect to a broker using a kind of transport. Generally TCP or VM are the first transports you'll use.

**Be careful about whitespace**

All of the following URI configurations are based on the java.net.URI class which does not allow whitespace to be used. So if you are using **failover:** or **static:** URIs, do not put any whitespace around the **','**symbol.

**The AUTO Transport**

Starting with 5.13.0 ActiveMQ has support for automatic wire protocol detection over TCP, SSL, NIO, and NIO SSL.  OpenWire, STOMP, AMQP, and MQTT are supported.  For details see the [AUTO](http://activemq.apache.org/auto.html)Transport Reference.

**The VM Transport**

The VM transport allows clients to connect to each other inside the VM without the overhead of the network communication. The connection used is not that of a socket connection but instead uses direct method invocations to enable a high performance embedded messaging system.

The first client to use the VM connection will boot an embedded broker. Subsequent connections will attach that the same broker. Once all VM connections to the broker have been closed, the embedded broker will automatically shutdown.

For more information see the [VM Transport Reference](http://activemq.apache.org/vm-transport-reference.html)

**The AMQP Transport**

As of 5.8.0 ActiveMQ has support for AMQP. For details see the [AMQP](http://activemq.apache.org/amqp.html) Transport Reference.

**The MQTT Transport**

Starting with 5.6.0 ActiveMQ also supports [MQTT](http://mqtt.org/). Its a light weight publish/subscribe messaging transport. See the [MQTT](http://activemq.apache.org/mqtt.html) Transport Reference for details.

**The TCP Transport**

The TCP transport allows clients to connect a remote ActiveMQ using a a TCP socket.

For more information see the [TCP Transport Reference](http://activemq.apache.org/tcp-transport-reference.html)

**The NIO Transport**

Same as the TCP transport, except that the [New I/O (NIO)](http://en.wikipedia.org/wiki/New_I/O) package is used, which may provide better performance. The Java NIO package should not be confused with IBM's [AIO4J](http://java.sys-con.com/read/46658.htm)package.

To switch from TCP to NIO, simply change the scheme portion of the URI. Here's an example as defined within a broker's XML configuration file.

|  |
| --- |
| <broker>    ...    <transportConnectors>      <transportConnector name="nio" uri="<nio://0.0.0.0:61616>"/>    </<transportConnectors>    ...  </broker> |

Trying to use nio transport url on the client side will instantiate the regular TCP transport. For more information see the [NIO Transport Reference](http://activemq.apache.org/nio-transport-reference.html)

**The SSL Transport**

This allows you to talk over TCP using SSL. For more information see the [SSL Transport Reference](http://activemq.apache.org/ssl-transport-reference.html)

**The NIO SSL Transport**

**Availability**

Available since 5.6

Implementing SSL transport over NIO. This allows you to connect large number of SSL clients to a single broker instance. It's server side transport-option only

|  |
| --- |
| <broker>    ...    <transportConnectors>      <transportConnector name="nio+ssl" uri="nio+<ssl://0.0.0.0:61616>"/>    </<transportConnectors>    ...  </broker> |

Trying to use nio+ssl transport url on the client side will instantiate the regular SSL transport.

**The Peer Transport**

The Peer transport provides a peer-to-peer network with ActiveMQ. What actually happens is the peer transport uses the VM transport to create and connect to a local embedded broker but which configures the embedded broker to establish network connections to other peer embedded brokers.

For more information see the [Peer Transport Reference](http://activemq.apache.org/peer-transport-reference.html)

**The UDP Transport**

This allows you to talk over UDP.

For more information see the [UDP Transport Reference](http://activemq.apache.org/udp-transport-reference.html)

**The Multicast Transport**

This allows you to talk over Multicast.

For more information see the [Multicast Transport Reference](http://activemq.apache.org/multicast-transport-reference.html)

**The HTTP and HTTPS Transport**

This allows the ActiveMQ client and broker to tunnel over HTTP. If the client is not JMS you might want to look at [REST](http://activemq.apache.org/rest.html) or [Ajax](http://activemq.apache.org/ajax.html) support instead.

For more information see the [HTTP and HTTPs Transports Reference](http://activemq.apache.org/http-and-https-transports-reference.html)

**The WebSockets Transport**

This transport uses the new HTML5 WebSockets to exchange messages with the broker. For more information see the [WebSockets](http://activemq.apache.org/websockets.html) Transport Reference

**The Stomp Transport**

A plain text transport that can be used with many languages. See [Stomp](http://activemq.apache.org/stomp.html) for more details.

**General Purpose URIs**

You can configure other features via the URI syntax as follows...

**Connection Configuration URI**

Any Apache ActiveMQ JMS connection can be configured using the URL or explicitly setting properties on the [ActiveMQConnection](http://incubator.apache.org/activemq/maven/activemq-core/apidocs/org/apache/activemq/ActiveMQConnection.html) or [ActiveMQConnectionFactory](http://incubator.apache.org/activemq/maven/activemq-core/apidocs/org/apache/activemq/ActiveMQConnectionFactory.html) objects themselves.

For more information see [Connection Configuration URI](http://activemq.apache.org/connection-configuration-uri.html)

**Destination Options**

You can configure various consumer related options using [Destination Options](http://activemq.apache.org/destination-options.html) which allow you to configure destinations using URI syntax.

**Broker Configuration URI**

You can use a [Broker Configuration URI](http://activemq.apache.org/broker-configuration-uri.html) to configure an embedded broker, either using the BrokerFactory helper class from Java or using the activemq shell script. For more details see [How to Run a Broker](http://activemq.apache.org/run-broker.html)

**Configuring Wire Formats**

Any transport which involves marshalling messages onto some kind of network transport like TCP or UDP will typically use the [OpenWire](http://activemq.apache.org/openwire.html)format. This is configurable to customize how things appear on the wire.

For more information see [Configuring Wire Formats](http://activemq.apache.org/configuring-wire-formats.html)

**High Level Protocol URIs**

The following higher level protocols can be configured via URI

**The Failover Transport**

The Failover transport layers reconnect logic on top of any of the other transports. This is what used to be the Reliable transport in ActiveMQ 3. Its configuration syntax allows you to specify any number of composite URIs. The Failover transport randomly chooses one of the composite URIs and attempts to establish a connection to it. If it does not succeed or if it subsequently fails, a new connection is established to one of the other URIs in the list.

For more information see the [Failover Transport Reference](http://activemq.apache.org/failover-transport-reference.html)

**The Fanout Transport**

The Fanout transport layers reconnect and replicate logic on top of any of the other transports. It is used replicate commands to multiple brokers.

For more information see the [Fanout Transport Reference](http://activemq.apache.org/fanout-transport-reference.html)

**Using Discovery**

Often when using transports like TCP you want to use [Discovery](http://activemq.apache.org/discovery.html) to locate the available brokers. This is different from using, say, [Multicast](http://activemq.apache.org/multicast-transport-reference.html) - as the actual main communication is over TCP but multicast is purely used to discover the location of brokers.

**The Discovery Transport**

The Discovery transport works just like the reliable transport, except that it uses a discovery agent to locate the list of URIs to connect to.

For more information see the [Discovery Transport Reference](http://activemq.apache.org/discovery-transport-reference.html)

**The ZeroConf Transport**

The ZeroConf transport provides [Discovery](http://activemq.apache.org/discovery.html) and it works like the [Discovery Transport](http://activemq.apache.org/discovery-transport-reference.html) but rather than using our own multicast based discovery mechanism (which allows you to configure the exact multicast address and port, etc.), the [ZeroConf](http://activemq.apache.org/zeroconf.html) transport is used instead.

For more information see the [ZeroConf Transport Reference](http://activemq.apache.org/zeroconf-transport-reference.html)

**Server side options**

There are a number of options that can be used for changing behavior on the server for the **TransportConnector** in the ActiveMQ broker configuration. These are:

|  |  |  |
| --- | --- | --- |
| property name | default | description |
| allowLinkStealing | false | This is enabled for default for MQTT transport.  Link Stealing is where the last of two or more connections with the same id (clientID for JMS) is deemed the valid connection and the older one is closed by the broker. |
| discoveryURI | null | If set, the multicast discovery address for client connections to find the broker. |
| enableStatusMonitor | false | Will monitor connections to determine if they are blocked. |
| name | null | The name of the **TransportConnector** instance. |
| rebalanceClusterClients | false | Will automatically re-balance clients across the cluster on changes of topology. |
| updateClusterClients | false | If enabled, will update client connections (if they use the **failover://** transport) of changes to the broker cluster. |
| updateClusterClientsOnRemove | false | Will update clients if a broker is removed from the cluster. |
| updateClusterFilter | null | Comma separated list of regular expressions. Brokers with a name matching the pattern will be included for client updates. |
| uri | null | The bind address for the transport. |

**Note**: properties in red are version 5.10 (and higher) options only.

Example configuration:

|  |
| --- |
| <broker>     <!-- ... -->       <transportConnectors>       <transportConnector name="openwire" uri="<tcp://0.0.0.0:61616>" enableStatusMonitor="true"/>     </<transportConnectors>       <!-- ... -->  </broker> |

# LDAP Broker Discovery Mechanism

Configuring network topologies can be quite tedious when the number of brokers in the system is large. To help ease the configuration overhead for these types of situations, a broker can be configured to look up its broker connections using a LDAP v3 directory server.

**Note**

The basic feature was added to satisfy [AMQ-358](https://issues.apache.org/activemq/browse/AMQ-358). There are known problems and limitations with this implementation. These deficiencies have been addressed in [AMQ-1587](https://issues.apache.org/activemq/browse/AMQ-1587). The features discussed on this page require the patch attached to JIRA issue [AMQ-1587](https://issues.apache.org/activemq/browse/AMQ-1587). This patch should apply cleanly to the ActiveMQ 5.0.0 release or the current development trunk.

**LDAP v3 Directory Server Compliance**

The following table lists a known subset of directory servers and their compliance to work with the LDAP discovery feature. Most LDAP v3 directory servers will support this feature if they properly implement the [RFC2307](http://www.ietf.org/rfc/rfc2307.txt) schemas. In order to support the persistent search capabilities the server must implement the extension defined in [draft-ietf-ldapext-psearch-03.txt](http://www.ietf.org/proceedings/01aug/I-D/draft-ietf-ldapext-psearch-03.txt).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vendor | Product | Version | [RFC2307](http://www.ietf.org/rfc/rfc2307.txt) | [draft-ietf-ldapext-psearch-03.txt](http://www.ietf.org/proceedings/01aug/I-D/draft-ietf-ldapext-psearch-03.txt) |
| Apache | ApacheDS | 1.0.x | (tick) | (tick) |
| Microsoft | Active Directory | Windows 2000  Windows 2003 | (warning) | (error) |
| Microsoft | Active Directory | Windows 2003 R2 | (tick) | (error) |
| Sun | OpenDS | 0.9.x | (tick) | (tick) |
| OpenLDAP | OpenLDAP | 2.3.x  2.4.x | (tick) | (error) |

(warning) LDAP v3 directory server which do not support [RFC2307](http://www.ietf.org/rfc/rfc2307.txt) by default. Support can be added by manually importing them. See vendor specific setup requirements on how to do this.  
(error) LDAP v3 directory servers which do not support the [draft-ietf-ldapext-psearch-03.txt](http://www.ietf.org/proceedings/01aug/I-D/draft-ietf-ldapext-psearch-03.txt).

**LDAP Network Connector Properties**

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Default | Required | Description |
| uri | null | **Yes** | The URI of the LDAP v3 Server to connect to (i.e. ldap://host:port, failover://(ldap://host1:port,ldap://host2:port). |
| base | null | **Yes** | The RDN base used as the root for the search criteria. |
| user | null | **Yes**, if not using anonymousAuthentication | The username needed to bind to the server. |
| password | null | **Yes**, if not using anonymousAuthentication | The password needed to bind to the server. |
| anonymousAuthentication | false | **Yes**, if not using user/password | Enable if you want to bind to the server anonymously. This is recommended over using user/password properties since your login credentials will not be stored in an unencrypted XML file. |
| searchEventListener | false | No | Enable if you want the broker to stay in sync with changes made to entries matching the search criteria. |
| searchScope | ONELEVEL\_SCOPE | No | Can be any of search scopes defined by javax.naming.directory.SearchControls class.  OBJECT\_SCOPE - search the named object defined by base.  ONELEVEL\_SCOPE - search one level of the base.  SUBTREE\_SCOPE - search entire subtree rooted at the base. |
| searchFilter | (&(objectClass=ipHost)(objectClass=ipService)) | No | Can be any filter that conforms to [RFC2254](http://www.ietf.org/rfc/rfc2254.txt). If a custom one is specified the actual search filter used will be (&(&(objectClass=ipHost)(objectClass=ipService))(USER CUSTOM)) |

**Other Properties**

All of the properties defined in [Networks of Brokers](http://activemq.apache.org/networks-of-brokers.html) are also available to the ldapNetworkConnector. Any of the properties defined for a normal networkConnector will be used as connection parameters to any discovered brokers matching the search criteria.

**Example 1: Simple Network of Brokers**

**Network Configuration**

|  |  |  |
| --- | --- | --- |
| Topology | LDAP v3 Directory Structure | Entry |
| http://activemq.apache.org/ldap-broker-discovery-mechanism.data/Example1-Topology.jpg | http://activemq.apache.org/ldap-broker-discovery-mechanism.data/Example1-DirectoryStructure.jpg | http://activemq.apache.org/ldap-broker-discovery-mechanism.data/Example1-Entry.jpg |

**ActiveMQ Configuration (activemq.xml)**

**srv-a.mydomain.com**

|  |
| --- |
| <broker brokerName="srv-a.mydomain.com" ...>     ...       <networkConnectors>        <ldapNetworkConnector uri="<ldap://myldap.mydomain.com:389>"                              base="dc=brokers,dc=mydomain,dc=com"                              anonymousAuthentication="true"                              searchFilter="(cn=\*)"                              searchScope="SUBTREE\_SCOPE"                              />     </networkConnectors>       ...  </broker> |

**srv-b.mydomain.com**

|  |
| --- |
| <broker brokerName="srv-b.mydomain.com" ...>     ...       <networkConnectors/>        <!-- NO NETWORK CONNECTORS -->     </networkConnectors>       ...  </broker> |

**Example 2: Larger Network of Brokers**

**Network Configuration**

|  |  |
| --- | --- |
| Topology | LDAP v3 Directory Structure |
| http://activemq.apache.org/ldap-broker-discovery-mechanism.data/Example2-Topology.jpg | http://activemq.apache.org/ldap-broker-discovery-mechanism.data/Example2-DirectoryStructure.jpg |

**ActiveMQ Configuration (activemq.xml)**

**srv-a.mydomain.com**

|  |
| --- |
| <broker brokerName="srv-a.mydomain.com" ...>     ...       <networkConnectors>        <ldapNetworkConnector uri="<ldap://myldap.mydomain.com:389>"                              base="dc=brokers-for-srv-a,dc=mydomain,dc=com"                              anonymousAuthentication="true"                              searchFilter="(cn=\*)"                              searchScope="SUBTREE\_SCOPE"                              networkTTL="2"                              />        <!-- networkTTL=2 since we want messages to flow from srv-a to srv-c -->     </networkConnectors>       ...  </broker> |

**srv-b.mydomain.com**

|  |
| --- |
| <broker brokerName="srv-b.mydomain.com" ...>     ...       <networkConnectors/>        <ldapNetworkConnector uri="<ldap://myldap.mydomain.com:389>"                              base="dc=brokers-other,dc=mydomain,dc=com"                              anonymousAuthentication="true"                              searchFilter="(cn=\*)"                              searchScope="SUBTREE\_SCOPE"                              />     </networkConnectors>       ...  </broker> |

**srv-c.mydomain.com**

|  |
| --- |
| <broker brokerName="srv-c.mydomain.com" ...>     ...       <networkConnectors/>        <!-- NO NETWORK CONNECTORS -->     </networkConnectors>       ...  </broker> |

**srv-d.mydomain.com**

|  |
| --- |
| <broker brokerName="srv-d.mydomain.com" ...>     ...       <networkConnectors/>        <ldapNetworkConnector uri="<ldap://myldap.mydomain.com:389>"                              base="dc=brokers-other,dc=mydomain,dc=com"                              anonymousAuthentication="true"                              searchFilter="(cn=\*)"                              searchScope="SUBTREE\_SCOPE"                              />     </networkConnectors>       ...  </broker> |

# Version 5 Examples

Since version**5.12.0**, Apache ActiveMQ comes with the new tool that can be used to produce and consume messages from the broker.

## Prerequisites

Before running the examples you should try [running a JMS broker](http://activemq.apache.org/run-broker.html) on your machine. Follow the [Installation](http://activemq.apache.org/installation.html) instructions to use a binary distribution of ActiveMQ. To run the broker in a command shell, type:

|  |
| --- |
| bin/activemq console |

This starts up ActiveMQ.

## Running

From the[command line](http://activemq.apache.org/unix-shell-script.html) you can run

|  |
| --- |
| ${ACTIVEMQ\_HOME}/bin/activemq producer  ${ACTIVEMQ\_HOME}/bin/activemq consumer |

If you have activemq-all jar available you can achieve the same with

|  |
| --- |
| java -jar activemq-all-5.x.x.jar producer  java -jar activemq-all-5.x.x.jar consumer |

If you run inside [Karaf](http://activemq.apache.org/osgi-integration.html), you can run the commands as

|  |
| --- |
| activemq:producer  activemq:consumer |

## Options

For all options on the commands, run them with --help parameter. The up to date, options for [producer](https://github.com/apache/activemq/blob/master/activemq-console/src/main/resources/org/apache/activemq/console/command/producer.txt) and [consumer](https://github.com/apache/activemq/blob/master/activemq-console/src/main/resources/org/apache/activemq/console/command/consumer.txt) can be found in the source.

## Examples

Here are a couple of examples of more advanced features.

To send a text message with custom text use

|  |
| --- |
| bin/activemq producer --message "My message" --messageCount 1 |

To send byte message of custom length use

|  |
| --- |
| bin/activemq producer --messageSize 100 --messageCount 1 |

To send a text message with content obtained from an url

|  |
| --- |
| bin/activemq producer --payloadUrl http://activemq.apache.org/schema/core/activemq-core.xsd --messageCount 1 |

To consume in transaction use

|  |
| --- |
| bin/activemq consumer --transacted true |

To use client acknowledgment use

|  |
| --- |
| bin/activemq consumer --ackMode CLIENT\_ACKNOWLEDGE |

To use durable topic subscribers use

|  |
| --- |
| bin/activemq consumer --durable true --clientId example --destination topic://TEST |

## Old examples

In older versions of ActiveMQ, the corresponding examples were located in examples/ or examples/openwire/swissarmy/ directories, where you can do the similar tasks with an ant script.

|  |
| --- |
| ant producer  ant consumer |

## Other examples

In examples/ directory (depending on the version) you can find more examples of using the broker with variety of protocols (mqtt, amqp, ...) and clients (Java, Ruby, JavaScript, ...), so it's the good place to start learning. Also, examples/conf/ directory contains a lot of different configuration examples that you can use as a starting point for your deployment.

## See Also

* [Web Samples](http://activemq.apache.org/web-samples.html)
* [Web Console](http://activemq.apache.org/web-console.html)
* [JMX](http://activemq.apache.org/jmx.html)

# Version 5 Hello World

The following is a very simple JMS application with multiple, concurrent, consumers and producers. See the [Initial Configuration](http://activemq.apache.org/initial-configuration.html) guide for details on how to setup your classpath correctly.

Things you might do after running this example:

* [Setup a broker](http://activemq.apache.org/run-broker.html) instead of using the org.activemq.broker.impl.Main class directly
* Use JNDI to lookup a javax.jms.ConnectionFactory rather than creating ActiveMQConnectionFactory directly.
* Implement the javax.jms.MessageListener interface rather than calling consumer.receive()
* Use transactional sessions
* Use a Topic rather than a queue

The point of this example is to show you the basic code required to use JMS in a straightforward way. The Consumers and Producers could very easy be on completely different machines or in different processes.

**App.java**

|  |
| --- |
| import org.apache.activemq.ActiveMQConnectionFactory;    import javax.jms.Connection;  import javax.jms.DeliveryMode;  import javax.jms.Destination;  import javax.jms.ExceptionListener;  import javax.jms.JMSException;  import javax.jms.Message;  import javax.jms.MessageConsumer;  import javax.jms.MessageProducer;  import javax.jms.Session;  import javax.jms.TextMessage;    /\*\*   \* Hello world!   \*/  public class App {        public static void main(String[] args) throws Exception {          thread(new HelloWorldProducer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          Thread.sleep(1000);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          Thread.sleep(1000);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldProducer(), false);          Thread.sleep(1000);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldConsumer(), false);          thread(new HelloWorldProducer(), false);      }        public static void thread(Runnable runnable, boolean daemon) {          Thread brokerThread = new Thread(runnable);          brokerThread.setDaemon(daemon);          brokerThread.start();      }        public static class HelloWorldProducer implements Runnable {          public void run() {              try {                  // Create a ConnectionFactory                  ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("<vm://localhost>");                    // Create a Connection                  Connection connection = connectionFactory.createConnection();                  connection.start();                    // Create a Session                  Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);                    // Create the destination (Topic or Queue)                  Destination destination = session.createQueue("TEST.FOO");                    // Create a MessageProducer from the Session to the Topic or Queue                  MessageProducer producer = session.createProducer(destination);                  producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);                    // Create a messages                  String text = "Hello world! From: " + Thread.currentThread().getName() + " : " + this.hashCode();                  TextMessage message = session.createTextMessage(text);                    // Tell the producer to send the message                  System.out.println("Sent message: "+ message.hashCode() + " : " + Thread.currentThread().getName());                  producer.send(message);                    // Clean up                  session.close();                  connection.close();              }              catch (Exception e) {                  System.out.println("Caught: " + e);                  e.printStackTrace();              }          }      }        public static class HelloWorldConsumer implements Runnable, ExceptionListener {          public void run() {              try {                    // Create a ConnectionFactory                  ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("<vm://localhost>");                    // Create a Connection                  Connection connection = connectionFactory.createConnection();                  connection.start();                    connection.setExceptionListener(this);                    // Create a Session                  Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);                    // Create the destination (Topic or Queue)                  Destination destination = session.createQueue("TEST.FOO");                    // Create a MessageConsumer from the Session to the Topic or Queue                  MessageConsumer consumer = session.createConsumer(destination);                    // Wait for a message                  Message message = consumer.receive(1000);                    if (message instanceof TextMessage) {                      TextMessage textMessage = (TextMessage) message;                      String text = textMessage.getText();                      System.out.println("Received: " + text);                  } else {                      System.out.println("Received: " + message);                  }                    consumer.close();                  session.close();                  connection.close();              } catch (Exception e) {                  System.out.println("Caught: " + e);                  e.printStackTrace();              }          }            public synchronized void onException(JMSException ex) {              System.out.println("JMS Exception occured.  Shutting down client.");          }      }  } |

# Version 5 Initial Configuration

## Document Organization

* [Required JARs](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-RequiredJARs)
* [Optional JARS](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-OptionalJARS)
* [Persistence support](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-Persistencesupport)
* [Next steps](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-Nextsteps)
* [Additional resources](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-Additionalresources)
  + [Related reading](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-Relatedreading)
  + [Specifications](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-Specifications)
  + [Related open source projects](http://activemq.apache.org/version-5-initial-configuration.html#Version5InitialConfiguration-Relatedopensourceprojects)

Firstly you need to add the jars to your classpath.

## Required JARs

To make ActiveMQ easy to use, the default **activemq-all.jar** comes complete with all the libraries required. If you prefer to have explicit control over all the jars used by ActiveMQ here is the full list of individual jars required

* activemq-broker.jar
* activemq-client.jar
* activemq-kahadb-store.jar
* activemq-spring.jar
* hawtbuf-1.11.jar
* slf4j-api.jar
* slf4j-log4j12.jar
* log4j-1.2.17.jar
* J2EE APIs which could be the j2ee.jar from Sun or your J2EE container or you could use Geronimo's freely distributable geronimo-spec-j2ee.jar. If you are inside a servlet container and being dependent on the j2ee.jar causes you troubles, the parts of the J2EE jar we are dependent on are as follows...
  + geronimo-spec-jms.jar
  + geronimo-spec-jta.jar
  + geronimo-spec-j2ee-management.jar

If you want to grab a J2EE specification jar we recommend the Apache [repository](http://cvs.apache.org/repository/geronimo-spec/jars/)

## Optional JARS

* spring.jar - if you wish to use the XML configuration file for configuring the Message Broker
* if you wish to use message persistence then you need to add a persistent jar to your classpath (see below). If you just want a lightweight message bus with no durability you can leave this step out but we highly recommend persistence for production deployments.

## Persistence support

The default persistence is the [AMQ Message Store](http://activemq.apache.org/amq-message-store.html). We do still support persistence via [JDBC and a high performance journal](http://activemq.apache.org/persistence.html). For full explict control over configuration check out the [Xml Configuration](http://activemq.apache.org/xml-configuration.html).

If you're just doing some testing or in-VM SEDA based messaging you may wish to disable persistence. You can use the [Xml Configuration](http://activemq.apache.org/xml-configuration.html) for this.

You can do this by setting the usePersistence property to false either in the [Xml Configuration](http://activemq.apache.org/xml-configuration.html) or on the [broker URL](http://activemq.apache.org/configuring-transports.html).

## Next steps

One of the first things you might want to do is [start a broker](http://activemq.apache.org/run-broker.html). Once you have a broker running you could try using the [JNDI Support](http://activemq.apache.org/jndi-support.html) which shows how to run an example JMS program. Or there are some other [example programs](http://activemq.apache.org/examples.html)

If you don't want to use JNDI you can just instantiate an [ActiveMQConnectionFactory](http://activemq.codehaus.org/maven/apidocs/org/apache/activemq/ActiveMQConnectionFactory.html), configure its properties directly and then you're ready to use the standard JMS API to create Connections, Sessions, MessageProducer and MessageConsumer instances.

## Additional resources

### Related reading

* Sun's [JMS Tutorial](http://java.sun.com/products/jms/tutorial/) is a handy place to start looking at how to use the JMS API directly
* The ActiveMQ [Website](http://activemq.org/) for specifics on how to use ActiveMQ
* [ActiveMQ Topologies](http://activemq.apache.org/topologies.html)
* [ActiveMQ Clustering](http://activemq.apache.org/clustering.html)
* [ActiveMQ Network of Brokers](http://activemq.apache.org/networks-of-brokers.html)
* [Staged Event Driven Architecture (SEDA)](http://www.eecs.harvard.edu/~mdw/proj/seda/)

### Specifications

* [Java Connector Architecture 1.5](http://java.sun.com/j2ee/connector/)
* [Java Messaging Service](http://java.sun.com/products/jms/index.jsp)
* [WSIF](http://ws.apache.org/wsif/)
* [WS-Notification](http://www-128.ibm.com/developerworks/webservices/library/specification/ws-notification/)

### Related open source projects

* [Apache Geronimo](http://geronimo.apache.org/)
* [Hermes JMS](http://www.hermesjms.com/)
* [Jencks](http://jencks.org/) is a Spring JCA container allowing you to use connection & thread & POJO pooling for consuming JMS in highly concurrent servers
* [Lingo](http://lingo.codehaus.org/) is a Spring/POJO remoting layer for JMS. It allows you to reuse all the power of JMS from your POJOs without using any of the JMS APIs directly
* [Spring](http://www.springframework.org/)
* [Stomp](http://stomp.codehaus.org/) is an open wire protocol (similar to HTTP) for communicating with MOMs from different languages. It has clients for languages like C, C#, Python, Perl, Ruby etc.
* [XBean](http://xbean.org/) is used as the default XML configuration mechanism for ActiveMQ

# Version 5 Installation

* [Download](http://activemq.apache.org/download.html) a binary distribution of ActiveMQ and unpack it into some directory.
* To run an ActiveMQ broker, type the following commands from the directory in which you have just unpacked the ActiveMQ distribution.

|  |
| --- |
| cd bin  activemq |

The ActiveMQ broker should now run. You can configure the broker by specifying an [Xml Configuration](http://activemq.apache.org/xml-configuration.html) file as a parameter to the *activemq*command.

You can now run the [Examples](http://activemq.apache.org/examples.html) using Ant.

See the [getting started guide](http://activemq.apache.org/initial-configuration.html) for details of which jars you need to add to your classpath to start using ActiveMQ in your Java code

If you want to use JNDI to connect to your JMS provider then please view the [JNDI Support](http://activemq.apache.org/jndi-support.html). If you are a Spring user you should read about [Spring Support](http://activemq.apache.org/spring-support.html)

# Version 5 Performance Tuning

For a more complete overview see [Performance](http://activemq.apache.org/performance.html).

There are trade-offs between performance and reliabilty.   
By default, activemq strikes a balance between the two, so there are some things you can change to increase throughput.

**Async publishing**

First some background:

When an ActiveMQ message producer sends a non-persistent message, its dispatched asynchronously (fire and forget) - but for persistent messages, the publisher will block until it gets a notification that the message has been processed (saved to the store - queued to be dispatched to any active consumers etc) by the broker. messages are dispatched with delivery mode set to be persistent by default (which is required by the JMS spec). So if you are sending messages on a Topic, the publisher will block by default (even if there are no durable subscribers on the topic) until the broker has returned a notification.

So if you looking for good performance with topic messages, either set the delivery mode on the publisher to be non-persistent, or set the **useAsyncSend** property on the ActiveMQ ConnectionFactory to be **true**.

**Pre-fetch sizes for Consumers**

ActiveMQ will push as many messages to the consumer as fast as possible, where they will be queued for processing by an ActiveMQ Session. The maximum number of messages that ActiveMQ will push to a Consumer without the Consumer processing a message is set by the pre-fetch size. You can improve throughput by running ActiveMQ with larger pre-fetch sizes. Pre-fetch sizes are determined by the ActiveMQPrefetchPolicy bean, which is set on the ActiveMQ ConnectionFactory.

Default values:

|  |  |
| --- | --- |
| consumer type | default value |
| queue | 1000 |
| queue browser | 500 |
| topic | 32766 |
| durable topic | 1000 |

**Optimized Acknowledge**

When consuming messages in auto acknowledge mode (set when creating the consumers' session), ActiveMQ will acknowledge receipt of messages back to the broker in batches (to improve performance). The batch size is 50% of the prefetch limit for the Consumer. You switch batch acknowledgment off by setting the **optimizeAcknowledge** property on the ActiveMQ ConnectionFactory to be **false**

**Straight through Session Consumption**

By default, a Consumer's session will dispatch messages to the consumer in a separate thread. If you are using Consumers with auto acknowledge, you can increase throughput by passing messages straight through the Session to the Consumer by setting the **alwaysSessionAsync** property on the ActiveMQ ConnectionFactory to be **false**

**File based Persistence**

The default persistence store is the [AMQ Message Store](http://activemq.apache.org/amq-message-store.html)

# Version 5 Run Broker

# Running an ActiveMQ Broker

Note if you want to use an **embedded broker** then see [How do I embed a Broker inside a Connection](http://activemq.apache.org/how-do-i-embed-a-broker-inside-a-connection.html)

The [binary distribution](http://activemq.apache.org/download.html) of ActiveMQ comes with a script called 'activemq' which allows you to run a broker.   
For details regarding the activemq init script file review  [Unix Shell Script](http://activemq.apache.org/unix-shell-script.html)  and  [ActiveMQ Command Line Tools Reference](http://activemq.apache.org/activemq-command-line-tools-reference.html)

Typing the following will run an ActiveMQ Broker using the out of the box configuration in the foreground

|  |
| --- |
| bin/activemq console |

You can then use a [Broker Configuration URI](http://activemq.apache.org/broker-configuration-uri.html) to specify how to start and configure your broker using a single URI. For example

|  |
| --- |
| bin/activemq console broker:(tcp://localhost:61616,network:static:tcp://remotehost:61616)?persistent=false&useJmx=true |

Or you can a [Broker XBean URI](http://activemq.apache.org/broker-xbean-uri.html) to customize the Message Broker using the [Xml Configuration](http://activemq.apache.org/xml-configuration.html) to suit your needs. You can run a broker with a specific XML configuration as

|  |
| --- |
| bin/activemq console xbean:foo.xml |

Or you can use a [Broker Properties URI](http://activemq.apache.org/broker-properties-uri.html) to customize the Message Broker using a properties file; which avoids the dependency on Spring, xbean-spring and XML.

|  |
| --- |
| bin/activemq console properties:foo.properties |

### Monitoring the broker

You can monitor ActiveMQ using the [Web Console](http://activemq.apache.org/web-console.html) by pointing your browser at

<http://localhost:8161/admin>

From ActiveMQ 5.8 onwards the web apps is secured out of the box.  
The default username and password is admin/admin. You can configure this in the conf/jetty-real.properties file.

Or you can use the [JMX](http://activemq.apache.org/jmx.html) support to view the running state of ActiveMQ.

For more information see the file docs/WebConsole-README.txt in the distribution.

### Running the broker inside a Servlet Engine

See the source code (or WAR) of the [Web Console](http://activemq.apache.org/web-console.html) for an example of how to run the broker inside a web application using Spring.

### Running the broker inside your J2EE Application Server

Whether its Apache Geronmio, JBoss, WebLogic or some other J2EE container you should be able to just reconfigure and then deploy the activemq-\*.rar which is included in the binary distribution as a deployment unit in your app server. By default the rar is not configured to start an embedded broker. But by setting the brokerXmlConfig on the resource adapter configuration, the resource adapter will start an embedded broker.

For more details see [J2EE](http://activemq.apache.org/j2ee.html)

### Running the broker from the source code

From the latest [checkout](http://activemq.apache.org/source.html) of the code you can run a broker using the [ActiveMQ Performance Plugin](http://activemq.apache.org/activemq-performance-module-users-manual.html)

### Running the broker from maven

You can download and install the ActiveMQ Startup Maven Plugin via the following command if you are in a directory with a pom.xml. More detailed usage [here](http://activemq.apache.org/maven2-activemq-broker-plugin.html)

|  |
| --- |
| mvn org.apache.activemq.tooling:maven-activemq-plugin:5.0-SNAPSHOT:run |

You can also include it the pom and run it using:

|  |
| --- |
| mvn activemq:run |

## Handling JMS brokers going down

A common requirement is that if the JMS broker goes down you want to automatically detect the failure and try to reconnect under the covers so that your application does not have to worry about reconnection.

There is detailed documentation on this in [Configuring Version 5 Transports](http://activemq.apache.org/configuring-version-5-transports.html); briefly...

Just change your connection URI i to

|  |
| --- |
| failover:tcp://host:port |

And the JMS client will auto-reconnect to the broker if it is shutdown and restarted later on.

# Version 5 Topologies

ActiveMQ supports a wide range of different deployment topologies as well as [protocols](http://activemq.apache.org/uri-protocols.html) & wire formats. The following diagram shows a federated network of brokers with a few different kinds of topology.

Which topology you choose is up to you. We'll now describe a few of these protocols in a little more detail.

## In VM

A useful option when unit testing is to limit JMS communication to within a single JVM. For this use the protocol

|  |
| --- |
| vm://localhost |

You can segment the VM protocol to different groups - e.g. if you want to have logically different JMS networks within the same JVM, you can group networks using different URIs. e.g.

|  |
| --- |
| vm://localhost/foo |

This will ensure that different segments do not interfere with one another. Though typically we use unique topic and queue destinations so that all traffic can coexist happily on the same logical network.

## Client-Server

This is probably the most efficient and fastest solution for large numbers of clients requiring a diverse range of communication options from publish / subscribe to queue based communication. Typically the clients will connect with a Message Broker using a protocol, typically TCP or SSL but this could be NIO or other protocols.

We can load balance clients across brokers and provide broker failover so that we have a logical cluster of brokers with [HA](http://activemq.apache.org/ha.html).

e.g.

|  |
| --- |
| tcp://somehost:port |

Or for SSL

|  |
| --- |
| ssl://somehost:port |

You can use [Discovery](http://activemq.apache.org/discovery.html) to find the brokers available that you can connect to which makes it easier to seamlessly connect to a cluster of brokers.

## Embedded Broker

This is logically equivalent to Client-Server but some (or all) clients contain a locally embedded broker. So communcation between the client and server (broker) are all within the same JVM and so do not use real networking - though brokers may communicate with other brokers or clients connected to it.

This can avoid the extra hop required to go from producer to broker to consumer - which is a great optimisation for RMI / RPC style situations, where you want the performance benefits (reduced latency) of point to point networking but with the scalabilty of a flexible messaging fabric.

Embedded Brokers can also simplify deployment options as well, its one less process to run.

Another use case for embedded brokers is to provide store and forward isolation from each service - so that the remote brokers can fail quite happily without affecting the service with the embedded broker. e.g. the entire network could fail, but a service could continue publishing messages to its embedded broker.

You can find out how to [configure an embedded broker here](http://activemq.apache.org/how-do-i-embed-a-broker-inside-a-connection.html)

## Peer to Peer

This allows peer based clusters to be created where there is no server - just clients connecting together.  
There are various ways to implement a peer to peer JMS network. One easy way is just to use a multicast transport for communication; then all nodes on the same multicast address will receive all messages and the local embedded message broker will route messages to the necessary MessageConsumers.

We currently have 3 choices for multicast protocols

* multicast
* jgroups: uses the JGroups library to implement reliable multicast
* jrms: uses Sun's JRMS library to implement reliable multicast

Multicast is great in development though often you might want to disable this feature in production and have well known servers fixed on specific machines. Often socket based communication (using pointcast) is much faster & better for heavy-lifting - particularly on Java - so we tend to recommend to use multicast mostly for discovery and use TCP / SSL for your heavy duty messaging.

Often we can use the peer to peer topology as a bootstrap to create a cluster of clients & brokers and then autodeploy servers into the cluster for a true grid style network.

So you can get the effect of a peer based network using [Discovery](http://activemq.apache.org/discovery.html) together with either stand alone Brokers or using embedded brokers.

### JXTA

We have a JXTA transport which will use the full JXTA stack for negotiating NAT and across firewalls and so forth for creating a true peer based JMS network.

|  |
| --- |
| jxta://hostname:port |

Currently you need to run one server which everyone connects to via JXTA. We've not yet created a pure peer network with JXTA

# Version 5 Web Samples

There are a few example programs demonstrating the [REST](http://activemq.apache.org/rest.html) messaging or [Ajax](http://activemq.apache.org/ajax.html) in the [activemq/activemq-web-demo](https://svn.apache.org/repos/asf/activemq/trunk/activemq-web-demo/) module.

## Running the Web Samples

We have integrated the Web Samples into the binary distribution.

The steps to running the Web Samples are

* [Download](http://activemq.apache.org/download.html) a binary distribution
* [Run the Message Broker](http://activemq.apache.org/run-broker.html) e.g. by running bin/activemq
* point your browser to <http://localhost:8161/demo>

# Version 5 XML Configuration

* transport connectors which consist of transport channels and wire formats TODO: add a link to a page explaining what transport connectors are how to configure and use them.
* network connectors using network channels or discovery TODO: add a link to a page explaining what network connectors are how to configure and use them.
* discovery agents TODO: add a link to a page explaining what discovery agents are how to configure and use them.
* persistence providers & locations TODO: add a link to a page explaining what persistence providers are how to configure and use them.
* custom message containers (such as last image caching etc)

We use [XBean](http://xbean.org/) to perform the XML configuration.

For details of the XML see the [Xml Reference](http://activemq.apache.org/xml-reference.html)

Be careful with broker names and URIs

Make sure you do not use any strange characters in the names of brokers as they are converted to URIs which [do not allow things like underscores](http://java.sun.com/j2se/1.4.2/docs/api/java/net/URI.html) in them etc.

## Examples

The default ActiveMQ configuration: [current default config](http://svn.apache.org/repos/asf/activemq/trunk/assembly/src/release/conf/activemq.xml).

xml<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://activemq.apache.org/schema/core http://activemq.apache.org/schema/core/activemq-core.xsd"> <!-- Allows us to use system properties as variables in this configuration file --> <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"> <property name="locations"> <value>file:${activemq.conf}/credentials.properties</value> </property> </bean> <!-- Allows accessing the server log --> <bean id="logQuery" class="io.fabric8.insight.log.log4j.Log4jLogQuery" lazy-init="false" scope="singleton" init-method="start" destroy-method="stop"> </bean> <!-- The <broker> element is used to configure the ActiveMQ broker. --> <broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost" dataDirectory="${activemq.data}"> <destinationPolicy> <policyMap> <policyEntries> <policyEntry topic=">" > <!-- The constantPendingMessageLimitStrategy is used to prevent slow topic consumers to block producers and affect other consumers by limiting the number of messages that are retained For more information, see: http://activemq.apache.org/slow-consumer-handling.html --> <pendingMessageLimitStrategy> <constantPendingMessageLimitStrategy limit="1000"/> </pendingMessageLimitStrategy> </policyEntry> </policyEntries> </policyMap> </destinationPolicy> <!-- The managementContext is used to configure how ActiveMQ is exposed in JMX. By default, ActiveMQ uses the MBean server that is started by the JVM. For more information, see: http://activemq.apache.org/jmx.html --> <managementContext> <managementContext createConnector="false"/> </managementContext> <!-- Configure message persistence for the broker. The default persistence mechanism is the KahaDB store (identified by the kahaDB tag). For more information, see: http://activemq.apache.org/persistence.html --> <persistenceAdapter> <kahaDB directory="${activemq.data}/kahadb"/> </persistenceAdapter> <!-- The systemUsage controls the maximum amount of space the broker will use before disabling caching and/or slowing down producers. For more information, see: http://activemq.apache.org/producer-flow-control.html --> <systemUsage> <systemUsage> <memoryUsage> <memoryUsage percentOfJvmHeap="70" /> </memoryUsage> <storeUsage> <storeUsage limit="100 gb"/> </storeUsage> <tempUsage> <tempUsage limit="50 gb"/> </tempUsage> </systemUsage> </systemUsage> <!-- The transport connectors expose ActiveMQ over a given protocol to clients and other brokers. For more information, see: http://activemq.apache.org/configuring-transports.html --> <transportConnectors> <!-- DOS protection, limit concurrent connections to 1000 and frame size to 100MB --> <transportConnector name="openwire" uri="tcp://0.0.0.0:61616?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/> <transportConnector name="amqp" uri="amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/> <transportConnector name="stomp" uri="stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/> <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/> <transportConnector name="ws" uri="ws://0.0.0.0:61614?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600"/> </transportConnectors> <!-- destroy the spring context on shutdown to stop jetty --> <shutdownHooks> <bean xmlns="http://www.springframework.org/schema/beans" class="org.apache.activemq.hooks.SpringContextHook" /> </shutdownHooks> </broker> <!-- Enable web consoles, REST and Ajax APIs and demos The web consoles requires by default login, you can disable this in the jetty.xml file Take a look at ${ACTIVEMQ\_HOME}/conf/jetty.xml for more details --> <import resource="jetty.xml"/> </beans>

From a binary distribution there is an activemq script allowing you to run a Message Broker as a stand alone process from the command line easily providing the $ACTIVEMQ\_HOME/bin directory is on your PATH.

## Configuring embedded brokers

You can also use the XML Configuration to configure [embedded brokers](http://activemq.apache.org/how-do-i-embed-a-broker-inside-a-connection.html). For example using the JNDI configuration mechanism you can do the following  
[BrokerXmlConfigFromJNDITest](http://svn.apache.org/repos/asf/activemq/trunk/assembly/src/test/java/org/apache/activemq/config/BrokerXmlConfigFromJNDITest.java)  
Or of you want to explicitly configure the embedded broker via Java code you can do the following  
[BrokerXmlConfigTest](http://svn.apache.org/repos/asf/activemq/trunk/assembly/src/test/java/org/apache/activemq/config/BrokerXmlConfigTest.java)

## User Submitted Configurations

We have a page which allows users to submit details of their configurations.

* [User Submitted Configurations](http://activemq.apache.org/user-submitted-configurations.html)

## Background

Since ActiveMQ has so many strategy pattern plugins for transports, wire formats, persistence and many other things, we wanted to leave the configuration format open so that you the developer can configure and extend ActiveMQ in any direction you wish.

So we use the [Spring XML](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-basics) configuration file format, which allows any beans / POJOs to be wired together and configured. However often Spring's XML can be kinda verbose at times, so we have implemented an ActiveMQ extension to the Spring XML which knows about the common, standard ActiveMQ things you're likely to do (e.g. tags like connector, wireFormat, serverTransport, persistence) - but at any time you can fall back to the normal Spring way of doing things (with tags like bean, property etc).

To see documentation of the XML file we use or to get access to the XSD/DTD see the [Xml Reference](http://activemq.apache.org/xml-reference.html)