Table of Contents

Apache ActiveMQ Artemis User Manual

The User manual is an in depth manual on all aspects of Apache ActiveMQ Artemis

Legal Notice

Preface

What is Apache ActiveMQ Artemis?

- Apache ActiveMQ Artemis is an open source project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system.
- Apache ActiveMQ Artemis is an example of Message Oriented Middleware (MoM). For a description of MoMs and other messaging concepts please see the Messaging Concepts.

Why use Apache ActiveMQ Artemis? Here are just a few of the reasons:

- 100% open source software. Apache ActiveMQ Artemis is licensed using the Apache Software License v 2.0 to minimise barriers to adoption.
- Apache ActiveMQ Artemis is designed with usability in mind.
- Written in Java. Runs on any platform with a Java 8+ runtime, that's everything from Windows desktops to IBM mainframes.
- Amazing performance. Our ground-breaking high performance journal provides persistent messaging performance at rates normally seen for non-persistent messaging, our non-persistent messaging performance rocks the boat too.
- Full feature set. All the features you'd expect in any serious messaging system, and others you won't find anywhere else.
- Elegant, clean-cut design with minimal third party dependencies. Run ActiveMQ Artemis stand-alone, run it in integrated in your favourite Java EE application server, or run it embedded inside your own product. It's up to you.
- Seamless high availability. We provide a HA solution with automatic client failover so you can guarantee zero message loss or duplication in event of server failure.
- Hugely flexible clustering. Create clusters of servers that know how to load balance messages. Link geographically distributed clusters over unreliable connections to form a global network. Configure routing of messages in a highly flexible way.

Project Information

The official Apache ActiveMQ Artemis project page is http://activemq.apache.org/artemis/.

Software Download

The software can be download from the Download page:http://activemq.apache.org/artemis/download.html

Project Information

- If you have any user questions please use our user forum
- If you have development related questions, please use our developer forum
- Pop in and chat to us in our IRC channel
- Apache ActiveMQ Artemis Git repository is https://github.com/apache/activemq-artemis
- All release tags are available from https://github.com/apache/activemq-artemis/releases

And many thanks to all our contributors, both old and new who helped create Apache ActiveMQ Artemis.

Versions

This chapter provides the following information for each release:

- A link to the full release notes which includes all issues resolved in the release.
- A brief list of "highlights" when applicable.
- If necessary, specific steps required when upgrading from the previous version.
  - **Note:** If the upgrade spans multiple versions then the steps from **each** version need to be followed in order.
  - **Note:** Follow the general upgrade procedure outlined in the Upgrading the Broker chapter in addition to any version-specific upgrade instructions outlined here.

2.14.0

Full release notes.

Highlights:

- Removing `broker.xml` queue parameters now causes these to be reset to their default values.

Upgrading from older versions

Make sure the existing queues have their parameters set according to the `broker.xml` values before upgrading.

2.11.0

Full release notes.

Highlights:

- Support retroactive addresses.
- Support downstream federated queues and addresses.
- Make security manager configurable via XML.
- Support pluggable SSL TrustManagerFactory.
- Add plugin support for federated queues/addresses.
- Support `com.sun.jndi.ldap.read.timeout` in LDAPLoginModule.

2.10.0

Full release notes.

This was mainly a bug-fix release with a notable dependency change impacting version upgrade.

Upgrading from 2.9.0

Due to the WildFly dependency upgrade the broker start scripts/configuration need to be adjusted after upgrading.

On *nix

Locate this statement in `bin/artemis`:

```
WILDFLY_COMMON="$ARTEMIS_HOME/lib/wildfly-common-1.5.1.Final.jar"
```

This needs to be replaced with this:

```
WILDFLY_COMMON="$ARTEMIS_HOME/lib/wildfly-common-1.5.2.Final.jar"
```

On Windows

Locate this part of `JAVA_ARGS` in `etc/artemis.profile.cmd` respectively `bin/artemis-service.xml`:

```
%ARTEMIS_HOME%\lib\wildfly-common-1.5.1.Final.jar
```

This needs to be replaced with this:

```
%ARTEMIS_HOME%\lib\wildfly-common-1.5.2.Final.jar
```

2.9.0

Full release notes.

This was a light release. It included a handful of bug fixes, a few improvements, and one major new feature.

Highlights:

- Support exporting metrics.

2.8.1

Full release notes.

This was mainly a bug-fix release with a notable dependency change impacting version upgrade.

Upgrading from 2.8.0

Due to the dependency upgrade made on ARTEMIS-2319 the broker start scripts need to be adjusted after upgrading.

On *nix

Locate this `if` statement in `bin/artemis`:

```
if [ -z "$LOG_MANAGER" ] ; then
 # this is the one found when the server was created
 LOG_MANAGER="$ARTEMIS_HOME/lib/jboss-logmanager-2.0.3.Final.jar"
fi
```

This needs to be replaced with this block:

```
if [ -z "$LOG_MANAGER" ] ; then
 # this is the one found when the server was created
 LOG_MANAGER="$ARTEMIS_HOME/lib/jboss-logmanager-2.1.10.Final.jar"
fi

WILDFLY_COMMON=`ls $ARTEMIS_HOME/lib/wildfly-common*jar 2>/dev/null`
if [ -z "$WILDFLY_COMMON" ] ; then
 # this is the one found when the server was created
 WILDFLY_COMMON="$ARTEMIS_HOME/lib/wildfly-common-1.5.1.Final.jar"
fi
```

Notice that the `jboss-logmanager` version has changed and there is also a new `wildfly-common` library.

Not much further down there is this line:

```
-Xbootclasspath/a:"$LOG_MANAGER" \
```

This line should be changed to be:

```
-Xbootclasspath/a:"$LOG_MANAGER:$WILDFLY_COMMON" \
```

On Windows

Locate this part of `JAVA_ARGS` in `etc/artemis.profile.cmd` respectively `bin/artemis-service.xml`:

```
-Xbootclasspath/a:%ARTEMIS_HOME%\lib\jboss-logmanager-2.1.10.Final.jar
```

This needs to be replaced with this:

```
-Xbootclasspath/a:%ARTEMIS_HOME%\lib\jboss-logmanager-2.1.10.Final.jar;%ARTEMI
```

2.8.0

Full release notes.

Highlights:

- Support ActiveMQ5 feature JMSXGroupFirstForConsumer.
- Clarify handshake timeout error with remote address.
- Support duplicate detection for AMQP messages the same as core.

2.7.0

Full release notes.

Highlights:

- Support advanced destination options like `consumersBeforeDispatchStarts` and `timeBeforeDispatchStarts` from 5.x.
- Add support for delays before deleting addresses and queues via `auto-delete-queues-delay` and `auto-delete-addresses-delay` Address Settings.
- Support logging HTTP access.
- Add a CLI command to purge a queue.
- Support user and role manipulation for PropertiesLoginModule via management interfaces.
- Docker images.
- Audit logging.
- Implementing consumer priority.
- Support FQQN for producers.
- Track routed and unrouted messages sent to an address.
- Support connection pooling in LDAPLoginModule.
- Support configuring a default consumer window size via `default-consumer-window-size` Address Setting.
- Support masking `key-store-password` and `trust-store-password` in management.xml.
- Support `JMSXGroupSeq` -1 to close/reset message groups from 5.x.
- Allow configuration of RMI registry port.
- Support routing-type configuration on core bridge.
- Move artemis-native as its own project, as activemq-artemis-native.
- Support federated queues and addresses.

2.6.4

Full release notes.

This was mainly a bug-fix release with a few improvements a couple notable new features:

Highlights:

- Added the ability to set the text message content on the `producer` CLI command.
- Support reload logging configuration at runtime.

2.6.3

Full release notes.

This was mainly a bug-fix release with a few improvements but no substantial new features.

2.6.2

Full release notes.

This was a bug-fix release with no substantial new features or improvements.

2.6.1

Full release notes.

This was a bug-fix release with no substantial new features or improvements.

2.6.0

Full release notes.

Highlights:

- Support regular expressions for matching client certificates.
- Support `SASL_EXTERNAL` for AMQP clients.
- New examples showing virtual topic mapping and exclusive queue features.

2.5.0

Full release notes.

Highlights:

- Exclusive consumers.
- Equivalent ActiveMQ 5.x Virtual Topic naming abilities.
- SSL Certificate revocation list.
- Last-value queue support for OpenWire.
- Support masked passwords in bootstrap.xm and login.config
- Configurable broker plugin implementation for logging various broker events (i.e. `LoggingActiveMQServerPlugin`).
- Option to use OpenSSL provider for Netty via the `sslProvider` URL parameter.
- Enable splitting of broker.xml into multiple files.
- Enhanced message count and size metrics for queues.

Upgrading from 2.4.0

1. Due to changes from ARTEMIS-1644 any `acceptor` that needs to be compatible with HornetQ and/or Artemis 1.x clients needs to have

`anycastPrefix=jms.queue.;multicastPrefix=jms.topic.` in the `acceptor` url. This prefix used to be configured automatically behind the scenes when the broker detected these old types of clients, but that broke certain use-cases with no possible work-around. See ARTEMIS-1644 for more details.

2.4.0

Full release notes.

Highlights:

- JMX configuration via XML rather than having to use system properties via command line or start script.
- Configuration of max frame payload length for STOMP web-socket.
- Ability to configure HA using JDBC persistence.
- Implement role-based access control for management objects.

Upgrading from 2.3.0

1. Create `<ARTEMIS_INSTANCE>/etc/management.xml`. At the very least, the file must contain this:

   ```
   <management-context xmlns="http://activemq.org/schema"/>
   ```

   This configures role based authorisation for JMX. Read more in the Management documentation.
2. If configured, remove the Jolokia war file from the `web` element in `<ARTEMIS_INSTANCE>/etc/bootstrap.xml`:

   ```
   <app url="jolokia" war="jolokia.war"/>
   ```

   This is no longer required as the Jolokia REST interface is now integrated into the console web application.

   If the following is absent and you desire to deploy the web console then add:

   ```
   <app url="console" war="console.war"/>
   ```

   Note: the Jolokia REST interface URL will now be at `http://<host>:<port>/console/jolokia`

2.3.0

Full release notes.

Highlights:

- Web admin console!
- Critical Analysis and deadlock detection on broker
- Support Netty native kqueue on Mac.

- Last-value queue for AMQP

Upgrading from 2.2.0

1. If you desire to deploy the web console then add the following to the `web` element in `<ARTEMIS_INSTANCE>/etc/bootstrap.xml`:

```
<app url="console" war="console.war"/>
```

2.2.0

Full release notes.

Highlights:

- Scheduled messages with the STOMP protocol.
- Support for JNDIReferenceFactory and JNDIStorable.
- Ability to delete queues and addresses when broker.xml changes.
- Client authentication via Kerberos TLS Cipher Suites (RFC 2712).

2.1.0

Full release notes.

Highlights:

- Broker plugin support.
- Support Netty native epoll on Linux.
- Ability to configure arbitrary security role mappings.
- AMQP performance improvements.

2.0.0

Full release notes.

Highlights:

- Huge update involving a significant refactoring of the addressing model yielding the following benefits:
  - Simpler and more flexible XML configuration.
  - Support for additional messaging use-cases.
  - Eliminates confusing JMS-specific queue naming conventions (i.e. "jms.queue." & "jms.topic." prefixes).
- Pure encoding of messages so protocols like AMQP don't need to convert messages to "core" format unless absolutely necessary.
- "MAPPED" journal type for increased performance in certain use-cases.

1.5.6

Full release notes.

Highlights:

- Bug fixes.

1.5.5

Full release notes.

Highlights:

- Bug fixes.

1.5.4

Full release notes.

Highlights:

- Support Oracle12C for JDBC persistence.
- Bug fixes.

1.5.3

Full release notes.

Highlights:

- Support "byte notation" (e.g. "K", "KB", "Gb", etc.) in broker XML configuration.
- CLI command to recalculate disk sync times.
- Bug fixes.

1.5.2

Full release notes.

Highlights:

- Support for paging using JDBC.
- Bug fixes.

1.5.1

Full release notes.

Highlights:

- Support outgoing connections for AMQP.
- Bug fixes.

1.5.0

Full release notes.

Highlights:

- AMQP performance improvements.
- JUnit rule implementation so messaging resources like brokers can be easily configured in tests.
- Basic CDI integration.
- Store user's password in hash form by default.

1.4.0

Full release notes.

Highlights:

- "Global" limit for disk usage.
- Detect and reload certain XML configuration changes at runtime.
- MQTT interceptors.
- Support adding/deleting queues via CLI.
- New "browse" security permission for clients who only wish to look at messages.
- Option to populate JMSXUserID.
- "Dual authentication" support to authenticate SSL-based and non-SSL-based clients differently.

1.3.0

Full release notes.

Highlights:

- Better support of OpenWire features (e.g. reconnect, producer flow-control, optimized acknowledgements)
- SSL keystore reload at runtime.
- Initial support for JDBC persistence.
- Support scheduled messages on last-value queue.

1.2.0

Full release notes.

Highlights:

- Improvements around performance
- OSGi support.
- Support functionality equivalent to all 5.x JAAS login modules including:
  - Properties file

  - LDAP
  - SSL certificate
  - "Guest"

1.1.0

Full release notes.

Highlights:

- MQTT support.
- The examples now use the CLI programmatically to create, start, stop, etc. servers reflecting real cases used in production.
- CLI improvements. There are new tools to compact the journal and additional improvements to the user experience.
- Configurable resource limits.
- Ability to disable server-side message load-balancing.

1.0.0

Full release notes.

Highlights:

- First release of the donated code-base as ActiveMQ Artemis!
- Lots of features for parity with ActiveMQ 5.x including:
  - OpenWire support
  - AMQP 1.0 support
  - URL based connections
  - Auto-create addresses/queues
  - Jolokia integration

Messaging Concepts

Apache ActiveMQ Artemis is an asynchronous messaging system, an example of Message Oriented Middleware , we'll just call them messaging systems in the remainder of this book.

We'll first present a brief overview of what kind of things messaging systems do, where they're useful and the kind of concepts you'll hear about in the messaging world.

If you're already familiar with what a messaging system is and what it's capable of, then you can skip this chapter.

General Concepts

Messaging systems allow you to loosely couple heterogeneous systems together, whilst typically providing reliability, transactions and many other features.

Unlike systems based on a Remote Procedure Call (RPC) pattern, messaging systems primarily use an asynchronous message passing pattern with no tight relationship between requests and responses. Most messaging systems also support a request-response mode but this is not a primary feature of messaging systems.

Designing systems to be asynchronous from end-to-end allows you to really take advantage of your hardware resources, minimizing the amount of threads blocking on IO operations, and to use your network bandwidth to its full capacity. With an RPC approach you have to wait for a response for each request you make so are limited by the network round trip time, or *latency* of your network. With an asynchronous system you can pipeline flows of messages in different directions, so are limited by the network *bandwidth* not the latency. This typically allows you to create much higher performance applications.

Messaging systems decouple the senders of messages from the consumers of messages. The senders and consumers of messages are completely independent and know nothing of each other. This allows you to create flexible, loosely coupled systems.

Often, large enterprises use a messaging system to implement a message bus which loosely couples heterogeneous systems together. Message buses often form the core of an Enterprise Service Bus. (ESB). Using a message bus to de-couple disparate systems can allow the system to grow and adapt more easily. It also allows more flexibility to add new systems or retire old ones since they don't have brittle dependencies on each other.

Messaging styles

Messaging systems normally support two main styles of asynchronous messaging: message queue messaging (also known as *point-to-point messaging*) and publish subscribe messaging. We'll summarise them briefly here:

Point-to-Point

With this type of messaging you send a message to a queue. The message is then typically persisted to provide a guarantee of delivery, then some time later the messaging system delivers the message to a consumer. The consumer then processes the message and when it is done, it acknowledges the message. Once the message is acknowledged it disappears from the queue and is not available to be delivered again. If the system crashes before the messaging server receives an acknowledgement from the consumer, then on recovery, the message will be available to be delivered to a consumer again.

With point-to-point messaging, there can be many consumers on the queue but a particular message will only ever be consumed by a maximum of one of them. Senders (also known as *producers*) to the queue are completely decoupled from receivers (also known as *consumers*) of the queue - they do not know of each other's existence.

A classic example of point to point messaging would be an order queue in a company's book ordering system. Each order is represented as a message which is sent to the order queue. Let's imagine there are many front end ordering systems which send orders to the order queue. When a message arrives on the queue it is persisted - this ensures that if the server crashes the order is not lost. Let's also imagine there are many consumers on the order queue - each representing an instance of an order processing component - these can be on different physical machines but consuming from the same queue. The messaging system delivers each message to one and only one of the ordering processing components. Different messages can be processed by different order processors, but a single order is only processed by one order processor - this ensures orders aren't processed twice.

As an order processor receives a message, it fulfills the order, sends order information to the warehouse system and then updates the order database with the order details. Once it's done that it acknowledges the message to tell the server that the order has been processed and can be forgotten about. Often the send to the warehouse system, update in database and acknowledgement will be completed in a single transaction to ensure ACID properties.

Publish-Subscribe

With publish-subscribe messaging many senders can send messages to an entity on the server, often called a *topic* (e.g. in the JMS world).

There can be many *subscriptions* on a topic, a subscription is just another word for a consumer of a topic. Each subscription receives a *copy* of *each* message sent to the topic. This differs from the message queue pattern where each message is only consumed by a single consumer.

Subscriptions can optionally be *durable* which means they retain a copy of each message sent to the topic until the subscriber consumes them - even if the server crashes or is restarted in between. Non-durable subscriptions only last a maximum of the lifetime of the connection that created them.

An example of publish-subscribe messaging would be a news feed. As news articles are created by different editors around the world they are sent to a news feed topic. There are many subscribers around the world who are interested in receiving news items - each one creates a subscription and the messaging system ensures that a copy of each news message is delivered to each subscription.

Delivery guarantees

A key feature of most messaging systems is *reliable messaging*. With reliable messaging the server gives a guarantee that the message will be delivered once and only once to each consumer of a queue or each durable subscription of a topic, even in the event of system failure. This is crucial for many businesses; e.g. you don't want your orders fulfilled more than once or any of your orders to be lost.

In other cases you may not care about a once and only once delivery guarantee and are happy to cope with duplicate deliveries or lost messages - an example of this might be transient stock price updates - which are quickly superseded by the next update on the same stock. The messaging system allows you to configure which delivery guarantees you require.

Transactions

Messaging systems typically support the sending and acknowledgement of multiple messages in a single local transaction. Apache ActiveMQ Artemis also supports the sending and acknowledgement of message as part of a large global transaction - using the Java mapping of XA: JTA.

Durability

Messages are either durable or non durable. Durable messages will be persisted in permanent storage and will survive server failure or restart. Non durable messages will not survive server failure or restart. Examples of durable messages might be orders or trades, where they cannot be lost. An example of a non durable message might be a stock price update which is transitory and doesn't need to survive a restart.

Messaging APIs and protocols

How do client applications interact with messaging systems in order to send and consume messages?

Several messaging systems provide their own proprietary APIs with which the client communicates with the messaging system.

There are also some standard ways of operating with messaging systems and some emerging standards in this space.

Let's take a brief look at these:

Java Message Service (JMS)

JMS is part of Oracle's Java EE specification. It's a Java API that encapsulates both message queue and publish-subscribe messaging patterns. JMS is a lowest common denominator specification - i.e. it was created to encapsulate common functionality of the already existing messaging systems that were available at the time of its creation.

JMS is a very popular API and is implemented by most messaging systems. JMS is only available to clients running Java.

JMS does not define a standard wire format - it only defines a programmatic API so JMS clients and servers from different vendors cannot directly interoperate since each will use the vendor's own internal wire protocol.

Apache ActiveMQ Artemis provides a fully compliant JMS 1.1 and JMS 2.0 client implementation.

System specific APIs

Many systems provide their own programmatic API for which to interact with the messaging system. The advantage of this it allows the full set of system functionality to be exposed to the client application. API's like JMS are not normally rich enough to expose all the extra features that most messaging systems provide.

Apache ActiveMQ Artemis provides its own core client API for clients to use if they wish to have access to functionality over and above that accessible via the JMS API.

Please see Core for using the Core API with Apache ActiveMQ Artemis.

RESTful API

REST approaches to messaging are showing a lot interest recently.

It seems plausible that API standards for cloud computing may converge on a REST style set of interfaces and consequently a REST messaging approach is a very strong contender for becoming the de-facto method for messaging interoperability.

With a REST approach messaging resources are manipulated as resources defined by a URI and typically using a simple set of operations on those resources, e.g. PUT, POST, GET etc. REST approaches to messaging often use HTTP as their underlying protocol.

The advantage of a REST approach with HTTP is in its simplicity and the fact the internet is already tuned to deal with HTTP optimally.

Please see Rest Interface for using Apache ActiveMQ Artemis's RESTful interface.

AMQP

AMQP is a specification for interoperable messaging. It also defines a wire format, so any AMQP client can work with any messaging system that supports AMQP. AMQP clients are available in many different programming languages.

Apache ActiveMQ Artemis implements the AMQP 1.0 specification. Any client that supports the 1.0 specification will be able to interact with Apache ActiveMQ Artemis.

Please see AMQP for using AMQP with Apache ActiveMQ Artemis.

MQTT

MQTT is a lightweight connectivity protocol. It is designed to run in environments where device and networks are constrained. Out of the box Apache ActiveMQ Artemis supports version MQTT 3.1.1. Any client supporting this version of the protocol will work against Apache ActiveMQ Artemis.

Please see MQTT for using MQTT with Apache ActiveMQ Artemis.

STOMP

Stomp is a very simple text protocol for interoperating with messaging systems. It defines a wire format, so theoretically any Stomp client can work with any messaging system that supports Stomp. Stomp clients are available in many different programming languages.

Please see Stomp for using STOMP with Apache ActiveMQ Artemis.

OpenWire

ActiveMQ 5.x defines its own wire protocol: OpenWire. In order to support ActiveMQ 5.x clients, Apache ActiveMQ Artemis supports OpenWire. Any ActiveMQ 5.12.x or higher can be used with Apache ActiveMQ Artemis.

Please see OpenWire for using OpenWire with Apache ActiveMQ Artemis.

High Availability

High Availability (HA) means that the system should remain operational after failure of one or more of the servers. The degree of support for HA varies between various messaging systems.

Apache ActiveMQ Artemis provides automatic failover where your sessions are automatically reconnected to the backup server on event of live server failure.

For more information on HA, please see High Availability and Failover.

Clusters

Many messaging systems allow you to create groups of messaging servers called *clusters*. Clusters allow the load of sending and consuming messages to be spread over many servers. This allows your system to scale horizontally by adding new servers to the cluster.

Degrees of support for clusters varies between messaging systems, with some systems having fairly basic clusters with the cluster members being hardly aware of each other.

Apache ActiveMQ Artemis provides very configurable state-of-the-art clustering model where messages can be intelligently load balanced between the servers in the cluster, according to the number of consumers on each node, and whether they are ready for messages.

Apache ActiveMQ Artemis also has the ability to automatically redistribute messages between nodes of a cluster to prevent starvation on any particular node.

For full details on clustering, please see Clusters.

Bridges and routing

Some messaging systems allow isolated clusters or single nodes to be bridged together, typically over unreliable connections like a wide area network (WAN), or the internet.

A bridge normally consumes from a queue on one server and forwards messages to another queue on a different server. Bridges cope with unreliable connections, automatically reconnecting when the connections becomes available again.

Apache ActiveMQ Artemis bridges can be configured with filter expressions to only forward certain messages, and transformation can also be hooked in.

Apache ActiveMQ Artemis also allows routing between queues to be configured in server side configuration. This allows complex routing networks to be set up forwarding or copying messages from one destination to another, forming a global network of interconnected brokers.

For more information please see Core Bridges and Diverting and Splitting Message Flows.

Core Architecture

Apache ActiveMQ Artemis core is designed simply as set of Plain Old Java Objects (POJOs) - we hope you like its clean-cut design.

Each Apache ActiveMQ Artemis server has its own ultra high performance persistent journal, which it uses for message and other persistence.

Using a high performance journal allows outrageous persistence message performance, something not achievable when using a relational database for persistence (although JDBC is still an option if necessary).

Apache ActiveMQ Artemis clients, potentially on different physical machines, interact with the Apache ActiveMQ Artemis broker. Apache ActiveMQ Artemis currently ships two API implementations for messaging at the client side:

1. Core client API. This is a simple intuitive Java API that is aligned with the Artemis internal Core. Allowing more control of broker objects (e.g direct creation of addresses and queues). The Core API also offers a full set of messaging functionality without some of the complexities of JMS.
2. JMS 2.0 client API. The standard JMS API is available at the client side.

Apache ActiveMQ Artemis also provides different protocol implementations on the server so you can use respective clients for these protocols:

- AMQP
- OpenWire
- MQTT
- STOMP
- HornetQ (for use with HornetQ clients).
- Core (Artemis CORE protocol)

JMS semantics are implemented by a JMS facade layer on the client side.

The Apache ActiveMQ Artemis broker does not speak JMS and in fact does not know anything about JMS, it is a protocol agnostic messaging server designed to be used with multiple different protocols.

When a user uses the JMS API on the client side, all JMS interactions are translated into operations on the Apache ActiveMQ Artemis core client API before being transferred over the wire using the core protocol.

The broker always just deals with core API interactions.

A schematic illustrating this relationship is shown in figure 3.1 below:

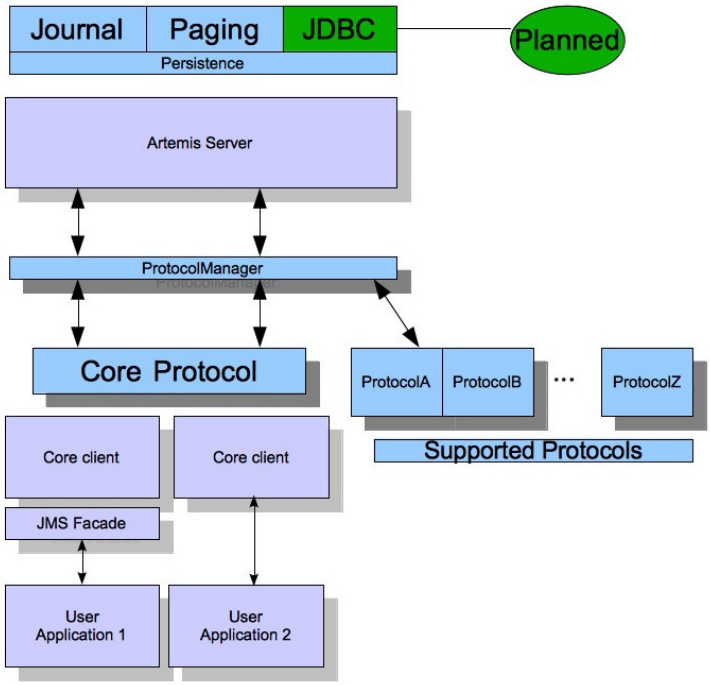


Figure 3.1 Artemis High Level Architecture

Figure 3.1 shows two user applications interacting with an Apache ActiveMQ Artemis server. User Application 1 is using the JMS API, while User Application 2 is using the core client API directly.

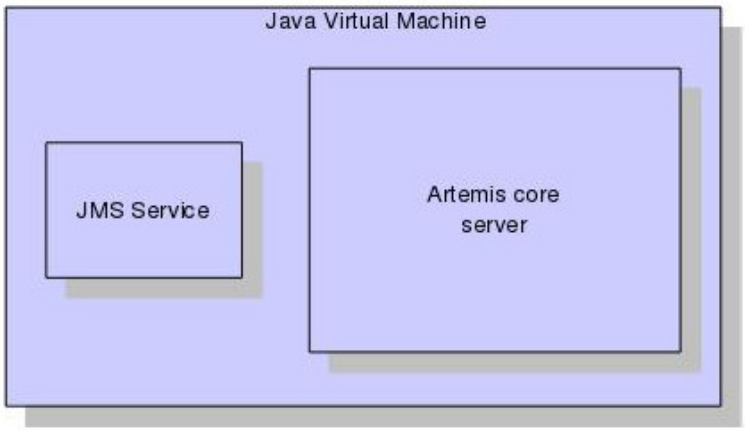
You can see from the diagram that the JMS API is implemented by a thin facade layer on the client side.

Stand-alone Broker

The normal stand-alone messaging broker configuration comprises a core messaging broker and a number of protocol managers that provide support for the various protocol mentioned earlier.

The stand-alone broker configuration uses Airline for bootstrapping the Broker.

The stand-alone broker architecture is shown in figure 3.3 below:



For more information on server configuration files see Server Configuration

Embedded Broker

Apache ActiveMQ Artemis core is designed as a set of simple POJOs so if you have an application that requires messaging functionality internally but you don't want to expose that as an Apache ActiveMQ Artemis broker you can directly instantiate and embed brokers in your own application.

Read more about embedding Apache ActiveMQ Artemis.

Integrated with a Java EE application server

Apache ActiveMQ Artemis provides its own fully functional Java Connector Architecture (JCA) adaptor which enables it to be integrated easily into any Java EE compliant application server or servlet engine.

Java EE application servers provide Message Driven Beans (MDBs), which are a special type of Enterprise Java Beans (EJBs) that can process messages from sources such as JMS systems or mail systems.

Probably the most common use of an MDB is to consume messages from a JMS messaging system.

According to the Java EE specification, a Java EE application server uses a JCA adapter to integrate with a JMS messaging system so it can consume messages for MDBs.

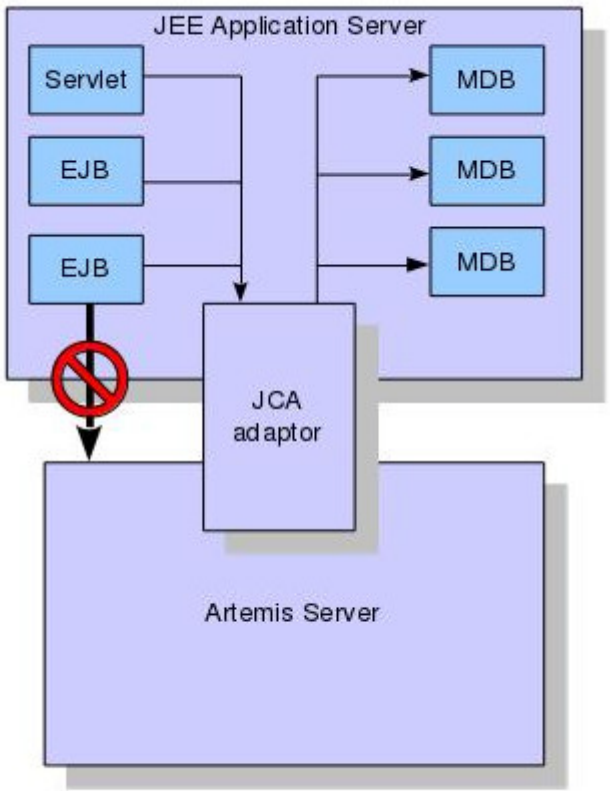
However, the JCA adapter is not only used by the Java EE application server for *consuming* messages via MDBs, it is also used when sending message to the JMS messaging system e.g. from inside an EJB or servlet.

When integrating with a JMS messaging system from inside a Java EE application server it is always recommended that this is done via a JCA adaptor. In fact, communicating with a JMS messaging system directly, without using JCA would be illegal according to the Java EE specification.

The application server's JCA service provides extra functionality such as connection pooling and automatic transaction enlistment, which are desirable when using messaging, say, from inside an EJB. It is possible to talk to a JMS messaging system directly from an EJB, MDB or servlet without going through a JCA adapter, but this is not recommended since you will not be able to take advantage of the JCA features, such as caching of JMS sessions, which can result in poor performance.

Figure 3.2 below shows a Java EE application server integrating with a Apache ActiveMQ Artemis server via the Apache ActiveMQ Artemis JCA adaptor. Note that all communication between EJB sessions or entity beans and Message Driven beans go through the adaptor and not directly to Apache ActiveMQ Artemis.

The large arrow with the prohibited sign shows an EJB session bean talking directly to the Apache ActiveMQ Artemis server. This is not recommended as you'll most likely end up creating a new connection and session every time you want to interact from the EJB, which is an anti-pattern.



Using the Server

This chapter will familiarise you with how to use the Apache ActiveMQ Artemis server.

We'll show where it is, how to start and stop it, and we'll describe the directory layout and what all the files are and what they do.

For the remainder of this chapter when we talk about the Apache ActiveMQ Artemis server we mean the Apache ActiveMQ Artemis standalone server, in its default configuration with a JMS Service enabled.

This document will refer to the full path of the directory where the ActiveMQ distribution has been extracted to as `${ARTEMIS_HOME}` directory.

Installation

After downloading the distribution, the following highlights some important folders on the distribution:

```
|___ bin
|
|___ examples
|      |___ common
|      |___ features
|      |___ perf
|      |___ protocols
|
|___ lib
|      |___ client
|
|___ schema
|
|___ web
       |___ api
       |___ hacking-guide
       |___ migration-guide
       |___ user-manual
```

- `bin` - binaries and scripts needed to run ActiveMQ Artemis.
- `examples` - All manner of examples. Please refer to the examples chapter for details on how to run them.
- `lib` - jars and libraries needed to run ActiveMQ Artemis
- `schema` - XML Schemas used to validate ActiveMQ Artemis configuration files
- `web` - The folder where the web context is loaded when the broker runs.
- `api` - The api documentation is placed under the web folder.
- `user-manual` - The user manual is placed under the web folder.

Creating a Broker Instance

A broker instance is the directory containing all the configuration and runtime data, such as logs and data files, associated with a broker process. It is recommended that you do *not* create the instance directory under `${ARTEMIS_HOME}`. This separation is encouraged so that you can more easily upgrade when the next version of ActiveMQ Artemis is released.

On Unix systems, it is a common convention to store this kind of runtime data under the `/var/lib` directory. For example, to create an instance at '/var/lib/mybroker', run the following commands in your command line shell:

```
cd /var/lib
${ARTEMIS_HOME}/bin/artemis create mybroker
```

A broker instance directory will contain the following sub directories:

- `bin` : holds execution scripts associated with this instance.
- `etc` : hold the instance configuration files
- `data` : holds the data files used for storing persistent messages
- `log` : holds rotating log files
- `tmp` : holds temporary files that are safe to delete between broker runs

At this point you may want to adjust the default configuration located in the `etc` directory.

Options

There are several options you can use when creating an instance.

For a full list of updated properties always use:

```
$./artemis help create
NAME
        artemis create - creates a new broker instance

SYNOPSIS
       artemis create [--addresses <addresses>] [--aio] [--allow-anonymous]
               [--autocreate] [--blocking] [--cluster-password <clusterPasswo
               [--cluster-user <clusterUser>] [--clustered] [--data <data>]
               [--default-port <defaultPort>] [--disable-persistence]
               [--encoding <encoding>] [--etc <etc>] [--failover-on-shutdown]
               [--global-max-size <globalMaxSize>] [--home <home>] [--host <h
               [--http-host <httpHost>] [--http-port <httpPort>]
               [--java-options <javaOptions>] [--mapped] [--max-hops <maxHops
               [--message-load-balancing <messageLoadBalancing>] [--name <nam
               [--nio] [--no-amqp-acceptor] [--no-autocreate] [--no-autotune]
               [--no-fsync] [--no-hornetq-acceptor] [--no-mqtt-acceptor]
               [--no-stomp-acceptor] [--no-web] [--paging] [--password <passw
               [--ping <ping>] [--port-offset <portOffset>] [--queues <queues
               [--replicated] [--require-login] [--role <role>] [--shared-sto
               [--silent] [--slave] [--ssl-key <sslKey>]
               [--ssl-key-password <sslKeyPassword>] [--ssl-trust <sslTrust>]
               [--ssl-trust-password <sslTrustPassword>] [--use-client-auth]
               [--user <user>] [--verbose] [--] <directory>

OPTIONS
        --addresses <addresses>
            Comma separated list of addresses

        --aio
            Sets the journal as asyncio.

        --allow-anonymous
            Enables anonymous configuration on security, opposite of
            --require-login (Default: input)

        --autocreate
            Auto create addresses. (default: true)

        --blocking
            Block producers when address becomes full, opposite of --paging
            (Default: false)

        --cluster-password <clusterPassword>
            The cluster password to use for clustering. (Default: input)

        --cluster-user <clusterUser>
            The cluster user to use for clustering. (Default: input)

        --clustered
            Enable clustering

        --data <data>
            Directory where ActiveMQ data are stored. Paths can be absolute o
            relative to artemis.instance directory ('data' by default)

        --default-port <defaultPort>
            The port number to use for the main 'artemis' acceptor (Default:
            61616)

        --disable-persistence
            Disable message persistence to the journal

        --encoding <encoding>
            The encoding that text files should use
```

```
          --etc <etc>
              Directory where ActiveMQ configuration is located. Paths can be a
              relative to artemis.instance directory ('etc' by default)

          --failover-on-shutdown
              Valid for shared store: will shutdown trigger a failover? (Defaul
              false)

          --force
              Overwrite configuration at destination directory

          --global-max-size <globalMaxSize>
              Maximum amount of memory which message data may consume (Default:
              Undefined, half of the system's memory)

          --home <home>
              Directory where ActiveMQ Artemis is installed

          --host <host>
              The host name of the broker (Default: 0.0.0.0 or input if cluster

          --http-host <httpHost>
              The host name to use for embedded web server (Default: localhost)

          --http-port <httpPort>
              The port number to use for embedded web server (Default: 8161)

          --java-options <javaOptions>
              Extra java options to be passed to the profile

          --mapped
              Sets the journal as mapped.

          --max-hops <maxHops>
              Number of hops on the cluster configuration

          --message-load-balancing <messageLoadBalancing>
              Load balancing policy on cluster. [ON_DEMAND (default) | STRICT |
              OFF]

          --name <name>
              The name of the broker (Default: same as host)

          --nio
              Sets the journal as nio.

          --no-amqp-acceptor
              Disable the AMQP specific acceptor.

          --no-autocreate
              Disable Auto create addresses.

          --no-autotune
              Disable auto tuning on the journal.

          --no-fsync
              Disable usage of fdatasync (channel.force(false) from java nio) o
              the journal

          --no-hornetq-acceptor
              Disable the HornetQ specific acceptor.

          --no-mqtt-acceptor
              Disable the MQTT specific acceptor.

          --no-stomp-acceptor
```

```
            Disable the STOMP specific acceptor.

        --no-web
            Remove the web-server definition from bootstrap.xml

        --paging
            Page messages to disk when address becomes full, opposite of
            --blocking (Default: true)

        --password <password>
            The user's password (Default: input)

        --ping <ping>
            A comma separated string to be passed on to the broker config as
            network-check-list. The broker will shutdown when all these
            addresses are unreachable.

        --port-offset <portOffset>
            Off sets the ports of every acceptor

        --queues <queues>
            Comma separated list of queues.

        --replicated
            Enable broker replication

        --require-login
            This will configure security to require user / password, opposite
            --allow-anonymous

        --role <role>
            The name for the role created (Default: amq)

        --shared-store
            Enable broker shared store

        --silent
            It will disable all the inputs, and it would make a best guess fo
            any required input

        --slave
            Valid for shared store or replication: this is a slave server?

        --ssl-key <sslKey>
            The key store path for embedded web server

        --ssl-key-password <sslKeyPassword>
            The key store password

        --ssl-trust <sslTrust>
            The trust store path in case of client authentication

        --ssl-trust-password <sslTrustPassword>
            The trust store password

        --use-client-auth
            If the embedded server requires client authentication

        --user <user>
            The username (Default: input)

        --verbose
            Adds more information on the execution

        --
            This option can be used to separate command-line options from the
```

```
              list of argument, (useful when arguments might be mistaken for
              command-line options

          <directory>
              The instance directory to hold the broker's configuration and dat
              Path must be writable.
```

Some of these properties may be mandatory in certain configurations and the system may ask you for additional input.

```
./artemis create /usr/server
Creating ActiveMQ Artemis instance at: /user/server

--user: is a mandatory property!
Please provide the default username:
admin

--password: is mandatory with this configuration:
Please provide the default password:


--allow-anonymous | --require-login: is a mandatory property!
Allow anonymous access?, valid values are Y,N,True,False
y

Auto tuning journal ...
done! Your system can make 0.34 writes per millisecond, your journal-buffer-ti

You can now start the broker by executing:

   "/user/server/bin/artemis" run

Or you can run the broker in the background using:

   "/user/server/bin/artemis-service" start
```

Starting and Stopping a Broker Instance

Assuming you created the broker instance under `/var/lib/mybroker` all you need to do start running the broker instance is execute:

```
/var/lib/mybroker/bin/artemis run
```

Now that the broker is running, you can optionally run some of the included examples to verify the the broker is running properly.

To stop the Apache ActiveMQ Artemis instance you will use the same `artemis` script, but with the `stop` argument. Example:

```
/var/lib/mybroker/bin/artemis stop
```

Please note that Apache ActiveMQ Artemis requires a Java 7 or later runtime to run.

By default the `etc/bootstrap.xml` configuration is used. The configuration can be changed e.g. by running `./artemis run -- xml:path/to/bootstrap.xml` or another config of your choosing.

Environment variables are used to provide ease of changing ports, hosts and data directories used and can be found in `etc/artemis.profile` on linux and `etc\artemis.profile.cmd` on Windows.

Server JVM settings

The run scripts set some JVM settings for tuning the garbage collection policy and heap size. We recommend using a parallel garbage collection algorithm to smooth out latency and minimise large GC pauses.

By default Apache ActiveMQ Artemis runs in a maximum of 1GiB of RAM. To increase the memory settings change the `-Xms` and `-Xmx` memory settings as you would for any Java program.

If you wish to add any more JVM arguments or tune the existing ones, the run scripts are the place to do it.

Library Path

If you're using the Asynchronous IO Journal on Linux, you need to specify `java.library.path` as a property on your Java options. This is done automatically in the scripts.

If you don't specify `java.library.path` at your Java options then the JVM will use the environment variable `LD_LIBRARY_PATH`.

You will need to make sure libaio is installed on Linux. For more information refer to the libaio chapter.

System properties

Apache ActiveMQ Artemis can take a system property on the command line for configuring logging.

For more information on configuring logging, please see the section on Logging.

Configuration files

The configuration file used to bootstrap the server (e.g. `bootstrap.xml` by default) references the specific broker configuration files.

- `broker.xml`. This is the main ActiveMQ configuration file. All the parameters in this file are described here

It is also possible to use system property substitution in all the configuration files. by replacing a value with the name of a system property. Here is an example of this with a connector configuration:

```
<connector name="netty">tcp://${activemq.remoting.netty.host:localhost}:${acti
```

Here you can see we have replaced 2 values with system properties `activemq.remoting.netty.host` and `activemq.remoting.netty.port`. These values will be replaced by the value found in the system property if there is one, if not they default back to localhost or 61616 respectively. It is also possible to not supply a default. i.e. `${activemq.remoting.netty.host}`, however the system property *must* be supplied in that case.

Bootstrap configuration file

The stand-alone server is basically a set of POJOs which are instantiated by Airline commands.

The bootstrap file is very simple. Let's take a look at an example:

```
<broker xmlns="http://activemq.org/schema">

   <jaas-security domain="activemq"/>

   <server configuration="file:/path/to/broker.xml"/>

   <web bind="http://localhost:8161" path="web">
       <app url="activemq-branding" war="activemq-branding.war"/>
       <app url="artemis-plugin" war="artemis-plugin.war"/>
       <app url="console" war="console.war"/>
   </web>
</broker>
```

- `server` - Instantiates a core server using the configuration file from the `configuration` attribute. This is the main broker POJO necessary to do all the real messaging work.
- `jaas-security` - Configures JAAS-based security for the server. The `domain` attribute refers to the relevant login module entry in `login.config`. If different behavior is needed then a custom security manager can be configured by replacing `jaas-security` with `security-manager`. See the "Custom Security Manager" section in the security chapter for more details.
- `web` - Configures an embedded Jetty instance to serve web applications like the admin console.

Broker configuration file

The configuration for the Apache ActiveMQ Artemis core server is contained in `broker.xml`. This is what the FileConfiguration bean uses to configure the messaging server.

There are many attributes which you can configure Apache ActiveMQ Artemis. In most cases the defaults will do fine, in fact every attribute can be defaulted which means a file with a single empty `configuration` element is a valid configuration file. The different configuration will be explained throughout the manual or you can refer to the configuration reference here.

Windows Server

On windows you will have the option to run ActiveMQ Artemis as a service. Just use the following command to install it:

```
$ ./artemis-service.exe install
```

The create process should give you a hint of the available commands available for the artemis-service.exe

Adding Bootstrap Dependencies

Bootstrap dependencies like logging handlers must be accessible by the log manager at boot time. Package the dependency in a jar and put it on the boot classpath before of log manager jar. This can be done appending the jar at the variable `JAVA_ARGS`, defined in `artemis.profile`, with the option `-Xbootclasspath/a`.

Adding Runtime Dependencies

Runtime dependencies like diverts, transformers, broker plugins, JDBC drivers, password decoders, etc. must be accessible by the broker at runtime. Package the dependency in a jar, and put it on the broker's classpath. This can be done by placing the jar file in the `lib` directory of the broker distribution itself or in the `lib` directory of the broker instance. A broker instance does not have a `lib` directory by default so it may need to be created. It should be on the "top" level with the `bin`, `data`, `log`, etc. directories.

Upgrading the Broker

Apache ActiveMQ 5.x (and previous versions) is runnable out of the box by executing the command: `./bin/activemq run`. The ActiveMQ Artemis broker follows a different paradigm where the project distribution serves as the broker "home" and one or more broker "instances" are created which reference the "home" for resources (e.g. jar files) which can be safely shared between broker instances. Therefore, an instance of the broker must be created before it can be run. This may seems like an overhead at first glance, but it becomes very practical when updating to a new Artemis version for example.

To create an Artemis broker instance navigate into the Artemis home folder and run: `./bin/artemis create /path/to/myBrokerInstance` on the command line.

Because of this separation it's very easy to upgrade Artemis in most cases.

> **Note:**
>
> It's recommended to choose a folder different than the on where Apache Artemis was downloaded. This separation allows you run multiple broker instances with the same Artemis "home" for example. It also simplifies updating to newer versions of Artemis.

General Upgrade Procedure

Upgrading may require some specific steps noted in the versions, but the general process is as follows:

1. Navigate to the `etc` folder of the broker instance that's being upgraded
2. Open `artemis.profile` (`artemis.profile.cmd` on Windows). It contains a property which is relevant for the upgrade:

```
ARTEMIS_HOME='/path/to/apache-artemis-version'
```

If you run Artemis as a service on windows you have to do the following additional steps:

1. Navigate to the `bin` folder of the broker instance that's being upgraded
2. Open `artemis-service.xml`. It contains a property which is relevant for the upgrade:

```
<env name="ARTEMIS_HOME" value="/path/to/apache-artemis-version"/>
```

The `ARTEMIS_HOME` property is used to link the instance with the home. *In most cases* the instance can be upgraded to a newer version simply by changing the value of this property to the location of the new broker home. Please refer to the aforementioned versions document for additional upgrade steps (if required).

Addressing Model

Apache ActiveMQ Artemis has a unique addressing model that is both powerful and flexible and that offers great performance. The addressing model comprises three main concepts: **addresses**, **queues**, and **routing types**.

Address

An address represents a messaging endpoint. Within the configuration, a typical address is given a unique name, 0 or more queues, and a routing type.

Queue

A queue is associated with an address. There can be multiple queues per address. Once an incoming message is matched to an address, the message will be sent on to one or more of its queues, depending on the routing type configured. Queues can be configured to be automatically created and deleted.

Routing Types

A routing type determines how messages are sent to the queues associated with an address. An Apache ActiveMQ Artemis address can be configured with two different routing types.

Table 1. Routing Types

If you want your messages routed to...	Use this routing type...
A single queue within the matching address, in a point-to-point manner.	Anycast
Every queue within the matching address, in a publish-subscribe manner.	Multicast

Note: It is possible to define more than one routing type per address, but this typically results in an anti-pattern and is therefore not recommended. If an address does use both routing types, however, and the client does not show a preference for either one, the broker typically defaults to the anycast routing type.

The one exception is when the client uses the MQTT protocol. In that case, the default routing type is multicast.

For additional details about these concepts refer to the core chapter.

Basic Address Configuration

The following examples show how to configure basic point to point and publish subscribe addresses.

Point-to-Point Messaging

Point-to-point messaging is a common scenario in which a message sent by a producer has only one consumer. AMQP and JMS message producers and consumers can make use of point-to-point messaging queues, for example. Define an anycast routing type for an address so that its queues receive messages in a point-to-point manner.

When a message is received on an address using anycast, Apache ActiveMQ Artemis locates the queue associated with the address and routes the message to it. When consumers request to consume from the address, the broker locates the relevant queue and associates this queue with the appropriate consumers. If multiple consumers are connected to the same queue, messages are distributed amongst each consumer equally, providing the consumers are equally able to handle them.

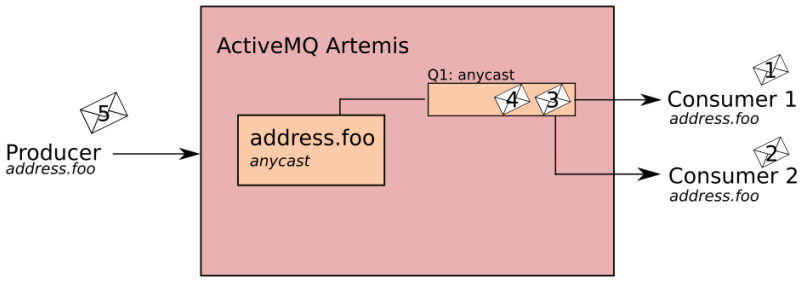


Figure 1. Point to Point Messaging

Using the Anycast Routing Type

Open the file `<broker-instance>/etc/broker.xml` for editing.

Add an address configuration element and its associated queue if they do not exist already.

Note: For normal Point to Point semantics, the queue name **MUST** match the address name.

```
<addresses>
   <address name="orders">
      <anycast>
         <queue name="orders"/>
      </anycast>
   </address>
</addresses>
```

Publish-Subscribe Messaging

In a publish-subscribe scenario, messages are sent to every consumer subscribed to an address. JMS topics and MQTT subscriptions are two examples of publish-subscribe messaging.

To configure an address with publish-subscribe semantics, create an address with the multicast routing type.

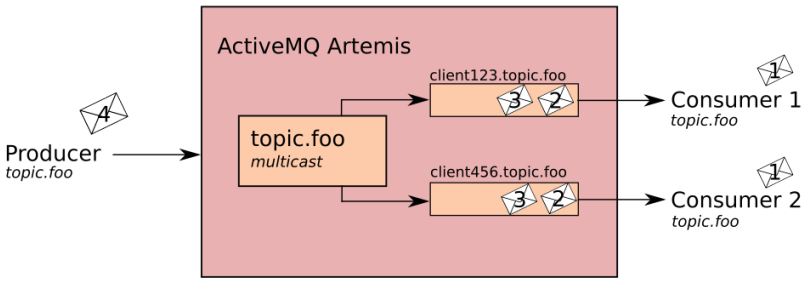


Figure 2. Publish-Subscribe

Using the Multicast Routing Type

Open the file `<broker-instance>/etc/broker.xml` for editing.

Add an address configuration element with multicast routing type.

```
<addresses>
   <address name="pubsub.foo">
      <multicast/>
   </address>
</addresses>
```

When clients connect to an address with the multicast element, a subscription queue for the client will be automatically created for the client. It is also possible to pre-configure subscription queues and connect to them directly using the queue's Fully Qualified Queue names.

Optionally add one or more queue elements to the address and wrap the multicast element around them. This step is typically not needed since the broker will automatically create a queue for each subscription requested by a client.

```
<addresses>
   <address name="pubsub.foo">
      <multicast>
         <queue name="client123.pubsub.foo"/>
         <queue name="client456.pubsub.foo"/>
      </multicast>
   </address>
</addresses>
```

Figure 3. Point-to-Point with Two Queues

Point-to-Point Address multiple Queues

It is actually possible to define more than one queue on an address with an anycast routing type. When messages are received on such an address, they are firstly distributed evenly across all the defined queues. Using Fully Qualified Queue names, clients are able to select the queue that they would like to subscribe to. Should more than one consumer connect directly to a single queue, Apache ActiveMQ Artemis will take care of distributing messages between them, as in the example above.

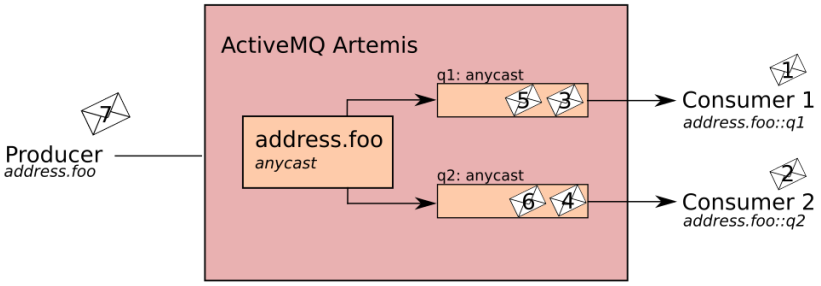


Figure 3. Point-to-Point with Two Queues

Note: This is how Apache ActiveMQ Artemis handles load balancing of queues across multiple nodes in a cluster. Configuring a Point-to-Point Address with two queues, open the file `<broker-instance>/etc/broker.xml` for editing.

Add an address configuration with Anycast routing type element and its associated queues.

```
<addresses>
   <address name="address.foo">
      <anycast>
         <queue name="q1"/>
         <queue name="q2"/>
      </anycast>
   </address>
</addresses>
```

Point-to-Point and Publish-Subscribe Addresses

It is possible to define an address with both point-to-point and publish-subscribe semantics enabled. While not typically recommend, this can be useful when you want, for example, a JMS Queue say orders and a JMS Topic named orders. The different routing types make the addresses appear to be distinct.

Using an example of JMS Clients, the messages sent by a JMS message producer will be routed using the anycast routing type. Messages sent by a JMS topic producer will use the multicast routing type. In addition when a JMS topic consumer attaches, it will be attached to it's own subscription queue. JMS queue consumer will be attached to the anycast queue.

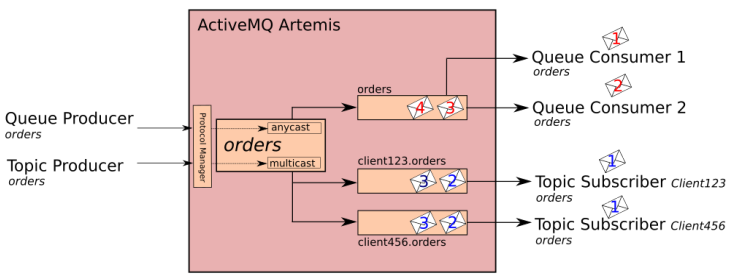


Figure 4. Point-to-Point and Publish-Subscribe

Note: The behavior in this scenario is dependent on the protocol being used. For JMS there is a clear distinction between topic and queue producers and consumers, which make the logic straight forward. Other protocols like AMQP do not make this distinction. A message being sent via AMQP will be routed by both

anycast and multicast and consumers will default to anycast. For more information, please check the behavior of each protocol in the sections on protocols.

The XML snippet below is an example of what the configuration for an address using both anycast and multicast would look like in `<broker-instance>/etc/broker.xml`. Note that subscription queues are typically created on demand, so there is no need to list specific queue elements inside the multicast routing type.

```
<addresses>
   <address name="foo.orders">
      <anycast>
         <queue name="orders"/>
      </anycast>
      <multicast/>
   </address>
</addresses>
```

How to filter messages

Apache ActiveMQ Artemis supports the ability to filter messages using Apache Artemis Filter Expressions.

Filters can be applied in two places, on a queue and on a consumer.

Queue Filter

When a filter is applied to a queue, messages are filtered before they are sent to the queue. To add a queue filter use the filter element when configuring a queue. Open up `<broker-instance>/etc/broker.xml` and add an address with a queue, using the filter element to configure a filter on this queue.

```
<addresses>
   <address name="filter">
      <queue name="filter">
         <filter string="color='red'"/>
      </queue>
   </address>
</addresses>
```

The filter defined above ensures that only messages with an attribute `"color='red'"` is sent to this queue.

Consumer Filters

Consumer filters are applied after messages have reached a queue and are defined using the appropriate client APIs. The following JMS example shows how consumer filters work.

1. Define an address with a single queue, with no filter applied.

```
<addresses>
   <address name="filter">
      <queue name="filter"/>
   </address>
</addresses>
```

```
...
// Send some messages
for (int i = 0; i < 3; i ++) {
  TextMessage redMessage = senderSession.createTextMessage("Red");
  redMessage.setStringProperty("color", "red");
  producer.send(redMessage)

  TextMessage greenMessage = senderSession.createTextMessage("Green");
  greenMessage.setStringProperty("color", "green");
  producer.send(greenMessage)
}
```

At this point the queue would have 6 messages: red,green,red,green,red,green

```
MessageConsumer redConsumer = redSession.createConsumer(queue, "color='red'");
```

The redConsumer has a filter that only matches "red" messages. The redConsumer will receive 3 messages.

```
red, red, red
```

The resulting queue would now be

```
green, green, green
```

Automatic Address/Queue Management

You can configure Apache ActiveMQ Artemis to automatically create addresses and queues, and then delete them when they are no longer in use. This saves you from having to preconfigure each address and queue before a client can connect to it. Automatic creation and deletion is configured on a per address basis and is controlled by following:

Parameter	Description
`auto-create-addresses`	When set to true, the broker will create the address requested by the client if it does not exist already. The default is `true`.
`auto-delete-addresses`	When set to true, the broker will be delete any **auto-created** adddress once all of it's queues have been deleted. The default is `true`
`default-address-routing-type`	The routing type to use if the client does not specify one. Possible values are `MULTICAST` and `ANYCAST`. See earlier in this chapter for more information about routing types. The default value is `MULTICAST`.

Auto Address Creation

- Edit the file `<broker-instance>/etc/broker.xml` and add the `auto-create-addresses` element to the `address-setting` you want the broker to automatically create.
- (Optional) Add the `address-setting` if it does not exist. Use the match parameter and the wildcard syntax to match more than one specific address.
- Set `auto-create-addresses` to `true`
- (Optional) Assign `MULTICAST` or `ANYCAST` as the default routing type for the address.

The example below configures an `address-setting` to be automatically created by the broker. The default routing type to be used if not specified by the client is MULTICAST. Note that wildcard syntax is used. Any address starting with `/news/politics/` will be automatically created by the broker.

```
<address-setting match="/news/politics/#">
  <auto-create-addresses>true</auto-create-addresses>
  <default-address-routing-type>MULTICAST</default-address-routing-type>
</address-setting>
```

Auto Address Deletion

- Edit the file `<broker-instance>/etc/broker.xml` and add the `auto-delete-addresses` element to the `address-setting` you want the broker to automatically create.
- (Optional) Add the `address-setting` if it does not exist. Use the match parameter and the wildcard syntax to match more than one specific address.
- Set `auto-delete-addresses` to `true`

The example below configures an `address-setting` to be automatically deleted by the broker. Note that wildcard syntax is used. Any address request by the client that starts with `/news/politics/` is configured to be automatically deleted by the broker.

```
<address-setting match="/news/politics/#">
  <auto-delete-addresses>true</auto-delete-addresses>
  <default-address-routing-type>MULTICAST</default-address-routing-type>
</address-setting>
```

"Fully Qualified" Queue Names

Internally the broker maps a client's request for an address to specific queues. The broker decides on behalf of the client which queues to send messages to or from which queue to receive messages. However, more advanced use cases

might require that the client specify a queue directly. In these situations the client uses a fully qualified queue name, by specifying both the address name and the queue name, separated by a ::.

Currently Artemis supports fully qualified queue names on Core, AMQP, JMS, OpenWire, MQTT and STOMP protocols for receiving messages only.

Specifying a Fully Qualified Queue Name

In this example, the address foo is configured with two queues q1, q2 as shown in the configuration below.

```
<addresses>
   <address name="foo">
      <anycast>
         <queue name="q1" />
         <queue name="q2" />
      </anycast>
   </address>
</addresses>
```

In the client code, use both the address name and the queue name when requesting a connection from the broker. Remember to use two colons, `::`, to separate the names, as in the example Java code below.

```
String FQQN = "foo::q1";
Queue q1 session.createQueue(FQQN);
MessageConsumer consumer = session.createConsumer(q1);
```

Using Prefixes to Determine Routing Type

Normally, if the broker receives a message sent to a particular address, that has both `ANYCAST` and `MULTICAST` routing types enable, it will route a copy of the message to **one** of the `ANYCAST` queues and to **all** of the `MULTICAST` queues.

However, clients can specify a special prefix when connecting to an address to indicate which kind of routing type to use. The prefixes are custom values that are designated using the anycastPrefix and multicastPrefix parameters within the URL of an acceptor.

Configuring an Anycast Prefix

In `<broker-instance>/etc/broker.xml`, add the `anycastPrefix` to the URL of the desired acceptor. In the example below, the acceptor is configured to use `anycast://` for the `anycastPrefix`. Client code can specify `anycast://foo/` if the client needs to send a message to only one of the `ANYCAST` queues.

```
<acceptor name="artemis">tcp://0.0.0.0:61616?protocols=AMQP;anycastPrefix=anyc
```

Configuring a Multicast Prefix

In `<broker-instance>/etc/broker.xml`, add the `multicastPrefix` to the URL of the desired acceptor. In the example below, the acceptor is configured to use `multicast://` for the `multicastPrefix`. Client code can specify `multicast://foo/` if the client needs to send a message to only one of the `MULTICAST` queues.

```
<acceptor name="artemis">tcp://0.0.0.0:61616?protocols=AMQP;multicastPrefix=mu
```

Advanced Address Configuration

Static Subscription Queues

In most cases it's not necessary to statically configure subscription queues. The relevant protocol managers take care of dynamically creating subscription queues when clients request to subscribe to an address. The type of subscription queue created depends on what properties the client request. For example, durable, non-shared, shared etc. Protocol managers use special queue naming conventions to identify which queues belong to which consumers and users need not worry about the details.

However, there are scenarios where a user may want to use broker side configuration to statically configure a subscription and later connect to that queue directly using a Fully Qualified Queue name. The examples below show how to use broker side configuration to statically configure a queue with publish subscribe behavior for shared, non-shared, durable and non-durable subscription behavior.

Shared, Durable Subscription Queue using max-consumers

The default behavior for queues is to not limit the number connected queue consumers. The **max-consumers** parameter of the queue element can be used to limit the number of connected consumers allowed at any one time.

Open the file `<broker-instance>/etc/broker.xml` for editing.

```
<addresses>
   <address name="durable.foo">
      <multicast>
         <!-- pre-configured shared durable subscription queue -->
         <queue name="q1" max-consumers="10">
            <durable>true</durable>
         </queue>
      </multicast>
   </address>
</addresses>
```

Non-shared, Durable Subscription Queue

The broker can be configured to prevent more than one consumer from connecting to a queue at any one time. The subscriptions to queues configured this way are therefore "non-shared". To do this simply set the **max-consumers** parameter to `1`:

```
<addresses>
   <address name="durable.foo">
      <multicast>
         <!-- pre-configured non shared durable subscription queue -->
         <queue name="q1" max-consumers="1">
            <durable>true</durable>
         </queue>
      </multicast>
   </address>
</addresses>
```

Non-durable Subscription Queue

Non-durable subscriptions are again usually managed by the relevant protocol manager, by creating and deleting temporary queues.

If a user requires to pre-create a queue that behaves like a non-durable subscription queue the **purge-on-no-consumers** flag can be enabled on the queue. When **purge-on-no-consumers** is set to **true**. The queue will not start receiving messages until a consumer is attached. When the last consumer is detached from the queue. The queue is purged (its messages are removed) and will not receive any more messages until a new consumer is attached.

Open the file `<broker-instance>/etc/broker.xml` for editing.

```
<addresses>
   <address name="non.shared.durable.foo">
      <multicast>
         <queue name="orders1" purge-on-no-consumers="true"/>
      </multicast>
   </address>
</addresses>
```

Exclusive Consumer Queue

If a user requires to statically configure a queue that routes exclusively to one active consumer the **exclusive** flag can be enabled on the queue.

When **exclusive** is set to **true** the queue will route messages to the a single active consumer. When the active consumer that is being routed to is detached from the queue, if another active consumer exist, one will be chosen and routing will now be exclusive to it.

See Exclusive Queue for further information.

Open the file `<broker-instance>/etc/broker.xml` for editing.

```
<addresses>
   <address name="foo.bar">
      <multicast>
         <queue name="orders1" exclusive="true"/>
      </multicast>
   </address>
</addresses>
```

Disabled Queue

If a user requires to statically configure a queue and disable routing to it, for example where a queue needs to be defined so a consumer can bind, but you want to disable message routing to it for the time being.

Or you need to stop message flow to the queue to allow investigation keeping the consumer bound, but dont wish to have further messages routed to the queue to avoid message build up.

When **enabled** is set to **true** the queue will have messages routed to it. (default)

When **enabled** is set to **false** the queue will NOT have messages routed to it.

Open the file `<broker-instance>/etc/broker.xml` for editing.

```
<addresses>
   <address name="foo.bar">
      <multicast>
         <queue name="orders1" enabled="false"/>
      </multicast>
   </address>
</addresses>
```

Warning: Disabling all the queues on an address means that any message sent to that address will be silently dropped.

Protocol Managers

A "protocol manager" maps protocol-specific concepts down to the core addressing model (using addresses, queues and routing types). For example, when a client sends a MQTT subscription packet with the addresses:

```
/house/room1/lights
/house/room2/lights
```

The MQTT protocol manager understands that the two addresses require `MULTICAST` semantics. The protocol manager will therefore first look to ensure that `MULTICAST` is enabled for both addresses. If not, it will attempt to dynamically create them. If successful, the protocol manager will then create special subscription queues with special names, for each subscription requested by the client.

The special name allows the protocol manager to quickly identify the required client subscription queues should the client disconnect and reconnect at a later date. If the subscription is temporary the protocol manager will delete the queue once the client disconnects.

When a client requests to subscribe to a point to point address. The protocol manager will look up the queue associated with the point to point address. This queue should have the same name as the addresss.

Note: If the queue is auto created, it will be auto deleted once there are no consumers and no messages in it. For more information on auto create see the next section Configuring Addresses and Queues via Address Settings

Configuring Addresses and Queues via Address Settings

There are some attributes that are defined against an address wildcard rather than a specific address/queue. Here an example of an `address-setting` entry that would be found in the `broker.xml` file.

```
<address-settings>
   <address-setting match="order.foo">
      <dead-letter-address>DLA</dead-letter-address>
      <auto-create-dead-letter-resources>false</auto-create-dead-letter-resour
      <dead-letter-queue-prefix>DLQ.</dead-letter-queue-prefix>
      <dead-letter-queue-suffix></dead-letter-queue-suffix>
      <expiry-address>ExpiryQueue</expiry-address>
      <auto-create-expiry-resources>false</auto-create-expiry-resources>
      <expiry-queue-prefix>EXP.</expiry-queue-prefix>
      <expiry-queue-suffix></expiry-queue-suffix>
      <expiry-delay>123</expiry-delay>
      <redelivery-delay>5000</redelivery-delay>
      <redelivery-delay-multiplier>1.0</redelivery-delay-multiplier>
      <redelivery-collision-avoidance-factor>0.0</redelivery-collision-avoidan
      <max-redelivery-delay>10000</max-redelivery-delay>
      <max-delivery-attempts>3</max-delivery-attempts>
      <max-size-bytes>100000</max-size-bytes>
      <max-size-bytes-reject-threshold>-1</max-size-bytes-reject-threshold>
      <page-size-bytes>20000</page-size-bytes>
      <page-max-cache-size></page-max-cache-size>
      <address-full-policy>PAGE</address-full-policy>
      <message-counter-history-day-limit></message-counter-history-day-limit>
      <last-value-queue>true</last-value-queue> <!-- deprecated! see default-l
      <default-last-value-queue>false</default-last-value-queue>
      <default-non-destructive>false</default-non-destructive>
      <default-exclusive-queue>false</default-exclusive-queue>
      <default-consumers-before-dispatch>0</default-consumers-before-dispatch>
      <default-delay-before-dispatch>-1</default-delay-before-dispatch>
      <redistribution-delay>0</redistribution-delay>
      <send-to-dla-on-no-route>true</send-to-dla-on-no-route>
      <slow-consumer-threshold>-1</slow-consumer-threshold>
      <slow-consumer-policy>NOTIFY</slow-consumer-policy>
      <slow-consumer-check-period>5</slow-consumer-check-period>
      <auto-create-jms-queues>true</auto-create-jms-queues> <!-- deprecated! s
      <auto-delete-jms-queues>true</auto-delete-jms-queues> <!-- deprecated! s
      <auto-create-jms-topics>true</auto-create-jms-topics> <!-- deprecated! s
      <auto-delete-jms-topics>true</auto-delete-jms-topics> <!-- deprecated! s
      <auto-create-queues>true</auto-create-queues>
      <auto-delete-queues>true</auto-delete-queues>
      <auto-delete-created-queues>false</auto-delete-created-queues>
      <auto-delete-queues-delay>0</auto-delete-queues-delay>
      <auto-delete-queues-message-count>0</auto-delete-queues-message-count>
      <config-delete-queues>OFF</config-delete-queues>
      <auto-create-addresses>true</auto-create-addresses>
      <auto-delete-addresses>true</auto-delete-addresses>
      <auto-delete-addresses-delay>0</auto-delete-addresses-delay>
      <config-delete-addresses>OFF</config-delete-addresses>
      <management-browse-page-size>200</management-browse-page-size>
      <default-purge-on-no-consumers>false</default-purge-on-no-consumers>
      <default-max-consumers>-1</default-max-consumers>
      <default-queue-routing-type></default-queue-routing-type>
      <default-address-routing-type></default-address-routing-type>
      <default-ring-size>-1</default-ring-size>
      <retroactive-message-count>0</retroactive-message-count>
      <enable-metrics>true</enable-metrics>
   </address-setting>
</address-settings>
```

The idea with address settings, is you can provide a block of settings which will be applied against any addresses that match the string in the `match` attribute. In the above example the settings would only be applied to the address "order.foo" address but you can also use wildcards to apply settings.

For example, if you used the `match` string `queue.#` the settings would be applied to all addresses which start with `queue.`

The meaning of the specific settings are explained fully throughout the user manual, however here is a brief description with a link to the appropriate chapter if available.

`dead-letter-address` is the address to which messages are sent when they exceed `max-delivery-attempts`. If no address is defined here then such messages will simply be discarded. Read more about undelivered messages.

`auto-create-dead-letter-resources` determines whether or not the broker will automatically create the defined `dead-letter-address` and a corresponding dead-letter queue when a message is undeliverable. Read more in the chapter about undelivered messages.

`dead-letter-queue-prefix` defines the prefix used for automatically created dead-letter queues. Read more in the chapter about undelivered messages.

`dead-letter-queue-suffix` defines the suffix used for automatically created dead-letter queues. Read more in the chapter about undelivered messages.

`expiry-address` defines where to send a message that has expired. If no address is defined here then such messages will simply be discarded. Read more about message expiry.

`auto-create-expiry-resources` determines whether or not the broker will automatically create the defined `expiry-address` and a corresponding expiry queue when a message expired. Read more in the chapter about undelivered messages.

`expiry-queue-prefix` defines the prefix used for automatically created expiry queues. Read more in the chapter about message expiry.

`expiry-queue-suffix` defines the suffix used for automatically created expiry queues. Read more in the chapter about message expiry.

`expiry-delay` defines the expiration time that will be used for messages which are using the default expiration time (i.e. 0). For example, if `expiry-delay` is set to "10" and a message which is using the default expiration time (i.e. 0) arrives then its expiration time of "0" will be changed to "10." However, if a message which is using an expiration time of "20" arrives then its expiration time will remain unchanged. Setting `expiry-delay` to "-1" will disable this feature. The default is "-1". Read more about message expiry.

`max-delivery-attempts` defines how many time a cancelled message can be redelivered before sending to the `dead-letter-address`. Read more about undelivered messages.

`redelivery-delay` defines how long to wait before attempting redelivery of a cancelled message. Default is `0`. Read more about undelivered messages.

`redelivery-delay-multiplier` defines the number by which the `redelivery-delay` will be multiplied on each subsequent redelivery attempt. Default is `1.0`. Read more about undelivered messages.

`redelivery-collision-avoidance-factor` defines an additional factor used to calculate an adjustment to the `redelivery-delay` (up or down). Default is `0.0`. Valid values are between 0.0 and 1.0. Read more about undelivered messages.

`max-size-bytes`, `page-size-bytes`, & `page-max-cache-size` are used to configure paging on an address. This is explained here.

`max-size-bytes-reject-threshold` is used with the address full `BLOCK` policy, the maximum size (in bytes) an address can reach before messages start getting rejected. Works in combination with `max-size-bytes` **for AMQP clients only**. Default is `-1` (i.e. no limit).

`address-full-policy`. This attribute can have one of the following values: `PAGE`, `DROP`, `FAIL` or `BLOCK` and determines what happens when an address where `max-size-bytes` is specified becomes full. The default value is `PAGE`. If the value is `PAGE` then further messages will be paged to disk. If the value is `DROP` then further messages will be silently dropped. If the value is `FAIL` then further messages will be dropped and an exception will be thrown on the client-side. If the value is `BLOCK` then client message producers will block when they try and send further messages. See the Flow Control and Paging chapters for more info.

`message-counter-history-day-limit` is the number of days to keep message counter history for this address assuming that `message-counter-enabled` is `true`. Default is `0`.

`last-value-queue` is **deprecated**. See `default-last-value-queue`. It defines whether a queue only uses last values or not. Default is `false`. Read more about last value queues.

`default-last-value-queue` defines whether a queue only uses last values or not. Default is `false`. This value can be overridden at the queue level using the `last-value` boolean. Read more about last value queues.

`default-exclusive-queue` defines whether a queue will serve only a single consumer. Default is `false`. This value can be overridden at the queue level using the `exclusive` boolean. Read more about exclusive queues.

`default-consumers-before-dispatch` defines the number of consumers needed on a queue bound to the matching address before messages will be dispatched to those consumers. Default is `0`. This value can be overridden at the queue level using the `consumers-before-dispatch` boolean. This behavior can be tuned using `delay-before-dispatch` on the queue itself or by using the `default-delay-before-dispatch` address-setting.

`default-delay-before-dispatch` defines the number of milliseconds the broker will wait for the configured number of consumers to connect to the matching queue before it will begin to dispatch messages. Default is `-1` (wait forever).

`redistribution-delay` defines how long to wait when the last consumer is closed on a queue before redistributing any messages. Read more about clusters.

`send-to-dla-on-no-route`. If a message is sent to an address, but the server does not route it to any queues (e.g. there might be no queues bound to that address, or none of the queues have filters that match) then normally that message would

be discarded. However, if this parameter is `true` then such a message will instead be sent to the `dead-letter-address` (DLA) for that address, if it exists.

`slow-consumer-threshold`. The minimum rate of message consumption allowed before a consumer is considered "slow." Measured in messages-per-second. Default is `-1` (i.e. disabled); any other valid value must be greater than 0. Read more about slow consumers.

`slow-consumer-policy`. What should happen when a slow consumer is detected. `KILL` will kill the consumer's connection (which will obviously impact any other client threads using that same connection). `NOTIFY` will send a CONSUMER_SLOW management notification which an application could receive and take action with. Read more about slow consumers.

`slow-consumer-check-period`. How often to check for slow consumers on a particular queue. Measured in *seconds*. Default is `5`. Read more about slow consumers.

`auto-create-jms-queues` is **deprecated**. See `auto-create-queues`. Whether or not the broker should automatically create a JMS queue when a JMS message is sent to a queue whose name fits the address `match` (remember, a JMS queue is just a core queue which has the same address and queue name) or a JMS consumer tries to connect to a queue whose name fits the address `match`. Queues which are auto-created are durable, non-temporary, and non-transient. Default is `true`.

`auto-delete-jms-queues` is **deprecated**. See `auto-delete-queues`. Whether or not the broker should automatically delete auto-created JMS queues when they have both 0 consumers and 0 messages. Default is `true`.

`auto-create-jms-topics` is **deprecated**. See `auto-create-addresses`. Whether or not the broker should automatically create a JMS topic when a JMS message is sent to a topic whose name fits the address `match` (remember, a JMS topic is just a core address which has one or more core queues mapped to it) or a JMS consumer tries to subscribe to a topic whose name fits the address `match`. Default is `true`.

`auto-delete-jms-topics` is **deprecated**. See `auto-delete-addresses`. Whether or not the broker should automatically delete auto-created JMS topics once the last subscription on the topic has been closed. Default is `true`.

`auto-create-queues`. Whether or not the broker should automatically create a queue when a message is sent or a consumer tries to connect to a queue whose name fits the address `match`. Queues which are auto-created are durable, non-temporary, and non-transient. Default is `true`. **Note:** automatic queue creation does *not* work for the core client. The core API is a low-level API and is not meant to have such automation.

`auto-delete-queues`. Whether or not the broker should automatically delete auto-created queues when they have both 0 consumers and the message count is less than or equal to `auto-delete-queues-message-count`. Default is `true`.

`auto-delete-created-queues`. Whether or not the broker should automatically delete created queues when they have both 0 consumers and the message count is less than or equal to `auto-delete-queues-message-count`. Default is `false`.

`auto-delete-queues-delay`. How long to wait (in milliseconds) before deleting auto-created queues after the queue has 0 consumers and the message count is less than or equal to `auto-delete-queues-message-count`. Default is `0` (delete immediately). The broker's `address-queue-scan-period` controls how often (in milliseconds) queues are scanned for potential deletion. Use `-1` to disable scanning. The default scan value is `30000`.

`auto-delete-queues-message-count`. The message count that the queue must be less than or equal to before deleting auto-created queues. To disable message count check `-1` can be set. Default is `0` (empty queue).

Note: the above auto-delete address settings can also be configured individually at the queue level when a client auto creates the queue.

For Core API it is exposed in createQueue methods.

For Core JMS you can set it using the destination queue attributes `my.destination?auto-delete=true&auto-delete-delay=120000&auto-delete-message-count=-1`

`config-delete-queues`. How the broker should handle queues deleted on config reload, by delete policy: `OFF` or `FORCE`. Default is `OFF`. Read more about configuration reload.

`auto-create-addresses`. Whether or not the broker should automatically create an address when a message is sent to or a consumer tries to consume from a queue which is mapped to an address whose name fits the address `match`. Default is `true`. **Note:** automatic address creation does *not* work for the core client. The core API is a low-level API and is not meant to have such automation.

`auto-delete-addresses`. Whether or not the broker should automatically delete auto-created addresses once the address no longer has any queues. Default is `true`.

`auto-delete-addresses-delay`. How long to wait (in milliseconds) before deleting auto-created addresses after they no longer have any queues. Default is `0` (delete immediately). The broker's `address-queue-scan-period` controls how often (in milliseconds) addresses are scanned for potential deletion. Use `-1` to disable scanning. The default scan value is `30000`.

`config-delete-addresses`. How the broker should handle addresses deleted on config reload, by delete policy: `OFF` or `FORCE`. Default is `OFF`. Read more about configuration reload.

`management-browse-page-size` is the number of messages a management resource can browse. This is relevant for the "browse" management method exposed on the queue control. Default is `200`.

`default-purge-on-no-consumers` defines a queue's default `purge-on-no-consumers` setting if none is provided on the queue itself. Default is `false`. This value can be overridden at the queue level using the `purge-on-no-consumers` boolean. Read

more about this functionality.

`default-max-consumers` defines a queue's default `max-consumers` setting if none is provided on the queue itself. Default is `-1` (i.e. no limit). This value can be overridden at the queue level using the `max-consumers` boolean. Read more about this functionality.

`default-queue-routing-type` defines the routing-type for an auto-created queue if the broker is unable to determine the routing-type based on the client and/or protocol semantics. Default is `MULTICAST`. Read more about routing types.

`default-address-routing-type` defines the routing-type for an auto-created address if the broker is unable to determine the routing-type based on the client and/or protocol semantics. Default is `MULTICAST`. Read more about routing types.

`default-consumer-window-size` defines the default `consumerWindowSize` value for a `CORE` protocol consumer, if not defined the default will be set to 1 MiB (1024 * 1024 bytes). The consumer will use this value as the window size if the value is not set on the client. Read more about flow control.

`default-ring-size` defines the default `ring-size` value for any matching queue which doesn't have `ring-size` explicitly defined. If not defined the default will be set to -1. Read more about ring queues.

`retroactive-message-count` defines the number of messages to preserve for future queues created on the matching address. Defaults to 0. Read more about retroactive addresses.

`enable-metrics` determines whether or not metrics will be published to any configured metrics plugin for the matching address. Default is `true`. Read more about metrics.

Protocols and Interoperability

Apache ActiveMQ Artemis has a powerful & flexible core which provides a foundation upon which other protocols can be implemented. Each protocol implementation translates the ideas of its specific protocol onto this core.

The broker ships with a client implementation which interacts directly with this core. It uses what's called the "core" API, and it communicates over the network using the "core" protocol.

Supported Protocols & APIs

The broker has a pluggable protocol architecture. Protocol plugins come in the form of protocol modules. Each protocol module is included on the broker's class path and loaded by the broker at boot time. The broker ships with 5 protocol modules out of the box. The 5 modules offer support for the following protocols:

- AMQP
- OpenWire
- MQTT
- STOMP
- HornetQ

APIs and Other Interfaces

Although JMS is a standardized API, it does not define a network protocol. The ActiveMQ Artemis JMS 2.0 client is implemented on top of the core protocol. We also provide a client-side JNDI implementation.

The broker also ships with a REST messaging interface (not to be confused with the REST management API provided via our integration with Jolokia).

Configuring Acceptors

In order to make use of a particular protocol, a transport must be configured with the desired protocol enabled. There is a whole section on configuring transports that can be found here.

The default configuration shipped with the ActiveMQ Artemis distribution comes with a number of acceptors already defined, one for each of the above protocols plus a generic acceptor that supports all protocols. To enable protocols on a particular acceptor simply add the `protocols` url parameter to the acceptor url where the value is one or more protocols (separated by commas). If the `protocols` parameter is omitted from the url **all** protocols are enabled.

- The following example enables only MQTT on port 1883

```
<acceptors>
 <acceptor>tcp://localhost:1883?protocols=MQTT</acceptor>
</acceptors>
```

- The following example enables MQTT and AMQP on port 5672

```
<acceptors>
 <acceptor>tcp://localhost:5672?protocols=MQTT,AMQP</acceptor>
</acceptors>
```

- The following example enables **all** protocols on `61616`:

```
<acceptors>
 <acceptor>tcp://localhost:61616</acceptor>
</acceptors>
```

Here are the supported protocols and their corresponding value used in the `protocols` url parameter.

Protocol	`protocols` value
Core (Artemis & HornetQ native)	`CORE`
OpenWire (5.x native)	`OPENWIRE`
AMQP	`AMQP`
MQTT	`MQTT`
STOMP	`STOMP`

AMQP

Apache ActiveMQ Artemis supports the AMQP 1.0 specification. By default there are `acceptor` elements configured to accept AMQP connections on ports `61616` and `5672`.

See the general Protocols and Interoperability chapter for details on configuring an `acceptor` for AMQP.

You can use *any* AMQP 1.0 compatible clients.

A short list includes:

- qpid clients
- .NET Clients
- Javascript NodeJS
- Java Script RHEA
- ... and many others.

Examples

We have a few examples as part of the Artemis distribution:

- .NET:
  - ./examples/protocols/amqp/dotnet
- ProtonCPP
  - ./examples/protocols/amqp/proton-cpp
  - ./examples/protocols/amqp/proton-clustered-cpp
- Ruby
  - ./examples/protocols/amqp/proton-ruby
- Java (Using the qpid JMS Client)
  - ./examples/protocols/amqp/queue
- Interceptors
  - ./examples/features/standard/interceptor-amqp
  - ./examples/features/standard/broker-plugin

Message Conversions

The broker will not perform any message conversion to any other protocols when sending AMQP and receiving AMQP.

However if you intend your message to be received by an AMQP JMS Client, you must follow the JMS Mapping Conventions. If you send a body type that is not recognized by this specification the conversion between AMQP and any other protocol will make it a Binary Message. Make sure you follow these conventions if you intend to cross protocols or languages. Especially on the message body.

A compatibility setting allows aligning the naming convention of AMQP queues (JMS Durable and Shared Subscriptions) with CORE. For backwards compatibility reasons, you need to explicitly enable this via broker configuration:

- `amqp-use-core-subscription-naming`
  - `true` - use queue naming convention that is aligned with CORE.
  - `false` (default) - use older naming convention.

Intercepting and changing messages

We don't recommend changing messages at the server's side for a few reasons:

- AMQP messages are meant to be immutable
- The message won't be the original message the user sent
- AMQP has the possibility of signing messages. The signature would be broken.
- For performance reasons. We try not to re-encode (or even decode) messages.

If regardless these recommendations you still need and want to intercept and change AMQP messages, look at the aforementioned interceptor examples.

AMQP and security

The Apache ActiveMQ Artemis Server accepts the PLAIN, ANONYMOUS, and GSSAPI SASL mechanism. These are implemented on the broker's security infrastructure.

AMQP and destinations

If an AMQP Link is dynamic then a temporary queue will be created and either the remote source or remote target address will be set to the name of the temporary queue. If the Link is not dynamic then the address of the remote target or source will be used for the queue. In case it does not exist, it will be auto-created if the settings allow.

AMQP and Multicast Addresses (Topics)

Although AMQP has no notion of "topics" it is still possible to treat AMQP consumers or receivers as subscriptions rather than just consumers on a queue. By default any receiving link that attaches to an address that has only `multicast` enabled will be treated as a subscription and a corresponding subscription queue will be created. If the Terminus Durability is either `UNSETTLED_STATE` or `CONFIGURATION` then the queue will be made durable (similar to a JMS durable subscription) and given a name made up from the container id and the link name, something like `my-container-id:my-link-name` . If the Terminus Durability is configured as `NONE` then a volatile `multicast` queue will be created.

AMQP and Coordinations - Handling Transactions

An AMQP links target can also be a Coordinator. A Coordinator is used to handle transactions. If a coordinator is used then the underlying server session will be transacted and will be either rolled back or committed via the coordinator.

> **Note:**
>
> AMQP allows the use of multiple transactions per session, `amqp:multi-txns-per-ssn`, however in this version of Apache ActiveMQ Artemis will only support single transactions per session.

AMQP scheduling message delivery

An AMQP message can provide scheduling information that controls the time in the future when the message will be delivered at the earliest. This information is provided by adding a message annotation to the sent message.

There are two different message annotations that can be used to schedule a message for later delivery:

- `x-opt-delivery-time` The specified value must be a positive long corresponding to the time the message should be made available for delivery (in milliseconds).
- `x-opt-delivery-delay` The specified value must be a positive long corresponding to the amount of milliseconds after the broker receives the given message before it should be made available for delivery.

If both annotations are present in the same message then the broker will prefer the more specific `x-opt-delivery-time` value.

DLQ and Expiry transfer

AMQP Messages will be copied before transferred to a DLQ or ExpiryQueue and will receive properties and annotations during this process.

The broker also keeps an internal only property (called extra property) that is not exposed to the clients, and those will also be filled during this process.

Here is a list of Annotations and Property names AMQP Messages will receive when transferred:

Annotation name	Internal Property Name	Description
x-opt-ORIG-MESSAGE-ID	_AMQ_ORIG_MESSAGE_ID	The original message ID before the transfer
x-opt-ACTUAL-EXPIRY	_AMQ_ACTUAL_EXPIRY	When the expiry took place. Milliseconds since epoch times
x-opt-ORIG-QUEUE	_AMQ_ORIG_QUEUE	The original queue name before the transfer
x-opt-ORIG-ADDRESS	_AMQ_ORIG_ADDRESS	The original address name before the transfer

Filtering on Message Annotations

It is possible to filter on messaging annotations if you use the prefix "m." before the annotation name.

For example if you want to filter messages sent to a specific destination, you could create your filter accordingly to this:

```
ConnectionFactory factory = new JmsConnectionFactory("amqp://localhost:5672");
Connection connection = factory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();
javax.jms.Queue queue = session.createQueue("my-DLQ");


MessageConsumer consumer = session.createConsumer(queue, "\"m.x-opt-ORIG-ADDRE
Message message = consumer.receive();
```

The broker will set internal properties. If you intend to filter after DLQ or Expiry you may choose the internal property names:

```
// Replace the consumer creation on the previous example:
MessageConsumer consumer = session.createConsumer(queue, "_AMQ_ORIG_ADDRESS='O
```

Configuring AMQP Idle Timeout

It is possible to configure the AMQP Server's IDLE Timeout by setting the property amqpIdleTimeout in milliseconds on the acceptor.

This will make the server to send an AMQP frame open to the client, with your configured timeout / 2.

So, if you configured your AMQP Idle Timeout to be 60000, the server will tell the client to send frames every 30,000 milliseconds.

```
<acceptor name="amqp">.... ;amqpIdleTimeout=<configured-timeout>; ..... </acce
```

Disabling Keep alive checks

if you set amqpIdleTimeout=0 that will tell clients to not sending keep alive packets towards the server. On this case you will rely on TCP to determine when the socket needs to be closed.

```
<acceptor name="amqp">.... ;amqpIdleTimeout=0; ..... </acceptor>
```

This contains a real example for configuring amqpIdleTimeout:

```
<acceptor name="amqp">tcp://0.0.0.0:5672?amqpIdleTimeout=0;tcpSendBufferSize=1
```

Web Sockets

Apache ActiveMQ Artemis also supports AMQP over Web Sockets. Modern web browsers which support Web Sockets can send and receive AMQP messages.

AMQP over Web Sockets is supported via a normal AMQP acceptor:

```
<acceptor name="amqp-ws-acceptor">tcp://localhost:5672?protocols=AMQP</accepto
```

With this configuration, Apache ActiveMQ Artemis will accept AMQP connections over Web Sockets on the port `5672`. Web browsers can then connect to `ws://<server>:5672` using a Web Socket to send and receive AMQP messages.

MQTT

MQTT is a light weight, client to server, publish / subscribe messaging protocol. MQTT has been specifically designed to reduce transport overhead (and thus network traffic) and code footprint on client devices. For this reason MQTT is ideally suited to constrained devices such as sensors and actuators and is quickly becoming the defacto standard communication protocol for IoT.

Apache ActiveMQ Artemis supports MQTT v3.1.1 (and also the older v3.1 code message format). By default there are `acceptor` elements configured to accept MQTT connections on ports `61616` and `1883`.

See the general Protocols and Interoperability chapter for details on configuring an `acceptor` for MQTT.

The best source of information on the MQTT protocol is in the 3.1.1 specification.

Refer to the MQTT examples for a look at some of this functionality in action.

MQTT Quality of Service

MQTT offers 3 quality of service levels.

Each message (or topic subscription) can define a quality of service that is associated with it. The quality of service level defined on a topic is the maximum level a client is willing to accept. The quality of service level on a message is the desired quality of service level for this message. The broker will attempt to deliver messages to subscribers at the highest quality of service level based on what is defined on the message and topic subscription.

Each quality of service level offers a level of guarantee by which a message is sent or received:

- QoS 0: `AT MOST ONCE`

  Guarantees that a particular message is only ever received by the subscriber a maximum of one time. This does mean that the message may never arrive. The sender and the receiver will attempt to deliver the message, but if something fails and the message does not reach its destination (say due to a network connection) the message may be lost. This QoS has the least network traffic overhead and the least burden on the client and the broker and is often useful for telemetry data where it doesn't matter if some of the data is lost.

- QoS 1: `AT LEAST ONCE`

  Guarantees that a message will reach its intended recipient one or more times. The sender will continue to send the message until it receives an acknowledgment from the recipient, confirming it has received the message. The result of this QoS is that the recipient may receive the message multiple

times, and also increases the network overhead than QoS 0, (due to acks). In addition more burden is placed on the sender as it needs to store the message and retry should it fail to receive an ack in a reasonable time.

- QoS 2: `EXACTLY ONCE`

  The most costly of the QoS (in terms of network traffic and burden on sender and receiver) this QoS will ensure that the message is received by a recipient exactly one time. This ensures that the receiver never gets any duplicate copies of the message and will eventually get it, but at the extra cost of network overhead and complexity required on the sender and receiver.

MQTT Retain Messages

MQTT has an interesting feature in which messages can be "retained" for a particular address. This means that once a retain message has been sent to an address, any new subscribers to that address will receive the last sent retain message before any others messages, this happens even if the retained message was sent before a client has connected or subscribed. An example of where this feature might be useful is in environments such as IoT where devices need to quickly get the current state of a system when they are on boarded into a system.

Will Messages

A will message can be sent when a client initially connects to a broker. Clients are able to set a "will message" as part of the connect packet. If the client abnormally disconnects, say due to a device or network failure the broker will proceed to publish the will message to the specified address (as defined also in the connect packet). Other subscribers to the will topic will receive the will message and can react accordingly. This feature can be useful in an IoT style scenario to detect errors across a potentially large scale deployment of devices.

Debug Logging

Detailed protocol logging (e.g. packets in/out) can be activated via the following steps:

1. Open `<ARTEMIS_INSTANCE>/etc/logging.properties`
2. Add `org.apache.activemq.artemis.core.protocol.mqtt` to the `loggers` list.
3. Add this line to enable `TRACE` logging for this new logger: `logger.org.apache.activemq.artemis.core.protocol.mqtt.level=TRACE`
4. Ensure the `level` for the `handler` you want to log the message doesn't block the `TRACE` logging. For example, modify the `level` of the `CONSOLE handler` like so: `handler.CONSOLE.level=TRACE` .

The MQTT specification doesn't dictate the format of the payloads which clients publish. As far as the broker is concerned a payload is just just an array of bytes. However, to facilitate logging the broker will encode the payloads as UTF-8 strings and print them up to 256 characters. Payload logging is limited to avoid filling the logs with potentially hundreds of megabytes of unhelpful information.

Wild card subscriptions

MQTT addresses are hierarchical much like a file system, and they use a special character (i.e. `/` by default) to separate hierarchical levels. Subscribers are able to subscribe to specific topics or to whole branches of a hierarchy.

To subscribe to branches of an address hierarchy a subscriber can use wild cards. These wild cards (including the aforementioned separator) are configurable. See the Wildcard Syntax chapter for details about how to configure custom wild cards.

There are 2 types of wild cards in MQTT:

- **Multi level** (`#` by default)

  Adding this wild card to an address would match all branches of the address hierarchy under a specified node. For example: `/uk/#` Would match `/uk/cities` , `/uk/cities/newcastle` and also `/uk/rivers/tyne` . Subscribing to an address `#` would result in subscribing to all topics in the broker. This can be useful, but should be done so with care since it has significant performance implications.

- **Single level** (`+` by default)

  Matches a single level in the address hierarchy. For example `/uk/+/stores` would match `/uk/newcastle/stores` but not `/uk/cities/newcastle/stores` .

Web Sockets

Apache ActiveMQ Artemis also supports MQTT over Web Sockets. Modern web browsers which support Web Sockets can send and receive MQTT messages.

MQTT over Web Sockets is supported via a normal MQTT acceptor:

```
<acceptor name="mqtt-ws-acceptor">tcp://localhost:1883?protocols=MQTT</accepto
```

With this configuration, Apache ActiveMQ Artemis will accept MQTT connections over Web Sockets on the port `1883` . Web browsers can then connect to `ws://<server>:1883` using a Web Socket to send and receive MQTT messages.

STOMP

STOMP is a text-orientated wire protocol that allows STOMP clients to communicate with STOMP Brokers. Apache ActiveMQ Artemis supports STOMP 1.0, 1.1 and 1.2.

STOMP clients are available for several languages and platforms making it a good choice for interoperability.

By default there are `acceptor` elements configured to accept STOMP connections on ports `61616` and `61613`.

See the general Protocols and Interoperability chapter for details on configuring an `acceptor` for STOMP.

Refer to the STOMP examples for a look at some of this functionality in action.

Limitations

The STOMP specification identifies **transactional acknowledgements** as an optional feature. Support for transactional acknowledgements is not implemented in Apache ActiveMQ Artemis. The `ACK` frame can not be part of a transaction. It will be ignored if its `transaction` header is set.

Virtual Hosting

Apache ActiveMQ Artemis currently doesn't support virtual hosting, which means the `host` header in `CONNECT` frame will be ignored.

Mapping STOMP destinations to addresses and queues

STOMP clients deals with *destinations* when sending messages and subscribing. Destination names are simply strings which are mapped to some form of destination on the server - how the server translates these is left to the server implementation.

In Apache ActiveMQ Artemis, these destinations are mapped to *addresses* and *queues* depending on the operation being done and the desired semantics (e.g. anycast or multicast).

Logging

Incoming and outgoing STOMP frames can be logged by enabling `DEBUG` for `org.apache.activemq.artemis.core.protocol.stomp.StompConnection`. This can be extremely useful for debugging or simply monitoring client activity. Along with the

STOMP frame itself the remote IP address of the client is logged as well as the internal connection ID so that frames from the same client can be correlated.

Routing Semantics

The STOMP specification is intentionally ambiguous about message routing semantics. When providing an overview of the protocol the STOMP 1.2 specification says:

```
A STOMP server is modelled as a set of destinations to which messages can be s
The STOMP protocol treats destinations as opaque string and their syntax is
server implementation specific. Additionally STOMP does not define what the
delivery semantics of destinations should be. The delivery, or
"message exchange", semantics of destinations can vary from server to server a
even from destination to destination. This allows servers to be creative with
semantics that they can support with STOMP.
```

Therefore, there are a handful of different ways to specify which semantics are desired both on the client-side and broker-side.

Configuring Routing Semantics from the Client Side

Sending

When a STOMP client sends a message (using a `SEND` frame), the protocol manager looks at the `destination-type` header to determine where to route it and potentially how to create the address and/or queue to which it is being sent. Valid values are `ANYCAST` and `MULTICAST` (case sensitive). If no indication of routing type is supplied (either by the client or the broker) then the default defined in the corresponding `default-address-routing-type` & `default-queue-routing-type` address-settings will be used as necessary.

The `destination` header maps to an address of the same name if `MULTICAST` is used and additionally to a queue of the same name if `ANYCAST` is used.

Subscribing

When a STOMP client subscribes to a destination (using a `SUBSCRIBE` frame), the protocol manager looks at the `subscription-type` header frame to determine what subscription semantics to use and potentially how to create the address and/or queue for the subscription. If no indication of routing type is supplied (either by the client or the broker) then the default defined in the corresponding `default-address-routing-type` & `default-queue-routing-type` address-settings will be used as necessary.

The `destination` header maps to an address of the same name if `MULTICAST` is used and additionally to a queue of the same name if `ANYCAST` is used.

Configuring Routing Semantics from the Broker side

On the broker-side there are two main options for specifying routing semantics - prefixes and address settings

Prefixes

Using prefixes involves specifying the `anycastPrefix` and/or the `multicastPrefix` on the acceptor which the STOMP client is using. For the STOMP use-case these prefixes tell the broker that destinations using them should be treated as anycast or multicast. For example, if the acceptor has `anycastPrefix=queue/` then when a STOMP client sends a message to `destination:queue/foo` the broker will auto-create the address `foo` and queue `foo` appropriately as anycast and the message will be placed in that queue. Additionally, if the acceptor has `multicastPrefix=topic/` then when a STOMP client sends a message to `destination:topic/bar` the broker will auto-create the address bar as multicast, but it won't create a queue since multicast (i.e. pub/sub) semantics require a client to explicitly create a subscription to receive those messages.

Note: The `anycastPrefix` and/or `multicastPrefix` on the acceptor will be stripped from the `destination` value.

Address Settings

Using address settings involves defining address-setting elements whose `match` corresponds with the destination names the clients will use along with the proper `delimiter` to enabled matching. For example, broker.xml could use the following:

```
<address-settings>
   <address-setting match="queue/#">
      <default-address-routing-type>ANYCAST</default-address-routing-type>
      <default-queue-routing-type>ANYCAST</default-queue-routing-type>
   </address>
   <address-setting match="topic/#">
      <default-address-routing-type>MULTICAST</default-address-routing-type>
      <default-queue-routing-type>MULTICAST</default-queue-routing-type>
   </address>
</address-settings>
<wildcard-addresses>
   <delimiter>/</delimiter>
</wildcard-addresses>
```

Then if a STOMP client sends a message to `destination:queue/foo` the broker will auto-create the address `queue/foo` and queue `queue/foo` appropriately as anycast and the message will be placed in that queue. Additionally, if a STOMP client sends a message to `destination:topic/bar` the broker will auto-create the address `topic/bar` as multicast, but it won't create a queue as previously explained.

STOMP heart-beating and connection-ttl

Well behaved STOMP clients will always send a `DISCONNECT` frame before closing their connections. In this case the server will clear up any server side resources such as sessions and consumers synchronously. However if STOMP clients exit without sending a `DISCONNECT` frame or if they crash the server will have no way of knowing immediately whether the client is still alive or not. STOMP connections therefore default to a `connection-ttl` value of 1 minute (see chapter on connection-ttl for more information. This value can be overridden using the `connection-ttl-override` property or if you need a specific connectionTtl for your stomp connections without affecting the broker-wide `connection-ttl-override` setting, you can configure your stomp acceptor with the `connectionTtl` property, which is used to set the ttl for connections that are created from that acceptor. For example:

```
<acceptor name="stomp-acceptor">tcp://localhost:61613?protocols=STOMP;connecti
```

The above configuration will make sure that any STOMP connection that is created from that acceptor and does not include a `heart-beat` header or disables client-to-server heart-beats by specifying a `0` value will have its `connection-ttl` set to 20 seconds. The `connectionTtl` set on an acceptor will take precedence over `connection-ttl-override`. The default `connectionTtl` is 60,000 milliseconds.

Since STOMP 1.0 does not support heart-beating then all connections from STOMP 1.0 clients will have a connection TTL imposed upon them by the broker based on the aforementioned configuration options. Likewise, any STOMP 1.1 or 1.2 clients that don't specify a `heart-beat` header or disable client-to-server heart-beating (e.g. by sending `0,X` in the `heart-beat` header) will have a connection TTL imposed upon them by the broker.

For STOMP 1.1 and 1.2 clients which send a non-zero client-to-server `heart-beat` header value then their connection TTL will be set accordingly. However, the broker will not strictly set the connection TTL to the same value as the specified in the `heart-beat` since even small network delays could then cause spurious disconnects. Instead, the client-to-server value in the `heart-beat` will be multiplied by the `heartBeatToConnectionTtlModifier` specified on the acceptor. The `heartBeatToConnectionTtlModifier` is a decimal value that defaults to `2.0` so for example, if a client sends a `heart-beat` header of `1000,0` the the connection TTL will be set to `2000` so that the data or ping frames sent every 1000 milliseconds will have a sufficient cushion so as not to be considered late and trigger a disconnect. This is also in accordance with the STOMP 1.1 and 1.2 specifications which both state, "because of timing inaccuracies, the receiver SHOULD be tolerant and take into account an error margin."

The minimum and maximum connection TTL allowed can also be specified on the acceptor via the `connectionTtlMin` and `connectionTtlMax` properties respectively. The default `connectionTtlMin` is 1000 and the default `connectionTtlMax` is Java's `Long.MAX_VALUE` meaning there essentially is no max connection TTL by default. Keep in mind that the `heartBeatToConnectionTtlModifier` is relevant here. For example, if a client sends a `heart-beat` header of `20000,0` and the acceptor is using a `connectionTtlMax` of `30000` and a default

`heartBeatToConnectionTtlModifier` of `2.0` then the connection TTL would be `40000` (i.e. `20000` * `2.0`) which would exceed the `connectionTtlMax`. In this case the server would respond to the client with a `heart-beat` header of `0,15000` (i.e. `30000` / `2.0`). As described previously, this is to make sure there is a sufficient cushion for the client heart-beats in accordance with the STOMP 1.1 and 1.2 specifications. The same kind of calculation is done for `connectionTtlMin`.

The minimum server-to-client heart-beat value is 500ms.

> **Note:**
>
> Please note that the STOMP protocol version 1.0 does not contain any heart-beat frame. It is therefore the user's responsibility to make sure data is sent within connection-ttl or the server will assume the client is dead and clean up server side resources. With STOMP 1.1 users can use heart-beats to maintain the life cycle of stomp connections.

Selector/Filter expressions

STOMP subscribers can specify an expression used to select or filter what the subscriber receives using the `selector` header. The filter expression syntax follows the *core filter syntax* described in the Filter Expressions documentation.

STOMP and JMS interoperability

Sending and consuming STOMP message from JMS or Core API

STOMP is mainly a text-orientated protocol. To make it simpler to interoperate with JMS and Core API, our STOMP implementation checks for presence of the `content-length` header to decide how to map a STOMP 1.0 message to a JMS Message or a Core message.

If the STOMP 1.0 message does *not* have a `content-length` header, it will be mapped to a JMS *TextMessage* or a Core message with a *single nullable SimpleString in the body buffer*.

Alternatively, if the STOMP 1.0 message *has* a `content-length` header, it will be mapped to a JMS *BytesMessage* or a Core message with a *byte[] in the body buffer*.

The same logic applies when mapping a JMS message or a Core message to STOMP. A STOMP 1.0 client can check the presence of the `content-length` header to determine the type of the message body (String or bytes).

Message IDs for STOMP messages

When receiving STOMP messages via a JMS consumer or a QueueBrowser, the messages have no properties like JMSMessageID by default. However this may bring some inconvenience to clients who wants an ID for their purpose. The broker STOMP provides a parameter to enable message ID on each incoming STOMP message. If you want each STOMP message to have a unique ID, just set the `stompEnableMessageId` to true. For example:

```
<acceptor name="stomp-acceptor">tcp://localhost:61613?protocols=STOMP;stompEna
```

When the server starts with the above setting, each stomp message sent through this acceptor will have an extra property added. The property key is `amqMessageId` and the value is a String representation of a long type internal message id prefixed with `STOMP`, like:

```
amqMessageId : STOMP12345
```

The default `stompEnableMessageId` value is `false`.

Durable Subscriptions

The `SUBSCRIBE` and `UNSUBSCRIBE` frames can be augmented with special headers to create and destroy durable subscriptions respectively.

To create a durable subscription the `client-id` header must be set on the `CONNECT` frame and the `durable-subscription-name` must be set on the `SUBSCRIBE` frame. The combination of these two headers will form the identity of the durable subscription.

To delete a durable subscription the `client-id` header must be set on the `CONNECT` frame and the `durable-subscription-name` must be set on the `UNSUBSCRIBE` frame. The values for these headers should match what was set on the `SUBSCRIBE` frame to delete the corresponding durable subscription.

Aside from `durable-subscription-name`, the broker also supports `durable-subscriber-name` (a deprecated property used before `durable-subscription-name`) as well as `activemq.subscriptionName` from ActiveMQ 5.x. This is the order of precedence if the frame contains more than one of these:

1) `durable-subscriber-name` 2) `durable-subscription-name` 3) `activemq.subscriptionName`

It is possible to pre-configure durable subscriptions since the STOMP implementation creates the queue used for the durable subscription in a deterministic way (i.e. using the format of `client-id.subscription-name`). For example, if you wanted to configure a durable subscription on the address `myAddress` with a client-id of `myclientid` and a subscription name of `mysubscription` then configure the durable subscription:

```
<addresses>
   <address name="myAddress">
      <multicast>
         <queue name="myclientid.mysubscription"/>
      </multicast>
   </address>
</addresses>
```

Handling of Large Messages with STOMP

STOMP clients may send very large frame bodies which can exceed the size of the broker's internal buffer, causing unexpected errors. To prevent this situation from happening, the broker provides a STOMP configuration attribute `stompMinLargeMessageSize`. This attribute can be configured inside a stomp acceptor, as a parameter. For example:

```
<acceptor name="stomp-acceptor">tcp://localhost:61613?protocols=STOMP;stompMin
```

The type of this attribute is integer. When this attributed is configured, the broker will check the size of the body of each STOMP frame arrived from connections established with this acceptor. If the size of the body is equal or greater than the value of `stompMinLargeMessageSize`, the message will be persisted as a large message. When a large message is delievered to a STOMP consumer, the broker will automatically handle the conversion from a large message to a normal message, before sending it to the client.

If a large message is compressed, the server will uncompressed it before sending it to stomp clients. The default value of `stompMinLargeMessageSize` is the same as the default value of minLargeMessageSize.

Web Sockets

Apache ActiveMQ Artemis also supports STOMP over Web Sockets. Modern web browsers which support Web Sockets can send and receive STOMP messages.

STOMP over Web Sockets is supported via the normal STOMP acceptor:

```
<acceptor name="stomp-ws-acceptor">tcp://localhost:61614?protocols=STOMP</acce
```

With this configuration, Apache ActiveMQ Artemis will accept STOMP connections over Web Sockets on the port `61614`. Web browsers can then connect to `ws://<server>:61614` using a Web Socket to send and receive STOMP messages.

A companion JavaScript library to ease client-side development is available from GitHub (please see its documentation for a complete description).

The payload length of Web Socket frames can vary between client implementations. By default the broker will accept frames with a payload length of 65,536. If the client needs to send payloads longer than this in a single frame this

length can be adjusted by using the `stompMaxFramePayloadLength` URL parameter on the acceptor.

The `stomp-websockets` example shows how to configure an Apache ActiveMQ Artemis broker to have web browsers and Java applications exchanges messages.

OpenWire

Apache ActiveMQ Artemis supports the OpenWire protocol so that an Apache ActiveMQ 5.x JMS client can talk directly to an Apache ActiveMQ Artemis server. By default there is an `acceptor` configured to accept OpenWire connections on port `61616`.

See the general Protocols and Interoperability chapter for details on configuring an `acceptor` for OpenWire.

Refer to the OpenWire examples for a look at this functionality in action.

Connection Monitoring

OpenWire has a few parameters to control how each connection is monitored, they are:

- `maxInactivityDuration`

  It specifies the time (milliseconds) after which the connection is closed by the broker if no data was received. Default value is 30000.

- `maxInactivityDurationInitalDelay`

  It specifies the maximum delay (milliseconds) before inactivity monitoring is started on the connection. It can be useful if a broker is under load with many connections being created concurrently. Default value is 10000.

- `useInactivityMonitor`

  A value of false disables the InactivityMonitor completely and connections will never time out. By default it is enabled. On broker side you don't neet set this. Instead you can set the connection-ttl to -1.

- `useKeepAlive`

  Whether or not to send a KeepAliveInfo on an idle connection to prevent it from timing out. Enabled by default. Disabling the keep alive will still make connections time out if no data was received on the connection for the specified amount of time.

Note at the beginning the InactivityMonitor negotiates the appropriate `maxInactivityDuration` and `maxInactivityDurationInitalDelay`. The shortest duration is taken for the connection.

Fore more details please see ActiveMQ InactivityMonitor.

Disable/Enable Advisories

By default, advisory topics (ActiveMQ Advisory) are created in order to send certain type of advisory messages to listening clients. As a result, advisory addresses and queues will be displayed on the management console, along with user deployed addresses and queues. This sometimes cause confusion because the advisory objects are internally managed without user being aware of them. In addition, users may not want the advisory topics at all (they cause extra resources and performance penalty) and it is convenient to disable them at all from the broker side.

The protocol provides two parameters to control advisory behaviors on the broker side.

- `supportAdvisory`

  Whether or not the broker supports advisory messages. If the value is true, advisory addresses/queues will be created. If the value is false, no advisory addresses/queues are created. Default value is `true`.

- `suppressInternalManagementObjects`

  Whether or not the advisory addresses/queues, if any, will be registered to management service (e.g. JMX registry). If set to true, no advisory addresses/queues will be registered. If set to false, those are registered and will be displayed on the management console. Default value is `true`.

The two parameters are configured on an OpenWire `acceptor`, e.g.:

```
<acceptor name="artemis">tcp://localhost:61616?protocols=OPENWIRE;supportAdvis
```

Virtual Topic Consumer Destination Translation

For existing OpenWire consumers of virtual topic destinations it is possible to configure a mapping function that will translate the virtual topic consumer destination into a FQQN address. This address will then represents the consumer as a multicast binding to an address representing the virtual topic.

The configuration string property `virtualTopicConsumerWildcards` has two parts separated by a `;`. The first is the 5.x style destination filter that identifies the destination as belonging to a virtual topic. The second identifies the number of `paths` that identify the consumer queue such that it can be parsed from the destination. For example, the default 5.x virtual topic with consumer prefix of `Consumer.*.`, would require a `virtualTopicConsumerWildcards` filter of `Consumer.*.>;2`. As a url parameter this transforms to `Consumer.*.%3E%3B2` when the url significant characters `>;` are escaped with their hex code points. In an `acceptor` url it would be:

```
<acceptor name="artemis">tcp://localhost:61616?protocols=OPENWIRE;virtualTopic
```

This will translate `Consumer.A.VirtualTopic.Orders` into a FQQN of `VirtualTopic.Orders::Consumer.A.VirtualTopic.Orders` using the int component `2` of the configuration to identify the consumer queue as the first two paths of the destination. `virtualTopicConsumerWildcards` is multi valued using a `,` separator.

Please see Virtual Topic Mapping example contained in the OpenWire examples.

Using Core

Apache ActiveMQ Artemis core is a messaging system with its own API. We call this the *core API*.

If you don't want to use the JMS API or any of the other supported protocols you can use the core API directly. The core API provides all the functionality of JMS but without much of the complexity. It also provides features that are not available using JMS.

Core Messaging Concepts

Some of the core messaging concepts are similar to JMS concepts, but core messaging concepts are also different in some ways as well. In general the core API is simpler than the JMS API, since we remove distinctions between queues, topics and subscriptions. We'll discuss each of the major core messaging concepts in turn, but to see the API in detail please consult the Javadoc.

Also refer to the addressing model chapter for a high-level overview of these concepts as well as configuration details.

Message

- A message is the unit of data which is sent between clients and servers.
- A message has a body which is a buffer containing convenient methods for reading and writing data into it.
- A message has a set of properties which are key-value pairs. Each property key is a string and property values can be of type integer, long, short, byte, byte[], String, double, float or boolean.
- A message has an *address* it is being sent to. When the message arrives on the server it is routed to any queues that are bound to the address. The routing semantics (i.e. anycast or multicast) are determined by the "routing type" of the address and queue. If the queues are bound with any filter, the message will only be routed to that queue if the filter matches. An address may have many queues bound to it or even none. There may also be entities other than queues (e.g. *diverts*) bound to addresses.
- Messages can be either durable or non durable. Durable messages in a durable queue will survive a server crash or restart. Non durable messages will never survive a server crash or restart.
- Messages can be specified with a priority value between 0 and 9. 0 represents the lowest priority and 9 represents the highest. The broker will attempt to deliver higher priority messages before lower priority ones.
- Messages can be specified with an optional expiry time. The broker will not deliver messages after its expiry time has been exceeded.

- Messages also have an optional timestamp which represents the time the message was sent.
- Apache ActiveMQ Artemis also supports the sending/consuming of very large messages much larger than can fit in available RAM at any one time.

Address

A server maintains a mapping between an address and a set of queues. Zero or more queues can be bound to a single address. Each queue can be bound with an optional message filter. When a message is routed, it is routed to the set of queues bound to the message's address. If any of the queues are bound with a filter expression, then the message will only be routed to the subset of bound queues which match that filter expression.

Other entities, such as *diverts* can also be bound to an address and messages will also be routed there.

> **Note:**
>
> Although core supports publish-subscribe semantics there is no such thing as a "topic" per se. "Topic" is mainly a JMS term. In core we just deal with *addresses*, *queues*, and *routing types*.
>
> For example, a JMS topic would be implemented by a single address to which many queues are bound using multicast routing. Each queue represents a "subscription" in normal "topic" terms. A JMS queue would be implemented as a single address to which one queue is bound using anycast routing - that queue represents the JMS queue.

Queue

Queues can be durable, meaning the messages they contain survive a server crash or restart, as long as the messages in them are durable. Non durable queues do not survive a server restart or crash even if the messages they contain are durable.

Queues can also be temporary, meaning they are automatically deleted when the client connection is closed, if they are not explicitly deleted before that.

Queues can be bound with an optional filter expression. If a filter expression is supplied then the server will only route messages that match that filter expression to any queues bound to the address.

Many queues can be bound to a single address. A particular queue is only bound to a maximum of one address.

Routing Type

The routing type determines the semantics used when routing messages to the queues bound to the address where the message was sent. Two types are supported:

- `ANYCAST`

  The message is routed to only **one** of the queues bound to the address. If multiple queues are bound to the address then messages are routed to them in a round-robin fashion.

- `MULTICAST`

  The message is route to **all** of the queues bound to the address.

Core API

ServerLocator

Clients use `ServerLocator` instances to create `ClientSessionFactory` instances. `ServerLocator` instances are used to locate servers and create connections to them.

In JMS terms think of a `ServerLocator` in the same way you would a JMS Connection Factory.

`ServerLocator` instances are created using the `ActiveMQClient` factory class.

ClientSessionFactory

Clients use `ClientSessionFactory` instances to create `ClientSession` instances. `ClientSessionFactory` instances are basically the connection to a server

In JMS terms think of them as JMS Connections.

`ClientSessionFactory` instances are created using the `ServerLocator` class.

ClientSession

A client uses a `ClientSession` for consuming and producing messages and for grouping them in transactions. `ClientSession` instances can support both transactional and non transactional semantics and also provide an `XAResource` interface so messaging operations can be performed as part of a JTA transaction.

`ClientSession` instances group `ClientConsumer` instances and `ClientProducer` instances.

`ClientSession` instances can be registered with an optional `SendAcknowledgementHandler`. This allows your client code to be notified asynchronously when sent messages have successfully reached the server. This unique Apache ActiveMQ Artemis feature, allows you to have full guarantees that sent messages have reached the server without having to block on each message sent until a response is received. Blocking on each messages sent is costly since it requires a network round trip for each message sent. By not blocking and receiving send acknowledgements asynchronously you can create true end to end asynchronous systems which is not possible using the standard JMS API. For more information on this advanced feature please see the section Guarantees of sends and commits.

ClientConsumer

Clients use `ClientConsumer` instances to consume messages from a queue. Core messaging supports both synchronous and asynchronous message consumption semantics. `ClientConsumer` instances can be configured with an optional filter expression and will only consume messages which match that expression.

ClientProducer

Clients create `ClientProducer` instances on `ClientSession` instances so they can send messages. `ClientProducer` instances can specify an address to which all sent messages are routed, or they can have no specified address, and the address is specified at send time for the message.

> **Warning**
>
> Please note that `ClientSession`, `ClientProducer` and `ClientConsumer` instances are *designed to be re-used*.
>
> It's an anti-pattern to create new `ClientSession`, `ClientProducer` and `ClientConsumer` instances for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning Performance Tuning.

A simple example of using Core

Here's a very simple program using the core messaging API to send and receive a message. Logically it's comprised of two sections: firstly setting up the producer to write a message to an *addresss*, and secondly, creating a *queue* for the consumer using anycast routing, creating the consumer, and *starting* it.

```
ServerLocator locator = ActiveMQClient.createServerLocator("vm://0");

// In this simple example, we just use one session for both producing and rece

ClientSessionFactory factory =  locator.createClientSessionFactory();
ClientSession session = factory.createSession();

// A producer is associated with an address ...

ClientProducer producer = session.createProducer("example");
ClientMessage message = session.createMessage(true);
message.getBodyBuffer().writeString("Hello");

// We need a queue attached to the address ...

session.createQueue("example", RoutingType.ANYCAST, "example", true);

// And a consumer attached to the queue ...

ClientConsumer consumer = session.createConsumer("example");

// Once we have a queue, we can send the message ...

producer.send(message);

// We need to start the session before we can -receive- messages ...

session.start();
ClientMessage msgReceived = consumer.receive();

System.out.println("message = " + msgReceived.getBodyBuffer().readString());

session.close();
```

Mapping JMS Concepts to the Core API

This chapter describes how JMS destinations are mapped to Apache ActiveMQ Artemis addresses.

Apache ActiveMQ Artemis core is JMS-agnostic. It does not have any concept of a JMS topic. A JMS topic is implemented in core as an address with name=(the topic name) and with a MULTICAST routing type with zero or more queues bound to it. Each queue bound to that address represents a topic subscription.

Likewise, a JMS queue is implemented as an address with name=(the JMS queue name) with an ANYCAST routing type associated with it.

Note: While it is possible to configure a JMS topic and queue with the same name, it is not a recommended configuration for use with cross protocol.

Using JMS

Although Apache ActiveMQ Artemis provides a JMS agnostic messaging API, many users will be more comfortable using JMS.

JMS is a very popular API standard for messaging, and most messaging systems provide a JMS API. If you are completely new to JMS we suggest you follow the Oracle JMS tutorial - a full JMS tutorial is out of scope for this guide.

Apache ActiveMQ Artemis also ships with a wide range of examples, many of which demonstrate JMS API usage. A good place to start would be to play around with the simple JMS Queue and Topic example, but we also provide examples for many other parts of the JMS API. A full description of the examples is available in Examples.

In this section we'll go through the main steps in configuring the server for JMS and creating a simple JMS program. We'll also show how to configure and use JNDI, and also how to use JMS with Apache ActiveMQ Artemis without using any JNDI.

A simple ordering system

For this chapter we're going to use a very simple ordering system as our example. It is a somewhat contrived example because of its extreme simplicity, but it serves to demonstrate the very basics of setting up and using JMS.

We will have a single JMS Queue called `OrderQueue`, and we will have a single `MessageProducer` sending an order message to the queue and a single `MessageConsumer` consuming the order message from the queue.

The queue will be a `durable` queue, i.e. it will survive a server restart or crash. We also want to pre-deploy the queue, i.e. specify the queue in the server configuration so it is created automatically without us having to explicitly create it from the client.

JNDI

The JMS specification establishes the convention that *administered objects* (i.e. JMS queue, topic and connection factory instances) are made available via the JNDI API. Brokers are free to implement JNDI as they see fit assuming the implementation fits the API. Apache ActiveMQ Artemis does not have a JNDI server. Rather, it uses a client-side JNDI implementation that relies on special properties set in the environment to construct the appropriate JMS objects. In other words, no objects are stored in JNDI on the Apache ActiveMQ Artemis server, instead they are simply instantiated on the client based on the provided configuration. Let's look at the different kinds of administered objects and how to configure them.

> **Note:**
>
> The following configuration properties *are strictly required when Apache ActiveMQ Artemis is running in stand-alone mode*. When Apache ActiveMQ Artemis is integrated to an application server (e.g. Wildfly) the application server itself will almost certainly provide a JNDI client with its own properties.

ConnectionFactory JNDI

A JMS connection factory is used by the client to make connections to the server. It knows the location of the server it is connecting to, as well as many other configuration parameters.

Here's a simple example of the JNDI context environment for a client looking up a connection factory to access an *embedded* instance of Apache ActiveMQ Artemis:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
connectionFactory.invmConnectionFactory=vm://0
```

In this instance we have created a connection factory that is bound to `invmConnectionFactory`, any entry with prefix `connectionFactory.` will create a connection factory.

In certain situations there could be multiple server instances running within a particular JVM. In that situation each server would typically have an InVM acceptor with a unique server-ID. A client using JMS and JNDI can account for this by specifying a connction factory for each server, like so:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
connectionFactory.invmConnectionFactory0=vm://0
connectionFactory.invmConnectionFactory1=vm://1
connectionFactory.invmConnectionFactory2=vm://2
```

Here is a list of all the supported URL schemes:

- `vm`
- `tcp`
- `udp`
- `jgroups`

Most clients won't be connecting to an embedded broker. Clients will most commonly connect across a network a remote broker. Here's a simple example of a client configuring a connection factory to connect to a remote broker running on myhost:5445:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
connectionFactory.ConnectionFactory=tcp://myhost:5445
```

In the example above the client is using the `tcp` scheme for the provider URL. A client may also specify multiple comma-delimited host:port combinations in the URL (e.g. `(tcp://remote-host1:5445,remote-host2:5445)`). Whether there is one or

many host:port combinations in the URL they are treated as the *initial connector(s)* for the underlying connection.

The `udp` scheme is also supported which should use a host:port combination that matches the `group-address` and `group-port` from the corresponding `broadcast-group` configured on the ActiveMQ Artemis server(s).

Each scheme has a specific set of properties which can be set using the traditional URL query string format (e.g. `scheme://host:port?key1=value1&key2=value2`) to customize the underlying transport mechanism. For example, if a client wanted to connect to a remote server using TCP and SSL it would create a connection factory like so, `tcp://remote-host:5445?ssl-enabled=true`.

All the properties available for the `tcp` scheme are described in the documentation regarding the Netty transport.

Note if you are using the `tcp` scheme and multiple addresses then a query can be applied to all the url's or just to an individual connector, so where you have

- `(tcp://remote-host1:5445?httpEnabled=true,remote-host2:5445?httpEnabled=true)?clientID=1234`

then the `httpEnabled` property is only set on the individual connectors where as the `clientId` is set on the actual connection factory. Any connector specific properties set on the whole URI will be applied to all the connectors.

The `udp` scheme supports 4 properties:

- `localAddress` - If you are running with multiple network interfaces on the same machine, you may want to specify that the discovery group listens only only a specific interface. To do this you can specify the interface address with this parameter.
- `localPort` - If you want to specify a local port to which the datagram socket is bound you can specify it here. Normally you would just use the default value of -1 which signifies that an anonymous port should be used. This parameter is always specified in conjunction with `localAddress`.
- `refreshTimeout` - This is the period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. You would normally set this to a value significantly higher than the broadcast-period on the broadcast group otherwise servers might intermittently disappear from the list even though they are still broadcasting due to slight differences in timing. This parameter is optional, the default value is 10000 milliseconds (10 seconds).
- `discoveryInitialWaitTimeout` - If the connection factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default value for this parameter is 10000 milliseconds.

Lastly, the `jgroups` scheme is supported which provides an alternative to the `udp` scheme for server discovery. The URL pattern is either `jgroups://channelName?file=jgroups-xml-conf-filename` where `jgroups-xml-conf-filename` refers to an XML file on the classpath that contains the JGroups configuration or it can be `jgroups://channelName?properties=some-jgroups-properties`. In both instance the `channelName` is the name given to the jgroups channel created.

The `refreshTimeout` and `discoveryInitialWaitTimeout` properties are supported just like with `udp`.

The default type for the default connection factory is of type `javax.jms.ConnectionFactory`. This can be changed by setting the type like so

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
java.naming.provider.url=tcp://localhost:5445?type=CF
```

In this example it is still set to the default, below shows a list of types that can be set.

Configuration for Connection Factory Types

type	interface
CF (default)	javax.jms.ConnectionFactory
XA_CF	javax.jms.XAConnectionFactory
QUEUE_CF	javax.jms.QueueConnectionFactory
QUEUE_XA_CF	javax.jms.XAQueueConnectionFactory
TOPIC_CF	javax.jms.TopicConnectionFactory
TOPIC_XA_CF	javax.jms.XATopicConnectionFactory

Destination JNDI

JMS destinations are also typically looked up via JNDI. As with connection factories, destinations can be configured using special properties in the JNDI context environment. The property *name* should follow the pattern: `queue.<jndi-binding>` or `topic.<jndi-binding>`. The property *value* should be the name of the queue hosted by the Apache ActiveMQ Artemis server. For example, if the server had a JMS queue configured like so:

```
<address name="OrderQueue">
   <queue name="OrderQueue"/>
</address>
```

And if the client wanted to bind this queue to "queues/OrderQueue" then the JNDI properties would be configured like so:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
java.naming.provider.url=tcp://myhost:5445
queue.queues/OrderQueue=OrderQueue
```

It is also possible to look-up JMS destinations which haven't been configured explicitly in the JNDI context environment. This is possible using `dynamicQueues/` or `dynamicTopics/` in the look-up string. For example, if the client wanted to look-up the aforementioned "OrderQueue" it could do so simply by using the string "dynamicQueues/OrderQueue". Note, the text that follows `dynamicQueues/` or `dynamicTopics/` must correspond *exactly* to the name of the destination on the server.

The code

Here's the code for the example:

First we'll create a JNDI initial context from which to lookup our JMS objects. If the above properties are set in `jndi.properties` and it is on the classpath then any new, empty `InitialContext` will be initialized using those properties:

```
InitialContext ic = new InitialContext();

//Now we'll look up the connection factory from which we can create
//connections to myhost:5445:

ConnectionFactory cf = (ConnectionFactory)ic.lookup("ConnectionFactory");

//And look up the Queue:

Queue orderQueue = (Queue)ic.lookup("queues/OrderQueue");

//Next we create a JMS connection using the connection factory:

Connection connection = cf.createConnection();

//And we create a non transacted JMS Session, with AUTO\_ACKNOWLEDGE
//acknowledge mode:

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//We create a MessageProducer that will send orders to the queue:

MessageProducer producer = session.createProducer(orderQueue);

//And we create a MessageConsumer which will consume orders from the
//queue:

MessageConsumer consumer = session.createConsumer(orderQueue);

//We make sure we start the connection, or delivery won't occur on it:

connection.start();

//We create a simple TextMessage and send it:

TextMessage message = session.createTextMessage("This is an order");
producer.send(message);

//And we consume the message:

TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());
```

It is as simple as that. For a wide range of working JMS examples please see the examples directory in the distribution.

> **Warning**
>
> Please note that JMS connections, sessions, producers and consumers are *designed to be re-used*.
>
> It is an anti-pattern to create new connections, sessions, producers and consumers for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning Performance Tuning.

Directly instantiating JMS Resources without using JNDI

Although it is a very common JMS usage pattern to lookup JMS *Administered Objects* (that's JMS Queue, Topic and ConnectionFactory instances) from JNDI, in some cases you just think "Why do I need JNDI? Why can't I just instantiate these objects directly?"

With Apache ActiveMQ Artemis you can do exactly that. Apache ActiveMQ Artemis supports the direct instantiation of JMS Queue, Topic and ConnectionFactory instances, so you don't have to use JNDI at all.

> For a full working example of direct instantiation please look at the Instantiate JMS Objects Directly example under the JMS section of the examples.

Here's our simple example, rewritten to not use JNDI at all:

We create the JMS ConnectionFactory object via the ActiveMQJMSClient Utility class, note we need to provide connection parameters and specify which transport we are using, for more information on connectors please see Configuring the Transport.

```
TransportConfiguration transportConfiguration = new TransportConfiguration(Net

ConnectionFactory cf = ActiveMQJMSClient.createConnectionFactoryWithoutHA(JMSF

//We also create the JMS Queue object via the ActiveMQJMSClient Utility
//class:

Queue orderQueue = ActiveMQJMSClient.createQueue("OrderQueue");

//Next we create a JMS connection using the connection factory:

Connection connection = cf.createConnection();

//And we create a non transacted JMS Session, with AUTO\_ACKNOWLEDGE
//acknowledge mode:

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);


//We create a MessageProducer that will send orders to the queue:

MessageProducer producer = session.createProducer(orderQueue);

//And we create a MessageConsumer which will consume orders from the
//queue:

MessageConsumer consumer = session.createConsumer(orderQueue);

//We make sure we start the connection, or delivery won't occur on it:

connection.start();

//We create a simple TextMessage and send it:

TextMessage message = session.createTextMessage("This is an order");
producer.send(message);

//And we consume the message:

TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());
```

Setting The Client ID

This represents the client id for a JMS client and is needed for creating durable subscriptions. It is possible to configure this on the connection factory and can be set via the `clientId` element. Any connection created by this connection factory will have this set as its client id.

Setting The Batch Size for DUPS_OK

When the JMS acknowledge mode is set to `DUPS_OK` it is possible to configure the consumer so that it sends acknowledgements in batches rather that one at a time, saving valuable bandwidth. This can be configured via the connection factory via the `dupsOkBatchSize` element and is set in bytes. The default is 1024 * 1024 bytes = 1 MiB.

Setting The Transaction Batch Size

When receiving messages in a transaction it is possible to configure the consumer to send acknowledgements in batches rather than individually saving valuable bandwidth. This can be configured on the connection factory via the `transactionBatchSize` element and is set in bytes. The default is 1024 * 1024.

Setting The Destination Cache

Many frameworks such as Spring resolve the destination by name on every operation, this can cause a performance issue and extra calls to the broker, in a scenario where destinations (addresses) are permanent broker side, such as they are managed by a platform or operations team. using `cacheDestinations` element, you can toggle on the destination cache to improve the performance and reduce the calls to the broker. This should not be used if destinations (addresses) are not permanent broker side, as in dynamic creation/deletion.

The Client Classpath

Apache ActiveMQ Artemis requires just a single jar on the *client classpath*.

> **Warning**
>
> The client jar mentioned here can be found in the `lib/client` directory of the Apache ActiveMQ Artemis distribution. Be sure you only use the jar from the correct version of the release, you *must not* mix and match versions of jars from different Apache ActiveMQ Artemis versions. Mixing and matching different jar versions may cause subtle errors and failures to occur.

Whether you are using JMS or just the Core API simply add the `artemis-jms-client-all.jar` from the `lib/client` directory to your client classpath. This is a "shaded" jar that contains all the Artemis code plus dependencies (e.g. JMS spec, Netty, etc.).

Examples

The Apache ActiveMQ Artemis distribution comes with over 90 run out-of-the-box examples demonstrating many of the features.

The examples are available in both the binary and source distribution under the `examples` directory. Examples are split by the following source tree:

- features - Examples containing broker specific features.
  - clustered - examples showing load balancing and distribution capabilities.
  - ha - examples showing failover and reconnection capabilities.
  - perf - examples allowing you to run a few performance tests on the server
  - standard - examples demonstrating various broker features.
  - sub-modules - examples of integrated external modules.
- protocols - Protocol specific examples
  - amqp
  - mqtt
  - openwire
  - stomp

Running the Examples

To run any example, simply `cd` into the appropriate example directory and type `mvn verify` or `mvn install` (For details please read the readme.html in each example directory).

You can use the profile `-Pexamples` to run multiple examples under any example tree.

For each example, you will have a created server under `./target/server0` (some examples use more than one server).

You have the option to prevent the example from starting the server (e.g. if you want to start the server manually) by simply specifying the `-PnoServer` profile, e.g.:

```
# running an example without running the server
mvn verify -PnoServer
```

Also under `./target` there will be a script repeating the commands to create each server. Here is the `create-server0.sh` generated by the `Queue` example. This is useful to see exactly what command(s) are required to configure the server(s).

```
# These are the commands used to create server0
/myInstallDirectory/apache-artemis/bin/artemis create --allow-anonymous --sile
```

Several examples use UDP clustering which may not work in your environment by default. On linux the command would be:

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
```

This command should be run as root. This will redirect any traffic directed to `224.0.0.0` to the loopback interface. On Mac OS X, the command is slightly different:

```
sudo route add 224.0.0.0 127.0.0.1 -netmask 240.0.0.0
```

All the examples use the Maven plugin, which can be useful for running your test servers as well.

This is the common output when running an example. On this case taken from the `Queue` example:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building ActiveMQ Artemis JMS Queue Example 2.5.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-enforcer-plugin:1.4:enforce (enforce-maven) @ queue ---
[INFO]
[INFO] --- maven-enforcer-plugin:1.4:enforce (enforce-java) @ queue ---
[INFO]
[INFO] --- maven-remote-resources-plugin:1.5:process (process-resource-bundles
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ queue --
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 3 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ queue ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-checkstyle-plugin:2.17:check (default) @ queue ---
[INFO]
[INFO] --- apache-rat-plugin:0.12:check (default) @ queue ---
[INFO] RAT will not execute since it is configured to be skipped via system pr
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /home/user/activemq-artemis/example
[INFO] Copying 3 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ queue
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.18.1:test (default-test) @ queue ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ queue ---
[INFO] Building jar: /home/user/activemq-artemis/examples/features/standard/qu
[INFO]
[INFO] --- maven-site-plugin:3.3:attach-descriptor (attach-descriptor) @ queue
[INFO]
[INFO] >>> maven-source-plugin:2.2.1:jar (attach-sources) > generate-sources @
[INFO]
[INFO] --- maven-enforcer-plugin:1.4:enforce (enforce-maven) @ queue ---
[INFO]
[INFO] --- maven-enforcer-plugin:1.4:enforce (enforce-java) @ queue ---
[INFO]
[INFO] <<< maven-source-plugin:2.2.1:jar (attach-sources) < generate-sources @
[INFO]
[INFO]
[INFO] --- maven-source-plugin:2.2.1:jar (attach-sources) @ queue ---
[INFO] Building jar: /home/user/activemq-artemis/examples/features/standard/qu
[INFO]
[INFO] >>> maven-source-plugin:2.2.1:jar (default) > generate-sources @ queue
[INFO]
[INFO] --- maven-enforcer-plugin:1.4:enforce (enforce-maven) @ queue ---
[INFO]
[INFO] --- maven-enforcer-plugin:1.4:enforce (enforce-java) @ queue ---
[INFO]
[INFO] <<< maven-source-plugin:2.2.1:jar (default) < generate-sources @ queue
[INFO]
[INFO]
[INFO] --- maven-source-plugin:2.2.1:jar (default) @ queue ---
[INFO]
[INFO] --- dependency-check-maven:1.4.3:check (default) @ queue ---
[INFO] Skipping dependency-check
```

```
[INFO]
[INFO] --- artemis-maven-plugin:2.5.0:create (create) @ queue ---
[INFO] Local       id: local
      url: file:///home/user/.m2/repository/
   layout: default
snapshots: [enabled => true, update => always]
 releases: [enabled => true, update => always]

[INFO] Entries.size 2
[INFO] ... key=project = MavenProject: org.apache.activemq.examples.broker:que
[INFO] ... key=pluginDescriptor = Component Descriptor: role: 'org.apache.mave
role: 'org.apache.maven.plugin.Mojo', implementation: 'org.apache.activemq.art
role: 'org.apache.maven.plugin.Mojo', implementation: 'org.apache.activemq.art
role: 'org.apache.maven.plugin.Mojo', implementation: 'org.apache.activemq.art
---
Executing org.apache.activemq.artemis.cli.commands.Create create --allow-anony
Home::/home/user/activemq-artemis/examples/features/standard/queue/../../../..
Creating ActiveMQ Artemis instance at: /home/user/activemq-artemis/examples/fe

You can now start the broker by executing:

   "/home/user/activemq-artemis/examples/features/standard/queue/target/server

Or you can run the broker in the background using:

   "/home/user/activemq-artemis/examples/features/standard/queue/target/server

[INFO] ------------------------------------------------------------------------
[INFO] create-server0.sh created with commands to reproduce server0
[INFO] under /home/user/activemq-artemis/examples/features/standard/queue/targ
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- artemis-maven-plugin:2.5.0:cli (start) @ queue ---
[INFO] awaiting server to start
server-out:     _        _               _
server-out:    / \  ____| |_  ___ __  __(_) _____
server-out:   / _ \|  _ \ __|/ _ \  \/  | |/  __/
server-out:  / ___ \ | \/ |_/  __/ |\/| | |\___ \
server-out: /_/   \_\|   \__\____|_|  |_|_|/___ /
server-out: Apache ActiveMQ Artemis 2.5.0
server-out:
server-out:
server-out:2018-03-13 09:06:37,980 WARN  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,052 INFO  [org.apache.activemq.artemis.integrat
[INFO] awaiting server to start
server-out:2018-03-13 09:06:38,123 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,146 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,178 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,197 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,198 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,198 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,198 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,199 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,199 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,261 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,262 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,386 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,445 INFO  [org.apache.activemq.artemis.core.ser
[INFO] awaiting server to start
server-out:2018-03-13 09:06:38,739 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,741 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,742 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,744 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,746 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,752 INFO  [org.apache.activemq.artemis.core.ser
server-out:2018-03-13 09:06:38,752 INFO  [org.apache.activemq.artemis.core.ser
```

```
[INFO] Server started
[INFO]
[INFO] --- artemis-maven-plugin:2.5.0:runClient (runClient) @ queue ---
Sent message: This is a text message
Received message: This is a text message
[INFO]
[INFO] --- artemis-maven-plugin:2.5.0:cli (stop) @ queue ---
server-out:2018-03-13 09:06:40,888 INFO  [org.apache.activemq.artemis.core.ser
server-out:Server stopped!
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6.641 s
[INFO] Finished at: 2018-03-13T09:06:40-05:00
[INFO] Final Memory: 43M/600M
[INFO] ------------------------------------------------------------------------
```

This includes a preview list of a few examples that we distribute with Artemis. Please refer to the distribution for a more accurate list.

Applet

This example shows you how to send and receive JMS messages from an Applet.

Application-Layer Failover

Apache ActiveMQ Artemis also supports Application-Layer failover, useful in the case that replication is not enabled on the server side.

With Application-Layer failover, it's up to the application to register a JMS `ExceptionListener` with Apache ActiveMQ Artemis which will be called by Apache ActiveMQ Artemis in the event that connection failure is detected.

The code in the `ExceptionListener` then recreates the JMS connection, session, etc on another node and the application can continue.

Application-layer failover is an alternative approach to High Availability (HA). Application-layer failover differs from automatic failover in that some client side coding is required in order to implement this. Also, with Application-layer failover, since the old session object dies and a new one is created, any uncommitted work in the old session will be lost, and any unacknowledged messages might be redelivered.

Core Bridge Example

The `bridge` example demonstrates a core bridge deployed on one server, which consumes messages from a local queue and forwards them to an address on a second server.

Core bridges are used to create message flows between any two Apache ActiveMQ Artemis servers which are remotely separated. Core bridges are resilient and will cope with temporary connection failure allowing them to be an

ideal choice for forwarding over unreliable connections, e.g. a WAN.

Browser

The `browser` example shows you how to use a JMS `QueueBrowser` with Apache ActiveMQ Artemis.

Queues are a standard part of JMS, please consult the JMS 2.0 specification for full details.

A `QueueBrowser` is used to look at messages on the queue without removing them. It can scan the entire content of a queue or only messages matching a message selector.

Camel

The `camel` example demonstrates how to build and deploy a Camel route to the broker using a web application archive (i.e. `war` file).

Client Kickoff

The `client-kickoff` example shows how to terminate client connections given an IP address using the JMX management API.

Client side failover listener

The `client-side-failoverlistener` example shows how to register a listener to monitor failover events

Client-Side Load-Balancing

The `client-side-load-balancing` example demonstrates how sessions created from a single JMS `Connection` can be created to different nodes of the cluster. In other words it demonstrates how Apache ActiveMQ Artemis does client-side load-balancing of sessions across the cluster.

Clustered Durable Subscription

This example demonstrates a clustered JMS durable subscription

Clustered Grouping

This is similar to the message grouping example except that it demonstrates it working over a cluster. Messages sent to different nodes with the same group id will be sent to the same node and the same consumer.

Clustered Queue

The `clustered-queue` example demonstrates a queue deployed on two different nodes. The two nodes are configured to form a cluster. We then create a consumer for the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

Clustering with JGroups

The `clustered-jgroups` example demonstrates how to form a two node cluster using JGroups as its underlying topology discovery technique, rather than the default UDP broadcasting. We then create a consumer for the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

Clustered Standalone

The `clustered-standalone` example demonstrates how to configure and starts 3 cluster nodes on the same machine to form a cluster. A subscriber for a JMS topic is created on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that the 3 subscribers receive all the sent messages.

Clustered Static Discovery

This example demonstrates how to configure a cluster using a list of connectors rather than UDP for discovery

Clustered Static Cluster One Way

This example demonstrates how to set up a cluster where cluster connections are one way, i.e. server A -> Server B -> Server C

Clustered Topic

The `clustered-topic` example demonstrates a JMS topic deployed on two different nodes. The two nodes are configured to form a cluster. We then create a subscriber on the topic on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both subscribers receive all the sent messages.

Message Consumer Rate Limiting

With Apache ActiveMQ Artemis you can specify a maximum consume rate at which a JMS MessageConsumer will consume messages. This can be specified when creating or deploying the connection factory.

If this value is specified then Apache ActiveMQ Artemis will ensure that messages are never consumed at a rate higher than the specified rate. This is a form of consumer throttling.

Dead Letter

The `dead-letter` example shows you how to define and deal with dead letter messages. Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back).

Such a message goes back to the JMS destination ready to be redelivered. However, this means it is possible for a message to be delivered again and again without any success and remain in the destination, clogging the system.

To prevent this, messaging systems define dead letter messages: after a specified unsuccessful delivery attempts, the message is removed from the destination and put instead in a dead letter destination where they can be consumed for further investigation.

Delayed Redelivery

The `delayed-redelivery` example demonstrates how Apache ActiveMQ Artemis can be configured to provide a delayed redelivery in the case a message needs to be redelivered.

Delaying redelivery can often be useful in the case that clients regularly fail or roll-back. Without a delayed redelivery, the system can get into a "thrashing" state, with delivery being attempted, the client rolling back, and delivery being re-attempted in quick succession, using up valuable CPU and network resources.

Divert

Apache ActiveMQ Artemis diverts allow messages to be transparently "diverted" or copied from one address to another with just some simple configuration defined on the server side.

Durable Subscription

The `durable-subscription` example shows you how to use a durable subscription with Apache ActiveMQ Artemis. Durable subscriptions are a standard part of JMS, please consult the JMS 1.1 specification for full details.

Unlike non-durable subscriptions, the key function of durable subscriptions is that the messages contained in them persist longer than the lifetime of the subscriber - i.e. they will accumulate messages sent to the topic even if there is no active subscriber on them. They will also survive server restarts or crashes. Note that for the messages to be persisted, the messages sent to them must be marked as durable messages.

Embedded

The `embedded` example shows how to embed a broker within your own code using POJO instantiation and no config files.

Embedded Simple

The `embedded-simple` example shows how to embed a broker within your own code using regular Apache ActiveMQ Artemis XML files.

Exclusive Queue

The `exlusive-queue` example shows you how to use exclusive queues, that route all messages to only one consumer at a time.

Message Expiration

The `expiry` example shows you how to define and deal with message expiration. Messages can be retained in the messaging system for a limited period of time before being removed. JMS specification states that clients should not receive messages that have been expired (but it does not guarantee this will not happen).

Apache ActiveMQ Artemis can assign an expiry address to a given queue so that when messages are expired, they are removed from the queue and sent to the expiry address. These "expired" messages can later be consumed from the expiry address for further inspection.

Apache ActiveMQ Artemis Resource Adapter example

This examples shows how to build the activemq resource adapters a rar for deployment in other Application Server's

HTTP Transport

The `http-transport` example shows you how to configure Apache ActiveMQ Artemis to use the HTTP protocol as its transport layer.

Instantiate JMS Objects Directly

Usually, JMS Objects such as `ConnectionFactory`, `Queue` and `Topic` instances are looked up from JNDI before being used by the client code. This objects are called "administered objects" in JMS terminology.

However, in some cases a JNDI server may not be available or desired. To come to the rescue Apache ActiveMQ Artemis also supports the direct instantiation of these administered objects on the client side so you don't have to use JNDI for JMS.

Interceptor

Apache ActiveMQ Artemis allows an application to use an interceptor to hook into the messaging system. Interceptors allow you to handle various message events in Apache ActiveMQ Artemis.

Interceptor AMQP

Similar to the Interceptor example, but using AMQP interceptors.

Interceptor Client

Similar to the Interceptor example, but using interceptors on the **client** rather than the broker.

Interceptor MQTT

Similar to the Interceptor example, but using MQTT interceptors.

JAAS

The `jaas` example shows you how to configure Apache ActiveMQ Artemis to use JAAS for security. Apache ActiveMQ Artemis can leverage JAAS to delegate user authentication and authorization to existing security infrastructure.

JMS Auto Closable

The `jms-auto-closeable` example shows how JMS resources, such as connections, sessions and consumers, in JMS 2 can be automatically closed on error.

JMS Completion Listener

The `jms-completion-listener` example shows how to send a message asynchronously to Apache ActiveMQ Artemis and use a CompletionListener to be notified of the Broker receiving it.

JMS Bridge

The `jms-bridge` example shows how to setup a bridge between two standalone Apache ActiveMQ Artemis servers.

JMS Context

The `jms-context` example shows how to send and receive a message to/from an address/queue using Apache ActiveMQ Artemis by using a JMS Context.

A JMSContext is part of JMS 2.0 and combines the JMS Connection and Session Objects into a simple Interface.

JMS Shared Consumer

The `jms-shared-consumer` example shows you how can use shared consumers to share a subscription on a topic. In JMS 1.1 this was not allowed and so caused a scalability issue. In JMS 2 this restriction has been lifted so you can share the load across different threads and connections.

JMX Management

The `jmx` example shows how to manage Apache ActiveMQ Artemis using JMX.

Large Message

The `large-message` example shows you how to send and receive very large messages with Apache ActiveMQ Artemis. Apache ActiveMQ Artemis supports the sending and receiving of huge messages, much larger than can fit in available RAM on the client or server. Effectively the only limit to message size is the amount of disk space you have on the server.

Large messages are persisted on the server so they can survive a server restart. In other words Apache ActiveMQ Artemis doesn't just do a simple socket stream from the sender to the consumer.

Last-Value Queue

The `last-value-queue` example shows you how to define and deal with last-value queues. Last-value queues are special queues which discard any messages when a newer message with the same value for a well-defined last-value property

is put in the queue. In other words, a last-value queue only retains the last value.

A typical example for last-value queue is for stock prices, where you are only interested by the latest price for a particular stock.

Management

The `management` example shows how to manage Apache ActiveMQ Artemis using JMS Messages to invoke management operations on the server.

Management Notification

The `management-notification` example shows how to receive management notifications from Apache ActiveMQ Artemis using JMS messages. Apache ActiveMQ Artemis servers emit management notifications when events of interest occur (consumers are created or closed, addresses are created or deleted, security authentication fails, etc.).

Message Counter

The `message-counters` example shows you how to use message counters to obtain message information for a queue.

Message Group

The `message-group` example shows you how to configure and use message groups with Apache ActiveMQ Artemis. Message groups allow you to pin messages so they are only consumed by a single consumer. Message groups are sets of messages that has the following characteristics:

- Messages in a message group share the same group id, i.e. they have same JMSXGroupID string property values
- The consumer that receives the first message of a group will receive all the messages that belongs to the group

Message Group

The `message-group2` example shows you how to configure and use message groups with Apache ActiveMQ Artemis via a connection factory.

Message Priority

Message Priority can be used to influence the delivery order for messages.

It can be retrieved by the message's standard header field 'JMSPriority' as defined in JMS specification version 1.1.

The value is of type integer, ranging from 0 (the lowest) to 9 (the highest). When messages are being delivered, their priorities will effect their order of delivery. Messages of higher priorities will likely be delivered before those of lower priorities.

Messages of equal priorities are delivered in the natural order of their arrival at their destinations. Please consult the JMS 1.1 specification for full details.

Multiple Failover

This example demonstrates how to set up a live server with multiple backups

Multiple Failover Failback

This example demonstrates how to set up a live server with multiple backups but forcing failover back to the original live server

No Consumer Buffering

By default, Apache ActiveMQ Artemis consumers buffer messages from the server in a client side buffer before you actually receive them on the client side. This improves performance since otherwise every time you called receive() or had processed the last message in a `MessageListener onMessage()` method, the Apache ActiveMQ Artemis client would have to go the server to request the next message, which would then get sent to the client side, if one was available.

This would involve a network round trip for every message and reduce performance. Therefore, by default, Apache ActiveMQ Artemis pre-fetches messages into a buffer on each consumer.

In some case buffering is not desirable, and Apache ActiveMQ Artemis allows it to be switched off. This example demonstrates that.

Non-Transaction Failover With Server Data Replication

The `non-transaction-failover` example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a *non-transacted* JMS session failing over from live to backup when the live server is crashed.

Apache ActiveMQ Artemis implements failover of client connections between live and backup servers. This is implemented by the replication of state between live and backup nodes. When replication is configured and a live node crashes, the

client connections can carry and continue to send and consume messages. When non-transacted sessions are used, once and only once message delivery is not guaranteed and it is possible that some messages will be lost or delivered twice.

OpenWire

The `Openwire` example shows how to configure an Apache ActiveMQ Artemis server to communicate with an Apache ActiveMQ Artemis JMS client that uses open-wire protocol.

You will find the queue example for open wire, and the chat example. The virtual-topic-mapping examples shows how to map the ActiveMQ 5.x Virtual Topic naming convention to work with the Artemis Address model.

Paging

The `paging` example shows how Apache ActiveMQ Artemis can support huge queues even when the server is running in limited RAM. It does this by transparently *paging* messages to disk, and *depaging* them when they are required.

Pre-Acknowledge

Standard JMS supports three acknowledgement modes: `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`. For a full description on these modes please consult the JMS specification, or any JMS tutorial.

All of these standard modes involve sending acknowledgements from the client to the server. However in some cases, you really don't mind losing messages in event of failure, so it would make sense to acknowledge the message on the server before delivering it to the client. This example demonstrates how Apache ActiveMQ Artemis allows this with an extra acknowledgement mode.

Message Producer Rate Limiting

The `producer-rte-limit` example demonstrates how, with Apache ActiveMQ Artemis, you can specify a maximum send rate at which a JMS message producer will send messages.

Queue

A simple example demonstrating a queue.

Message Redistribution

The `queue-message-redistribution` example demonstrates message redistribution between queues with the same name deployed in different nodes of a cluster.

Queue Requestor

A simple example demonstrating a JMS queue requestor.

Queue with Message Selector

The `queue-selector` example shows you how to selectively consume messages using message selectors with queue consumers.

Reattach Node example

The `Reattach Node` example shows how a client can try to reconnect to the same server instead of failing the connection immediately and notifying any user ExceptionListener objects. Apache ActiveMQ Artemis can be configured to automatically retry the connection, and reattach to the server when it becomes available again across the network.

Replicated Failback example

An example showing how failback works when using replication, In this example a live server will replicate all its Journal to a backup server as it updates it. When the live server crashes the backup takes over from the live server and the client reconnects and carries on from where it left off.

Replicated Failback static example

An example showing how failback works when using replication, but this time with static connectors

Replicated multiple failover example

An example showing how to configure multiple backups when using replication

Replicated Failover transaction example

An example showing how failover works with a transaction when using replication

Request-Reply example

A simple example showing the JMS request-response pattern.

Scheduled Message

The `scheduled-message` example shows you how to send a scheduled message to an address/queue with Apache ActiveMQ Artemis. Scheduled messages won't get delivered until a specified time in the future.

Security

The `security` example shows you how configure and use role based security with Apache ActiveMQ Artemis.

Security LDAP

The `security-ldap` example shows you how configure and use role based security with Apache ActiveMQ Artemis & an embedded instance of the Apache DS LDAP server.

Send Acknowledgements

The `send-acknowledgements` example shows you how to use Apache ActiveMQ Artemis's advanced *asynchronous send acknowledgements* feature to obtain acknowledgement from the server that sends have been received and processed in a separate stream to the sent messages.

Slow Consumer

The `slow-consumer` example shows you how to detect slow consumers and configure a slow consumer policy in Apache ActiveMQ Artemis's

Spring Integration

This example shows how to use embedded JMS using Apache ActiveMQ Artemis's Spring integration.

SSL Transport

The `ssl-enabled` shows you how to configure SSL with Apache ActiveMQ Artemis to send and receive message.

Static Message Selector

The `static-selector` example shows you how to configure an Apache ActiveMQ Artemis core queue with static message selectors (filters).

Static Message Selector Using JMS

The `static-selector-jms` example shows you how to configure an Apache ActiveMQ Artemis queue with static message selectors (filters) using JMS.

Stomp

The `stomp` example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages.

Stomp1.1

The `stomp` example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages via a Stomp 1.1 connection.

Stomp1.2

The `stomp` example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages via a Stomp 1.2 connection.

Stomp Over Web Sockets

The `stomp-websockets` example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages directly from Web browsers (provided they support Web Sockets).

Symmetric Cluster

The `symmetric-cluster` example demonstrates a symmetric cluster set-up with Apache ActiveMQ Artemis.

Apache ActiveMQ Artemis has extremely flexible clustering which allows you to set-up servers in many different topologies. The most common topology that you'll perhaps be familiar with if you are used to application server clustering is a symmetric cluster.

With a symmetric cluster, the cluster is homogeneous, i.e. each node is configured the same as every other node, and every node is connected to every other node in the cluster.

Temporary Queue

A simple example demonstrating how to use a JMS temporary queue.

Topic

A simple example demonstrating a JMS topic.

Topic Hierarchy

Apache ActiveMQ Artemis supports topic hierarchies. With a topic hierarchy you can register a subscriber with a wild-card and that subscriber will receive any messages sent to an address that matches the wild card.

Topic Selector 1

The `topic-selector-example1` example shows you how to send message to a JMS Topic, and subscribe them using selectors with Apache ActiveMQ Artemis.

Topic Selector 2

The `topic-selector-example2` example shows you how to selectively consume messages using message selectors with topic consumers.

Transaction Failover

The `transaction-failover` example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a transacted JMS session failing over from live to backup when the live server is crashed.

Apache ActiveMQ Artemis implements failover of client connections between live and backup servers. This is implemented by the sharing of a journal between the servers. When a live node crashes, the client connections can carry and continue to send and consume messages. When transacted sessions are used, once and only once message delivery is guaranteed.

Failover Without Transactions

The `stop-server-failover` example demonstrates failover of the JMS connection from one node to another when the live server crashes using a JMS non-transacted session.

Transactional Session

The `transactional` example shows you how to use a transactional Session with Apache ActiveMQ Artemis.

XA Heuristic

The `xa-heuristic` example shows you how to make an XA heuristic decision through Apache ActiveMQ Artemis Management Interface. A heuristic decision is a unilateral decision to commit or rollback an XA transaction branch after it has been prepared.

XA Receive

The `xa-receive` example shows you how message receiving behaves in an XA transaction in Apache ActiveMQ Artemis.

XA Send

The `xa-send` example shows you how message sending behaves in an XA transaction in Apache ActiveMQ Artemis.

Routing Messages With Wild Cards

Apache ActiveMQ Artemis allows the routing of messages via wildcard addresses.

If a queue is created with an address of say `queue.news.#` then it will receive any messages sent to addresses that match this, for instance `queue.news.europe` or `queue.news.usa` or `queue.news.usa.sport` . If you create a consumer on this queue, this allows a consumer to consume messages which are sent to a *hierarchy* of addresses.

> **Note:**
>
> In JMS terminology this allows "topic hierarchies" to be created.

This functionality is enabled by default. To turn it off add the following to the `broker.xml` configuration.

```
<wildcard-addresses>
   <routing-enabled>false</routing-enabled>
</wildcard-addresses>
```

For more information on the wild card syntax and how to configure it, take a look at wildcard syntax chapter, also see the topic hierarchy example in the examples.

Wildcard Syntax

Apache ActiveMQ Artemis uses a specific syntax for representing wildcards in security settings, address settings and when creating consumers.

The syntax is similar to that used by AMQP.

An Apache ActiveMQ Artemis wildcard expression contains words delimited by the character '`.`' (full stop).

The special characters '`#`' and '`*`' also have special meaning and can take the place of a word.

The character '`#`' means 'match any sequence of zero or more words'.

The character '`*`' means 'match a single word'.

So the wildcard 'news.europe.#' would match 'news.europe', 'news.europe.sport', 'news.europe.politics', and 'news.europe.politics.regional' but would not match 'news.usa', 'news.usa.sport' nor 'entertainment'.

The wildcard 'news.*' would match 'news.europe', but not 'news.europe.sport'.

The wildcard 'news.*.sport' would match 'news.europe.sport' and also 'news.usa.sport', but not 'news.europe.politics'.

Customizing the Syntax

It's possible to further configure the syntax of the wildcard addresses using the broker configuration. For that, the `<wildcard-addresses>` configuration tag is used.

```
<wildcard-addresses>
   <routing-enabled>true</routing-enabled>
   <delimiter>.</delimiter>
   <any-words>#</any-words>
   <single-word>*</single-word>
</wildcard-addresses>
```

The example above shows the default configuration.

Filter Expressions

Apache ActiveMQ Artemis provides a powerful filter language based on a subset of the SQL 92 expression syntax.

It is the same as the syntax used for JMS selectors, but the predefined identifiers are different. For documentation on JMS selector syntax please the JMS javadoc for javax.jms.Message.

Filter expressions are used in several places in Apache ActiveMQ Artemis

- Predefined Queues. When pre-defining a queue, in `broker.xml` in either the core or jms configuration a filter expression can be defined for a queue. Only messages that match the filter expression will enter the queue.
- Core bridges can be defined with an optional filter expression, only matching messages will be bridged (see Core Bridges).
- Diverts can be defined with an optional filter expression, only matching messages will be diverted (see Diverts).
- Filter are also used programmatically when creating consumers, queues and in several places as described in management.

There are some differences between JMS selector expressions and Apache ActiveMQ Artemis core filter expressions. Whereas JMS selector expressions operate on a JMS message, Apache ActiveMQ Artemis core filter expressions operate on a core message.

The following identifiers can be used in a core filter expressions to refer to attributes of the core message in an expression:

- `AMQPriority`. To refer to the priority of a message. Message priorities are integers with valid values from `0 - 9`. `0` is the lowest priority and `9` is the highest. E.g. `AMQPriority = 3 AND animal = 'aardvark'`
- `AMQExpiration`. To refer to the expiration time of a message. The value is a long integer.
- `AMQDurable`. To refer to whether a message is durable or not. The value is a string with valid values: `DURABLE` or `NON_DURABLE`.
- `AMQTimestamp`. The timestamp of when the message was created. The value is a long integer.
- `AMQSize`. The size of a message in bytes. The value is an integer.

Any other identifiers used in core filter expressions will be assumed to be properties of the message.

The JMS spec states that a String property should not get converted to a numeric when used in a selector. So for example, if a message has the `age` property set to String `21` then the following selector should not match it: `age > 18`. Since Apache ActiveMQ Artemis supports STOMP clients which can only send

messages with string properties, that restriction is a bit limiting. Therefore, if you want your filter expressions to auto-convert String properties to the appropriate number type, just prefix it with `convert_string_expressions:`. If you changed the filter expression in the previous example to be `convert_string_expressions:age > 18`, then it would match the aforementioned message.

The JMS spec also states that property identifiers (and therefore the identifiers which are valid for use in a filter expression) are an, "unlimited-length sequence of letters and digits, the first of which must be a letter. A letter is any character for which the method `Character.isJavaLetter` returns `true`. This includes `_` and `$`. A letter or digit is any character for which the method `Character.isJavaLetterOrDigit` returns `true`." This constraint means that hyphens (i.e. `-`) cannot be used. However, this constraint can be overcome by using the `hyphenated_props:` prefix. For example, if a message had the `foo-bar` property set to `0` then the filter expression `hyphenated_props:foo-bar = 0` would match it.

Persistence

Apache ActiveMQ Artemis ships with two persistence options. The file journal which is highly optimized for the messaging use case and gives great performance, and also the JDBC Store, which uses JDBC to connect to a database of your choice. The JDBC Store is still under development, but it is possible to use its journal features, (essentially everything except for paging and large messages).

File Journal (Default)

The file journal is an *append only* journal. It consists of a set of files on disk. Each file is pre-created to a fixed size and initially filled with padding. As operations are performed on the server, e.g. add message, update message, delete message, records are appended to the journal. When one journal file is full we move to the next one.

Because records are only appended, i.e. added to the end of the journal we minimise disk head movement, i.e. we minimise random access operations which is typically the slowest operation on a disk.

Making the file size configurable means that an optimal size can be chosen, i.e. making each file fit on a disk cylinder. Modern disk topologies are complex and we are not in control over which cylinder(s) the file is mapped onto so this is not an exact science. But by minimising the number of disk cylinders the file is using, we can minimise the amount of disk head movement, since an entire disk cylinder is accessible simply by the disk rotating - the head does not have to move.

As delete records are added to the journal, Apache ActiveMQ Artemis has a sophisticated file garbage collection algorithm which can determine if a particular journal file is needed any more - i.e. has all its data been deleted in the same or other files. If so, the file can be reclaimed and re-used.

Apache ActiveMQ Artemis also has a compaction algorithm which removes dead space from the journal and compresses up the data so it takes up less files on disk.

The journal also fully supports transactional operation if required, supporting both local and XA transactions.

The majority of the journal is written in Java, however we abstract out the interaction with the actual file system to allow different pluggable implementations. Apache ActiveMQ Artemis ships with two implementations:

Java NIO

The first implementation uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform where there's a Java 6+ runtime.

Linux Asynchronous IO

The second implementation uses a thin native code wrapper to talk to the Linux asynchronous IO library (AIO). With AIO, Apache ActiveMQ Artemis will be called back when the data has made it to disk, allowing us to avoid explicit syncs altogether and simply send back confirmation of completion when AIO informs us that the data has been persisted.

Using AIO will typically provide even better performance than using Java NIO.

This journal option is only available when running Linux kernel 2.6 or later and after having installed libaio (if it's not already installed). For instructions on how to install libaio please see Installing AIO section.

Also, please note that AIO will only work with the following file systems: ext2, ext3, ext4, jfs, xfs and NFSV4.

For more information on libaio please see lib AIO.

libaio is part of the kernel project.

Memory mapped

The third implementation uses a file-backed READ_WRITE memory mapping against the OS page cache to interface with the file system.

This provides extremely good performance (especially under strictly process failure durability requirements), almost zero copy (actually *is* the kernel page cache) and zero garbage (from the Java HEAP perspective) operations and runs on any platform where there's a Java 4+ runtime.

Under power failure durability requirements it will perform at least on par with the NIO journal with the only exception of Linux OS with kernel less or equals 2.6, in which the *msync*) implementation necessary to ensure durable writes was different (and slower) from the *fsync* used is case of NIO journal.

It benefits by the configuration of OS huge pages, in particular when is used a big number of journal files and sizing them as multiple of the OS page size in bytes.

Standard Files

The standard Apache ActiveMQ Artemis core server uses two instances of the journal:

- Bindings journal.

  This journal is used to store bindings related data. That includes the set of queues that are deployed on the server and their attributes. It also stores data such as id sequence counters.

  The bindings journal is always a NIO journal as it is typically low throughput compared to the message journal.

The files on this journal are prefixed as `activemq-bindings`. Each file has a `bindings` extension. File size is `1048576`, and it is located at the bindings folder.

- Message journal.

  This journal instance stores all message related data, including the message themselves and also duplicate-id caches.

  By default Apache ActiveMQ Artemis will try and use an AIO journal. If AIO is not available, e.g. the platform is not Linux with the correct kernel version or AIO has not been installed then it will automatically fall back to using Java NIO which is available on any Java platform.

  The files on this journal are prefixed as `activemq-data`. Each file has an `amq` extension. File size is by the default `10485760` (configurable), and it is located at the journal folder.

For large messages, Apache ActiveMQ Artemis persists them outside the message journal. This is discussed in Large Messages.

Apache ActiveMQ Artemis can also be configured to page messages to disk in low memory situations. This is discussed in Paging.

If no persistence is required at all, Apache ActiveMQ Artemis can also be configured not to persist any data at all to storage as discussed in the Configuring the broker for Zero Persistence section.

Configuring the bindings journal

The bindings journal is configured using the following attributes in `broker.xml`

- `bindings-directory`

  This is the directory in which the bindings journal lives. The default value is `data/bindings`.

- `create-bindings-dir`

  If this is set to `true` then the bindings directory will be automatically created at the location specified in `bindings-directory` if it does not already exist. The default value is `true`

Configuring the jms journal

The jms config shares its configuration with the bindings journal.

Configuring the message journal

The message journal is configured using the following attributes in `broker.xml`

- `journal-directory`

  This is the directory in which the message journal lives. The default value is `data/journal`.

For the best performance, we recommend the journal is located on its own physical volume in order to minimise disk head movement. If the journal is on a volume which is shared with other processes which might be writing other files (e.g. bindings journal, database, or transaction coordinator) then the disk head may well be moving rapidly between these files as it writes them, thus drastically reducing performance.

When the message journal is stored on a SAN we recommend each journal instance that is stored on the SAN is given its own LUN (logical unit).

- `node-manager-lock-directory`

  This is the directory in which the node manager file lock lives. By default has the same value of `journal-directory`.

  This is useful when the message journal is on a SAN and is being used a Shared Store HA policy with the broker instances on the same physical machine.

- `create-journal-dir`

  If this is set to `true` then the journal directory will be automatically created at the location specified in `journal-directory` if it does not already exist. The default value is `true`

- `journal-type`

  Valid values are `NIO`, `ASYNCIO` or `MAPPED`.

  Choosing `NIO` chooses the Java NIO journal. Choosing `ASYNCIO` chooses the Linux asynchronous IO journal. If you choose `ASYNCIO` but are not running Linux or you do not have libaio installed then Apache ActiveMQ Artemis will detect this and automatically fall back to using `NIO`. Choosing `MAPPED` chooses the Java Memory Mapped journal.

- `journal-sync-transactional`

  If this is set to true then Apache ActiveMQ Artemis will make sure all transaction data is flushed to disk on transaction boundaries (commit, prepare and rollback). The default value is `true`.

- `journal-sync-non-transactional`

  If this is set to true then Apache ActiveMQ Artemis will make sure non transactional message data (sends and acknowledgements) are flushed to disk each time. The default value for this is `true`.

- `journal-file-size`

  The size of each journal file in bytes. The default value for this is `10485760` bytes (10MiB).

- `journal-min-files`

  The minimum number of files the journal will maintain. When Apache ActiveMQ Artemis starts and there is no initial message data, Apache ActiveMQ Artemis will pre-create `journal-min-files` number of files.

Creating journal files and filling them with padding is a fairly expensive operation and we want to minimise doing this at run-time as files get filled. By pre-creating files, as one is filled the journal can immediately resume with the next one without pausing to create it.

Depending on how much data you expect your queues to contain at steady state you should tune this number of files to match that total amount of data.

- `journal-pool-files`

  The system will create as many files as needed however when reclaiming files it will shrink back to the `journal-pool-files`.

  The default to this parameter is -1, meaning it will never delete files on the journal once created.

  Notice that the system can't grow infinitely as you are still required to use paging for destinations that can grow indefinitely.

  Notice: in case you get too many files you can use compacting.

- `journal-max-io`

  Write requests are queued up before being submitted to the system for execution. This parameter controls the maximum number of write requests that can be in the IO queue at any one time. If the queue becomes full then writes will block until space is freed up.

  When using NIO, this value should always be equal to `1`

  When using ASYNCIO, the default should be `500`.

  The system maintains different defaults for this parameter depending on whether it's NIO or ASYNCIO (default for NIO is 1, default for ASYNCIO is 500)

  There is a limit and the total max ASYNCIO can't be higher than what is configured at the OS level (/proc/sys/fs/aio-max-nr) usually at 65536.

- `journal-buffer-timeout`

  Instead of flushing on every write that requires a flush, we maintain an internal buffer, and flush the entire buffer either when it is full, or when a timeout expires, whichever is sooner. This is used for both NIO and ASYNCIO and allows the system to scale better with many concurrent writes that require flushing.

  This parameter controls the timeout at which the buffer will be flushed if it hasn't filled already. ASYNCIO can typically cope with a higher flush rate than NIO, so the system maintains different defaults for both NIO and ASYNCIO (default for NIO is 3333333 nanoseconds - 300 times per second, default for ASYNCIO is 500000 nanoseconds - ie. 2000 times per second).

  Setting this property to 0 will disable the internal buffer and writes will be directly written to the journal file immediately.

> **Note:**
>
> By increasing the timeout, you may be able to increase system throughput at the expense of latency, the default parameters are chosen to give a reasonable balance between throughput and latency.

- `journal-buffer-size`

  The size of the timed buffer on ASYNCIO. The default value is `490KiB`.

- `journal-compact-min-files`

  The minimal number of files before we can consider compacting the journal. The compacting algorithm won't start until you have at least `journal-compact-min-files`

  Setting this to 0 will disable the feature to compact completely. This could be dangerous though as the journal could grow indefinitely. Use it wisely!

  The default for this parameter is `10`

- `journal-compact-percentage`

  The threshold to start compacting. When less than this percentage is considered live data, we start compacting. Note also that compacting won't kick in until you have at least `journal-compact-min-files` data files on the journal

  The default for this parameter is `30`

- `journal-datasync` (default: true)

  This will disable the use of fdatasync on journal writes. When enabled it ensures full power failure durability, otherwise process failure durability on journal writes (OS guaranteed). This is particular effective for `NIO` and `MAPPED` journals, which rely on *fsync*/*msync* to force write changes to disk.

Note on disabling `journal-datasync`

> Any modern OS guarantees that on process failures (i.e. crash) all the uncommitted changes to the page cache will be flushed to the file system, maintaining coherence between subsequent operations against the same pages and ensuring that no data will be lost. The predictability of the timing of such flushes, in case of a disabled *journal-datasync*, depends on the OS configuration, but without compromising (or relaxing) the process failure durability semantics as described above. Rely on the OS page cache sacrifice the power failure protection, while increasing the effectiveness of the journal operations, capable of exploiting the read caching and write combining features provided by the OS's kernel page cache subsystem.

Note on disabling disk write cache

> **Warning**
>
> Most disks contain hardware write caches. A write cache can increase the apparent performance of the disk because writes just go into the cache and are then lazily written to the disk later.
>
> This happens irrespective of whether you have executed a fsync() from the operating system or correctly synced data from inside a Java program!
>
> By default many systems ship with disk write cache enabled. This means that even after syncing from the operating system there is no guarantee the data has actually made it to disk, so if a failure occurs, critical data can be lost.
>
> Some more expensive disks have non volatile or battery backed write caches which won't necessarily lose data on event of failure, but you need to test them!
>
> If your disk does not have an expensive non volatile or battery backed cache and it's not part of some kind of redundant array (e.g. RAID), and you value your data integrity you need to make sure disk write cache is disabled.
>
> Be aware that disabling disk write cache can give you a nasty shock performance wise. If you've been used to using disks with write cache enabled in their default setting, unaware that your data integrity could be compromised, then disabling it will give you an idea of how fast your disk can perform when acting really reliably.
>
> On Linux you can inspect and/or change your disk's write cache settings using the tools `hdparm` (for IDE disks) or `sdparm` or `sginfo` (for SDSI/SATA disks)
>
> On Windows you can check / change the setting by right clicking on the disk and clicking properties.

Installing AIO

The Java NIO journal gives great performance, but If you are running Apache ActiveMQ Artemis using Linux Kernel 2.6 or later, we highly recommend you use the `ASYNCIO` journal for the very best persistence performance.

It's not possible to use the ASYNCIO journal under other operating systems or earlier versions of the Linux kernel.

If you are running Linux kernel 2.6 or later and don't already have `libaio` installed, you can easily install it using the following steps:

Using yum, (e.g. on Fedora or Red Hat Enterprise Linux):

```
yum install libaio
```

Using aptitude, (e.g. on Ubuntu or Debian system):

```
apt-get install libaio
```

JDBC Persistence

WARNING: The Apache ActiveMQ Artemis JDBC persistence store is under development and is included for evaluation purposes.

The Apache ActiveMQ Artemis JDBC persistence layer offers the ability to store broker state (Messages, Addresses and other application state) using a database. N.B. Address full policy Paging (See: The section on Paging) is currently not supported with the JDBC persistence layer.

Using the ActiveMQ Artemis File Journal is the recommended configuration as it offers higher levels of performance and is more mature. The JDBC persistence layer is targeted to those users who must use a database e.g. due to internal company policy.

ActiveMQ Artemis currently has support for a limited number of database vendors (older versions may work but mileage may vary):

1. PostgreSQL 9.4.x
2. MySQL 5.7.x
3. Apache Derby 10.11.1.1

The JDBC store uses a JDBC connection to store messages and bindings data in records in database tables. The data stored in the database tables is encoded using Apache ActiveMQ Artemis internal encodings.

Configuring JDBC Persistence

To configure Apache ActiveMQ Artemis to use a database for persisting messages and bindings data you must do two things.

1. See the documentation on adding runtime dependencies to understand how to make the JDBC driver available to the broker.
2. Create a store element in your broker.xml config file under the `<core>` element. For example:

```
<store>
   <database-store>
      <jdbc-connection-url>jdbc:derby:data/derby/database-store;create=true</j
      <bindings-table-name>BINDINGS_TABLE</bindings-table-name>
      <message-table-name>MESSAGE_TABLE</message-table-name>
      <page-store-table-name>MESSAGE_TABLE</page-store-table-name>
      <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-table-name
      <node-manager-store-table-name>NODE_MANAGER_TABLE</node-manager-store-ta
      <jdbc-driver-class-name>org.apache.derby.jdbc.EmbeddedDriver</jdbc-drive
   </database-store>
</store>
```

- `jdbc-connection-url`

The full JDBC connection URL for your database server. The connection url should include all configuration parameters and database name. **Note:** When configuring the server using the XML configuration files please ensure to escape any illegal chars; "&" for example, is typical in JDBC connection url and should be escaped to "&".

- `bindings-table-name`

  The name of the table in which bindings data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference.

- `message-table-name`

  The name of the table in which bindings data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference.

- `large-message-table-name`

  The name of the table in which messages and related data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference.

- `page-store-table-name`

  The name of the table to house the page store directory information. Note that each address will have its own page table which will use this name appended with a unique id of up to 20 characters.

- `node-manager-store-table-name`

  The name of the table in which the HA Shared Store locks (ie live and backup) and HA related data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference. Each Shared Store live/backup pairs must use the same table name and isn't supported to share the same table between multiple (and unrelated) live/backup pairs.

- `jdbc-driver-class-name`

  The fully qualified class name of the desired database Driver.

- `jdbc-network-timeout`

  The JDBC network connection timeout in milliseconds. The default value is 20000 milliseconds (ie 20 seconds). When using a shared store it is recommended to set it less then or equal to `jdbc-lock-expiration`.

- `jdbc-lock-renew-period`

  The period in milliseconds of the keep alive service of a JDBC lock. The default value is 2000 milliseconds (ie 2 seconds).

- `jdbc-lock-expiration`

The time in milliseconds a JDBC lock is considered valid without keeping it alive. The default value is 20000 milliseconds (ie 20 seconds).

- `jdbc-journal-sync-period`

  The time in milliseconds the journal will be synced with JDBC. The default value is 5 milliseconds.

Note that some DBMS (e.g. Oracle, 30 chars) have restrictions on the size of table names, this should be taken into consideration when configuring table names for the Artemis database store, pay particular attention to the page store table name, which can be appended with a unique ID of up to 20 characters. (for Oracle this would mean configuring a page-store-table-name of max size of 10 chars).

It is also possible to explicitly add the user and password rather than in the JDBC url if you need to encode it, this would look like:

```
<store>
   <database-store>
      <jdbc-connection-url>jdbc:derby:data/derby/database-store;create=true</j
      <jdbc-user>ENC(dasfn353cewc)</jdbc-user>
      <jdbc-password>ENC(ucwiurfjtew345)</jdbc-password>
      <bindings-table-name>BINDINGS_TABLE</bindings-table-name>
      <message-table-name>MESSAGE_TABLE</message-table-name>
      <page-store-table-name>MESSAGE_TABLE</page-store-table-name>
      <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-table-name
      <node-manager-store-table-name>NODE_MANAGER_TABLE</node-manager-store-ta
      <jdbc-driver-class-name>org.apache.derby.jdbc.EmbeddedDriver</jdbc-drive
   </database-store>
</store>
```

Zero Persistence

In some situations, zero persistence is sometimes required for a messaging system. Configuring Apache ActiveMQ Artemis to perform zero persistence is straightforward. Simply set the parameter `persistence-enabled` in `broker.xml` to `false`.

Please note that if you set this parameter to false, then *zero* persistence will occur. That means no bindings data, message data, large message data, duplicate id caches or paging data will be persisted.

Configuring the Transport

In this chapter we'll describe the concepts required for understanding Apache ActiveMQ Artemis transports and where and how they're configured.

Acceptors

One of the most important concepts in Apache ActiveMQ Artemis transports is the *acceptor*. Let's dive straight in and take a look at an acceptor defined in xml in the configuration file `broker.xml`.

```
<acceptor name="netty">tcp://localhost:61617</acceptor>
```

Acceptors are always defined inside an `acceptors` element. There can be one or more acceptors defined in the `acceptors` element. There's no upper limit to the number of acceptors per server.

Each acceptor defines a way in which connections can be made to the Apache ActiveMQ Artemis server.

In the above example we're defining an acceptor that uses Netty to listen for connections at port `61617`.

The `acceptor` element contains a `URL` that defines the kind of Acceptor to create along with its configuration. The `schema` part of the `URL` defines the Acceptor type which can either be `tcp` or `vm` which is `Netty` or an In VM Acceptor respectively. For `Netty` the host and the port of the `URL` define what host and port the `acceptor` will bind to. For In VM the `Authority` part of the `URL` defines a unique server id.

The `acceptor` can also be configured with a set of key=value pairs used to configure the specific transport, the set of valid key=value pairs depends on the specific transport be used and are passed straight through to the underlying transport. These are set on the `URL` as part of the query, like so:

```
<acceptor name="netty">tcp://localhost:61617?sslEnabled=true&keyStorePath=/pat
```

Connectors

Whereas acceptors are used on the server to define how we accept connections, connectors are used to define how to connect to a server.

Let's look at a connector defined in our `broker.xml` file:

```
<connector name="netty">tcp://localhost:61617</connector>
```

Connectors can be defined inside a `connectors` element. There can be one or more connectors defined in the `connectors` element. There's no upper limit to the number of connectors per server.

A `connector` is used when the server acts as a client itself, e.g.:

- When one server is bridged to another
- When a server takes part in a cluster

In these cases the server needs to know how to connect to other servers. That's defined by `connectors` .

Configuring the Transport Directly from the Client

How do we configure a core `ClientSessionFactory` with the information that it needs to connect with a server?

Connectors are also used indirectly when configuring a core `ClientSessionFactory` to directly talk to a server. Although in this case there's no need to define such a connector in the server side configuration, instead we just specify the appropriate URI.

Here's an example of creating a `ClientSessionFactory` which will connect directly to the acceptor we defined earlier in this chapter, it uses the standard Netty TCP transport and will try and connect on port 61617 to localhost (default):

```
ServerLocator locator = ActiveMQClient.createServerLocator("tcp://localhost:61

ClientSessionFactory sessionFactory = locator.createClientSessionFactory();

ClientSession session = sessionFactory.createSession(...);
```

Similarly, if you're using JMS, you can configure the JMS connection factory directly on the client side:

```
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://loc

Connection jmsConnection = connectionFactory.createConnection();
```

Configuring the Netty transport

Out of the box, Apache ActiveMQ Artemis currently uses Netty, a high performance low level network library.

Our Netty transport can be configured in several different ways; to use straightforward TCP sockets, SSL, or to tunnel over HTTP or HTTPS..

We believe this caters for the vast majority of transport requirements.

Single Port Support

Apache ActiveMQ Artemis supports using a single port for all protocols, Apache ActiveMQ Artemis will automatically detect which protocol is being used CORE, AMQP, STOMP, MQTT or OPENWIRE and use the appropriate Apache ActiveMQ Artemis handler. It will also detect whether protocols such as HTTP or Web Sockets are being used and also use the appropriate decoders. Web Sockets are supported for AMQP, STOMP, and MQTT.

It is possible to limit which protocols are supported by using the `protocols` parameter on the Acceptor like so:

```
<acceptor name="netty">tcp://localhost:61617?protocols=CORE,AMQP</acceptor>
```

Configuring Netty TCP

Netty TCP is a simple unencrypted TCP sockets based transport. If you're running connections across an untrusted network please bear in mind this transport is unencrypted. You may want to look at the SSL or HTTPS configurations.

With the Netty TCP transport all connections are initiated from the client side (i.e. the server does not initiate any connections to the client). This works well with firewall policies that typically only allow connections to be initiated in one direction.

All the valid keys for the `tcp` URL scheme used for Netty are defined in the class `org.apache.activemq.artemis.core.remoting.impl.netty.TransportConstants`. Most parameters can be used either with acceptors or connectors, some only work with acceptors. The following parameters can be used to configure Netty for simple TCP:

> **Note:**
>
> The `host` and `port` parameters are only used in the core API, in XML configuration these are set in the URI host and port.

- `host`. This specifies the host name or IP address to connect to (when configuring a connector) or to listen on (when configuring an acceptor). The default value for this property is `localhost`. When configuring acceptors, multiple hosts or IP addresses can be specified by separating them with commas. It is also possible to specify `0.0.0.0` to accept connection from all the host's network interfaces. It's not valid to specify multiple addresses when specifying the host for a connector; a connector makes a connection to one specific address.

  > **Note:**
  >
  > Don't forget to specify a host name or IP address! If you want your server able to accept connections from other nodes you must specify a hostname or IP address at which the acceptor will bind and listen for incoming connections. The default is localhost which of course is not accessible from remote nodes!

- `port`. This specified the port to connect to (when configuring a connector) or to listen on (when configuring an acceptor). The default value for this property is `61616`.
- `tcpNoDelay`. If this is `true` then Nagle's algorithm will be disabled. This is a Java (client) socket option. The default value for this property is `true`.
- `tcpSendBufferSize`. This parameter determines the size of the TCP send buffer in bytes. The default value for this property is `32768` bytes (32KiB).

  TCP buffer sizes should be tuned according to the bandwidth and latency of your network. Here's a good link that explains the theory behind this.

  In summary TCP send/receive buffer sizes should be calculated as:

  ```
  buffer_size = bandwidth * RTT.
  ```

  Where bandwidth is in *bytes per second* and network round trip time (RTT) is in seconds. RTT can be easily measured using the `ping` utility.

  For fast networks you may want to increase the buffer sizes from the defaults.
- `tcpReceiveBufferSize`. This parameter determines the size of the TCP receive buffer in bytes. The default value for this property is `32768` bytes (32KiB).
- `writeBufferLowWaterMark`. This parameter determines the low water mark of the Netty write buffer. Once the number of bytes queued in the write buffer exceeded the high water mark and then dropped down below this value, Netty's channel will start to be writable again. The default value for this property is `32768` bytes (32KiB).
- `writeBufferHighWaterMark`. This parameter determines the high water mark of the Netty write buffer. If the number of bytes queued in the write buffer exceeds this value, Netty's channel will start to be not writable. The default value for this property is `131072` bytes (128KiB).
- `batchDelay`. Before writing packets to the transport, Apache ActiveMQ Artemis can be configured to batch up writes for a maximum of `batchDelay` milliseconds. This can increase overall throughput for very small messages. It does so at the expense of an increase in average latency for message transfer. The default value for this property is `0` ms.
- `directDeliver`. When a message arrives on the server and is delivered to waiting consumers, by default, the delivery is done on the same thread as that on which the message arrived. This gives good latency in environments with relatively small messages and a small number of consumers, but at the cost of overall throughput and scalability - especially on multi-core machines. If you want the lowest latency and a possible reduction in throughput then you can use the default value for `directDeliver` (i.e. `true`). If you are willing to take some small extra hit on latency but want the highest throughput set `directDeliver` to `false`.

- `nioRemotingThreads` This is deprecated. It is replaced by `remotingThreads`, if you are using this please update your configuration
- `remotingThreads`. Apache ActiveMQ Artemis will, by default, use a number of threads equal to three times the number of cores (or hyper-threads) as reported by `Runtime.getRuntime().availableProcessors()` for processing incoming packets. If you want to override this value, you can set the number of threads by specifying this parameter. The default value for this parameter is `-1` which means use the value from `Runtime.getRuntime().availableProcessors()` * 3.
- `localAddress`. When configured a Netty Connector it is possible to specify which local address the client will use when connecting to the remote address. This is typically used in the Application Server or when running Embedded to control which address is used for outbound connections. If the local-address is not set then the connector will use any local address available
- `localPort`. When configured a Netty Connector it is possible to specify which local port the client will use when connecting to the remote address. This is typically used in the Application Server or when running Embedded to control which port is used for outbound connections. If the local-port default is used, which is 0, then the connector will let the system pick up an ephemeral port. valid ports are 0 to 65535
- `connectionsAllowed`. This is only valid for acceptors. It limits the number of connections which the acceptor will allow. When this limit is reached a DEBUG level message is issued to the log, and the connection is refused. The type of client in use will determine what happens when the connection is refused. In the case of a `core` client, it will result in a `org.apache.activemq.artemis.api.core.ActiveMQConnectionTimedOutException`.
- `handshake-timeout`. Prevents an unauthorised client opening a large number of connections and just keeping them open. As connections each require a file handle this consumes resources that are then unavailable to other clients. Once the connection is authenticated, the usual rules can be enforced regarding resource consumption. Default value is set to 10 seconds. Each integer is valid value. When set value to zero or negative integer this feature is turned off. Changing value needs to restart server to take effect.

Configuring Netty Native Transport

Netty Native Transport support exists for selected OS platforms. This allows Apache ActiveMQ Artemis to use native sockets/io instead of Java NIO.

These Native transports add features specific to a particular platform, generate less garbage, and generally improve performance when compared to Java NIO based transport.

Both Clients and Server can benefit from this.

Current Supported Platforms.

- Linux running 64bit JVM
- MacOS running 64bit JVM

Apache ActiveMQ Artemis will by default enable the corresponding native transport if a supported platform is detected.

If running on an unsupported platform or any issues loading native libs, Apache ActiveMQ Artemis will fallback onto Java NIO.

Linux Native Transport

On supported Linux platforms Epoll is used, @see https://en.wikipedia.org/wiki/Epoll.

The following properties are specific to this native transport:

- `useEpoll` enables the use of epoll if a supported linux platform is running a 64bit JVM is detected. Setting this to `false` will force the use of Java NIO instead of epoll. Default is `true`

MacOS Native Transport

On supported MacOS platforms KQueue is used, @see https://en.wikipedia.org/wiki/Kqueue.

The following properties are specific to this native transport:

- `useKQueue` enables the use of kqueue if a supported MacOS platform running a 64bit JVM is detected. Setting this to `false` will force the use of Java NIO instead of kqueue. Default is `true`

Configuring Netty SSL

Netty SSL is similar to the Netty TCP transport but it provides additional security by encrypting TCP connections using the Secure Sockets Layer SSL

Please see the examples for a full working example of using Netty SSL.

Netty SSL uses all the same properties as Netty TCP but adds the following additional properties:

- `sslContext`

A key that can be used in conjunction with `org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory` to cache created SSLContext and avoid recreating. Look Configuring a SSLContextFactory for more details.

- `sslEnabled`

  Must be `true` to enable SSL. Default is `false`.

- `keyStorePath`

When used on an `acceptor` this is the path to the SSL key store on the server which holds the server's certificates (whether self-signed or signed by an authority).

When used on a `connector` this is the path to the client-side SSL key store which holds the client certificates. This is only relevant for a `connector` if you are using 2-way SSL (i.e. mutual authentication). Although this value is configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.keyStore" system property or the ActiveMQ-specific "org.apache.activemq.ssl.keyStore" system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

- `keyStorePassword`

  When used on an `acceptor` this is the password for the server-side keystore.

  When used on a `connector` this is the password for the client-side keystore. This is only relevant for a `connector` if you are using 2-way SSL (i.e. mutual authentication). Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.keyStorePassword" system property or the ActiveMQ-specific "org.apache.activemq.ssl.keyStorePassword" system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

- `trustStorePath`

  When used on an `acceptor` this is the path to the server-side SSL key store that holds the keys of all the clients that the server trusts. This is only relevant for an `acceptor` if you are using 2-way SSL (i.e. mutual authentication).

  When used on a `connector` this is the path to the client-side SSL key store which holds the public keys of all the servers that the client trusts. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.trustStore" system property or the ActiveMQ-specific "org.apache.activemq.ssl.trustStore" system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

- `trustStorePassword`

  When used on an `acceptor` this is the password for the server-side trust store. This is only relevant for an `acceptor` if you are using 2-way SSL (i.e. mutual authentication).

When used on a `connector` this is the password for the client-side truststore. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.trustStorePassword" system property or the ActiveMQ-specific "org.apache.activemq.ssl.trustStorePassword" system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

- `enabledCipherSuites`

  Whether used on an `acceptor` or `connector` this is a comma separated list of cipher suites used for SSL communication. The default value is `null` which means the JVM's default will be used.

- `enabledProtocols`

  Whether used on an `acceptor` or `connector` this is a comma separated list of protocols used for SSL communication. The default value is `null` which means the JVM's default will be used.

- `needClientAuth`

  This property is only for an `acceptor`. It tells a client connecting to this acceptor that 2-way SSL is required. Valid values are `true` or `false`. Default is `false`.

  Note: This property takes precedence over `wantClientAuth` and if its value is set to true then `wantClientAuth` will be ignored.

- `wantClientAuth`

  This property is only for an `acceptor`. It tells a client connecting to this acceptor that 2-way SSL is requested but not required. Valid values are `true` or `false`. Default is `false`.

  Note: If the property `needClientAuth` is set to `true` then that property will take precedence and this property will be ignored.

- `verifyHost`

  When used on an `acceptor` the `CN` of the connecting client's SSL certificate will be compared to its hostname to verify they match. This is useful only for 2-way SSL.

  When used on a `connector` the `CN` of the server's SSL certificate will be compared to its hostname to verify they match. This is useful for both 1-way and 2-way SSL.

  Valid values are `true` or `false`. Default is `false`.

- `trustAll`

When used on a `connector` the client will trust the provided server certificate implicitly, regardless of any configured trust store. **Warning:** This setting is primarily for testing purposes only and should not be used in production.

Valid values are `true` or `false`. Default is `false`.

- `forceSSLParameters`

  When used on a `connector` any SSL settings that are set as parameters on the connector will be used instead of JVM system properties including both javax.net.ssl and ActiveMQ system properties to configure the SSL context for this connector.

  Valid values are `true` or `false`. Default is `false`.

- `useDefaultSslContext`

  Only valid on a `connector`. Allows the `connector` to use the "default" SSL context (via `SSLContext.getDefault()`) which can be set programmatically by the client (via `SSLContext.setDefault(SSLContext)`). If set to `true` all other SSL related parameters except for `sslEnabled` are ignored.

  Valid values are `true` or `false`. Default is `false`.

- `sslProvider`

  Used to change the SSL Provider between `JDK` and `OPENSSL`. The default is `JDK`. If used with `OPENSSL` you can add `netty-tcnative` to your classpath to use the native installed openssl. This can be useful if you want to use special ciphersuite - elliptic curve combinations which are support through openssl but not through the JDK provider. See https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations for more information's.

- `sniHost`

  When used on an `acceptor` the `sniHost` is a *regular expression* used to match the `server_name` extension on incoming SSL connections. If the name doesn't match then the connection to the acceptor will be rejected. A WARN message will be logged if this happens. If the incoming connection doesn't include the `server_name` extension then the connection will be accepted.

  When used on a `connector` the `sniHost` value is used for the `server_name` extension on the SSL connection.

- `trustManagerFactoryPlugin`

  This is valid on either an `acceptor` or `connector`. It defines the name of the class which implements `org.apache.activemq.artemis.api.core.TrustManagerFactoryPlugin`. This is a simple interface with a single method which returns a `javax.net.ssl.TrustManagerFactory`. The `TrustManagerFactory` will be used when the underlying `javax.net.ssl.SSLContext` is initialized. This allows fine-grained customization of who/what the broker & client trusts.

This value takes precedence of all other SSL parameters which apply to the trust manager (i.e. `trustAll`, `truststoreProvider`, `truststorePath`, `truststorePassword`, `crlPath`).

Any plugin specified will need to be placed on the broker's classpath.

Configuring a SSLContextFactory

If you have a `JDK` provider you can configure which SSLContextFactory to use. Currently we provide two implementations:

- `org.apache.activemq.artemis.core.remoting.impl.ssl.DefaultSSLContextFactory`
- `org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory` but you can also add your own implementation of `org.apache.activemq.artemis.spi.core.remoting.ssl.SSLContextFactory`.

The implementations are loaded by a ServiceLoader, thus you need to declare your implementation in a `META-INF/services/org.apache.activemq.artemis.spi.core.remoting.ssl.SSLContextFactory` file. If several implementations are available, the one with the highest `priority` will be selected. So for example, if you want to use `org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory` you need to add a `META-INF/services/org.apache.activemq.artemis.spi.core.remoting.ssl.SSLContextFactory` file to your classpath with the content `org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory`.

Note: This mechanism doesn't work if you have selected `OPENSSL` as provider.

Configuring Netty HTTP

Netty HTTP tunnels packets over the HTTP protocol. It can be useful in scenarios where firewalls only allow HTTP traffic to pass.

Please see the examples for a full working example of using Netty HTTP.

Netty HTTP uses the same properties as Netty TCP but adds the following additional properties:

- `httpEnabled`. This is now no longer needed. With single port support Apache ActiveMQ Artemis will now automatically detect if http is being used and configure itself.
- `httpClientIdleTime`. How long a client can be idle before sending an empty http request to keep the connection alive
- `httpClientIdleScanPeriod`. How often, in milliseconds, to scan for idle clients
- `httpResponseTime`. How long the server can wait before sending an empty http response to keep the connection alive
- `httpServerScanPeriod`. How often, in milliseconds, to scan for clients needing responses

- `httpRequiresSessionId`. If `true` the client will wait after the first call to receive a session id. Used the http connector is connecting to servlet acceptor (not recommended)

Configuration Reload

The system will perform a periodic check on the configuration files, configured by `configuration-file-refresh-period`, with the default at 5000, in milliseconds.

Once the configuration file is changed (broker.xml) the following modules will be reloaded automatically:

- Address Settings
- Security Settings
- Diverts
- Addresses & queues

If using modulised broker.xml ensure you also read Reloading modular configuration files

Note:

Deletion of Address's and Queue's, not auto created is controlled by Address Settings

- config-delete-addresses
  - OFF (DEFAULT) - will not remove upon config reload.
  - FORCE - will remove the address and its queues upon config reload, even if messages remains, losing the messages in the address & queues.
- config-delete-queues
  - OFF (DEFAULT) - will not remove upon config reload.
  - FORCE - will remove the queue upon config reload, even if messages remains, losing the messages in the queue.

By default both settings are OFF as such address & queues won't be removed upon reload, given the risk of losing messages.

When OFF You may execute explicit CLI or Management operations to remove address & queues.

Reloadable Parameters

The broker configuration file has 2 main parts, `<core>` and `<jms>`. Some of the parameters in the 2 parts are monitored and, if modified, reloaded into the broker at runtime.

Note: Elements under `<jms>` are **deprecated**. Users are encouraged to use `<core>` configuration entities.

> **Note:**
>
> Most parameters reloaded take effect immediately after reloading. However there are some that won't take any effect unless you restarting the broker. Such parameters are specifically indicated in the following text.

`<core>`

`<security-settings>`

- `<security-setting>` element

Changes to any `<security-setting>` elements will be reloaded. Each `<security-setting>` defines security roles for a matched address.

- The `match` attribute

  This attribute defines the address for which the security-setting is defined. It can take wildcards such as '#' and '*'.

- The `<permission>` sub-elements

Each `<security-setting>` can have a list of `<permission>` elements, each of which defines a specific permission-roles mapping. Each permission has 2 attributes 'type' and 'roles'. The 'type' attribute defines the type of operation allowed, the 'roles' defines which roles are allowed to perform such operation. Refer to the user's manual for a list of operations that can be defined.

> **Note:**
>
> Once loaded the security-settings will take effect immediately. Any new clients will subject to the new security settings. Any existing clients will subject to the new settings as well, as soon as they performs a new security-sensitive operation.

Below lists the effects of adding, deleting and updating of an element/attribute within the `<security-settings>` element, whether a change can be done or can't be done.

Operation	Add	Delete	Update
`<securitv-settings>`	X* (at most one element is allowed)	Deleting it means delete the whole security settings from the running broker.	N/A*
`<security-setting>`	Adding one element means adding a new set of security roles for an address in the running broker	Deleting one element means removing a set of security roles for an address in the running broker	Updating one element means updating the security roles for an address (if match attribute is not changed), or means removing the old match address settings and adding a new one (if match attribute is changed)
attribute `match`	N/A*	X*	Changing this value is same as deleting the whole with the old match value and adding
`<permission>`	Adding one means adding a new permission definition to runtime broker	Deleting a permission from the runtime broker	Updating a permission-roles in the runtime broker
attribute `type`	N/A*	X*	Changing the type value means remove the permission of the old one and add the permission of this type to the running broker.
attribute `roles`	N/A*	X*	Changing the ‘roles’ value means updating the permission’s allowed roles to the running broker

> - `N/A` means this operation is not applicable.
> - `X` means this operation is not allowed.

`<address-settings>`

- `<address-settings>` element

Changes to elements under `<address-settings>` will be reloaded into runtime broker. It contains a list of `<address-setting>` elements.

- `<address-setting>` element

  Each address-setting element has a ‘match’ attribute that defines an address pattern for which this address-setting is defined. It also has a list of sub-elements used to define the properties of a matching address.

  > **Note:**
  >
  > Parameters reloaded in this category will take effect immediately after reloading. The effect of deletion of Address's and Queue's, not auto created is controlled by parameter `config-delete-addresses` and `config-delete-queues` as described in the doc.

Below lists the effects of adding, deleting and updating of an element/attribute within the address-settings element, whether a change can be done or can’t be done.

Operation	Add	Delete	Update
`<address-settings>`	X(at most one element is allowed)	Deleting it means delete the whole address settings from the running broker	N/A
`<address-setting>`	Adding one element means adding a set of address-setting for a new address in the running broker	Deleting one means removing a set of address-setting for an address in the running broker	Updating one element means updating the address setting for an address (if match attribute is not changed), or means removing the old match address settings and adding a new one (if match attribute is changed)
attribute `match`	N/A	X	Changing this value is same as deleting the whole with the old match value and adding a new one with the new match value.
`<dead-letter-address>`	X (no more than one can be present)	Removing the configured dead-letter-address address from running broker.	The dead letter address of the matching address will be updated after reloading
`<expiry-address>`	X (no more than one can be present)	Removing the configured expiry address from running broker.	The expiry address of the matching address will be updated after reloading
`<expiry-delay>`	X (no more than one can be present)	The configured expiry-delay will be removed from running broker.	The expiry-delay for the matching address will be updated after reloading.
`<redelivery-delay>`	X (no more than one can be present)	The configured redelivery-delay will be removed from running broker after reloading	The redelivery-delay for the matchin address will be updated after reloading.

Operation	Add	Delete	Update
`<redelivery-delay-multiplier>`	X (no more than one can be present)	The configured redelivery-delay-multiplier will be removed from running broker after reloading.	The redelivery-delay-multiplier will be updated after reloading.
`<max-redelivery-delay>`	X (no more than one can be present)	The configured max-redelivery-delay will be removed from running broker after reloading.	The max-redelivery-delay will be updated after reloading.
`<max-delivery-attempts>`	X (no more than one can be present)	The configured max-delivery-attempts will be removed from running broker after reloading.	The max-delivery-attempts will be updated after reloading.
`<max-size-bytes>`	X (no more than one can be present)	The configured max-size-bytes will be removed from running broker after reloading.	The max-size-bytes will be updated after reloading.
`<page-size-bytes>`	X (no more than one can be present)	The configured page-size-bytes will be removed from running broker after reloading.	The page-size-bytes will be updated after reloading.
`<page-max-cache-size>`	X (no more than one can be present)	The configured page-max-cache-size will be removed from running broker after reloading.	The page-max-cache-size will be updated after reloading.

Operation	Add	Delete	Update
`<address-full-policy>`	X (no more than one can be present)	The configured address-full-policy will be removed from running broker after reloading.	The address-full-policy will be updated after reloading.
`<message-counter-history-day-limit>`	X (no more than one can be present)	The configured message-counter-history-day-limit will be removed from running broker after reloading.	The message-counter-history-day-limit will be updated after reloading.
`<last-value-queue>`	X (no more than one can be present)	The configured last-value-queue will be removed from running broker after reloading (no longer a last value queue).	The last-value-queue will be updated after reloading.
`<redistribution-delay>`	X (no more than one can be present)	The configured redistribution-delay will be removed from running broker after reloading.	The redistribution-delay will be updated after reloading.
`<send-to-dla-on-no-route>`	X (no more than one can be present)	The configured send-to-dla-on-no-route will be removed from running broker after reloading.	The send-to-dla-on-no-route will be updated after reloading.
`<slow-consumer-threshold>`	X (no more than one can be present)	The configured slow-consumer-threshold will be removed from running broker after reloading.	The slow-consumer-threshold will be updated after reloading.

Operation	Add	Delete	Update
`<slow-consumer-policy>`	X (no more than one can be present)	The configured slow-consumer-policy will be removed from running broker after reloading.	The slow-consumer-policy will be updated after reloading.
`<slow-consumer-check-period>`	X (no more than one can be present)	The configured slow-consumer-check-period will be removed from running broker after reloading. (meaning the slow consumer checker thread will be cancelled)	The slow-consumer-check-period will be updated after reloading.
`<auto-create-queues>`	X (no more than one can be present)	The configured auto-create-queues will be removed from running broker after reloading.	The auto-create-queues will be updated after reloading.
`<auto-delete-queues>`	X (no more than one can be present)	The configured auto-delete-queues will be removed from running broker after reloading.	The auto-delete-queues will be updated after reloading.
`<config-delete-queues>`	X (no more than one can be present)	The configured config-delete-queues will be removed from running broker after reloading.	The config-delete-queues will be updated after reloading.
`<auto-create-addresses>`	X (no more than one can be present)	The configured auto-create-addresses will be removed from running broker after reloading.	The auto-create-addresses will be updated after reloading.

Operation	Add	Delete	Update
`<auto-delete-addresses>`	X (no more than one can be present)	The configured auto-delete-addresses will be removed from running broker after reloading.	The auto-delete-addresses will be updated after reloading.
`<config-delete-addresses>`	X (no more than one can be present)	The configured config-delete-addresses will be removed from running broker after reloading.	The config-delete-addresses will be updated after reloading.
`<management-browse-page-size>`	X (no more than one can be present)	The configured management-browse-page-size will be removed from running broker after reloading.	The management-browse-page-size will be updated after reloading.
`<default-purge-on-no-consumers>`	X (no more than one can be present)	The configured default-purge-on-no-consumers will be removed from running broker after reloading.	The default-purge-on-no-consumers will be updated after reloading.
`<default-max-consumers>`	X (no more than one can be present)	The configured default-max-consumers will be removed from running broker after reloading.	The default-max-consumers will be updated after reloading.
`<default-queue-routing-type>`	X (no more than one can be present)	The configured default-queue-routing-type will be removed from running broker after reloading.	The default-queue-routing-type will be updated after reloading.

Operation	Add	Delete	Update
`<default-address-routing-type>`	X (no more than one can be present)	The configured default-address-routing-type will be removed from running broker after reloading.	The default-address-routing-type will be updated after reloading.

`<diverts>`

All `<divert>` elements will be reloaded. Each `<divert>` element has a ‘name’ and several sub-elements that defines the properties of a divert.

> **Note:**
>
> Existing diverts get undeployed if you delete their `<divert>` element.

Below lists the effects of adding, deleting and updating of an element/attribute within the diverts element, whether a change can be done or can’t be done.

Operation	Add	Delete	Update
`<diverts>`	X (no more than one can be present)	Deleting it means delete (undeploy) all diverts in running broker.	N/A
`<divert>`	Adding a new divert. It will be deployed after reloading	Deleting it means the divert will be undeployed after reloading	No effect on the deployed divert (unless restarting broker, in which case the divert will be redeployed)
attribute `name`	N/A	X	A new divert with the name will be deployed. (if it is not already there in broker). Otherwise no effect.
`<transformer-class-name>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker, in which case the divert will be deployed without the transformer class)	No effect on the deployed divert. (unless restarting broker, in which case the divert has the transformer class)
`<exclusive>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker)	No effect on the deployed divert. (unless restarting broker)
`<routing-name>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker)	No effect on the deployed divert. (unless restarting broker)
`<address>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker)	No effect on the deployed divert. (unless restarting broker)
`<forwarding-address>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker)	No effect on the deployed divert. (unless restarting broker)
`<filter>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker)	No effect on the deployed divert. (unless restarting broker)

Operation	Add	Delete	Update
`<routing-type>`	X (no more than one can be present)	No effect on the deployed divert. (unless restarting broker)	No effect on the deployed divert. (unless restarting broker)

`<addresses>`

The `<addresses>` element contains a list `<address>` elements. Once changed, all `<address>` elements in `<addresses>` will be reloaded.

> **Note:**
>
> Once reloaded, all new addresses (as well as the pre-configured queues) will be deployed to the running broker and all those that are missing from the configuration will be undeployed.
>
> **Note:**
>
> Parameters reloaded in this category will take effect immediately after reloading. The effect of deletion of Address's and Queue's, not auto created is controlled by parameter `config-delete-addresses` and `config-delete-queues` as described in this doc.

Below lists the effects of adding, deleting and updating of an element/attribute within the `<addresses>` element, whether a change can be done or can’t be done.

Operation	Add	Delete	Update
`<addresses>`	X(no more than one is present)	Deleting it means delete (undeploy) all diverts in running broker.	N/A
`<address>`	A new address will be deployed in the running broker	The corresponding address will be undeployed.	N/A
attribute `name`	N/A	X	After reloading the address of the old name will be undeployed and the new will be deployed.
`<anycast>`	X(no more than one is present)	The anycast routing type will be undeployed from this address, as well as its containing queues after reloading	N/A
`<queue>` (under `<anycast>`)	An anycast queue will be deployed after reloading	The anycast queue will be undeployed	For updating queues please see next section `<queues>`
`<multicast>`	X(no more than one is present)	The multicast routing type will be undeployed from this address, as well as its containing queues after reloading	N/A
`<queue>` (under `<multicast>`)	A multicast queue will be deployed after reloading	The multicast queue will be undeployed	For updating queues please see next section `<queues>`

`<queues>`

The `<queues>` element contains a list `<queue>` elements. Once changed, all `<queue>` elements in `<queues>` will be reloaded.

> **Note:**
>
> Once reloaded, all new queues will be deployed to the running broker and all queues that are missing from the configuration will be undeployed.
>
> **Note:**
>
> Parameters reloaded in this category will take effect immediately after reloading. The effect of deletion of Address's and Queue's, not auto created is controlled by parameter `config-delete-addresses` and `config-delete-queues` as described in this doc.

Below lists the effects of adding, deleting and updating of an element/attribute within the `<queues>` element, and whether a change can be done or can't be done.

Operation	Add	Delete	Update
`<queues>`	X(no more than one is present)	Deleting it means delete (undeploy) all queues from running broker.	N/A
`<queue>`	A new queue is deployed after reloading	The queue will be undeployed after reloading.	N/A
attribute `name`	N/A	X	A queue with new name will be deployed and the queue with old name will be updeployed after reloading (see Note above).
attribute `max-consumers`	If max-consumers > current consumers max-consumers will update on reload	max-consumers will be set back to the default `-1`	If max-consumers > current consumers max-consumers will update on reload
attribute `purge-on-no-consumers`	On reload purge-on-no-consumers will be updated	Will be set back to the default `false`	On reload purge-on-no-consumers will be updated
attribute `address`	N/A	No effect unless starting broker	No effect unless starting broker
attribute `filter`	The filter will be added after reloading	The filter will be removed after reloading	The filter will be updated after reloading
attribute `durable`	The queue durability will be set to the given value after reloading	The queue durability will be set to the default `true` after reloading	The queue durability will be set to the new value after reloading

`<jms>` *(Deprecated)*

`<queue>`

Changes to any `<queue>` elements will be reloaded to the running broker.

> **Note:**
>
> Once reloaded, new queues defined in the new changes will be deployed to the running broker. However existing queues won't get undeployed even if the matching element is deleted/missing. Also new queue elements matching existing queues won't get re-created – they remain unchanged.

Operation	Add	Delete	Update
`<queue>`	A new jms queue will be deployed after reloading	No effect unless starting broker	No effect unless starting broker
attribute `<name>`	N/A	X	A jms queue of the new name will be deployed after reloading
`<selector>`	X(no more than one is present)	No effect unless starting broker	No effect unless starting broker
`<durable>`	X(no more than one is present)	No effect unless starting broker	No effect unless starting broker

`<topic>`

Changes to any `<topic>` elements will be reloaded to the running broker.

> **Note:**
>
> Once reloaded, new topics defined in the new changes will be deployed to the running broker. However existing topics won't get undeployed even if the matching element is deleted/missing. Also any `<topic>` elements matching existing topics won't get re-deployed – they remain unchanged.

Operation	Add	Delete	Update
`<topic>`	A new jms topic will be deployed after reloading	No effect unless starting broker	No effect unless starting broker
attribute `name`	N/A	X	A jms topic of the new name will be deployed after reloading

Detecting Dead Connections

In this section we will discuss connection time-to-live (TTL) and explain how Apache ActiveMQ Artemis deals with crashed clients and clients which have exited without cleanly closing their resources.

Cleaning up Resources on the Server

Before an Apache ActiveMQ Artemis client application exits it is considered good practice that it should close its resources in a controlled manner, using a `finally` block.

Here's an example of a well behaved core client application closing its session and session factory in a finally block:

```
ServerLocator locator = null;
ClientSessionFactory sf = null;
ClientSession session = null;

try {
   locator = ActiveMQClient.createServerLocatorWithoutHA(..);

   sf = locator.createClientSessionFactory();;

   session = sf.createSession(...);

   ... do some stuff with the session...
} finally {
   if (session != null) {
      session.close();
   }

   if (sf != null) {
      sf.close();
   }

   if(locator != null) {
      locator.close();
   }
}
```

And here's an example of a well behaved JMS client application:

```
Connection jmsConnection = null;

try {
   ConnectionFactory jmsConnectionFactory = new ActiveMQConnectionFactory("tcp

   jmsConnection = jmsConnectionFactory.createConnection();

   ... do some stuff with the connection...
} finally {
   if (connection != null) {
      connection.close();
   }
}
```

Or with using auto-closeable feature from Java, which can save a few lines of code:

```
try (
     ActiveMQConnectionFactory jmsConnectionFactory = new ActiveMQConnectionFa
     Connection jmsConnection = jmsConnectionFactory.createConnection()) {
   ... do some stuff with the connection...
}
```

Unfortunately users don't always write well behaved applications, and sometimes clients just crash so they don't have a chance to clean up their resources!

If this occurs then it can leave server side resources, like sessions, hanging on the server. If these were not removed they would cause a resource leak on the server and over time this result in the server running out of memory or other resources.

We have to balance the requirement for cleaning up dead client resources with the fact that sometimes the network between the client and the server can fail and then come back, allowing the client to reconnect. Apache ActiveMQ Artemis supports client reconnection, so we don't want to clean up "dead" server side resources too soon or this will prevent any client from reconnecting, as it won't be able to find its old sessions on the server.

Apache ActiveMQ Artemis makes all of this configurable via a *connection TTL*. Basically, the TTL determines how long the server will keep a connection alive in the absence of any data arriving from the client. The client will automatically send "ping" packets periodically to prevent the server from closing it down. If the server doesn't receive any packets on a connection for the connection TTL time, then it will automatically close all the sessions on the server that relate to that connection.

The connection TTL is configured on the URI using the `connectionTTL` parameter.

The default value for connection ttl on an "unreliable" connection (e.g. a Netty connection using the `tcp` URL scheme) is `60000` ms, i.e. 1 minute. The default value for connection ttl on a "reliable" connection (e.g. an in-vm connection using the `vm` URL scheme) is `-1`. A value of `-1` for `connectionTTL` means the server will never time out the connection on the server side.

If you do not wish clients to be able to specify their own connection TTL, you can override all values used by a global value set on the server side. This can be done by specifying the `connection-ttl-override` attribute in the server side configuration. The default value for `connection-ttl-override` is `-1` which means "do not override" (i.e. let clients use their own values).

The logic to check connections for TTL violations runs periodically on the broker. By default, the checks are done every 2,000 milliseconds. However, this can be changed if necessary by using the `connection-ttl-check-interval` attribute.

Closing Forgotten Resources

As previously discussed, it's important that all core client sessions and JMS connections are always closed explicitly in a `finally` block when you are finished using them.

If you fail to do so, Apache ActiveMQ Artemis will detect this at garbage collection time, and log a warning (If you are using JMS the warning will involve a JMS connection).

Apache ActiveMQ Artemis will then close the connection / client session for you.

Note that the log will also tell you the exact line of your user code where you created the JMS connection / client session that you later did not close. This will enable you to pinpoint the error in your code and correct it appropriately.

Detecting Failure from the Client

In the previous section we discussed how the client sends pings to the server and how "dead" connection resources are cleaned up by the server. There's also another reason for pinging, and that's for the *client* to be able to detect that the server or network has failed.

As long as the client is receiving data from the server it will consider the connection to be still alive.

If the client does not receive any packets for a configurable number of milliseconds then it will consider the connection failed and will either initiate failover, or call any `FailureListener` instances (or `ExceptionListener` instances if you are using JMS) depending on how it has been configured.

This is controlled by setting the `clientFailureCheckPeriod` parameter on the URI your client is using to connect, e.g. `tcp://localhost:61616?clientFailureCheckPeriod=30000`.

The default value for client failure check period on an "unreliable" connection (e.g. a Netty connection) is `30000` ms, i.e. 30 seconds. The default value for client failure check period on a "reliable" connection (e.g. an in-vm connection) is `-1`. A value of `-1` means the client will never fail the connection on the client side if no data is received from the server. Typically this is much lower than connection TTL to allow clients to reconnect in case of transitory failure.

Configuring Asynchronous Connection Execution

Most packets received on the server side are executed on the remoting thread. These packets represent short-running operations and are always executed on the remoting thread for performance reasons.

However, by default some kinds of packets are executed using a thread from a thread pool so that the remoting thread is not tied up for too long. Please note that processing operations asynchronously on another thread adds a little more latency. These packets are:

- `org.apache.activemq.artemis.core.protocol.core.impl.wireformat.RollbackMessage`
- `org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionCloseMessage`
- `org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionCommitMessage`
- `org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXACommitMessage`
- `org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXAPrepareMessage`
- `org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXARollbackMessage`

To disable asynchronous connection execution, set the parameter `async-connection-execution-enabled` in `broker.xml` to `false` (default value is `true`).

Detecting Slow Consumers

In this section we will discuss how Apache ActiveMQ Artemis can be configured to deal with slow consumers. A slow consumer with a server-side queue (e.g. JMS topic subscriber) can pose a significant problem for broker performance. If messages build up in the consumer's server-side queue then memory will begin filling up and the broker may enter paging mode which would impact performance negatively. However, criteria can be set so that consumers which don't acknowledge messages quickly enough can potentially be disconnected from the broker which in the case of a non-durable JMS subscriber would allow the broker to remove the subscription and all of its messages freeing up valuable server resources.

Required Configuration

By default the server will not detect slow consumers. If slow consumer detection is desired then see address model chapter for more details on the required address settings.

The calculation to determine whether or not a consumer is slow only inspects the number of messages a particular consumer has *acknowledged*. It doesn't take into account whether or not flow control has been enabled on the consumer, whether or not the consumer is streaming a large message, etc. Keep this in mind when configuring slow consumer detection.

Please note that slow consumer checks are performed using the scheduled thread pool and that each queue on the broker with slow consumer detection enabled will cause a new entry in the internal `java.util.concurrent.ScheduledThreadPoolExecutor` instance. If there are a high number of queues and the `slow-consumer-check-period` is relatively low then there may be delays in executing some of the checks. However, this will not impact the accuracy of the calculations used by the detection algorithm. See thread pooling for more details about this pool.

Example

See the slow consumer example which shows how to detect a slow consumer with Apache ActiveMQ Artemis.

Network Isolation (Split Brain)

It is possible that if a replicated live or backup server becomes isolated in a network that failover will occur and you will end up with 2 live servers serving messages in a cluster, this we call split brain. There are different configurations you can choose from that will help mitigate this problem

Quorum Voting

Quorum voting is used by both the live and the backup to decide what to do if a replication connection is disconnected. Basically the server will request each live server in the cluster to vote as to whether it thinks the server it is replicating to or from is still alive. You can also configure the time for which the quorum manager will wait for the quorum vote response. The default time is 30 seconds you can configure like so for master and also for the slave:

```
<ha-policy>
  <replication>
    <master>
       <quorum-vote-wait>12</quorum-vote-wait>
    </master>
  </replication>
</ha-policy>
```

This being the case the minimum number of live/backup pairs needed is 3. If less than 3 pairs are used then the only option is to use a Network Pinger which is explained later in this chapter or choose how you want each server to react which the following details:

Backup Voting

By default if a replica loses its replication connection to the live broker it makes a decision as to whether to start or not with a quorum vote. This of course requires that there be at least 3 pairs of live/backup nodes in the cluster. For a 3 node cluster it will start if it gets 2 votes back saying that its live server is no longer available, for 4 nodes this would be 3 votes and so on. When a backup loses connection to the master it will keep voting for a quorum until it either receives a vote allowing it to start or it detects that the master is still live. for the latter it will then restart as a backup. How many votes and how long between each vote the backup should wait is configured like so:

```
<ha-policy>
  <replication>
    <slave>
       <vote-retries>12</vote-retries>
       <vote-retry-wait>5000</vote-retry-wait>
    </slave>
  </replication>
</ha-policy>
```

It's also possible to statically set the quorum size that should be used for the case where the cluster size is known up front, this is done on the Replica Policy like so:

```
<ha-policy>
  <replication>
    <slave>
       <quorum-size>2</quorum-size>
    </slave>
  </replication>
</ha-policy>
```

In this example the quorum size is set to 2 so if you were using a single pair and the backup lost connectivity it would never start.

Live Voting

By default, if the live server loses its replication connection then it will just carry on and wait for a backup to reconnect and start replicating again. In the event of a possible split brain scenario this may mean that the live stays live even though the backup has been activated. It is possible to configure the live server to vote for a quorum if this happens, in this way if the live server doesn't not receive a majority vote then it will shutdown. This is done by setting the *vote-on-replication-failure* to true.

```
<ha-policy>
  <replication>
    <master>
       <vote-on-replication-failure>true</vote-on-replication-failure>
       <quorum-size>2</quorum-size>
    </master>
  </replication>
</ha-policy>
```

As in the backup policy it is also possible to statically configure the quorum size.

Pinging the network

You may configure one more addresses on the broker.xml that are part of your network topology, that will be pinged through the life cycle of the server.

The server will stop itself until the network is back on such case.

If you execute the create command passing a -ping argument, you will create a default xml that is ready to be used with network checks:

```
./artemis create /myDir/myServer --ping 10.0.0.1
```

This XML part will be added to your broker.xml:

```
<!--
You can verify the network health of a particular NIC by specifying the <netwo
 <network-check-NIC>theNicName</network-check-NIC>
-->

<!--
Use this to use an HTTP server to validate the network
 <network-check-URL-list>http://www.apache.org</network-check-URL-list> -->

<network-check-period>10000</network-check-period>
<network-check-timeout>1000</network-check-timeout>

<!-- this is a comma separated list, no spaces, just DNS or IPs
   it should accept IPV6

   Warning: Make sure you understand your network topology as this is meant to
            Using IPs that could eventually disappear or be partially visible
            You can use a list of multiple IPs, any successful ping will make
<network-check-list>10.0.0.1</network-check-list>

<!-- use this to customize the ping used for ipv4 addresses -->
<network-check-ping-command>ping -c 1 -t %d %s</network-check-ping-command>

<!-- use this to customize the ping used for ipv addresses -->
<network-check-ping6-command>ping6 -c 1 %2$s</network-check-ping6-command>
```

Once you lose connectivity towards 10.0.0.1 on the given example, you will see see this output at the server:

```
09:49:24,562 WARN  [org.apache.activemq.artemis.core.server.NetworkHealthCheck
09:49:36,577 INFO  [org.apache.activemq.artemis.core.server.NetworkHealthCheck
09:49:36,625 INFO  [org.apache.activemq.artemis.core.server] AMQ221002: Apache

09:50:00,653 WARN  [org.apache.activemq.artemis.core.server.NetworkHealthCheck
09:50:10,656 WARN  [org.apache.activemq.artemis.core.server.NetworkHealthCheck
    at java.net.Inet6AddressImpl.isReachable0(Native Method) [rt.jar:1.8.0_73]
    at java.net.Inet6AddressImpl.isReachable(Inet6AddressImpl.java:77) [rt.jar
    at java.net.InetAddress.isReachable(InetAddress.java:502) [rt.jar:1.8.0_73
    at org.apache.activemq.artemis.core.server.NetworkHealthCheck.check(Networ
    at org.apache.activemq.artemis.core.server.NetworkHealthCheck.check(Networ
    at org.apache.activemq.artemis.core.server.NetworkHealthCheck.run(NetworkH
    at org.apache.activemq.artemis.core.server.ActiveMQScheduledComponent$2.ru
    at org.apache.activemq.artemis.core.server.ActiveMQScheduledComponent$3.ru
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308) [rt.ja
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.ac
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.ru
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.ja
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.j
    at java.lang.Thread.run(Thread.java:745) [rt.jar:1.8.0_73]
```

Once you re establish your network connections towards the configured check list:

```
09:53:23,461 INFO  [org.apache.activemq.artemis.core.server.NetworkHealthCheck
09:53:23,462 INFO  [org.apache.activemq.artemis.core.server] AMQ221000: live M
09:53:23,462 INFO  [org.apache.activemq.artemis.core.server] AMQ221013: Using
09:53:23,462 INFO  [org.apache.activemq.artemis.core.server] AMQ221043: Protoc
09:53:23,463 INFO  [org.apache.activemq.artemis.core.server] AMQ221043: Protoc
09:53:23,463 INFO  [org.apache.activemq.artemis.core.server] AMQ221043: Protoc
09:53:23,463 INFO  [org.apache.activemq.artemis.core.server] AMQ221043: Protoc
09:53:23,464 INFO  [org.apache.activemq.artemis.core.server] AMQ221043: Protoc
09:53:23,464 INFO  [org.apache.activemq.artemis.core.server] AMQ221043: Protoc
09:53:23,541 INFO  [org.apache.activemq.artemis.core.server] AMQ221003: Deploy
09:53:23,541 INFO  [org.apache.activemq.artemis.core.server] AMQ221003: Deploy
09:53:23,549 INFO  [org.apache.activemq.artemis.core.server] AMQ221020: Starte
09:53:23,550 INFO  [org.apache.activemq.artemis.core.server] AMQ221020: Starte
09:53:23,554 INFO  [org.apache.activemq.artemis.core.server] AMQ221020: Starte
09:53:23,555 INFO  [org.apache.activemq.artemis.core.server] AMQ221020: Starte
09:53:23,556 INFO  [org.apache.activemq.artemis.core.server] AMQ221020: Starte
09:53:23,556 INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server
09:53:23,556 INFO  [org.apache.activemq.artemis.core.server] AMQ221001: Apache
```

> ## Warning
>
> Make sure you understand your network topology as this is meant to validate your network. Using IPs that could eventually disappear or be partially visible may defeat the purpose. You can use a list of multiple IPs. Any successful ping will make the server OK to continue running

Critical Analysis of the broker

There are a few things that can go wrong on a production environment:

- Bugs, for more than we try they still happen! We always try to correct them, but that's the only constant in software development.
- IO Errors, disks and hardware can go bad
- Memory issues, the CPU can go crazy by another process

For cases like this, we added a protection to the broker to shut itself down when bad things happen.

This is a feature I hope you won't need it, think it as a safeguard:

We measure time response in places like:

- Queue delivery (add to the queue)
- Journal storage
- Paging operations

If the response time goes beyond a configured timeout, the broker is considered unstable and an action will be taken to either shutdown the broker or halt the VM.

You can use these following configuration options on broker.xml to configure how the critical analysis is performed.

Name	Description
critical-analyzer	Enable or disable the critical analysis (default true)
critical-analyzer-timeout	Timeout used to do the critical analysis (default 120000 milliseconds)
critical-analyzer-check-period	Time used to check the response times (default half of critical-analyzer-timeout)
critical-analyzer-policy	Should the server log, be halted or shutdown upon failures (default `LOG`)

The default for critical-analyzer-policy is `LOG`, however the generated broker.xml will have it set to `HALT`. That is because we cannot halt the VM if you are embedding ActiveMQ Artemis into an application server or on a multi tenant environment.

The broker on the distribution will then have it set to `HALT`, but if you use it in any other way the default will be `LOG`.

What to Expect

- You will see some logs

If you have critical-analyzer-policy=HALT

```
[Artemis Critical Analyzer] 18:10:00,831 ERROR [org.apache.activemq.artemis.co
```

While if you have critical-analyzer-policy= `SHUTDOWN`

```
[Artemis Critical Analyzer] 18:07:53,475 ERROR [org.apache.activemq.artemis.co
```

Or if you have critical-analyzer-policy=LOG

```
[Artemis Critical Analyzer] 18:11:52,145 WARN [org.apache.activemq.artemis.cor
```

You will see a simple thread dump of the server

```
[Artemis Critical Analyzer] 18:10:00,836 WARN  [org.apache.activemq.artemis.co
********************************************************************************
================================================================================
AMQ119002: Thread Thread[Thread-1 (ActiveMQ-scheduled-threads),5,main] name = 

sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Ab
java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(Schedul
java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(Schedul
java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617
java.lang.Thread.run(Thread.java:745)
================================================================================


..... blablablablaba ..........


================================================================================
AMQ119003: End Thread dump
********************************************************************************
```

- The Server will be halted if configured to `HALT`
- The system will be stopped if `SHUTDOWN` is used. **Notice**: If the system is not behaving well, there is no guarantees the stop will work.

Resource Manager Configuration

Apache ActiveMQ Artemis has its own Resource Manager for handling the lifespan of JTA transactions. When a transaction is started the resource manager is notified and keeps a record of the transaction and its current state. It is possible in some cases for a transaction to be started but then forgotten about. Maybe the client died and never came back. If this happens then the transaction will just sit there indefinitely.

To cope with this Apache ActiveMQ Artemis can, if configured, scan for old transactions and rollback any it finds. The default for this is 3000000 milliseconds (5 minutes), i.e. any transactions older than 5 minutes are removed. This timeout can be changed by editing the `transaction-timeout` property in `broker.xml` (value must be in milliseconds). The property `transaction-timeout-scan-period` configures how often, in milliseconds, to scan for old transactions.

Please note that Apache ActiveMQ Artemis will not unilaterally rollback any XA transactions in a prepared state - this must be heuristically rolled back via the management API if you are sure they will never be resolved by the transaction manager.

Flow Control

Flow control is used to limit the flow of data between a client and server, or a server and another server in order to prevent the client or server being overwhelmed with data.

Consumer Flow Control

This controls the flow of data between the server and the client as the client consumes messages. For performance reasons clients normally buffer messages before delivering to the consumer via the `receive()` method or asynchronously via a message listener. If the consumer cannot process messages as fast as they are being delivered and stored in the internal buffer, then you could end up with a situation where messages would keep building up possibly causing out of memory on the client if they cannot be processed in time.

Window-Based Flow Control

By default, Apache ActiveMQ Artemis consumers buffer messages from the server in a client side buffer before the client consumes them. This improves performance: otherwise every time the client consumes a message, Apache ActiveMQ Artemis would have to go the server to request the next message. In turn, this message would then get sent to the client side, if one was available.

A network round trip would be involved for *every* message and considerably reduce performance.

To prevent this, Apache ActiveMQ Artemis pre-fetches messages into a buffer on each consumer. The total maximum size of messages (in bytes) that will be buffered on each consumer is determined by the `consumerWindowSize` parameter.

By default, the `consumerWindowSize` is set to 1 MiB (1024 * 1024 bytes) unless overridden via (Address Settings)

The value can be:

- `-1` for an *unbounded* buffer
- `0` to not buffer any messages.
- `>0` for a buffer with the given maximum size in bytes.

Setting the consumer window size can considerably improve performance depending on the messaging use case. As an example, let's consider the two extremes:

Fast consumers

Fast consumers can process messages as fast as they consume them (or even faster)

To allow fast consumers, set the `consumerWindowSize` to -1. This will allow *unbounded* message buffering on the client side.

Use this setting with caution: it can overflow the client memory if the consumer is not able to process messages as fast as it receives them.

Slow consumers

Slow consumers takes significant time to process each message and it is desirable to prevent buffering messages on the client side so that they can be delivered to another consumer instead.

Consider a situation where a queue has 2 consumers; 1 of which is very slow. Messages are delivered in a round robin fashion to both consumers, the fast consumer processes all of its messages very quickly until its buffer is empty. At this point there are still messages awaiting to be processed in the buffer of the slow consumer thus preventing them being processed by the fast consumer. The fast consumer is therefore sitting idle when it could be processing the other messages.

To allow slow consumers, set `consumerWindowSize` on the URI to 0 (for no buffer at all). This will prevent the slow consumer from buffering any messages on the client side. Messages will remain on the server side ready to be consumed by other consumers.

Setting this to 0 can give deterministic distribution between multiple consumers on a queue.

Most of the consumers cannot be clearly identified as fast or slow consumers but are in-between. In that case, setting the value of `consumerWindowSize` to optimize performance depends on the messaging use case and requires benchmarks to find the optimal value, but a value of 1MiB is fine in most cases.

Please see the examples chapter for an example which shows how to configure ActiveMQ Artemis to prevent consumer buffering when dealing with slow consumers.

Rate limited flow control

It is also possible to control the *rate* at which a consumer can consume messages. This is a form of throttling and can be used to make sure that a consumer never consumes messages at a rate faster than the rate specified. This is configured using the `consumerMaxRate` URI parameter.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting this to `-1` disables rate limited flow control. The default value is `-1`.

Please see the examples chapter for a working example of limiting consumer rate.

> **Note:**
>
> Rate limited flow control can be used in conjunction with window based flow control. Rate limited flow control only effects how many messages a client can consume in a second and not how many messages are in its buffer. So if you had a slow rate limit and a high window based limit the clients internal buffer would soon fill up with messages.

Producer flow control

Apache ActiveMQ Artemis also can limit the amount of data sent from a client to a server to prevent the server being overwhelmed.

Window based flow control

In a similar way to consumer window based flow control, Apache ActiveMQ Artemis producers, by default, can only send messages to an address as long as they have sufficient credits to do so. The amount of credits required to send a message is given by the size of the message.

As producers run low on credits they request more from the server, when the server sends them more credits they can send more messages.

The amount of credits a producer requests in one go is known as the *window size* and it is controlled by the `producerWindowSize` URI parameter.

The window size therefore determines the amount of bytes that can be in-flight at any one time before more need to be requested - this prevents the remoting connection from getting overloaded.

Blocking CORE Producers

When using the CORE protocol (used by both the Artemis Core Client and Artemis JMS Client) the server will always aim give the same number of credits as have been requested. However, it is also possible to set a maximum size on any address, and the server will never send more credits to any one producer than what is available according to the address's upper memory limit. Although a single producer will be issued more credits than available (at the time of issue) it is possible that more than 1 producer be associated with the same address and so it is theoretically possible that more credits are allocated across total producers than what is available. It is therefore possible to go over the address limit by approximately:

```
total number of producers on address * producer window size
```

For example, if I have a queue called "myqueue", I could set the maximum memory size to 10MiB, and the the server will control the number of credits sent to any producers which are sending any messages to myqueue such that the total

messages in the queue never exceeds 10MiB.

When the address gets full, producers will block on the client side until more space frees up on the address, i.e. until messages are consumed from the queue thus freeing up space for more messages to be sent.

We call this blocking producer flow control, and it's an efficient way to prevent the server running out of memory due to producers sending more messages than can be handled at any time.

It is an alternative approach to paging, which does not block producers but instead pages messages to storage.

To configure an address with a maximum size and tell the server that you want to block producers for this address if it becomes full, you need to define an AddressSettings (Configuring Queues Via Address Settings) block for the address and specify `max-size-bytes` and `address-full-policy`

The address block applies to all queues registered to that address. I.e. the total memory for all queues bound to that address will not exceed `max-size-bytes`. In the case of JMS topics this means the *total* memory of all subscriptions in the topic won't exceed max-size-bytes.

Here's an example:

```
<address-settings>
   <address-setting match="exampleQueue">
      <max-size-bytes>100000</max-size-bytes>
      <address-full-policy>BLOCK</address-full-policy>
   </address-setting>
</address-settings>
```

The above example would set the max size of the queue "exampleQueue" to be 100000 bytes and would block any producers sending to that address to prevent that max size being exceeded.

Note the policy must be set to `BLOCK` to enable blocking producer flow control.

> **Note:**
>
> Note that in the default configuration all addresses are set to block producers after 10 MiB of message data is in the address. This means you cannot send more than 10MiB of message data to an address without it being consumed before the producers will be blocked. If you do not want this behaviour increase the `max-size-bytes` parameter or change the address full message policy.
>
> **Note:**
>
> Producer credits are allocated from the broker to the client. Flow control credit checking (i.e. checking a producer has enough credit) is done on the client side only. It is possible for the broker to over allocate credits, like in the multiple producer scenario outlined above. It is also possible for a misbehaving client to ignore the flow control credits issued by the broker and continue sending with out sufficient credit.

Blocking AMQP Producers

Apache ActiveMQ Artemis ships with out of the box with 2 protocols that support flow control. Artemis CORE protocol and AMQP. Both protocols implement flow control slightly differently and therefore address full BLOCK policy behaves slightly different for clients that use each protocol respectively.

As explained earlier in this chapter the CORE protocol uses a producer window size flow control system. Where credits (representing bytes) are allocated to producers, if a producer wants to send a message it should wait until it has enough byte credits available for it to send. AMQP flow control credits are not representative of bytes but instead represent the number of messages a producer is permitted to send (regardless of the message size).

BLOCK for AMQP works mostly in the same way as the producer window size mechanism above. Artemis will issue 100 credits to a client at a time and refresh them when the clients credits reaches 30. The broker will stop issuing credits once an address is full. However, since AMQP credits represent whole messages and not bytes, it would be possible in some scenarios for an AMQP client to significantly exceed an address upper bound should the broker continue accepting messages until the clients credits are exhausted. For this reason there is an additional parameter available on address settings that specifies an upper bound on an address size in bytes. Once this upper bound is reach Artemis will start rejecting AMQP messages. This limit is the max-size-bytes-reject-threshold and is by default set to -1 (or no limit). This is additional parameter allows a kind of soft and hard limit, in normal circumstances the broker will utilize the max-size-bytes parameter using using flow control to put back pressure on the client, but will protect the broker by rejecting messages once the address size is reached.

Rate limited flow control

Apache ActiveMQ Artemis also allows the rate a producer can emit message to be limited, in units of messages per second. By specifying such a rate, Apache ActiveMQ Artemis will ensure that producer never produces messages at a rate higher than that specified. This is controlled by the `producerMaxRate` URL parameter.

The `producerMaxRate` must be a positive integer to enable this functionality and is the maximum desired message production rate specified in units of messages per second. Setting this to `-1` disables rate limited flow control. The default value is `-1`.

Please see the examples chapter for a working example of limiting producer rate.

Guarantees of Sends and Commits

Transaction Completion

When committing or rolling back a transaction with Apache ActiveMQ Artemis, the request to commit or rollback is sent to the server, and the call will block on the client side until a response has been received from the server that the commit or rollback was executed.

When the commit or rollback is received on the server, it will be committed to the journal, and depending on the value of the parameter `journal-sync-transactional` the server will ensure that the commit or rollback is durably persisted to storage before sending the response back to the client. If this parameter has the value `false` then commit or rollback may not actually get persisted to storage until some time after the response has been sent to the client. In event of server failure this may mean the commit or rollback never gets persisted to storage. The default value of this parameter is `true` so the client can be sure all transaction commits or rollbacks have been persisted to storage by the time the call to commit or rollback returns.

Setting this parameter to `false` can improve performance at the expense of some loss of transaction durability.

This parameter is set in `broker.xml`

Non Transactional Message Sends

If you are sending messages to a server using a non transacted session, Apache ActiveMQ Artemis can be configured to block the call to send until the message has definitely reached the server, and a response has been sent back to the client. This can be configured individually for durable and non-durable messages, and is determined by the following two URL parameters:

- `blockOnDurableSend`. If this is set to `true` then all calls to send for durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is `true`.
- `blockOnNonDurableSend`. If this is set to `true` then all calls to send for non-durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is `false`.

Setting block on sends to `true` can reduce performance since each send requires a network round trip before the next send can be performed. This means the performance of sending messages will be limited by the network round trip time (RTT) of your network, rather than the bandwidth of your network. For better performance we recommend either batching many messages sends together in a

transaction since with a transactional session, only the commit / rollback blocks not every send, or, using Apache ActiveMQ Artemis's advanced *asynchronous send acknowledgements feature* described in Asynchronous Send Acknowledgements.

When the server receives a message sent from a non transactional session, and that message is durable and the message is routed to at least one durable queue, then the server will persist the message in permanent storage. If the journal parameter `journal-sync-non-transactional` is set to `true` the server will not send a response back to the client until the message has been persisted and the server has a guarantee that the data has been persisted to disk. The default value for this parameter is `true`.

Non Transactional Acknowledgements

If you are acknowledging the delivery of a message at the client side using a non transacted session, Apache ActiveMQ Artemis can be configured to block the call to acknowledge until the acknowledge has definitely reached the server, and a response has been sent back to the client. This is configured with the parameter `BlockOnAcknowledge`. If this is set to `true` then all calls to acknowledge on non transacted sessions will block until the acknowledge has reached the server, and a response has been sent back. You might want to set this to `true` if you want to implement a strict *at most once* delivery policy. The default value is `false`

Asynchronous Send Acknowledgements

If you are using a non transacted session but want a guarantee that every message sent to the server has reached it, then, as discussed in Guarantees of Non Transactional Message Sends, you can configure Apache ActiveMQ Artemis to block the call to send until the server has received the message, persisted it and sent back a response. This works well but has a severe performance penalty - each call to send needs to block for at least the time of a network round trip (RTT) - the performance of sending is thus limited by the latency of the network, *not* limited by the network bandwidth.

Let's do a little bit of maths to see how severe that is. We'll consider a standard 1Gib ethernet network with a network round trip between the server and the client of 0.25 ms.

With a RTT of 0.25 ms, the client can send *at most* 1000/ 0.25 = 4000 messages per second if it blocks on each message send.

If each message is < 1500 bytes and a standard 1500 bytes MTU (Maximum Transmission Unit) size is used on the network, then a 1GiB network has a *theoretical* upper limit of (1024 * 1024 * 1024 / 8) / 1500 = 89478 messages per second if messages are sent without blocking! These figures aren't an exact science but you can clearly see that being limited by network RTT can have serious effect on performance.

To remedy this, Apache ActiveMQ Artemis provides an advanced new feature called *asynchronous send acknowledgements*. With this feature, Apache ActiveMQ Artemis can be configured to send messages without blocking in one direction and asynchronously getting acknowledgement from the server that the messages were received in a separate stream. By de-coupling the send from the acknowledgement of the send, the system is not limited by the network RTT, but is limited by the network bandwidth. Consequently better throughput can be achieved than is possible using a blocking approach, while at the same time having absolute guarantees that messages have successfully reached the server.

The window size for send acknowledgements is determined by the confirmation-window-size parameter on the connection factory or client session factory. Please see Client Reconnection and Session Reattachment for more info on this.

To use the feature using the core API, you implement the interface `org.apache.activemq.artemis.api.core.client.SendAcknowledgementHandler` and set a handler instance on your `ClientSession`.

Then, you just send messages as normal using your `ClientSession`, and as messages reach the server, the server will send back an acknowledgement of the send asynchronously, and some time later you are informed at the client side by Apache ActiveMQ Artemis calling your handler's `sendAcknowledged(ClientMessage message)` method, passing in a reference to the message that was sent.

To enable asynchronous send acknowledgements you must make sure `confirmationWindowSize` is set to a positive integer value, e.g. 10MiB

Please see the examples chapter for a full working example.

Message Redelivery and Undelivered Messages

Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back). Such a message goes back to its queue ready to be redelivered. However, this means it is possible for a message to be delivered again and again without success thus remaining in the queue indefinitely, clogging the system.

There are 2 ways to deal with these undelivered messages:

- Delayed redelivery.

  It is possible to delay messages redelivery. This gives the client some time to recover from any transient failures and to prevent overloading its network or CPU resources.

- Dead Letter Address.

  It is also possible to configure a dead letter address so that after a specified number of unsuccessful deliveries, messages are removed from their queue and sent to the dead letter address. These messages will not be delivered again from this queue.

Both options can be combined for maximum flexibility.

Delayed Redelivery

Delaying redelivery can often be useful in cases where clients regularly fail or rollback. Without a delayed redelivery, the system can get into a "thrashing" state, with delivery being attempted, the client rolling back, and delivery being re-attempted ad infinitum in quick succession, consuming valuable CPU and network resources.

Configuring Delayed Redelivery

Delayed redelivery is defined in the address-setting configuration:

```
<!-- delay redelivery of messages for 5s -->
<address-setting match="exampleQueue">
   <!-- default is 1.0 -->
   <redelivery-delay-multiplier>1.5</redelivery-delay-multiplier>
   <!-- default is 0 (no delay) -->
   <redelivery-delay>5000</redelivery-delay>
   <!-- default is 0.0) -->
   <redelivery-collision-avoidance-factor>0.15</redelivery-collision-avoidance
   <!-- default is redelivery-delay * 10 -->
   <max-redelivery-delay>50000</max-redelivery-delay>
</address-setting>
```

If a `redelivery-delay` is specified, Apache ActiveMQ Artemis will wait this delay before redelivering the messages.

By default, there is no redelivery delay (`redelivery-delay` is set to 0).

Other subsequent messages will be delivery regularly, only the cancelled message will be sent asynchronously back to the queue after the delay.

You can specify a multiplier (the `redelivery-delay-multiplier`) that will take effect on top of the `redelivery-delay` . Each time a message is redelivered the delay period will be equal to the previous delay `redelivery-delay-multiplier` . *A* `max-redelivery-delay` *can be set to prevent the delay from becoming too large. The* `max-redelivery-delay` *is defaulted to* `redelivery-delay` \ 10.

Example:

- redelivery-delay=5000, redelivery-delay-multiplier=2, max-redelivery-delay=15000, redelivery-collision-avoidance-factor=0.0
- Delivery Attempt 1. (Unsuccessful)
- Wait Delay Period: 5000
- Delivery Attempt 2. (Unsuccessful)
- Wait Delay Period: 10000 // (5000 * 2) < max-delay-period. Use 10000
- Delivery Attempt 3: (Unsuccessful)
- Wait Delay Period: 15000 // (10000 * 2) > max-delay-period: Use max-delay-delivery

Address wildcards can be used to configure redelivery delay for a set of addresses (see Understanding the Wildcard Syntax), so you don't have to specify redelivery delay individually for each address.

The `redelivery-delay` can be also be modified by configuring the `redelivery-collision-avoidance-factor` . This factor will be made either positive or negative at random to control whether the ultimate value will increase or decrease the `redelivery-delay` . Then it's multiplied by a random number between 0.0 and 1.0. This result is then multiplied by the `redelivery-delay` and then added to the `redelivery-delay` to arrive at the final value.

The algorithm may sound complicated but the bottom line is quite simple: the larger `redelivery-collision-avoidance-factor` you choose the larger the variance of the `redelivery-delay` will be. The `redelivery-collision-avoidance-factor` must be between 0.0 and 1.0.

Example:

- redelivery-delay=1000, redelivery-delay-multiplier=1, max-redelivery-delay=15000, redelivery-collision-avoidance-factor=0.5, (bold values chosen using `java.util.Random`)
- Delivery Attempt 1. (Unsuccessful)
- Wait Delay Period: 875 // 1000 + (1000 *((0.5* \ **-1**) * **.25**)
- Delivery Attempt 2. (Unsuccessful)
- Wait Delay Period: 1375 // 1000 + (1000 *((0.5* \ **1**) * **.75**)

- Delivery Attempt 3: (Unsuccessful)
- Wait Delay Period: 975 // 1000 + (1000 *((0.5* \ **-1**) * **.05**)

This feature can be particularly useful in environments where there are multiple consumers on the same queue all interacting transactionally with the same external system (e.g. a database). If there is overlapping data in messages which are consumed concurrently then one transaction can succeed while all the rest fail. If those failed messages are redelivered at the same time then this process where one consumer succeeds and the rest fail will continue. By randomly padding the redelivery-delay by a small, configurable amount these redelivery "collisions" can be avoided.

Example

See the examples chapter for an example which shows how delayed redelivery is configured and used with JMS.

Dead Letter Addresses

To prevent a client infinitely receiving the same undelivered message (regardless of what is causing the unsuccessful deliveries), messaging systems define *dead letter addresses*: after a specified unsuccessful delivery attempts, the message is removed from its queue and sent to a dead letter address.

Any such messages can then be diverted to queue(s) where they can later be perused by the system administrator for action to be taken.

Apache ActiveMQ Artemis's addresses can be assigned a dead letter address. Once the messages have been unsuccessfully delivered for a given number of attempts, they are removed from their queue and sent to the relevant dead letter address. These *dead letter* messages can later be consumed from the dead letter address for further inspection.

Configuring Dead Letter Addresses

Dead letter address is defined in the address-setting configuration:

```
<!-- undelivered messages in exampleQueue will be sent to the dead letter addr
deadLetterQueue after 3 unsuccessful delivery attempts -->
<address-setting match="exampleQueue">
   <dead-letter-address>deadLetterAddress</dead-letter-address>
   <max-delivery-attempts>3</max-delivery-attempts>
</address-setting>
```

If a `dead-letter-address` is not specified, messages will removed after `max-delivery-attempts` unsuccessful attempts.

By default, messages are redelivered 10 times at the maximum. Set `max-delivery-attempts` to -1 for infinite redeliveries.

A `dead letter address` can be set globally for a set of matching addresses and you can set `max-delivery-attempts` to -1 for a specific address setting to allow infinite redeliveries only for this address.

Address wildcards can be used to configure dead letter settings for a set of addresses (see Understanding the Wildcard Syntax).

Dead Letter Properties

Dead letter messages get special properties.

Automatically Creating Dead Letter Resources

It's common to segregate undelivered messages by their original address. For example, a message sent to the `stocks` address that couldn't be delivered for some reason might be ultimately routed to the `DLQ.stocks` queue, and likewise a message sent to the `orders` address that couldn't be delivered might be routed to the `DLQ.orders` queue.

Using this pattern can make it easy to track and administrate undelivered messages. However, it can pose a challenge in environments which predominantly use auto-created addresses and queues. Typically administrators in those environments don't want to manually create an `address-setting` to configure the `dead-letter-address` much less the actual `address` and `queue` to hold the undelivered messages.

The solution to this problem is to set the `auto-create-dead-letter-resources` `address-setting` to `true` (it's `false` by default) so that the broker will create the `address` and `queue` to deal with the undelivered messages automatically. The `address` created will be the one defined by the `dead-letter-address`. A `MULTICAST` `queue` will be created on that `address`. It will be named by the `address` to which the message was previously sent, and it will have a filter defined using the property `_AMQ_ORIG_ADDRESS` so that it will only receive messages sent to the relevant `address`. The `queue` name can be configured with a prefix and suffix. See the relevant settings in the table below:

address-setting	default
dead-letter-queue-prefix	DLQ.
dead-letter-queue-suffix	(empty string)

Here is an example configuration:

```
<address-setting match="#">
   <dead-letter-address>DLA</dead-letter-address>
   <max-delivery-attempts>3</max-delivery-attempts>
   <auto-create-dead-letter-resources>true</auto-create-dead-letter-resources>
   <dead-letter-queue-prefix></dead-letter-queue-prefix> <!-- override the def
   <dead-letter-queue-suffix>.DLQ</dead-letter-queue-suffix>
</address-setting>
```

The queue holding the undeliverable messages can be accessed directly either by using the queue's name by itself (e.g. when using the core client) or by using the fully qualified queue name (e.g. when using a JMS client) just like any other queue. Also, note that the queue is auto-created which means it will be auto-deleted as per the relevant `address-settings`.

Example

See: Dead Letter section of the Examples for an example that shows how dead letter resources can be statically configured and used with JMS.

Delivery Count Persistence

In normal use, Apache ActiveMQ Artemis does not update delivery count *persistently* until a message is rolled back (i.e. the delivery count is not updated *before* the message is delivered to the consumer). In most messaging use cases, the messages are consumed, acknowledged and forgotten as soon as they are consumed. In these cases, updating the delivery count persistently before delivering the message would add an extra persistent step *for each message delivered*, implying a significant performance penalty.

However, if the delivery count is not updated persistently before the message delivery happens, in the event of a server crash, messages might have been delivered but that will not have been reflected in the delivery count. During the recovery phase, the server will not have knowledge of that and will deliver the message with `redelivered` set to `false` while it should be `true`.

As this behavior breaks strict JMS semantics, Apache ActiveMQ Artemis allows to persist delivery count before message delivery but this feature is disabled by default due to performance implications.

To enable it, set `persist-delivery-count-before-delivery` to `true` in `broker.xml`:

```
<persist-delivery-count-before-delivery>true</persist-delivery-count-before-de
```

Message Expiry

Messages can be set with an optional *time to live* when sending them.

Apache ActiveMQ Artemis will not deliver a message to a consumer after it's time to live has been exceeded. If the message hasn't been delivered by the time that time to live is reached the server can discard it.

Apache ActiveMQ Artemis's addresses can be assigned an expiry address so that, when messages are expired, they are removed from the queue and sent to the expiry address. Many different queues can be bound to an expiry address. These *expired* messages can later be consumed for further inspection.

Core API

Using Apache ActiveMQ Artemis Core API, you can set an expiration time directly on the message:

```
// message will expire in 5000ms from now
message.setExpiration(System.currentTimeMillis() + 5000);
```

JMS MessageProducer allows to set a TimeToLive for the messages it sent:

```
// messages sent by this producer will be retained for 5s (5000ms) before expi
producer.setTimeToLive(5000);
```

Expired messages get special properties plus this additional property:

- `_AMQ_ACTUAL_EXPIRY`

  a Long property containing the *actual expiration time* of the expired message

Configuring Expiry Delay

Default expiry delay can be configured in the address-setting configuration:

```
<!-- expired messages in exampleQueue will be sent to the expiry address expir
<address-setting match="exampleQueue">
   <expiry-address>expiryQueue</expiry-address>
   <expiry-delay>10</expiry-delay>
</address-setting>
```

`expiry-delay` defines the expiration time in milliseconds that will be used for messages which are using the default expiration time (i.e. 0).

For example, if `expiry-delay` is set to "10" and a message which is using the default expiration time (i.e. 10) arrives then its expiration time of "0" will be changed to "10." However, if a message which is using an expiration time of "20"

arrives then its expiration time will remain unchanged. Setting `expiry-delay` to "-1" will disable this feature.

The default is `-1`.

If `expiry-delay` is *not set* then minimum and maximum expiry delay values can be configured in the address-setting configuration.

```
<address-setting match="exampleQueue">
   <min-expiry-delay>10</min-expiry-delay>
   <max-expiry-delay>100</max-expiry-delay>
</address-setting>
```

Semantics are as follows:

- Messages *without* an expiration will be set to `max-expiry-delay`. If `max-expiry-delay` is not defined then the message will be set to `min-expiry-delay`. If `min-expiry-delay` is not defined then the message will not be changed.
- Messages with an expiration *above* `max-expiry-delay` will be set to `max-expiry-delay`
- Messages with an expiration *below* `min-expiry-delay` will be set to `min-expiry-delay`
- Messages with an expiration *within* `min-expiry-delay` and `max-expiry-delay` range will not be changed
- Any value set for `expiry-delay` other than the default (i.e. `-1`) will override the aforementioned min/max settings.

The default for both `min-expiry-delay` and `max-expiry-delay` is `-1` (i.e. disabled).

Configuring Expiry Addresses

Expiry address are defined in the address-setting configuration:

```
<!-- expired messages in exampleQueue will be sent to the expiry address expir
<address-setting match="exampleQueue">
   <expiry-address>expiryQueue</expiry-address>
</address-setting>
```

If messages are expired and no expiry address is specified, messages are simply removed from the queue and dropped. Address wildcards can be used to configure expiry address for a set of addresses.

Configuring Automatic Creation of Expiry Resources

It's common to segregate expired messages by their original address. For example, a message sent to the `stocks` address that expired for some reason might be ultimately routed to the `EXP.stocks` queue, and likewise a message

sent to the `orders` address that expired might be routed to the `EXP.orders` queue.

Using this pattern can make it easy to track and administrate expired messages. However, it can pose a challenge in environments which predominantly use auto-created addresses and queues. Typically administrators in those environments don't want to manually create an `address-setting` to configure the `expiry-address` much less the actual `address` and `queue` to hold the expired messages.

The solution to this problem is to set the `auto-create-expiry-resources` `address-setting` to `true` (it's `false` by default) so that the broker will create the `address` and `queue` to deal with the expired messages automatically. The `address` created will be the one defined by the `expiry-address`. A `MULTICAST` `queue` will be created on that `address`. It will be named by the `address` to which the message was previously sent, and it will have a filter defined using the property `_AMQ_ORIG_ADDRESS` so that it will only receive messages sent to the relevant `address`. The `queue` name can be configured with a prefix and suffix. See the relevant settings in the table below:

address-setting	default
expiry-queue-prefix	EXP.
expiry-queue-suffix	(empty string)

Here is an example configuration:

```
<address-setting match="#">
   <expiry-address>expiryAddress</expiry-address>
   <auto-create-expiry-resources>true</auto-create-expiry-resources>
   <expiry-queue-prefix></expiry-queue-prefix> <!-- override the default -->
   <expiry-queue-suffix>.EXP</expiry-queue-suffix>
</address-setting>
```

The queue holding the expired messages can be accessed directly either by using the queue's name by itself (e.g. when using the core client) or by using the fully qualified queue name (e.g. when using a JMS client) just like any other queue. Also, note that the queue is auto-created which means it will be auto-deleted as per the relevant `address-settings`.

Configuring The Expiry Reaper Thread

A reaper thread will periodically inspect the queues to check if messages have expired.

The reaper thread can be configured with the following properties in `broker.xml`

- `message-expiry-scan-period`

  How often the queues will be scanned to detect expired messages (in milliseconds, default is 30000ms, set to `-1` to disable the reaper thread)

Example

See the Message Expiration Example which shows how message expiry is configured and used with JMS.

Large Messages

Apache ActiveMQ Artemis can be configured to store messages as files when these messages are beyond a configured value.

Instead of keeping these messages in memory ActiveMQ Artemis will hold just a thin object on the queues with a reference to a file into a specific folder configured as large-messages-directory.

This is supported on Core Protocol and on the AMQP Protocol.

Configuring the server

Large messages are stored on a disk directory on the server side, as configured on the main configuration file.

The configuration property `large-messages-directory` specifies where large messages are stored. For JDBC persistence the `large-message-table` should be configured.

```
<configuration xmlns="urn:activemq"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:activemq /schema/artemis-server.xsd">
   <core xmlns="urn:activemq:core" xmlns:xsi="http://www.w3.org/2001/XMLSchema
      ...
      <large-messages-directory>/data/large-messages</large-messages-directory
      ...
   </core>
</configuration>
```

By default the large message directory is `data/largemessages` and `large-message-table` is configured as "LARGE_MESSAGE_TABLE".

For the best performance we recommend using file store with large messages directory stored on a different physical volume to the message journal or paging directory.

Configuring the Core Client

Any message larger than a certain size is considered a large message. Large messages will be split up and sent in fragments. This is determined by the URL parameter `minLargeMessageSize`

> **Note:**
>
> Apache ActiveMQ Artemis messages are encoded using 2 bytes per character so if the message data is filled with ASCII characters (which are 1 byte) the size of the resulting Apache ActiveMQ Artemis message would roughly double. This is important when calculating the size of a "large" message as it may appear to be less than the `minLargeMessageSize` before it is sent, but it then turns into a "large" message once it is encoded.

The default value is 100KiB.

Configuring the transport directly from the client side will provide more information on how to instantiate the core session factory or JMS connection factory.

Compressed Large Messages on Core Protocol

You can choose to send large messages in compressed form using `compressLargeMessages` URL parameter.

If you specify the boolean URL parameter `compressLargeMessages` as true, The system will use the ZIP algorithm to compress the message body as the message is transferred to the server's side. Notice that there's no special treatment at the server's side, all the compressing and uncompressing is done at the client.

If the compressed size of a large message is below `minLargeMessageSize` , it is sent to server as regular messages. This means that the message won't be written into the server's large-message data directory, thus reducing the disk I/O.

Streaming large messages from Core Protocol

Apache ActiveMQ Artemis supports setting the body of messages using input and output streams (`java.lang.io`)

These streams are then used directly for sending (input streams) and receiving (output streams) messages.

When receiving messages there are 2 ways to deal with the output stream; you may choose to block while the output stream is recovered using the method `ClientMessage.saveOutputStream` or alternatively using the method `ClientMessage.setOutputstream` which will asynchronously write the message to the stream. If you choose the latter the consumer must be kept alive until the message has been fully received.

You can use any kind of stream you like. The most common use case is to send files stored in your disk, but you could also send things like JDBC Blobs, `SocketInputStream` , things you recovered from `HTTPRequests` etc. Anything as long as it implements `java.io.InputStream` for sending messages or `java.io.OutputStream` for receiving them.

Streaming over Core API

The following table shows a list of methods available at `ClientMessage` which are also available through JMS by the use of object properties.

Name	Description	JMS Equivale
setBodyInputStream(InputStream)	Set the InputStream used to read a message body when sending it.	JMS_AMQ_InputSt
setOutputStream(OutputStream)	Set the OutputStream that will receive the body of a message. This method does not block.	JMS_AMQ_Output
saveOutputStream(OutputStream)	Save the body of the message to the `OutputStream`. It will block until the entire content is transferred to the `OutputStream`.	JMS_AMQ_SaveSt

To set the output stream when receiving a core message:

```
ClientMessage msg = consumer.receive(...);

// This will block here until the stream was transferred
msg.saveOutputStream(someOutputStream);

ClientMessage msg2 = consumer.receive(...);

// This will not wait the transfer to finish
msg2.setOutputStream(someOtherOutputStream);
```

Set the input stream when sending a core message:

```
ClientMessage msg = session.createMessage();
msg.setInputStream(dataInputStream);
```

Notice also that for messages with more than 2GiB the getBodySize() will return invalid values since this is an integer (which is also exposed to the JMS API). On those cases you can use the message property _AMQ_LARGE_SIZE.

Streaming over JMS

When using JMS, Apache ActiveMQ Artemis maps the streaming methods on the core API (see ClientMessage API table above) by setting object properties . You can use the method `Message.setObjectProperty` to set the input and output streams.

The `InputStream` can be defined through the JMS Object Property JMS_AMQ_InputStream on messages being sent:

```
BytesMessage message = session.createBytesMessage();

FileInputStream fileInputStream = new FileInputStream(fileInput);

BufferedInputStream bufferedInput = new BufferedInputStream(fileInputStream);

message.setObjectProperty("JMS_AMQ_InputStream", bufferedInput);

someProducer.send(message);
```

The `OutputStream` can be set through the JMS Object Property JMS_AMQ_SaveStream on messages being received in a blocking way.

```
BytesMessage messageReceived = (BytesMessage)messageConsumer.receive(120000);

File outputFile = new File("huge_message_received.dat");

FileOutputStream fileOutputStream = new FileOutputStream(outputFile);

BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutputStrea

// This will block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_AMQ_SaveStream", bufferedOutput);
```

Setting the `OutputStream` could also be done in a non blocking way using the property JMS_AMQ_OutputStream.

```
// This won't wait the stream to finish. You need to keep the consumer active.
messageReceived.setObjectProperty("JMS_AMQ_OutputStream", bufferedOutput);
```

> **Note:**
>
> When using JMS, Streaming large messages are only supported on `StreamMessage` and `BytesMessage` .

Streaming Alternative on Core Protocol

If you choose not to use the `InputStream` or `OutputStream` capability of Apache ActiveMQ Artemis You could still access the data directly in an alternative fashion.

On the Core API just get the bytes of the body as you normally would.

```
ClientMessage msg = consumer.receive();

byte[] bytes = new byte[1024];
for (int i = 0 ;  i < msg.getBodySize(); i += bytes.length)
{
   msg.getBody().readBytes(bytes);
   // Whatever you want to do with the bytes
}
```

If using JMS API, `BytesMessage` and `StreamMessage` also supports it transparently.

```
BytesMessage rm = (BytesMessage)cons.receive(10000);

byte data[] = new byte[1024];

for (int i = 0; i < rm.getBodyLength(); i += 1024)
{
   int numberOfBytes = rm.readBytes(data);
   // Do whatever you want with the data
}
```

Configuring AMQP Acceptor

You can configure the property `amqpMinLargeMessageSize` at the acceptor.

The default value is 102400 (100KBytes).

Setting it to -1 will disable large message support.

Warning: setting amqpMinLargeMessageSize to -1, your AMQP message might be stored as a Core Large Message if the size of the message does not fit into the journal. This is the former semantic of the broker and it is kept this way for compatibility reasons.

```
<acceptors>
      <!-- AMQP Acceptor.  Listens on default AMQP port for AMQP traffic.-->
      <acceptor name="amqp">tcp://0.0.0.0:5672?;   ..... amqpMinLargeMessageSi
</acceptors>
```

Large message example

Please see the Large Message Example which shows how large messages are configured and used with JMS.

Paging

Apache ActiveMQ Artemis transparently supports huge queues containing millions of messages while the server is running with limited memory.

In such a situation it's not possible to store all of the queues in memory at any one time, so Apache ActiveMQ Artemis transparently *pages* messages into and out of memory as they are needed, thus allowing massive queues with a low memory footprint.

Apache ActiveMQ Artemis will start paging messages to disk, when the size of all messages in memory for an address exceeds a configured maximum size.

The default configuration from Artemis has destinations with paging.

Page Files

Messages are stored per address on the file system. Each address has an individual folder where messages are stored in multiple files (page files). Each file will contain messages up to a max configured size (`page-size-bytes`). The system will navigate the files as needed, and it will remove the page file as soon as all the messages are acknowledged up to that point.

Browsers will read through the page-cursor system.

Consumers with selectors will also navigate through the page-files and it will ignore messages that don't match the criteria.

> *Warning:*
>
> When you have a queue, and consumers filtering the queue with a very restrictive selector you may get into a situation where you won't be able to read more data from paging until you consume messages from the queue.
>
> Example: in one consumer you make a selector as 'color="red"' but you only have one color red 1 millions messages after blue, you won't be able to consume red until you consume blue ones.
>
> This is different to browsing as we will "browse" the entire queue looking for messages and while we "depage" messages while feeding the queue.

Configuration

You can configure the location of the paging folder in `broker.xml` .

- `paging-directory` Where page files are stored. Apache ActiveMQ Artemis will create one folder for each address being paged under this configured location. Default is `data/paging` .

Paging Mode

As soon as messages delivered to an address exceed the configured size, that address alone goes into page mode.

> **Note:**
>
> Paging is done individually per address. If you configure a max-size-bytes for an address, that means each matching address will have a maximum size that you specified. It DOES NOT mean that the total overall size of all matching addresses is limited to max-size-bytes.

Configuration

Configuration is done at the address settings in `broker.xml`.

```
<address-settings>
   <address-setting match="jms.someaddress">
      <max-size-bytes>104857600</max-size-bytes>
      <page-size-bytes>10485760</page-size-bytes>
      <address-full-policy>PAGE</address-full-policy>
   </address-setting>
</address-settings>
```

> **Note:** The management-address settings cannot be changed or overridden ie management messages aren't allowed to page/block/fail and are considered an internal broker management mechanism. The memory occupation of the management-address is not considered while evaluating if global-max-size is hit and can't cause other non-management addresses to trigger a configured `address-full-policy`.

This is the list of available parameters on the address settings.

Property Name	Description	Default
`max-size-bytes`	What's the max memory the address could have before entering on page mode.	-1 (disabled)
`page-size-bytes`	The size of each page file used on the paging system	10MB
`address-full-policy`	This must be set to `PAGE` for paging to enable. If the value is `PAGE` then further messages will be paged to disk. If the value is `DROP` then further messages will be silently dropped. If the value is `FAIL` then the messages will be dropped and the client message producers will receive an exception. If the value is `BLOCK` then client message producers will block when they try and send further messages.	`PAGE`
`page-max-cache-size`	The system will keep up to `page-max-cache-size` page files in memory to optimize IO during paging navigation.	5

Global Max Size

Beyond the `max-size-bytes` on the address you can also set the global-max-size on the main configuration. If you set `max-size-bytes` = `-1` on paging the `global-max-size` can still be used.

When you have more messages than what is configured `global-max-size` any new produced message will make that destination to go through its paging policy.

`global-max-size` is calculated as half of the max memory available to the Java Virtual Machine, unless specified on the `broker.xml` configuration.

Dropping messages

Instead of paging messages when the max size is reached, an address can also be configured to just drop messages when the address is full.

To do this just set the `address-full-policy` to `DROP` in the address settings

Dropping messages and throwing an exception to producers

Instead of paging messages when the max size is reached, an address can also be configured to drop messages and also throw an exception on the client-side when the address is full.

To do this just set the `address-full-policy` to `FAIL` in the address settings

Blocking producers

Instead of paging messages when the max size is reached, an address can also be configured to block producers from sending further messages when the address is full, thus preventing the memory being exhausted on the server.

When memory is freed up on the server, producers will automatically unblock and be able to continue sending.

To do this just set the `address-full-policy` to `BLOCK` in the address settings

In the default configuration, all addresses are configured to block producers after 10 MiB of data are in the address.

Caution with Addresses with Multiple Multicast Queues

When a message is routed to an address that has multiple multicast queues bound to it, e.g. a JMS subscription in a Topic, there is only 1 copy of the message in memory. Each queue only deals with a reference to this. Because of this the memory is only freed up once all queues referencing the message have delivered it.

If you have a single lazy subscription, the entire address will suffer IO performance hit as all the queues will have messages being sent through an extra storage on the paging system.

For example:

- An address has 10 multicast queues
- One of the queues does not deliver its messages (maybe because of a slow consumer).
- Messages continually arrive at the address and paging is started.
- The other 9 queues are empty even though messages have been sent.

In this example all the other 9 queues will be consuming messages from the page system. This may cause performance issues if this is an undesirable state.

Max Disk Usage

The System will perform scans on the disk to determine if the disk is beyond a configured limit. These are configured through `max-disk-usage` in percentage. Once that limit is reached any message will be blocked. (unless the protocol doesn't support flow control on which case there will be an exception thrown and the connection for those clients dropped).

Page Sync Timeout

The pages are synced periodically and the sync period is configured through `page-sync-timeout` in nanoseconds. When using NIO journal, by default has the same value of `journal-buffer-timeout`. When using ASYNCIO, the default should be `3333333`.

Example

See the Paging Example which shows how to use paging with Apache ActiveMQ Artemis.

Scheduled Messages

Scheduled messages differ from normal messages in that they won't be delivered until a specified time in the future, at the earliest.

To do this, a special property is set on the message before sending it.

Scheduled Delivery Property

The property name used to identify a scheduled message is `"_AMQ_SCHED_DELIVERY"` (or the constant `Message.HDR_SCHEDULED_DELIVERY_TIME`).

The specified value must be a positive `long` corresponding to the time the message must be delivered (in milliseconds). An example of sending a scheduled message using the JMS API is as follows.

```
TextMessage message = session.createTextMessage("This is a scheduled message m
message.setLongProperty("_AMQ_SCHED_DELIVERY", System.currentTimeMillis() + 50
producer.send(message);

...

// message will not be received immediately but 5 seconds later
TextMessage messageReceived = (TextMessage) consumer.receive();
```

Scheduled messages can also be sent using the core API, by setting the same property on the core message before sending.

Example

See the Scheduled Message Example which shows how scheduled messages can be used with JMS.

Last-Value Queues

Last-Value queues are special queues which discard any messages when a newer message with the same value for a well-defined Last-Value property is put in the queue. In other words, a Last-Value queue only retains the last value.

A typical example for Last-Value queue is for stock prices, where you are only interested by the latest value for a particular stock.

Messages sent to an Last-Value queue without the specified property will be delivered as normal and will never be "replaced".

Configuration

Last Value Key Configuration

Last-Value queues can be statically configured in broker.xml via the `last-value-key`

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" last-value-key="reuters_code" />
   </multicast>
</address>
```

Specified on creating a queue by using the CORE api specifying the parameter `lastValue` to `true`.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?last-value-key=reuters_
Topic topic = session.createTopic("my.destination.name?last-value-key=reuters_
```

Address wildcards can be used to configure Last-Value queues for a set of addresses (see here).

```
<address-setting match="lastValueQueue">
   <default-last-value-key>reuters_code</default-last-value-key>
</address-setting>
```

By default, `default-last-value-key` is null.

Legacy Last Value Configuration

Last-Value queues can also just be configured via the `last-value` boolean property, doing so it will default the last-value-key to `"_AMQ_LVQ_NAME"`.

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" last-value="true" />
   </multicast>
</address>
```

Specified on creating a queue by using the CORE api specifying the parameter `lastValue` to `true`.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?last-value=true");
Topic topic = session.createTopic("my.destination.name?last-value=true");
```

Also the default for all queues under and address can be defaulted using the `address-setting` configuration:

```
<address-setting match="lastValueQueue">
   <default-last-value-queue>true</default-last-value-queue>
</address-setting>
```

By default, `default-last-value-queue` is false.

Note that `address-setting` `last-value-queue` config is deprecated, please use `default-last-value-queue` instead.

Last-Value Property

The property name used to identify the last value is configurable at the queue level mentioned above.

If using the legacy setting to configure an LVQ then the default property `"_AMQ_LVQ_NAME"` is used (or the constant `Message.HDR_LAST_VALUE_NAME` from the Core API).

For example, using the sample configuration

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" last-value-key="reuters_code" />
   </multicast>
</address>
```

if two messages with the same value for the Last-Value property are sent to a Last-Value queue, only the latest message will be kept in the queue:

```
// send 1st message with Last-Value property `reuters_code` set to `VOD`
TextMessage message = session.createTextMessage("1st message with Last-Value p
message.setStringProperty("reuters_code", "VOD");
producer.send(message);

// send 2nd message with Last-Value property `reuters_code` set to `VOD`
message = session.createTextMessage("2nd message with Last-Value property set"
message.setStringProperty("reuters_code", "VOD");
producer.send(message);

...

// only the 2nd message will be received: it is the latest with
// the Last-Value property set
TextMessage messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.format("Received message: %s\n", messageReceived.getText());
```

Forcing all consumers to be non-destructive

When a consumer attaches to a queue, the normal behaviour is that messages are sent to that consumer are acquired exclusively by that consumer, and when the consumer acknowledges them, the messages are removed from the queue.

Another common pattern is to have queue "browsers" which send all messages to the browser, but do not prevent other consumers from receiving the messages, and do not remove them from the queue when the browser is done with them. Such a browser is an instance of a "non-destructive" consumer.

If every consumer on a queue is non destructive then we can obtain some interesting behaviours. In the case of a LVQ then the queue will always contain the most up to date value for every key.

A queue can be created to enforce all consumers are non-destructive for last value queue. This can be be achieved using the following queue configuration:

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" last-value-key="reuters_code" non-destructive="tru
   </multicast>
</address>
```

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?last-value-key=reuters_
Topic topic = session.createTopic("my.destination.name?last-value-key=reuters_
```

Also the default for all queues under and address can be defaulted using the `address-setting` configuration:

```
<address-setting match="lastValueQueue">
   <default-last-value-key>reuters_code</default-last-value-key>
   <default-non-destructive>true</default-non-destructive>
</address-setting>
```

By default, `default-non-destructive` is false.

Bounding size using expiry-delay

For queues other than LVQs, having only non-destructive consumers could mean that messages would never get deleted, leaving the queue to grow unconstrainedly. To prevent this you can use the ability to set a default `expiry-delay`.

See expiry-delay for more details on this.

Example

See the last-value queue example which shows how last value queues are configured and used with JMS.

Ring Queue

Queues operate with first-in, first-out (FIFO) semantics which means that messages, in general, are added to the "tail" of the queue and removed from the "head." A "ring" queue is a special type of queue with a *fixed* size. The fixed size is maintained by removing the message at the head of the queue when the number of messages on the queue reaches the configured size.

For example, consider a queue configured with a ring size of 3 and a producer which sends the messages `A`, `B`, `C`, & `D` in that order. Once `C` is sent the number of messages in the queue will be 3 which is the same as the configured ring size. We can visualize the queue growth like this...

After `A` is sent:

```
             |---|
head/tail -> | A |
             |---|
```

After `B` is sent:

```
        |---|
head -> | A |
        |---|
tail -> | B |
        |---|
```

After `C` is sent:

```
        |---|
head -> | A |
        |---|
        | B |
        |---|
tail -> | C |
        |---|
```

When `D` is sent it will be added to the tail of the queue and the message at the head of the queue (i.e. `A`) will be removed so the queue will look like this:

```
        |---|
head -> | B |
        |---|
        | C |
        |---|
tail -> | D |
        |---|
```

This example covers the most basic use case with messages being added to the tail of the queue. However, there are a few other important use cases involving:

- Messages in delivery & rollbacks

- Scheduled messages
- Paging

However, before we get to those use cases let's look at the basic configuration of a ring queue.

Configuration

There are 2 parameters related to ring queue configuration.

The `ring-size` parameter can be set directly on the `queue` element. The default value comes from the `default-ring-size` `address-setting` (see below).

```
<addresses>
   <address name="myRing">
      <anycast>
         <queue name="myRing" ring-size="3" />
      </anycast>
   </address>
</addresses>
```

The `default-ring-size` is an `address-setting` which applies to queues on matching addresses which don't have an explicit `ring-size` set. This is especially useful for auto-created queues. The default value is `-1` (i.e. no limit).

```
<address-settings>
   <address-setting match="ring.#">
      <default-ring-size>3</default-ring-size>
   </address-setting>
</address-settings>
```

The `ring-size` may be updated at runtime. If the new `ring-size` is set *lower* than the previous `ring-size` the broker will not immediately delete enough messages from the head of the queue to enforce the new size. New messages sent to the queue will force the deletion of old messages (i.e. the queue won't grow any larger), but the queue will not reach its new size until it does so *naturally* through the normal consumption of messages by clients.

Messages in Delivery & Rollbacks

When messages are "in delivery" they are in an in-between state where they are not technically on the queue but they are also not yet acknowledged. The broker is at the consumer's mercy to either acknowledge such messages or not. In the context of a ring queue, messages which are in-delivery cannot be removed from the queue.

This presents a few dilemmas.

Due to the nature of messages in delivery a client can actually send more messages to a ring queue than it would otherwise permit. This can make it appear that the ring-size is not being enforced properly. Consider this simple scenario:

- Queue `foo` with `ring-size="3"`
- 1 Consumer on queue `foo`
- Message `A` sent to `foo` & dispatched to consumer
- `messageCount` =1, `deliveringCount` =1
- Message `B` sent to `foo` & dispatched to consumer
- `messageCount` =2, `deliveringCount` =2
- Message `C` sent to `foo` & dispatched to consumer
- `messageCount` =3, `deliveringCount` =3
- Message `D` sent to `foo` & dispatched to consumer
- `messageCount` =4, `deliveringCount` =4

The `messageCount` for `foo` is now 4, one *greater* than the `ring-size` of 3! However, the broker has no choice but to allow this because it cannot remove messages from the queue which are in delivery.

Now consider that the consumer is closed without actually acknowledging any of these 4 messages. These 4 in-delivery, unacknowledged messages will be cancelled back to the broker and added to the *head* of the queue in the reverse order from which they were consumed. This, of course, will put the queue over its configured `ring-size`. Therefore, since a ring queue prefers messages at the tail of the queue over messages at the head it will keep `B`, `C`, & `D` and delete `A` (since `A` was the last message added to the head of the queue).

Transaction or core session rollbacks are treated the same way.

If you wish to avoid these kinds of situations and you're using the core client directly or the core JMS client you can minimize messages in delivery by reducing the size of `consumerWindowSize` (1024 * 1024 bytes by default).

Scheduled Messages

When a scheduled message is sent to a queue it isn't immediately added to the tail of the queue like normal messages. It is held in an intermediate buffer and scheduled for delivery onto the *head* of the queue according to the details of the message. However, scheduled messages are nevertheless reflected in the message count of the queue. As with messages which are in delivery this can make it appear that the ring queue's size is not being enforced. Consider this simple scenario:

- Queue `foo` with `ring-size="3"`
- At 12:00 message `A` sent to `foo` scheduled for 12:05
- `messageCount` =1, `scheduledCount` =1
- At 12:01 message `B` sent to `foo`
- `messageCount` =2, `scheduledCount` =1
- At 12:02 message `C` sent to `foo`
- `messageCount` =3, `scheduledCount` =1
- At 12:03 message `D` sent to `foo`
- `messageCount` =4, `scheduledCount` =1

The `messageCount` for `foo` is now 4, one *greater* than the `ring-size` of 3! However, the scheduled message is not technically on the queue yet (i.e. it is on the broker and scheduled to be put on the queue). When the scheduled delivery time for 12:05 comes the message will put on the head of the queue, but since the ring queue's size has already been reach the scheduled message `A` will be removed.

Paging

Similar to scheduled messages and messages in delivery, paged messages don't count against a ring queue's size because messages are actually paged at the *address* level, not the queue level. A paged message is not technically on a queue although it is reflected in a queue's `messageCount`.

It is recommended that paging is not used for addresses with ring queues. In other words, ensure that the entire address will be able to fit into memory or use the `DROP`, `BLOCK` or `FAIL` `address-full-policy`.

Retroactive Addresses

A "retroactive" address is an address that will preserve messages sent to it for queues which will be created on it in the future. This can be useful in, for example, publish-subscribe use cases where clients want to receive the messages sent to the address *before* they they actually connected and created their multicast "subscription" queue. Typically messages sent to an address before a queue was created on it would simply be unavailable to those queues, but with a retroactive address a fixed number of messages can be preserved by the broker and automatically copied into queues subsequently created on the address. This works for both anycast and multicast queues.

Internal Retroactive Resources

To implement this functionality the broker will create 4 internal resources for each retroactive address:

1. A non-exclusive divert to grab the messages from the retroactive address.
2. An address to receive the messages from the divert.
3. **Two** ring queues to hold the messages sent to the address by the divert - one for anycast and one for multicast. The general caveats for ring queues still apply here. See the chapter on ring queues for more details.

These resources are important to be aware of as they will show up in the web console and other management or metric views. They will be named according to the following pattern:

```
<internal-naming-prefix><delimiter><source-address><delimiter>(divert|address|
```

For example, if an address named `myAddress` had a `retroactive-message-count` of 10 and the default `internal-naming-prefix` (i.e. `$.artemis.internal.`) and the default delimiter (i.e. `.`) were being used then resources with these names would be created:

1. A divert on `myAddress` named `$.artemis.internal.myAddress.divert.retro`
2. An address named `$.artemis.internal.myAddress.address.retro`
3. A multicast queue on the address from step #2 named `$.artemis.internal.myAddress.queue.multicast.retro` with a `ring-size` of 10.
4. An anycast queue on the address from step #2 named `$.artemis.internal.myAddress.queue.anycast.retro` with a `ring-size` of 10.

This pattern is important to note as it allows one to configure address-settings if necessary. To configure custom address-settings you'd use a match like:

```
*.*.*.<source-address>.*.retro
```

Using the same example as above the `match` would be:

```
*.*.*.myAddress.*.retro
```

> Note:
>
> Changing the broker's `internal-naming-prefix` once these retroactive resources are created will break the retroactive functionality.

Configuration

To configure an address to be "retroactive" simply configure the `retroactive-message-count` `address-setting` to reflect the number of messages you want the broker to preserve, e.g.:

```
<address-settings>
   <address-setting match="orders">
      <retroactive-message-count>100</retroactive-message-count>
   </address-setting>
</address-settings>
```

The value for `retroactive-message-count` can be updated at runtime either via `broker.xml` or via the management API just like any other address-setting. However, if you *reduce* the value of `retroactive-message-count` an additional administrative step will be required since this functionality is implemented via ring queues. This is because a ring queue whose ring-size is reduced will not automatically delete messages from the queue to meet the new ring-size in order to avoid unintended message loss. Therefore, administrative action will be required in this case to manually reduce the number of messages in the ring queue via the management API.

Exclusive Queues

Exclusive queues are special queues which route all messages to only one consumer at a time.

This is useful when you want all messages to be processed serially by the same consumer, when a producer does not specify Message Grouping.

An example might be orders sent to an address and you need to consume them in the exact same order they were produced.

Obviously exclusive queues have a draw back that you cannot scale out the consumers to improve consumption as only one consumer would technically be active. Here we advise that you look at message groups first.

Configuring Exclusive Queues

Exclusive queues can be statically configured using the `exclusive` boolean property:

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" exclusive="true"/>
   </multicast>
</address>
```

Specified on creating a Queue by using the CORE api specifying the parameter `exclusive` to `true`.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?exclusive=true");
Topic topic = session.createTopic("my.destination.name?exclusive=true");
```

Also the default for all queues under and address can be defaulted using the `address-setting` configuration:

```
<address-setting match="lastValueQueue">
   <default-exclusive-queue>true</default-exclusive-queue>
</address-setting>
```

By default, `default-exclusive-queue` is `false`. Address wildcards can be used to configure exclusive queues for a set of addresses.

Example

See the exclusive queue example which shows how exclusive queues are configured and used with JMS.

Message Grouping

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group id, i.e. they have same group identifier property (`JMSXGroupID` for JMS, `_AMQ_GROUP_ID` for Apache ActiveMQ Artemis Core API).
- Messages in a message group are always consumed by the same consumer, even if there are many consumers on a queue. They pin all messages with the same group id to the same consumer. If that consumer closes another consumer is chosen and will receive all messages with the same group id.

Message groups are useful when you want all messages for a certain value of the property to be processed serially by the same consumer.

An example might be orders for a certain stock. You may want orders for any particular stock to be processed serially by the same consumer. To do this you can create a pool of consumers (perhaps one for each stock, but less will work too), then set the stock name as the value of the _AMQ_GROUP_ID property.

This will ensure that all messages for a particular stock will always be processed by the same consumer.

> **Note:**
>
> Grouped messages can impact the concurrent processing of non-grouped messages due to the underlying FIFO semantics of a queue. For example, if there is a chunk of 100 grouped messages at the head of a queue followed by 1,000 non-grouped messages then all the grouped messages will need to be sent to the appropriate client (which is consuming those grouped messages serially) before any of the non-grouped messages can be consumed. The functional impact in this scenario is a temporary suspension of concurrent message processing while all the grouped messages are processed. This can be a performance bottleneck so keep it in mind when determining the size of your message groups, and consider whether or not you should isolate your grouped messages from your non-grouped messages.

Using Core API

The property name used to identify the message group is `"_AMQ_GROUP_ID"` (or the constant `MessageImpl.HDR_GROUP_ID`). Alternatively, you can set `autogroup` to true on the `SessionFactory` which will pick a random unique id.

Using JMS

The property name used to identify the message group is `JMSXGroupID` .

```
// send 2 messages in the same group to ensure the same
// consumer will receive both
Message message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);

message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);
```

Alternatively, you can set `autogroup` to true on the `ActiveMQConnectonFactory` which will pick a random unique id. This can also be set in the JNDI context environment, e.g. `jndi.properties`. Here's a simple example using the "ConnectionFactory" connection factory which is available in the context by default

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
connectionFactory.myConnectionFactory=tcp://localhost:61616?autoGroup=true
```

Alternatively you can set the group id via the connection factory. All messages sent with producers created via this connection factory will set the `JMSXGroupID` to the specified value on all messages sent. This can also be set in the JNDI context environment, e.g. `jndi.properties`. Here's a simple example using the "ConnectionFactory" connection factory which is available in the context by default:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialCo
connectionFactory.myConnectionFactory=tcp://localhost:61616?groupID=Group-0
```

Closing a Message Group

You generally don't need to close a message group, you just keep using it.

However if you really do want to close a group you can add a negative sequence number.

Example:

```
Mesasge message = session.createTextMessage("<foo>hey</foo>");
message.setStringProperty("JMSXGroupID", "Group-0");
message.setIntProperty("JMSXGroupSeq", -1);
...
producer.send(message);
```

This then closes the message group so if another message is sent in the future with the same message group ID it will be reassigned to a new consumer.

Notifying Consumer of Group Ownership change

ActiveMQ supports putting a boolean header, set on the first message sent to a consumer for a particular message group.

To enable this, you must set a header key that the broker will use to set the flag.

In the examples we use `JMSXGroupFirstForConsumer` but it can be any header key value you want.

By setting `group-first-key` to `JMSXGroupFirstForConsumer` at the queue level, every time a new group is assigned a consumer the header `JMSXGroupFirstForConsumer` will be set to true on the first message.

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" group-first-key="JMSXGroupFirstForConsumer"/>
   </multicast>
</address>
```

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?group-first-key=JMSXGro
Topic topic = session.createTopic("my.destination.name?group-first-key=JMSXGro
```

Also the default for all queues under and address can be defaulted using the `address-setting` configuration:

```
<address-setting match="my.address">
   <default-group-first-key>JMSXGroupFirstForConsumer</default-group-first-key
</address-setting>
```

By default this is null, and therefor OFF.

Rebalancing Message Groups

Sometimes after new consumers are added you can find that if you have long lived groups, that they have no groups assigned, and thus are not being utilised, this is because the long lived groups will already be assigned to existing consumers.

It is possibly to rebalance the groups.

note during the split moment of reset, a message to the original associated consumer could be in flight at the same time, a new message for the same group is dispatched to the new associated consumer.

Manually

via the management API or managment console by invoking `resetAllGroups`

Automatically

By setting `group-rebalance` to `true` at the queue level, every time a consumer is added it will trigger a rebalance/reset of the groups.

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" group-rebalance="true"/>
   </multicast>
</address>
```

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?group-rebalance=true");
Topic topic = session.createTopic("my.destination.name?group-rebalance=true");
```

Also the default for all queues under and address can be defaulted using the `address-setting` configuration:

```
<address-setting match="my.address">
   <default-group-rebalance>true</default-group-rebalance>
</address-setting>
```

By default, `default-group-rebalance` is `false` meaning this is disabled/off.

Group Buckets

For handling groups in a queue with bounded memory allowing better scaling of groups, you can enable group buckets, essentially the group id is hashed into a bucket instead of keeping track of every single group id.

Setting `group-buckets` to `-1` keeps default behaviour which means the queue keeps track of every group but suffers from unbounded memory use.

Setting `group-buckets` to `0` disables grouping (0 buckets), on a queue. This can be useful on a multicast address, where many queues exist but one queue you may not care for ordering and prefer to keep round robin behaviour.

There is a number of ways to set `group-buckets`.

```
<address name="foo.bar">
   <multicast>
      <queue name="orders1" group-buckets="1024"/>
   </multicast>
</address>
```

Specified on creating a Queue by using the CORE api specifying the parameter `group-buckets` to `20`.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?group-buckets=1024");
Topic topic = session.createTopic("my.destination.name?group-buckets=1024");
```

Also the default for all queues under and address can be defaulted using the `address-setting` configuration:

```
<address-setting match="my.bucket.address">
   <default-group-buckets>1024</default-group-buckets>
</address-setting>
```

By default, `default-group-buckets` is `-1` this is to keep compatibility with existing default behaviour.

Address wildcards can be used to configure group-buckets for a set of addresses.

Example

See the Message Group Example which shows how message groups are configured and used with JMS and via a connection factory.

Clustered Grouping

Using message groups in a cluster is a bit more complex. This is because messages with a particular group id can arrive on any node so each node needs to know about which group id's are bound to which consumer on which node. The consumer handling messages for a particular group id may be on a different node of the cluster, so each node needs to know this information so it can route the message correctly to the node which has that consumer.

To solve this there is the notion of a grouping handler. Each node will have its own grouping handler and when a messages is sent with a group id assigned, the handlers will decide between them which route the message should take.

Here is a sample config for each type of handler. This should be configured in `broker.xml`.

```
<grouping-handler name="my-grouping-handler">
   <type>LOCAL</type>
   <address>jms</address>
   <timeout>5000</timeout>
</grouping-handler>

<grouping-handler name="my-grouping-handler">
   <type>REMOTE</type>
   <address>jms</address>
   <timeout>5000</timeout>
</grouping-handler>
```

- `type` two types of handlers are supported - `LOCAL` and `REMOTE`. Each cluster should choose 1 node to have a `LOCAL` grouping handler and all the other nodes should have `REMOTE` handlers. It's the `LOCAL` handler that actually makes the decision as to what route should be used, all the other `REMOTE` handlers converse with this.
- `address` refers to a cluster connection and the address it uses. Refer to the clustering section on how to configure clusters.

- `timeout` how long to wait for a decision to be made. An exception will be thrown during the send if this timeout is reached, this ensures that strict ordering is kept.

The decision as to where a message should be routed to is initially proposed by the node that receives the message. The node will pick a suitable route as per the normal clustered routing conditions, i.e. round robin available queues, use a local queue first and choose a queue that has a consumer. If the proposal is accepted by the grouping handlers the node will route messages to this queue from that point on, if rejected an alternative route will be offered and the node will again route to that queue indefinitely. All other nodes will also route to the queue chosen at proposal time. Once the message arrives at the queue then normal single server message group semantics take over and the message is pinned to a consumer on that queue.

You may have noticed that there is a single point of failure with the single local handler. If this node crashes then no decisions will be able to be made. Any messages sent will be not be delivered and an exception thrown. To avoid this happening Local Handlers can be replicated on another backup node. Simple create your back up node and configure it with the same Local handler.

Clustered Grouping Best Practices

Some best practices should be followed when using clustered grouping:

1. Make sure your consumers are distributed evenly across the different nodes if possible. This is only an issue if you are creating and closing consumers regularly. Since messages are always routed to the same queue once pinned, removing a consumer from this queue may leave it with no consumers meaning the queue will just keep receiving the messages. Avoid closing consumers or make sure that you always have plenty of consumers, i.e., if you have 3 nodes have 3 consumers.
2. Use durable queues if possible. If queues are removed once a group is bound to it, then it is possible that other nodes may still try to route messages to it. This can be avoided by making sure that the queue is deleted by the session that is sending the messages. This means that when the next message is sent it is sent to the node where the queue was deleted meaning a new proposal can successfully take place. Alternatively you could just start using a different group id.
3. Always make sure that the node that has the Local Grouping Handler is replicated. These means that on failover grouping will still occur.
4. In case you are using group-timeouts, the remote node should have a smaller group-timeout with at least half of the value on the main coordinator. This is because this will determine how often the last-time-use value should be updated with a round trip for a request to the group between the nodes.

Clustered Grouping Example

See the Clustered Grouping Example which shows how to configure message groups with a ActiveMQ Artemis Cluster.

Consumer Priority

Consumer priorities allow you to ensure that high priority consumers receive messages while they are active.

Normally, active consumers connected to a queue receive messages from it in a round-robin fashion. When consumer priorities are in use, messages are delivered round-robin if multiple active consumers exist with the same high priority.

Messages will only going to lower priority consumers when the high priority consumers do not have credit available to consume the message, or those high priority consumers have declined to accept the message (for instance because it does not meet the criteria of any selectors associated with the consumer).

Where a consumer does not set, the default priority **0** is used.

Core

JMS Example

When using the JMS Client you can set the priority to be used, by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?consumer-priority=50");
Topic topic = session.createTopic("my.destination.name?consumer-priority=50");

consumer = session.createConsumer(queue);
```

The range of priority values is -2^{31} to $2^{31}-1$.

OpenWire

JMS Example

The priority for a consumer is set using Destination Options as follows:

```
queue = new ActiveMQQueue("TEST.QUEUE?consumer.priority=10");
consumer = session.createConsumer(queue);
```

Because of the limitation of OpenWire, the range of priority values is: 0 to 127. The highest priority is 127.

AMQP

In AMQP 1.0 the priority of the consumer is set in the properties map of the attach frame where the broker side of the link represents the sending side of the link.

The key for the entry must be the literal string priority, and the value of the entry must be an integral number in the range -2^{31} to $2^{31}-1$.

Extra Acknowledge Modes

JMS specifies 3 acknowledgement modes:

- `AUTO_ACKNOWLEDGE`
- `CLIENT_ACKNOWLEDGE`
- `DUPS_OK_ACKNOWLEDGE`

Apache ActiveMQ Artemis supports two additional modes: `PRE_ACKNOWLEDGE` and `INDIVIDUAL_ACKNOWLEDGE`

In some cases you can afford to lose messages in event of failure, so it would make sense to acknowledge the message on the server *before* delivering it to the client.

This extra mode is supported by Apache ActiveMQ Artemis and will call it *pre-acknowledge* mode.

The disadvantage of acknowledging on the server before delivery is that the message will be lost if the system crashes *after* acknowledging the message on the server but *before* it is delivered to the client. In that case, the message is lost and will not be recovered when the system restart.

Depending on your messaging case, `preAcknowledgement` mode can avoid extra network traffic and CPU at the cost of coping with message loss.

An example of a use case for pre-acknowledgement is for stock price update messages. With these messages it might be reasonable to lose a message in event of crash, since the next price update message will arrive soon, overriding the previous price.

> **Note:**
>
> Please note, that if you use pre-acknowledge mode, then you will lose transactional semantics for messages being consumed, since clearly they are being acknowledged first on the server, not when you commit the transaction. This may be stating the obvious but we like to be clear on these things to avoid confusion!

Using PRE_ACKNOWLEDGE

This can be configured by setting the boolean URL parameter `preAcknowledge` to `true`.

Alternatively, when using the JMS API, create a JMS Session with the `ActiveMQSession.PRE_ACKNOWLEDGE` constant.

```
// messages will be acknowledge on the server *before* being delivered to the
Session session = connection.createSession(false, ActiveMQJMSConstants.PRE_ACK
```

Individual Acknowledge

A valid use-case for individual acknowledgement would be when you need to have your own scheduling and you don't know when your message processing will be finished. You should prefer having one consumer per thread worker but this is not possible in some circumstances depending on how complex is your processing. For that you can use the individual acknowledgement.

You basically setup Individual ACK by creating a session with the acknowledge mode with `ActiveMQJMSConstants.INDIVIDUAL_ACKNOWLEDGE`. Individual ACK inherits all the semantics from Client Acknowledge, with the exception the message is individually acked.

> **Note:**
>
> Please note, that to avoid confusion on MDB processing, Individual ACKNOWLEDGE is not supported through MDBs (or the inbound resource adapter). this is because you have to finish the process of your message inside the MDB.

Example

See the Pre-acknowledge Example which shows how to use pre-acknowledgement mode with JMS.

Management

Apache ActiveMQ Artemis has an extensive *management API* that allows a user to modify a server configuration, create new resources (e.g. addresses and queues), inspect these resources (e.g. how many messages are currently held in a queue) and interact with it (e.g. to remove messages from a queue). Apache ActiveMQ Artemis also allows clients to subscribe to management notifications.

There are four ways to access Apache ActiveMQ Artemis management API:

- Using JMX -- *JMX* is the standard way to manage Java applications
- Using Jolokia -- Jolokia exposes the JMX API of an application through a *REST interface*
- Using the Core Client -- management operations are sent to Apache ActiveMQ Artemis server using *Core Client messages*
- Using any JMS Client -- management operations are sent to Apache ActiveMQ Artemis server using *JMS Client messages*

Although there are four different ways to manage Apache ActiveMQ Artemis, each API supports the same functionality. If it is possible to manage a resource using JMX it is also possible to achieve the same result using Core messages.

Besides these four management interfaces, a Web Console and a Command Line *management utility* are also available to administrators of ActiveMQ Artemis.

The choice depends on your requirements, your application settings, and your environment to decide which way suits you best.

> **Note:**
>
> In version 2 of Apache ActiveMQ Artemis the syntax used for MBean Object names has changed significantly due to changes in the addressing scheme. See the documentation for each individual resource for details on the new syntax.

The Management API

Regardless of the way you *invoke* management operations, the management API is the same.

For each *managed resource*, there exists a Java interface describing what operations can be invoked for this type of resource.

To learn about available *management operations*, see the Javadoc for these interfaces. They are located in the `org.apache.activemq.artemis.api.core.management` package and they are named with the word `Control` at the end.

The way to invoke management operations depends on whether JMX, Core messages, or JMS messages are used.

Management API

For full details of the API please consult the Javadoc. In summary:

Server Management

The `ActiveMQServerControl` interface is the entry point for broker management.

- Listing, creating, deploying and destroying queues

  A list of deployed queues can be retrieved using the `getQueueNames()` method.

  Queues can be created or destroyed using the management operations `createQueue()` or `deployQueue()` or `destroyQueue()`.

  `createQueue` will fail if the queue already exists while `deployQueue` will do nothing.

- Listing and closing remote connections

  Client's remote addresses can be retrieved using `listRemoteAddresses()`. It is also possible to close the connections associated with a remote address using the `closeConnectionsForAddress()` method.

  Alternatively, connection IDs can be listed using `listConnectionIDs()` and all the sessions for a given connection ID can be listed using `listSessions()`.

- Transaction heuristic operations

  In case of a server crash, when the server restarts, it it possible that some transaction requires manual intervention. The `listPreparedTransactions()` method lists the transactions which are in the prepared states (the transactions are represented as opaque Base64 Strings.) To commit or rollback a given prepared transaction, the `commitPreparedTransaction()` or `rollbackPreparedTransaction()` method can be used to resolve heuristic transactions. Heuristically completed transactions can be listed using the `listHeuristicCommittedTransactions()` and `listHeuristicRolledBackTransactions` methods.

- Enabling and resetting Message counters

  Message counters can be enabled or disabled using the `enableMessageCounters()` or `disableMessageCounters()` method. To reset message counters, it is possible to invoke `resetAllMessageCounters()` and `resetAllMessageCounterHistories()` methods.

- Retrieving the server configuration and attributes

  The `ActiveMQServerControl` exposes Apache ActiveMQ Artemis server configuration through all its attributes (e.g. `getVersion()` method to retrieve the server's version, etc.)

- Listing, creating and destroying Core bridges and diverts

  A list of deployed core bridges (resp. diverts) can be retrieved using the `getBridgeNames()` (resp. `getDivertNames()`) method.

  Core bridges (resp. diverts) can be created or destroyed using the management operations `createBridge()` and `destroyBridge()` (resp. `createDivert()` and `destroyDivert()`).

  Diverts can be updated using the management operation `updateDivert()` .

- It is possible to stop the server and force failover to occur with any currently attached clients.

  To do this use the `forceFailover()` operation.

  > **Note:**
  >
  > Since this method actually stops the server you will probably receive some sort of error depending on which management service you use to call it.

Address Management

Individual addresses can be managed using the `AddressControl` interface.

- Modifying roles and permissions for an address

  You can add or remove roles associated to a queue using the `addRole()` or `removeRole()` methods. You can list all the roles associated to the queue with the `getRoles()` method

- Pausing and resuming Address

  The `AddressControl` can pause and resume an address and all the queues that are bound to it. Newly added queue will be paused too until the address is resumed. Thus all messages sent to the address will be recived but not delivered. When it is resumed, delivering will occur again.

Queue Management

The bulk of the management API deals with queues. The `QueueControl` interface defines the queue management operations.

Most of the management operations on queues take either a single message ID (e.g. to remove a single message) or a filter (e.g. to expire all messages with a given property.)

> **Note:**
>
> Passing `null` or an empty string in the `filter` parameter means that the management operation will be performed on *all messages* in a queue.

- Expiring, sending to a dead letter address and moving messages

Messages can be expired from a queue by using the `expireMessages()` method. If an expiry address is defined, messages will be sent to it, otherwise they are discarded.

Messages can also be sent to a dead letter address with the `sendMessagesToDeadLetterAddress()` method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, message are removed from the queue and discarded.

Messages can also be moved from a queue to another queue by using the `moveMessages()` method.

- Listing and removing messages

  Messages can be listed from a queue by using the `listMessages()` method which returns an array of `Map`, one `Map` for each message.

  Messages can also be removed from the queue by using the `removeMessages()` method which returns a `boolean` for the single message ID variant or the number of removed messages for the filter variant. The `removeMessages()` method takes a `filter` argument to remove only filtered messages. Setting the filter to an empty string will in effect remove all messages.

- Counting messages

  The number of messages in a queue is returned by the `getMessageCount()` method. Alternatively, the `countMessages()` will return the number of messages in the queue which *match a given filter*.

- Changing message priority

  The message priority can be changed by using the `changeMessagesPriority()` method which returns a `boolean` for the single message ID variant or the number of updated messages for the filter variant.

- Message counters

  Message counters can be listed for a queue with the `listMessageCounter()` and `listMessageCounterHistory()` methods (see Message Counters section). The message counters can also be reset for a single queue using the `resetMessageCounter()` method.

- Retrieving the queue attributes

  The `QueueControl` exposes queue settings through its attributes (e.g. `getFilter()` to retrieve the queue's filter if it was created with one, `isDurable()` to know whether the queue is durable or not, etc.)

- Pausing and resuming Queues

  The `QueueControl` can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it's resumed, it'll begin delivering the queued messages, if any.

- Disabling and Enabling Queues

The `QueueControl` can disable and enable the underlying queue. When a queue is disabled, it will not longer have messages routed to it. When it's enabled, it'll begin having messages routed to it again.

This is useful where you may need to disable message routing to a queue but wish to keep consumers active to investigate issues, without causing further message build up in the queue.

Other Resources Management

Apache ActiveMQ Artemis allows to start and stop its remote resources (acceptors, diverts, bridges, etc.) so that a server can be taken off line for a given period of time without stopping it completely (e.g. if other management operations must be performed such as resolving heuristic transactions). These resources are:

- Acceptors

  They can be started or stopped using the `start()` or. `stop()` method on the `AcceptorControl` interface. The acceptors parameters can be retrieved using the `AcceptorControl` attributes (see Understanding Acceptors)

- Diverts

  They can be started or stopped using the `start()` or `stop()` method on the `DivertControl` interface. Diverts parameters can be retrieved using the `DivertControl` attributes (see Diverting and Splitting Message Flows))

- Bridges

  They can be started or stopped using the `start()` (resp. `stop()`) method on the `BridgeControl` interface. Bridges parameters can be retrieved using the `BridgeControl` attributes (see Core bridges)

- Broadcast groups

  They can be started or stopped using the `start()` or `stop()` method on the `BroadcastGroupControl` interface. Broadcast groups parameters can be retrieved using the `BroadcastGroupControl` attributes (see Clusters)

- Cluster connections

  They can be started or stopped using the `start()` or `stop()` method on the `ClusterConnectionControl` interface. Cluster connections parameters can be retrieved using the `ClusterConnectionControl` attributes (see Clusters)

Using Management Via JMX

Apache ActiveMQ Artemis can be managed using JMX.

The management API is exposed by Apache ActiveMQ Artemis using MBeans interfaces. Apache ActiveMQ Artemis registers its resources with the domain `org.apache.activemq.artemis` .

For example, the `ObjectName` to manage the anycast queue `exampleQueue` on the address `exampleAddress` is:

```
org.apache.activemq.artemis:broker=<brokerName>,component=addresses,address="e
```

and the MBean is:

```
org.apache.activemq.artemis.api.core.management.QueueControl
```

The MBean `ObjectName` 's are built using the helper class `org.apache.activemq.artemis.api.core.management.ObjectNameBuilder` . You can also use `jconsole` to find the `ObjectName` of the MBean you want to manage.

Example usage of the `ObjectNameBuilder` to obtain `ActiveMQServerControl` 's name:

```
brokerName = "0.0.0.0";  // configured e.g. in broker.xml <broker-name> elemen
objectNameBuilder = ObjectNameBuilder.create(ArtemisResolver.DEFAULT_DOMAIN, b
serverObjectName = objectNameBuilder.getActiveMQServerObjectName()
```

Managing Apache ActiveMQ Artemis using JMX is identical to management of any Java Applications using JMX. It can be done by reflection or by creating proxies of the MBeans.

Configuring JMX

By default, JMX is enabled to manage Apache ActiveMQ Artemis. It can be disabled by setting `jmx-management-enabled` to `false` in `broker.xml` :

```
<!-- false to disable JMX management for Apache ActiveMQ Artemis -->
<jmx-management-enabled>false</jmx-management-enabled>
```

Role Based Authorisation for JMX

Although by default Artemis uses the Java Virtual Machine's `Platform MBeanServer` this is guarded using role based authentication that leverages Artemis's JAAS plugin support. This is configured via the `authorisation` element in the `management.xml` configuration file and can be used to restrict access to attributes and methods on mbeans.

There are 3 elements within the `authorisation` element, `whitelist` , `default-access` and `role-access` , Lets discuss each in turn.

Whitelist contains a list of mBeans that will by pass the authentication, this is typically used for any mbeans that are needed by the console to run etc. The default configuration is:

```
<whitelist>
   <entry domain="hawtio"/>
</whitelist>
```

This means that any mbean with the domain `hawtio` will be allowed access without authorisation. for instance `hawtio:plugin=artemis`. You can also use wildcards for the mBean properties so the following would also match.

```
<whitelist>
   <entry domain="hawtio" key="type=*"/>
</whitelist>
```

The `role-access` defines how roles are mapped to particular mBeans and its attributes and methods, the default configuration looks like:

```
<role-access>
  <match domain="org.apache.activemq.artemis">
     <access method="list*" roles="view,update,amq"/>
     <access method="get*" roles="view,update,amq"/>
     <access method="is*" roles="view,update,amq"/>
     <access method="set*" roles="update,amq"/>
     <access method="*" roles="amq"/>
  </match>
</role-access>
```

This contains 1 match and will be applied to any mBean that has the domain `org.apache.activemq.artemis`. Any access to any mBeans that have this domain are controlled by the `access` elements which contain a method and a set of roles. The method being invoked will be used to pick the closest matching method and the roles for this will be applied for access. For instance if you try the invoke a method called `listMessages` on an mBean with the `org.apache.activemq.artemis` domain then this would match the `access` with the method of `list*`. You could also explicitly configure this by using the full method name, like so:

```
<access method="listMessages" roles="view,update,amq"/>
```

You can also match specific mBeans within a domain by adding a key attribute that is used to match one of the properties on the mBean, like:

```
<match domain="org.apache.activemq.artemis" key="subcomponent=queues">
   <access method="list*" roles="view,update,amq"/>
   <access method="get*" roles="view,update,amq"/>
   <access method="is*" roles="view,update,amq"/>
   <access method="set*" roles="update,amq"/>
   <access method="*" roles="amq"/>
</match>
```

You could also match a specific queue for instance:

```
org.apache.activemq.artemis:broker=<brokerName>,component=addresses,address="e
```

by configuring:

```
<match domain="org.apache.activemq.artemis" key="queue=exampleQueue">
   <access method="list*" roles="view,update,amq"/>
   <access method="get*" roles="view,update,amq"/>
   <access method="is*" roles="view,update,amq"/>
   <access method="set*" roles="update,amq"/>
   <access method="*" roles="amq"/>
</match>
```

You can also use wildcards for the mBean properties so the following would also match, allowing prefix match for the mBean properties.

```
<match domain="org.apache.activemq.artemis" key="queue=example*">
   <access method="list*" roles="view,update,amq"/>
   <access method="get*" roles="view,update,amq"/>
   <access method="is*" roles="view,update,amq"/>
   <access method="set*" roles="update,amq"/>
   <access method="*" roles="amq"/>
</match>
```

In case of multiple matches, the exact matches have higher priority than the wildcard matches and the longer wildcard matches have higher priority than the shorter wildcard matches.

Access to JMX mBean attributes are converted to method calls so these are controlled via the `set*`, `get*` and `is*`. The `*` access is the catch all for everything other method that isn't specifically matched.

The `default-access` element is basically the catch all for every method call that isn't handled via the `role-access` configuration. This has the same semantics as a `match` element.

> **Note:**
>
> If JMX is enabled, Apache ActiveMQ Artemis can *not* be managed locally using `jconsole` when connecting as a local process, this is because jconsole does not using any authentication when connecting this way. If you want to use jconsole you will either have to disable authentication, by removing the `authentication` element or enable remote access.

Configuring remote JMX Access

By default remote JMX access to Artemis is disabled for security reasons.

Artemis has a JMX agent which allows access to JMX mBeans remotely. This is configured via the `connector` element in the `management.xml` configuration file. To enable this you simply add the following xml:

```
<connector connector-port="1099"/>
```

This exposes the agent remotely on the port 1099. If you were connecting via jconsole you would connect as a remote process using the service url `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi` and an appropriate user name and password.

You can also configure the connector using the following:

- `connector-host`

  The host to expose the agent on.

- `connector-port`

  The port to expose the agent on.

- `rmi-registry-port`

  The port that the RMI registry binds to. If not set, the port is always random. Set to avoid problems with remote JMX connections tunnelled through firewall.

- `jmx-realm`

  The jmx realm to use for authentication, defaults to `activemq` to match the JAAS configuration.

- `object-name`

  The object name to expose the remote connector on; default is `connector:name=rmi` .

- `secured`

  Whether the connector is secured using SSL.

- `key-store-path`

  The location of the keystore.

- `key-store-password`

  The keystore password. This can be masked.

- `key-store-provider`

  The provider; `JKS` by default.

- `trust-store-path`

  The location of the truststore.

- `trust-store-password`

  The trustore password. This can be masked.

- `trust-store-provider`

  The provider; `JKS` by default.

- `password-codec`

  The fully qualified class name of the password codec to use. See the password masking documentation for more details on how this works.

> **Note:**
>
> It is important to note that the rmi registry will pick an ip address to bind to, If you have a multi IP addresses/NICs present on the system then you can choose the ip address to use by adding the following to artemis.profile `-Djava.rmi.server.hostname=localhost`
>
> **Note:**
>
> Remote connections using the default JVM Agent not enabled by default as Artemis exposes the mBean Server via its own configuration. This is so Artemis can leverage the JAAS authentication layer via JMX. If you want to expose this then you will need to disable both the connector and the authorisation by removing them from the `management.xml` configuration. Please refer to Java Management guide to configure the server for remote management (system properties must be set in `artemis.profile`).

By default, Apache ActiveMQ Artemis server uses the JMX domain "org.apache.activemq.artemis". To manage several Apache ActiveMQ Artemis servers from the *same* MBeanServer, the JMX domain can be configured for each individual Apache ActiveMQ Artemis server by setting `jmx-domain` in `broker.xml`:

```
<!-- use a specific JMX domain for ActiveMQ Artemis MBeans -->
<jmx-domain>my.org.apache.activemq</jmx-domain>
```

Example

See the JMX Management Example which shows how to use a remote connection to JMX and MBean proxies to manage Apache ActiveMQ Artemis.

Exposing JMX using Jolokia

The default Broker configuration ships with the Jolokia HTTP agent deployed as a web application. Jolokia is a remote JMX-over-HTTP bridge that exposes MBeans. For a full guide as to how to use it refer to Jolokia Documentation, however a simple example to query the broker's version would be to use a browser and go to the URL http://username:password@localhost:8161/console/jolokia/read/org.apache.activemq.artemis:broker="0.0.0.0"/Version.

This would give you back something like the following:

```
{"request":{"mbean":"org.apache.activemq.artemis:broker=\"0.0.0.0\"","attribut
```

JMX and the Console

The console that ships with Artemis uses Jolokia under the covers which in turn uses JMX. This will use the authentication configuration in the `management.xml` file as described in the previous section. This means that when mBeans are accessed via the console the credentials used to log into the console and the

roles associated with them. By default access to the console is only allow via users with the amq role. This is configured in the `artemis.profile` via the system property `-Dhawtio.role=amq`. You can configure multiple roles by changing this to `-Dhawtio.roles=amq,view,update`.

If a user doesn't have the correct role to invoke a specific operation then this will display an authorisation exception in the console.

Using Management Message API

The management message API in ActiveMQ Artemis is accessed by sending Core Client messages to a special address, the *management address*.

Management messages are regular Core Client messages with well-known properties that the server needs to understand to interact with the management API:

- The name of the managed resource
- The name of the management operation
- The parameters of the management operation

When such a management message is sent to the management address, Apache ActiveMQ Artemis server will handle it, extract the information, invoke the operation on the managed resources and send a *management reply* to the management message's reply-to address (specified by `ClientMessageImpl.REPLYTO_HEADER_NAME`).

A `ClientConsumer` can be used to consume the management reply and retrieve the result of the operation (if any) stored in the reply's body. For portability, results are returned as a JSON String rather than Java Serialization (the `org.apache.activemq.artemis.api.core.management.ManagementHelper` can be used to convert the JSON string to Java objects).

These steps can be simplified to make it easier to invoke management operations using Core messages:

1. Create a `ClientRequestor` to send messages to the management address and receive replies
2. Create a `ClientMessage`
3. Use the helper class `org.apache.activemq.artemis.api.core.management.ManagementHelper` to fill the message with the management properties
4. Send the message using the `ClientRequestor`
5. Use the helper class `org.apache.activemq.artemis.api.core.management.ManagementHelper` to retrieve the operation result from the management reply.

For example, to find out the number of messages in the queue `exampleQueue`:

```
ClientSession session = ...
ClientRequestor requestor = new ClientRequestor(session, "activemq.management"
ClientMessage message = session.createMessage(false);
ManagementHelper.putAttribute(message, "queue.exampleQueue", "messageCount");
session.start();
ClientMessage reply = requestor.request(m);
int count = (Integer) ManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

Management operation name and parameters must conform to the Java interfaces defined in the `management` packages.

Names of the resources are built using the helper class `org.apache.activemq.artemis.api.core.management.ResourceNames` and are straightforward (e.g. `queue.exampleQueue` for `QueueControl` of the Queue `exampleQueue`, or `broker` for the `ActiveMQServerControl`).

> **Note:**
>
> The `ManagementHelper` class can be used only with Core JMS messages. When called with a message from a different JMS library, an exception will be thrown.

Configuring Management

The management address to send management messages is configured in `broker.xml`:

```
<management-address>activemq.management</management-address>
```

By default, the address is `activemq.management`.

The management address requires a *special* user permission `manage` to be able to receive and handle management messages. This is also configured in broker.xml:

```
<!-- users with the admin role will be allowed to manage -->
<!-- Apache ActiveMQ Artemis using management messages    -->
<security-setting match="activemq.management">
   <permission type="manage" roles="admin" />
</security-setting>
```

Example

See the Management Example which shows how to use JMS messages to manage the Apache ActiveMQ Artemis server.

Management Notifications

Apache ActiveMQ Artemis emits *notifications* to inform listeners of potentially interesting events (creation of new resources, security violation, etc.).

These notifications can be received by two different ways:

- JMX notifications
- Notification messages

JMX Notifications

If JMX is enabled (see Configuring JMX section), JMX notifications can be received by subscribing to `org.apache.activemq.artemis:type=Broker,brokerName=<broker name>,module=Core,serviceType=Server` for notifications on resources.

Notification Messages

Apache ActiveMQ Artemis defines a special *management notification address*. Queues can be bound to this address so that clients will receive management notifications as messages.

A client which wants to receive management notifications must create a queue bound to the management notification address. It can then receive the notifications from its queue.

Notifications messages are regular messages with additional properties corresponding to the notification (its type, when it occurred, the resources which were concerned, etc.).

Since notifications are regular messages, it is possible to use message selectors to filter out notifications and receives only a subset of all the notifications emitted by the server.

Configuring The Management Notification Address

The management notification address to receive management notifications is configured in `broker.xml`:

```
<management-notification-address>activemq.notifications</management-notificati
```

By default, the address is `activemq.notifications`.

Receiving Notification Messages

Apache ActiveMQ Artemis's Core JMS Client can be used to receive notifications:

```
Topic notificationsTopic = ActiveMQJMSClient.createTopic("activemq.notificatio

Session session = ...
MessageConsumer notificationConsumer = session.createConsumer(notificationsTop
notificationConsumer.setMessageListener(new MessageListener() {
   public void onMessage(Message notif) {
    System.out.println("------------------------");
    System.out.println("Received notification:");
    try {
     Enumeration propertyNames = notif.getPropertyNames();
     while (propertyNames.hasMoreElements()) {
      String propertyName = (String)propertyNames.nextElement();
      System.out.format("  %s: %s\n", propertyName, notif.getObjectProperty(pr
     }
    } catch (JMSException e) {
    }
    System.out.println("------------------------");
   }
});
```

Example

See the Management Notification Example which shows how to use a JMS `MessageListener` to receive management notifications from ActiveMQ Artemis server.

Notification Types and Headers

Below is a list of all the different kinds of notifications as well as which headers are on the messages. Every notification has a `_AMQ_NotifType` (value noted in parentheses) and `_AMQ_NotifTimestamp` header. The timestamp is the un-formatted result of a call to `java.lang.System.currentTimeMillis()`.

- `BINDING_ADDED` (0)

  `_AMQ_Binding_Type`, `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`, `_AMQ_Binding_ID`, `_AMQ_Distance`, `_AMQ_FilterString`

- `BINDING_REMOVED` (1)

  `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`, `_AMQ_Binding_ID`, `_AMQ_Distance`, `_AMQ_FilterString`

- `CONSUMER_CREATED` (2)

  `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`, `_AMQ_Distance`, `_AMQ_ConsumerCount`, `_AMQ_User`, `_AMQ_ValidatedUser`, `_AMQ_RemoteAddress`, `_AMQ_SessionName`, `_AMQ_FilterString`, `_AMQ_CertSubjectDN`

- `CONSUMER_CLOSED` (3)

  `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`, `_AMQ_Distance`, `_AMQ_ConsumerCount`, `_AMQ_User`, `_AMQ_RemoteAddress`, `_AMQ_SessionName`, `_AMQ_FilterString`

- `SECURITY_AUTHENTICATION_VIOLATION` (6)

  `_AMQ_User`, `_AMQ_CertSubjectDN`, `_AMQ_RemoteAddress`

- `SECURITY_PERMISSION_VIOLATION` (7)

  `_AMQ_Address` , `_AMQ_CheckType` , `_AMQ_User`

- `DISCOVERY_GROUP_STARTED` (8)

  `name`

- `DISCOVERY_GROUP_STOPPED` (9)

  `name`

- `BROADCAST_GROUP_STARTED` (10)

  `name`

- `BROADCAST_GROUP_STOPPED` (11)

  `name`

- `BRIDGE_STARTED` (12)

  `name`

- `BRIDGE_STOPPED` (13)

  `name`

- `CLUSTER_CONNECTION_STARTED` (14)

  `name`

- `CLUSTER_CONNECTION_STOPPED` (15)

  `name`

- `ACCEPTOR_STARTED` (16)

  `factory` , `id`

- `ACCEPTOR_STOPPED` (17)

  `factory` , `id`

- `PROPOSAL` (18)

  `_JBM_ProposalGroupId` , `_JBM_ProposalValue` , `_AMQ_Binding_Type` , `_AMQ_Address` , `_AMQ_Distance`

- `PROPOSAL_RESPONSE` (19)

  `_JBM_ProposalGroupId` , `_JBM_ProposalValue` , `_JBM_ProposalAltValue` , `_AMQ_Binding_Type` , `_AMQ_Address` , `_AMQ_Distance`

- `CONSUMER_SLOW` (21)

  `_AMQ_Address` , `_AMQ_ConsumerCount` , `_AMQ_RemoteAddress` , `_AMQ_ConnectionName` , `_AMQ_ConsumerName` , `_AMQ_SessionName`

- `ADDRESS_ADDED` (22)

  `_AMQ_Address` , `_AMQ_Routing_Type`

- `ADDRESS_REMOVED` (23)

  `_AMQ_Address` , `_AMQ_Routing_Type`

- `CONNECTION_CREATED` (24)

  `_AMQ_ConnectionName`, `_AMQ_RemoteAddress`

- `CONNECTION_DESTROYED` (25)

  `_AMQ_ConnectionName`, `_AMQ_RemoteAddress`

- `SESSION_CREATED` (26)

  `_AMQ_ConnectionName`, `_AMQ_User`, `_AMQ_SessionName`

- `SESSION_CLOSED` (27)

  `_AMQ_ConnectionName`, `_AMQ_User`, `_AMQ_SessionName`

- `MESSAGE_DELIVERED` (28)

  `_AMQ_Address`, `_AMQ_Routing_Type`, `_AMQ_RoutingName`, `_AMQ_ConsumerName`, `_AMQ_Message_ID`

- `MESSAGE_EXPIRED` (29)

  `_AMQ_Address`, `_AMQ_Routing_Type`, `_AMQ_RoutingName`, `_AMQ_ConsumerName`, `_AMQ_Message_ID`

Message Counters

Message counters can be used to obtain information on queues *over time* as Apache ActiveMQ Artemis keeps a history on queue metrics.

They can be used to show *trends* on queues. For example, using the management API, it would be possible to query the number of messages in a queue at regular interval. However, this would not be enough to know if the queue is used: the number of messages can remain constant because nobody is sending or receiving messages from the queue or because there are as many messages sent to the queue than messages consumed from it. The number of messages in the queue remains the same in both cases but its use is widely different.

Message counters give additional information about the queues:

- `count`

  The *total* number of messages added to the queue since the server was started

- `countDelta`

  the number of messages added to the queue *since the last message counter update*

- `messageCount`

  The *current* number of messages in the queue

- `messageCountDelta`

The *overall* number of messages added/removed from the queue *since the last message counter update*. For example, if `messageCountDelta` is equal to `-10` this means that overall 10 messages have been removed from the queue (e.g. 2 messages were added and 12 were removed)

- `lastAddTimestamp`

  The timestamp of the last time a message was added to the queue

- `udpateTimestamp`

  The timestamp of the last message counter update

  These attributes can be used to determine other meaningful data as well. For example, to know specifically how many messages were *consumed* from the queue since the last update simply subtract the `messageCountDelta` from `countDelta`.

Configuring Message Counters

By default, message counters are disabled as it might have a small negative effect on memory.

To enable message counters, you can set it to `true` in `broker.xml`:

```
<message-counter-enabled>true</message-counter-enabled>
```

Message counters keep a history of the queue metrics (10 days by default) and sample all the queues at regular interval (10 seconds by default). If message counters are enabled, these values should be configured to suit your messaging use case in `broker.xml`:

```
<!-- keep history for a week -->
<message-counter-max-day-history>7</message-counter-max-day-history>
<!-- sample the queues every minute (60000ms) -->
<message-counter-sample-period>60000</message-counter-sample-period>
```

Message counters can be retrieved using the Management API. For example, to retrieve message counters on a queue using JMX:

```
// retrieve a connection to Apache ActiveMQ Artemis's MBeanServer
MBeanServerConnection mbsc = ...
QueueControlMBean queueControl = (QueueControl)MBeanServerInvocationHandler.ne
   on,
   QueueControl.class,
   false);
// message counters are retrieved as a JSON String
String counters = queueControl.listMessageCounter();
// use the MessageCounterInfo helper class to manipulate message counters more
MessageCounterInfo messageCounter = MessageCounterInfo.fromJSON(counters);
System.out.format("%s message(s) in the queue (since last sample: %s)\n",
messageCounter.getMessageCount(),
messageCounter.getMessageCountDelta());
```

Example

See the Message Counter Example which shows how to use message counters to retrieve information on a queue.

Management Console

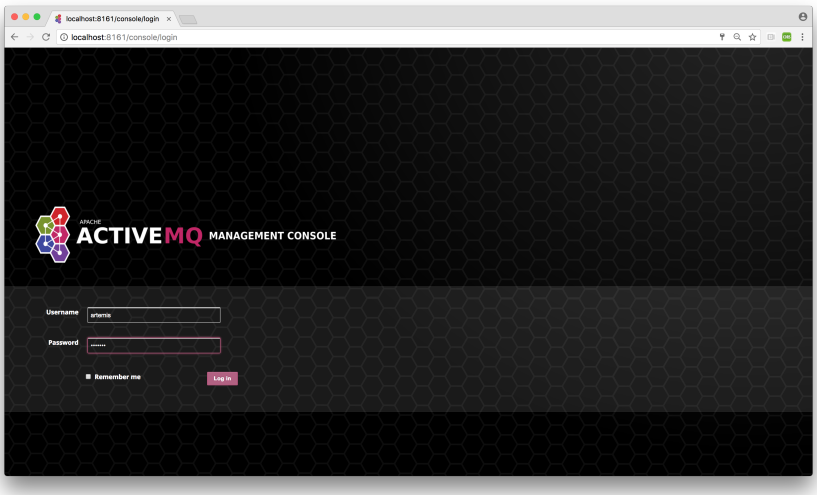
Apache ActiveMQ Artemis ships by default with a management console. It is powered by Hawt.io.

Its purpose is to expose the Management API via a user friendly web ui.

Login

To access the management console use a browser and go to the URL http://localhost:8161/console.

A login screen will be presented, if your broker is secure, you will need to use a user with admin role, if it is unsecure simply enter any user/password.

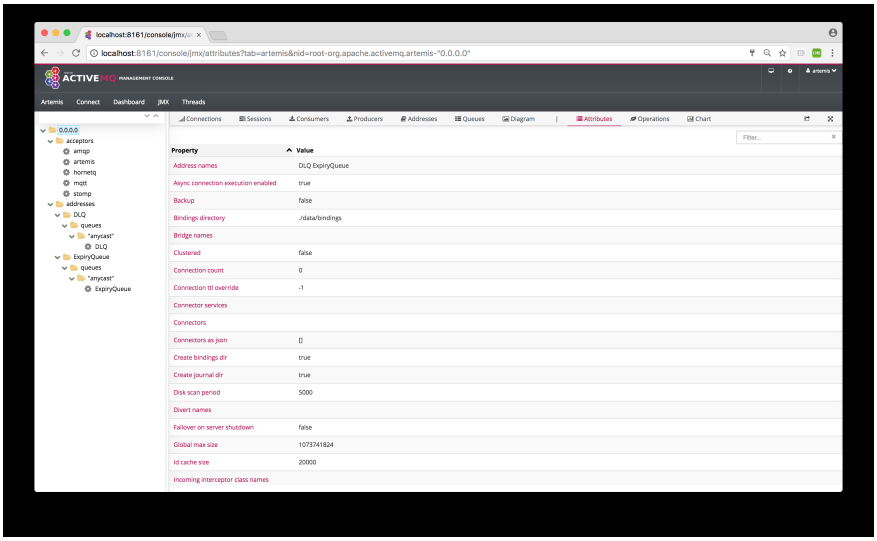


Security

That Jolokia JMX-HTTP bridge is secured via a policy file in the broker configuration directory: 'etc/jolokia-access.xml'. The contents of that file should be modified as described in the Jolokia Security Guide. By default the console is locked down to 'localhost', pay particular attention to the 'CORS' restrictions when exposing the console web endpoint over the network.

Console

Once logged in you should be presented with a screen similar to.



Navigation Menu

On the top right is small menu area you will see some icons.

- `question mark` This will load the artemis documentation in the console main window
- `person` will provide a drop down menu with
- `about` this will load an about screen, here you will be able to see and validate versions
- `log out` self descriptive.

Navigation Tabs

Running below the Navigation Menu you will see several default feature tabs.

- `Artemis` This is the core tab for Apache ActiveMQ Artemis specific functionality. The rest of this document will focus on this.
- `Connect` This allows you to connect to a remote broker from the same console.
- `Dashboard` Here you can create and save graphs and tables of metrics available via JMX, a default jvm health dashboard is provided.
- `JMX` This exposes the raw Jolokia JMX so you can browse/access all the JMX endpoints exposed by the JVM.
- `Threads` This allows you to monitor the thread usage and their state.

You can install further hawtio plugins if you wish to have further functionality.

Artemis Tab

Click `Artemis` in the top navigation bar to see the Artemis specific plugin. (The Artemis tab won't appear if there is no broker in this JVM). The Artemis plugin works very much the same as the JMX plugin however with a focus on interacting

with an Artemis broker.

Tree View

The tree view on the left-hand side shows the top level JMX tree of each broker instance running in the JVM. Expanding the tree will show the various MBeans registered by Artemis that you can inspect via the **Attributes** tab.

Acceptors

This expands to show and expose details of the current configured acceptors.

Addresses

This expands to show the current configured available `addresses`.

Under the address you can expand to find the `queues` for the address exposing attributes

Key Operations

Creating a new Address

To create a new address simply click on the broker or the address folder in the jmx tree and click on the create tab.

Once you have created an address you should be able to **Send** to it by clicking on it in the jmx tree and clicking on the send tab.

Creating a new Queue

To create a new queue click on the address you want to bind the queue to and click on the create tab.

Once you have created a queue you should be able to **Send** a message to it or **Browse** it or view the **Attributes** or **Charts**. Simply click on the queue in th ejmx tree and click on the appropriate tab.

You can also see a graphical view of all brokers, addresses, queues and their consumers using the **Diagram** tab.

Metrics

Apache ActiveMQ Artemis can export metrics to a variety of monitoring systems via the Micrometer vendor-neutral application metrics facade.

Important runtime metrics have been instrumented via the Micrometer API, and all a user needs to do is implement `org.apache.activemq.artemis.core.server.metrics.ActiveMQMetricsPlugin` in order to instantiate and configure a `io.micrometer.core.instrument.MeterRegistry` implementation. Relevant implementations of `MeterRegistry` are available from the Micrometer code-base.

This is a simple interface:

```
public interface ActiveMQMetricsPlugin extends Serializable {

   ActiveMQMetricsPlugin init(Map<String, String> options);

   MeterRegistry getRegistry();
}
```

When the broker starts it will call `init` and pass in the `options` which can be specified in XML as key/value properties. At this point the plugin should instantiate and configure the `io.micrometer.core.instrument.MeterRegistry` implementation.

Later during the broker startup process it will call `getRegistry` in order to get the `MeterRegistry` implementation and use it for registering meters.

The broker ships with two `ActiveMQMetricsPlugin` implementations:

- `org.apache.activemq.artemis.core.server.metrics.plugins.LoggingMetricsPlugin` This plugin simply logs metrics. It's not very useful for production, but can serve as a demonstration of the Micrometer integration. It takes no key/value properties for configuration.
- `org.apache.activemq.artemis.core.server.metrics.plugins.SimpleMetricsPlugin` This plugin is used for testing. It is in-memory only and provides no external output. It takes no key/value properties for configuration.

Metrics

The following metrics are exported, categorized by component. A description for each metric is exported along with the metric itself therefore the description will not be repeated here.

Broker

- connection.count
- total.connection.count
- address.memory.usage

Address

- routed.message.count
- unrouted.message.count

Queue

- message.count
- durable.message.count
- persistent.size
- durable.persistent.size
- delivering.message.count
- delivering.durable.message.count
- delivering.persistent.size
- delivering.durable.persistent.size
- scheduled.message.count
- scheduled.durable.message.count
- scheduled.persistent.size
- scheduled.durable.persistent.size
- messages.acknowledged
- messages.added
- messages.killed
- messages.expired
- consumer.count

It may appear that some higher level broker metrics are missing (e.g. total message count). However, these metrics can be deduced by aggregating the lower level metrics (e.g. aggregate the message.count metrics from all queues to get the total).

JVM memory metrics are also exported by default and GC and thread metrics can be configured

Configuration

Metrics for all addresses and queues are enabled by default. If you want to disable metrics for a particular address or set of addresses you can do so by setting the `enable-metrics` `address-setting` to `false`.

In `broker.xml` use the `metrics` element to configure which JVM metrics are reported and to configure the plugin itself. Here's a configuration with all JVM metrics:

```
<metrics>
   <jvm-memory>true</jvm-memory> <!-- defaults to true -->
   <jvm-gc>true</jvm-gc> <!-- defaults to false -->
   <jvm-threads>true</jvm-threads> <!-- defaults to false -->
   <plugin class-name="org.apache.activemq.artemis.core.server.metrics.plugins
</metrics>
```

The plugin can also be configured with key/value properties in order to customize the implementation as necessary, e.g.:

```
<metrics>
   <plugin class-name="org.example.MyMetricsPlugin">
      <property key="host" value="example.org" />
      <property key="port" value="5162" />
      <property key="foo" value="10" />
   </plugin>
</metrics>
```

Security

This chapter describes how security works with Apache ActiveMQ Artemis and how you can configure it.

To disable security completely simply set the `security-enabled` property to `false` in the `broker.xml` file.

For performance reasons security is cached and invalidated every so long. To change this period set the property `security-invalidation-interval`, which is in milliseconds. The default is `10000` ms.

Tracking the Validated User

To assist in security auditing the `populate-validated-user` option exists. If this is `true` then the server will add the name of the validated user to the message using the key `_AMQ_VALIDATED_USER`. For JMS and Stomp clients this is mapped to the key `JMSXUserID`. For users authenticated based on their SSL certificate this name is the name to which their certificate's DN maps. If `security-enabled` is `false` and `populate-validated-user` is `true` then the server will simply use whatever user name (if any) the client provides. This option is `false` by default.

Role based security for addresses

Apache ActiveMQ Artemis contains a flexible role-based security model for applying security to queues, based on their addresses.

As explained in Using Core, Apache ActiveMQ Artemis core consists mainly of sets of queues bound to addresses. A message is sent to an address and the server looks up the set of queues that are bound to that address, the server then routes the message to those set of queues.

Apache ActiveMQ Artemis allows sets of permissions to be defined against the queues based on their address. An exact match on the address can be used or a wildcard match can be used.

Eight different permissions can be given to the set of queues which match the address. Those permissions are:

- `createAddress`. This permission allows the user to create an address fitting the `match`.
- `deleteAddress`. This permission allows the user to delete an address fitting the `match`.
- `createDurableQueue`. This permission allows the user to create a durable queue under matching addresses.

- `deleteDurableQueue`. This permission allows the user to delete a durable queue under matching addresses.
- `createNonDurableQueue`. This permission allows the user to create a non-durable queue under matching addresses.
- `deleteNonDurableQueue`. This permission allows the user to delete a non-durable queue under matching addresses.
- `send`. This permission allows the user to send a message to matching addresses.
- `consume`. This permission allows the user to consume a message from a queue bound to matching addresses.
- `browse`. This permission allows the user to browse a queue bound to the matching address.
- `manage`. This permission allows the user to invoke management operations by sending management messages to the management address.

For each permission, a list of roles who are granted that permission is specified. If the user has any of those roles, he/she will be granted that permission for that set of addresses.

Let's take a simple example, here's a security block from `broker.xml` file:

```
<security-setting match="globalqueues.europe.#">
   <permission type="createDurableQueue" roles="admin"/>
   <permission type="deleteDurableQueue" roles="admin"/>
   <permission type="createNonDurableQueue" roles="admin, guest, europe-users"
   <permission type="deleteNonDurableQueue" roles="admin, guest, europe-users"
   <permission type="send" roles="admin, europe-users"/>
   <permission type="consume" roles="admin, europe-users"/>
</security-setting>
```

Using the default wildcard syntax the `#` character signifies "any sequence of words". Words are delimited by the `.` character. Therefore, the above security block applies to any address that starts with the string "globalqueues.europe.".

Only users who have the `admin` role can create or delete durable queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles `admin`, `guest`, or `europe-users` can create or delete temporary queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles `admin` or `europe-users` can send messages to these addresses or consume messages from queues bound to an address that starts with the string "globalqueues.europe."

The mapping between a user and what roles they have is handled by the security manager. Apache ActiveMQ Artemis ships with a user manager that reads user credentials from a file on disk, and can also plug into JAAS or JBoss Application Server security.

For more information on configuring the security manager, please see 'Changing the Security Manager'.

There can be zero or more `security-setting` elements in each xml file. Where more than one match applies to a set of addresses the *more specific* match takes precedence.

Let's look at an example of that, here's another `security-setting` block:

```
<security-setting match="globalqueues.europe.orders.#">
   <permission type="send" roles="europe-users"/>
   <permission type="consume" roles="europe-users"/>
</security-setting>
```

In this `security-setting` block the match 'globalqueues.europe.orders.#' is more specific than the previous match 'globalqueues.europe.#'. So any addresses which match 'globalqueues.europe.orders.#' will take their security settings *only* from the latter security-setting block.

Note that settings are not inherited from the former block. All the settings will be taken from the more specific matching block, so for the address 'globalqueues.europe.orders.plastics' the only permissions that exist are `send` and `consume` for the role europe-users. The permissions `createDurableQueue`, `deleteDurableQueue`, `createNonDurableQueue`, `deleteNonDurableQueue` are not inherited from the other security-setting block.

By not inheriting permissions, it allows you to effectively deny permissions in more specific security-setting blocks by simply not specifying them. Otherwise it would not be possible to deny permissions in sub-groups of addresses.

Security Setting Plugin

Aside from configuring sets of permissions via XML these permissions can alternatively be configured via a plugin which implements `org.apache.activemq.artemis.core.server.SecuritySettingPlugin` e.g.:

```
<security-settings>
   <security-setting-plugin class-name="org.apache.activemq.artemis.core.serve
      <setting name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFa
      <setting name="connectionURL" value="ldap://localhost:1024"/>
      <setting name="connectionUsername" value="uid=admin,ou=system"/>
      <setting name="connectionPassword" value="secret"/>
      <setting name="connectionProtocol" value="s"/>
      <setting name="authentication" value="simple"/>
   </security-setting-plugin>
</security-settings>
```

Most of this configuration is specific to the plugin implementation. However, there are two configuration details that will be specified for every implementation:

- `class-name`. This attribute of `security-setting-plugin` indicates the name of the class which implements `org.apache.activemq.artemis.core.server.SecuritySettingPlugin`.

- `setting`. Each of these elements represents a name/value pair that will be passed to the implementation for configuration purposes.

See the JavaDoc on `org.apache.activemq.artemis.core.server.SecuritySettingPlugin` for further details about the interface and what each method is expected to do.

Available plugins

LegacyLDAPSecuritySettingPlugin

This plugin will read the security information that was previously handled by `LDAPAuthorizationMap` and the `cachedLDAPAuthorizationMap` in Apache ActiveMQ 5.x and turn it into Artemis security settings where possible. The security implementations of ActiveMQ 5.x and Artemis don't match perfectly so some translation must occur to achieve near equivalent functionality.

Here is an example of the plugin's configuration:

```
<security-setting-plugin class-name="org.apache.activemq.artemis.core.server.i
   <setting name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFacto
   <setting name="connectionURL" value="ldap://localhost:1024"/>
   <setting name="connectionUsername" value="uid=admin,ou=system"/>
   <setting name="connectionPassword" value="secret"/>
   <setting name="connectionProtocol" value="s"/>
   <setting name="authentication" value="simple"/>
</security-setting-plugin>
```

- `class-name`. The implementation is `org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin`.
- `initialContextFactory`. The initial context factory used to connect to LDAP. It must always be set to `com.sun.jndi.ldap.LdapCtxFactory` (i.e. the default value).
- `connectionURL`. Specifies the location of the directory server using an ldap URL, `ldap://Host:Port`. You can optionally qualify this URL, by adding a forward slash, `/`, followed by the DN of a particular node in the directory tree. For example, `ldap://ldapserver:10389/ou=system`. The default is `ldap://localhost:1024`.
- `connectionUsername`. The DN of the user that opens the connection to the directory server. For example, `uid=admin,ou=system`. Directory servers generally require clients to present username/password credentials in order to open a connection.
- `connectionPassword`. The password that matches the DN from `connectionUsername`. In the directory server, in the DIT, the password is normally stored as a `userPassword` attribute in the corresponding directory entry.

- `connectionProtocol`. Currently the only supported value is a blank string. In future, this option will allow you to select the Secure Socket Layer (SSL) for the connection to the directory server. **Note:** this option must be set explicitly to an empty string, because it has no default value.
- `authentication`. Specifies the authentication method used when binding to the LDAP server. Can take either of the values, `simple` (username and password, the default value) or `none` (anonymous). **Note:** Simple Authentication and Security Layer (SASL) authentication is currently not supported.
- `destinationBase`. Specifies the DN of the node whose children provide the permissions for all destinations. In this case the DN is a literal value (that is, no string substitution is performed on the property value). For example, a typical value of this property is `ou=destinations,o=ActiveMQ,ou=system` (i.e. the default value).
- `filter`. Specifies an LDAP search filter, which is used when looking up the permissions for any kind of destination. The search filter attempts to match one of the children or descendants of the queue or topic node. The default value is `(cn=*)`.
- `roleAttribute`. Specifies an attribute of the node matched by `filter`, whose value is the DN of a role. Default value is `uniqueMember`.
- `adminPermissionValue`. Specifies a value that matches the `admin` permission. The default value is `admin`.
- `readPermissionValue`. Specifies a value that matches the `read` permission. The default value is `read`.
- `writePermissionValue`. Specifies a value that matches the `write` permission. The default value is `write`.
- `enableListener`. Whether or not to enable a listener that will automatically receive updates made in the LDAP server and update the broker's authorization configuration in real-time. The default value is `true`.
- `mapAdminToManage`. Whether or not to map the legacy `admin` permission to the `manage` permission. See details of the mapping semantics below. The default value is `false`.

The name of the queue or topic defined in LDAP will serve as the "match" for the security-setting, the permission value will be mapped from the ActiveMQ 5.x type to the Artemis type, and the role will be mapped as-is.

ActiveMQ 5.x only has 3 permission types - `read`, `write`, and `admin`. These permission types are described on their website. However, as described previously, ActiveMQ Artemis has 9 permission types - `createAddress`, `deleteAddress`, `createDurableQueue`, `deleteDurableQueue`, `createNonDurableQueue`, `deleteNonDurableQueue`, `send`, `consume`, `browse`, and `manage`. Here's how the old types are mapped to the new types:

- `read` - `consume`, `browse`

- `write` - `send`
- `admin` - `createAddress`, `deleteAddress`, `createDurableQueue`, `deleteDurableQueue`, `createNonDurableQueue`, `deleteNonDurableQueue`, `manage` (if `mapAdminToManage` is `true`)

As mentioned, there are a few places where a translation was performed to achieve some equivalence.:

- This mapping doesn't include the Artemis `manage` permission type by default since there is no type analogous for that in ActiveMQ 5.x. However, if `mapAdminToManage` is `true` then the legacy `admin` permission will be mapped to the `manage` permission.
- The `admin` permission in ActiveMQ 5.x relates to whether or not the broker will auto-create a destination if it doesn't exist and the user sends a message to it. Artemis automatically allows the automatic creation of a destination if the user has permission to send message to it. Therefore, the plugin will map the `admin` permission to the 6 aforementioned permissions in Artemis by default. If `mapAdminToManage` is `true` then the legacy `admin` permission will be mapped to the `manage` permission as well.

Secure Sockets Layer (SSL) Transport

When messaging clients are connected to servers, or servers are connected to other servers (e.g. via bridges) over an untrusted network then Apache ActiveMQ Artemis allows that traffic to be encrypted using the Secure Sockets Layer (SSL) transport.

For more information on configuring the SSL transport, please see Configuring the Transport.

User credentials

Apache ActiveMQ Artemis ships with two security manager implementations:

- The legacy, deprecated `ActiveMQSecurityManager` that reads user credentials, i.e. user names, passwords and role information from properties files on the classpath called `artemis-users.properties` and `artemis-roles.properties`.
- The flexible, pluggable `ActiveMQJAASSecurityManager` which supports any standard JAAS login module. Artemis ships with several login modules which will be discussed further down. This is the default security manager.

JAAS Security Manager

When using the Java Authentication and Authorization Service (JAAS) much of the configuration depends on which login module is used. However, there are a few commonalities for every case. The first place to look is in `bootstrap.xml`. Here is an example using the `PropertiesLogin` JAAS login module which reads user, password, and role information from properties files:

```
<jaas-security domain="PropertiesLogin"/>
```

No matter what login module you're using, you'll need to specify it here in `bootstrap.xml`. The `domain` attribute here refers to the relevant login module entry in `login.config`. For example:

```
PropertiesLogin {
    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule r
        debug=true
        org.apache.activemq.jaas.properties.user="artemis-users.properties"
        org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

The `login.config` file is a standard JAAS configuration file. You can read more about this file on Oracle's website. In short, the file defines:

- an alias for an entry (e.g. `PropertiesLogin`)
- the implementation class for the login module (e.g. `org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule`)
- a flag which indicates whether the success of the login module is `required`, `requisite`, `sufficient`, or `optional` (see more details on these flags in the JavaDoc
- a list of configuration options specific to the login module implementation

By default, the location and name of `login.config` is specified on the Artemis command-line which is set by `etc/artemis.profile` on linux and `etc\artemis.profile.cmd` on Windows.

Dual Authentication

The JAAS Security Manager also supports another configuration parameter - `certificate-domain`. This is useful when you want to authenticate clients connecting with SSL connections based on their SSL certificates (e.g. using the `CertificateLoginModule` discussed below) but you still want to authenticate clients connecting with non-SSL connections with, e.g., username and password. Here's an example of what would go in `bootstrap.xml`:

```
<jaas-security domain="PropertiesLogin" certificate-domain="CertLogin"/>
```

And here's the corresponding `login.config`:

```
PropertiesLogin {
   org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule re
       debug=false
       org.apache.activemq.jaas.properties.user="artemis-users.properties"
       org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};

CertLogin {
   org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginM
       debug=true
       org.apache.activemq.jaas.textfiledn.user="cert-users.properties"
       org.apache.activemq.jaas.textfiledn.role="cert-roles.properties";
};
```

When the broker is configured this way then any client connecting with SSL and a client certificate will be authenticated using `CertLogin` and any client connecting without SSL will be authenticated using `PropertiesLogin`.

JAAS Login Modules

GuestLoginModule

Allows users without credentials (and, depending on how it is configured, possibly also users with invalid credentials) to access the broker. Normally, the guest login module is chained with another login module, such as a properties login module. It is implemented by `org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule`.

- `org.apache.activemq.jaas.guest.user` - the user name to assign; default is "guest"
- `org.apache.activemq.jaas.guest.role` - the role name to assign; default is "guests"
- `credentialsInvalidate` - boolean flag; if `true`, reject login requests that include a password (i.e. guest login succeeds only when the user does not provide a password); default is `false`
- `debug` - boolean flag; if `true`, enable debugging; this is used only for testing or debugging; normally, it should be set to `false`, or omitted; default is `false`

There are two basic use cases for the guest login module, as follows:

- Guests with no credentials or invalid credentials.
- Guests with no credentials only.

The following snippet shows how to configure a JAAS login entry for the use case where users with no credentials or invalid credentials are logged in as guests. In this example, the guest login module is used in combination with the properties login module.

```
activemq-domain {
  org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule suf
      debug=true
      org.apache.activemq.jaas.properties.user="artemis-users.properties"
      org.apache.activemq.jaas.properties.role="artemis-roles.properties";

  org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule sufficie
      debug=true
      org.apache.activemq.jaas.guest.user="anyone"
      org.apache.activemq.jaas.guest.role="restricted";
};
```

Depending on the user login data, authentication proceeds as follows:

- User logs in with a valid password — the properties login module successfully authenticates the user and returns immediately. The guest login module is not invoked.
- User logs in with an invalid password — the properties login module fails to authenticate the user, and authentication proceeds to the guest login module. The guest login module successfully authenticates the user and returns the guest principal.
- User logs in with a blank password — the properties login module fails to authenticate the user, and authentication proceeds to the guest login module. The guest login module successfully authenticates the user and returns the guest principal.

The following snipped shows how to configure a JAAS login entry for the use case where only those users with no credentials are logged in as guests. To support this use case, you must set the credentialsInvalidate option to true in the configuration of the guest login module. You should also note that, compared with the preceding example, the order of the login modules is reversed and the flag attached to the properties login module is changed to requisite.

```
activemq-guest-when-no-creds-only-domain {
    org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule suffic
        debug=true
       credentialsInvalidate=true
       org.apache.activemq.jaas.guest.user="guest"
       org.apache.activemq.jaas.guest.role="guests";

    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule r
        debug=true
        org.apache.activemq.jaas.properties.user="artemis-users.properties"
        org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

Depending on the user login data, authentication proceeds as follows:

- User logs in with a valid password — the guest login module fails to authenticate the user (because the user has presented a password while the credentialsInvalidate option is enabled) and authentication proceeds to the properties login module. The properties login module successfully authenticates the user and returns.

- User logs in with an invalid password — the guest login module fails to authenticate the user and authentication proceeds to the properties login module. The properties login module also fails to authenticate the user. The net result is authentication failure.
- User logs in with a blank password — the guest login module successfully authenticates the user and returns immediately. The properties login module is not invoked.

PropertiesLoginModule

The JAAS properties login module provides a simple store of authentication data, where the relevant user data is stored in a pair of flat files. This is convenient for demonstrations and testing, but for an enterprise system, the integration with LDAP is preferable. It is implemented by `org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule`.

- `org.apache.activemq.jaas.properties.user` - the path to the file which contains user and password properties
- `org.apache.activemq.jaas.properties.role` - the path to the file which contains user and role properties
- `reload` - boolean flag; whether or not to reload the properties files when a modification occurs; default is `false`
- `debug` - boolean flag; if `true`, enable debugging; this is used only for testing or debugging; normally, it should be set to `false`, or omitted; default is `false`

In the context of the properties login module, the `artemis-users.properties` file consists of a list of properties of the form, `UserName=Password`. For example, to define the users `system`, `user`, and `guest`, you could create a file like the following:

```
system=manager
user=password
guest=password
```

Passwords in `artemis-users.properties` can be hashed. Such passwords should follow the syntax `ENC(<hash>)`. Hashed passwords can easily be added to `artemis-users.properties` using the `user` CLI command from the Artemis *instance*. This command will not work from the Artemis home.

```
./artemis user add --user guest --password guest --role admin
```

This will use the default codec to perform a "one-way" hash of the password and alter both the `artemis-users.properties` and `artemis-roles.properties` files with the specified values.

The `artemis-roles.properties` file consists of a list of properties of the form, `Role=UserList`, where UserList is a comma-separated list of users. For example, to define the roles `admins`, `users`, and `guests`, you could create a file like the following:

```
admins=system
users=system,user
guests=guest
```

As mentioned above, the Artemis command-line interface supports a command to `add` a user. Commands to `list` (one or all) users, `remove` a user, and `reset` a user's password and/or role(s) are also supported via the command-line interface as well as the normal management interfaces (e.g. JMX, web console, etc.).

> **Warning**
>
> Management and CLI operations to manipulate user & role data are only available when using the `PropertiesLoginModule`.

LDAPLoginModule

The LDAP login module enables you to perform authentication and authorization by checking the incoming credentials against user data stored in a central X.500 directory server. For systems that already have an X.500 directory server in place, this means that you can rapidly integrate ActiveMQ Artemis with the existing security database and user accounts can be managed using the X.500 system. It is implemented by `org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule`.

- `initialContextFactory` - must always be set to `com.sun.jndi.ldap.LdapCtxFactory`
- `connectionURL` - specify the location of the directory server using an ldap URL, ldap://Host:Port. You can optionally qualify this URL, by adding a forward slash, `/`, followed by the DN of a particular node in the directory tree. For example, ldap://ldapserver:10389/ou=system.
- `authentication` - specifies the authentication method used when binding to the LDAP server. Can take either of the values, `simple` (username and password), `GSSAPI` (Kerberos SASL) or `none` (anonymous).
- `connectionUsername` - the DN of the user that opens the connection to the directory server. For example, `uid=admin,ou=system`. Directory servers generally require clients to present username/password credentials in order to open a connection.
- `connectionPassword` - the password that matches the DN from `connectionUsername`. In the directory server, in the DIT, the password is normally stored as a `userPassword` attribute in the corresponding directory entry.

- `saslLoginConfigScope` - the scope in JAAS configuration (login.config) to use to obtain Kerberos initiator credentials when the `authentication` method is SASL `GSSAPI`. The default value is `broker-sasl-gssapi`.
- `connectionProtocol` - currently, the only supported value is a blank string. In future, this option will allow you to select the Secure Socket Layer (SSL) for the connection to the directory server. This option must be set explicitly to an empty string, because it has no default value.
- `connectionPool` - boolean, enable the LDAP connection pool property 'com.sun.jndi.ldap.connect.pool'. Note that the pool is configured at the jvm level with system properties.
- `connectionTimeout` - specifies the string representation of an integer representing the connection timeout in milliseconds. If the LDAP provider cannot establish a connection within that period, it aborts the connection attempt. The integer should be greater than zero. An integer less than or equal to zero means to use the network protocol's (i.e., TCP's) timeout value.

  If `connectionTimeout` is not specified, the default is to wait for the connection to be established or until the underlying network times out.

  When connection pooling has been requested for a connection, this property also determines the maximum wait time for a connection when all connections in the pool are in use and the maximum pool size has been reached. If the value of this property is less than or equal to zero under such circumstances, the provider will wait indefinitely for a connection to become available; otherwise, the provider will abort the wait when the maximum wait time has been exceeded. See `connectionPool` for more details.
- `readTimeout` - specifies the string representation of an integer representing the read timeout in milliseconds for LDAP operations. If the LDAP provider cannot get a LDAP response within that period, it aborts the read attempt. The integer should be greater than zero. An integer less than or equal to zero means no read timeout is specified which is equivalent to waiting for the response infinitely until it is received.

  If `readTimeout` is not specified, the default is to wait for the response until it is received.
- `userBase` - selects a particular subtree of the DIT to search for user entries. The subtree is specified by a DN, which specifes the base node of the subtree. For example, by setting this option to `ou=User,ou=ActiveMQ,ou=system`, the search for user entries is restricted to the subtree beneath the `ou=User,ou=ActiveMQ,ou=system` node.
- `userSearchMatching` - specifies an LDAP search filter, which is applied to the subtree selected by `userBase`. Before passing to the LDAP search operation, the string value you provide here is subjected to string substitution, as implemented by the `java.text.MessageFormat` class. Essentially, this means that the special string, `{0}`, is substituted by the username, as extracted from the incoming client credentials.

After substitution, the string is interpreted as an LDAP search filter, where the LDAP search filter syntax is defined by the IETF standard, RFC 2254. A short introduction to the search filter syntax is available from Oracle's JNDI tutorial, Search Filters.

For example, if this option is set to `(uid={0})` and the received username is `jdoe`, the search filter becomes `(uid=jdoe)` after string substitution. If the resulting search filter is applied to the subtree selected by the user base, `ou=User,ou=ActiveMQ,ou=system`, it would match the entry, `uid=jdoe,ou=User,ou=ActiveMQ,ou=system` (and possibly more deeply nested entries, depending on the specified search depth—see the `userSearchSubtree` option).

- `userSearchSubtree` - specify the search depth for user entries, relative to the node specified by `userBase`. This option is a boolean. `false` indicates it will try to match one of the child entries of the `userBase` node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`). `true` indicates it will try to match any entry belonging to the subtree of the `userBase` node (maps to `javax.naming.directory.SearchControls.SUBTREE_SCOPE`).
- `userRoleName` - specifies the name of the multi-valued attribute of the user entry that contains a list of role names for the user (where the role names are interpreted as group names by the broker's authorization plug-in). If you omit this option, no role names are extracted from the user entry.
- `roleBase` - if you want to store role data directly in the directory server, you can use a combination of role options (`roleBase`, `roleSearchMatching`, `roleSearchSubtree`, and `roleName`) as an alternative to (or in addition to) specifying the `userRoleName` option. This option selects a particular subtree of the DIT to search for role/group entries. The subtree is specified by a DN, which specifes the base node of the subtree. For example, by setting this option to `ou=Group,ou=ActiveMQ,ou=system`, the search for role/group entries is restricted to the subtree beneath the `ou=Group,ou=ActiveMQ,ou=system` node.
- `roleName` - specifies the attribute type of the role entry that contains the name of the role/group (e.g. C, O, OU, etc.). If you omit this option the full DN of the role is used.
- `roleSearchMatching` - specifies an LDAP search filter, which is applied to the subtree selected by `roleBase`. This works in a similar manner to the `userSearchMatching` option, except that it supports two substitution strings, as follows:
  - `{0}` - substitutes the full DN of the matched user entry (that is, the result of the user search). For example, for the user, `jdoe`, the substituted string could be `uid=jdoe,ou=User,ou=ActiveMQ,ou=system`.
  - `{1}` - substitutes the received username. For example, `jdoe`.

    For example, if this option is set to `(member=uid={1})` and the received username is `jdoe`, the search filter becomes `(member=uid=jdoe)` after string substitution (assuming ApacheDS search filter syntax). If the resulting search filter is applied to the subtree selected by the role base,

`ou=Group,ou=ActiveMQ,ou=system`, it matches all role entries that have a `member` attribute equal to `uid=jdoe` (the value of a `member` attribute is a DN).

This option must always be set to enable role searching because it has no default value. Leaving it unset disables role searching and the role information must come from `userRoleName`.

If you use OpenLDAP, the syntax of the search filter is `(member:=uid=jdoe)`.

- `roleSearchSubtree` - specify the search depth for role entries, relative to the node specified by `roleBase`. This option can take boolean values, as follows:
  - `false` (default) - try to match one of the child entries of the roleBase node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`).
  - `true` — try to match any entry belonging to the subtree of the roleBase node (maps to `javax.naming.directory.SearchControls.SUBTREE_SCOPE`).
- `authenticateUser` - boolean flag to disable authentication. Useful as an optimisation when this module is used just for role mapping of a Subject's existing authenticated principals; default is `false`.
- `referral` - specify how to handle referrals; valid values: `ignore`, `follow`, `throw`; default is `ignore`.
- `ignorePartialResultException` - boolean flag for use when searching Active Directory (AD). AD servers don't handle referrals automatically, which causes a `PartialResultException` to be thrown when referrals are encountered by a search, even if `referral` is set to `ignore`. Set to `true` to ignore these exceptions; default is `false`.
- `expandRoles` - boolean indicating whether to enable the role expansion functionality or not; default false. If enabled, then roles within roles will be found. For example, role `A` is in role `B`. User `X` is in role `A`, which means user `X` is in role `B` by virtue of being in role `A`.
- `expandRolesMatching` - specifies an LDAP search filter which is applied to the subtree selected by `roleBase`. Before passing to the LDAP search operation, the string value you provide here is subjected to string substitution, as implemented by the `java.text.MessageFormat` class. Essentially, this means that the special string, `{0}`, is substituted by the role name as extracted from the previous role search. This option must always be set to enable role expansion because it has no default value. Example value: `(member={0})`.
- `debug` - boolean flag; if `true`, enable debugging; this is used only for testing or debugging; normally, it should be set to `false`, or omitted; default is `false`

Add user entries under the node specified by the `userBase` option. When creating a new user entry in the directory, choose an object class that supports the `userPassword` attribute (for example, the `person` or `inetOrgPerson` object classes are typically suitable). After creating the user entry, add the `userPassword` attribute, to hold the user's password.

If you want to store role data in dedicated role entries (where each node represents a particular role), create a role entry as follows. Create a new child of the `roleBase` node, where the `objectClass` of the child is `groupOfNames`. Set the `cn` (or whatever attribute type is specified by `roleName`) of the new child node equal to the name of the role/group. Define a `member` attribute for each member of the role/group, setting the `member` value to the DN of the corresponding user (where the DN is specified either fully, `uid=jdoe,ou=User,ou=ActiveMQ,ou=system`, or partially, `uid=jdoe`).

If you want to add roles to user entries, you would need to customize the directory schema, by adding a suitable attribute type to the user entry's object class. The chosen attribute type must be capable of handling multiple values.

CertificateLoginModule

The JAAS certificate authentication login module must be used in combination with SSL and the clients must be configured with their own certificate. In this scenario, authentication is actually performed during the SSL/TLS handshake, not directly by the JAAS certificate authentication plug-in. The role of the plug-in is as follows:

- To further constrain the set of acceptable users, because only the user DNs explicitly listed in the relevant properties file are eligible to be authenticated.
- To associate a list of groups with the received user identity, facilitating integration with the authorization feature.
- To require the presence of an incoming certificate (by default, the SSL/TLS layer is configured to treat the presence of a client certificate as optional).

The JAAS certificate login module stores a collection of certificate DNs in a pair of flat files. The files associate a username and a list of group IDs with each DN.

The certificate login module is implemented by the following class:

```
org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginMod
```

The following `CertLogin` login entry shows how to configure certificate login module in the login.config file:

```
CertLogin {
    org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLogi
        debug=true
        org.apache.activemq.jaas.textfiledn.user="users.properties"
        org.apache.activemq.jaas.textfiledn.role="roles.properties";
};
```

In the preceding example, the JAAS realm is configured to use a single `org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule` login module. The options supported by this login module are as follows:

- `debug` - boolean flag; if true, enable debugging; this is used only for testing or debugging; normally, it should be set to `false`, or omitted; default is `false`
- `org.apache.activemq.jaas.textfiledn.user` - specifies the location of the user properties file (relative to the directory containing the login configuration file).
- `org.apache.activemq.jaas.textfiledn.role` - specifies the location of the role properties file (relative to the directory containing the login configuration file).
- `reload` - boolean flag; whether or not to reload the properties files when a modification occurs; default is `false`

In the context of the certificate login module, the `users.properties` file consists of a list of properties of the form, `UserName=StringifiedSubjectDN` or `UserName=/SubjectDNRegExp/`. For example, to define the users, `system`, `user` and `guest` as well as a `hosts` user matching several DNs, you could create a file like the following:

```
system=CN=system,O=Progress,C=US
user=CN=humble user,O=Progress,C=US
guest=CN=anon,O=Progress,C=DE
hosts=/CN=host\\d+\\.acme\\.com,O=Acme,C=UK/
```

Note that the backslash character has to be escaped because it has a special treatment in properties files.

Each username is mapped to a subject DN, encoded as a string (where the string encoding is specified by RFC 2253). For example, the system username is mapped to the `CN=system,O=Progress,C=US` subject DN. When performing authentication, the plug-in extracts the subject DN from the received certificate, converts it to the standard string format, and compares it with the subject DNs in the `users.properties` file by testing for string equality. Consequently, you must be careful to ensure that the subject DNs appearing in the `users.properties` file are an exact match for the subject DNs extracted from the user certificates.

- `org.apache.activemq.jaas.textfiledn.user` - specifies the location of the user properties file (relative to the directory containing the login configuration file).
- `org.apache.activemq.jaas.textfiledn.role` - specifies the location of the role properties file (relative to the directory containing the login configuration file).
- `reload` - boolean flag; whether or not to reload the properties files when a modification occurs; default is `false`

In the context of the certificate login module, the `users.properties` file consists of a list of properties of the form, `UserName=StringifiedSubjectDN`. For example, to define the users, system, user, and guest, you could create a file like the following:

```
system=CN=system,O=Progress,C=US
user=CN=humble user,O=Progress,C=US
guest=CN=anon,O=Progress,C=DE
```

Each username is mapped to a subject DN, encoded as a string (where the string encoding is specified by RFC 2253). For example, the system username is mapped to the `CN=system,O=Progress,C=US` subject DN. When performing authentication, the plug-in extracts the subject DN from the received certificate, converts it to the standard string format, and compares it with the subject DNs in the `users.properties` file by testing for string equality. Consequently, you must be careful to ensure that the subject DNs appearing in the `users.properties` file are an exact match for the subject DNs extracted from the user certificates.

Note: Technically, there is some residual ambiguity in the DN string format. For example, the `domainComponent` attribute could be represented in a string either as the string, `DC`, or as the OID, `0.9.2342.19200300.100.1.25`. Normally, you do not need to worry about this ambiguity. But it could potentially be a problem, if you changed the underlying implementation of the Java security layer.

The easiest way to obtain the subject DNs from the user certificates is by invoking the `keytool` utility to print the certificate contents. To print the contents of a certificate in a keystore, perform the following steps:

1. Export the certificate from the keystore file into a temporary file. For example, to export the certificate with alias `broker-localhost` from the `broker.ks` keystore file, enter the following command:

   ```
   keytool -export -file broker.export -alias broker-localhost -keystore brok
   ```

   After running this command, the exported certificate is in the file, `broker.export`.

2. Print out the contents of the exported certificate. For example, to print out the contents of `broker.export`, enter the following command:

   ```
   keytool -printcert -file broker.export
   ```

   Which should produce output similar to that shown here:

   ```
   Owner: CN=localhost, OU=broker, O=Unknown, L=Unknown, ST=Unknown, C=Unknow
   Issuer: CN=localhost, OU=broker, O=Unknown, L=Unknown, ST=Unknown, C=Unkno
   Serial number: 4537c82e
   Valid from: Thu Oct 19 19:47:10 BST 2006 until: Wed Jan 17 18:47:10 GMT 20
   Certificate fingerprints:

            MD5:  3F:6C:0C:89:A8:80:29:CC:F5:2D:DA:5C:D7:3F:AB:37
            SHA1: F0:79:0D:04:38:5A:46:CE:86:E1:8A:20:1F:7B:AB:3A:46:E4:34:5C
   ```

   The string following `Owner:` gives the subject DN. The format used to enter the subject DN depends on your platform. The `Owner:` string above could be represented as either `CN=localhost,\ OU=broker,\ O=Unknown,\ L=Unknown,\`

`ST=Unknown,\ C=Unknown` or `CN=localhost,OU=broker,O=Unknown,L=Unknown,ST=Unknown,C=Unknown`.

The `roles.properties` file consists of a list of properties of the form, `Role=UserList`, where `UserList` is a comma-separated list of users. For example, to define the roles `admins`, `users`, and `guests`, you could create a file like the following:

```
admins=system
users=system,user
guests=guest
```

Krb5LoginModule

The Kerberos login module is used to propagate a validated SASL GSSAPI kerberos token identity into a validated JAAS UserPrincipal. This allows subsequent login modules to do role mapping for the kerberos identity.

```
org.apache.activemq.artemis.spi.core.security.jaas.Krb5LoginModule required
    ;
```

ExternalCertificateLoginModule

The external certificate login module is used to propagate a validated TLS client certificate's subjectDN into a JAAS UserPrincipal. This allows subsequent login modules to do role mapping for the TLS client certificate.

```
org.apache.activemq.artemis.spi.core.security.jaas.ExternalCertificateLoginMod
    ;
```

The simplest way to make the login configuration available to JAAS is to add the directory containing the file, `login.config`, to your CLASSPATH.

Kerberos Authentication

You must have the Kerberos infrastructure set up in your deployment environment before the server can accept Kerberos credentials. The server can acquire its Kerberos acceptor credentials by using JAAS and a Kerberos login module. The JDK provides the Krb5LoginModule which executes the necessary Kerberos protocol steps to authenticate and obtain Kerberos credentials.

GSSAPI SASL Mechanism

Using SASL over AMQP, Kerberos authentication is supported using the `GSSAPI` SASL mechanism. With SASL doing Kerberos authentication, TLS can be used to provide integrity and confidentially to the communications channel in the normal way.

The `GSSAPI` SASL mechanism must be enabled on the AMQP acceptor in `broker.xml` by adding it to the `saslMechanisms` list url parameter: `saslMechanisms="GSSAPI<,PLAIN, etc>`.

```
<acceptor name="amqp">tcp://0.0.0.0:5672?protocols=AMQP;saslMechanisms=GSSAPI<
```

The GSSAPI mechanism implementation on the server will use a JAAS configuration scope named `amqp-sasl-gssapi` to obtain its Kerberos acceptor credentials. An alternative configuration scope can be specified on the AMQP acceptor using the url parameter: `saslLoginConfigScope=<some other scope>`.

An example configuration scope for `login.config` that will pick up a Kerberos keyTab for the Kerberos acceptor Principal `amqp/localhost` is as follows:

```
amqp-sasl-gssapi {
    com.sun.security.auth.module.Krb5LoginModule required
    isInitiator=false
    storeKey=true
    useKeyTab=true
    principal="amqp/localhost"
    debug=true;
};
```

Role Mapping

On the server, the Kerberos authenticated Peer Principal can be added to the Subject's principal set as an Apache ActiveMQ Artemis UserPrincipal using the Apache ActiveMQ Artemis `Krb5LoginModule` login module. The PropertiesLoginModule or LDAPLoginModule can then be used to map the authenticated Kerberos Peer Principal to an Apache ActiveMQ Artemis Role. Note that the Kerberos Peer Principal does not exist as an Apache ActiveMQ Artemis user, only as a role member.

```
org.apache.activemq.artemis.spi.core.security.jaas.Krb5LoginModule required
    ;
org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule optional
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connectionURL="ldap://localhost:1024"
    authentication=GSSAPI
    saslLoginConfigScope=broker-sasl-gssapi
    connectionProtocol=s
    userBase="ou=users,dc=example,dc=com"
    userSearchMatching="(krb5PrincipalName={0})"
    userSearchSubtree=true
    authenticateUser=false
    roleBase="ou=system"
    roleName=cn
    roleSearchMatching="(member={0})"
    roleSearchSubtree=false
    ;
```

TLS Kerberos Cipher Suites

The legacy rfc2712 defines TLS Kerberos cipher suites that can be used by TLS to negotiate Kerberos authentication. The cypher suites offered by rfc2712 are dated and insecure and rfc2712 has been superseded by SASL GSSAPI. However, for clients that don't support SASL (core client), using TLS can provide Kerberos authentication over an *unsecure* channel.

Mapping external roles

Roles from external authentication providers (i.e. LDAP) can be mapped to internally used roles. The is done through role-mapping entries in the security-settings block:

```
<security-settings>
   [...]
   <role-mapping from="cn=admins,ou=Group,ou=ActiveMQ,ou=system" to="my-admin-
   <role-mapping from="cn=users,ou=Group,ou=ActiveMQ,ou=system" to="my-user-ro
</security-settings>
```

Note: Role mapping is additive. That means the user will keep the original role(s) as well as the newly assigned role(s).

Note: This role mapping only affects the roles which are used to authorize queue access through the configured acceptors. It can not be used to map the role required to access the web console.

SASL

AMQP supports SASL. The following mechanisms are supported; PLAIN, EXTERNAL, ANONYMOUS, GSSAPI. The published list can be constrained via the amqp acceptor `saslMechanisms` property. Note: EXTERNAL will only be chosen if a subject is available from the TLS client certificate.

Changing the username/password for clustering

In order for cluster connections to work correctly, each node in the cluster must make connections to the other nodes. The username/password they use for this should always be changed from the installation default to prevent a security risk.

Please see Management for instructions on how to do this.

Securing the console

Artemis comes with a web console that allows user to browse Artemis documentation via an embedded server. By default the web access is plain HTTP. It is configured in `bootstrap.xml` :

```
<web bind="http://localhost:8161" path="web">
    <app url="console" war="console.war"/>
</web>
```

Alternatively you can edit the above configuration to enable secure access using HTTPS protocol. e.g.:

```
<web bind="https://localhost:8443"
    path="web"
    keyStorePath="${artemis.instance}/etc/keystore.jks"
    keyStorePassword="password">
    <app url="jolokia" war="jolokia-war-1.3.5.war"/>
</web>
```

As shown in the example, to enable https the first thing to do is config the `bind` to be an `https` url. In addition, You will have to configure a few extra properties desribed as below.

- `keyStorePath` - The path of the key store file.
- `keyStorePassword` - The key store's password.
- `clientAuth` - The boolean flag indicates whether or not client authentication is required. Default is `false`.
- `trustStorePath` - The path of the trust store file. This is needed only if `clientAuth` is `true`.
- `trustStorePassword` - The trust store's password.

Controlling JMS ObjectMessage deserialization

Artemis provides a simple class filtering mechanism with which a user can specify which packages are to be trusted and which are not. Objects whose classes are from trusted packages can be deserialized without problem, whereas those from 'not trusted' packages will be denied deserialization.

Artemis keeps a `black list` to keep track of packages that are not trusted and a `white list` for trusted packages. By default both lists are empty, meaning any serializable object is allowed to be deserialized. If an object whose class matches one of the packages in black list, it is not allowed to be deserialized. If it matches one in the white list the object can be deserialized. If a package appears in both black list and white list, the one in black list takes precedence. If a class neither matches with `black list` nor with the `white list`, the class deserialization will be denied unless the white list is empty (meaning the user doesn't specify the white list at all).

A class is considered as a 'match' if

- its full name exactly matches one of the entries in the list.

- its package matches one of the entries in the list or is a sub-package of one of the entries.

For example, if a class full name is "org.apache.pkg1.Class1", some matching entries could be:

- `org.apache.pkg1.Class1` - exact match.
- `org.apache.pkg1` - exact package match.
- `org.apache` -- sub package match.

A `*` means 'match-all' in a black or white list.

Config via Connection Factories

To specify the white and black lists one can use the URL parameters `deserializationBlackList` and `deserializationWhiteList`. For example, using JMS:

```
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("vm://0?dese
```

The above statement creates a factory that has a black list contains two forbidden packages, "org.apache.pkg1" and "org.some.pkg2", separated by a comma.

Config via system properties

There are two system properties available for specifying black list and white list:

- `org.apache.activemq.artemis.jms.deserialization.whitelist` - comma separated list of entries for the white list.
- `org.apache.activemq.artemis.jms.deserialization.blacklist` - comma separated list of entries for the black list.

Once defined, all JMS object message deserialization in the VM is subject to checks against the two lists. However if you create a ConnectionFactory and set a new set of black/white lists on it, the new values will override the system properties.

Config for resource adapters

Message beans using a JMS resource adapter to receive messages can also control their object deserialization via properly configuring relevant properties for their resource adapters. There are two properties that you can configure with connection factories in a resource adapter:

- `deserializationBlackList` - comma separated values for black list
- `deserializationWhiteList` - comma separated values for white list

These properties, once specified, are eventually set on the corresponding internal factories.

Config for REST interface

Apache Artemis REST interface (Rest) allows interactions between jms client and rest clients. It uses JMS ObjectMessage to wrap the actual user data between the 2 types of clients and deserialization is needed during this process. If you want to control the deserialization for REST, you need to set the black/white lists for it separately as Apache Artemis REST Interface is deployed as a web application. You need to put the black/white lists in its web.xml, as context parameters, as follows

```
<web-app>
    <context-param>
        <param-name>org.apache.activemq.artemis.jms.deserialization.whitelist<
        <param-value>some.allowed.class</param-value>
    </context-param>
    <context-param>
        <param-name>org.apache.activemq.artemis.jms.deserialization.blacklist<
        <param-value>some.forbidden.class</param-value>
    </context-param>
...
</web-app>
```

The param-value for each list is a comma separated string value representing the list.

Masking Passwords

For details about masking passwords in broker.xml please see the Masking Passwords chapter.

Custom Security Manager

The underpinnings of the broker's security implementation can be changed if so desired. The broker uses a component called a "security manager" to implement the actual authentication and authorization checks. By default, the broker uses `org.apache.activemq.artemis.spi.core.security.ActiveMQJAASSecurityManager` to provide JAAS integration, but users can provide their own implementation of `org.apache.activemq.artemis.spi.core.security.ActiveMQSecurityManager3` and configure it in `bootstrap.xml` using the `security-manager` element, e.g.:

```
<broker xmlns="http://activemq.org/schema">

   <security-manager class-name="com.foo.MySecurityManager">
      <property key="myKey1" value="myValue1"/>
      <property key="myKey2" value="myValue2"/>
   </security-manager>

   ...
</broker>
```

The `security-manager` example demonstrates how to do this is more detail.

Per-Acceptor Security Domains

It's possible to override the broker's JAAS security domain by specifying a security domain on an individual `acceptor`. Simply use the `securityDomain` parameter and indicate which domain from your `login.config` to use, e.g.:

```
<acceptor name="myAcceptor">tcp://127.0.0.1:61616?securityDomain=mySecurityDom
```

Any client connecting to this acceptor will be have security enforced using `mySecurityDomain`.

Masking Passwords

By default all passwords in Apache ActiveMQ Artemis server's configuration files are in plain text form. This usually poses no security issues as those files should be well protected from unauthorized accessing. However, in some circumstances a user doesn't want to expose its passwords to more eyes than necessary.

Apache ActiveMQ Artemis can be configured to use 'masked' passwords in its configuration files. A masked password is an obscure string representation of a real password. To mask a password a user will use an 'codec'. The codec takes in the real password and outputs the masked version. A user can then replace the real password in the configuration files with the new masked password. When Apache ActiveMQ Artemis loads a masked password it uses the codec to decode it back into the real password.

Apache ActiveMQ Artemis provides a default codec. Optionally users can use or implement their own codec for masking the passwords.

In general, a masked password can be identified using one of two ways. The first one is the `ENC()` syntax, i.e. any string value wrapped in `ENC()` is to be treated as a masked password. For example

```
ENC(xyz)
```

The above indicates that the password is masked and the masked value is `xyz`.

The `ENC()` syntax is the **preferred way** of masking a password and is universally supported in every password configuration in Artemis.

The other, legacy way is to use a `mask-password` attribute to tell that a password in a configuration file should be treated as 'masked'. For example:

```
<mask-password>true</mask-password>
<cluster-password>xyz</cluster-password>
```

This method is now **deprecated** and exists only to maintain backward-compatibility. Newer configurations may not support it.

Generating a Masked Password

To get a mask for a password using the broker's default codec run the `mask` command from your Artemis *instance*. This command will not work from the Artemis home:

```
./artemis mask <plaintextPassword>
```

You'll get something like

```
result: 32c6f67dae6cd61b0a7ad1702033aa81e6b2a760123f4360
```

Just copy `32c6f67dae6cd61b0a7ad1702033aa81e6b2a760123f4360` and replace your plaintext password with it using the `ENC()` syntax, e.g. `ENC(32c6f67dae6cd61b0a7ad1702033aa81e6b2a760123f4360)`.

This process works for passwords in:

- `broker.xml`
- `login.config`
- `bootstrap.xml`
- `management.xml`

This process does **not** work for passwords in:

- `artemis-users.properties`

Maksed passwords for `artemis-users.properties` *can* be generated using the `mask` command using the `--hash` command-line option. However, we recommend using the set of tools provided by the `user` command described below.

Masking Configuration

Besides supporting the `ENC()` syntax, the server configuration file (i.e. broker.xml) has a property that defines the default masking behaviors over the entire file scope.

`mask-password` : this boolean type property indicates if a password should be masked or not. Set it to "true" if you want your passwords masked. The default value is "false". As noted above, this configuration parameter is deprecated.

`password-codec` : this string type property identifies the name of the class which will be used to decode the masked password within the broker. If not specified then the default `org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec` will be used.

artemis-users.properties

Apache ActiveMQ Artemis's built-in security manager uses plain properties files where the user passwords are specified in a hashed form by default. Note, the passwords are technically *hashed* rather than masked in this context. The default `PropertiesLoginModule` will not decode the passwords in `artemis-users.properties` but will instead hash the input and compare the two hashed values for password verification.

Use the following command from the CLI of the Aremtis *instance* you wish to add the user/password to. This command will not work from the Artemis home used to create the instance. For example:

```
./artemis user add --user guest --password guest --role admin
```

This will use the default codec to perform a "one-way" hash of the password and alter both the `artemis-users.properties` and `artemis-roles.properties` files with the specified values.

Passwords in `artemis-users.properties` are automatically detected as hashed or not by looking for the syntax `ENC(<hash>)`. The `mask-password` parameter does not need to be `true` to use hashed passwords here.

cluster-password

If it is specified in `ENC()` syntax it will be treated as masked, or if `mask-password` is `true` the `cluster-password` will be treated as masked.

Connectors & Acceptors

In broker.xml `connector` and `acceptor` configurations sometimes needs to specify passwords. For example, if a user wants to use an `acceptor` with `sslEnabled=true` it can specify `keyStorePassword` and `trustStorePassword`. Because Acceptors and Connectors are pluggable implementations, each transport will have different password masking needs.

The preferred way is simply to use the `ENC()` syntax.

If using the legacy `mask-password` and `password-codec` values then when a `connector` or `acceptor` is initialised, Apache ActiveMQ Artemis will add these values to the parameters using the keys `activemq.usemaskedpassword` and `activemq.passwordcodec` respectively. The Netty and InVM implementations will use these as needed and any other implementations will have access to these to use if they so wish.

Core Bridges

Core Bridges are configured in the server configuration file and so the masking of its `password` properties follows the same rules as that of `cluster-password`. It supports `ENC()` syntax.

For using `mask-password` property, the following table summarizes the relations among the above-mentioned properties

mask-password	cluster-password	acceptor/connector passwords	bridge password
absent	plain text	plain text	plain text
false	plain text	plain text	plain text
true	masked	masked	masked

It is recommended that you use the `ENC()` syntax for new applications/deployments.

Examples

Note: In the following examples if related attributed or properties are absent, it means they are not specified in the configure file.

- Unmasked

```
<cluster-password>bbc</cluster-password>
```

  This indicates the cluster password is a plain text value `bbc`.

- Masked 1

```
<cluster-password>ENC(80cf731af62c290)</cluster-password>
```

  This indicates the cluster password is a masked value `80cf731af62c290`.

- Masked 2

```
<mask-password>true</mask-password>
<cluster-password>80cf731af62c290</cluster-password>
```

  This indicates the cluster password is a masked value and Apache ActiveMQ Artemis will use its built-in codec to decode it. All other passwords in the configuration file, Connectors, Acceptors and Bridges, will also use masked passwords.

bootstrap.xml

The broker embeds a web-server for hosting some web applications such as a management console. It is configured in `bootstrap.xml` as a web component. The web server can be secured using the `https` protocol, and it can be configured with a keystore password and/or truststore password which by default are specified in plain text forms.

To mask these passwords you need to use `ENC()` syntax. The `mask-password` boolean is not supported here.

You can also set the `passwordCodec` attribute if you want to use a password codec other than the default one. For example

```
<web bind="https://localhost:8443" path="web"
     keyStorePassword="ENC(-5a2376c61c668aaf)"
     trustStorePassword="ENC(3d617352d12839eb71208edf41d66b34)">
    <app url="activemq-branding" war="activemq-branding.war"/>
</web>
```

management.xml

The broker embeds a JMX connector which is used for management. The connector can be secured using SSL and it can be configured with a keystore password and/or truststore password which by default are specified in plain text forms.

To mask these passwords you need to use `ENC()` syntax. The `mask-password` boolean is not supported here.

You can also set the `password-codec` attribute if you want to use a password codec other than the default one. For example

```
<connector
      connector-port="1099"
      connector-host="localhost"
      secured="true"
      key-store-path="myKeystore.jks"
      key-store-password="ENC(3a34fd21b82bf2a822fa49a8d8fa115d"
      trust-store-path="myTruststore.jks"
      trust-store-password="ENC(3a34fd21b82bf2a822fa49a8d8fa115d)"/>
```

With this configuration, both passwords in ra.xml and all of its MDBs will have to be in masked form.

login.config

Artemis supports LDAP login modules to be configured in JAAS configuration file (default name is `login.config`). When connecting to a LDAP server usually you need to supply a connection password in the config file. By default this password is in plain text form.

To mask it you need to configure the passwords in your login module using `ENC()` syntax. To specify a codec using the following property:

`passwordCodec` - the password codec class name. (the default codec will be used if it is absent)

For example:

```
LDAPLoginExternalPasswordCodec {
    org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule require
        debug=true
        initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
        connectionURL="ldap://localhost:1024"
        connectionUsername="uid=admin,ou=system"
        connectionPassword="ENC(-170b9ef34d79ed12)"
        passwordCodec="org.apache.activemq.artemis.utils.DefaultSensitiveStrin
        connectionProtocol=s
        authentication=simple
        userBase="ou=system"
        userSearchMatching="(uid={0})"
        userSearchSubtree=false
        roleBase="ou=system"
        roleName=dummyRoleName
        roleSearchMatching="(uid={1})"
        roleSearchSubtree=false
        ;
};
```

JCA Resource Adapter

Both ra.xml and MDB activation configuration have a `password` property that can be masked preferably using `ENC()` syntax.

Alternatively it can use an optional attribute in ra.xml to indicate that a password is masked:

`UseMaskedPassword` -- If setting to "true" the passwords are masked. Default is false.

There is another property in ra.xml that can specify a codec:

`PasswordCodec` -- Class name and its parameters for the codec used to decode the masked password. Ignored if UseMaskedPassword is false. The format of this property is a full qualified class name optionally followed by key/value pairs. It is the same format as that for JMS Bridges. Example:

Example 1 Using the `ENC()` syntax:

```
<config-property>
  <config-property-name>password</config-property-name>
  <config-property-type>String</config-property-type>
  <config-property-value>ENC(80cf731af62c290)</config-property-value>
</config-property>
<config-property>
  <config-property-name>PasswordCodec</config-property-name>
  <config-property-type>java.lang.String</config-property-type>
  <config-property-value>com.foo.ACodec;key=helloworld</config-property-value>
</config-property>
```

Example 2 Using the "UseMaskedPassword" property:

```
<config-property>
  <config-property-name>UseMaskedPassword</config-property-name>
  <config-property-type>boolean</config-property-type>
  <config-property-value>true</config-property-value>
</config-property>
<config-property>
  <config-property-name>password</config-property-name>
  <config-property-type>String</config-property-type>
  <config-property-value>80cf731af62c290</config-property-value>
</config-property>
<config-property>
  <config-property-name>PasswordCodec</config-property-name>
  <config-property-type>java.lang.String</config-property-type>
  <config-property-value>com.foo.ACodec;key=helloworld</config-property-value>
</config-property>
```

Choosing a codec for password masking

As described in the previous sections, all password masking requires a codec. A codec uses an algorithm to convert a masked password into its original clear text form in order to be used in various security operations. The algorithm used for decoding must match that for encoding. Otherwise the decoding may not be successful.

For user's convenience Apache ActiveMQ Artemis provides a default codec. However a user can implement their own if they wish.

The Default Codec

Whenever no codec is specified in the configuration, the default codec is used. The class name for the default codec is `org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec`. It has hashing, encoding, and decoding capabilities. It uses `java.crypto.Cipher` utilities to hash or encode a plaintext password and also to decode a masked string using the same algorithm and key.

Using a custom codec

It is possible to use a custom codec rather than the built-in one. Simply make sure the codec is in Apache ActiveMQ Artemis's classpath. The custom codec can also be service loaded rather than class loaded, if the codec's service provider is installed in the classpath. Then configure the server to use it as follows:

```
<password-codec>com.foo.SomeCodec;key1=value1;key2=value2</password-codec>
```

If your codec needs params passed to it you can do this via key/value pairs when configuring. For instance if your codec needs say a "key-location" parameter, you can define like so:

```
<password-codec>com.foo.NewCodec;key-location=/some/url/to/keyfile</password-c
```

Then configure your cluster-password like this:

```
<cluster-password>ENC(masked_password)</cluster-password>
```

When Apache ActiveMQ Artemis reads the cluster-password it will initialize the `NewCodec` and use it to decode "mask_password". It also process all passwords using the new defined codec.

Implementing Custom Codecs

To use a different codec than the built-in one, you either pick one from existing libraries or you implement it yourself. All codecs must implement the `org.apache.activemq.artemis.utils.SensitiveDataCodec<T>` interface:

```
public interface SensitiveDataCodec<T> {

   T decode(Object mask) throws Exception;

   T encode(Object secret) throws Exception;

   default void init(Map<String, String> params) throws Exception {
   };
}
```

This is a generic type interface but normally for a password you just need String type. So a new codec would be defined like

```
public class MyCodec implements SensitiveDataCodec<String> {
   @Override
   public String decode(Object mask) throws Exception {
      // Decode the mask into clear text password.
      return "the password";
   }

   @Override
   public String encode(Object secret) throws Exception {
      // Mask the clear text password.
      return "the masked password"";
   }

   @Override
   public void init(Map<String, String> params) {
      // Initialization done here. It is called right after the codec has been
   }
}
```

Last but not least, once you get your own codec please add it to the classpath otherwise the broker will fail to load it!

Apache ActiveMQ Artemis Plugin Support

Apache ActiveMQ Artemis is designed to allow extra functionality to be added by creating a plugin. Multiple plugins can be registered at the same time and they will be chained together and executed in the order they are registered (i.e. the first plugin registered is always executed first).

Creating a plugin is very simple. It requires:

- Implementing the `ActiveMQServerPlugin` interface
- Making sure the plugin is on the classpath
- Registering it with the broker either via xml or programmatically.

Only the methods that you want to add behavior for need to be implemented as all of the interface methods are default methods.

Registering a Plugin

To register a plugin with by XML you need to add the `broker-plugins` element at the `broker.xml`. It is also possible to pass configuration to a plugin using the `property` child element(s). These properties (zero to many) will be read and passed into the plugin's `init(Map<String, String>)` operation after the plugin has been instantiated.

```
<broker-plugins>
   <broker-plugin class-name="some.plugin.UserPlugin">
      <property key="property1" value="val_1" />
      <property key="property2" value="val_2" />
   </broker-plugin>
</broker-plugins>
```

Registering a Plugin Programmatically

For registering a plugin programmatically you need to call the `registerBrokerPlugin()` method and pass in a new instance of your plugin. In the example below assuming your plugin is called `UserPlugin`, registering it looks like the following:

```
...

Configuration config = new ConfigurationImpl();
...

config.registerBrokerPlugin(new UserPlugin());
```

Using the `LoggingActiveMQServerPlugin`

The `LoggingActiveMQServerPlugin` logs specific broker events.

You can select which events are logged by setting the following configuration properties to `true` .

Property	Trigger Event	Default Value
`LOG_CONNECTION_EVENTS`	Connection is created/destroy.	`false`
`LOG_SESSION_EVENTS`	Session is created/closed.	`false`
`LOG_CONSUMER_EVENTS`	Consumer is created/closed	`false`
`LOG_DELIVERING_EVENTS`	Message is delivered to a consumer and when a message is acknowledged by a consumer.	`false`
`LOG_SENDING_EVENTS`	When a message has been sent to an address and when a message has been routed within the broker.	`false`
`LOG_INTERNAL_EVENTS`	When a queue created/destroyed, when a message is expired, when a bridge is deployed and when a critical failure occurs.	`false`
`LOG_ALL_EVENTS`	Includes all the above events.	`false`

By default the `LoggingActiveMQServerPlugin` will not log any information. The logging is activated by setting one (or a selection) of the above configuration properties to `true` .

To configure the plugin, you can add the following configuration to the broker. In the example below both `LOG_DELIVERING_EVENTS` and `LOG_SENDING_EVENTS` will be logged by the broker.

```
<broker-plugins>
   <broker-plugin class-name="org.apache.activemq.artemis.core.server.plugin.i
      <property key="LOG_DELIVERING_EVENTS" value="true" />
      <property key="LOG_SENDING_EVENTS" value="true" />
   </broker-plugin>
</broker-plugins>
```

Most events in the `LoggingActiveMQServerPlugin` follow a `beforeX` and `afterX` notification pattern (e.g `beforeCreateConsumer()` and `afterCreateConsumer()`).

At Log Level `INFO` , the LoggingActiveMQServerPlugin logs an entry when an `afterX` notification occurs. By setting the logger `org.apache.activemq.artemis.core.server.plugin.impl` to `DEBUG` , log entries are generated for both `beforeX` and `afterX` notifications. Log level `DEBUG` will also log more information for a notification when available.

Using the NotificationActiveMQServerPlugin

The NotificationActiveMQServerPlugin can be configured to send extra notifications for specific broker events.

You can select which notifications are sent by setting the following configuration properties to `true`.

Property	Property Description	Default Value
`SEND_CONNECTION_NOTIFICATIONS`	Sends a notification when a Connection is created/destroy.	`false`.
`SEND_SESSION_NOTIFICATIONS`	Sends a notification when a Session is created/closed.	`false`.
`SEND_ADDRESS_NOTIFICATIONS`	Sends a notification when an Address is added/removed.	`false`.
`SEND_DELIVERED_NOTIFICATIONS`	Sends a notification when message is delivered to a consumer.	`false`
`SEND_EXPIRED_NOTIFICATIONS`	Sends a notification when message has been expired by the broker.	`false`

By default the NotificationActiveMQServerPlugin will not send any notifications. The plugin is activated by setting one (or a selection) of the above configuration properties to `true`.

To configure the plugin, you can add the following configuration to the broker. In the example below both `SEND_CONNECTION_NOTIFICATIONS` and `SEND_SESSION_NOTIFICATIONS` will be sent by the broker.

```
<broker-plugins>
   <broker-plugin class-name="org.apache.activemq.artemis.core.server.plugin.i
      <property key="SEND_CONNECTION_NOTIFICATIONS" value="true" />
      <property key="SEND_SESSION_NOTIFICATIONS" value="true" />
   </broker-plugin>
</broker-plugins>
```

Resource Limits

Sometimes it's helpful to set particular limits on what certain users can do beyond the normal security settings related to authorization and authentication. For example, limiting how many connections a user can create or how many queues a user can create. This chapter will explain how to configure such limits.

Configuring Limits Via Resource Limit Settings

Here is an example of the XML used to set resource limits:

```
<resource-limit-settings>
   <resource-limit-setting match="myUser">
      <max-connections>5</max-connections>
      <max-queues>3</max-queues>
   </resource-limit-setting>
</resource-limit-settings>
```

Unlike the `match` from `address-setting`, this `match` does not use any wild-card syntax. It's a simple 1:1 mapping of the limits to a **user**.

- `max-connections` defines how many connections the matched user can make to the broker. The default is -1 which means there is no limit.
- `max-queues` defines how many queues the matched user can create. The default is -1 which means there is no limit.

The JMS Bridge

Apache ActiveMQ Artemis includes a fully functional JMS message bridge.

The function of the bridge is to consume messages from a source queue or topic, and send them to a target queue or topic, typically on a different server.

> **Note:**
>
> The JMS Bridge is not intended as a replacement for transformation and more expert systems such as Camel. The JMS Bridge may be useful for fast transfers as this chapter covers, but keep in mind that more complex scenarios requiring transformations will require you to use a more advanced transformation system that will play on use cases that will go beyond Apache ActiveMQ Artemis.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, and where the connection may be unreliable.

A bridge can be deployed as a standalone application or as a web application managed by the embedded Jetty instance bootstrapped with Apache ActiveMQ Artemis. The source and the target can be located in the same virtual machine or another one.

The bridge can also be used to bridge messages from other non Apache ActiveMQ Artemis JMS servers, as long as they are JMS 1.1 compliant.

> **Note:**
>
> Do not confuse a JMS bridge with a core bridge. A JMS bridge can be used to bridge any two JMS 1.1 compliant JMS providers and uses the JMS API. A core bridge) is used to bridge any two Apache ActiveMQ Artemis instances and uses the core API. Always use a core bridge if you can in preference to a JMS bridge. The core bridge will typically provide better performance than a JMS bridge. Also the core bridge can provide *once and only once* delivery guarantees without using XA.

The bridge has built-in resilience to failure so if the source or target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the source and/or target until they come back online. When it comes back online it will resume operation as normal.

The bridge can be configured with an optional JMS selector, so it will only consume messages matching that JMS selector

It can be configured to consume from a queue or a topic. When it consumes from a topic it can be configured to consume using a non durable or durable subscription

The JMS Bridge is a simple POJO so can be deployed with most frameworks, simply instantiate the `org.apache.activemq.artemis.api.jms.bridge.impl.JMSBridgeImpl` class and set the appropriate parameters.

JMS Bridge Parameters

The main POJO is the `JMSBridge`. It is is configurable by the parameters passed to its constructor.

- Source Connection Factory Factory

  This injects the `SourceCFF` bean (also defined in the beans file). This bean is used to create the *source* `ConnectionFactory`

- Target Connection Factory Factory

  This injects the `TargetCFF` bean (also defined in the beans file). This bean is used to create the *target* `ConnectionFactory`

- Source Destination Factory Factory

  This injects the `SourceDestinationFactory` bean (also defined in the beans file). This bean is used to create the *source* `Destination`

- Target Destination Factory Factory

  This injects the `TargetDestinationFactory` bean (also defined in the beans file). This bean is used to create the *target* `Destination`

- Source User Name

  this parameter is the username for creating the *source* connection

- Source Password

  this parameter is the parameter for creating the *source* connection

- Target User Name

  this parameter is the username for creating the *target* connection

- Target Password

  this parameter is the password for creating the *target* connection

- Selector

  This represents a JMS selector expression used for consuming messages from the source destination. Only messages that match the selector expression will be bridged from the source to the target destination

  The selector expression must follow the JMS selector syntax

- Failure Retry Interval

This represents the amount of time in ms to wait between trying to recreate connections to the source or target servers when the bridge has detected they have failed

- Max Retries

  This represents the number of times to attempt to recreate connections to the source or target servers when the bridge has detected they have failed. The bridge will give up after trying this number of times. `-1` represents 'try forever'

- Quality Of Service

  This parameter represents the desired quality of service mode

  Possible values are:

  - `AT_MOST_ONCE`
  - `DUPLICATES_OK`
  - `ONCE_AND_ONLY_ONCE`

  See Quality Of Service section for an explanation of these modes.

- Max Batch Size

  This represents the maximum number of messages to consume from the source destination before sending them in a batch to the target destination. Its value must `>= 1`

- Max Batch Time

  This represents the maximum number of milliseconds to wait before sending a batch to target, even if the number of messages consumed has not reached `MaxBatchSize`. Its value must be `-1` to represent 'wait forever', or `>= 1` to specify an actual time

- Subscription Name

  If the source destination represents a topic, and you want to consume from the topic using a durable subscription then this parameter represents the durable subscription name

- Client ID

  If the source destination represents a topic, and you want to consume from the topic using a durable subscription then this attribute represents the the JMS client ID to use when creating/looking up the durable subscription

- Add MessageID In Header

  If `true`, then the original message's message ID will be appended in the message sent to the destination in the header `ACTIVEMQ_BRIDGE_MSG_ID_LIST`. If the message is bridged more than once, each message ID will be appended. This enables a distributed request-response pattern to be used

> **Note:**
>
> when you receive the message you can send back a response using the correlation id of the first message id, so when the original sender gets it back it will be able to correlate it.

- MBean Server

  To manage the JMS Bridge using JMX, set the MBeanServer where the JMS Bridge MBean must be registered (e.g. the JVM Platform MBeanServer)

- ObjectName

  If you set the MBeanServer, you also need to set the ObjectName used to register the JMS Bridge MBean (must be unique)

The "transactionManager" property points to a JTA transaction manager implementation and should be set if you need to use the 'ONCE_AND_ONCE_ONLY' Quality of Service. Apache ActiveMQ Artemis doesn't ship with such an implementation, but if you are running within an Application Server you can inject the Transaction Manager that is shipped.

Source and Target Connection Factories

The source and target connection factory factories are used to create the connection factory used to create the connection for the source or target server.

The configuration example above uses the default implementation provided by Apache ActiveMQ Artemis that looks up the connection factory using JNDI. For other Application Servers or JMS providers a new implementation may have to be provided. This can easily be done by implementing the interface `org.apache.activemq.artemis.jms.bridge.ConnectionFactoryFactory`.

Source and Target Destination Factories

Again, similarly, these are used to create or lookup up the destinations.

In the configuration example above, we have used the default provided by Apache ActiveMQ Artemis that looks up the destination using JNDI.

A new implementation can be provided by implementing `org.apache.activemq.artemis.jms.bridge.DestinationFactory` interface.

Quality Of Service

The quality of service modes used by the bridge are described here in more detail.

AT_MOST_ONCE

With this QoS mode messages will reach the destination from the source at most once. The messages are consumed from the source and acknowledged before sending to the destination. Therefore there is a possibility that if failure occurs between removing them from the source and them arriving at the destination they could be lost. Hence delivery will occur at most once.

This mode is available for both durable and non-durable messages.

DUPLICATES_OK

With this QoS mode, the messages are consumed from the source and then acknowledged after they have been successfully sent to the destination. Therefore there is a possibility that if failure occurs after sending to the destination but before acknowledging them, they could be sent again when the system recovers. I.e. the destination might receive duplicates after a failure.

This mode is available for both durable and non-durable messages.

ONCE_AND_ONLY_ONCE

This QoS mode ensures messages will reach the destination from the source once and only once. (Sometimes this mode is known as "exactly once"). If both the source and the destination are on the same Apache ActiveMQ Artemis server instance then this can be achieved by sending and acknowledging the messages in the same local transaction. If the source and destination are on different servers this is achieved by enlisting the sending and consuming sessions in a JTA transaction. The JTA transaction is controlled by a JTA Transaction Manager which will need to be set via the settransactionManager method on the Bridge.

This mode is only available for durable messages.

> **Note:**
>
> For a specific application it may possible to provide once and only once semantics without using the ONCE_AND_ONLY_ONCE QoS level. This can be done by using the DUPLICATES_OK mode and then checking for duplicates at the destination and discarding them. Some JMS servers provide automatic duplicate message detection functionality, or this may be possible to implement on the application level by maintaining a cache of received message ids on disk and comparing received messages to them. The cache would only be valid for a certain period of time so this approach is not as watertight as using ONCE_AND_ONLY_ONCE but may be a good choice depending on your specific application.

Time outs and the JMS bridge

There is a possibility that the target or source server will not be available at some point in time. If this occurs then the bridge will try `Max Retries` to reconnect every `Failure Retry Interval` milliseconds as specified in the JMS Bridge definition.

If you implement your own factories for looking up JMS resources then you will have to bear in mind timeout issues.

Examples

Please see JMS Bridge Example which shows how to programmatically instantiate and configure a JMS Bridge to send messages to the source destination and consume them from the target destination between two standalone Apache ActiveMQ Artemis brokers.

Client Reconnection and Session Reattachment

Apache ActiveMQ Artemis clients can be configured to automatically reconnect or re-attach to the server in the event that a failure is detected in the connection between the client and the server.

100% Transparent session re-attachment

If the failure was due to some transient failure such as a temporary network failure, and the target server was not restarted, then the sessions will still be existent on the server, assuming the client hasn't been disconnected for more than connection-ttl

In this scenario, Apache ActiveMQ Artemis will automatically re-attach the client sessions to the server sessions when the connection reconnects. This is done 100% transparently and the client can continue exactly as if nothing had happened.

The way this works is as follows:

As Apache ActiveMQ Artemis clients send commands to their servers they store each sent command in an in-memory buffer. In the case that connection failure occurs and the client subsequently reattaches to the same server, as part of the reattachment protocol the server informs the client during reattachment with the id of the last command it successfully received from that client.

If the client has sent more commands than were received before failover it can replay any sent commands from its buffer so that the client and server can reconcile their states.Ac

The size of this buffer is configured with the `confirmationWindowSize` parameter on the connection URL. When the server has received `confirmationWindowSize` bytes of commands and processed them it will send back a command confirmation to the client, and the client can then free up space in the buffer.

The window is specified in bytes.

Setting this parameter to `-1` disables any buffering and prevents any re-attachment from occurring, forcing reconnect instead. The default value for this parameter is `-1`. (Which means by default no auto re-attachment will occur)

Session reconnection

Alternatively, the server might have actually been restarted after crashing or being stopped. In this case any sessions will no longer be existent on the server and it won't be possible to 100% transparently re-attach to them.

In this case, Apache ActiveMQ Artemis will automatically reconnect the connection and *recreate* any sessions and consumers on the server corresponding to the sessions and consumers on the client. This process is exactly the same as what happens during failover onto a backup server.

Client reconnection is also used internally by components such as core bridges to allow them to reconnect to their target servers.

Please see the section on failover Automatic Client Failover to get a full understanding of how transacted and non-transacted sessions are reconnected during failover/reconnect and what you need to do to maintain *once and only once* delivery guarantees.

Configuring reconnection/reattachment attributes

Client reconnection is configured using the following parameters:

- `retryInterval`. This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `2000` milliseconds.
- `retryIntervalMultiplier`. This optional parameter determines determines a multiplier to apply to the time since the last retry to compute the time to the next retry.

  This allows you to implement an *exponential backoff* between retry attempts.

  Let's take an example:

  If we set `retryInterval` to `1000` ms and we set `retryIntervalMultiplier` to `2.0`, then, if the first reconnect attempt fails, we will wait `1000` ms then `2000` ms then `4000` ms between subsequent reconnection attempts.

  The default value is `1.0` meaning each reconnect attempt is spaced at equal intervals.
- `maxRetryInterval`. This optional parameter determines the maximum retry interval that will be used. When setting `retryIntervalMultiplier` it would otherwise be possible that subsequent retries exponentially increase to ridiculously large values. By setting this parameter you can set an upper limit on that value. The default value is `2000` milliseconds.
- `reconnectAttempts`. This optional parameter determines the total number of reconnect attempts to make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `0`.

All of these parameters are set on the URL used to connect to the broker.

If your client does manage to reconnect but the session is no longer available on the server, for instance if the server has been restarted or it has timed out, then the client won't be able to re-attach, and any `ExceptionListener` or `FailureListener` instances registered on the connection or session will be called.

ExceptionListeners and SessionFailureListeners

Please note, that when a client reconnects or re-attaches, any registered JMS `ExceptionListener` or core API `SessionFailureListener` will be called.

Diverting and Splitting Message Flows

Apache ActiveMQ Artemis allows you to configure objects called *diverts* with some simple server configuration.

Diverts allow you to transparently divert messages routed to one address to some other address, without making any changes to any client application logic.

Diverts can be *exclusive*, meaning that the message is diverted to the new address, and does not go to the old address at all, or they can be *non-exclusive* which means the message continues to go the old address, and a *copy* of it is also sent to the new address. Non-exclusive diverts can therefore be used for *splitting* message flows, e.g. there may be a requirement to monitor every order sent to an order queue.

When an address has both exclusive and non-exclusive diverts configured, the exclusive ones are processed first. If any of the exclusive diverts diverted the message, the non-exclusive ones are not processed.

Diverts can also be configured to have an optional message filter. If specified then only messages that match the filter will be diverted.

Diverts can apply a particular routing-type to the message, strip the existing routing type, or simply pass the existing routing-type through. This is useful in situations where the message may have its routing-type set but you want to divert it to an address using a different routing-type. It's important to keep in mind that a message with the `anycast` routing-type will not actually be routed to queues using `multicast` and vice-versa. By configuring the `routing-type` of the divert you have the flexibility to deal with any situation. Valid values are `ANYCAST`, `MULTICAST`, `PASS`, & `STRIP`. The default is `STRIP`.

Diverts can also be configured to apply a `Transformer`. If specified, all diverted messages will have the opportunity of being transformed by the `Transformer`. When an address has multiple diverts configured, all of them receive the same, original message. This means that the results of a transformer on a message are not directly available for other diverts or their filters on the same address.

See the documentation on adding runtime dependencies to understand how to make your transformer available to the broker.

A divert will only divert a message to an address on the *same server*, however, if you want to divert to an address on a different server, a common pattern would be to divert to a local store-and-forward queue, then set up a bridge which consumes from that queue and forwards to an address on a different server.

Diverts are therefore a very sophisticated concept, which when combined with bridges can be used to create interesting and complex routings. The set of diverts on a server can be thought of as a type of routing table for messages. Combining

diverts with bridges allows you to create a distributed network of reliable routing connections between multiple geographically distributed servers, creating your global messaging mesh.

Diverts are defined as xml in the `broker.xml` file at the `core` attribute level. There can be zero or more diverts in the file.

Diverted messages get special properties.

Please see the examples for a full working example showing you how to configure and use diverts.

Let's take a look at some divert examples:

Exclusive Divert

Let's take a look at an exclusive divert. An exclusive divert diverts all matching messages that are routed to the old address to the new address. Matching messages do not get routed to the old address.

Here's some example xml configuration for an exclusive divert, it's taken from the divert example:

```
<divert name="prices-divert">
   <address>priceUpdates</address>
   <forwarding-address>priceForwarding</forwarding-address>
   <filter string="office='New York'"/>
   <transformer-class-name>
      org.apache.activemq.artemis.jms.example.AddForwardingTimeTransformer
   </transformer-class-name>
   <exclusive>true</exclusive>
</divert>
```

We define a divert called `prices-divert` that will divert any messages sent to the address `priceUpdates` to another local address `priceForwarding` .

We also specify a message filter string so only messages with the message property `office` with value `New York` will get diverted, all other messages will continue to be routed to the normal address. The filter string is optional, if not specified then all messages will be considered matched.

In this example a transformer class is specified without any configuration properties. Again this is optional, and if specified the transformer will be executed for each matching message. This allows you to change the messages body or properties before it is diverted. In this example the transformer simply adds a header that records the time the divert happened. See the transformer chapter for more details about transformer-specific configuration.

This example is actually diverting messages to a local store and forward queue, which is configured with a bridge which forwards the message to an address on another ActiveMQ Artemis server. Please see the example for more details.

Non-exclusive Divert

Now we'll take a look at a non-exclusive divert. Non exclusive diverts are the same as exclusive diverts, but they only forward a *copy* of the message to the new address. The original message continues to the old address

You can therefore think of non-exclusive diverts as *splitting* a message flow.

Non exclusive diverts can be configured in the same way as exclusive diverts with an optional filter and transformer, here's an example non-exclusive divert, again from the divert example:

```
<divert name="order-divert">
   <address>orders</address>
   <forwarding-address>spyTopic</forwarding-address>
   <exclusive>false</exclusive>
</divert>
```

The above divert example takes a copy of every message sent to the address ' `orders` ' and sends it to a local address called ' `spyTopic` '.

Core Bridges

The function of a bridge is to consume messages from a source queue, and forward them to a target address, typically on a different Apache ActiveMQ Artemis server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, or internet and where the connection may be unreliable.

The bridge has built in resilience to failure so if the target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

In summary, bridges are a way to reliably connect two separate Apache ActiveMQ Artemis servers together. With a core bridge both source and target servers must be Apache ActiveMQ Artemis servers.

Bridges can be configured to provide *once and only once* delivery guarantees even in the event of the failure of the source or the target server. They do this by using duplicate detection (described in Duplicate Detection).

> **Note:**
>
> Although they have similar function, don't confuse core bridges with JMS bridges!
>
> Core bridges are for linking an Apache ActiveMQ Artemis node with another Apache ActiveMQ Artemis node and do not use the JMS API. A JMS Bridge is used for linking any two JMS 1.1 compliant JMS providers. So, a JMS Bridge could be used for bridging to or from different JMS compliant messaging system. It's always preferable to use a core bridge if you can. Core bridges use duplicate detection to provide *once and only once* guarantees. To provide the same guarantee using a JMS bridge you would have to use XA which has a higher overhead and is more complex to configure.

Configuring Bridges

Bridges are configured in `broker.xml`. Let's kick off with an example (this is actually from the bridge example):

```
<bridge name="my-bridge">
   <queue-name>sausage-factory</queue-name>
   <forwarding-address>mincing-machine</forwarding-address>
   <filter string="name='aardvark'"/>
   <transformer-class-name>
      org.apache.activemq.artemis.jms.example.HatColourChangeTransformer
   </transformer-class-name>
   <retry-interval>1000</retry-interval>
   <ha>true</ha>
   <retry-interval-multiplier>1.0</retry-interval-multiplier>
   <initial-connect-attempts>-1</initial-connect-attempts>
   <reconnect-attempts>-1</reconnect-attempts>
   <failover-on-server-shutdown>false</failover-on-server-shutdown>
   <use-duplicate-detection>true</use-duplicate-detection>
   <confirmation-window-size>10000000</confirmation-window-size>
   <user>foouser</user>
   <password>foopassword</password>
   <routing-type>PASS</routing-type>
   <static-connectors>
      <connector-ref>remote-connector</connector-ref>
   </static-connectors>
   <!-- alternative to static-connectors
   <discovery-group-ref discovery-group-name="bridge-discovery-group"/>
   -->
</bridge>
```

In the above example we have shown all the parameters its possible to configure for a bridge. In practice you might use many of the defaults so it won't be necessary to specify them all explicitly.

Let's take a look at all the parameters in turn:

- `name` attribute. All bridges must have a unique name in the server.
- `queue-name`. This is the unique name of the local queue that the bridge consumes from, it's a mandatory parameter.

  The queue must already exist by the time the bridge is instantiated at start-up.
- `forwarding-address`. This is the address on the target server that the message will be forwarded to. If a forwarding address is not specified, then the original address of the message will be retained.
- `filter-string`. An optional filter string can be supplied. If specified then only messages which match the filter expression specified in the filter string will be forwarded. The filter string follows the ActiveMQ Artemis filter expression syntax described in Filter Expressions.
- `transformer-class-name`. An *optional* transformer can be specified. This gives you the opportunity to transform the message's header or body before forwarding it. See the transformer chapter for more details about transformer-specific configuration.
- `ha`. This optional parameter determines whether or not this bridge should support high availability. True means it will connect to any available server in a cluster and support failover. The default value is `false`.

- `retry-interval`. This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `2000` milliseconds.
- `retry-interval-multiplier`. This optional parameter determines determines a multiplier to apply to the time since the last retry to compute the time to the next retry.

  This allows you to implement an *exponential backoff* between retry attempts.

  Let's take an example:

  If we set `retry-interval` to `1000` ms and we set `retry-interval-multiplier` to `2.0`, then, if the first reconnect attempt fails, we will wait `1000` ms then `2000` ms then `4000` ms between subsequent reconnection attempts.

  The default value is `1.0` meaning each reconnect attempt is spaced at equal intervals.
- `initial-connect-attempts`. This optional parameter determines the total number of initial connect attempts the bridge will make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `-1`.
- `reconnect-attempts`. This optional parameter determines the total number of reconnect attempts the bridge will make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `-1`.
- `failover-on-server-shutdown`. This optional parameter determines whether the bridge will attempt to failover onto a backup server (if specified) when the target server is cleanly shutdown rather than crashed.

  The bridge connector can specify both a live and a backup server, if it specifies a backup server and this parameter is set to `true` then if the target server is *cleanly* shutdown the bridge connection will attempt to failover onto its backup. If the bridge connector has no backup server configured then this parameter has no effect.

  Sometimes you want a bridge configured with a live and a backup target server, but you don't want to failover to the backup if the live server is simply taken down temporarily for maintenance, this is when this parameter comes in handy.

  The default value for this parameter is `false`.
- `use-duplicate-detection`. This optional parameter determines whether the bridge will automatically insert a duplicate id property into each message that it forwards.

  Doing so, allows the target server to perform duplicate detection on messages it receives from the source server. If the connection fails or server crashes, then, when the bridge resumes it will resend unacknowledged

messages. This might result in duplicate messages being sent to the target server. By enabling duplicate detection allows these duplicates to be screened out and ignored.

This allows the bridge to provide a *once and only once* delivery guarantee without using heavyweight methods such as XA (see Duplicate Detection for more information).

The default value for this parameter is `true`.

- `confirmation-window-size`. This optional parameter determines the `confirmation-window-size` to use for the connection used to forward messages to the target node. This attribute is described in section Reconnection and Session Reattachment

  > **Warning**
  >
  > When using the bridge to forward messages to an address which uses the `BLOCK` `address-full-policy` from a queue which has a `max-size-bytes` set it's important that `confirmation-window-size` is less than or equal to `max-size-bytes` to prevent the flow of messages from ceasing.

- `producer-window-size`. This optional parameter determines the producer flow control through the bridge. You usually leave this off unless you are dealing with huge large messages.

  Default=-1 (disabled)

- `user`. This optional parameter determines the user name to use when creating the bridge connection to the remote server. If it is not specified the default cluster user specified by `cluster-user` in `broker.xml` will be used.

- `password`. This optional parameter determines the password to use when creating the bridge connection to the remote server. If it is not specified the default cluster password specified by `cluster-password` in `broker.xml` will be used.

- `routing-type`. Bridges can apply a particular routing-type to the messages it forwards, strip the existing routing type, or simply pass the existing routing-type through. This is useful in situations where the message may have its routing-type set but you want to bridge it to an address using a different routing-type. It's important to keep in mind that a message with the `anycast` routing-type will not actually be routed to queues using `multicast` and vice-versa. By configuring the `routing-type` of the bridge you have the flexibility to deal with any situation. Valid values are `ANYCAST`, `MULTICAST`, `PASS`, & `STRIP`. The default is `PASS`.

- `static-connectors` or `discovery-group-ref`. Pick either of these options to connect the bridge to the target server.

  The `static-connectors` is a list of `connector-ref` elements pointing to `connector` elements defined elsewhere. A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the

server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Configuring the Transport.

The `discovery-group-ref` element has one attribute - `discovery-group-name`. This attribute points to a `discovery-group` defined elsewhere. For more information about what discovery-groups are and how to configure them, please see Discovery Groups.

Transformers

A transfomer, as the name suggests, is a component which transforms a message. For example, a transformer could modify the body of a message or add or remove properties. Both diverts and core bridges support.

A transformer is simply a class which implements the interface `org.apache.activemq.artemis.core.server.transformer.Transformer`:

```
public interface Transformer {

   default void init(Map<String, String> properties) { }

   Message transform(Message message);
}
```

The `init` method is called immediately after the broker instantiates the class. There is a default method implementation so implementing `init` is optional. However, if the transformer needs any configuration properties it should implement `init` and the broker will pass the configured key/value pairs to the transformer using a `java.util.Map`.

Configuration

The most basic configuration requires only specifying the transformer's class name, e.g.:

```
<transformer-class-name>
   org.foo.MyTransformer
</transformer-class-name>
```

However, if the transformer needs any configuration properties those can be specified using a slightly different syntax, e.g.:

```
<transformer>
   <class-name>org.foo.MyTransformerWithProperties</class-name>
   <property key="transformerKey1" value="transformerValue1"/>
   <property key="transformerKey2" value="transformerValue2"/>
</transformer>
```

Any transformer implementation needs to be added to the broker's classpath. See the documentation on adding runtime dependencies to understand how to make your transformer available to the broker.

Duplicate Message Detection

Apache ActiveMQ Artemis includes powerful automatic duplicate message detection, filtering out duplicate messages without you having to code your own fiddly duplicate detection logic at the application level. This chapter will explain what duplicate detection is, how Apache ActiveMQ Artemis uses it and how and where to configure it.

When sending messages from a client to a server, or indeed from a server to another server, if the target server or connection fails sometime after sending the message, but before the sender receives a response that the send (or commit) was processed successfully then the sender cannot know for sure if the message was sent successfully to the address.

If the target server or connection failed after the send was received and processed but before the response was sent back then the message will have been sent to the address successfully, but if the target server or connection failed before the send was received and finished processing then it will not have been sent to the address successfully. From the senders point of view it's not possible to distinguish these two cases.

When the server recovers this leaves the client in a difficult situation. It knows the target server failed, but it does not know if the last message reached its destination ok. If it decides to resend the last message, then that could result in a duplicate message being sent to the address. If each message was an order or a trade then this could result in the order being fulfilled twice or the trade being double booked. This is clearly not a desirable situation.

Sending the message(s) in a transaction does not help out either. If the server or connection fails while the transaction commit is being processed it is also indeterminate whether the transaction was successfully committed or not!

To solve these issues Apache ActiveMQ Artemis provides automatic duplicate messages detection for messages sent to addresses.

Using Duplicate Detection for Message Sending

Enabling duplicate message detection for sent messages is simple: you just need to set a special property on the message to a unique value. You can create the value however you like, as long as it is unique. When the target server receives the message it will check if that property is set, if it is, then it will check in its in memory cache if it has already received a message with that value of the header. If it has received a message with the same value before then it will ignore the message.

> **Note:**
>
> Using duplicate detection to move messages between nodes can give you the same *once and only once* delivery guarantees as if you were using an XA transaction to consume messages from source and send them to the target, but with less overhead and much easier configuration than using XA.

If you're sending messages in a transaction then you don't have to set the property for *every* message you send in that transaction, you only need to set it once in the transaction. If the server detects a duplicate message for any message in the transaction, then it will ignore the entire transaction.

The name of the property that you set is given by the value of `org.apache.activemq.artemis.api.core.Message.HDR_DUPLICATE_DETECTION_ID`, which is `_AMQ_DUPL_ID`

The value of the property can be of type `byte[]` or `SimpleString` if you're using the core API. If you're using JMS it must be a `String`, and its value should be unique. An easy way of generating a unique id is by generating a UUID.

Here's an example of setting the property using the core API:

```
...

ClientMessage message = session.createMessage(true);

SimpleString myUniqueID = "This is my unique id";   // Could use a UUID for th

message.setStringProperty(HDR_DUPLICATE_DETECTION_ID, myUniqueID);
```

And here's an example using the JMS API:

```
...

Message jmsMessage = session.createMessage();

String myUniqueID = "This is my unique id";   // Could use a UUID for this

message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);

...
```

Configuring the Duplicate ID Cache

The server maintains caches of received values of the `org.apache.activemq.artemis.core.message.impl.HDR_DUPLICATE_DETECTION_ID` property sent to each address. Each address has its own distinct cache.

The cache is a circular fixed size cache. If the cache has a maximum size of `n` elements, then the `n + 1` th id stored will overwrite the `0` th element in the cache.

The maximum size of the cache is configured by the parameter `id-cache-size` in `broker.xml`, the default value is `20000` elements.

The caches can also be configured to persist to disk or not. This is configured by the parameter `persist-id-cache`, also in `broker.xml`. If this is set to `true` then each id will be persisted to permanent storage as they are received. The default value for this parameter is `true`.

> **Note:**
>
> When choosing a size of the duplicate id cache be sure to set it to a larger enough size so if you resend messages all the previously sent ones are in the cache not having been overwritten.

Duplicate Detection and Bridges

Core bridges can be configured to automatically add a unique duplicate id value (if there isn't already one in the message) before forwarding the message to its target. This ensures that if the target server crashes or the connection is interrupted and the bridge resends the message, then if it has already been received by the target server, it will be ignored.

To configure a core bridge to add the duplicate id header, simply set the `use-duplicate-detection` to `true` when configuring a bridge in `broker.xml`.

The default value for this parameter is `true`.

For more information on core bridges and how to configure them, please see Core Bridges.

Duplicate Detection and Cluster Connections

Cluster connections internally use core bridges to move messages reliable between nodes of the cluster. Consequently they can also be configured to insert the duplicate id header for each message they move using their internal bridges.

To configure a cluster connection to add the duplicate id header, simply set the `use-duplicate-detection` to `true` when configuring a cluster connection in `broker.xml`.

The default value for this parameter is `true`.

For more information on cluster connections and how to configure them, please see Clusters.

Clusters

Overview

Apache ActiveMQ Artemis clusters allow groups of Apache ActiveMQ Artemis servers to be grouped together in order to share message processing load. Each active node in the cluster is an active Apache ActiveMQ Artemis server which manages its own messages and handles its own connections.

The cluster is formed by each node declaring *cluster connections* to other nodes in the core configuration file `broker.xml`. When a node forms a cluster connection to another node, internally it creates a *core bridge* (as described in Core Bridges) connection between it and the other node, this is done transparently behind the scenes - you don't have to declare an explicit bridge for each node. These cluster connections allow messages to flow between the nodes of the cluster to balance load.

Nodes can be connected together to form a cluster in many different topologies, we will discuss a couple of the more common topologies later in this chapter.

We'll also discuss client side load balancing, where we can balance client connections across the nodes of the cluster, and we'll consider message redistribution where Apache ActiveMQ Artemis will redistribute messages between nodes to avoid starvation.

Another important part of clustering is *server discovery* where servers can broadcast their connection details so clients or other servers can connect to them with the minimum of configuration.

> **Warning**
>
> Once a cluster node has been configured it is common to simply copy that configuration to other nodes to produce a symmetric cluster. However, care must be taken when copying the Apache ActiveMQ Artemis files. Do not copy the Apache ActiveMQ Artemis *data* (i.e. the `bindings`, `journal`, and `large-messages` directories) from one node to another. When a node is started for the first time and initializes its journal files it also persists a special identifier to the `journal` directory. This id *must* be unique among nodes in the cluster or the cluster will not form properly.

Server discovery

Server discovery is a mechanism by which servers can propagate their connection details to:

- Messaging clients. A messaging client wants to be able to connect to the servers of the cluster without having specific knowledge of which servers in the cluster are up at any one time.

- Other servers. Servers in a cluster want to be able to create cluster connections to each other without having prior knowledge of all the other servers in the cluster.

This information, let's call it the Cluster Topology, is actually sent around normal Apache ActiveMQ Artemis connections to clients and to other servers over cluster connections. This being the case we need a way of establishing the initial first connection. This can be done using dynamic discovery techniques like UDP and JGroups, or by providing a list of initial connectors.

Dynamic Discovery

Server discovery uses UDP multicast or JGroups to broadcast server connection settings.

Broadcast Groups

A broadcast group is the means by which a server broadcasts connectors over the network. A connector defines a way in which a client (or other server) can make connections to the server. For more information on what a connector is, please see Configuring the Transport.

The broadcast group takes a set of connector pairs, each connector pair contains connection settings for a live and backup server (if one exists) and broadcasts them on the network. Depending on which broadcasting technique you configure the cluster, it uses either UDP or JGroups to broadcast connector pairs information.

Broadcast groups are defined in the server configuration file `broker.xml`. There can be many broadcast groups per Apache ActiveMQ Artemis server. All broadcast groups must be defined in a `broadcast-groups` element.

Let's take a look at an example broadcast group from `broker.xml` that defines a UDP broadcast group:

```
<broadcast-groups>
   <broadcast-group name="my-broadcast-group">
    <local-bind-address>172.16.9.3</local-bind-address>
    <local-bind-port>5432</local-bind-port>
    <group-address>231.7.7.7</group-address>
    <group-port>9876</group-port>
    <broadcast-period>2000</broadcast-period>
    <connector-ref>netty-connector</connector-ref>
   </broadcast-group>
</broadcast-groups>
```

Some of the broadcast group parameters are optional and you'll normally use the defaults, but we specify them all in the above example for clarity. Let's discuss each one in turn:

- `name` attribute. Each broadcast group in the server must have a unique name.

- `local-bind-address`. This is the local bind address that the datagram socket is bound to. If you have multiple network interfaces on your server, you would specify which one you wish to use for broadcasts by setting this property. If this property is not specified then the socket will be bound to the wildcard address, an IP address chosen by the kernel. This is a UDP specific attribute.
- `local-bind-port`. If you want to specify a local port to which the datagram socket is bound you can specify it here. Normally you would just use the default value of `-1` which signifies that an anonymous port should be used. This parameter is always specified in conjunction with `local-bind-address`. This is a UDP specific attribute.
- `group-address`. This is the multicast address to which the data will be broadcast. It is a class D IP address in the range `224.0.0.0` to `239.255.255.255`, inclusive. The address `224.0.0.0` is reserved and is not available for use. This parameter is mandatory. This is a UDP specific attribute.
- `group-port`. This is the UDP port number used for broadcasting. This parameter is mandatory. This is a UDP specific attribute.
- `broadcast-period`. This is the period in milliseconds between consecutive broadcasts. This parameter is optional, the default value is `2000` milliseconds.
- `connector-ref`. This specifies the connector and optional backup connector that will be broadcasted (see Configuring the Transport for more information on connectors).

Here is another example broadcast group that defines a JGroups broadcast group:

```
<broadcast-groups>
   <broadcast-group name="my-broadcast-group">
      <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>
      <jgroups-channel>activemq_broadcast_channel</jgroups-channel>
      <broadcast-period>2000</broadcast-period>
      <connector-ref>netty-connector</connector-ref>
   </broadcast-group>
</broadcast-groups>
```

To be able to use JGroups to broadcast, one must specify two attributes, i.e. `jgroups-file` and `jgroups-channel`, as discussed in details as following:

- `jgroups-file` attribute. This is the name of JGroups configuration file. It will be used to initialize JGroups channels. Make sure the file is in the java resource path so that Apache ActiveMQ Artemis can load it.
- `jgroups-channel` attribute. The name that JGroups channels connect to for broadcasting.

> **Note:**
>
> The JGroups attributes (`jgroups-file` and `jgroups-channel`) and UDP specific attributes described above are exclusive of each other. Only one set can be specified in a broadcast group configuration. Don't mix them!

The following is an example of a JGroups file

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/JGro
   <TCP loopback="true"
      recv_buf_size="20000000"
      send_buf_size="640000"
      discard_incompatible_packets="true"
      max_bundle_size="64000"
      max_bundle_timeout="30"
      enable_bundling="true"
      use_send_queues="false"
      sock_conn_timeout="300"

      thread_pool.enabled="true"
      thread_pool.min_threads="1"
      thread_pool.max_threads="10"
      thread_pool.keep_alive_time="5000"
      thread_pool.queue_enabled="false"
      thread_pool.queue_max_size="100"
      thread_pool.rejection_policy="run"

      oob_thread_pool.enabled="true"
      oob_thread_pool.min_threads="1"
      oob_thread_pool.max_threads="8"
      oob_thread_pool.keep_alive_time="5000"
      oob_thread_pool.queue_enabled="false"
      oob_thread_pool.queue_max_size="100"
      oob_thread_pool.rejection_policy="run"/>

   <FILE_PING location="../file.ping.dir"/>
   <MERGE2 max_interval="30000"
      min_interval="10000"/>
   <FD_SOCK/>
   <FD timeout="10000" max_tries="5" />
   <VERIFY_SUSPECT timeout="1500"  />
   <BARRIER />
   <pbcast.NAKACK
      use_mcast_xmit="false"
      retransmit_timeout="300,600,1200,2400,4800"
      discard_delivered_msgs="true"/>
   <UNICAST timeout="300,600,1200" />
   <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
      max_bytes="400000"/>
   <pbcast.GMS print_local_addr="true" join_timeout="3000"
      view_bundling="true"/>
   <FC max_credits="2000000"
      min_threshold="0.10"/>

   <FRAG2 frag_size="60000"  />
   <pbcast.STATE_TRANSFER/>
   <pbcast.FLUSH timeout="0"/>
</config>
```

As it shows, the file content defines a jgroups protocol stacks. If you want Apache ActiveMQ Artemis to use this stacks for channel creation, you have to make sure the value of `jgroups-file` in your broadcast-group/discovery-group configuration to be the name of this jgroups configuration file. For example if the above stacks configuration is stored in a file named "jgroups-stacks.xml" then your `jgroups-file` should be like

```
<jgroups-file>jgroups-stacks.xml</jgroups-file>
```

Discovery Groups

While the broadcast group defines how connector information is broadcasted from a server, a discovery group defines how connector information is received from a broadcast endpoint (a UDP multicast address or JGroup channel).

A discovery group maintains a list of connector pairs - one for each broadcast by a different server. As it receives broadcasts on the broadcast endpoint from a particular server it updates its entry in the list for that server.

If it has not received a broadcast from a particular server for a length of time it will remove that server's entry from its list.

Discovery groups are used in two places in Apache ActiveMQ Artemis:

- By cluster connections so they know how to obtain an initial connection to download the topology
- By messaging clients so they know how to obtain an initial connection to download the topology

Although a discovery group will always accept broadcasts, its current list of available live and backup servers is only ever used when an initial connection is made, from then server discovery is done over the normal Apache ActiveMQ Artemis connections.

> **Note:**
>
> Each discovery group must be configured with broadcast endpoint (UDP or JGroups) that matches its broadcast group counterpart. For example, if broadcast is configured using UDP, the discovery group must also use UDP, and the same multicast address.

Defining Discovery Groups on the Server

For cluster connections, discovery groups are defined in the server side configuration file `broker.xml`. All discovery groups must be defined inside a `discovery-groups` element. There can be many discovery groups defined by Apache ActiveMQ Artemis server. Let's look at an example:

```
<discovery-groups>
   <discovery-group name="my-discovery-group">
      <local-bind-address>172.16.9.7</local-bind-address>
      <group-address>231.7.7.7</group-address>
      <group-port>9876</group-port>
      <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>
```

We'll consider each parameter of the discovery group:

- `name` attribute. Each discovery group must have a unique name per server.
- `local-bind-address`. If you are running with multiple network interfaces on the same machine, you may want to specify that the discovery group listens only only a specific interface. To do this you can specify the interface address with this parameter. This parameter is optional. This is a UDP specific attribute.
- `group-address`. This is the multicast IP address of the group to listen on. It should match the `group-address` in the broadcast group that you wish to listen from. This parameter is mandatory. This is a UDP specific attribute.
- `group-port`. This is the UDP port of the multicast group. It should match the `group-port` in the broadcast group that you wish to listen from. This parameter is mandatory. This is a UDP specific attribute.
- `refresh-timeout`. This is the period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. You would normally set this to a value significantly higher than the `broadcast-period` on the broadcast group otherwise servers might intermittently disappear from the list even though they are still broadcasting due to slight differences in timing. This parameter is optional, the default value is `10000` milliseconds (10 seconds).

Here is another example that defines a JGroups discovery group:

```
<discovery-groups>
   <discovery-group name="my-broadcast-group">
      <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>
      <jgroups-channel>activemq_broadcast_channel</jgroups-channel>
      <refresh-timeout>10000</refresh-timeout>
   </discovery-group>
</discovery-groups>
```

To receive broadcast from JGroups channels, one must specify two attributes, `jgroups-file` and `jgroups-channel`, as discussed in details as following:

- `jgroups-file` attribute. This is the name of JGroups configuration file. It will be used to initialize JGroups channels. Make sure the file is in the java resource path so that Apache ActiveMQ Artemis can load it.
- `jgroups-channel` attribute. The name that JGroups channels connect to for receiving broadcasts.

> **Note:**
>
> The JGroups attributes (`jgroups-file` and `jgroups-channel`) and UDP specific attributes described above are exclusive of each other. Only one set can be specified in a discovery group configuration. Don't mix them!

Discovery Groups on the Client Side

Let's discuss how to configure an Apache ActiveMQ Artemis client to use discovery to discover a list of servers to which it can connect. The way to do this differs depending on whether you're using JMS or the core API.

Configuring client discovery

Use the `udp` URL scheme and a host:port combination matches the group-address and group-port from the corresponding `broadcast-group` on the server:

```
udp://231.7.7.7:9876
```

The element `discovery-group-ref` specifies the name of a discovery group defined in `broker.xml` .

Connections created using this URI will be load-balanced across the list of servers that the discovery group maintains by listening on the multicast address specified in the discovery group configuration.

The aforementioned `refreshTimeout` parameter can be set directly in the URI.

There is also a URL parameter named `initialWaitTimeout` . If the corresponding JMS connection factory or core session factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default value for this parameter is `10000` milliseconds.

Discovery using static Connectors

Sometimes it may be impossible to use UDP on the network you are using. In this case its possible to configure a connection with an initial list of possible servers. This could be just one server that you know will always be available or a list of servers where at least one will be available.

This doesn't mean that you have to know where all your servers are going to be hosted, you can configure these servers to use the reliable servers to connect to. Once they are connected their connection details will be propagated via the server it connects to

Configuring a Cluster Connection

For cluster connections there is no extra configuration needed, you just need to make sure that any connectors are defined in the usual manner, (see Configuring the Transport for more information on connectors). These are then referenced by

the cluster connection configuration.

Configuring a Client Connection

A static list of possible servers can also be used by a normal client.

Configuring client discovery

A list of servers to be used for the initial connection attempt can be specified in the connection URI using a syntax with `()`, e.g.:

```
(tcp://myhost:61616,tcp://myhost2:61616)?reconnectAttempts=5
```

The brackets are expanded so the same query can be appended after the last bracket for ease.

Server-Side Message Load Balancing

If cluster connections are defined between nodes of a cluster, then Apache ActiveMQ Artemis will load balance messages arriving at a particular node from a client.

Let's take a simple example of a cluster of four nodes A, B, C, and D arranged in a *symmetric cluster* (described in Symmetrical Clusters section). We have a queue called `OrderQueue` deployed on each node of the cluster.

We have client Ca connected to node A, sending orders to the server. We have also have order processor clients Pa, Pb, Pc, and Pd connected to each of the nodes A, B, C, D. If no cluster connection was defined on node A, then as order messages arrive on node A they will all end up in the `OrderQueue` on node A, so will only get consumed by the order processor client attached to node A, Pa.

If we define a cluster connection on node A, then as ordered messages arrive on node A instead of all of them going into the local `OrderQueue` instance, they are distributed in a round-robin fashion between all the nodes of the cluster. The messages are forwarded from the receiving node to other nodes of the cluster. This is all done on the server side, the client maintains a single connection to node A.

For example, messages arriving on node A might be distributed in the following order between the nodes: B, D, C, A, B, D, C, A, B, D. The exact order depends on the order the nodes started up, but the algorithm used is round robin.

Apache ActiveMQ Artemis cluster connections can be configured to always blindly load balance messages in a round robin fashion irrespective of whether there are any matching consumers on other nodes, but they can be a bit cleverer than that and also be configured to only distribute to other nodes if they have matching consumers. We'll look at both these cases in turn with some examples, but first we'll discuss configuring cluster connections in general.

Configuring Cluster Connections

Cluster connections group servers into clusters so that messages can be load balanced between the nodes of the cluster. Let's take a look at a typical cluster connection. Cluster connections are always defined in `broker.xml` inside a `cluster-connection` element. There can be zero or more cluster connections defined per Apache ActiveMQ Artemis server.

```
<cluster-connections>
   <cluster-connection name="my-cluster">
      <address></address>
      <connector-ref>netty-connector</connector-ref>
      <check-period>1000</check-period>
      <connection-ttl>5000</connection-ttl>
      <min-large-message-size>50000</min-large-message-size>
      <call-timeout>5000</call-timeout>
      <retry-interval>500</retry-interval>
      <retry-interval-multiplier>1.0</retry-interval-multiplier>
      <max-retry-interval>5000</max-retry-interval>
      <initial-connect-attempts>-1</initial-connect-attempts>
      <reconnect-attempts>-1</reconnect-attempts>
      <use-duplicate-detection>true</use-duplicate-detection>
      <message-load-balancing>ON_DEMAND</message-load-balancing>
      <max-hops>1</max-hops>
      <confirmation-window-size>32000</confirmation-window-size>
      <call-failover-timeout>30000</call-failover-timeout>
      <notification-interval>1000</notification-interval>
      <notification-attempts>2</notification-attempts>
      <discovery-group-ref discovery-group-name="my-discovery-group"/>
   </cluster-connection>
</cluster-connections>
```

In the above cluster connection all parameters have been explicitly specified. The following shows all the available configuration options

- `address` Each cluster connection only applies to addresses that match the specified `address` field. An address is matched on the cluster connection when it begins with the string specified in this field. The `address` field on a cluster connection also supports comma separated lists and an exclude syntax `!`. To prevent an address from being matched on this cluster connection, prepend a cluster connection address string with `!`.

  In the case shown above the cluster connection will load balance messages sent to all addresses (since it's empty).

  The address can be any value and you can have many cluster connections with different values of `address`, simultaneously balancing messages for those addresses, potentially to different clusters of servers. By having multiple cluster connections on different addresses a single Apache ActiveMQ Artemis Server can effectively take part in multiple clusters simultaneously.

  Be careful not to have multiple cluster connections with overlapping values of `address`, e.g. "europe" and "europe.news" since this could result in the same messages being distributed between more than one cluster connection, possibly resulting in duplicate deliveries.

  Examples:

  - 'eu' matches all addresses starting with 'eu'
  - '!eu' matches all address except for those starting with 'eu'
  - 'eu.uk,eu.de' matches all addresses starting with either 'eu.uk' or 'eu.de'
  - 'eu,!eu.uk' matches all addresses starting with 'eu' but not those starting with 'eu.uk'

  Note::

  - Address exclusion will always takes precedence over address inclusion.
  - Address matching on cluster connections does not support wild-card matching.

- `connector-ref`. This is the connector which will be sent to other nodes in the cluster so they have the correct cluster topology.

  This parameter is mandatory.

- `check-period`. The period (in milliseconds) used to check if the cluster connection has failed to receive pings from another server. Default is 30000.

- `connection-ttl`. This is how long a cluster connection should stay alive if it stops receiving messages from a specific node in the cluster. Default is 60000.

- `min-large-message-size`. If the message size (in bytes) is larger than this value then it will be split into multiple segments when sent over the network to other cluster members. Default is 102400.

- `call-timeout`. When a packet is sent via a cluster connection and is a blocking call, i.e. for acknowledgements, this is how long it will wait (in milliseconds) for the reply before throwing an exception. Default is 30000.

- `retry-interval`. We mentioned before that, internally, cluster connections cause bridges to be created between the nodes of the cluster. If the cluster connection is created and the target node has not been started, or say, is being rebooted, then the cluster connections from other nodes will retry connecting to the target until it comes back up, in the same way as a bridge does.

  This parameter determines the interval in milliseconds between retry attempts. It has the same meaning as the `retry-interval` on a bridge (as described in Core Bridges).

  This parameter is optional and its default value is `500` milliseconds.

- `retry-interval-multiplier`. This is a multiplier used to increase the `retry-interval` after each reconnect attempt, default is 1.

- `max-retry-interval`. The maximum delay (in milliseconds) for retries. Default is 2000.

- `initial-connect-attempts`. The number of times the system will try to connect a node in the cluster initially. If the max-retry is achieved this node will be considered permanently down and the system will not route messages to this node. Default is -1 (infinite retries).

- `reconnect-attempts`. The number of times the system will try to reconnect to a node in the cluster. If the max-retry is achieved this node will be considered permanently down and the system will stop routing messages to this node. Default is -1 (infinite retries).

- `use-duplicate-detection`. Internally cluster connections use bridges to link the nodes, and bridges can be configured to add a duplicate id property in each message that is forwarded. If the target node of the bridge crashes and then recovers, messages might be resent from the source node. By enabling duplicate detection any duplicate messages will be filtered out and ignored on receipt at the target node.

  This parameter has the same meaning as `use-duplicate-detection` on a bridge. For more information on duplicate detection, please see Duplicate Detection. Default is true.

- `message-load-balancing`. This parameter determines if/how messages will be distributed between other nodes of the cluster. It can be one of three values - `OFF`, `STRICT`, or `ON_DEMAND` (default). This parameter replaces the deprecated `forward-when-no-consumers` parameter.

  If this is set to `OFF` then messages will never be forwarded to another node in the cluster

  If this is set to `STRICT` then each incoming message will be round robin'd even though the same queues on the other nodes of the cluster may have no consumers at all, or they may have consumers that have non matching message filters (selectors). Note that Apache ActiveMQ Artemis will *not* forward messages to other nodes if there are no *queues* of the same name on the other nodes, even if this parameter is set to `STRICT`. Using `STRICT` is like setting the legacy `forward-when-no-consumers` parameter to `true`.

  If this is set to `ON_DEMAND` then Apache ActiveMQ Artemis will only forward messages to other nodes of the cluster if the address to which they are being forwarded has queues which have consumers, and if those consumers have message filters (selectors) at least one of those selectors must match the message. Using `ON_DEMAND` is like setting the legacy `forward-when-no-consumers` parameter to `false`.

  Keep in mind that this message forwarding/balancing is what we call "initial distribution." It is different than *redistribution* which is discussed below. This distinction is important because redistribution is configured differently and has unique semantics (e.g. it *does not* support filters (selectors)).

  Default is `ON_DEMAND`.

- `max-hops`. When a cluster connection decides the set of nodes to which it might load balance a message, those nodes do not have to be directly connected to it via a cluster connection. Apache ActiveMQ Artemis can be configured to also load balance messages to nodes which might be connected to it only indirectly with other Apache ActiveMQ Artemis servers as intermediates in a chain.

This allows Apache ActiveMQ Artemis to be configured in more complex topologies and still provide message load balancing. We'll discuss this more later in this chapter.

The default value for this parameter is `1`, which means messages are only load balanced to other Apache ActiveMQ Artemis serves which are directly connected to this server. This parameter is optional.

- `confirmation-window-size`. The size (in bytes) of the window used for sending confirmations from the server connected to. So once the server has received `confirmation-window-size` bytes it notifies its client, default is 1048576. A value of -1 means no window.
- `producer-window-size`. The size for producer flow control over cluster connection. it's by default disabled through the cluster connection bridge but you may want to set a value if you are using really large messages in cluster. A value of -1 means no window.
- `call-failover-timeout`. Similar to `call-timeout` but used when a call is made during a failover attempt. Default is -1 (no timeout).
- `notification-interval`. How often (in milliseconds) the cluster connection should broadcast itself when attaching to the cluster. Default is 1000.
- `notification-attempts`. How many times the cluster connection should broadcast itself when connecting to the cluster. Default is 2.
- `discovery-group-ref`. This parameter determines which discovery group is used to obtain the list of other servers in the cluster that this cluster connection will make connections to.

Alternatively if you would like your cluster connections to use a static list of servers for discovery then you can do it like this.

```
<cluster-connection name="my-cluster">
   ...
   <static-connectors>
      <connector-ref>server0-connector</connector-ref>
      <connector-ref>server1-connector</connector-ref>
   </static-connectors>
</cluster-connection>
```

Here we have defined 2 servers that we know for sure will that at least one will be available. There may be many more servers in the cluster but these will; be discovered via one of these connectors once an initial connection has been made.

Cluster User Credentials

When creating connections between nodes of a cluster to form a cluster connection, Apache ActiveMQ Artemis uses a cluster user and cluster password which is defined in `broker.xml`:

```
<cluster-user>ACTIVEMQ.CLUSTER.ADMIN.USER</cluster-user>
<cluster-password>CHANGE ME!!</cluster-password>
```

> **Warning**
>
> It is imperative that these values are changed from their default, or remote clients will be able to make connections to the server using the default values. If they are not changed from the default, Apache ActiveMQ Artemis will detect this and pester you with a warning on every start-up.

Client-Side Load balancing

With Apache ActiveMQ Artemis client-side load balancing, subsequent sessions created using a single session factory can be connected to different nodes of the cluster. This allows sessions to spread smoothly across the nodes of a cluster and not be "clumped" on any particular node.

The load balancing policy to be used by the client factory is configurable. Apache ActiveMQ Artemis provides four out-of-the-box load balancing policies, and you can also implement your own and use that.

The out-of-the-box policies are

- Round Robin. With this policy the first node is chosen randomly then each subsequent node is chosen sequentially in the same order.

  For example nodes might be chosen in the order B, C, D, A, B, C, D, A, B or D, A, B, C, D, A, B, C, D or C, D, A, B, C, D, A, B, C.

  Use `org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy` as the `<connection-load-balancing-policy-class-name>`.

- Random. With this policy each node is chosen randomly.

  Use `org.apache.activemq.artemis.api.core.client.loadbalance.RandomConnectionLoadBalancingPolicy` as the `<connection-load-balancing-policy-class-name>`.

- Random Sticky. With this policy the first node is chosen randomly and then re-used for subsequent connections.

  Use `org.apache.activemq.artemis.api.core.client.loadbalance.RandomStickyConnectionLoadBalancingPolicy` as the `<connection-load-balancing-policy-class-name>`.

- First Element. With this policy the "first" (i.e. 0th) node is always returned.

  Use `org.apache.activemq.artemis.api.core.client.loadbalance.FirstElementConnectionLoadBalancingPolicy` as the `<connection-load-balancing-policy-class-name>`.

You can also implement your own policy by implementing the interface `org.apache.activemq.artemis.api.core.client.loadbalance.ConnectionLoadBalancingPolicy`

Specifying which load balancing policy to use differs whether you are using JMS or the core API. If you don't specify a policy then the default will be used which is `org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy`.

The parameter `connectionLoadBalancingPolicyClassName` can be set on the URI to configure what load balancing policy to use:

```
tcp://localhost:61616?connectionLoadBalancingPolicyClassName=org.apache.activer
```

The set of servers over which the factory load balances can be determined in one of two ways:

- Specifying servers explicitly in the URL. This also requires setting the `useTopologyForLoadBalancing` parameter to `false` on the URL.
- Using discovery. This is the default behavior.

Specifying Members of a Cluster Explicitly

Sometimes you want to explicitly define a cluster more explicitly, that is control which server connect to each other in the cluster. This is typically used to form non symmetrical clusters such as chain cluster or ring clusters. This can only be done using a static list of connectors and is configured as follows:

```
<cluster-connection name="my-cluster">
   <address/>
   <connector-ref>netty-connector</connector-ref>
   <retry-interval>500</retry-interval>
   <use-duplicate-detection>true</use-duplicate-detection>
   <message-load-balancing>STRICT</message-load-balancing>
   <max-hops>1</max-hops>
   <static-connectors allow-direct-connections-only="true">
      <connector-ref>server1-connector</connector-ref>
   </static-connectors>
</cluster-connection>
```

In this example we have set the attribute `allow-direct-connections-only` which means that the only server that this server can create a cluster connection to is server1-connector. This means you can explicitly create any cluster topology you want.

Message Redistribution

Another important part of clustering is message redistribution. Earlier we learned how server side message load balancing round robins messages across the cluster. If `message-load-balancing` is `OFF` or `ON_DEMAND` then messages won't be forwarded to nodes which don't have matching consumers. This is great and ensures that messages aren't moved to a queue which has no consumers to consume them. However, there is a situation it doesn't solve: What happens if the

consumers on a queue close after the messages have been sent to the node? If there are no consumers on the queue the message won't get consumed and we have a *starvation* situation.

This is where message redistribution comes in. With message redistribution Apache ActiveMQ Artemis can be configured to automatically *redistribute* messages from queues which have no consumers back to other nodes in the cluster which do have matching consumers. To enable this functionality `message-load-balancing` must be `ON_DEMAND`.

Message redistribution can be configured to kick in immediately after the last consumer on a queue is closed, or to wait a configurable delay after the last consumer on a queue is closed before redistributing. By default message redistribution is disabled.

Message redistribution can be configured on a per address basis, by specifying the redistribution delay in the address settings. For more information on configuring address settings, please see Configuring Addresses and Queues via Address Settings.

Here's an address settings snippet from `broker.xml` showing how message redistribution is enabled for a set of queues:

```
<address-settings>
   <address-setting match="#">
      <redistribution-delay>0</redistribution-delay>
   </address-setting>
</address-settings>
```

The above `address-settings` block would set a `redistribution-delay` of `0` for any queue which is bound to any address. So the above would enable instant (no delay) redistribution for all addresses.

The attribute `match` can be an exact match or it can be a string that conforms to the Apache ActiveMQ Artemis wildcard syntax (described in Wildcard Syntax).

The element `redistribution-delay` defines the delay in milliseconds after the last consumer is closed on a queue before redistributing messages from that queue to other nodes of the cluster which do have matching consumers. A delay of zero means the messages will be immediately redistributed. A value of `-1` signifies that messages will never be redistributed. The default value is `-1`.

It often makes sense to introduce a delay before redistributing as it's a common case that a consumer closes but another one quickly is created on the same queue, in such a case you probably don't want to redistribute immediately since the new consumer will arrive shortly.

Redistribution and filters (selectors)

Although "initial distribution" (described above) does support filters (selectors), redistribution does *not* support filters. Consider this scenario:

1. A cluster of 2 nodes - `A` and `B` - using a `redistribution-delay` of `0` and a `message-load-balancing` of `ON_DEMAND`.
2. `A` and `B` each has the queue `foo`.
3. A producer sends a message which is routed to queue `foo` on node `A`. The message has property named `myProperty` with a value of `10`.
4. A consumer connects to queue `foo` on node `A` with the filter `myProperty=5`. This filter doesn't match the message.
5. A consumer connects to queue `foo` on node `B` with the filter `myProperty=10`. This filter *does* match the message .

Despite the fact that the filter of the consumer on queue `foo` on node `B` matches the message, the message will *not* be redistributed from node `A` to node `B` because a consumer for the queue exists on node `A`.

Not supporting redistribution based on filters was an explicit design decision in order to avoid two main problems - queue scanning and unnecessary redistribution.

From a performance perspective a consumer with a filter on a queue is already costly due to the scanning that the broker must do on the queue to find matching messages. In general, this is a bit of an anti-pattern as it turns the broker into something akin to a database where you can "select" the data you want using a filter. If brokers are configured in a cluster and a consumer with a filter connects and no matches are found after scanning the local queue then potentially every instance of that queue in the cluster would need to be scanned. This turns into a bit of a scalability nightmare with lots of consumers (especially short-lived consumers) with filters connecting & disconnecting frequently. The time & computing resources used for queue scanning would go through the roof.

It is also possible to get into a pathological situation where short-lived consumers with filters connect to nodes around the cluster and messages get redistributed back and forth between nodes without ever actually being consumed.

One common use-case for consumers with filters (selectors) on queues is request/reply using a correlation ID. Following the standard pattern can be problematic in a cluster due to the lack of redistribution based on filters already described. However, there is a simple way to ensure an application using this request/reply pattern gets its reply even when using a correlation ID filter in a cluster - create the consumer before the request is sent. This will ensure that when the reply is sent it will be routed the proper cluster node since "*initial* distribution" (described above) does support filters. For example, in the scenario outlined above if steps 3 and 5 were switched (i.e. if the consumers were created before the message was sent) then the consumer on node `B` would in fact receive the message.

Cluster topologies

Apache ActiveMQ Artemis clusters can be connected together in many different topologies, let's consider the two most common ones here

Symmetric cluster

A symmetric cluster is probably the most common cluster topology.

With a symmetric cluster every node in the cluster is connected to every other node in the cluster. In other words every node in the cluster is no more than one hop away from every other node.

To form a symmetric cluster every node in the cluster defines a cluster connection with the attribute `max-hops` set to `1`. Typically the cluster connection will use server discovery in order to know what other servers in the cluster it should connect to, although it is possible to explicitly define each target server too in the cluster connection if, for example, UDP is not available on your network.

With a symmetric cluster each node knows about all the queues that exist on all the other nodes and what consumers they have. With this knowledge it can determine how to load balance and redistribute messages around the nodes.

Don't forget this warning when creating a symmetric cluster.

Chain cluster

With a chain cluster, each node in the cluster is not connected to every node in the cluster directly, instead the nodes form a chain with a node on each end of the chain and all other nodes just connecting to the previous and next nodes in the chain.

An example of this would be a three node chain consisting of nodes A, B and C. Node A is hosted in one network and has many producer clients connected to it sending order messages. Due to corporate policy, the order consumer clients need to be hosted in a different network, and that network is only accessible via a third network. In this setup node B acts as a mediator with no producers or consumers on it. Any messages arriving on node A will be forwarded to node B, which will in turn forward them to node C where they can get consumed. Node A does not need to directly connect to C, but all the nodes can still act as a part of the cluster.

To set up a cluster in this way, node A would define a cluster connection that connects to node B, and node B would define a cluster connection that connects to node C. In this case we only want cluster connections in one direction since we're only moving messages from node A->B->C and never from C->B->A.

For this topology we would set `max-hops` to `2`. With a value of `2` the knowledge of what queues and consumers that exist on node C would be propagated from node C to node B to node A. Node A would then know to distribute messages to node B when they arrive, even though node B has no consumers itself, it would know that a further hop away is node C which does have consumers.

Scaling Down

Apache ActiveMQ Artemis supports scaling down a cluster with no message loss (even for non-durable messages). This is especially useful in certain environments (e.g. the cloud) where the size of a cluster may change relatively frequently. When scaling up a cluster (i.e. adding nodes) there is no risk of message loss, but when scaling down a cluster (i.e. removing nodes) the messages on those nodes would be lost unless the broker sent them to another node in the cluster. Apache ActiveMQ Artemis can be configured to do just that.

The simplest way to enable this behavior is to set `scale-down` to `true`. If the server is clustered and `scale-down` is `true` then when the server is shutdown gracefully (i.e. stopped without crashing) it will find another node in the cluster and send *all* of its messages (both durable and non-durable) to that node. The messages are processed in order and go to the *back* of the respective queues on the other node (just as if the messages were sent from an external client for the first time).

If more control over where the messages go is required then specify `scale-down-group-name`. Messages will only be sent to another node in the cluster that uses the same `scale-down-group-name` as the server being shutdown.

> **Warning**
>
> If cluster nodes are grouped together with different `scale-down-group-name` values beware. If all the nodes in a single group are shut down then the messages from that node/group will be lost.

If the server is using multiple `cluster-connection` then use `scale-down-clustername` to identify the name of the `cluster-connection` which should be used for scaling down.

Federation

Introduction

Federation allows transmission of messages between brokers without requiring clustering.

A federated address can replicate messages published from an upstream address to a local address. n.b. This is only supported with multicast addresses.

A federated queue lets a local consumer receive messages from an upstream queue.

A broker can contain federated and local-only components - you don't need to federate everything if you don't want to.

Benefits

WAN

The source and target servers do not have to be in the same cluster which makes federation suitable for reliably sending messages from one cluster to another, for instance across a WAN, between cloud regions or there internet and where the connection may be unreliable.

Federation has built in resilience to failure so if the target server connection is lost, e.g. due to network failure, federation will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

Loose Coupling of Brokers

Federation can transmit messages between brokers (or clusters) in different administrative domains:

- they may have different configuration, users and setup;
- they may run on different versions of ActiveMQ Artemis

Dynamic and Selective

Federation is applied by policies, that match address and queue names, and then apply.

This means that federation can dynamically be applied as queues or addresses are added and removed, without needing to hard configure each and every one.

Like wise policies are selective, in that they apply with multiple include and exclude matches.

Mutliple policies can applied directly to multiple upstreams, as well policies can be grouped into policy sets and then applied to upstreams to make managing easier.

Address Federation

Address federation is like full multicast over the connected brokers, in that every message sent to address on `Broker-A` will be delivered to every queue on that broker, but like wise will be delivered to `Broker-B` and all attached queues there.

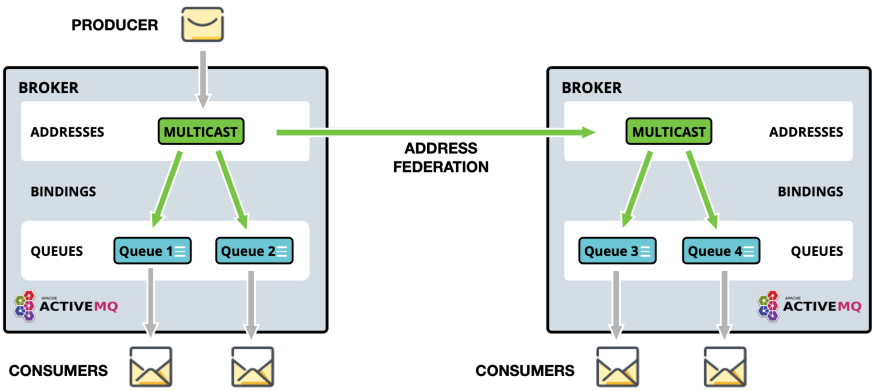


Figure 1. Address Federation

For further details please goto Address Federation.

Queue Federation

Effectively, all federated queues act as a single logical queue, with multiple receivers on multiple machines. So federated queues can be used for load balancing. Typically if the brokers are in the same AZ you would look to cluster them, the advantage of queue federation is that it does not require clustering so is suitable for over WAN, cross-region, on-off prem.

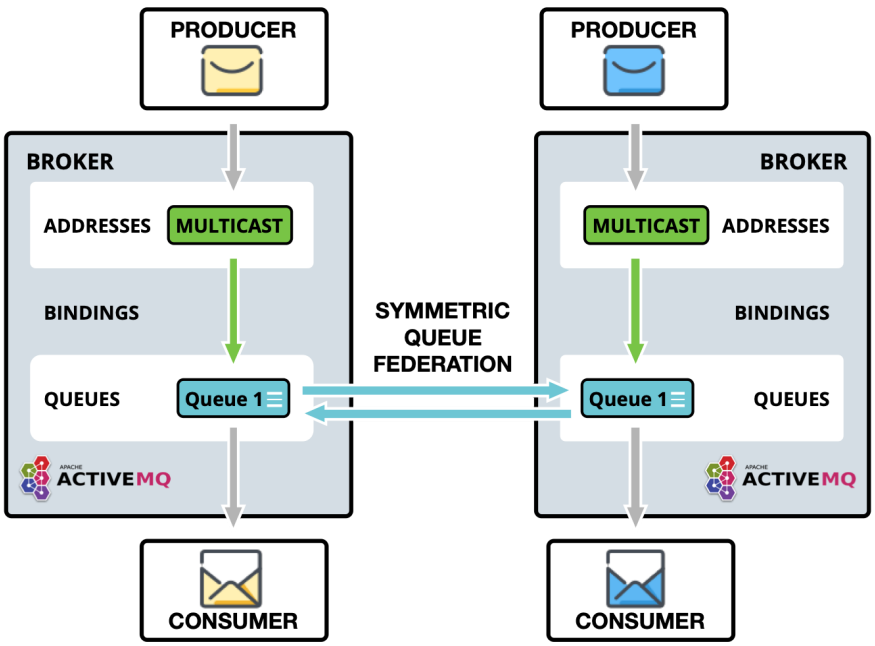


Figure 2. Queue Federation

For further details please goto Queue Federation.

WAN Full Mesh

With federation it is possible to provide a WAN mesh of brokers, replicating with Address Federation or routing and load balancing with Queue Federation.

Linking producers and consumers distant from each other.

Figure 3. Example possible full federation mesh

Configuring Federation

Federation is configured in `broker.xml`.

Sample:

```
<federations>
    <federation name="eu-north-1-federation">
        <upstream name="eu-west-1" user="westuser" password="32a10275cf4ab4e9"
           <static-connectors>
              <connector-ref>connector1</connector-ref>
           </static-connectors>
           <policy ref="policySetA"/>
        </upstream>
        <upstream name="eu-east-1" user="eastuser" password="32a10275cf4ab4e9"
           <discovery-group-ref discovery-group-name="ue-west-dg"/>
           <policy ref="policySetA"/>
        </upstream>

        <policy-set name="policySetA">
           <policy ref="address-federation" />
           <policy ref="queue-federation" />
        </policy-set>

        <queue-policy name="queue-federation" >
           <exclude queue-match="federated_queue" address-match="#" />
        </queue-policy>

        <address-policy name="address-federation" >
           <include address-match="federated_address" />
        </address-policy>
    </federation>
</federations>
```

In the above example we have shown the basic key parameters needed to configure federation for a queue and address to multiple upstream.

The example shows a broker `eu-north-1` connecting to two upstream brokers `eu-east-1` and `eu-west-1`, and applying queue federation to queue `federated_queue`, and also applying address federation to `federated_address`.

It is important that federation name is globally unique.

There are many configuration options that you can apply these are detailed in the individual docs for Address Federation and Queue Federation.

Address Federation

Introduction

Address federation is like full multicast over the connected brokers, in that every message sent to address on `Broker-A` will be delivered to every queue on that broker, but like wise will be delivered to `Broker-B` and all attached queues there.

Address federation dynamically links to other addresses in upstream or downstream brokers. It automatically creates a queue on the remote address for itself, to which then it consumes, copying to the local address, as though they were published directly to it.

The upstream brokers do not need to be reconfigured or the address, simply permissions to the address need to be given to the address for the downstream broker. Similarly the same applies for downstream configurations.

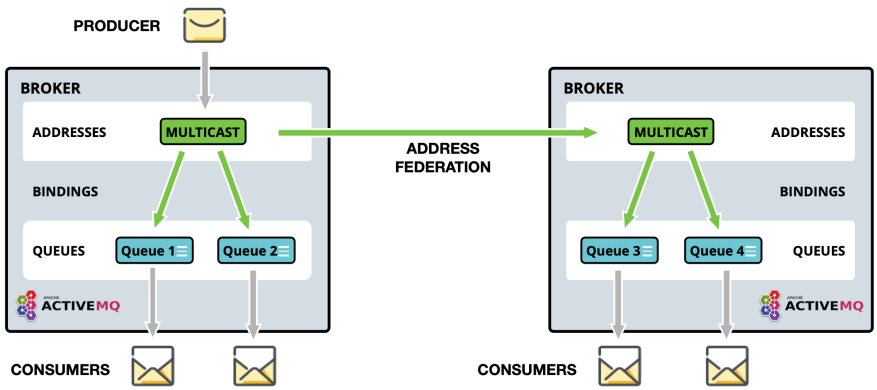


Figure 1. Address Federation

Topology Patterns

Symmetric

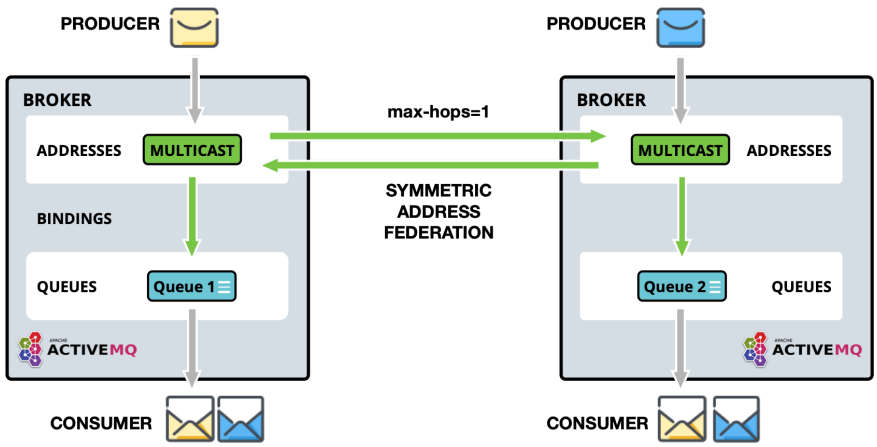


Figure 2. Address Federation - Symmetric

As seen above, a publisher and consumer are connected to each broker. Queues and thus consumers on those queues, can receive messages published by either publisher.

It is important in this setup to set `max-hops=1` to so that messages are copied only one and avoid cyclic replication. If `max-hops` is not configured correctly, consumers will get multiple copies of the same message.

Full Mesh

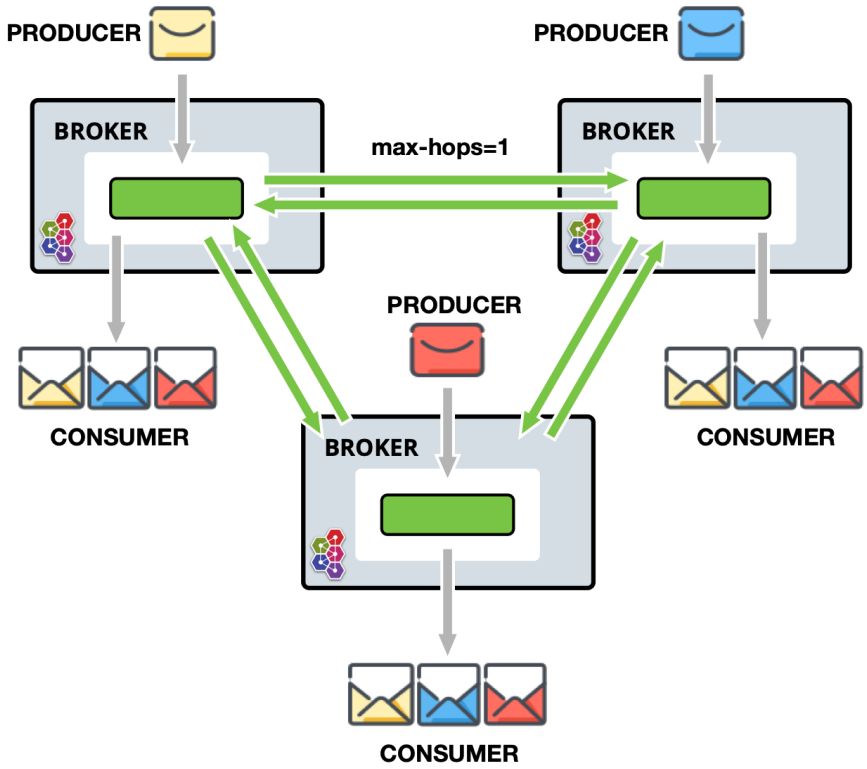


Figure 3. Address Federation - Full Mesh

If not already spotted, the setup is identical to symemtric but simply where all brokers are symmetrically federating each other, creating a full mesh.

As illustrated, a publisher and consumer are connected to each broker. Queues and thus consumers on those queues, can receive messages published by either publisher.

As with symmetric setup, it is important in this setup to set `max-hops=1` to so that messages are copied only one and avoid cyclic replication. If `max-hops` is not configured correctly, consumers will get multiple copies of the same message.

Ring

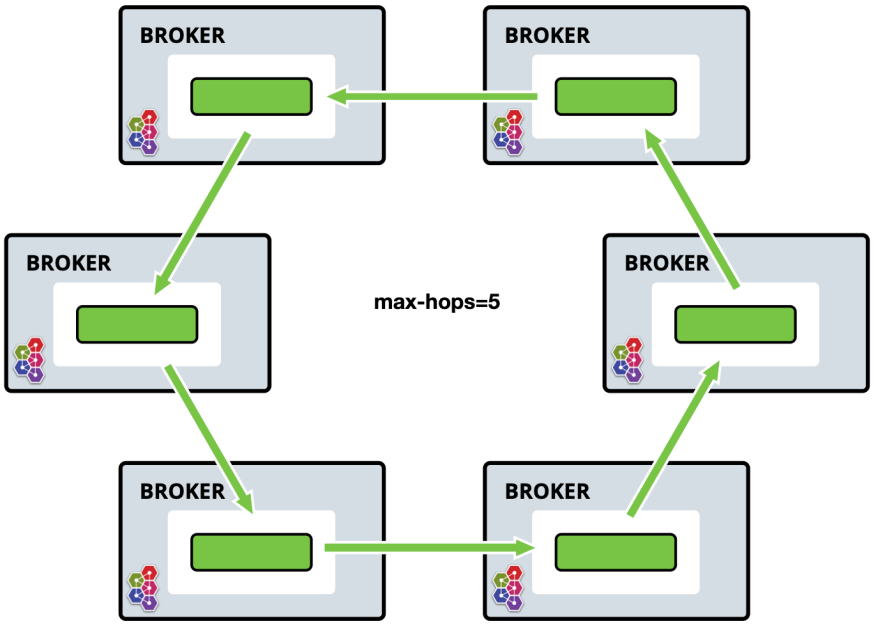


Figure 4. Address Federation - Symmetric

In a ring of brokers each federated address is `upstream` to just one other in the ring. To avoid the cyclic issue, it is important to set `max-hops` to `n - 1` where `n` is the number of nodes in the ring. e.g. in the example above property is set to 5 so that every address in the ring sees the message exactly once.

Whilst this setup is cheap in regards to connections, it is brittle, in that if a single broker fails, the ring fails.

Fan out

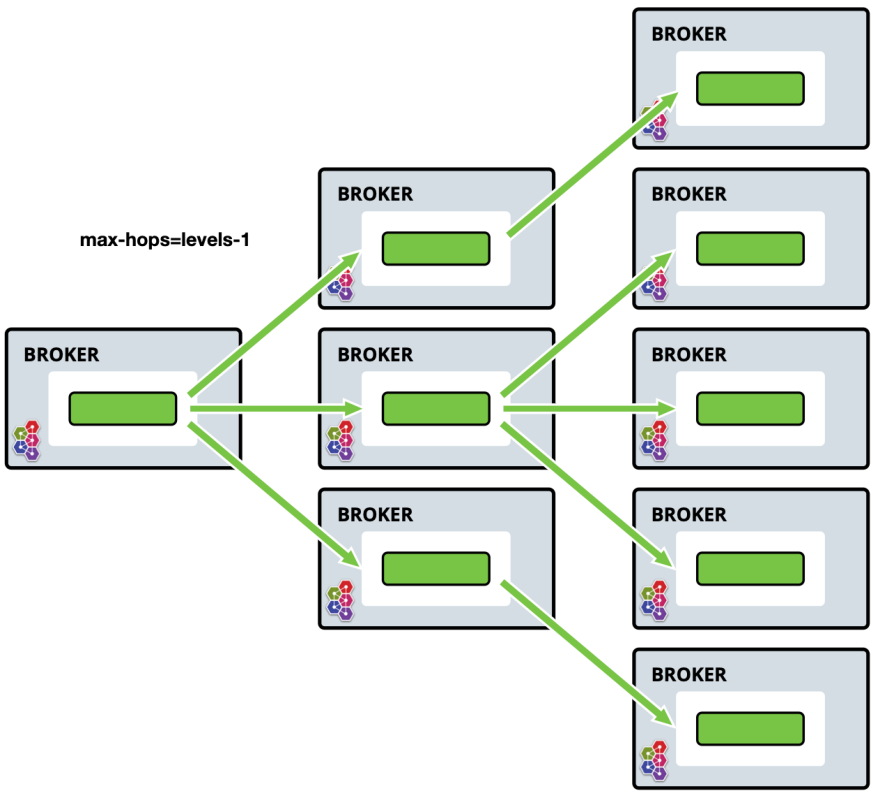


Figure 5. Address Federation - Fan Out

One master address (it would required no configuration) is linked to by a tree of downstream federated addresses, the tree can extend to any depth, and can be extended to without needing to re-configure existing brokers.

In this case messages published to the master address can be received by any consumer connected to any broker in the tree.

Divert Binding Support

Divert binding support can be added as part of the address policy configuration. This will allow the federation to respond to divert bindings to create demand. For example, let's say there is one address called "test.federation.source" that is included as a match for the federated address and another address called "test.federation.target" that is not included. Normally when a queue is created on "test.federation.target" this would not cause a federated consumer to be created because the address is not part of the included matches. However, if we create a divert binding such that "test.federation.source" is the source address and "test.federation.target" is the forwarded address then demand will now be created. The source address still must be multicast but the target address can be multicast or anycast.

An example use case for this might be a divert that redirects JMS topics (multicast addresses) to a JMS queue (anycast addresses) to allow for load balancing of the messages on a topic for legacy consumers not supporting JMS 2.0 and shared subscriptions.

Configuring Address Federation

Federation is configured in `broker.xml`.

Sample Address Federation setup:

```
<federations>
    <federation name="eu-north-1" user="federation_username" password="32a1027
        <upstream name="eu-east-1">
           <static-connectors>
              <connector-ref>eu-east-connector1</connector-ref>
              <connector-ref>eu-east-connector1</connector-ref>
           </static-connectors>
           <policy ref="news-address-federation"/>
        </upstream>
        <upstream name="eu-west-1" >
           <static-connectors>
              <connector-ref>eu-west-connector1</connector-ref>
              <connector-ref>eu-west-connector1</connector-ref>
           </static-connectors>
           <policy ref="news-address-federation"/>
        </upstream>

        <address-policy name="news-address-federation" max-hops="1" auto-delet
           <include address-match="queue.bbc.new" />
           <include address-match="queue.usatoday" />
           <include address-match="queue.news.#" />

           <exclude address-match="queue.news.sport.#" />
        </address-policy>

        <transformer name="news-transformer">
           <class-name>org.foo.NewsTransformer</class-name>
           <property key="key1" value="value1"/>
           <property key="key2" value="value2"/>
        </transformer>
    </federation>
</federations>
```

In the above setup downstream broker `eu-north-1` is configured to connect to two upstream brokers `eu-east-1` and `eu-east-2` , the credentials used for both connections to both brokers in this sample are shared, you can set user and password at the upstream level should they be different per upstream.

Both upstreams are configured with the same address-policy `news-address-federation` , that is selecting addresses which match any of the include criteria, but will exclude anything that starts `queue.news.sport` .

It is important that federation name is globally unique.

Let's take a look at all the `address-policy` parameters in turn, in order of priority.

- `name` attribute. All address-policies must have a unique name in the server.
- `include` the address-match pattern to whitelist addresses, multiple of these can be set. If none are set all addresses are matched.
- `exclude` the address-match pattern to blacklist addresses, multiple of these can be set.
- `max-hops` . The number of hops that a message can have made for it to be federated, see Topology Patterns above for more details.
- `auto-delete` . For address federation, the downstream dynamically creates a durable queue on the upstream address. This is used to mark if the upstream queue should be deleted once downstream disconnects, and the delay and

message count params have been met. This is useful if you want to automate the clean up, though you may wish to disable this if you want messages to queued for the downstream when disconnect no matter what.

- `auto-delete-delay`. The amount of time in milliseconds after the downstream broker has disconnected before the upstream queue can be eligable for `auto-delete`.
- `auto-delete-message-count`. The amount number messages in the upstream queue that the message count must be equal or below before the downstream broker has disconnected before the upstream queue can be eligable for `auto-delete`.
- `transformer-ref`. The ref name for a transformer (see transformer config) that you may wish to configure to transform the message on federation transfer.
- `enable-divert-bindings`. Setting to true will enable divert bindings to be listened for demand. If there is a divert binding with an address that matches the included addresses for the stream, any queue bindings that match the forward address of the divert will create demand. Default is false

note `address-policy`'s and `queue-policy`'s are able to be defined in the same federation, and be linked to the same upstream.

Now look at all the `transformer` parameters in turn, in order of priority:

- `name` attribute. This must be a unique name in the server, and is used to ref the transformer in `address-policy` and `queue-policy`
- `transformer-class-name`. An optional transformer-class-name can be specified. This is the name of a user-defined class which implements the `org.apache.activemq.artemis.core.server.transformer.Transformer` interface.

  If this is specified then the transformer's `transform()` method will be invoked with the message before it is transferred. This gives you the opportunity to transform the message's header or body before it is federated.
- `property` holds key, value pairs that can be used to configure the transformer.

Finally look at `upstream`, this is what defines the upstream broker connection and the policies to use against it.

- `name` attribute. This must be a unique name in the server, and is used to ref the transformer in `address-policy` and `queue-policy`
- `user`. This optional attribute determines the user name to use when creating the upstream connection to the remote server. If it is not specified the shared federation user and password will be used if set.
- `password`. This optional attribute determines the password to use when creating the upstream connection to the remote server. If it is not specified the shared federation user and password will be used if set.

- `static-connectors` or `discovery-group-ref`. Pick either of these options to connect the bridge to the target server.

  The `static-connectors` is a list of `connector-ref` elements pointing to `connector` elements defined elsewhere. A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Configuring the Transport.

  The `discovery-group-ref` element has one attribute - `discovery-group-name`. This attribute points to a `discovery-group` defined elsewhere. For more information about what discovery-groups are and how to configure them, please see Discovery Groups.

- `ha`. This optional parameter determines whether or not this bridge should support high availability. True means it will connect to any available server in a cluster and support failover. The default value is `false`.

- `circuit-breaker-timeout` in milliseconds, When a connection issue occurs, as the single connection is shared by many federated queue and address consumers, to avoid each one trying to reconnect and possibly causing a thundering heard issue, the first one will try, if unsuccessful the circuit breaker will open, returning the same exception to all, this is the timeout until the circuit can be closed and connection retried.

- `share-connection`. If there is a downstream and upstream connection configured for the same broker then the same connection will be shared as long as both stream configs set this flag to true. Default is false.

- `check-period`. The period (in milliseconds) used to check if the federation connection has failed to receive pings from another server. Default is 30000.

- `connection-ttl`. This is how long a federation connection should stay alive if it stops receiving messages from the remote broker. Default is 60000.

- `call-timeout`. When a packet is sent via a federation connection and is a blocking call, i.e. for acknowledgements, this is how long it will wait (in milliseconds) for the reply before throwing an exception. Default is 30000.

- `call-failover-timeout`. Similar to `call-timeout` but used when a call is made during a failover attempt. Default is -1 (no timeout).

- `retry-interval`. This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `500` milliseconds.

- `retry-interval-multiplier`. This is a multiplier used to increase the `retry-interval` after each reconnect attempt, default is 1.

- `max-retry-interval`. The maximum delay (in milliseconds) for retries. Default is 2000.

- `initial-connect-attempts`. The number of times the system will try to connect to the remote broker in the federation. If the max-retry is achieved this broker will be considered permanently down and the system will not route messages to this broker. Default is -1 (infinite retries).
- `reconnect-attempts`. The number of times the system will try to reconnect to the remote broker in the federation. If the max-retry is achieved this broker will be considered permanently down and the system will stop routing messages to this broker. Default is -1 (infinite retries).

Configuring Downstream Federation

Similarly to `upstream` configuration, a downstream configuration can be configured. This works by sending a command to the `downstream` broker to have it create an `upstream` connection back to the downstream broker. The benefit of this is being able to configure everything for federation on one broker in some cases to make it easier, such as a hub and spoke topology

All of the same configuration options apply to to `downstream` as does `upstream` with the exception of one extra configuration flag that needs to be set:

The `transport-connector-ref` is an element pointing to a `connector` elements defined elsewhere. This ref is used to tell the downstream broker what connector to use to create a new upstream connection back to the downstream broker.

A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Configuring the Transport.

Sample Downstream Address Federation setup:

```
   <!--Other config Here -->

<connectors>
   <connector name="netty-connector">tcp://localhost:61616</connector>
   <connector name="eu-west-1-connector">tcp://localhost:61616</connector>
   <connector name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>

<acceptors>
   <acceptor name="netty-acceptor">tcp://localhost:61616</acceptor>
</acceptors>

   <!--Other config Here -->

<federations>
   <federation name="eu-north-1" user="federation_username" password="32a10275
      <downstream name="eu-east-1">
          <static-connectors>
             <connector-ref>eu-east-connector1</connector-ref>
          </static-connectors>
          <transport-connector-ref>netty-connector</transport-connector-ref>
          <policy ref="news-address-federation"/>
      </downstream>
      <downstream name="eu-west-1" >
         <static-connectors>
            <connector-ref>eu-west-connector1</connector-ref>
         </static-connectors>
         <transport-connector-ref>netty-connector</transport-connector-ref>
         <policy ref="news-address-federation"/>
      </downstream>

      <address-policy name="news-address-federation" max-hops="1" auto-delete=
         <include address-match="queue.bbc.new" />
         <include address-match="queue.usatoday" />
         <include address-match="queue.news.#" />

         <exclude address-match="queue.news.sport.#" />
      </address-policy>

      <transformer name="news-transformer">
         <class-name>org.foo.NewsTransformer</class-name>
         <property key="key1" value="value1"/>
         <property key="key2" value="value2"/>
      </transformer>
   </federation>
</federations>
```

Queue Federation

Introduction

This feature provides a way of balancing the load of a single queue across remote brokers.

A federated queue links to other queues (called upstream queues). It will retrieve messages from upstream queues in order to satisfy demand for messages from local consumers. The upstream queues do not need to be reconfigured and they do not have to be on the same broker or in the same cluster.

All of the configuration needed to establish the upstream links and the federated queue is in the downstream broker.

Use Cases

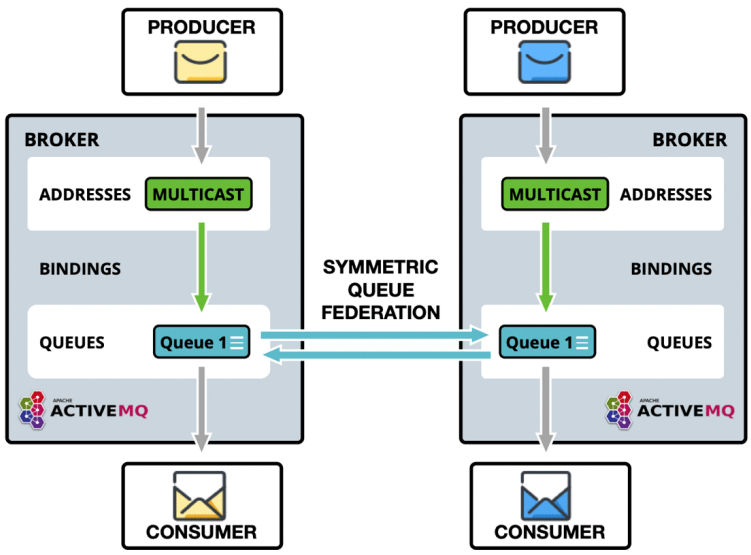
This is not an exhaustive list of what you can do with and the benefits of federated queues, but simply some ideas.

- Higher capacity

  By having a "logical" queue distributed over many brokers. Each broker would declare a federated queue with all the other federated queues upstream. (The links would form a complete bi-directional graph on n queues.)

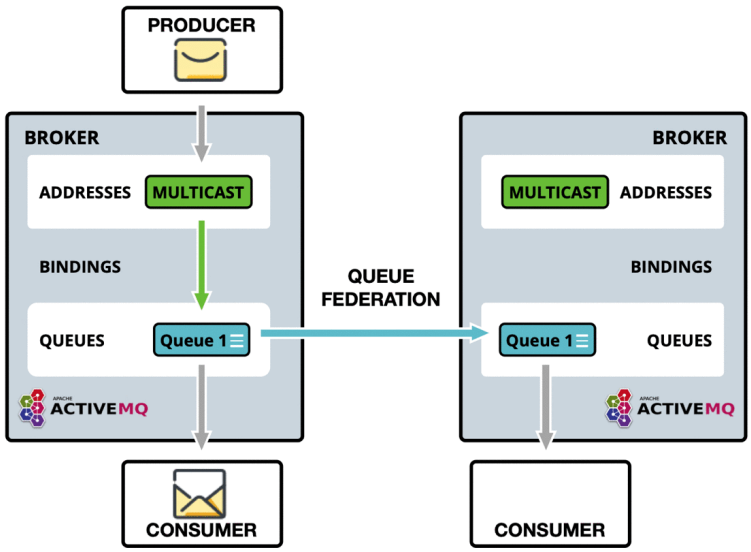
By having this a logical distributed queue is capable of having a much higher capacity than a single queue on a single broker. When will perform best when there is some degree of locality.

e.g. as many messages as possible are consumed from the same broker as they were published to, where federation only needs to move messages around in order to perform load balancing.



- Supporting multi region or venue

  In a multi region setup you may have producers in one region or venue and the consumer in another. typically you want producers and consumer to keep their connections local to the region, in such as case you can deploy brokers in each region where producers and consumer are, and use federation to move messages over the WAN between regions.



- Communication between the secure enterprise lan and the DMZ.

  Where a number of producer apps maybe in the DMZ and a number of consumer apps in the secure enterprise lan, it may not suitable to allow the producers to connect through to the broker in the secure enterprise lan.

  In this scenario you could deploy a broker in the DMZ where the producers publish to, and then have the broker in the enterprise lan connect out to the DMZ broker and federate the queues so that messages can traverse.

  This is similar to supporting multi region or venue.

- Migrating between two clusters. Consumers and publishers can be moved in any order and the messages won't be duplicated (which is the case if you do exchange federation). Instead, messages are transferred to the new cluster when your consumers are there. Here for such a migration with blue/green or canary moving a number of consumers on the same queue, you may want to set the `priority-adjustment` to 0, or even a positive value, so message would actively flow to the federated queue.

Configuring Queue Federation

Federation is configured in `broker.xml`.

Sample Queue Federation setup:

```
<federations>
    <federation name="eu-north-1" user="federation_username" password="32a1027
        <upstream name="eu-east-1">
           <static-connectors>
              <connector-ref>eu-east-connector1</connector-ref>
              <connector-ref>eu-east-connector1</connector-ref>
           </static-connectors>
           <policy ref="news-queue-federation"/>
        </upstream>
        <upstream name="eu-west-1" >
           <static-connectors>
              <connector-ref>eu-west-connector1</connector-ref>
              <connector-ref>eu-west-connector1</connector-ref>
           </static-connectors>
           <policy ref="news-queue-federation"/>
        </upstream>

        <queue-policy name="news-queue-federation" priority-adjustment="-5" in
           <include queue-match="#" address-match="queue.bbc.new" />
           <include queue-match="#" address-match="queue.usatoday" />
           <include queue-match="#" address-match="queue.news.#" />

           <exclude queue-match="#.local" address-match="#" />
        </queue-policy>

        <transformer name="news-transformer">
           <class-name>org.foo.NewsTransformer</class-name>
           <property key="key1" value="value1"/>
           <property key="key2" value="value2"/>
        </transformer>
    </federation>
</federations>
```

In the above setup downstream broker `eu-north-1` is configured to connect to two upstream brokers `eu-east-1` and `eu-east-2` , the credentials used for both connections to both brokers in this sample are shared, you can set user and password at the upstream level should they be different per upstream.

Both upstreams are configured with the same queue-policy `news-queue-federation` , that is selecting addresses which match any of the include criteria, but will exclude any queues that end with `.local` , keeping these as local queues only.

It is important that federation name is globally unique.

Let's take a look at all the `queue-policy` parameters in turn, in order of priority.

- `name` attribute. All address-policies must have a unique name in the server.
- `include` the address-match pattern to whitelist addresses, multiple of these can be set. If none are set all addresses are matched.
- `exclude` the address-match pattern to blacklist addresses, multiple of these can be set.
- `priority-adjustment` when a consumer attaches its priority is used to make the upstream consumer, but with an adjustment by default -1, so that local consumers get load balanced first over remote, this enables this to be configurable should it be wanted/needed.

- `include-federated` by default this is false, we dont federate a federated consumer, this is to avoid issue, where in symmetric or any closed loop setup you could end up when no "real" consumers attached with messages flowing round and round endlessly.

  There is though a valid case that if you dont have a close loop setup e.g. three brokers in a chain (A->B->C) with producer at broker A and consumer at C, you would want broker B to re-federate the consumer onto A.

- `transformer-ref` . The ref name for a transformer (see transformer config) that you may wish to configure to transform the message on federation transfer.

note `address-policy` 's and `queue-policy` 's are able to be defined in the same federation, and be linked to the same upstream.

Now look at all the `transformer` parameters in turn, in order of priority:

- `name` attribute. This must be a unique name in the server, and is used to ref the transformer in `address-policy` and `queue-policy`

- `transformer-class-name` . An optional transformer-class-name can be specified. This is the name of a user-defined class which implements the `org.apache.activemq.artemis.core.server.transformer.Transformer` interface.

  If this is specified then the transformer's `transform()` method will be invoked with the message before it is transferred. This gives you the opportunity to transform the message's header or body before it is federated.

- `property` holds key, value pairs that can be used to configure the transformer.

Finally look at `upstream` , this is what defines the upstream broker connection and the policies to use against it.

- `name` attribute. This must be a unique name in the server, and is used to ref the transformer in `address-policy` and `queue-policy`

- `user` . This optional attribute determines the user name to use when creating the upstream connection to the remote server. If it is not specified the shared federation user and password will be used if set.

- `password` . This optional attribute determines the password to use when creating the upstream connection to the remote server. If it is not specified the shared federation user and password will be used if set.

- `static-connectors` or `discovery-group-ref` . Pick either of these options to connect the bridge to the target server.

  The `static-connectors` is a list of `connector-ref` elements pointing to `connector` elements defined elsewhere. A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Configuring the Transport.

The `discovery-group-ref` element has one attribute - `discovery-group-name`. This attribute points to a `discovery-group` defined elsewhere. For more information about what discovery-groups are and how to configure them, please see Discovery Groups.

- `ha`. This optional parameter determines whether or not this bridge should support high availability. True means it will connect to any available server in a cluster and support failover. The default value is `false`.
- `circuit-breaker-timeout` in milliseconds, When a connection issue occurs, as the single connection is shared by many federated queue and address consumers, to avoid each one trying to reconnect and possibly causing a thrundering heard issue, the first one will try, if unsuccessful the circuit breaker will open, returning the same exception to all, this is the timeout until the circuit can be closed and connection retried.
- `share-connection`. If there is a downstream and upstream connection configured for the same broker then the same connection will be shared as long as both stream configs set this flag to true. Default is false.
- `check-period`. The period (in milliseconds) used to check if the federation connection has failed to receive pings from another server. Default is 30000.
- `connection-ttl`. This is how long a federation connection should stay alive if it stops receiving messages from the remote broker. Default is 60000.
- `call-timeout`. When a packet is sent via a federation connection and is a blocking call, i.e. for acknowledgements, this is how long it will wait (in milliseconds) for the reply before throwing an exception. Default is 30000.
- `call-failover-timeout`. Similar to `call-timeout` but used when a call is made during a failover attempt. Default is -1 (no timeout).
- `retry-interval`. This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `500` milliseconds.
- `retry-interval-multiplier`. This is a multiplier used to increase the `retry-interval` after each reconnect attempt, default is 1.
- `max-retry-interval`. The maximum delay (in milliseconds) for retries. Default is 2000.
- `initial-connect-attempts`. The number of times the system will try to connect to the remote broker in the federation. If the max-retry is achieved this broker will be considered permanently down and the system will not route messages to this broker. Default is -1 (infinite retries).
- `reconnect-attempts`. The number of times the system will try to reconnect to the remote broker in the federation. If the max-retry is achieved this broker will be considered permanently down and the system will stop routing messages to this broker. Default is -1 (infinite retries).

Configuring Downstream Federation

Similarly to `upstream` configuration, a downstream configuration can be configured. This works by sending a command to the `downstream` broker to have it create an `upstream` connection back to the downstream broker. The benefit of this is being able to configure everything for federation on one broker in some cases to make it easier, such as a hub and spoke topology.

All of the same configuration options apply to to `downstream` as does `upstream` with the exception of one extra configuration flag that needs to be set:

The `transport-connector-ref` is an element pointing to a `connector` elements defined elsewhere. This ref is used to tell the downstream broker what connector to use to create a new upstream connection back to the downstream broker.

A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Configuring the Transport.

Sample Downstream Address Federation setup:

```
  <!--Other config Here -->

<connectors>
   <connector name="netty-connector">tcp://localhost:61616</connector>
   <connector name="eu-west-1-connector">tcp://localhost:61616</connector>
   <connector name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>

<acceptors>
   <acceptor name="netty-acceptor">tcp://localhost:61616</acceptor>
</acceptors>

   <!--Other config Here -->

<federations>
   <federation name="eu-north-1" user="federation_username" password="32a10275
      <downstream name="eu-east-1">
          <static-connectors>
             <connector-ref>eu-east-connector1</connector-ref>
          </static-connectors>
          <transport-connector-ref>netty-connector</transport-connector-ref>
          <policy ref="news-address-federation"/>
      </downstream>
      <downstream name="eu-west-1" >
         <static-connectors>
            <connector-ref>eu-west-connector1</connector-ref>
         </static-connectors>
         <transport-connector-ref>netty-connector</transport-connector-ref>
         <policy ref="news-address-federation"/>
      </downstream>

      <queue-policy name="news-queue-federation" priority-adjustment="-5" incl
         <include queue-match="#" address-match="queue.bbc.new" />
         <include queue-match="#" address-match="queue.usatoday" />
         <include queue-match="#" address-match="queue.news.#" />

         <exclude queue-match="#.local" address-match="#" />
      </queue-policy>

      <transformer name="news-transformer">
         <class-name>org.foo.NewsTransformer</class-name>
         <property key="key1" value="value1"/>
         <property key="key2" value="value2"/>
      </transformer>
   </federation>
</federations>
```

High Availability and Failover

We define high availability as the *ability for the system to continue functioning after failure of one or more of the servers*.

A part of high availability is *failover* which we define as the *ability for client connections to migrate from one server to another in event of server failure so client applications can continue to operate*.

Live - Backup Groups

Apache ActiveMQ Artemis allows servers to be linked together as *live - backup* groups where each live server can have 1 or more backup servers. A backup server is owned by only one live server. Backup servers are not operational until failover occurs, however 1 chosen backup, which will be in passive mode, announces its status and waits to take over the live servers work

Before failover, only the live server is serving the Apache ActiveMQ Artemis clients while the backup servers remain passive or awaiting to become a backup server. When a live server crashes or is brought down in the correct mode, the backup server currently in passive mode will become live and another backup server will become passive. If a live server restarts after a failover then it will have priority and be the next server to become live when the current live server goes down, if the current live server is configured to allow automatic failback then it will detect the live server coming back up and automatically stop.

HA Policies

Apache ActiveMQ Artemis supports two different strategies for backing up a server *shared store* and *replication*. Which is configured via the `ha-policy` configuration element.

```
<ha-policy>
   <replication/>
</ha-policy>
```

or

```
<ha-policy>
   <shared-store/>
</ha-policy>
```

As well as these 2 strategies there is also a 3rd called `live-only`. This of course means there will be no Backup Strategy and is the default if none is provided, however this is used to configure `scale-down` which we will cover in a later chapter.

> **Note:**
>
> The `ha-policy` configurations replaces any current HA configuration in the root of the `broker.xml` configuration. All old configuration is now deprecated although best efforts will be made to honour it if configured this way.
>
> **Note:**
>
> Only persistent message data will survive failover. Any non persistent message data will not be available after failover.

The `ha-policy` type configures which strategy a cluster should use to provide the backing up of a servers data. Within this configuration element is configured how a server should behave within the cluster, either as a master (live), slave (backup) or colocated (both live and backup). This would look something like:

```
<ha-policy>
   <replication>
      <master/>
   </replication>
</ha-policy>
```

or

```
<ha-policy>
   <shared-store>
      <slave/>
   </shared-store>
</ha-policy>
```

or

```
<ha-policy>
   <replication>
      <colocated/>
   </replication>
</ha-policy>
```

Data Replication

When using replication, the live and the backup servers do not share the same data directories, all data synchronization is done over the network. Therefore all (persistent) data received by the live server will be duplicated to the backup.

Notice that upon start-up the backup server will first need to synchronize all existing data from the live server before becoming capable of replacing the live server should it fail. So unlike when using shared storage, a replicating backup will not be a fully operational backup right after start-up, but only after it finishes synchronizing the data with its live server. The time it will take for this to happen will depend on the amount of data to be synchronized and the connection speed.

> **Note:**
>
> In general, synchronization occurs in parallel with current network traffic so this won't cause any blocking on current clients. However, there is a critical moment at the end of this process where the replicating server must complete the synchronization and ensure the replica acknowledges this completion. This exchange between the replicating server and replica will block any journal related operations. The maximum length of time that this exchange will block is controlled by the `initial-replication-sync-timeout` configuration element.

Replication will create a copy of the data at the backup. One issue to be aware of is: in case of a successful fail-over, the backup's data will be newer than the one at the live's storage. If you configure your live server to perform a failback to live server when restarted, it will synchronize its data with the backup's. If both servers are shutdown, the administrator will have to determine which one has the latest data.

The replicating live and backup pair must be part of a cluster. The Cluster Connection also defines how backup servers will find the remote live servers to pair with. Refer to Clusters for details on how this is done, and how to configure a cluster connection. Notice that:

- Both live and backup servers must be part of the same cluster. Notice that even a simple live/backup replicating pair will require a cluster configuration.
- Their cluster user and password must match.

Within a cluster, there are two ways that a backup server will locate a live server to replicate from, these are:

- `specifying a node group`. You can specify a group of live servers that a backup server can connect to. This is done by configuring `group-name` in either the `master` or the `slave` element of the `broker.xml`. A Backup server will only connect to a live server that shares the same node group name
- `connecting to any live`. This will be the behaviour if `group-name` is not configured allowing a backup server to connect to any live server

> **Note:**
>
> A `group-name` example: suppose you have 5 live servers and 6 backup servers:
>
> - `live1`, `live2`, `live3`: with `group-name=fish`
> - `live4`, `live5`: with `group-name=bird`
> - `backup1`, `backup2`, `backup3`, `backup4`: with `group-name=fish`
> - `backup5`, `backup6`: with `group-name=bird`
>
> After joining the cluster the backups with `group-name=fish` will search for live servers with `group-name=fish` to pair with. Since there is one backup too many, the `fish` will remain with one spare backup.
>
> The 2 backups with `group-name=bird` (`backup5` and `backup6`) will pair with live servers `live4` and `live5`.

The backup will search for any live server that it is configured to connect to. It then tries to replicate with each live server in turn until it finds a live server that has no current backup configured. If no live server is available it will wait until the cluster topology changes and repeats the process.

> **Note:**
>
> This is an important distinction from a shared-store backup, if a backup starts and does not find a live server, the server will just activate and start to serve client requests. In the replication case, the backup just keeps waiting for a live server to pair with. Note that in replication the backup server does not know whether any data it might have is up to date, so it really cannot decide to activate automatically. To activate a replicating backup server using the data it has, the administrator must change its configuration to make it a live server by changing `slave` to `master`.

Much like in the shared-store case, when the live server stops or crashes, its replicating backup will become active and take over its duties. Specifically, the backup will become active when it loses connection to its live server. This can be problematic because this can also happen because of a temporary network problem. In order to address this issue, the backup will try to determine whether it still can connect to the other servers in the cluster. If it can connect to more than half the servers, it will become active, if more than half the servers also disappeared with the live, the backup will wait and try reconnecting with the live. This avoids a split brain situation.

Configuration

To configure the live and backup servers to be a replicating pair, configure the live server in ' `broker.xml` to have:

```
<ha-policy>
   <replication>
      <master/>
   </replication>
</ha-policy>
...
<cluster-connections>
   <cluster-connection name="my-cluster">
      ...
   </cluster-connection>
</cluster-connections>
```

The backup server must be similarly configured but as a `slave`

```
<ha-policy>
   <replication>
      <slave/>
   </replication>
</ha-policy>
```

All Replication Configuration

The following table lists all the `ha-policy` configuration elements for HA strategy Replication for `master`:

- `check-for-live-server`

Whether to check the cluster for a (live) server using our own server ID when starting up. This is an important option to avoid split-brain when failover happens and the master is restarted. Default is `false`.

- `cluster-name`

Name of the cluster configuration to use for replication. This setting is only necessary if you configure multiple cluster connections. If configured then the connector configuration of the cluster configuration with this name will be used when connecting to the cluster to discover if a live server is already running, see `check-for-live-server`. If unset then the default cluster connections configuration is used (the first one configured).

- `group-name`

If set, backup servers will only pair with live servers with matching group-name.

- `initial-replication-sync-timeout`

The amount of time the replicating server will wait at the completion of the initial replication process for the replica to acknowledge it has received all the necessary data. The default is 30,000 milliseconds. **Note:** during this interval any journal related operations will be blocked.

The following table lists all the `ha-policy` configuration elements for HA strategy Replication for `slave`:

- `cluster-name`

Name of the cluster configuration to use for replication. This setting is only necessary if you configure multiple cluster connections. If configured then the connector configuration of the cluster configuration with this name will be used when connecting to the cluster to discover if a live server is already running, see `check-for-live-server`. If unset then the default cluster connections configuration is used (the first one configured).

- `group-name`

If set, backup servers will only pair with live servers with matching group-name

- `max-saved-replicated-journals-size`

This specifies how many times a replicated backup server can restart after moving its files on start. Once there are this number of backup journal files the server will stop permanently after if fails back.

- `allow-failback`

Whether a server will automatically stop when another places a request to take over its place. The use case is when the backup has failed over.

- `initial-replication-sync-timeout`

After failover and the slave has become live, this is set on the new live server. It represents the amount of time the replicating server will wait at the completion of the initial replication process for the replica to acknowledge it has received all the necessary data. The default is 30,000 milliseconds. **Note:** during this interval any journal related operations will be blocked.

Shared Store

When using a shared store, both live and backup servers share the *same* entire data directory using a shared file system. This means the paging directory, journal directory, large messages and binding journal.

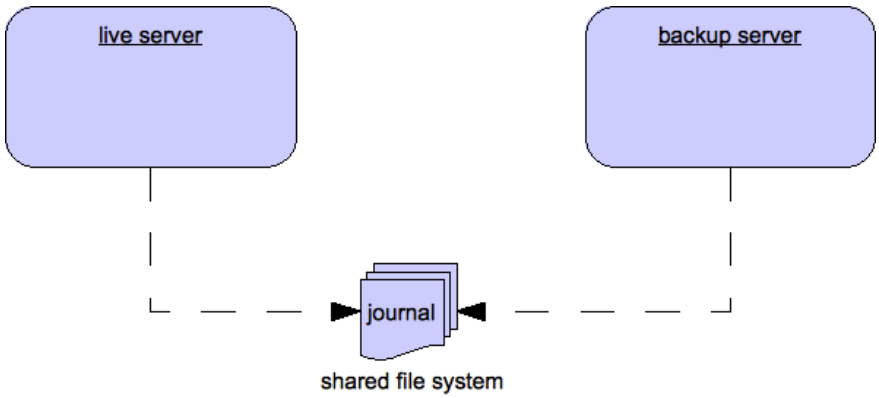
When failover occurs and a backup server takes over, it will load the persistent storage from the shared file system and clients can connect to it.

This style of high availability differs from data replication in that it requires a shared file system which is accessible by both the live and backup nodes. Typically this will be some kind of high performance Storage Area Network (SAN). We do not recommend you use Network Attached Storage (NAS), e.g. NFS mounts to store any shared journal (NFS is slow).

The advantage of shared-store high availability is that no replication occurs between the live and backup nodes, this means it does not suffer any performance penalties due to the overhead of replication during normal operation.

The disadvantage of shared store replication is that it requires a shared file system, and when the backup server activates it needs to load the journal from the shared store which can take some time depending on the amount of data in the store.

If you require the highest performance during normal operation, have access to a fast SAN and live with a slightly slower failover (depending on amount of data).



Configuration

To configure the live and backup servers to share their store, configure id via the `ha-policy` configuration in `broker.xml` :

```
<ha-policy>
   <shared-store>
      <master/>
   </shared-store>
</ha-policy>
...
<cluster-connections>
   <cluster-connection name="my-cluster">
      ...
   </cluster-connection>
</cluster-connections>
```

The backup server must also be configured as a backup.

```
<ha-policy>
   <shared-store>
      <slave/>
   </shared-store>
</ha-policy>
```

In order for live - backup groups to operate properly with a shared store, both servers must have configured the location of journal directory to point to the *same shared location* (as explained in Configuring the message journal)

> **Note:**
>
> todo write something about GFS

Also each node, live and backups, will need to have a cluster connection defined even if not part of a cluster. The Cluster Connection info defines how backup servers announce there presence to its live server or any other nodes in the cluster. Refer to Clusters for details on how this is done.

Failing Back to live Server

After a live server has failed and a backup taken has taken over its duties, you may want to restart the live server and have clients fail back.

In case of "shared disk", simply restart the original live server and kill the new live server. You can do this by killing the process itself. Alternatively you can set `allow-fail-back` to `true` on the slave config which will force the backup that has become live to automatically stop. This configuration would look like:

```
<ha-policy>
   <shared-store>
      <slave>
         <allow-failback>true</allow-failback>
      </slave>
   </shared-store>
</ha-policy>
```

In replication HA mode you need to set an extra property `check-for-live-server` to `true` in the `master` configuration. If set to true, during start-up a live server will first search the cluster for another server using its nodeID. If it finds one, it will contact this server and try to "fail-back". Since this is a remote replication scenario, the "starting live" will have to synchronize its data with the server running with its ID, once they are in sync, it will request the other server (which it assumes it is a back that has assumed its duties) to shutdown for it to take over. This is necessary because otherwise the live server has no means to know whether there was a fail-over or not, and if there was if the server that took its duties is still running or not. To configure this option at your `broker.xml` configuration file as follows:

```
<ha-policy>
   <replication>
      <master>
         <check-for-live-server>true</check-for-live-server>
      </master>
   </replication>
</ha-policy>
```

> **Warning**
>
> Be aware that if you restart a live server while after failover has occurred then `check-for-live-server` must be set to `true`. If not the live server will restart and server the same messages that the backup has already handled causing duplicates.

It is also possible, in the case of shared store, to cause failover to occur on normal server shutdown, to enable this set the following property to true in the `ha-policy` configuration on either the `master` or `slave` like so:

```
<ha-policy>
   <shared-store>
      <master>
         <failover-on-shutdown>true</failover-on-shutdown>
      </master>
   </shared-store>
</ha-policy>
```

By default this is set to false, if by some chance you have set this to false but still want to stop the server normally and cause failover then you can do this by using the management API as explained at Management

You can also force the running live server to shutdown when the old live server comes back up allowing the original live server to take over automatically by setting the following property in the `broker.xml` configuration file as follows:

```
<ha-policy>
   <shared-store>
      <slave>
         <allow-failback>true</allow-failback>
      </slave>
   </shared-store>
</ha-policy>
```

All Shared Store Configuration

The following table lists all the `ha-policy` configuration elements for HA strategy shared store for `master`:

- `failover-on-shutdown`

  If set to true then when this server is stopped normally the backup will become live assuming failover. If false then the backup server will remain passive. Note that if false you want failover to occur the you can use the the management API as explained at Management.

- `wait-for-activation`

  If set to true then server startup will wait until it is activated. If set to false then server startup will be done in the background. Default is true.

The following table lists all the `ha-policy` configuration elements for HA strategy Shared Store for `slave`:

- `failover-on-shutdown`

  In the case of a backup that has become live. then when set to true then when this server is stopped normally the backup will become liveassuming failover. If false then the backup server will remain passive. Note that if false you want failover to occur the you can use the the management API as explained at Management.

- `allow-failback`

  Whether a server will automatically stop when another places a request to take over its place. The use case is when the backup has failed over.

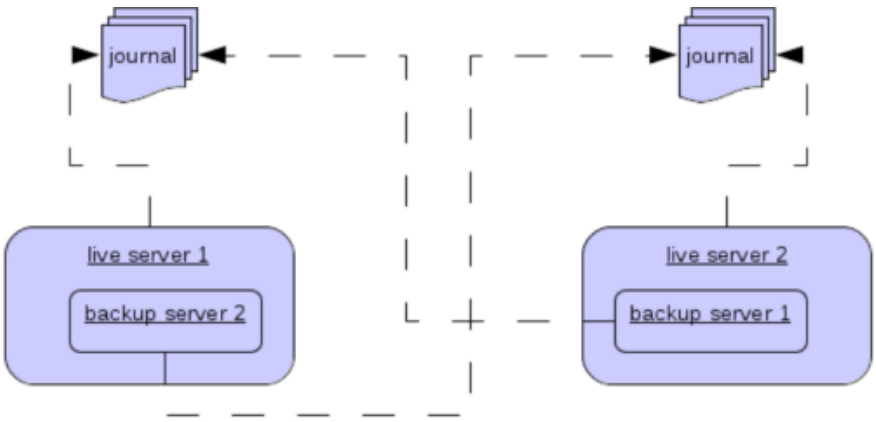
Colocated Backup Servers

It is also possible when running standalone to colocate backup servers in the same JVM as another live server. Live Servers can be configured to request another live server in the cluster to start a backup server in the same JVM either using shared store or replication. The new backup server will inherit its

configuration from the live server creating it apart from its name, which will be set to `colocated_backup_n` where n is the number of backups the server has created, and any directories and its Connectors and Acceptors which are discussed later on in this chapter. A live server can also be configured to allow requests from backups and also how many backups a live server can start. this way you can evenly distribute backups around the cluster. This is configured via the `ha-policy` element in the `broker.xml` file like so:

```
<ha-policy>
   <replication>
      <colocated>
         <request-backup>true</request-backup>
         <max-backups>1</max-backups>
         <backup-request-retries>-1</backup-request-retries>
         <backup-request-retry-interval>5000</backup-request-retry-interval>
         <master/>
         <slave/>
      </colocated>
   </replication>
</ha-policy>
```

the above example is configured to use replication, in this case the `master` and `slave` configurations must match those for normal replication as in the previous chapter. `shared-store` is also supported



Configuring Connectors and Acceptors

If the HA Policy is colocated then connectors and acceptors will be inherited from the live server creating it and offset depending on the setting of `backup-port-offset` configuration element. If this is set to say 100 (which is the default) and a connector is using port 61616 then this will be set to 61716 for the first server created, 61816 for the second, and so on.

> **Note:**
>
> for INVM connectors and Acceptors the id will have `colocated_backup_n` appended, where n is the backup server number.

Remote Connectors

It may be that some of the Connectors configured are for external servers and hence should be excluded from the offset. for instance a connector used by the cluster connection to do quorum voting for a replicated backup server, these can be omitted from being offset by adding them to the `ha-policy` configuration like so:

```
<ha-policy>
   <replication>
      <colocated>
         <excludes>
            <connector-ref>remote-connector</connector-ref>
         </excludes>
.........
</ha-policy>
```

Configuring Directories

Directories for the Journal, Large messages and Paging will be set according to what the HA strategy is. If shared store the the requesting server will notify the target server of which directories to use. If replication is configured then directories will be inherited from the creating server but have the new backups name appended.

The following table lists all the `ha-policy` configuration elements for colocated policy:

- `request-backup`

  If true then the server will request a backup on another node

- `backup-request-retries`

  How many times the live server will try to request a backup, -1 means for ever.

- `backup-request-retry-interval`

  How long to wait for retries between attempts to request a backup server.

- `max-backups`

  How many backups a live server can create

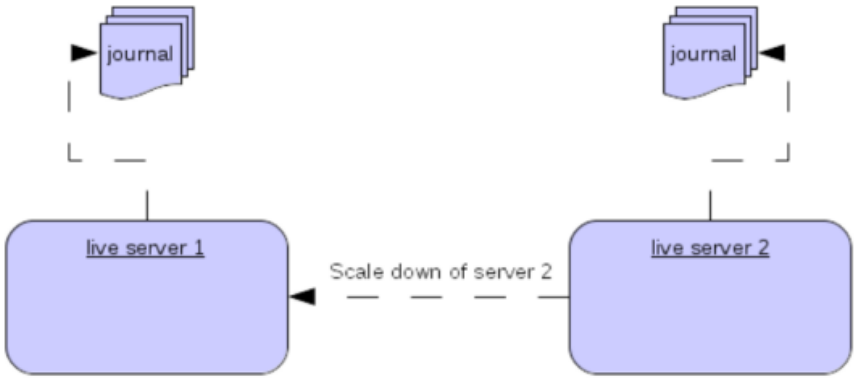
- `backup-port-offset`

  The offset to use for the Connectors and Acceptors when creating a new backup server.

Scaling Down

An alternative to using Live/Backup groups is to configure scaledown. when configured for scale down a server can copy all its messages and transaction state to another live server. The advantage of this is that you dont need full backups to provide some form of HA, however there are disadvantages with this approach the first being that it only deals with a server being stopped and not a server crash. The caveat here is if you configure a backup to scale down.

Another disadvantage is that it is possible to lose message ordering. This happens in the following scenario, say you have 2 live servers and messages are distributed evenly between the servers from a single producer, if one of the servers scales down then the messages sent back to the other server will be in the queue after the ones already there, so server 1 could have messages 1,3,5,7,9 and server 2 would have 2,4,6,8,10, if server 2 scales down the order in server 1 would be 1,3,5,7,9,2,4,6,8,10.



The configuration for a live server to scale down would be something like:

```
<ha-policy>
   <live-only>
      <scale-down>
         <connectors>
            <connector-ref>server1-connector</connector-ref>
         </connectors>
      </scale-down>
   </live-only>
</ha-policy>
```

In this instance the server is configured to use a specific connector to scale down, if a connector is not specified then the first INVM connector is chosen, this is to make scale down fromm a backup server easy to configure. It is also possible to use discovery to scale down, this would look like:

```
<ha-policy>
   <live-only>
      <scale-down>
         <discovery-group-ref discovery-group-name="my-discovery-group"/>
      </scale-down>
   </live-only>
</ha-policy>
```

Scale Down with groups

It is also possible to configure servers to only scale down to servers that belong in the same group. This is done by configuring the group like so:

```
<ha-policy>
   <live-only>
      <scale-down>
         ...
         <group-name>my-group</group-name>
      </scale-down>
   </live-only>
</ha-policy>
```

In this scenario only servers that belong to the group `my-group` will be scaled down to

Scale Down and Backups

It is also possible to mix scale down with HA via backup servers. If a slave is configured to scale down then after failover has occurred, instead of starting fully the backup server will immediately scale down to another live server. The most appropriate configuration for this is using the `colocated` approach. it means as you bring up live server they will automatically be backed up by server and as live servers are shutdown, there messages are made available on another live server. A typical configuration would look like:

```
<ha-policy>
   <replication>
      <colocated>
         <backup-request-retries>44</backup-request-retries>
         <backup-request-retry-interval>33</backup-request-retry-interval>
         <max-backups>3</max-backups>
         <request-backup>false</request-backup>
         <backup-port-offset>33</backup-port-offset>
         <master>
            <group-name>purple</group-name>
            <check-for-live-server>true</check-for-live-server>
            <cluster-name>abcdefg</cluster-name>
         </master>
         <slave>
            <group-name>tiddles</group-name>
            <max-saved-replicated-journals-size>22</max-saved-replicated-journ
            <cluster-name>33rrrrr</cluster-name>
            <restart-backup>false</restart-backup>
            <scale-down>
               <!--a grouping of servers that can be scaled down to-->
               <group-name>boo!</group-name>
               <!--either a discovery group-->
               <discovery-group-ref discovery-group-name="wahey"/>
            </scale-down>
         </slave>
      </colocated>
   </replication>
</ha-policy>
```

Scale Down and Clients

When a server is stopping and preparing to scale down it will send a message to all its clients informing them which server it is scaling down to before disconnecting them. At this point the client will reconnect however this will only succeed once the server has completed scaledown. This is to ensure that any

state such as queues or transactions are there for the client when it reconnects. The normal reconnect settings apply when the client is reconnecting so these should be high enough to deal with the time needed to scale down.

Failover Modes

Apache ActiveMQ Artemis defines two types of client failover:

- Automatic client failover
- Application-level client failover

Apache ActiveMQ Artemis also provides 100% transparent automatic reattachment of connections to the same server (e.g. in case of transient network problems). This is similar to failover, except it is reconnecting to the same server and is discussed in Client Reconnection and Session Reattachment

During failover, if the client has consumers on any non persistent or temporary queues, those queues will be automatically recreated during failover on the backup node, since the backup node will not have any knowledge of non persistent queues.

Automatic Client Failover

Apache ActiveMQ Artemis clients can be configured to receive knowledge of all live and backup servers, so that in event of connection failure at the client - live server connection, the client will detect this and reconnect to the backup server. The backup server will then automatically recreate any sessions and consumers that existed on each connection before failover, thus saving the user from having to hand-code manual reconnection logic.

Apache ActiveMQ Artemis clients detect connection failure when it has not received packets from the server within the time given by `client-failure-check-period` as explained in section Detecting Dead Connections. If the client does not receive data in good time, it will assume the connection has failed and attempt failover. Also if the socket is closed by the OS, usually if the server process is killed rather than the machine itself crashing, then the client will failover straight away.

Apache ActiveMQ Artemis clients can be configured to discover the list of live-backup server groups in a number of different ways. They can be configured explicitly or probably the most common way of doing this is to use *server discovery* for the client to automatically discover the list. For full details on how to configure server discovery, please see Clusters. Alternatively, the clients can explicitly connect to a specific server and download the current servers and backups see Clusters.

To enable automatic client failover, the client must be configured to allow non-zero reconnection attempts (as explained in Client Reconnection and Session Reattachment).

By default failover will only occur after at least one connection has been made to the live server. In other words, by default, failover will not occur if the client fails to make an initial connection to the live server - in this case it will simply retry connecting to the live server according to the reconnect-attempts property and fail after this number of attempts.

Failing over on the Initial Connection

Since the client does not learn about the full topology until after the first connection is made there is a window where it does not know about the backup. If a failure happens at this point the client can only try reconnecting to the original live server. To configure how many attempts the client will make you can set the URL parameter `initialConnectAttempts`. The default for this is `0`, that is try only once. Once the number of attempts has been made an exception will be thrown.

For examples of automatic failover with transacted and non-transacted JMS sessions, please see the examples chapter.

A Note on Server Replication

Apache ActiveMQ Artemis does not replicate full server state between live and backup servers. When the new session is automatically recreated on the backup it won't have any knowledge of messages already sent or acknowledged in that session. Any in-flight sends or acknowledgements at the time of failover might also be lost.

By replicating full server state, theoretically we could provide a 100% transparent seamless failover, which would avoid any lost messages or acknowledgements, however this comes at a great cost: replicating the full server state (including the queues, session, etc.). This would require replication of the entire server state machine; every operation on the live server would have to replicated on the replica server(s) in the exact same global order to ensure a consistent replica state. This is extremely hard to do in a performant and scalable way, especially when one considers that multiple threads are changing the live server state concurrently.

It is possible to provide full state machine replication using techniques such as *virtual synchrony*, but this does not scale well and effectively serializes all operations to a single thread, dramatically reducing concurrency.

Other techniques for multi-threaded active replication exist such as replicating lock states or replicating thread scheduling but this is very hard to achieve at a Java level.

Consequently it has decided it was not worth massively reducing performance and concurrency for the sake of 100% transparent failover. Even without 100% transparent failover, it is simple to guarantee *once and only once* delivery, even in the case of failure, by using a combination of duplicate detection and retrying of transactions. However this is not 100% transparent to the client code.

Handling Blocking Calls During Failover

If the client code is in a blocking call to the server, waiting for a response to continue its execution, when failover occurs, the new session will not have any knowledge of the call that was in progress. This call might otherwise hang for ever, waiting for a response that will never come.

To prevent this, Apache ActiveMQ Artemis will unblock any blocking calls that were in progress at the time of failover by making them throw a `javax.jms.JMSException` (if using JMS), or a `ActiveMQException` with error code `ActiveMQException.UNBLOCKED`. It is up to the client code to catch this exception and retry any operations if desired.

If the method being unblocked is a call to commit(), or prepare(), then the transaction will be automatically rolled back and Apache ActiveMQ Artemis will throw a `javax.jms.TransactionRolledBackException` (if using JMS), or a `ActiveMQException` with error code `ActiveMQException.TRANSACTION_ROLLED_BACK` if using the core API.

Handling Failover With Transactions

If the session is transactional and messages have already been sent or acknowledged in the current transaction, then the server cannot be sure that messages sent or acknowledgements have not been lost during the failover.

Consequently the transaction will be marked as rollback-only, and any subsequent attempt to commit it will throw a `javax.jms.TransactionRolledBackException` (if using JMS), or a `ActiveMQException` with error code `ActiveMQException.TRANSACTION_ROLLED_BACK` if using the core API.

> **Warning**
>
> The caveat to this rule is when XA is used either via JMS or through the core API. If 2 phase commit is used and prepare has already been called then rolling back could cause a `HeuristicMixedException`. Because of this the commit will throw a `XAException.XA_RETRY` exception. This informs the Transaction Manager that it should retry the commit at some later point in time, a side effect of this is that any non persistent messages will be lost. To avoid this use persistent messages when using XA. With acknowledgements this is not an issue since they are flushed to the server before prepare gets called.

It is up to the user to catch the exception, and perform any client side local rollback code as necessary. There is no need to manually rollback the session - it is already rolled back. The user can then just retry the transactional operations again on the same session.

Apache ActiveMQ Artemis ships with a fully functioning example demonstrating how to do this, please see the examples chapter.

If failover occurs when a commit call is being executed, the server, as previously described, will unblock the call to prevent a hang, since no response will come back. In this case it is not easy for the client to determine whether the transaction

commit was actually processed on the live server before failure occurred.

> **Note:**
>
> If XA is being used either via JMS or through the core API then an `XAException.XA_RETRY` is thrown. This is to inform Transaction Managers that a retry should occur at some point. At some later point in time the Transaction Manager will retry the commit. If the original commit has not occurred then it will still exist and be committed, if it does not exist then it is assumed to have been committed although the transaction manager may log a warning.

To remedy this, the client can simply enable duplicate detection (Duplicate Message Detection) in the transaction, and retry the transaction operations again after the call is unblocked. If the transaction had indeed been committed on the live server successfully before failover, then when the transaction is retried, duplicate detection will ensure that any durable messages resent in the transaction will be ignored on the server to prevent them getting sent more than once.

> **Note:**
>
> By catching the rollback exceptions and retrying, catching unblocked calls and enabling duplicate detection, once and only once delivery guarantees for messages can be provided in the case of failure, guaranteeing 100% no loss or duplication of messages.

Handling Failover With Non Transactional Sessions

If the session is non transactional, messages or acknowledgements can be lost in the event of failover.

If you wish to provide *once and only once* delivery guarantees for non transacted sessions too, enabled duplicate detection, and catch unblock exceptions as described in Handling Blocking Calls During Failover

Getting Notified of Connection Failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: `java.jms.ExceptionListener`. Please consult the JMS javadoc or any good JMS tutorial for more information on how to use this.

The Apache ActiveMQ Artemis core API also provides a similar feature in the form of the class `org.apache.activemq.artemis.core.client.SessionFailureListener`

Any ExceptionListener or SessionFailureListener instance will always be called by ActiveMQ Artemis on event of connection failure, **irrespective** of whether the connection was successfully failed over, reconnected or reattached, however you can find out if reconnect or reattach has happened by either the `failedOver` flag passed in on the `connectionFailed` on `SessionfailureListener` or by inspecting the error code on the `javax.jms.JMSException` which will be one of the following:

JMSException error codes

- `FAILOVER`

  Failover has occurred and we have successfully reattached or reconnected.

- `DISCONNECT`

  No failover has occurred and we are disconnected.

Application-Level Failover

In some cases you may not want automatic client failover, and prefer to handle any connection failure yourself, and code your own manually reconnection logic in your own failure handler. We define this as *application-level* failover, since the failover is handled at the user application level.

To implement application-level failover, if you're using JMS then you need to set an `ExceptionListener` class on the JMS connection. The `ExceptionListener` will be called by Apache ActiveMQ Artemis in the event that connection failure is detected. In your `ExceptionListener`, you would close your old JMS connections, potentially look up new connection factory instances from JNDI and creating new connections.

For a working example of application-level failover, please see the Application-Layer Failover Example.

If you are using the core API, then the procedure is very similar: you would set a `FailureListener` on the core `ClientSession` instances.

Graceful Server Shutdown

In certain circumstances an administrator might not want to disconnect all clients immediately when stopping the broker. In this situation the broker can be configured to shutdown *gracefully* using the `graceful-shutdown-enabled` boolean configuration parameter.

When the `graceful-shutdown-enabled` configuration parameter is `true` and the broker is shutdown it will first prevent any additional clients from connecting and then it will wait for any existing connections to be terminated by the client before completing the shutdown process. The default value is `false`.

Of course, it's possible a client could keep a connection to the broker indefinitely effectively preventing the broker from shutting down gracefully. To deal with this of situation the `graceful-shutdown-timeout` configuration parameter is available. This tells the broker (in milliseconds) how long to wait for all clients to disconnect before forcefully disconnecting the clients and proceeding with the shutdown process. The default value is `-1` which means the broker will wait indefinitely for clients to disconnect.

Libaio Native Libraries

Apache ActiveMQ Artemis distributes a native library, used as a bridge for its fast journal, between Apache ActiveMQ Artemis and Linux libaio.

`libaio` is a library, developed as part of the Linux kernel project. With `libaio` we submit writes to the operating system where they are processed asynchronously. Some time later the OS will call our code back when they have been processed.

We use this in our high performance journal if configured to do so, please see Persistence.

These are the native libraries distributed by Apache ActiveMQ Artemis:

- libartemis-native-64.so - x86 64 bits
- We distributed a 32-bit version until early 2017. While it's not available on the distribution any longer it should still be possible to compile to a 32-bit environment if needed.

When using libaio, Apache ActiveMQ Artemis will always try loading these files as long as they are on the library path

Runtime dependencies

If you just want to use the provided native binaries you need to install the required libaio dependency.

You can install libaio using the following steps as the root user:

Using yum, (e.g. on Fedora or Red Hat Enterprise Linux):

```
yum install libaio
```

Using aptitude, (e.g. on Ubuntu or Debian system):

```
apt-get install libaio1
```

Compiling the native libraries

In the case that you are using Linux on a platform other than x86_32 or x86_64 (for example Itanium 64 bits or IBM Power) you may need to compile the native library, since we do not distribute binaries for those platforms with the release.

Compilation dependencies

> **Note:**
>
> The native layer is only available on Linux. If you are in a platform other than Linux the native compilation will not work

These are the required linux packages to be installed for the compilation to work:

- gcc - C Compiler
- gcc-c++ or g++ - Extension to gcc with support for C++
- libtool - Tool for link editing native libraries
- libaio - library to disk asynchronous IO kernel functions
- libaio-dev - Compilation support for libaio
- cmake
- A full JDK installed with the environment variable JAVA_HOME set to its location

To perform this installation on RHEL or Fedora, you can simply type this at a command line:

```
sudo yum install libtool gcc-c++ gcc libaio libaio-devel cmake
```

Or on Debian systems:

```
sudo apt-get install libtool gcc-g++ gcc libaio libaio- cmake
```

> **Note:**
>
> You could find a slight variation of the package names depending on the version and Linux distribution. (for example gcc-c++ on Fedora versus g++ on Debian systems)

Invoking the compilation

In the source distribution or git clone, in the `artemis-native` directory, execute the shell script `compile-native.sh`. This script will invoke the proper commands to perform the native build.

If you want more information refer to the cmake web pages.

Thread management

This chapter describes how Apache ActiveMQ Artemis uses and pools threads and how you can manage them.

First we'll discuss how threads are managed and used on the server side, then we'll look at the client side.

Server-Side Thread Management

Each Apache ActiveMQ Artemis Server maintains a single thread pool for general use, and a scheduled thread pool for scheduled use. A Java scheduled thread pool cannot be configured to use a standard thread pool, otherwise we could use a single thread pool for both scheduled and non scheduled activity.

Apache ActiveMQ Artemis will, by default, cap its thread pool at three times the number of cores (or hyper-threads) as reported by `Runtime.getRuntime().availableProcessors()` for processing incoming packets. To override this value, you can set the number of threads by specifying the parameter `nioRemotingThreads` in the transport configuration. See the configuring transports for more information on this.

There are also a small number of other places where threads are used directly, we'll discuss each in turn.

Server Scheduled Thread Pool

The server scheduled thread pool is used for most activities on the server side that require running periodically or with delays. It maps internally to a `java.util.concurrent.ScheduledThreadPoolExecutor` instance.

The maximum number of thread used by this pool is configure in `broker.xml` with the `scheduled-thread-pool-max-size` parameter. The default value is `5` threads. A small number of threads is usually sufficient for this pool.

General Purpose Server Thread Pool

This general purpose thread pool is used for most asynchronous actions on the server side. It maps internally to a `java.util.concurrent.ThreadPoolExecutor` instance.

The maximum number of thread used by this pool is configure in `broker.xml` with the `thread-pool-max-size` parameter.

If a value of `-1` is used this signifies that the thread pool has no upper bound and new threads will be created on demand if there are not enough threads available to satisfy a request. If activity later subsides then threads are timed-out and closed.

If a value of `n` where `n` is a positive integer greater than zero is used this signifies that the thread pool is bounded. If more requests come in and there are no free threads in the pool and the pool is full then requests will block until a thread becomes available. It is recommended that a bounded thread pool is used with caution since it can lead to dead-lock situations if the upper bound is chosen to be too low.

The default value for `thread-pool-max-size` is `30`.

See the J2SE javadoc for more information on unbounded (cached), and bounded (fixed) thread pools.

Expiry Reaper Thread

A single thread is also used on the server side to scan for expired messages in queues. We cannot use either of the thread pools for this since this thread needs to run at its own configurable priority.

For more information on configuring the reaper, please see message expiry.

Asynchronous IO

Asynchronous IO has a thread pool for receiving and dispatching events out of the native layer. You will find it on a thread dump with the prefix ActiveMQ-AIO-poller-pool. Apache ActiveMQ Artemis uses one thread per opened file on the journal (there is usually one).

There is also a single thread used to invoke writes on libaio. We do that to avoid context switching on libaio that would cause performance issues. You will find this thread on a thread dump with the prefix ActiveMQ-AIO-writer-pool.

Client-Side Thread Management

On the client side, Apache ActiveMQ Artemis maintains a single, "global" static scheduled thread pool and a single, "global" static general thread pool for use by all clients using the same classloader in that JVM instance.

The static scheduled thread pool has a maximum size of `5` threads by default. This can be changed using the `scheduledThreadPoolMaxSize` URI parameter.

The general purpose thread pool has an unbounded maximum size. This is changed using the `threadPoolMaxSize` URL parameter.

If required Apache ActiveMQ Artemis can also be configured so that each `ClientSessionFactory` instance does not use these "global" static pools but instead maintains its own scheduled and general purpose pool. Any sessions created from that `ClientSessionFactory` will use those pools instead. This is configured using the `useGlobalPools` boolean URL parameter.

Embedded Web Server

Apache ActiveMQ Artemis embeds the Jetty web server. Its main purpose is to host the Management Console. However, it can also host other web applications like the REST interface or even Spring-based web applications (e.g. using Camel).

Configuration

The embedded Jetty instance is configured in `etc/bootstrap.xml` via the `web` element, e.g.:

```
<web bind="http://localhost:8161" path="web">
   <app url="activemq-branding" war="activemq-branding.war"/>
   <app url="artemis-plugin" war="artemis-plugin.war"/>
   <app url="console" war="console.war"/>
</web>
```

The `web` element has the following attributes:

- `bind` The protocol to use (i.e. `http` or `https`) as well as the host and port on which to listen.
- `path` The name of the subdirectory in which to find the web application archives (i.e. WAR files). This is a subdirectory of the broker's home or instance directory.
- `customizer` The name of customizer class to load.
- `clientAuth` Whether or not clients should present an SSL certificate when they connect. Only applicable when using `https`.
- `passwordCodec` The custom coded to use for unmasking the `keystorePassword` and `trustorePassword`.
- `keystorePath` The location on disk of the keystore. Only applicable when using `https`.
- `keystorePassword` The password to the keystore. Only applicable when using `https`. Can be masked using `ENC()` syntax or by defining `passwordCodec`. See more in the password masking chapter.
- `truststorePath` The location on disk fo the truststore. Only applicable when using `https`.
- `truststorePassword` The password to the truststore. Only applicable when using `https`. Can be masked using `ENC()` syntax or by defining `passwordCodec`. See more in the password masking chapter.
- `includedTLSProtocols` A comma seperated list of included TLS protocols, ie `"TLSv1,TLSv1.1,TLSv1.2"`. Only applicable when using `https`.
- `excludedTLSProtocols` A comma seperated list of excluded TLS protocols, ie `"TLSv1,TLSv1.1,TLSv1.2"`. Only applicable when using `https`.
- `includedCipherSuites` A comma seperated list of included cipher suites. Only applicable when using `https`.

- `excludedCipherSuites` A comma seperated list of excluded cipher suites. Only applicable when using `https`.

Each web application should be defined in an `app` element. The `app` element has the following attributes:

- `url` The context to use for the web application.
- `war` The name of the web application archive on disk.

It's also possible to configure HTTP/S request logging via the `request-log` element which has the following attributes:

- `filename` The full path of the request log. This attribute is required.
- `append` Whether or not to append to the existing log or truncate it. Boolean flag.
- `extended` Whether or not to use the extended request log format. Boolean flag.
- `logCookies` Logging of the request cookies. Boolean flag.
- `logTimeZone` The output file name of the request log.
- `filenameDateFormat` The log file name date format.
- `retainDays` The number of days before rotated log files are deleted.
- `ignorePaths` Request paths that will not be logged. Comma delimited list.
- `logDateFormat` The timestamp format string for request log entries.
- `logLocale` The locale of the request log.
- `logLatency` Logging of request processing time. Boolean flag.
- `logServer` Logging of the request hostname. Boolean flag.
- `preferProxiedForAddress` Whether the actual IP address of the connection or the IP address from the `X-Forwarded-For` header will be logged. Boolean flag.

These attributes are essentially passed straight through to the underlying `org.eclipse.jetty.server.NCSARequestLog` instance. Default values are based on this implementation.

Here is an example configuration:

```
<web bind="http://localhost:8161" path="web">
   <app url="activemq-branding" war="activemq-branding.war"/>
   <app url="artemis-plugin" war="artemis-plugin.war"/>
   <app url="console" war="console.war"/>
   <request-log filename="${artemis.instance}/log/http-access-yyyy_MM_dd.log" 
</web>
```

Logging

Apache ActiveMQ Artemis uses the JBoss Logging framework to do its logging and is configurable via the `logging.properties` file found in the configuration directories. This is configured by Default to log to both the console and to a file.

There are 9 loggers available which are as follows:

Logger	Description
org.jboss.logging	Logs any calls not handled by the Apache ActiveMQ Artemis loggers
org.apache.activemq.artemis.core.server	Logs the core server
org.apache.activemq.artemis.utils	Logs utility calls
org.apache.activemq.artemis.journal	Logs Journal calls
org.apache.activemq.artemis.jms	Logs JMS calls
org.apache.activemq.artemis.integration.bootstrap	Logs bootstrap calls
org.apache.activemq.audit.base	audit log. Disabled by default
org.apache.activemq.audit.resource	resource audit log. Disabled by default
org.apache.activemq.audit.message	message audit log. Disabled by default

Logging in a client or with an Embedded server

Firstly, if you want to enable logging on the client side you need to include the JBoss logging jars in your library. If you are using Maven the simplest way is to use the "all" client jar.

```
<dependency>
   <groupId>org.jboss.logmanager</groupId>
   <artifactId>jboss-logmanager</artifactId>
   <version>2.0.3.Final</version>
</dependency>
<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>activemq-core-client</artifactId>
   <version>2.5.0</version>
</dependency>
```

There are 2 properties you need to set when starting your java program, the first is to set the Log Manager to use the JBoss Log Manager, this is done by setting the `-Djava.util.logging.manager` property i.e. `-`

`Djava.util.logging.manager=org.jboss.logmanager.LogManager`

The second is to set the location of the logging.properties file to use, this is done via the `-Dlogging.configuration` for instance `-Dlogging.configuration=file:///home/user/projects/myProject/logging.properties`.

> **Note:**
>
> The `logging.configuration` system property needs to be valid URL

The following is a typical `logging.properties for a client`

```
# Root logger option
loggers=org.jboss.logging,org.apache.activemq.artemis.core.server,org.apache.a

# Root logger level
logger.level=INFO
# Apache ActiveMQ Artemis logger levels
logger.org.apache.activemq.artemis.core.server.level=INFO
logger.org.apache.activemq.artemis.utils.level=INFO
logger.org.apache.activemq.artemis.jms.level=DEBUG

# Root logger handlers
logger.handlers=FILE,CONSOLE

# Console handler configuration
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.level=FINE
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN

# File handler configuration
handler.FILE=org.jboss.logmanager.handlers.FileHandler
handler.FILE.level=FINE
handler.FILE.properties=autoFlush,fileName
handler.FILE.autoFlush=true
handler.FILE.fileName=activemq.log
handler.FILE.formatter=PATTERN

# Formatter pattern configuration
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%d{HH:mm:ss,SSS} %-5p [%c] %s%E%n
```

Configuring Audit Logging

There are 3 audit loggers that can be enabled separately and audit sifferent types of events, these are:

1. Base logger: This is a highly verbose logger that will capture most events that occur on JMX beans
2. Resource logger: This logs creation, updates and deletion of resources such as Addresses and Queues and also authentication, the main purpose of this is to track console activity and access to broker.
3. Message logger: this logs message production and consumption of messages and will have a potentially negatibve affect on performance

These are disabled by default in the logging.properties configuration file:

```
logger.org.apache.activemq.audit.base.level=ERROR
logger.org.apache.activemq.audit.base.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.base.useParentHandlers=false

logger.org.apache.activemq.audit.resource.level=ERROR
logger.org.apache.activemq.audit.resource.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.resource.useParentHandlers=false

logger.org.apache.activemq.audit.message.level=ERROR
logger.org.apache.activemq.audit.message.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.message.useParentHandlers=false
...
```

To enable the audit log change the above level to INFO, like this:

```
logger.org.apache.activemq.audit.base.level=INFO
logger.org.apache.activemq.audit.base.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.base.useParentHandlers=false
...
```

The 3 audit loggers can be disable/enabled separately.

Once enabled, all audit records are written into a separate log file (by default audit.log).

Logging the clients remote address

It is possible to configure the audit loggers to log the remote address of any calling clients either through normal clients or through the console and JMX. This is configured by enabling a specific login module in the login config file.

```
org.apache.activemq.artemis.spi.core.security.jaas.AuditLoginModule optional
       debug=false;
```

> **Note:**
>
> This needs to be the first entry in the login.config file
>
> **Note:**
>
> This login module does no authentication, it is used only to catch client information through which ever path a client takes

Use Custom Handlers

To use a different handler than the built-in ones, you either pick one from existing libraries or you implement it yourself. All handlers must extends the java.util.logging.Handler class.

To enable a custom handler you need to append it to the handlers list `logger.handlers` and add its configuration to the `logging.configuration`.

Last but not least, once you get your own handler please add it to the boot classpath otherwise the log manager will fail to load it!

REST Interface

The Apache ActiveMQ Artemis REST interface allows you to leverage the reliability and scalability features of Apache ActiveMQ Artemis over a simple REST/HTTP interface. The REST Interface implementation sits on top of an Apache ActiveMQ Artemis JMS API and as such exposes JMS like concepts via REST.

Using the REST interface Messages can be produced and consumed by sending and receiving simple HTTP messages that contain the content you want to push around. For instance, here's a simple example of posting an order to an order processing queue express as an HTTP message:

```
POST /queue/orders/create HTTP/1.1
Host: example.com
Content-Type: application/xml

<order>
   <name>Bill</name>
   <item>iPhone 4</item>
   <cost>$199.99</cost>
</order>
```

As you can see, we're just posting some arbitrary XML document to a URL. When the XML is received on the server is it processed within Apache ActiveMQ Artemis as a JMS message and distributed through core Apache ActiveMQ Artemis. Simple and easy. Consuming messages from a queue or topic looks very similar. We'll discuss the entire interface in detail later.

Goals of REST Interface

Why would you want to use Apache ActiveMQ Artemis's REST interface? What are the goals of the REST interface?

- Easily usable by machine-based (code) clients.
- Zero client footprint. We want Apache ActiveMQ Artemis to be usable by any client/programming language that has an adequate HTTP client library. You shouldn't have to download, install, and configure a special library to interact with Apache ActiveMQ Artemis.
- Lightweight interoperability. The HTTP protocol is strong enough to be our message exchange protocol. Since interactions are RESTful the HTTP uniform interface provides all the interoperability you need to communicate between different languages, platforms, and even messaging implementations that choose to implement the same RESTful interface as Apache ActiveMQ Artemis (i.e. the REST-* effort.)

- No envelope (e.g. SOAP) or feed (e.g. Atom) format requirements. You shouldn't have to learn, use, or parse a specific XML document format in order to send and receive messages through Apache ActiveMQ Artemis's REST interface.
- Leverage the reliability, scalability, and clustering features of Apache ActiveMQ Artemis on the back end without sacrificing the simplicity of a REST interface.

Installation and Configuration

Apache ActiveMQ Artemis's REST interface is installed as a Web archive (WAR). It depends on the RESTEasy project and can currently only run within a servlet container. Installing the Apache ActiveMQ Artemis REST interface is a little bit different depending whether Apache ActiveMQ Artemis is already embedded (e.g. you're deploying within Wildfly) or configured on the network somewhere, or you want the ActiveMQ Artemis REST WAR itself to startup and manage the Apache ActiveMQ Artemis server.

Installing Within Pre-configured Environment

This section should be used when you want to use the Apache ActiveMQ Artemis REST interface in an environment that already has Apache ActiveMQ Artemis installed and running. You must create a Web archive (.WAR) file with the following web.xml settings:

```
<web-app>
   <listener>
      <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootst
   </listener>

   <listener>
      <listener-class>org.apache.activemq.artemis.rest.integration.RestMessagi
   </listener>

   <filter>
      <filter-name>Rest-Messaging</filter-name>
      <filter-class>org.jboss.resteasy.plugins.server.servlet.FilterDispatcher
   </filter>

   <filter-mapping>
      <filter-name>Rest-Messaging</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

Within your WEB-INF/lib directory you must have the artemis-rest.jar file. If RESTEasy is not installed within your environment, you must add the RESTEasy jar files within the lib directory as well. Here's a sample Maven pom.xml that can build a WAR with the Apache ActiveMQ Artemis REST library.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.o

   <modelVersion>4.0.0</modelVersion>
   <groupId>org.somebody</groupId>
   <artifactId>artemis-rest</artifactId>
   <packaging>war</packaging>
   <name>My App</name>
   <version>1.0-SNAPSHOT</version>

   <dependencies>
      <dependency>
         <groupId>org.apache.activemq.rest</groupId>
         <artifactId>artemis-rest</artifactId>
         <version>$VERSION</version>
         <exclusions>
            <exclusion>
               <groupId>*</groupId>
               <artifactId>*</artifactId>
            </exclusion>
         </exclusions>
      </dependency>
   </dependencies>
</project>
```

The project structure should look this like:

```
|-- pom.xml
`-- src
   `-- main
       `-- webapp
           `-- WEB-INF
               `-- web.xml
```

It is worth noting that when deploying a WAR in a Java EE application server like Wildfly the URL for the resulting application will include the name of the WAR by default. For example, if you've constructed a WAR as described above named "activemq-rest.war" then clients will access it at, e.g. http://localhost:8080/activemq-rest/[queues|topics]. We'll see more about this later.

Bootstrapping ActiveMQ Artemis Along with REST

You can bootstrap Apache ActiveMQ Artemis within your WAR as well. To do this, you must have the Apache ActiveMQ Artemis core and JMS jars along with Netty, RESTEasy, and the Apache ActiveMQ Artemis REST jar within your WEB-INF/lib. You must also have a broker.xml config file within WEB-INF/classes. The examples that come with the Apache ActiveMQ Artemis REST distribution show how to do this. You must also add an additional listener to your web.xml file. Here's an example:

```
<web-app>
   <listener>
      <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootst
   </listener>

   <listener>
      <listener-class>org.apache.activemq.artemis.rest.integration.ActiveMQBoo
   </listener>

   <listener>
      <listener-class>org.apache.activemq.artemis.rest.integration.RestMessagi
   </listener>

   <filter>
      <filter-name>Rest-Messaging</filter-name>
      <filter-class>org.jboss.resteasy.plugins.server.servlet.FilterDispatcher
   </filter>

   <filter-mapping>
      <filter-name>Rest-Messaging</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

Here's a Maven pom.xml file for creating a WAR for this environment. Make sure your Apache ActiveMQ Artemis configuration file(s) are within the src/main/resources directory so that they are stuffed within the WAR's WEB-INF/classes directory!

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.o

   <modelVersion>4.0.0</modelVersion>
   <groupId>org.somebody</groupId>
   <artifactId>artemis-rest</artifactId>
   <packaging>war</packaging>
   <name>My App</name>
   <version>1.0-SNAPSHOT</version>

   <dependencies>
      <dependency>
         <groupId>org.apache.activemq.rest</groupId>
         <artifactId>artemis-rest</artifactId>
         <version>$VERSION</version>
      </dependency>
   </dependencies>
</project>
```

The project structure should look this like:

```
|-- pom.xml
`-- src
   `-- main
       `-- resources
           `-- broker.xml
       `-- webapp
           `-- WEB-INF
               `-- web.xml
```

REST Configuration

The Apache ActiveMQ Artemis REST implementation does have some configuration options. These are configured via XML configuration file that must be in your WEB-INF/classes directory. You must set the web.xml context-param `rest.messaging.config.file` to specify the name of the configuration file. Below is the format of the XML configuration file and the default values for each.

```
<rest-messaging>
   <server-in-vm-id>0</server-in-vm-id> <!-- deprecated, use "url" -->
   <use-link-headers>false</use-link-headers>
   <default-durable-send>false</default-durable-send>
   <dups-ok>true</dups-ok>
   <topic-push-store-dir>topic-push-store</topic-push-store-dir>
   <queue-push-store-dir>queue-push-store</queue-push-store-dir>
   <producer-time-to-live>0</producer-time-to-live>
   <producer-session-pool-size>10</producer-session-pool-size>
   <session-timeout-task-interval>1</session-timeout-task-interval>
   <consumer-session-timeout-seconds>300</consumer-session-timeout-seconds>
   <consumer-window-size>-1</consumer-window-size> <!-- deprecated, use "url"
   <url>vm://0</url>
</rest-messaging>
```

Let's give an explanation of each config option.

- `server-in-vm-id` . The Apache ActiveMQ Artemis REST implementation was formerly hard-coded to use the in-vm transport to communicate with the embedded Apache ActiveMQ Artemis instance. This is the id of the embedded instance. It is "0" by default. **Note:** this is deprecated in favor of `url` which can be used to connect to an arbitrary instance of Apache ActiveMQ Artemis (including one over the network).
- `use-link-headers` . By default, all links (URLs) are published using custom headers. You can instead have the Apache ActiveMQ Artemis REST implementation publish links using the Link Header specification instead if you desire.
- `default-durable-send` . Whether a posted message should be persisted by default if the user does not specify a durable query parameter.
- `dups-ok` . If this is true, no duplicate detection protocol will be enforced for message posting.
- `topic-push-store-dir` . This must be a relative or absolute file system path. This is a directory where push registrations for topics are stored. See Pushing Messages.
- `queue-push-store-dir` . This must be a relative or absolute file system path. This is a directory where push registrations for queues are stored. See Pushing Messages.
- `producer-session-pool-size` . The REST implementation pools Apache ActiveMQ Artemis sessions for sending messages. This is the size of the pool. That number of sessions will be created at startup time.

- `producer-time-to-live`. Default time to live for posted messages. Default is no ttl.
- `session-timeout-task-interval`. Pull consumers and pull subscriptions can time out. This is the interval the thread that checks for timed-out sessions will run at. A value of 1 means it will run every 1 second.
- `consumer-session-timeout-seconds`. Timeout in seconds for pull consumers/subscriptions that remain idle for that amount of time.
- `consumer-window-size`. For consumers, this config option is the same as the Apache ActiveMQ Artemis one of the same name. It will be used by sessions created by the Apache ActiveMQ Artemis REST implementation. This is deprecated in favor of `url` as it can be specified as a URL parameter.
- `url`. The URL the Apache ActiveMQ Artemis REST implementation should use to connect to the Apache ActiveMQ Artemis instance. Default to "vm://0".

Apache ActiveMQ Artemis REST Interface Basics

The Apache ActiveMQ Artemis REST interface publishes a variety of REST resources to perform various tasks on a queue or topic. Only the top-level queue and topic URI schemes are published to the outside world. You must discover all other resources to interact with by looking for and traversing links. You'll find published links within custom response headers and embedded in published XML representations. Let's look at how this works.

Queue and Topic Resources

To interact with a queue or topic you do a HEAD or GET request on the following relative URI pattern:

```
/queues/{name}
/topics/{name}
```

The base of the URI is the base URL of the WAR you deployed the Apache ActiveMQ Artemis REST server within as defined in the Installation and Configuration section of this document. Replace the `{name}` string within the above URI pattern with the name of the queue or topic you are interested in interacting with. Next, perform your HEAD or GET request on this URI. Here's what a request/response would look like.

```
HEAD /queues/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/bar/create
msg-create-with-id: http://example.com/queues/bar/create/{id}
msg-pull-consumers: http://example.com/queues/bar/pull-consumers
msg-push-consumers: http://example.com/queues/bar/push-consumers
```

> **Note:**
>
> You can use the "curl" utility to test this easily. Simply execute a command like this:
>
> curl --head http://example.com/queues/bar

The HEAD or GET response contains a number of custom response headers that are URLs to additional REST resources that allow you to interact with the queue or topic in different ways. It is important not to rely on the scheme of the URLs returned within these headers as they are an implementation detail. Treat them as opaque and query for them each and every time you initially interact (at boot time) with the server. If you treat all URLs as opaque then you will be isolated from implementation changes as the Apache ActiveMQ Artemis REST interface evolves over time.

Queue Resource Response Headers

Below is a list of response headers you should expect when interacting with a Queue resource.

- `msg-create`. This is a URL you POST messages to. The semantics of this link are described in Posting Messages.
- `msg-create-with-id`. This is a URL *template* you can use to POST messages. The semantics of this link are described in Posting Messages.
- `msg-pull-consumers`. This is a URL for creating consumers that will pull from a queue. The semantics of this link are described in Consuming Messages via Pull.
- `msg-push-consumers`. This is a URL for registering other URLs you want the Apache ActiveMQ Artemis REST server to push messages to. The semantics of this link are described in Pushing Messages.

Topic Resource Response Headers

Below is a list of response headers you should expect when interacting with a Topic resource.

- `msg-create`. This is a URL you POST messages to. The semantics of this link are described in Posting Messages.
- `msg-create-with-id`. This is a URL *template* you can use to POST messages. The semantics of this link are described in Posting Messages.

- `msg-pull-subscriptions`. This is a URL for creating subscribers that will pull from a topic. The semantics of this link are described in Consuming Messages via Pull.
- `msg-push-subscriptions`. This is a URL for registering other URLs you want the Apache ActiveMQ Artemis REST server to push messages to. The semantics of this link are described in Pushing Messages.

Posting Messages

This chapter discusses the protocol for posting messages to a queue or a topic. In Apache ActiveMQ Artemis REST Interface Basics, you saw that a queue or topic resource publishes variable custom headers that are links to other RESTful resources. The `msg-create` header is a URL you can post a message to. Messages are published to a queue or topic by sending a simple HTTP message to the URL published by the `msg-create` header. The HTTP message contains whatever content you want to publish to the Apache ActiveMQ Artemis destination. Here's an example scenario:

> **Note:**
>
> You can also post messages to the URL template found in `msg-create-with-id`, but this is a more advanced use-case involving duplicate detection that we will discuss later in this section.

1. Obtain the starting `msg-create` header from the queue or topic resource.

```
HEAD /queues/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/bar/create
msg-create-with-id: http://example.com/queues/bar/create/{id}
```

2. Do a POST to the URL contained in the `msg-create` header.

```
POST /queues/bar/create
Host: example.com
Content-Type: application/xml

<order>
   <name>Bill</name>
   <item>iPhone4</name>
   <cost>$199.99</cost>
</order>

--- Response ---
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/bar/create
```

> **Note:**
>
> You can use the "curl" utility to test this easily. Simply execute a command like this:
>
> ```
> curl --verbose --data "123" http://example.com/queues/bar/create
> ```

A successful response will return a 201 response code. Also notice that a `msg-create-next` response header is sent as well. You must use this URL to POST your next message.

3. POST your next message to the queue using the URL returned in the `msg-create-next` header.

```
POST /queues/bar/create
Host: example.com
Content-Type: application/xml

<order>
   <name>Monica</name>
   <item>iPad</item>
   <cost>$499.99</cost>
</order>

--- Response --
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/bar/create
```

Continue using the new `msg-create-next` header returned with each response.

> **Warning**
>
> It is *VERY IMPORTANT* that you never re-use returned `msg-create-next` headers to post new messages. If the `dups-ok` configuration property is set to `false` on the server then this URL will be uniquely generated for each message and used for duplicate detection. If you lose the URL within the `msg-create-next` header, then just go back to the queue or topic resource to get the `msg-create` URL again.

Duplicate Detection

Sometimes you might have network problems when posting new messages to a queue or topic. You may do a POST and never receive a response. Unfortunately, you don't know whether or not the server received the message and so a re-post of the message might cause duplicates to be posted to the queue or topic. By default, the Apache ActiveMQ Artemis REST interface is configured to accept and post duplicate messages. You can change this by turning on duplicate message detection by setting the `dups-ok` config option to `false` as described in Apache ActiveMQ Artemis REST Interface Basics. When you do this, the initial POST to the `msg-create` URL will redirect you, using the standard HTTP 307 redirection mechanism to a unique URL to POST to. All other interactions remain the same as discussed earlier. Here's an example:

1. Obtain the starting `msg-create` header from the queue or topic resource.

```
HEAD /queues/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/bar/create
msg-create-with-id: http://example.com/queues/bar/create/{id}
```

2. Do a POST to the URL contained in the `msg-create` header.

```
POST /queues/bar/create
Host: example.com
Content-Type: application/xml

<order>
   <name>Bill</name>
   <item>iPhone4</name>
   <cost>$199.99</cost>
</order>

--- Response ---
HTTP/1.1 307 Redirect
Location: http://example.com/queues/bar/create/13582001787372
```

   A successful response will return a 307 response code. This is standard HTTP protocol. It is telling you that you must re-POST to the URL contained within the `Location` header.

3. re-POST your message to the URL provided within the `Location` header.

```
POST /queues/bar/create/13582001787372
Host: example.com
Content-Type: application/xml

<order>
   <name>Bill</name>
   <item>iPhone4</name>
   <cost>$199.99</cost>
</order>

--- Response --
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/bar/create/13582001787373
```

   You should receive a 201 Created response. If there is a network failure, just re-POST to the Location header. For new messages, use the returned `msg-create-next` header returned with each response.

4. POST any new message to the returned `msg-create-next` header.

```
POST /queues/bar/create/13582001787373
Host: example.com
Content-Type: application/xml

<order>
   <name>Monica</name>
   <item>iPad</name>
   <cost>$499.99</cost>
</order>

--- Response --
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/bar/create/13582001787374
```

If there ever is a network problem, just repost to the URL provided in the `msg-create-next` header.

How can this work? As you can see, with each successful response, the Apache ActiveMQ Artemis REST server returns a uniquely generated URL within the msg-create-next header. This URL is dedicated to the next new message you want to post. Behind the scenes, the code extracts an identify from the URL and uses Apache ActiveMQ Artemis's duplicate detection mechanism by setting the `DUPLICATE_DETECTION_ID` property of the JMS message that is actually posted to the system.

If you happen to use the same ID more than once you'll see a message like this on the server:

```
WARN  [org.apache.activemq.artemis.core.server] (Thread-3 (Apache ActiveMQ Art
ServerMessage[messageID=20,priority=4, bodySize=1500,expiration=0, durable=tru
```

An alternative to this approach is to use the `msg-create-with-id` header. This is not an invokable URL, but a URL template. The idea is that the client provides the `DUPLICATE_DETECTION_ID` and creates its own `create-next` URL. The `msg-create-with-id` header looks like this (you've see it in previous examples, but we haven't used it):

```
msg-create-with-id: http://example.com/queues/bar/create/{id}
```

You see that it is a regular URL appended with an `{id}`. This `{id}` is a pattern matching substring. A client would generate its `DUPLICATE_DETECTION_ID` and replace `{id}` with that generated id, then POST to the new URL. The URL the client creates works exactly like a `create-next` URL described earlier. The response of this POST would also return a new `msg-create-next` header. The client can continue to generate its own DUPLICATE_DETECTION_ID, or use the new URL returned via the `msg-create-nex` t header.

The advantage of this approach is that the client does not have to repost the message. It also only has to come up with a unique `DUPLICATE_DETECTION_ID` once.

Persistent Messages

By default, posted messages are not durable and will not be persisted in Apache ActiveMQ Artemis's journal. You can create durable messages by modifying the default configuration as expressed in Chapter 2 so that all messages are persisted when sent. Alternatively, you can set a URL query parameter called `durable` to true when you post your messages to the URLs returned in the `msg-create`, `msg-create-with-id`, or `msg-create-next` headers. here's an example of that.

```
POST /queues/bar/create?durable=true
Host: example.com
Content-Type: application/xml

<order>
   <name>Bill</name>
   <item>iPhone4</item>
   <cost>$199.99</cost>
</order>
```

TTL, Expiration and Priority

You can set the time to live, expiration, and/or the priority of the message in the queue or topic by setting an additional query parameter. The `expiration` query parameter is a long specifying the time in milliseconds since epoch (a long date). The `ttl` query parameter is a time in milliseconds you want the message active. The `priority` is another query parameter with an integer value between 0 and 9 expressing the priority of the message. i.e.:

```
POST /queues/bar/create?expiration=30000&priority=3
Host: example.com
Content-Type: application/xml

<order>
   <name>Bill</name>
   <item>iPhone4</item>
   <cost>$199.99</cost>
</order>
```

Consuming Messages via Pull

There are two different ways to consume messages from a topic or queue. You can wait and have the messaging server push them to you, or you can continuously poll the server yourself to see if messages are available. This chapter discusses the latter. Consuming messages via a pull works almost identically for queues and topics with some minor, but important caveats. To start consuming you must create a consumer resource on the server that is dedicated to your client. Now, this pretty much breaks the stateless principle of REST, but after much prototyping, this is the best way to work most effectively with Apache ActiveMQ Artemis through a REST interface.

You create consumer resources by doing a simple POST to the URL published by the `msg-pull-consumers` response header if you are interacting with a queue, the `msg-pull-subscribers` response header if you're interacting with a topic. These headers are provided by the main queue or topic resource discussed in Apache ActiveMQ Artemis REST Interface Basics. Doing an empty POST to one of these URLs will create a consumer resource that follows an auto-acknowledge protocol and, if you are interacting with a topic, creates a temporarily subscription to the topic. If you want to use the acknowledgement protocol and/or create a durable subscription (topics only), then you must use the form parameters (`application/x-www-form-urlencoded`) described below.

- `autoAck` . A value of `true` or `false` can be given. This defaults to `true` if you do not pass this parameter.
- `durable` . A value of `true` or `false` can be given. This defaults to `false` if you do not pass this parameter. Only available on topics. This specifies whether you want a durable subscription or not. A durable subscription persists through server restart.
- `name` . This is the name of the durable subscription. If you do not provide this parameter, the name will be automatically generated by the server. Only usable on topics.
- `selector` . This is an optional JMS selector string. The Apache ActiveMQ Artemis REST interface adds HTTP headers to the JMS message for REST produced messages. HTTP headers are prefixed with "http_" and every '-' character is converted to a '$'.
- `idle-timeout` . For a topic subscription, idle time in milliseconds in which the consumer connections will be closed if idle.
- `delete-when-idle` . Boolean value, If true, a topic subscription will be deleted (even if it is durable) when the idle timeout is reached.

> **Note:**
>
> If you have multiple pull-consumers active at the same time on the same destination be aware that unless the `consumer-window-size` is 0 then one consumer might buffer messages while the other consumer gets none.

Auto-Acknowledge

This section focuses on the auto-acknowledge protocol for consuming messages via a pull. Here's a list of the response headers and URLs you'll be interested in.

- `msg-pull-consumers` . The URL of a factory resource for creating queue consumer resources. You will pull from these created resources.
- `msg-pull-subscriptions` . The URL of a factory resource for creating topic subscription resources. You will pull from the created resources.
- `msg-consume-next` . The URL you will pull the next message from. This is returned with every response.

- `msg-consumer`. This is a URL pointing back to the consumer or subscription resource created for the client.

Creating an Auto-Ack Consumer or Subscription

Here is an example of creating an auto-acknowledged queue pull consumer.

1. Find the pull-consumers URL by doing a HEAD or GET request to the base queue resource.

```
HEAD /queues/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/bar/create
msg-pull-consumers: http://example.com/queues/bar/pull-consumers
msg-push-consumers: http://example.com/queues/bar/push-consumers
```

2. Next do an empty POST to the URL returned in the `msg-pull-consumers` header.

```
POST /queues/bar/pull-consumers HTTP/1.1
Host: example.com

--- response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/bar/pull-consumers/auto-ack/333
msg-consume-next: http://example.com/queues/bar/pull-consumers/auto-ack/33
```

   The `Location` header points to the JMS consumer resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

Creating an auto-acknowledged consumer for a topic is pretty much the same. Here's an example of creating a durable auto-acknowledged topic pull subscription.

1. Find the `pull-subscriptions` URL by doing a HEAD or GET request to the base topic resource

```
HEAD /topics/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/topics/foo/create
msg-pull-subscriptions: http://example.com/topics/foo/pull-subscriptions
msg-push-subscriptions: http://example.com/topics/foo/push-subscriptions
```

2. Next do a POST to the URL returned in the `msg-pull-subscriptions` header passing in a `true` value for the `durable` form parameter.

```
POST /topics/foo/pull-subscriptions HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

durable=true

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/foo/pull-subscriptions/auto-ack/222
msg-consume-next:
http://example.com/topics/foo/pull-subscriptions/auto-ack/222/consume-next
```

The `Location` header points to the JMS subscription resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

Consuming Messages

After you have created a consumer resource, you are ready to start pulling messages from the server. Notice that when you created the consumer for either the queue or topic, the response contained a `msg-consume-next` response header. POST to the URL contained within this header to consume the next message in the queue or topic subscription. A successful POST causes the server to extract a message from the queue or topic subscription, acknowledge it, and return it to the consuming client. If there are no messages in the queue or topic subscription, a 503 (Service Unavailable) HTTP code is returned.

> **Warning**
>
> For both successful and unsuccessful posts to the msg-consume-next URL, the response will contain a new msg-consume-next header. You must ALWAYS use this new URL returned within the new msg-consume-next header to consume new messages.

Here's an example of pulling multiple messages from the consumer resource.

1. Do a POST on the msg-consume-next URL that was returned with the consumer or subscription resource discussed earlier.

   ```
   POST /queues/bar/pull-consumers/consume-next-1
   Host: example.com

   --- Response ---
   HTTP/1.1 200 Ok
   Content-Type: application/xml
   msg-consume-next: http://example.com/queues/bar/pull-consumers/333/consume
   msg-consumer: http://example.com/queues/bar/pull-consumers/333

   <order>...</order>
   ```

   The POST returns the message consumed from the queue. It also returns a new msg-consume-next link. Use this new link to get the next message. Notice also a msg-consumer response header is returned. This is a URL that points back to the consumer or subscription resource. You will need that to clean up your connection after you are finished using the queue or topic.

2. The POST returns the message consumed from the queue. It also returns a new msg-consume-next link. Use this new link to get the next message.

```
POST /queues/bar/pull-consumers/consume-next-2
Host: example.com

--- Response ---
Http/1.1 503 Service Unavailable
Retry-After: 5
msg-consume-next: http://example.com/queues/bar/pull-consumers/333/consume
```

   In this case, there are no messages in the queue, so we get a 503 response back. As per the HTTP 1.1 spec, a 503 response may return a Retry-After head specifying the time in seconds that you should retry a post. Also notice, that another new msg-consume-next URL is present. Although it probably is the same URL you used last post, get in the habit of using URLs returned in response headers as future versions of Apache ActiveMQ Artemis REST might be redirecting you or adding additional data to the URL after timeouts like this.

3. POST to the URL within the last `msg-consume-next` to get the next message.

```
POST /queues/bar/pull-consumers/consume-next-2
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-consume-next: http://example.com/queues/bar/pull-consumers/333/consume

<order>...</order>
```

Recovering From Network Failures

If you experience a network failure and do not know if your post to a msg-consume-next URL was successful or not, just re-do your POST. A POST to a msg-consume-next URL is idempotent, meaning that it will return the same result if you execute on any one msg-consume-next URL more than once. Behind the scenes, the consumer resource caches the last consumed message so that if there is a message failure and you do a re-post, the cached last message will be returned (along with a new msg-consume-next URL). This is the reason why the protocol always requires you to use the next new msg-consume-next URL returned with each response. Information about what state the client is in is embedded within the actual URL.

Recovering From Client or Server Crashes

If the server crashes and you do a POST to the msg-consume-next URL, the server will return a 412 (Preconditions Failed) response code. This is telling you that the URL you are using is out of sync with the server. The response will contain a new msg-consume-next header to invoke on.

If the client crashes there are multiple ways you can recover. If you have remembered the last msg-consume-next link, you can just re-POST to it. If you have remembered the consumer resource URL, you can do a GET or HEAD request to obtain a new msg-consume-next URL. If you have created a topic subscription using the name parameter discussed earlier, you can re-create the consumer. Re-creation will return a msg-consume-next URL you can use. If you cannot do any of these things, you will have to create a new consumer.

The problem with the auto-acknowledge protocol is that if the client or server crashes, it is possible for you to skip messages. The scenario would happen if the server crashes after auto-acknowledging a message and before the client receives the message. If you want more reliable messaging, then you must use the acknowledgement protocol.

Manual Acknowledgement

The manual acknowledgement protocol is similar to the auto-ack protocol except there is an additional round trip to the server to tell it that you have received the message and that the server can internally ack the message. Here is a list of the response headers you will be interested in.

- `msg-pull-consumers` . The URL of a factory resource for creating queue consumer resources. You will pull from these created resources
- `msg-pull-subscriptions` . The URL of a factory resource for creating topic subscription resources. You will pull from the created resources.
- `msg-acknowledge-next` . URL used to obtain the next message in the queue or topic subscription. It does not acknowledge the message though.
- `msg-acknowledgement` . URL used to acknowledge a message.
- `msg-consumer` . This is a URL pointing back to the consumer or subscription resource created for the client.

Creating manually-acknowledged consumers or subscriptions

Here is an example of creating an auto-acknowledged queue pull consumer.

1. Find the pull-consumers URL by doing a HEAD or GET request to the base queue resource.

```
HEAD /queues/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/bar/create
msg-pull-consumers: http://example.com/queues/bar/pull-consumers
msg-push-consumers: http://example.com/queues/bar/push-consumers
```

2. Next do a POST to the URL returned in the `msg-pull-consumers` header passing in a `false` value to the `autoAck` form parameter .

```
POST /queues/bar/pull-consumers HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

autoAck=false

--- response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/bar/pull-consumers/acknowledged/333
msg-acknowledge-next: http://example.com/queues/bar/pull-consumers/acknowl
```

The `Location` header points to the JMS consumer resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

Creating a manually-acknowledged consumer for a topic is pretty much the same. Here's an example of creating a durable manually-acknowledged topic pull subscription.

1. Find the `pull-subscriptions` URL by doing a HEAD or GET request to the base topic resource

```
HEAD /topics/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/topics/foo/create
msg-pull-subscriptions: http://example.com/topics/foo/pull-subscriptions
msg-push-subscriptions: http://example.com/topics/foo/push-subscriptions
```

2. Next do a POST to the URL returned in the `msg-pull-subscriptions` header passing in a `true` value for the `durable` form parameter and a `false` value to the `autoAck` form parameter.

```
POST /topics/foo/pull-subscriptions HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

durable=true&autoAck=false

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/foo/pull-subscriptions/acknowledged/22
msg-acknowledge-next:
http://example.com/topics/foo/pull-subscriptions/acknowledged/222/consume-
```

The `Location` header points to the JMS subscription resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

Consuming and Acknowledging a Message

After you have created a consumer resource, you are ready to start pulling messages from the server. Notice that when you created the consumer for either the queue or topic, the response contained a `msg-acknowledge-next` response

header. POST to the URL contained within this header to consume the next message in the queue or topic subscription. If there are no messages in the queue or topic subscription, a 503 (Service Unavailable) HTTP code is returned. A successful POST causes the server to extract a message from the queue or topic subscription and return it to the consuming client. It does not acknowledge the message though. The response will contain the `acknowledgement` header which you will use to acknowledge the message.

Here's an example of pulling multiple messages from the consumer resource.

1. Do a POST on the msg-acknowledge-next URL that was returned with the consumer or subscription resource discussed earlier.

```
POST /queues/bar/pull-consumers/consume-next-1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-acknowledgement:
http://example.com/queues/bar/pull-consumers/333/acknowledgement/2
msg-consumer: http://example.com/queues/bar/pull-consumers/333

<order>...</order>
```

   The POST returns the message consumed from the queue. It also returns a `msg-acknowledgemen` t link. You will use this new link to acknowledge the message. Notice also a `msg-consumer` response header is returned. This is a URL that points back to the consumer or subscription resource. You will need that to clean up your connection after you are finished using the queue or topic.

2. Acknowledge or unacknowledge the message by doing a POST to the URL contained in the `msg-acknowledgement` header. You must pass an `acknowledge` form parameter set to `true` or `false` depending on whether you want to acknowledge or unacknowledge the message on the server.

```
POST /queues/bar/pull-consumers/acknowledgement/2
Host: example.com
Content-Type: application/x-www-form-urlencoded

acknowledge=true

--- Response ---
Http/1.1 204 Ok
msg-acknowledge-next:
http://example.com/queues/bar/pull-consumers/333/acknowledge-next-2
```

   Whether you acknowledge or unacknowledge the message, the response will contain a new msg-acknowledge-next header that you must use to obtain the next message.

Recovering From Network Failures

If you experience a network failure and do not know if your post to a `msg-acknowledge-next` or `msg-acknowledgement` URL was successful or not, just re-do your POST. A POST to one of these URLs is idempotent, meaning that it will return the same result if you re-post. Behind the scenes, the consumer resource keeps track of its current state. If the last action was a call to `msg-acknowledge-next`, it will have the last message cached, so that if a re-post is done, it will return the message again. Same goes with re-posting to `msg-acknowledgement`. The server remembers its last state and will return the same results. If you look at the URLs you'll see that they contain information about the expected current state of the server. This is how the server knows what the client is expecting.

Recovering From Client or Server Crashes

If the server crashes and while you are doing a POST to the `msg-acknowledge-next` URL, just re-post. Everything should reconnect all right. On the other hand, if the server crashes while you are doing a POST to `msg-acknowledgement`, the server will return a 412 (Preconditions Failed) response code. This is telling you that the URL you are using is out of sync with the server and the message you are acknowledging was probably re-enqueued. The response will contain a new `msg-acknowledge-next` header to invoke on.

As long as you have "bookmarked" the consumer resource URL (returned from `Location` header on a create, or the `msg-consumer` header), you can recover from client crashes by doing a GET or HEAD request on the consumer resource to obtain what state you are in. If the consumer resource is expecting you to acknowledge a message, it will return a `msg-acknowledgement` header in the response. If the consumer resource is expecting you to pull for the next message, the `msg-acknowledge-next` header will be in the response. With manual acknowledgement you are pretty much guaranteed to avoid skipped messages. For topic subscriptions that were created with a name parameter, you do not have to "bookmark" the returned URL. Instead, you can re-create the consumer resource with the same exact name. The response will contain the same information as if you did a GET or HEAD request on the consumer resource.

Blocking Pulls with Accept-Wait

Unless your queue or topic has a high rate of message flowing though it, if you use the pull protocol, you're going to be receiving a lot of 503 responses as you continuously pull the server for new messages. To alleviate this problem, the Apache ActiveMQ Artemis REST interface provides the `Accept-Wait` header. This is a generic HTTP request header that is a hint to the server for how long the client is willing to wait for a response from the server. The value of this header is the time in seconds the client is willing to block for. You would send this request header with your pull requests. Here's an example:

```
POST /queues/bar/pull-consumers/consume-next-2
Host: example.com
Accept-Wait: 30

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-consume-next: http://example.com/queues/bar/pull-consumers/333/consume-nex

<order>...</order>
```

In this example, we're posting to a msg-consume-next URL and telling the server that we would be willing to block for 30 seconds.

Clean Up Your Consumers!

When the client is done with its consumer or topic subscription it should do an HTTP DELETE call on the consumer URL passed back from the Location header or the msg-consumer response header. The server will time out a consumer with the value of `consumer-session-timeout-seconds` configured from REST configuration, so you don't have to clean up if you don't want to, but if you are a good kid, you will clean up your messes. A consumer timeout for durable subscriptions will not delete the underlying durable JMS subscription though, only the server-side consumer resource (and underlying JMS session).

Pushing Messages

You can configure the Apache ActiveMQ Artemis REST server to push messages to a registered URL either remotely through the REST interface, or by creating a pre-configured XML file for the Apache ActiveMQ Artemis REST server to load at boot time.

The Queue Push Subscription XML

Creating a push consumer for a queue first involves creating a very simple XML document. This document tells the server if the push subscription should survive server reboots (is it durable). It must provide a URL to ship the forwarded message to. Finally, you have to provide authentication information if the final endpoint requires authentication. Here's a simple example:

```
<push-registration>
   <durable>false</durable>
   <selector><![CDATA[
   SomeAttribute > 1
   ]]>
   </selector>
   <link rel="push" href="http://somewhere.com" type="application/json" method
   <maxRetries>5</maxRetries>
   <retryWaitMillis>1000</retryWaitMillis>
   <disableOnFailure>true</disableOnFailure>
</push-registration>
```

The `durable` element specifies whether the registration should be saved to disk so that if there is a server restart, the push subscription will still work. This element is not required. If left out it defaults to `false`. If durable is set to true, an XML file for the push subscription will be created within the directory specified by the `queue-push-store-dir` config variable defined in Chapter 2 (`topic-push-store-dir` for topics).

The `selector` element is optional and defines a JMS message selector. You should enclose it within CDATA blocks as some of the selector characters are illegal XML.

The `maxRetries` element specifies how many times a the server will try to push a message to a URL if there is a connection failure.

The `retryWaitMillis` element specifies how long to wait before performing a retry.

The `disableOnFailure` element, if set to true, will disable the registration if all retries have failed. It will not disable the connection on non-connection-failure issues (like a bad request for instance). In these cases, the dead letter queue logic of Apache ActiveMQ Artemis will take over.

The `link` element specifies the basis of the interaction. The `href` attribute contains the URL you want to interact with. It is the only required attribute. The `type` attribute specifies the content-type of what the push URL is expecting. The `method` attribute defines what HTTP method the server will use when it sends the message to the server. If it is not provided it defaults to POST. The `rel` attribute is very important and the value of it triggers different behavior. Here's the values a rel attribute can have:

- `destination`. The href URL is assumed to be a queue or topic resource of another Apache ActiveMQ Artemis REST server. The push registration will initially do a HEAD request to this URL to obtain a msg-create-with-id header. It will use this header to push new messages to the Apache ActiveMQ Artemis REST endpoint reliably. Here's an example:

```
<push-registration>
   <link rel="destination" href="http://somewhere.com/queues/foo"/>
</push-registration>
```

- `template`. In this case, the server is expecting the link element's href attribute to be a URL expression. The URL expression must have one and only one URL parameter within it. The server will use a unique value to create the endpoint URL. Here's an example:

```
<push-registration>
   <link rel="template" href="http://somewhere.com/resources/{id}/messages
</push-registration>
```

  In this example, the {id} sub-string is the one and only one URL parameter.

- `user defined`. If the rel attributes is not destination or template (or is empty or missing), then the server will send an HTTP message to the href URL using the HTTP method defined in the method attribute. Here's an example:

```
<push-registration>
   <link href="http://somewhere.com" type="application/json" method="PUT"/
</push-registration>
```

The Topic Push Subscription XML

The push XML for a topic is the same except the root element is push-topic-registration. (Also remember the `selector` element is optional). The rest of the document is the same. Here's an example of a template registration:

```
<push-topic-registration>
   <durable>true</durable>
   <selector><![CDATA[
   SomeAttribute > 1
   ]]>
   </selector>
   <link rel="template" href="http://somewhere.com/resources/{id}/messages" me
</push-topic registration>
```

Creating a Push Subscription at Runtime

Creating a push subscription at runtime involves getting the factory resource URL from the msg-push-consumers header, if the destination is a queue, or msg-push-subscriptions header, if the destination is a topic. Here's an example of creating a push registration for a queue:

1. First do a HEAD request to the queue resource:

```
HEAD /queues/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/bar/create
msg-pull-consumers: http://example.com/queues/bar/pull-consumers
msg-push-consumers: http://example.com/queues/bar/push-consumers
```

2. Next POST your subscription XML to the URL returned from msg-push-consumers header

```
POST /queues/bar/push-consumers
Host: example.com
Content-Type: application/xml

<push-registration>
   <link rel="destination" href="http://somewhere.com/queues/foo"/>
</push-registration>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/bar/push-consumers/1-333-1212
```

The Location header contains the URL for the created resource. If you want to unregister this, then do a HTTP DELETE on this URL.

Here's an example of creating a push registration for a topic:

1. First do a HEAD request to the topic resource:

```
HEAD /topics/bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/topics/bar/create
msg-pull-subscriptions: http://example.com/topics/bar/pull-subscriptions
msg-push-subscriptions: http://example.com/topics/bar/push-subscriptions
```

2. Next POST your subscription XML to the URL returned from msg-push-subscriptions header

```
POST /topics/bar/push-subscriptions
Host: example.com
Content-Type: application/xml

<push-registration>
   <link rel="template" href="http://somewhere.com/resources/{id}"/>
</push-registration>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/bar/push-subscriptions/1-333-1212
```

The Location header contains the URL for the created resource. If you want to unregister this, then do a HTTP DELETE on this URL.

Creating a Push Subscription by Hand

You can create a push XML file yourself if you do not want to go through the REST interface to create a push subscription. There is some additional information you need to provide though. First, in the root element, you must define a unique id attribute. You must also define a destination element to specify the queue you should register a consumer with. For a topic, the destination element is the name of the subscription that will be created. For a topic, you must also specify the topic name within the topic element.

Here's an example of a hand-created queue registration. This file must go in the directory specified by the queue-push-store-dir config variable defined in Chapter 2:

```
<push-registration id="111">
   <destination>bar</destination>
   <durable>true</durable>
   <link rel="template" href="http://somewhere.com/resources/{id}/messages" me
</push-registration>
```

Here's an example of a hand-created topic registration. This file must go in the directory specified by the topic-push-store-dir config variable defined in Chapter 2:

```
<push-topic-registration id="112">
   <destination>my-subscription-1</destination
   <durable>true</durable>
   <link rel="template" href="http://somewhere.com/resources/{id}/messages" me
   <topic>foo</topic>
</push-topic-registration>
```

Pushing to Authenticated Servers

Push subscriptions only support BASIC and DIGEST authentication out of the box. Here is an example of adding BASIC authentication:

```
<push-topic-registration>
   <durable>true</durable>
   <link rel="template" href="http://somewhere.com/resources/{id}/messages" me
   <authentication>
      <basic-auth>
         <username>guest</username>
         <password>geheim</password>
      </basic-auth>
   </authentication>
</push-topic-registration>
```

For DIGEST, just replace basic-auth with digest-auth.

For other authentication mechanisms, you can register headers you want transmitted with each request. Use the header element with the name attribute representing the name of the header. Here's what custom headers might look like:

```
<push-topic-registration>
   <durable>true</durable>
   <link rel="template" href="http://somewhere.com/resources/{id}/messages" me
   <header name="secret-header">jfdiwe3321</header>
</push-topic-registration>
```

Creating Destinations

You can create a durable queue or topic through the REST interface. Currently you cannot create a temporary queue or topic. To create a queue you do a POST to the relative URL /queues with an XML representation of the queue. For example:

```
POST /queues
Host: example.com
Content-Type: application/activemq.xml

<queue name="testQueue">
   <durable>true</durable>
</queue>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/testQueue
```

Notice that the Content-Type is application/activemq.xml.

Here's what creating a topic would look like:

```
POST /topics
Host: example.com
Content-Type: application/activemq.xml

<topic name="testTopic">
</topic>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/testTopic
```

Securing the Apache ActiveMQ Artemis REST Interface

Within Wildfly Application server

Securing the Apache ActiveMQ Artemis REST interface is very simple with the Wildfly Application Server. You turn on authentication for all URLs within your WAR's web.xml, and let the user Principal to propagate to Apache ActiveMQ Artemis. This only works if you are using the JAASSecurityManager with Apache ActiveMQ Artemis. See the Apache ActiveMQ Artemis documentation for more details.

Security in other environments

To secure the Apache ActiveMQ Artemis REST interface in other environments you must role your own security by specifying security constraints with your web.xml for every path of every queue and topic you have deployed. Here is a list of URI patterns:

Post	Description
/queues	secure the POST operation to secure queue creation
/queues/{queue-name}/create/	secure this URL pattern for producing messages.
/queues/{queue-name}/pull-consumers/	secure this URL pattern for pushing messages.
/queues/{queue-name}/push-consumers/	secure the POST operation to secure topic creation
/topics	secure the POST operation to secure topic creation
/topics/{topic-name}	secure the GET HEAD operation to getting information about the topic.
/topics/{topic-name}/create/	secure this URL pattern for producing messages
/topics/{topic-name}/pull-subscriptions/	secure this URL pattern for pulling messages
/topics/{topic-name}/push-subscriptions/	secure this URL pattern for pushing messages

Mixing JMS and REST

The Apache ActiveMQ Artemis REST interface supports mixing JMS and REST producers and consumers. You can send an ObjectMessage through a JMS Producer, and have a REST client consume it. You can have a REST client POST a message to a topic and have a JMS Consumer receive it. Some simple transformations are supported if you have the correct RESTEasy providers installed.

JMS Producers - REST Consumers

If you have a JMS producer, the Apache ActiveMQ Artemis REST interface only supports ObjectMessage type. If the JMS producer is aware that there may be REST consumers, it should set a JMS property to specify what Content-Type the Java object should be translated into by REST clients. The Apache ActiveMQ Artemis REST server will use RESTEasy content handlers (MessageBodyReader/Writers) to transform the Java object to the type desired. Here's an example of a JMS producer setting the content type of the message.

```
ObjectMessage message = session.createObjectMessage();
message.setStringProperty(org.apache.activemq.rest.HttpHeaderProperty.CONTENT_
```

If the JMS producer does not set the content-type, then this information must be obtained from the REST consumer. If it is a pull consumer, then the REST client should send an Accept header with the desired media types it wants to convert

the Java object into. If the REST client is a push registration, then the type attribute of the link element of the push registration should be set to the desired type.

REST Producers - JMS Consumers

If you have a REST client producing messages and a JMS consumer, Apache ActiveMQ Artemis REST has a simple helper class for you to transform the HTTP body to a Java object. Here's some example code:

```
public void onMessage(Message message) {
   MyType obj = org.apache.activemq.rest.Jms.getEntity(message, MyType.class);
}
```

The way the `getEntity()` method works is that if the message is an ObjectMessage, it will try to extract the desired type from it like any other JMS message. If a REST producer sent the message, then the method uses RESTEasy to convert the HTTP body to the Java object you want. See the Javadoc of this class for more helper methods.

Embedding Apache ActiveMQ Artemis

Apache ActiveMQ Artemis is designed as set of simple Plain Old Java Objects (POJOs). This means Apache ActiveMQ Artemis can be instantiated and run in any dependency injection framework such as Spring or Google Guice. It also means that if you have an application that could use messaging functionality internally then it can *directly instantiate* Apache ActiveMQ Artemis clients and servers in its own application code to perform that functionality. We call this *embedding* Apache ActiveMQ Artemis.

Examples of applications that might want to do this include any application that needs very high performance, transactional, persistent messaging but doesn't want the hassle of writing it all from scratch.

Embedding Apache ActiveMQ Artemis can be done in very few easy steps - supply a `broker.xml` on the classpath or instantiate the configuration object, instantiate the server, start it, and you have a Apache ActiveMQ Artemis running in your JVM. It's as simple and easy as that.

Embedding with XML configuration

The simplest way to embed Apache ActiveMQ Artemis is to use the embedded wrapper class and configure Apache ActiveMQ Artemis through `broker.xml`.

Here's a simple example `broker.xml`:

```
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="ur
   <core xmlns="urn:activemq:core">

      <persistence-enabled>false</persistence-enabled>

      <security-enabled>false</security-enabled>

      <acceptors>
         <acceptor name="in-vm">vm://0</acceptor>
      </acceptors>
   </core>
</configuration>
```

```
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;

...

EmbeddedActiveMQ embedded = new EmbeddedActiveMQ();
embedded.start();

ServerLocator serverLocator =  ActiveMQClient.createServerLocator("vm://0");
ClientSessionFactory factory =  serverLocator.createSessionFactory();
ClientSession session = factory.createSession();

session.createQueue(new QueueConfiguration("example"));

ClientProducer producer = session.createProducer("example");
ClientMessage message = session.createMessage(true);
message.getBody().writeString("Hello");
producer.send(message);

session.start();
ClientConsumer consumer = session.createConsumer("example");
ClientMessage msgReceived = consumer.receive();
System.out.println("message = " + msgReceived.getBody().readString());
session.close();
```

The `EmbeddedActiveMQ` class has a few additional setter methods that allow you to specify a different config file name as well as other properties. See the javadocs for this class for more details.

Embedding with programmatic configuration

You can follow this step-by-step guide to programmatically embed a broker instance.

Create the `Configuration` object. This contains configuration information for an Apache ActiveMQ Artemis instance. The setter methods of this class allow you to programmatically set configuration options as described in the Server Configuration section.

The acceptors are configured through `Configuration`. Just add the acceptor URL the same way you would through the main configuration file.

```
import org.apache.activemq.artemis.core.config.Configuration;
import org.apache.activemq.artemis.core.config.impl.ConfigurationImpl;

...

Configuration config = new ConfigurationImpl();

config.addAcceptorConfiguration("in-vm", "vm://0");
config.addAcceptorConfiguration("tcp", "tcp://127.0.0.1:61616");
```

You need to instantiate an instance of `org.apache.activemq.artemis.api.core.server.embedded.EmbeddedActiveMQ` and add the configuration object to it.

```
import org.apache.activemq.artemis.api.core.server.ActiveMQ;
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;

...

EmbeddedActiveMQ server = new EmbeddedActiveMQ();
server.setConfiguration(config);

server.start();
```

You also have the option of instantiating `ActiveMQServerImpl` directly:

```
ActiveMQServer server = new ActiveMQServerImpl(config);
server.start();
```

Dependency Frameworks

You may also choose to use a dependency injection framework such as The Spring Framework. See Spring Integration for more details on Spring and Apache ActiveMQ Artemis.

Apache ActiveMQ Artemis standalone uses Airline to bootstrap.

Artemis on Apache Karaf

Apache ActiveMQ Artemis is OSGi ready. Below you can find instruction on how to install and configure broker on Apache Karaf OSGi container.

Installation

Apache ActiveMQ Artemis provides features that makes it easy to install the broker on Apache Karaf (4.x or later). First you need to define the feature URL, like

```
karaf@root()> feature:repo-add mvn:org.apache.activemq/artemis-features/1.3.0-
```

This will add Artemis related features

```
karaf@root()> feature:list | grep artemis
artemis                       | 1.3.0.SNAPSHOT   |          | Uninstalled | ar
netty-core                    | 4.0.32.Final     |          | Uninstalled | ar
artemis-core                  | 1.3.0.SNAPSHOT   |          | Uninstalled | ar
artemis-amqp                  | 1.3.0.SNAPSHOT   |          | Uninstalled | ar
artemis-stomp                 | 1.3.0.SNAPSHOT   |          | Uninstalled | ar
artemis-mqtt                  | 1.3.0.SNAPSHOT   |          | Uninstalled | ar
artemis-hornetq               | 1.3.0.SNAPSHOT   |          | Uninstalled | ar
```

Feature named `artemis` contains full broker installation, so running

```
feature:install artemis
```

will install and run the broker.

Configuration

The broker is installed as `org.apache.activemq.artemis` OSGi component, so it's configured through `${KARAF_BASE}/etc/org.apache.activemq.artemis.cfg` file. An example of the file looks like

```
config=file:etc/artemis.xml
name=local
domain=karaf
rolePrincipalClass=org.apache.karaf.jaas.boot.principal.RolePrincipal
```

Name	Description	Default value
config	Location of the configuration file	${KARAF_BASE}/etc/artemis.xml
name	Name of the broker	local
domain	JAAS domain to use for security	karaf
rolePrincipalClass	Class name used for role authorization purposes	org.apache.karaf.jaas.boot.principal.R

The default broker configuration file is located in `${KARAF_BASE}/etc/artemis.xml`

Apache Tomcat Support

Resource Context Client Configuration

Apache ActiveMQ Artemis provides support for configuring the client, in the tomcat resource context.xml of Tomcat container.

This is very similar to the way this is done in ActiveMQ 5.x so anyone migrating should find this familiar. Please note though the connection url and properties that can be set for ActiveMQ Artemis are different please see Migration Documentation

Example of Connection Factory

```
<Context>
    ...
  <Resource name="jms/ConnectionFactory" auth="Container" type="org.apache.act
        factory="org.apache.activemq.artemis.jndi.JNDIReferenceFactory" broker
    ...
</Context>
`
```

Example of Destination (Queue and Topic)

```
<Context>
  ...
  <Resource name="jms/ExampleQueue" auth="Container" type="org.apache.activemq
        factory="org.apache.activemq.artemis.jndi.JNDIReferenceFactory" addres
  ...
  <Resource name="jms/ExampleTopic" auth="Container" type="org.apache.activemq
         factory="org.apache.activemq.artemis.jndi.JNDIReferenceFactory" addre
  ...
</Context>
`
```

Example Tomcat App

A sample Tomcat app with the container context configured as an example can be seen here:

/examples/features/sub-modules/tomcat

Spring Integration

Apache ActiveMQ Artemis provides a simple bootstrap class, `org.apache.activemq.artemis.integration.spring.SpringJmsBootstrap`, for integration with Spring. To use it, you configure Apache ActiveMQ Artemis as you always would, through its various configuration files like `broker.xml`.

The `SpringJmsBootstrap` class extends the EmbeddedJMS class talked about in embedding ActiveMQ and the same defaults and configuration options apply. See the javadocs for more details on other properties of the bean class.

Example

See the Spring Integration Example for a demonstration of how this can work.

CDI Integration

Apache ActiveMQ Artemis provides a simple CDI integration. It can either use an embedded broker or connect to a remote broker.

Configuring a connection

Configuration is provided by implementing the `ArtemisClientConfiguration` interface.

```
public interface ArtemisClientConfiguration {
   String getHost();

   Integer getPort();

   String getUsername();

   String getPassword();

   String getUrl();

   String getConnectorFactory();

   boolean startEmbeddedBroker();

   boolean isHa();

   boolean hasAuthentication();
}
```

There's a default configuration out of the box, if none is specified. This will generate an embedded broker.

Intercepting Operations

Apache ActiveMQ Artemis supports *interceptors* to intercept packets entering and exiting the server. Incoming and outgoing interceptors are be called for any packet entering or exiting the server respectively. This allows custom code to be executed, e.g. for auditing packets, filtering or other reasons. Interceptors can change the packets they intercept. This makes interceptors powerful, but also potentially dangerous.

Implementing The Interceptors

All interceptors are protocol specific.

An interceptor for the core protocol must implement the interface `Interceptor`:

```
package org.apache.activemq.artemis.api.core.interceptor;

public interface Interceptor
{
   boolean intercept(Packet packet, RemotingConnection connection) throws Acti
}
```

For stomp protocol an interceptor must implement the interface `StompFrameInterceptor`:

```
package org.apache.activemq.artemis.core.protocol.stomp;

public interface StompFrameInterceptor extends BaseInterceptor<StompFrame>
{
   boolean intercept(StompFrame stompFrame, RemotingConnection connection);
}
```

Likewise for MQTT protocol, an interceptor must implement the interface `MQTTInterceptor`:

```
package org.apache.activemq.artemis.core.protocol.mqtt;

public interface MQTTInterceptor extends BaseInterceptor<MqttMessage>
{
    boolean intercept(MqttMessage mqttMessage, RemotingConnection connection);
}
```

The returned boolean value is important:

- if `true` is returned, the process continues normally
- if `false` is returned, the process is aborted, no other interceptors will be called and the packet will not be processed further by the server.

Configuring The Interceptors

Both incoming and outgoing interceptors are configured in `broker.xml`:

```
<remoting-incoming-interceptors>
   <class-name>org.apache.activemq.artemis.jms.example.LoginInterceptor</class
   <class-name>org.apache.activemq.artemis.jms.example.AdditionalPropertyInter
</remoting-incoming-interceptors>

<remoting-outgoing-interceptors>
   <class-name>org.apache.activemq.artemis.jms.example.LogoutInterceptor</clas
   <class-name>org.apache.activemq.artemis.jms.example.AdditionalPropertyInter
</remoting-outgoing-interceptors>
```

See the documentation on adding runtime dependencies to understand how to make your interceptor available to the broker.

Interceptors on the Client Side

The interceptors can also be run on the Apache ActiveMQ Artemit client side to intercept packets either sent by the client to the server or by the server to the client. This is done by adding the interceptor to the `ServerLocator` with the `addIncomingInterceptor(Interceptor)` or `addOutgoingInterceptor(Interceptor)` methods.

As noted above, if an interceptor returns `false` then the sending of the packet is aborted which means that no other interceptors are be called and the packet is not be processed further by the client. Typically this process happens transparently to the client (i.e. it has no idea if a packet was aborted or not). However, in the case of an outgoing packet that is sent in a `blocking` fashion a `ActiveMQException` will be thrown to the caller. The exception is thrown because blocking sends provide reliability and it is considered an error for them not to succeed. `Blocking` sends occurs when, for example, an application invokes `setBlockOnNonDurableSend(true)` or `setBlockOnDurableSend(true)` on its `ServerLocator` or if an application is using a JMS connection factory retrieved from JNDI that has either `block-on-durable-send` or `block-on-non-durable-send` set to `true`. Blocking is also used for packets dealing with transactions (e.g. commit, roll-back, etc.). The `ActiveMQException` thrown will contain the name of the interceptor that returned false.

As on the server, the client interceptor classes (and their dependencies) must be added to the classpath to be properly instantiated and invoked.

Examples

See the following examples which show how to use interceptors:

- Interceptor
- Interceptor AMQP
- Interceptor Client
- Interceptor MQTT

Data Tools

You can use the Artemis CLI to execute data maintenance tools:

This is a list of sub-commands available

Name	Description
exp	Export the message data using a special and independent XML format
imp	Imports the journal to a running broker using the output from expt
data	Prints a report about journal records and summary of existent records, as well a report on paging
encode	shows an internal format of the journal encoded to String
decode	imports the internal journal format from encode

You can use the help at the tool for more information on how to execute each of the tools. For example:

```
$ ./artemis help data print
NAME
        artemis data print - Print data records information (WARNING: don't us
        while a production server is running)

SYNOPSIS
        artemis data print [--bindings <binding>] [--broker <brokerConfig>]
                [--f] [--jdbc] [--jdbc-bindings-table-name <jdbcBindings>]
                [--jdbc-connection-url <jdbcURL>]
                [--jdbc-driver-class-name <jdbcClassName>]
                [--jdbc-large-message-table-name <jdbcLargeMessages>]
                [--jdbc-message-table-name <jdbcMessages>]
                [--jdbc-page-store-table-name <jdbcPageStore>] [--journal <jou
                [--large-messages <largeMessges>] [--output <output>]
                [--paging <paging>] [--safe] [--verbose] [--] [<configuration>

OPTIONS
        --bindings <binding>
            The folder used for bindings (default from broker.xml)

        --broker <brokerConfig>
            This would override the broker configuration from the bootstrap

        --f
            This will allow certain tools like print-data to be performed
            ignoring any running servers. WARNING: Changing data concurrently
            with a running broker may damage your data. Be careful with this
            option.

        --jdbc
            It will activate jdbc

        --jdbc-bindings-table-name <jdbcBindings>
            Name of the jdbc bindigns table

        --jdbc-connection-url <jdbcURL>
            The connection used for the database

        --jdbc-driver-class-name <jdbcClassName>
            JDBC driver classname

        --jdbc-large-message-table-name <jdbcLargeMessages>
            Name of the large messages table

        --jdbc-message-table-name <jdbcMessages>
            Name of the jdbc messages table

        --jdbc-page-store-table-name <jdbcPageStore>
            Name of the page sotre messages table

        --journal <journal>
            The folder used for messages journal (default from broker.xml)

        --large-messages <largeMessges>
            The folder used for large-messages (default from broker.xml)

        --output <output>
            Output name for the file

        --paging <paging>
            The folder used for paging (default from broker.xml)

        --safe
            It will print your data structure without showing your data
```

```
        --verbose
            Adds more information on the execution

        --
            This option can be used to separate command-line options from the
            list of argument, (useful when arguments might be mistaken for
            command-line options

        <configuration>
            Broker Configuration URI, default
            'xml:${ARTEMIS_INSTANCE}/etc/bootstrap.xml'
```

For a full list of data tools commands available use:

```
$ ./artemis help data
NAME
        artemis data - data tools group (print|imp|exp|encode|decode|compact)
        (example ./artemis data print)

SYNOPSIS
        artemis data
        artemis data compact [--verbose] [--paging <paging>]
                [--journal <journal>] [--large-messages <largeMessges>]
                [--broker <brokerConfig>] [--bindings <binding>]
        artemis data decode [--verbose] [--suffix <suffix>] [--paging <paging>
                [--prefix <prefix>] [--file-size <size>] --input <input>
                [--journal <journal>] [--directory <directory>]
                [--large-messages <largeMessges>] [--broker <brokerConfig>]
                [--bindings <binding>]
        artemis data encode [--verbose] [--directory <directory>]
                [--suffix <suffix>] [--paging <paging>] [--prefix <prefix>]
                [--file-size <size>] [--journal <journal>]
                [--large-messages <largeMessges>] [--broker <brokerConfig>]
                [--bindings <binding>]
        artemis data exp [--jdbc-bindings-table-name <jdbcBindings>]
                [--jdbc-message-table-name <jdbcMessages>] [--paging <paging>]
                [--jdbc-connection-url <jdbcURL>]
                [--jdbc-large-message-table-name <jdbcLargeMessages>] [--f]
                [--large-messages <largeMessges>] [--broker <brokerConfig>]
                [--jdbc-page-store-table-name <jdbcPageStore>]
                [--jdbc-driver-class-name <jdbcClassName>] [--jdbc] [--verbose
                [--journal <journal>] [--output <output>] [--bindings <binding
        artemis data imp [--user <user>] [--legacy-prefixes] [--verbose]
                [--host <host>] [--port <port>] [--transaction] --input <input
                [--password <password>] [--sort]
        artemis data print [--jdbc-bindings-table-name <jdbcBindings>]
                [--jdbc-message-table-name <jdbcMessages>] [--paging <paging>]
                [--jdbc-connection-url <jdbcURL>]
                [--jdbc-large-message-table-name <jdbcLargeMessages>] [--f]
                [--large-messages <largeMessges>] [--broker <brokerConfig>]
                [--jdbc-page-store-table-name <jdbcPageStore>]
                [--jdbc-driver-class-name <jdbcClassName>] [--safe] [--jdbc] [
                [--journal <journal>] [--output <output>] [--bindings <binding

COMMANDS
        With no arguments, Display help information

        print
            Print data records information (WARNING: don't use while a
            production server is running)

            With --jdbc-bindings-table-name option, Name of the jdbc bindigns
            table

            With --jdbc-message-table-name option, Name of the jdbc messages
            table

            With --paging option, The folder used for paging (default from
            broker.xml)

            With --jdbc-connection-url option, The connection used for the
            database

            With --jdbc-large-message-table-name option, Name of the large
            messages table

            With --f option, This will allow certain tools like print-data to
            performed ignoring any running servers. WARNING: Changing data
            concurrently with a running broker may damage your data. Be carefu
```

```
            with this option.

            With --large-messages option, The folder used for large-messages
            (default from broker.xml)

            With --broker option, This would override the broker configuration
            from the bootstrap

            With --jdbc-page-store-table-name option, Name of the page sotre
            messages table

            With --jdbc-driver-class-name option, JDBC driver classname

            With --safe option, It will print your data structure without
            showing your data

            With --jdbc option, It will activate jdbc

            With --verbose option, Adds more information on the execution

            With --journal option, The folder used for messages journal (defau
            from broker.xml)

            With --output option, Output name for the file

            With --bindings option, The folder used for bindings (default from
            broker.xml)

        exp
            Export all message-data using an XML that could be interpreted by
            any system.

            With --jdbc-bindings-table-name option, Name of the jdbc bindigns
            table

            With --jdbc-message-table-name option, Name of the jdbc messages
            table

            With --paging option, The folder used for paging (default from
            broker.xml)

            With --jdbc-connection-url option, The connection used for the
            database

            With --jdbc-large-message-table-name option, Name of the large
            messages table

            With --f option, This will allow certain tools like print-data to
            performed ignoring any running servers. WARNING: Changing data
            concurrently with a running broker may damage your data. Be carefu
            with this option.

            With --large-messages option, The folder used for large-messages
            (default from broker.xml)

            With --broker option, This would override the broker configuration
            from the bootstrap

            With --jdbc-page-store-table-name option, Name of the page sotre
            messages table

            With --jdbc-driver-class-name option, JDBC driver classname

            With --jdbc option, It will activate jdbc

            With --verbose option, Adds more information on the execution
```

```
            With --journal option, The folder used for messages journal (defau
            from broker.xml)

            With --output option, Output name for the file

            With --bindings option, The folder used for bindings (default from
            broker.xml)

        imp
            Import all message-data using an XML that could be interpreted by
            any system.

            With --user option, User name used to import the data. (default
            null)

            With --legacy-prefixes option, Do not remove prefixes from legacy
            imports

            With --verbose option, Adds more information on the execution

            With --host option, The host used to import the data (default
            localhost)

            With --port option, The port used to import the data (default 6161

            With --transaction option, If this is set to true you will need a
            whole transaction to commit at the end. (default false)

            With --input option, The input file name (default=exp.dmp)

            With --password option, User name used to import the data. (defaul
            null)

            With --sort option, Sort the messages from the input (used for old
            versions that won't sort messages)

        decode
            Decode a journal's internal format into a new journal set of files

            With --verbose option, Adds more information on the execution

            With --suffix option, The journal suffix (default amq)

            With --paging option, The folder used for paging (default from
            broker.xml)

            With --prefix option, The journal prefix (default activemq-data)

            With --file-size option, The journal size (default 10485760)

            With --input option, The input file name (default=exp.dmp)

            With --journal option, The folder used for messages journal (defau
            from broker.xml)

            With --directory option, The journal folder (default journal folde
            from broker.xml)

            With --large-messages option, The folder used for large-messages
            (default from broker.xml)

            With --broker option, This would override the broker configuration
            from the bootstrap

            With --bindings option, The folder used for bindings (default from
```

```
            broker.xml)

        encode
            Encode a set of journal files into an internal encoded data format

            With --verbose option, Adds more information on the execution

            With --directory option, The journal folder (default the journal
            folder from broker.xml)

            With --suffix option, The journal suffix (default amq)

            With --paging option, The folder used for paging (default from
            broker.xml)

            With --prefix option, The journal prefix (default activemq-data)

            With --file-size option, The journal size (default 10485760)

            With --journal option, The folder used for messages journal (defau
            from broker.xml)

            With --large-messages option, The folder used for large-messages
            (default from broker.xml)

            With --broker option, This would override the broker configuration
            from the bootstrap

            With --bindings option, The folder used for bindings (default from
            broker.xml)

        compact
            Compacts the journal of a non running server

            With --verbose option, Adds more information on the execution

            With --paging option, The folder used for paging (default from
            broker.xml)

            With --journal option, The folder used for messages journal (defau
            from broker.xml)

            With --large-messages option, The folder used for large-messages
            (default from broker.xml)

            With --broker option, This would override the broker configuration
            from the bootstrap

            With --bindings option, The folder used for bindings (default from
            broker.xml)
```

Maven Plugins

Since Artemis 1.1.0 Artemis provides the possibility of using Maven Plugins to manage the life cycle of servers.

When to use it

These Maven plugins were initially created to manage server instances across our examples. They can create a server, start, and do any CLI operation over servers.

You could for example use these maven plugins on your testsuite or deployment automation.

Goals

There are three goals that you can use

- `create`

  This will create a server accordingly to your arguments. You can do some extra tricks here such as installing extra libraries for external modules.

- `cli`

  This will perform any CLI operation. This is basically a maven expression of the CLI classes

- `runClient`

  This is a simple wrapper around classes implementing a static main call. Notice that this won't spawn a new VM or new Thread.

Declaration

On your pom, use the plugins section:

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.activemq</groupId>
         <artifactId>artemis-maven-plugin</artifactId>
      </plugin>
   </plugins>
</build>
```

create goal

I won't detail every operation of the create plugin here, but I will try to describe the main parameters:

Name	Description
configuration	A place that will hold any file to replace on the configuration. For instance if you are providing your own broker.xml. Default is "${basedir}/target/classes/activemq/server0"
home	The location where you downloaded and installed artemis. Default is "${activemq.basedir}"
alternateHome	This is used case you have two possible locations for your home (e.g. one under compile and one under production
instance	Where the server is going to be installed. Default is "${basedir}/target/server0"
liblist[]	A list of libraries to be installed under ./lib. ex: "org.jgroups:jgroups:3.6.0.Final"

Example:

```
<execution>
   <id>create</id>
   <goals>
      <goal>create</goal>
   </goals>
   <configuration>
      <ignore>${noServer}</ignore>
   </configuration>
</execution>
```

cli goal

Some properties for the CLI

Name	Description
configuration	A place that will hold any file to replace on the configuration. For instance if you are providing your own broker.xml. Default is "${basedir}/target/classes/activemq/server0"
home	The location where you downloaded and installed artemis. Default is "${activemq.basedir}"
alternateHome	This is used case you have two possible locations for your home (e.g. one under compile and one under production
instance	Where the server is going to be installed. Default is "${basedir}/target/server0"

Similarly to the create plugin, the artemis exampels are using the cli plugin. Look at them for concrete examples.

Example:

```
<execution>
  <id>start</id>
  <goals>
     <goal>cli</goal>
  </goals>
  <configuration>
     <spawn>true</spawn>
     <ignore>${noServer}</ignore>
     <testURI>tcp://localhost:61616</testURI>
     <args>
        <param>run</param>
     </args>
  </configuration>
</execution>
```

runClient goal

This is a simple solution for running classes implementing the main method.

Name	Description
clientClass	A class implement a static void main(String arg[])
args	A string array of arguments passed to the method

Example:

```
<execution>
  <id>runClient</id>
  <goals>
     <goal>runClient</goal>
  </goals>

  <configuration>
     <clientClass>org.apache.activemq.artemis.jms.example.QueueExample</client
  </configuration>
</execution>
```

Complete example

The following example is a copy of the /examples/features/standard/queue example. You may refer to it directly under the examples directory tree.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.ap
   <modelVersion>4.0.0</modelVersion>

   <parent>
      <groupId>org.apache.activemq.examples.broker</groupId>
      <artifactId>jms-examples</artifactId>
      <version>1.1.0</version>
   </parent>

   <artifactId>queue</artifactId>
   <packaging>jar</packaging>
   <name>ActiveMQ Artemis JMS Queue Example</name>

   <properties>
      <activemq.basedir>${project.basedir}/../../../..</activemq.basedir>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.apache.activemq</groupId>
         <artifactId>artemis-jms-client</artifactId>
         <version>${project.version}</version>
      </dependency>
   </dependencies>

   <build>
      <plugins>
         <plugin>
            <groupId>org.apache.activemq</groupId>
            <artifactId>artemis-maven-plugin</artifactId>
            <executions>
               <execution>
                  <id>create</id>
                  <goals>
                     <goal>create</goal>
                  </goals>
                  <configuration>
                     <ignore>${noServer}</ignore>
                  </configuration>
               </execution>
               <execution>
                  <id>start</id>
                  <goals>
                     <goal>cli</goal>
                  </goals>
                  <configuration>
                     <spawn>true</spawn>
                     <ignore>${noServer}</ignore>
                     <testURI>tcp://localhost:61616</testURI>
                     <args>
                        <param>run</param>
                     </args>
                  </configuration>
               </execution>
               <execution>
                  <id>runClient</id>
                  <goals>
                     <goal>runClient</goal>
                  </goals>
                  <configuration>
                     <clientClass>org.apache.activemq.artemis.jms.example.Queu
                  </configuration>
               </execution>
               <execution>
```

```
                  <id>stop</id>
                  <goals>
                     <goal>cli</goal>
                  </goals>
                  <configuration>
                     <ignore>${noServer}</ignore>
                     <args>
                        <param>stop</param>
                     </args>
                  </configuration>
               </execution>
            </executions>
            <dependencies>
               <dependency>
                  <groupId>org.apache.activemq.examples.broker</groupId>
                  <artifactId>queue</artifactId>
                  <version>${project.version}</version>
               </dependency>
            </dependencies>
         </plugin>
      </plugins>
   </build>

</project>
```

Unit Testing

The package `artemis-junit` provides tools to facilitate how to run Artemis resources inside JUnit Tests.

These are provided as JUnit "rules" and can make it easier to embed messaging functionality on your tests.

Example

Import this on your pom.xml

```
<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>artemis-junit</artifactId>
   <!-- replace this for the version you are using -->
   <version>2.5.0</version>
   <scope>test</scope>
</dependency>
```

Declare a rule on your JUnit Test

```
import org.apache.activemq.artemis.junit.EmbeddedActiveMQResource;
import org.junit.Rule;

import org.junit.Test;

public class MyTest {

   @Rule
   public EmbeddedActiveMQResource resource = new EmbeddedActiveMQResource();

   @Test
   public void myTest() {

   }
}
```

This will start a server that will be available for your test:

```
[main] 17:00:16,644 INFO  [org.apache.activemq.artemis.core.server] AMQ221000:
[main] 17:00:16,666 INFO  [org.apache.activemq.artemis.core.server] AMQ221045:
[main] 17:00:16,688 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
[main] 17:00:16,801 INFO  [org.apache.activemq.artemis.core.server] AMQ221007:
[main] 17:00:16,801 INFO  [org.apache.activemq.artemis.core.server] AMQ221001:
[main] 17:00:16,891 INFO  [org.apache.activemq.artemis.core.server] AMQ221002:
```

Ordering rules

This is actually a JUnit feature, but this could be helpful on pre-determining the order on which rules are executed.

```
ActiveMQDynamicProducerResource producer = new ActiveMQDynamicProducerResource

@Rule
public RuleChain ruleChain = RuleChain.outerRule(new ThreadLeakCheckRule()).ar
```

Available Rules

Name	Description
EmbeddedActiveMQResource	Run a Server, without the JMS manager
EmbeddedJMSResource	Run a Server, including the JMS Manager
ActiveMQConsumerResource	Automate the creation of a consumer
ActiveMQProducerResource	Automate the creation of a producer
ThreadLeakCheckRule	Check that all threads have been finished after the test is finished

Performance Tuning

In this chapter we'll discuss how to tune Apache ActiveMQ Artemis for optimum performance.

Tuning persistence

- To get the best performance from Apache ActiveMQ Artemis whilst using persistent messages it is recommended that the file store is used. Apache ActiveMQ Artemis also supports JDBC persistence, but there is a performance cost when persisting to a database vs local disk.
- Put the message journal on its own physical volume. If the disk is shared with other processes e.g. transaction co-ordinator, database or other journals which are also reading and writing from it, then this may greatly reduce performance since the disk head may be skipping all over the place between the different files. One of the advantages of an append only journal is that disk head movement is minimised - this advantage is destroyed if the disk is shared. If you're using paging or large messages make sure they're ideally put on separate volumes too.
- Minimum number of journal files. Set `journal-min-files` to a number of files that would fit your average sustainable rate. This number represents the lower threshold of the journal file pool.
- To set the upper threshold of the journal file pool. (`journal-min-files` being the lower threshold). Set `journal-pool-files` to a number that represents something near your maximum expected load. The journal will spill over the pool should it need to, but will shrink back to the upper threshold, when possible. This allows reuse of files, without taking up more disk space than required. If you see new files being created on the journal data directory too often, i.e. lots of data is being persisted, you need to increase the journal-pool-size, this way the journal would reuse more files instead of creating new data files, increasing performance
- Journal file size. The journal file size should be aligned to the capacity of a cylinder on the disk. The default value 10MiB should be enough on most systems.
- Use `ASYNCIO` journal. If using Linux, try to keep your journal type as `ASYNCIO`. `ASYNCIO` will scale better than Java NIO.
- Tune `journal-buffer-timeout`. The timeout can be increased to increase throughput at the expense of latency.
- If you're running `ASYNCIO` you might be able to get some better performance by increasing `journal-max-io`. DO NOT change this parameter if you are running NIO.

- If you are 100% sure you don't need power failure durability guarantees, disable `journal-data-sync` and use `NIO` or `MAPPED` journal: you'll benefit a huge performance boost on writes with process failure durability guarantees.

Tuning JMS

There are a few areas where some tweaks can be done if you are using the JMS API

- Disable message id. Use the `setDisableMessageID()` method on the `MessageProducer` class to disable message ids if you don't need them. This decreases the size of the message and also avoids the overhead of creating a unique ID.
- Disable message timestamp. Use the `setDisableMessageTimeStamp()` method on the `MessageProducer` class to disable message timestamps if you don't need them.
- Avoid `ObjectMessage`. `ObjectMessage` is convenient but it comes at a cost. The body of a `ObjectMessage` uses Java serialization to serialize it to bytes. The Java serialized form of even small objects is very verbose so takes up a lot of space on the wire, also Java serialization is slow compared to custom marshalling techniques. Only use `ObjectMessage` if you really can't use one of the other message types, i.e. if you really don't know the type of the payload until run-time.
- Avoid `AUTO_ACKNOWLEDGE`. `AUTO_ACKNOWLEDGE` mode requires an acknowledgement to be sent from the server for each message received on the client, this means more traffic on the network. If you can, use `DUPS_OK_ACKNOWLEDGE` or use `CLIENT_ACKNOWLEDGE` or a transacted session and batch up many acknowledgements with one acknowledge/commit.
- Avoid durable messages. By default JMS messages are durable. If you don't really need durable messages then set them to be non-durable. Durable messages incur a lot more overhead in persisting them to storage.
- Batch many sends or acknowledgements in a single transaction. Apache ActiveMQ Artemis will only require a network round trip on the commit, not on every send or acknowledgement.

Other Tunings

There are various other places in Apache ActiveMQ Artemis where we can perform some tuning:

- Use Asynchronous Send Acknowledgements. If you need to send durable messages non transactionally and you need a guarantee that they have reached the server by the time the call to send() returns, don't set durable messages to be sent blocking, instead use asynchronous send

acknowledgements to get your acknowledgements of send back in a separate stream, see Guarantees of sends and commits for more information on this.

- Use pre-acknowledge mode. With pre-acknowledge mode, messages are acknowledged `before` they are sent to the client. This reduces the amount of acknowledgement traffic on the wire. For more information on this, see Extra Acknowledge Modes.
- Disable security. You may get a small performance boost by disabling security by setting the `security-enabled` parameter to `false` in `broker.xml` .
- Disable persistence. If you don't need message persistence, turn it off altogether by setting `persistence-enabled` to false in `broker.xml` .
- Sync transactions lazily. Setting `journal-sync-transactional` to `false` in `broker.xml` can give you better transactional persistent performance at the expense of some possibility of loss of transactions on failure. See Guarantees of sends and commits for more information.
- Sync non transactional lazily. Setting `journal-sync-non-transactional` to `false` in `broker.xml` can give you better non-transactional persistent performance at the expense of some possibility of loss of durable messages on failure. See Guarantees of sends and commits for more information.
- Send messages non blocking. Setting `block-on-durable-send` and `block-on-non-durable-send` to `false` in the jms config (if you're using JMS and JNDI) or directly on the ServerLocator. This means you don't have to wait a whole network round trip for every message sent. See Guarantees of sends and commits for more information.
- If you have very fast consumers, you can increase consumer-window-size. This effectively disables consumer flow control.
- Use the core API not JMS. Using the JMS API you will have slightly lower performance than using the core API, since all JMS operations need to be translated into core operations before the server can handle them. If using the core API try to use methods that take `SimpleString` as much as possible. `SimpleString` , unlike java.lang.String does not require copying before it is written to the wire, so if you re-use `SimpleString` instances between calls then you can avoid some unnecessary copying.
- If using frameworks like Spring, configure destinations permanently broker side and enable `cacheDestinations` on the client side. See the Setting The Destination Cache for more information on this.

Tuning Transport Settings

- TCP buffer sizes. If you have a fast network and fast machines you may get a performance boost by increasing the TCP send and receive buffer sizes. See the Configuring the Transport for more information on this.

> **Note:**
>
> Note that some operating systems like later versions of Linux include TCP auto-tuning and setting TCP buffer sizes manually can prevent auto-tune from working and actually give you worse performance!

- Increase limit on file handles on the server. If you expect a lot of concurrent connections on your servers, or if clients are rapidly opening and closing connections, you should make sure the user running the server has permission to create sufficient file handles.

  This varies from operating system to operating system. On Linux systems you can increase the number of allowable open file handles in the file `/etc/security/limits.conf` e.g. add the lines

  ```
  serveruser   soft  nofile  20000
  serveruser   hard  nofile  20000
  ```

  This would allow up to 20000 file handles to be open by the user `serveruser`.

- Use `batch-delay` and set `direct-deliver` to false for the best throughput for very small messages. Apache ActiveMQ Artemis comes with a preconfigured connector/acceptor pair (`netty-throughput`) in `broker.xml` and JMS connection factory (`ThroughputConnectionFactory`) in `activemq-jms.xml` which can be used to give the very best throughput, especially for small messages. See the Configuring the Transport for more information on this.

Tuning the VM

We highly recommend you use the latest Java JVM for the best performance. We test internally using the Sun JVM, so some of these tunings won't apply to JDKs from other providers (e.g. IBM or JRockit)

- Garbage collection. For smooth server operation we recommend using a parallel garbage collection algorithm, e.g. using the JVM argument `-XX:+UseParallelOldGC` on Sun JDKs.

- Memory settings. Give as much memory as you can to the server. Apache ActiveMQ Artemis can run in low memory by using paging (described in Paging) but if it can run with all queues in RAM this will improve performance. The amount of memory you require will depend on the size and number of your queues and the size and number of your messages. Use the JVM arguments `-Xms` and `-Xmx` to set server available RAM. We recommend setting them to the same high value.

  When under periods of high load, it is likely that Artemis will be generating and destroying lots of objects. This can result in a build up of stale objects. To reduce the chance of running out of memory and causing a full GC (which may introduce pauses and unintentional behaviour), it is recommended that the max heap size (`-Xmx`) for the JVM is set at least to 5 x the `global-max-`

`size` of the broker. As an example, in a situation where the broker is under high load and running with a `global-max-size` of 1GB, it is recommended the the max heap size is set to 5GB.

Avoiding Anti-Patterns

- Re-use connections / sessions / consumers / producers. Probably the most common messaging anti-pattern we see is users who create a new connection/session/producer for every message they send or every message they consume. This is a poor use of resources. These objects take time to create and may involve several network round trips. Always re-use them.

  > **Note:**
  >
  > Some popular libraries such as the Spring JMS Template are known to use these anti-patterns. If you're using Spring JMS Template and you're getting poor performance you know why. Don't blame Apache ActiveMQ Artemis! The Spring JMS Template can only safely be used in an app server which caches JMS sessions (e.g. using JCA), and only then for sending messages. It cannot be safely be used for synchronously consuming messages, even in an app server.

- Avoid fat messages. Verbose formats such as XML take up a lot of space on the wire and performance will suffer as result. Avoid XML in message bodies if you can.
- Don't create temporary queues for each request. This common anti-pattern involves the temporary queue request-response pattern. With the temporary queue request-response pattern a message is sent to a target and a reply-to header is set with the address of a local temporary queue. When the recipient receives the message they process it then send back a response to the address specified in the reply-to. A common mistake made with this pattern is to create a new temporary queue on each message sent. This will drastically reduce performance. Instead the temporary queue should be re-used for many requests.
- Don't use Message-Driven Beans for the sake of it. As soon as you start using MDBs you are greatly increasing the codepath for each message received compared to a straightforward message consumer, since a lot of extra application server code is executed. Ask yourself do you really need MDBs? Can you accomplish the same task using just a normal message consumer?

Troubleshooting

UDP not working

In certain situations UDP used on discovery may not work. Typical situations are:

1. The nodes are behind a firewall. If your nodes are on different machines then it is possible that the firewall is blocking the multicasts. you can test this by disabling the firewall for each node or adding the appropriate rules.
2. You are using a home network or are behind a gateway. Typically home networks will redirect any UDP traffic to the Internet Service Provider which is then either dropped by the ISP or just lost. To fix this you will need to add a route to the firewall/gateway that will redirect any multicast traffic back on to the local network instead.
3. All the nodes are in one machine. If this is the case then it is a similar problem to point 2 and the same solution should fix it. Alternatively you could add a multicast route to the loopback interface. On linux the command would be:

   ```
   # you should run this as root
   route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
   ```

   This will redirect any traffic directed to the 224.0.0.0 to the loopback interface. This will also work if you have no network at all. On Mac OS X, the command is slightly different:

   ```
   sudo route add 224.0.0.0 127.0.0.1 -netmask 240.0.0.0
   ```

Configuration Reference

This section is a quick index for looking up configuration. Click on the element name to go to the specific chapter.

Broker Configuration

broker.xml

This is the main core server configuration file which contains the `core` element. The `core` element contains the main server configuration.

Modularising broker.xml

XML XInclude support is provided in `broker.xml` so that you can break your configuration out into separate files.

To do this ensure the following is defined at the root configuration element.

```
xmlns:xi="http://www.w3.org/2001/XInclude"
```

You can now define include tag's where you want to bring in xml configuration from another file:

```
<xi:include href="my-address-settings.xml"/>
```

You should ensure xml elements in separated files should be namespaced correctly for example if address-settings element was separated, it should have the element namespace defined:

```
 <address-settings xmlns="urn:activemq:core">
```

An example can of this feature can be seen in the test suites:

```
./artemis-server/src/test/resources/ConfigurationTest-xinclude-config.xml
```

Note: if you use `xmllint` to validate the XML against the schema you should enable xinclude flag when running.

```
--xinclude
```

For further information on XInclude see:

https://www.w3.org/TR/xinclude/

Reloading modular configuration files

Certain changes in `broker.xml` can be picked up at runtime as discussed in the Configuration Reload chapter. Changes made directly to files which are included in `broker.xml` via `xi:include` will not be automatically picked up unless the file timestamp on `broker.xml` is also modified. For example, if `broker.xml` is including `my-address-settings.xml` and `my-address-settings.xml` is modified those changes won't be loaded until the user uses something like the touch command to update the `broker.xml` file's timestamp to trigger a reload.

System properties

It is possible to use System properties to replace some of the configuration properties. If you define a System property starting with "brokerconfig." that will be passed along to Bean Utils and the configuration would be replaced.

To define global-max-size=1000000 using a system property you would have to define this property, for example through java arguments:

```
java -Dbrokerconfig.globalMaxSize=1000000
```

You can also change the prefix through the `broker.xml` by setting:

```
<system-property-prefix>yourprefix</system-property-prefix>
```

This is to help you customize artemis on embedded systems.

The core configuration

This describes the root of the XML configuration. You will see here also multiple sub-types listed. For example on the main config you will have bridges and at the list of bridge type we will describe the properties for that configuration.

> **Warning**
>
> The default values listed below are the values which will be used if the configuration parameter is **not set** either programmatically or via `broker.xml`. Some of these values are set in the `broker.xml` which is available out-of-the-box. Any values set in the out-of-the-box configuration will override the default values listed here. Please consult your specific configuration to know which values will actually be used when the broker is running.

Name	Description	Default
acceptors	a list of remoting acceptors	n/a
acceptors.acceptor	Each acceptor is composed for just an URL	n/a
addresses	a list of addresses	n/a
address-settings	a list of address-setting	n/a
allow-failback	Should stop backup on live restart.	`true`
amqp-use-core-subscription-naming	If true uses CORE queue naming convention for AMQP.	`false`
async-connection-execution-enabled	If False delivery would be always asynchronous.	`true`
bindings-directory	The folder in use for the bindings folder	`data/bindings`
bridges	a list of core bridges	n/a
ha-policy	the HA policy of this server	none
broadcast-groups	a list of broadcast-group	n/a
broker-plugins	a list of broker-plugins	n/a
configuration-file-refresh-period	The frequency in milliseconds the configuration file is checked for changes	5000
check-for-live-server	Used for a live server to verify if there are other nodes with the same ID on the topology	n/a
cluster-connections	a list of cluster-connection	n/a
cluster-password	Cluster password. It applies to all cluster configurations.	n/a
cluster-user	Cluster username. It applies to all cluster configurations.	n/a
connection-ttl-override	if set, this will override how long (in ms) to keep a connection alive without receiving a ping. -1 disables this setting.	-1
connection-ttl-check-interval	how often (in ms) to check connections for ttl violation.	2000

Name	Description	Default
connectors.connector	The URL for the connector. This is a list	n/a
create-bindings-dir	true means that the server will create the bindings directory on start up.	`true`
create-journal-dir	true means that the journal directory will be created.	`true`
discovery-groups	a list of discovery-group	n/a
disk-scan-period	The interval where the disk is scanned for percentual usage.	5000
diverts	a list of diverts to use	n/a
global-max-size	The amount in bytes before all addresses are considered full.	Half of the JVM's `-Xm`
graceful-shutdown-enabled	true means that graceful shutdown is enabled.	`false`
graceful-shutdown-timeout	Timeout on waiting for clients to disconnect before server shutdown.	-1
grouping-handler	a message grouping handler	n/a
id-cache-size	The duplicate detection circular cache size.	20000
jmx-domain	the JMX domain used to registered MBeans in the MBeanServer.	`org.apache.activemq`
jmx-use-broker-name	whether or not to use the broker name in the JMX properties.	`true`
jmx-management-enabled	true means that the management API is available via JMX.	`true`
journal-buffer-size	The size of the internal buffer on the journal in KB.	490KB
journal-buffer-timeout	The Flush timeout for the journal buffer	500000 for ASYNCIO 3333333 for NIO
journal-compact-min-files	The minimal number of data files before we can start compacting. Setting this to 0 means compacting is disabled.	10

Name	Description	Default
journal-compact-percentage	The percentage of live data on which we consider compacting the journal.	30
journal-directory	the directory to store the journal files in.	`data/journal`
node-manager-lock-directory	the directory to store the node manager lock file.	same of `journal-directory`
journal-file-size	the size (in bytes) of each journal file.	10MB
journal-lock-acquisition-timeout	how long (in ms) to wait to acquire a file lock on the journal.	-1
journal-max-io	the maximum number of write requests that can be in the ASYNCIO queue at any one time.	4096 for ASYNCIO; 1 for NIO; ignored for MAPPED
journal-file-open-timeout	the length of time in seconds to wait when opening a new journal file before timing out and failing.	5
journal-min-files	how many journal files to pre-create.	2
journal-pool-files	The upper threshold of the journal file pool, -1 means no Limit. The system will create as many files as needed however when reclaiming files it will shrink back to the `journal-pool-files`	-1
journal-sync-non-transactional	if true wait for non transaction data to be synced to the journal before returning response to client.	`true`
journal-sync-transactional	if true wait for transaction data to be synchronized to the journal before returning response to client.	`true`
journal-type	the type of journal to use.	`ASYNCIO`
journal-datasync	It will use msync/fsync on journal operations.	`true`
large-messages-directory	the directory to store large messages.	`data/largemessages`
log-delegate-factory-class-name	**deprecated** the name of the factory class to use for log delegation.	n/a

Name	Description	Default
management-address	the name of the management address to send management messages to.	`activemq.management`
management-notification-address	the name of the address that consumers bind to receive management notifications.	`activemq.notificatio`
mask-password	This option controls whether passwords in server configuration need be masked. If set to "true" the passwords are masked.	`false`
max-saved-replicated-journals-size	This specifies how many times a replicated backup server can restart after moving its files on start. Once there are this number of backup journal files the server will stop permanently after if fails back. -1 Means no Limit; 0 don't keep a copy at all.	2
max-disk-usage	The max percentage of data we should use from disks. The broker will block while the disk is full. Disable by setting -1.	90
memory-measure-interval	frequency to sample JVM memory in ms (or -1 to disable memory sampling).	-1
memory-warning-threshold	Percentage of available memory which will trigger a warning log.	25
message-counter-enabled	true means that message counters are enabled.	`false`
message-counter-max-day-history	how many days to keep message counter history.	10
message-counter-sample-period	the sample period (in ms) to use for message counters.	10000
message-expiry-scan-period	how often (in ms) to scan for expired messages.	30000
message-expiry-thread-priority	**deprecated** the priority of the thread expiring messages.	3
metrics-plugin	a plugin to export metrics	n/a

Name	Description	Default
address-queue-scan-period	how often (in ms) to scan for addresses & queues that should be removed.	30000
name	node name; used in topology notifications if set.	n/a
password-codec	the name of the class (and optional configuration properties) used to decode masked passwords. Only valid when `mask-password` is `true`.	n/a
page-max-concurrent-io	The max number of concurrent reads allowed on paging.	5
page-sync-timeout	The time in nanoseconds a page will be synced.	3333333 for ASYNCI `journal-buffer-timeou` for NIO
read-whole-page	If true the whole page would be read, otherwise just seek and read while getting message.	`false`
paging-directory	the directory to store paged messages in.	`data/paging`
persist-delivery-count-before-delivery	True means that the delivery count is persisted before delivery. False means that this only happens after a message has been cancelled.	`false`
persistence-enabled	true means that the server will use the file based journal for persistence.	`true`
persist-id-cache	true means that ID's are persisted to the journal.	`true`
queues	**deprecated** use addresses	n/a
remoting-incoming-interceptors	a list of <class-name/> elements with the names of classes to use for intercepting incoming remoting packets	n/a
remoting-outgoing-interceptors	a list of <class-name/> elements with the names of classes to use for intercepting outgoing remoting packets	n/a

Name	Description	Default
resolveProtocols	Use ServiceLoader to load protocol modules.	`true`
resource-limit-settings	a list of resource-limits	n/a
scheduled-thread-pool-max-size	Maximum number of threads to use for the scheduled thread pool.	5
security-enabled	true means that security is enabled.	`true`
security-invalidation-interval	how long (in ms) to wait before invalidating the security cache.	10000
system-property-prefix	Prefix for replacing configuration settings using Bean Utils.	n/a
internal-naming-prefix	the prefix used when naming the internal queues and addresses required for implementing certain behaviours.	`$.activemq.internal`
populate-validated-user	whether or not to add the name of the validated user to the messages that user sends.	`false`
security-settings	a list of security-setting.	n/a
thread-pool-max-size	Maximum number of threads to use for the thread pool. -1 means 'no limits'.	30
transaction-timeout	how long (in ms) before a transaction can be removed from the resource manager after create time.	300000
transaction-timeout-scan-period	how often (in ms) to scan for timeout transactions.	1000
wild-card-routing-enabled	true means that the server supports wild card routing.	`true`
network-check-NIC	the NIC (Network Interface Controller) to be used on InetAddress.isReachable.	n/a
network-check-URL-list	the list of http URIs to be used to validate the network.	n/a
network-check-list	the list of pings to be used on ping or InetAddress.isReachable.	n/a

Name	Description	Default
network-check-period	a frequency in milliseconds to how often we should check if the network is still up.	10000
network-check-timeout	a timeout used in milliseconds to be used on the ping.	1000
network-check-ping-command	the command used to oping IPV4 addresses.	n/a
network-check-ping6-command	the command used to oping IPV6 addresses.	n/a
critical-analyzer	enable or disable the critical analysis.	`true`
critical-analyzer-timeout	timeout used to do the critical analysis.	120000 ms
critical-analyzer-check-period	time used to check the response times.	0.5 * `critical-analyz` `timeout`
critical-analyzer-policy	should the server log, be halted or shutdown upon failures.	`LOG`
resolve-protocols	if true then the broker will make use of any protocol managers that are in available on the classpath, otherwise only the core protocol will be available, unless in embedded mode where users can inject their own protocol managers.	`true`
resource-limit-settings	a list of resource-limit.	n/a
server-dump-interval	interval to log server specific information (e.g. memory usage etc).	-1
store	the store type used by the server.	n/a
wildcard-addresses	parameters to configure wildcard address matching format.	n/a

address-setting type

Name	Description	Default
match	The filter to apply to the setting	n/a
dead-letter-address	Dead letter address	n/a
auto-create-dead-letter-resources	Whether or not to auto-create dead-letter address and/or queue	`false`
dead-letter-queue-prefix	Prefix to use for auto-created dead-letter queues	`DLQ.`
dead-letter-queue-suffix	Suffix to use for auto-created dead-letter queues	`` (empty)
expiry-address	Expired messages address	n/a
expiry-delay	Expiration time override; -1 don't override	-1
redelivery-delay	Time to wait before redelivering a message	0
redelivery-delay-multiplier	Multiplier to apply to the `redelivery-delay`	1.0
redelivery-collision-avoidance-factor	an additional factor used to calculate an adjustment to the `redelivery-delay` (up or down)	0.0
max-redelivery-delay	Max value for the `redelivery-delay`	10 * `redelivery-delay`
max-delivery-attempts	Number of retries before dead letter address	10
max-size-bytes	Max size a queue can be before invoking `address-full-policy`	-1
max-size-bytes-reject-threshold	Used with `BLOCK`, the max size an address can reach before messages are rejected; works in combination with `max-size-bytes` **for AMQP clients only**.	-1
page-size-bytes	Size of each file on page	10485760
page-max-cache-size	Maximum number of files cached from paging	5
address-full-policy	What to do when a queue reaches `max-size-bytes`	`PAGE`
message-counter-history-day-limit	Days to keep message counter data	0

Name	Description	Default
last-value-queue	**deprecated** Queue is a last value queue; see `default-last-value-queue` instead	`false`
default-last-value-queue	`last-value` value if none is set on the queue	`false`
default-last-value-key	`last-value-key` value if none is set on the queue	`null`
default-exclusive-queue	`exclusive` value if none is set on the queue	`false`
default-non-destructive	`non-destructive` value if none is set on the queue	`false`
default-consumers-before-dispatch	`consumers-before-dispatch` value if none is set on the queue	0
default-delay-before-dispatch	`delay-before-dispatch` value if none is set on the queue	-1
redistribution-delay	Timeout before redistributing values after no consumers	-1
send-to-dla-on-no-route	Forward messages to DLA when no queues subscribing	`false`
slow-consumer-threshold	Min rate of msgs/sec consumed before a consumer is considered "slow"	-1
slow-consumer-policy	What to do when "slow" consumer is detected	`NOTIFY`
slow-consumer-check-period	How often to check for "slow" consumers	5
auto-create-jms-queues	**deprecated** Create JMS queues automatically; see `auto-create-queues` & `auto-create-addresses`	`true`
auto-delete-jms-queues	**deprecated** Delete JMS queues automatically; see `auto-create-queues` & `auto-create-addresses`	`true`
auto-create-jms-topics	**deprecated** Create JMS topics automatically; see `auto-create-queues` & `auto-create-addresses`	`true`
auto-delete-jms-topics	**deprecated** Delete JMS topics automatically; see `auto-create-queues` & `auto-create-addresses`	`true`
auto-create-queues	Create queues automatically	`true`

Name	Description	Default
auto-delete-queues	Delete auto-created queues automatically	`true`
auto-delete-created-queues	Delete created queues automatically	`false`
auto-delete-queues-delay	Delay for deleting auto-created queues	0
auto-delete-queues-message-count	Message count the queue must be at or below before it can be auto deleted	0
config-delete-queues	How to deal with queues deleted from XML at runtime	`OFF`
auto-create-addresses	Create addresses automatically	`true`
auto-delete-addresses	Delete auto-created addresses automatically	`true`
auto-delete-addresses-delay	Delay for deleting auto-created addresses	0
config-delete-addresses	How to deal with addresses deleted from XML at runtime	`OFF`
management-browse-page-size	Number of messages a management resource can browse	200
default-purge-on-no-consumers	`purge-on-no-consumers` value if none is set on the queue	`false`
default-max-consumers	`max-consumers` value if none is set on the queue	-1
default-queue-routing-type	Routing type for auto-created queues if the type can't be otherwise determined	`MULTICAST`
default-address-routing-type	Routing type for auto-created addresses if the type can't be otherwise determined	`MULTICAST`
default-ring-size	The ring-size applied to queues without an explicit `ring-size` configured	`-1`
retroactive-message-count	the number of messages to preserve for future queues created on the matching address	`0`

bridge type

Name	Description	Default
name	unique name	n/a
queue-name	name of queue that this bridge consumes from	n/a
forwarding-address	address to forward to. If omitted original address is used	n/a
ha	whether this bridge supports fail-over	`false`
filter	optional core filter expression	n/a
transformer-class-name	optional name of transformer class	n/a
min-large-message-size	Limit before message is considered large.	100KB
check-period	How often to check for TTL violation. -1 means disabled.	30000
connection-ttl	TTL for the Bridge. This should be greater than the ping period.	60000
retry-interval	period (in ms) between successive retries.	2000
retry-interval-multiplier	multiplier to apply to successive retry intervals.	1
max-retry-interval	Limit to the retry-interval growth.	2000
reconnect-attempts	maximum number of retry attempts.	-1 (no limit)
use-duplicate-detection	forward duplicate detection headers?	`true`
confirmation-window-size	number of bytes before confirmations are sent.	1MB
producer-window-size	Producer flow control size on the bridge.	-1 (disabled)
user	Username for the bridge, the default is the cluster username.	n/a
password	Password for the bridge, default is the cluster password.	n/a
reconnect-attempts-same-node	Number of retries before trying another node.	10
routing-type	how to set the routing-type on the bridged message	`PASS`

broadcast-group type

Name	Type
name	unique name
local-bind-address	Local bind address that the datagram socket is bound to.
local-bind-port	Local port to which the datagram socket is bound to.
group-address	Multicast address to which the data will be broadcast.
group-port	UDP port number used for broadcasting.
broadcast-period	Period in milliseconds between consecutive broadcasts. Default=2000.
jgroups-file	Name of JGroups configuration file.
jgroups-channel	Name of JGroups Channel.
connector-ref	The `connector` to broadcast.

cluster-connection type

Name	Description	Default
name	unique name	n/a
address	name of the address this cluster connection applies to	n/a
connector-ref	Name of the connector reference to use.	n/a
check-period	The period (in milliseconds) used to check if the cluster connection has failed to receive pings from another server	30000
connection-ttl	Timeout for TTL.	60000
min-large-message-size	Messages larger than this are considered large-messages.	100KB
call-timeout	Time(ms) before giving up on blocked calls.	30000
retry-interval	period (in ms) between successive retries.	500
retry-interval-multiplier	multiplier to apply to the retry-interval.	1
max-retry-interval	Maximum value for retry-interval.	2000
reconnect-attempts	How many attempts should be made to reconnect after failure.	-1
use-duplicate-detection	should duplicate detection headers be inserted in forwarded messages?	`true`
message-load-balancing	how should messages be load balanced?	`OFF`
max-hops	maximum number of hops cluster topology is propagated.	1
confirmation-window-size	The size (in bytes) of the window used for confirming data from the server connected to.	1048576
producer-window-size	Flow Control for the Cluster connection bridge.	-1 (disabled)
call-failover-timeout	How long to wait for a reply if in the middle of a fail-over. -1 means wait forever.	-1
notification-interval	how often the cluster connection will notify the cluster of its existence right after joining the cluster.	1000
notification-attempts	how many times this cluster connection will notify the cluster of its existence right after joining the cluster	2

discovery-group type

Name	Description
name	unique name
group-address	Multicast IP address of the group to listen on
group-port	UDP port number of the multi cast group
jgroups-file	Name of a JGroups configuration file. If specified, the server uses JGroups for discovery.
jgroups-channel	Name of a JGroups Channel. If specified, the server uses the named channel for discovery.
refresh-timeout	Period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. Default=10000
local-bind-address	local bind address that the datagram socket is bound to
local-bind-port	local port to which the datagram socket is bound to. Default=-1
initial-wait-timeout	time to wait for an initial broadcast to give us at least one node in the cluster. Default=10000

divert type

Name	Description
name	unique name
transformer-class-name	an optional class name of a transformer
exclusive	whether this is an exclusive divert. Default=false
routing-name	the routing name for the divert
address	the address this divert will divert from
forwarding-address	the forwarding address for the divert
filter	optional core filter expression
routing-type	how to set the routing-type on the diverted message. Default= `STRIP`

address type

Name	Description	
name	unique name	n/a
anycast	list of anycast queues	
multicast	list of multicast queues	

queue type

Name	Description	Default
name	unique name	n/a
filter	optional core filter expression	n/a
durable	whether the queue is durable (persistent).	`true`
user	the name of the user to associate with the creation of the queue	n/a
max-consumers	the max number of consumers allowed on this queue	-1 (no max)
purge-on-no-consumers	whether or not to delete all messages and prevent routing when no consumers are connected	`false`
exclusive	only deliver messages to one of the connected consumers	`false`
last-value	use last-value semantics	`false`
ring-size	the size this queue should maintain according to ring semantics	based on `default-ring-size` `address-setting`
consumers-before-dispatch	number of consumers required before dispatching messages	0
delay-before-dispatch	milliseconds to wait for `consumers-before-dispatch` to be met before dispatching messages anyway	-1 (wait forever)

security-setting type

Name	Description
match	address expression
permission	
permission.type	the type of permission
permission.roles	a comma-separated list of roles to apply the permission to
role-mapping	A simple role mapping that can be used to map roles from external authentication providers (i.e. LDAP) to internal roles
role-mapping.from	The external role which should be mapped
role-mapping.to	The internal role which should be assigned to the authenticated user

broker-plugin type

Name	Description
property	properties to configure a plugin
class-name	the name of the broker plugin class to instantiate

metrics-plugin type

Name	Description
property	properties to configure a plugin
class-name	the name of the metrics plugin class to instantiate

resource-limit type

Name	Description	Default
match	the name of the user to whom the limits should be applied	n/a
max-connections	how many connections are allowed by the matched user	-1 (no max)
max-queues	how many queues can be created by the matched user	-1 (no max)

grouping-handler type

Name	Description	Default
name	A unique name	n/a
type	`LOCAL` or `REMOTE`	n/a
address	A reference to a `cluster-connection` `address`	n/a
timeout	How long to wait for a decision	5000
group-timeout	How long a group binding will be used.	-1 (disabled)
reaper-period	How often the reaper will be run to check for timed out group bindings. Only valid for `LOCAL` handlers.	30000