Red Hat AMQ Core Protocol JMS 7.11

Using AMQ Core Protocol JMS

Developing an AMQ messaging client using Java

Last Updated: 2024-01-22

Red Hat AMQ Core Protocol JMS 7.11 Using AMQ Core Protocol JMS

Developing an AMQ messaging client using Java

Legal Notice

Abstract

This guide describes how to install and use your client with other AMQ components.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. OVERVIEW

AMQ Core Protocol JMS is a Java Message Service (JMS) 2.0 client for use in messaging applications that send and receive Artemis Core Protocol messages.

AMQ Core Protocol JMS is part of AMQ Clients, a suite of messaging libraries supporting multiple languages and platforms. For information on available clients, see AMQ Clients.

AMQ Core Protocol JMS is based on the JMS implementation from Apache ActiveMQ Artemis. For more information about the JMS API, see the JMS API reference and the JMS tutorial.

1.1. KEY FEATURES

- JMS 1.1 and 2.0 compatible
- SSL/TLS for secure communication
- Automatic reconnect and failover
- Distributed transactions (XA)
- Pure-Java implementation

1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ Core Protocol JMS supports the following industry-recognized standards and network protocols:

- Version 2.0 of the Java Message Service API
- Versions 1.0, 1.1, 1.2, and 1.3 of the Transport Layer Security (TLS) protocol, the successor to SSL
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

Refer to Red Hat AMQ Supported Configurations on the Red Hat Customer Portal for current information regarding AMQ Core Protocol JMS supported configurations.

1.4. TERMS AND CONCEPTS

This section introduces the core API entities and describes how they operate together.

Table 1.1. API terms

Entity	Description
ConnectionFactory	An entry point for creating connections.
Connection	A channel for communication between two peers on a network. It contains sessions.

Entity	Description
Session	A context for producing and consuming messages. It contains message producers and consumers.
MessageProducer	A channel for sending messages to a destination. It has a target destination.
MessageConsumer	A channel for receiving messages from a destination. It has a source destination.
Destination	A named location for messages, either a queue or a topic.
Queue	A stored sequence of messages.
Topic	A stored sequence of messages for multicast distribution.
Message	An application-specific piece of information.

AMQ Core Protocol JMS sends and receives *messages*. Messages are transferred between connected peers using *message producers* and *consumers*. Producers and consumers are established over *sessions*. Sessions are established over *connections*. Connections are created by *connection factories*.

A sending peer creates a producer to send messages. The producer has a *destination* that identifies a target queue or topic at the remote peer. A receiving peer creates a consumer to receive messages. Like the producer, the consumer has a destination that identifies a source queue or topic at the remote peer.

A destination is either a *queue* or a *topic*. In JMS, queues and topics are client-side representations of named broker entities that hold messages.

A queue implements point-to-point semantics. Each message is seen by only one consumer, and the message is removed from the queue after it is read. A topic implements publish-subscribe semantics. Each message is seen by multiple consumers, and the message remains available to other consumers after it is read.

See the JMS tutorial for more information.

1.5. DOCUMENT CONVENTIONS

The sudo command

In this document, **sudo** is used for any command that requires root privileges. Exercise caution when using **sudo** because any changes can affect the entire system. For more information about **sudo**, see Using the sudo command.

File paths

In this document, all file paths are valid for Linux, UNIX, and similar operating systems (for example, **/home/andrea**). On Microsoft Windows, you must use the equivalent Windows paths (for example, **C:\Users\andrea**).

Variable text

This document contains code blocks with variables that you must replace with values specific to your environment. Variable text is enclosed in arrow braces and styled as italic monospace. For example, in the following command, replace ***<project-dir>*** with the value for your environment:

```
$ cd <project-dir>
```

CHAPTER 2. INSTALLATION

This chapter guides you through the steps to install AMQ Core Protocol JMS in your environment.

2.1. PREREQUISITES

- You must have a subscription to access AMQ release files and repositories.
- To build programs with AMQ Core Protocol JMS, you must install Apache Maven.
- To use AMQ Core Protocol JMS, you must install Java.

2.2. USING THE RED HAT MAVEN REPOSITORY

Configure your Maven environment to download the client library from the Red Hat Maven repository.

Procedure

1. Add the Red Hat repository to your Maven settings or POM file. For example configuration files, see Section B.1, “Using the online repository” .

```
<repository>
  <id>red-hat-ga</id>
  <url>https://maven.repository.redhat.com/ga</url>
</repository>
```

2. Add the library dependency to your POM file.

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-jms-client</artifactId>
  <version>2.28.0.redhat-00011</version>
</dependency>
```

The client is now available in your Maven project.

2.3. INSTALLING A LOCAL MAVEN REPOSITORY

As an alternative to the online repository, AMQ Core Protocol JMS can be installed to your local filesystem as a file-based Maven repository.

Procedure

1. Use your subscription to download the **AMQ Broker 7.11.4 Maven repository**.zip file.
2. Extract the file contents into a directory of your choosing.
   On Linux or UNIX, use the **unzip** command to extract the file contents.

```
$ unzip amq-broker-7.11.4-maven-repository.zip
```

   On Windows, right-click the .zip file and select **Extract All**.

3. Configure Maven to use the repository in the **maven-repository** directory inside the extracted install directory. For more information, see Section B.2, “Using a local repository” .

2.4. INSTALLING THE EXAMPLES

Procedure

1. Use your subscription to download the **AMQ Broker 7.11.4**.zip file.

2. Extract the file contents into a directory of your choosing.
   On Linux or UNIX, use the **unzip** command to extract the file contents.

   ```
   $ unzip amq-broker-7.11.4.zip
   ```

   On Windows, right-click the .zip file and select **Extract All**.

   When you extract the contents of the .zip file, a directory named **amq-broker-7.11.4** is created. This is the top-level directory of the installation and is referred to as ***<install-dir>*** throughout this document.

CHAPTER 3. GETTING STARTED

This chapter guides you through the steps to set up your environment and run a simple messaging program.

3.1. PREREQUISITES

- To build the example, Maven must be configured to use the Red Hat repository or a local repository.
- You must install the examples.
- You must have a message broker listening for connections on **localhost**. It must have anonymous access enabled. For more information, see Starting the broker.
- You must have a queue named **exampleQueue**. For more information, see Creating a queue.

3.2. RUNNING YOUR FIRST EXAMPLE

The example creates a consumer and producer for a queue named **exampleQueue**. It sends a text message and then receives it back, printing the received message to the console.

Procedure

1. Use Maven to build the examples by running the following command in the ***<install-dir>*/examples/features/standard/queue** directory.

```
$ mvn clean package dependency:copy-dependencies -DincludeScope=runtime -DskipTests
```

   The addition of **dependency:copy-dependencies** results in the dependencies being copied into the **target/dependency** directory.

2. Use the **java** command to run the example.
   On Linux or UNIX:

```
$ java -cp "target/classes:target/dependency/*"
org.apache.activemq.artemis.jms.example.QueueExample
```

   On Windows:

```
> java -cp "target\classes;target\dependency\*"
org.apache.activemq.artemis.jms.example.QueueExample
```

For example, running it on Linux results in the following output:

```
$ java -cp "target/classes:target/dependency/*"
org.apache.activemq.artemis.jms.example.QueueExample
Sent message: This is a text message
Received message: This is a text message
```

The source code for the example is in the ***<install-dir>*/examples/features/standard/queue/src** directory. Additional examples are available in the ***<install-dir>*/examples/features/standard** directory.

CHAPTER 4. CONFIGURATION

This chapter describes the process for binding the AMQ Core Protocol JMS implementation to your JMS application and setting configuration options.

JMS uses the Java Naming Directory Interface (JNDI) to register and look up API implementations and other resources. This enables you to write code to the JMS API without tying it to a particular implementation.

Configuration options are exposed as query parameters on the connection URI.

4.1. CONFIGURING THE JNDI INITIAL CONTEXT

JMS applications use a JNDI **InitialContext** object obtained from an **InitialContextFactory** to look up JMS objects such as the connection factory. AMQ Core Protocol JMS provides an implementation of the **InitialContextFactory** in the **org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory** class.

The **InitialContextFactory** implementation is discovered when the **InitialContext** object is instantiated:

```
javax.naming.Context context = new javax.naming.InitialContext();
```

To find an implementation, JNDI must be configured in your environment. There are three ways of achieving this: using a **jndi.properties** file, using a system property, or using the initial context API.

Using a jndi.properties file

Create a file named **jndi.properties** and place it on the Java classpath. Add a property with the key **java.naming.factory.initial**.

Example: Setting the JNDI initial context factory using a jndi.properties file

```
java.naming.factory.initial = org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
```

In Maven-based projects, the **jndi.properties** file is placed in the ***<project-dir>*/src/main/resources** directory.

Using a system property

Set the **java.naming.factory.initial** system property.

Example: Setting the JNDI initial context factory using a system property

```
$ java -Djava.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
...
```

Using the initial context API

Use the JNDI initial context API to set properties programatically.

Example: Setting JNDI properties programatically

```
Hashtable<Object, Object> env = new Hashtable<>();

env.put("java.naming.factory.initial",
```

```
"org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory");

InitialContext context = new InitialContext(env);
```

Note that you can use the same API to set the JNDI properties for connection factories, queues, and topics.

4.2. CONFIGURING THE CONNECTION FACTORY

The JMS connection factory is the entry point for creating connections. It uses a connection URI that encodes your application-specific configuration settings.

To set the factory name and connection URI, create a property in the format below. You can store this configuration in a **jndi.properties** file or set the corresponding system property.

The JNDI property format for connection factories

```
connectionFactory.<lookup-name> = <connection-uri>
```

For example, this is how you might configure a factory named **app1**:

Example: Setting the connection factory in a jndi.properties file

```
connectionFactory.app1 = tcp://example.net:61616?clientID=backend
```

You can then use the JNDI context to look up your configured connection factory using the name **app1**:

```
ConnectionFactory factory = (ConnectionFactory) context.lookup("app1");
```

4.3. CONNECTION URIS

Connections are configured using a connection URI. The connection URI specifies the remote host, port, and a set of configuration options, which are set as query parameters. For more information about the available options, see Chapter 5, *Configuration options*.

The connection URI format

```
tcp://<host>:<port>[?<option>=<value>[&<option>=<value>...]]
```

For example, the following is a connection URI that connects to host **example.net** at port **61616** and sets the client ID to **backend**:

Example: A connection URI

```
tcp://example.net:61616?clientID=backend
```

In addition to **tcp**, AMQ Core Protocol JMS also supports the **vm**, **udp**, and **jgroups** schemes. These represent alternate transports and have corresponding acceptor configuration on the broker.

Failover URIs

URIs can contain multiple target connection URIs. If the initial connection to one target fails, another is tried. They take the following form:

The failover URI format

```
(<connection-uri>[,<connection-uri>])[?<option>=<value>[&<option>=<value>...]]
```

Options outside of the parentheses are applied to all of the connection URIs.

4.4. CONFIGURING QUEUE AND TOPIC NAMES

JMS provides the option of using JNDI to look up deployment-specific queue and topic resources.

To set queue and topic names in JNDI, create properties in the following format. Either place this configuration in a **jndi.properties** file or set corresponding system properties.

The JNDI property format for queues and topics

```
queue.<lookup-name> = <queue-name>
topic.<lookup-name> = <topic-name>
```

For example, the following properties define the names **jobs** and **notifications** for two deployment-specific resources:

Example: Setting queue and topic names in a jndi.properties file

```
queue.jobs = app1/work-items
topic.notifications = app1/updates
```

You can then look up the resources by their JNDI names:

```
Queue queue = (Queue) context.lookup("jobs");
Topic topic = (Topic) context.lookup("notifications");
```

CHAPTER 5. CONFIGURATION OPTIONS

This chapter lists the available configuration options for AMQ Core Protocol JMS.

JMS configuration options are set as query parameters on the connection URI. For more information, see Section 4.3, “Connection URIs”.

5.1. GENERAL OPTIONS

user

The user name the client uses to authenticate the connection.

password

The password the client uses to authenticate the connection.

clientID

The client ID that the client applies to the connection.

groupID

The group ID that the client applies to all produced messages.

autoGroup

If enabled, generate a random group ID and apply it to all produced messages.

preAcknowledge

If enabled, acknowledge messages as soon as they are sent and before delivery is complete. This provides "at most once" delivery. It is disabled by default.

blockOnDurableSend

If enabled, when sending non-transacted durable messages, block until the remote peer acknowledges receipt. It is enabled by default.

blockOnNonDurableSend

If enabled, when sending non-transacted non-durable messages, block until the remote peer acknowledges receipt. It is disabled by default.

blockOnAcknowledge

If enabled, when acknowledging non-transacted received messages, block until the remote peer confirms acknowledgment. It is disabled by default.

callTimeout

The time in milliseconds to wait for a blocking call to complete. The default is 30000 (30 seconds).

callFailoverTimeout

When the client is in the process of failing over, the time in millisconds to wait before starting a blocking call. The default is 30000 (30 seconds).

ackBatchSize

The number of bytes a client can receive and acknowledge before the acknowledgement is sent to the broker. The default is 1048576 (1 MiB).

dupsOKBatchSize

When using the **DUPS_OK_ACKNOWLEDGE** acknowledgment mode, the size in bytes of acknowledgment batches. The default is 1048576 (1 MiB).

transactionBatchSize

When receiving messsages in a transaction, the size in bytes of acknowledgment batches. The default is 1048576 (1 MiB).

cacheDestinations

If enabled, cache destination lookups. It is disabled by default.

5.2. TCP OPTIONS

tcpNoDelay

If enabled, do not delay and buffer TCP sends. It is enabled by default.

tcpSendBufferSize

The send buffer size in bytes. The default is 32768 (32 KiB).

tcpReceiveBufferSize

The receive buffer size in bytes. The default is 32768 (32 KiB).

writeBufferLowWaterMark

The limit in bytes below which the write buffer becomes writable. The default is 32768 (32 KiB).

writeBufferHighWaterMark

The limit in bytes above which the write buffer becomes non-writable. The default is 131072 (128 KiB).

5.3. SSL/TLS OPTIONS

sslEnabled

If enabled, use SSL/TLS to authenticate and encrypt connections. It is disabled by default.

keyStorePath

The path to the SSL/TLS key store. A key store is required for mutual SSL/TLS authentication. If unset, the value of the **javax.net.ssl.keyStore** system property is used.

keyStorePassword

The password for the SSL/TLS key store. If unset, the value of the **javax.net.ssl.keyStorePassword** system property is used.

trustStorePath

The path to the SSL/TLS trust store. If unset, the value of the **javax.net.ssl.trustStore** system property is used.

trustStorePassword

The password for the SSL/TLS trust store. If unset, the value of the **javax.net.ssl.trustStorePassword** system property is used.

trustAll

If enabled, trust the provided server certificate implicitly, regardless of any configured trust store. It is disabled by default.

verifyHost

If enabled, verify that the connection hostname matches the provided server certificate. It is disabled by default.

enabledCipherSuites

A comma-separated list of cipher suites to enable. If unset, the JVM default ciphers are used.

enabledProtocols

A comma-separated list of SSL/TLS protocols to enable. If unset, the JVM default protocols are used.

5.4. FAILOVER OPTIONS

initialConnectAttempts

The number of reconnect attempts allowed before the first successful connection and before the client discovers the broker topology. The default is 0, meaning only one attempt is allowed.

failoverOnInitialConnection

If enabled, attempt to connect to the backup server if the initial connection fails. It is disabled by default.

reconnectAttempts

The number of reconnect attempts allowed before reporting the connection as failed. A value of -1 signifies an unlimited number of attempts. The default value is 0.

retryInterval

The time in milliseconds between reconnect attempts. The default is 2000 (2 seconds).

retryIntervalMultiplier

The multiplier used to grow the retry interval. The default is 1.0, meaning equal intervals.

maxRetryInterval

The maximum time in milliseconds between reconnect attempts. The default is 2000 (2 seconds).

ha

If enabled, track changes in the topology of HA brokers. The host and port from the URI is used only for the initial connection. After initial connection, the client receives the current failover endpoints and any updates resulting from topology changes. It is disabled by default.

connectionTTL

The time in milliseconds after which the connection is failed if the server sends no ping packets. The default is 60000 (1 minute). -1 disables the timeout.

confirmationWindowSize

The size in bytes of the command replay buffer. This is used for automatic session re-attachment on reconnect. The default is -1, meaning no automatic re-attachment.

clientFailureCheckPeriod

The time in milliseconds between checks for dead connections. The default is 30000 (30 seconds). -1 disables checking.

5.5. FLOW CONTROL OPTIONS

For more information, see Chapter 8, *Flow control*.

consumerWindowSize

The size in bytes of the per-consumer message prefetch buffer. The default is 1048576 (1 MiB). -1 means no limit. 0 disables prefetching.

consumerMaxRate

The maximum number of messages to consume per second. The default is -1, meaning no limit.

producerWindowSize

The requested size in bytes for credit to produce more messages. This limits the total amount of data in flight at one time. The default is 1048576 (1 MiB). -1 means no limit.

producerMaxRate

The maximum number of messages to produce per second. The default is -1, meaning no limit.

5.6. LOAD BALANCING OPTIONS

useTopologyForLoadBalancing

If enabled, use the cluster topology for connection load balancing. It is enabled by default.

connectionLoadBalancingPolicyClassName

The class name of the connection load balancing policy. The default is **org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy**.

5.7. LARGE MESSAGE OPTIONS

The client can enable large message support by setting a value for the property **minLargeMessageSize**. Any message larger than **minLargeMessageSize** is considered a large message.

minLargeMessageSize

The minimum size in bytes at which a message is treated as a large message. The default is 102400 (100 KiB).

compressLargeMessage

If enabled, compress large messages, as defined by **minLargeMessageSize**. It is disabled by default.

NOTE

If the compressed size of a large message is less than the value of **minLargeMessageSize**, the message is sent as a regular message. Therefore, it is not written to the broker's large-message data directory.

5.8. THREADING OPTIONS

useGlobalPools

If enabled, use one pool of threads for all **ConnectionFactory** instances. Otherwise, use a separate pool for each instance. It is enabled by default.

threadPoolMaxSize

The maximum number of threads in the general thread pool. The default is -1, meaning no limit.

scheduledThreadPoolMaxSize

The maximum number of threads in the thread pool for scheduled operations. The default is 5.

CHAPTER 6. NETWORK CONNECTIONS

6.1. AUTOMATIC FAILOVER

A client can receive information about all master and slave brokers, so that in the event of a connection failure, it can reconnect to the slave broker. The slave broker then automatically re-creates any sessions and consumers that existed on each connection before failover. This feature saves you from having to hand-code manual reconnection logic in your applications.

When a session is recreated on the slave, it does not have any knowledge of messages already sent or acknowledged. Any in-flight sends or acknowledgements at the time of failover might also be lost. However, even without 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.

Clients detect connection failure when they have not received packets from the broker within a configurable period of time. See Section 6.3, "Detecting dead connections" for more information.

You have a number of methods to configure clients to receive information about master and slave. One option is to configure clients to connect to a specific broker and then receive information about the other brokers in the cluster. See Section 6.7, "Configuring static discovery" for more information. The most common way, however, is to use *broker discovery*. For details on how to configure broker discovery, see Section 6.6, "Configuring dynamic discovery".

Also, you can configure the client by adding parameters to the query string of the URI used to connect to the broker, as in the example below.

```
connectionFactory.ConnectionFactory=tcp://localhost:61616?ha=true&reconnectAttempts=3
```

Procedure

To configure your clients for failover through the use of a query string, ensure the following components of the URI are set properly:

1. The **host:port** portion of the URI must point to a master broker that is properly configured with a backup. This host and port is used only for the initial connection. The **host:port** value has nothing to do with the actual connection failover between a live and a backup server. In the example above, **localhost:61616** is used for the **host:port**.

2. (Optional) To use more than one broker as a possible initial connection, group the **host:port** entries as in the following example:

   ```
   connectionFactory.ConnectionFactory=(tcp://host1:port,tcp://host2:port)?
   ha=true&reconnectAttempts=3
   ```

3. Include the name-value pair **ha=true** as part of the query string to ensure the client receives information about each master and slave broker in the cluster.

4. Include the name-value pair **reconnectAttempts=n**, where **n** is an integer greater than 0. This parameter sets the number of times the client attempts to reconnect to a broker.

NOTE

Failover occurs only if **ha=true** and **reconnectAttempts** is greater than 0. Also, the client must make an initial connection to the master broker in order to receive information about other brokers. If the initial connection fails, the client can only retry to establish it. See Section 6.1.1, "Failing over during the initial connection" for more information.

6.1.1. Failing over during the initial connection

Because the client does not receive information about every broker until after the first connection to the HA cluster, there is a window of time where the client can connect only to the broker included in the connection URI. Therefore, if a failure happens during this initial connection, the client cannot failover to other master brokers, but can only try to re-establish the initial connection. Clients can be configured for a set number of reconnection attempts. Once the number of attempts has been made, an exception is thrown.

Setting the number of reconnection attempts
The examples below shows how to set the number of reconnection attempts to 3 using the AMQ Core Protocol JMS client. The default value is 0, that is, try only once.

Procedure

Set the number of reconnection attempts by passing a value to **ServerLocator.setInitialConnectAttempts()**.

```
ConnectionFactory cf = ActiveMQJMSClient.createConnectionFactory(...)
cf.setInitialConnectAttempts(3);
```

Setting a global number of reconnection attempts
Alternatively, you can apply a global value for the maximum number of reconnection attempts within the broker's configuration. The maximum is applied to all client connections.

Procedure

Edit **<broker-instance-dir>/etc/broker.xml** by adding the **initial-connect-attempts** configuration element and providing a value for the time-to-live, as in the example below.

```
<configuration>
 <core>
  ...
  <initial-connect-attempts>3</initial-connect-attempts> (1)
  ...
 </core>
</configuration>
```

(1) All clients connecting to the broker are allowed a maximum of three attempts to reconnect. The default is -1, which allows clients unlimited attempts.

6.1.2. Handling blocking calls during failover

When failover occurs and the client is waiting for a response from the broker to continue its execution, the newly created session does not have any knowledge of the call that was in progress. The initial call might otherwise hang forever, waiting for a response that never comes. To prevent this, the broker is designed to unblock any blocking calls that were in progress at the time of failover by making them throw an exception. Client code can catch these exceptions and retry any operations if desired.

When using AMQ Core Protocol JMS clients, if the unblocked method is a call to **commit()** or **prepare()**, the transaction is automatically rolled back and the broker throws an exception.

6.1.3. Handling failover with transactions

When using AMQ Core Protocol JMS clients, if the session is transactional and messages have already been sent or acknowledged in the current transaction, the broker cannot be sure that those messages or their acknowledgements were lost during the failover. Consequently, the transaction is marked for rollback only. Any subsequent attempt to commit it throws an **javax.jms.TransactionRolledBackException**.

WARNING

The caveat to this rule is when XA is used. If a two-phase commit is used and **prepare()** has already been called, rolling back could cause a **HeuristicMixedException**. Because of this, the commit throws an **XAException.XA_RETRY** exception, which informs the Transaction Manager it should retry the commit at some later point. If the original commit has not occurred, it still exists and can be committed. If the commit does not exist, it is assumed to have been committed, although the transaction manager might log a warning. A side effect of this exception is that any nonpersistent messages are lost. To avoid such losses, always use persistent messages when using XA. This is not an issue with acknowledgements since they are flushed to the broker before **prepare()** is called.

The AMQ Core Protocol JMS client code must catch the exception and perform any necessary client side rollback. There is no need to roll back the session, however, because it was already rolled back. The user can then retry the transactional operations again on the same session.

If failover occurs when a commit call is being executed, the broker unblocks the call to prevent the AMQ Core Protocol JMS client from waiting indefinitely for a response. Consequently, the client cannot determine whether the transaction commit was actually processed on the master broker before failure occurred.

To remedy this, the AMQ Core Protocol JMS client can enable duplicate detection in the transaction, and retry the transaction operations again after the call is unblocked. If the transaction was successfully committed on the master broker before failover, duplicate detection ensures that any durable messages present in the transaction when it is retried are ignored on the broker side. This prevents messages from being sent more than once.

If the session is non transactional, messages or acknowledgements can be lost in case of failover. If you want to provide *once and only once* delivery guarantees for non transacted sessions, enable duplicate detection and catch unblock exceptions.

6.1.4. Getting notified of connection failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: **java.jms.ExceptionListener**.

Any **ExceptionListener** or **SessionFailureListener** instance is always called by the broker if a connection failure occurs, whether the connection was successfully failed over, reconnected, or

reattached. You can find out if a reconnect or a reattach has happened by examining the **failedOver** flag passed in on the **connectionFailed** on **SessionFailureListener**. Alternatively, you can inspect the error code of the **javax.jms.JMSException**, which can be one of the following:

Table 6.1. JMSException error codes

Error code	Description
FAILOVER	Failover has occurred and the broker has successfully reattached or reconnected
DISCONNECT	No failover has occurred and the broker is disconnected

6.2. APPLICATION-LEVEL FAILOVER

In some cases you might not want automatic client failover, but prefer to code your own reconnection logic in a failure handler instead. This is known as *application-level* failover, since the failover is handled at the application level.

To implement application-level failover when using JMS, set an **ExceptionListener** class on the JMS connection. The **ExceptionListener** is called by the broker in the event that a connection failure is detected. In your **ExceptionListener**, you should close your old JMS connections. You might also want to look up new connection factory instances from JNDI and create new connections.

6.3. DETECTING DEAD CONNECTIONS

As long as the it is receiving data from the broker, the client considers a connection to be alive. Configure the client to check its connection for failure by providing a value for the **client-failure-check-period** property. The default check period for a network connection is 30,000 milliseconds, or 30 seconds, while the default value for an in-VM connection is -1, which means the client never fails the connection from its side if no data is received.

Typically, you set the check period to be much lower than the value used for the broker's connection time-to-live, which ensures that clients can reconnect in case of a temporary failure.

Setting the check period for detecting dead connections

The examples below show how to set the check period to 10,000 milliseconds.

Procedure

- If you are using JNDI, set the check period within the JNDI context environment, **jndi.properties**, for example, as below.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?
clientFailureCheckPeriod=10000
```

- If you are not using JNDI, set the check period directly by passing a value to **ActiveMQConnectionFactory.setClientFailureCheckPeriod()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setClientFailureCheckPeriod(10000);
```

6.4. CONFIGURING TIME-TO-LIVE

By default clients can set a time-to-live (TTL) for their own connections. The examples below show you how to set the TTL.

Procedure

- If you are using JNDI to instantiate your connection factory, you can specify it in the xml config, using the parameter **connectionTtl**.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?connectionTtl=30000
```

- If you are not using JNDI, the connection TTL is defined by the **ConnectionTTL** attribute on a **ActiveMQConnectionFactory** instance.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConnectionTTL(30000);
```

6.5. CLOSING CONNECTIONS

A client application must close its resources in a controlled manner before it exits to prevent dead connections from occurring. In Java, it is recommended to close connections inside a **finally** block:

```
Connection jmsConnection = null;
try {
   ConnectionFactory jmsConnectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(...);
   jmsConnection = jmsConnectionFactory.createConnection();
   ...use the connection...
}
finally {
   if (jmsConnection != null) {
      jmsConnection.close();
   }
}
```

6.6. CONFIGURING DYNAMIC DISCOVERY

You can configure AMQ Core Protocol JMS to discover a list of brokers when attempting to establish a connection.

If you are using JNDI on the client to look up your JMS connection factory instances, you can specify these parameters in the JNDI context environment. Typically the parameters are defined in a file named **jndi.properties**. The host and part in the URI for the connection factory should match the **group-address** and **group-port** from the corresponding **broadcast-group** inside broker's **broker.xml** configuration file. Below is an example of a **jndi.properties** file configured to connect to a broker's discovery group.

```
java.naming.factory.initial = ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=udp://231.7.7.7:9876
```

When this connection factory is downloaded from JNDI by a client application and JMS connections are

created from it, those connections will be load-balanced across the list of servers that the discovery group maintains by listening on the multicast address specified in the broker's discovery group configuration.

As an alternative to using JNDI, you can use specify the discovery group parameters directly in your Java code when creating the JMS connection factory. The code below provides an example of how to do this.

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;

DiscoveryGroupConfiguration discoveryGroupConfiguration = new DiscoveryGroupConfiguration();
UDPBroadcastEndpointFactory udpBroadcastEndpointFactory = new
UDPBroadcastEndpointFactory();
udpBroadcastEndpointFactory.setGroupAddress(groupAddress).setGroupPort(groupPort);
discoveryGroupConfiguration.setBroadcastEndpointFactory(udpBroadcastEndpointFactory);

ConnectionFactory jmsConnectionFactory = ActiveMQJMSClient.createConnectionFactoryWithHA
   (discoveryGroupConfiguration, JMSFactoryType.CF);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();
Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

The refresh timeout can be set directly on the **DiscoveryGroupConfiguration** by using the setter method **setRefreshTimeout()**. The default value is 10000 milliseconds.

On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default wait time is 10000 milliseconds, but you can change it by passing a new value to **DiscoveryGroupConfiguration.setDiscoveryInitialWaitTimeout()**.

6.7. CONFIGURING STATIC DISCOVERY

Sometimes it may be impossible to use UDP on the network you are using. In this case you can configure a connection with an initial list of possible servers. The list can be just one broker that you know will always be available, or a list of brokers where at least one will be available.

This does not 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. After they are connected, their connection details will be propagated from the server to the client.

If you are using JNDI on the client to look up your JMS connection factory instances, you can specify these parameters in the JNDI context environment. Typically the parameters are defined in a file named **jndi.properties**. Below is an example **jndi.properties** file that provides a static list of brokers instead of using dynamic discovery.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=(tcp://myhost:61616,tcp://myhost2:61616)
```

When the above connection factory is used by a client, its connections will be load-balanced across the list of brokers defined within the parentheses **()**.

If you are instantiating the JMS connection factory directly, you can specify the connector list explicitly when creating the JMS connection factory, as in the example below.

```
HashMap<String, Object> map = new HashMap<String, Object>();
map.put("host", "myhost");
```

```
map.put("port", "61616");
TransportConfiguration broker1 = new TransportConfiguration
    (NettyConnectorFactory.class.getName(), map);

HashMap<String, Object> map2 = new HashMap<String, Object>();
map2.put("host", "myhost2");
map2.put("port", "61617");
TransportConfiguration broker2 = new TransportConfiguration
    (NettyConnectorFactory.class.getName(), map2);

ActiveMQConnectionFactory cf = ActiveMQJMSClient.createConnectionFactoryWithHA
    (JMSFactoryType.CF, broker1, broker2);
```

6.8. CONFIGURING A BROKER CONNECTOR

Connectors define how clients can connect to the broker. You can configure them from the client using the JMS connection factory.

```
Map<String, Object> connectionParams = new HashMap<String, Object>();

connectionParams.put(org.apache.activemq.artemis.core.remoting.impl.netty.TransportConstants.POR
T_PROP_NAME, 61617);

TransportConfiguration transportConfiguration =
    new TransportConfiguration(
    "org.apache.activemq.artemis.core.remoting.impl.netty.NettyConnectorFactory",
connectionParams);

ConnectionFactory connectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(JMSFactoryType.CF,
transportConfiguration);

Connection jmsConnection = connectionFactory.createConnection();
```

CHAPTER 7. MESSAGE DELIVERY

7.1. WRITING TO A STREAMED LARGE MESSAGE

To write to a large message, use the **BytesMessage.writeBytes()** method. The following example reads bytes from a file and writes them to a message:

Example: Writing to a streamed large message

```
BytesMessage message = session.createBytesMessage();
File inputFile = new File(inputFilePath);
InputStream inputStream = new FileInputStream(inputFile);

int numRead;
byte[] buffer = new byte[1024];

while ((numRead = inputStream.read(buffer, 0, buffer.length)) != -1) {
   message.writeBytes(buffer, 0, numRead);
}
```

7.2. READING FROM A STREAMED LARGE MESSAGE

To read from a large message, use the **BytesMessage.readBytes()** method. The following example reads bytes from a message and writes them to a file:

Example: Reading from a streamed large message

```
BytesMessage message = (BytesMessage) consumer.receive();
File outputFile = new File(outputFilePath);
OutputStream outputStream = new FileOutputStream(outputFile);

int numRead;
byte buffer[] = new byte[1024];

for (int pos = 0; pos < message.getBodyLength(); pos += buffer.length) {
   numRead = message.readBytes(buffer);
   outputStream.write(buffer, 0, numRead);
}
```

7.3. USING MESSAGE GROUPS

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group ID. That is, they have same group identifier property. For JMS messages, the property is **JMSXGroupID**.
- Messages in a message group are always consumed by the same consumer, even if there are many consumers on a queue. Another consumer is chosen to receive a message group if the original consumer is closed.

Message groups are useful when you want all messages for a certain value of the property to be processed serially by the same consumer. For example, you may want orders for any particular stock purchase to be processed serially by the same consumer. To do this, you could create a pool of

consumers and then set the stock name as the value of the message property. This ensures that all messages for a particular stock are always processed by the same consumer.

Setting the group ID

The examples below show how to use message groups with AMQ Core Protocol JMS.

Procedure

- If you are using JNDI to establish a JMS connection factory for your JMS client, add the **groupID** parameter and supply a value. All messages sent using this connection factory have the property **JMSXGroupID** set to the specified value.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?groupID=MyGroup
```

- If you are not using JNDI, set the **JMSXGroupID** property using the **setStringProperty()** method.

```
Message message = new TextMessage();
message.setStringProperty("JMSXGroupID", "MyGroup");
producer.send(message);
```

Additional resources

See **message-group** and **message-group2** under ***<install-dir>*/examples/features/standard** for working examples of how message groups are configured and used.

7.4. USING DUPLICATE MESSAGE DETECTION

AMQ Broker includes automatic duplicate message detection, which filters out any duplicate messages it receives so you do not have to code your own duplicate detection logic.

To enable duplicate message detection, provide a unique value for the message property **_AMQ_DUPL_ID**. When a broker receives a message, it checks if **_AMQ_DUPL_ID** has a value. If it does, the broker then checks in its memory cache to see if it has already received a message with that value. If a message with the same value is found, the incoming message is ignored.

If you are sending messages in a transaction, you do not have to set **_AMQ_DUPL_ID** for every message in the transaction, but only in one of them. If the broker detects a duplicate message for any message in the transaction, it ignores the entire transaction.

Setting the duplicate ID message property

The following example shows how to set the duplicate detection property using AMQ Core Protocol JMS. Note that for convenience, the clients use the value of the constant **org.apache.activemq.artemis.api.core.Message.HDR_DUPLICATE_DETECTION_ID** for the name of the duplicate ID property, **_AMQ_DUPL_ID**.

Procedure

Set the value for **_AMQ_DUPL_ID** to a unique string value.

```
Message jmsMessage = session.createMessage();
String myUniqueID = "This is my unique id";
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
```

7.5. USING MESSAGE INTERCEPTORS

With AMQ Core Protocol JMS you can intercept packets entering or exiting the client, allowing you to audit packets or filter messages. Interceptors can change the packets that they intercept. This makes interceptors powerful, but also a feature that you should use with caution.

Interceptors must implement the **intercept()** method, which returns a **boolean** value. If the returned value is **true**, the message packet continues onward. If the returned value is **false**, the process is aborted, no other interceptors are called, and the message packet is not processed further.

Message interception occurs transparently to the main client code except when an outgoing packet is sent in blocking send mode. When an outgoing packet is sent with blocking enabled and that packet encounters an interceptor that returns **false**, an ActiveMQException is thrown to the caller. The thrown exception contains the name of the interceptor.

Your interceptor must implement the **org.apache.artemis.activemq.api.core.Interceptor** interface. The client interceptor classes and their dependencies must be added to the Java classpath of the client to be properly instantiated and invoked.

```
package com.example;

import org.apache.artemis.activemq.api.core.Interceptor;
import org.apache.activemq.artemis.core.protocol.core.Packet;
import org.apache.activemq.artemis.spi.core.protocol.RemotingConnection;

public class MyInterceptor implements Interceptor {
   private final int ACCEPTABLE_SIZE = 1024;

   @Override
   boolean intercept(Packet packet, RemotingConnection connection) throws ActiveMQException {
      int size = packet.getPacketSize();
      if (size <= ACCEPTABLE_SIZE) {
         System.out.println("This Packet has an acceptable size.");
         return true;
      }
      return false;
   }
}
```

CHAPTER 8. FLOW CONTROL

Flow control prevents producers and consumers from becoming overburdened by limiting the flow of data between them. AMQ Core Protocol JMS allows you to configure flow control for both consumers and producers.

Consumer flow control

Consumer flow control regulates the flow of data between the broker and the client as the client consumes messages from the broker. AMQ Core Protocol JMS buffers messages by default before delivering them to consumers. Without a buffer, the client would first need to request each message from the broker before consuming it. This type of "round-trip" communication is costly. Regulating the flow of data on the client side is important because out of memory issues can result when a consumer cannot process messages quickly enough and the buffer begins to overflow with incoming messages.

Producer flow control

In a similar way to consumer window-based flow control, the client can limit the amount of data sent from a producer to a broker to prevent the broker from being overburdened with too much data. In the case of a producer, the window size determines the number of bytes that can be in flight at any one time.

8.1. SETTING THE CONSUMER WINDOW SIZE

The maximum size of messages held in the client-side buffer is determined by its *window size*. The default size of the window for AMQ Core Protocol JMS is 1 MiB, or 1024 * 1024 bytes. The default is fine for most use cases. For other cases, finding the optimal value for the window size might require benchmarking your system. AMQ Core Protocol JMS allows you to set the buffer window size if you need to change the default.

The following examples show how to set the consumer window size parameter when using AMQ Core Protocol JMS. Each example sets the consumer window size to 300,000 bytes.

Procedure

- If the client uses JNDI to instantiate its connection factory, include the **consumerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?
consumerWindowSize=300000
```

- If the client does not use JNDI to instantiate its connection factory, pass a value to **ActiveMQConnectionFactory.setConsumerWindowSize()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(300000);
```

8.2. SETTING THE PRODUCER WINDOW SIZE

The window size is negotiated between the broker and producer on the basis of credits, one credit for each byte in the window. As messages are sent and credits are used, the producer must request, and be granted, credits from the broker before it can send more messages. The exchange of credits between producer and broker regulates the flow of data between them.

The following examples show how to set the producer window size to 1024 bytes when using AMQ Core Protocol JMS.

Procedure

- If the client uses JNDI to instantiate its connection factory, include the **producerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616?producerWindowSize=1024
```

- If the client does not use JNDI to instantiate its connection factory, pass the value to **ActiveMQConnectionFactory.setProducerWindowSize()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setProducerWindowSize(1024);
```

8.3. HANDLING FAST CONSUMERS

Fast consumers can process messages as fast as they consume them. If you are confident that the consumers in your messaging system are that fast, consider setting the window size to -1. Setting the window size to this value allows unbounded message buffering on the client. Use this setting with caution, however. Memory on the client can overflow if the consumer is not able to process messages as fast as it receives them.

Setting the window size for fast consumers

The examples below show how to set the window size to -1 when using a AMQ Core Protocol JMS client that is a fast consumer of messages.

Procedure

- If the client uses JNDI to instantiate its connection factory, include the **consumerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?consumerWindowSize=-1
```

- If the client does not use JNDI to instantiate its connection factory, pass a value to **ActiveMQConnectionFactory.setConsumerWindowSize()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(-1);
```

8.4. HANDLING SLOW CONSUMERS

Slow consumers take significant time to process each message. In these cases, buffering messages on the client is not recommended. Messages remain on the broker ready to be consumed by other consumers instead. One benefit of turning off the buffer is that it provides deterministic distribution between multiple consumers on a queue. To handle slow consumers by disabling the client-side buffer, set the window size to 0.

Setting the window size for slow consumers

The examples below show how to set the window size to 0 when using a AMQ Core Protocol JMS client that is a slow consumer of messages.

Procedure

- If the client uses JNDI to instantiate its connection factory, include the **consumerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?consumerWindowSize=0
```

- If the client does not use JNDI to instantiate its connection factory, pass a value to **ActiveMQConnectionFactory.setConsumerWindowSize()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(0);
```

Additional resources

See the example **no-consumer-buffering** in ***<install-dir>*/examples/standard** for an example that shows how to configure the broker to prevent consumer buffering when dealing with slow consumers.

8.5. SETTING THE RATE OF MESSAGE CONSUMPTION

You can regulate the rate at which a consumer can consume messages. Also known as *throttling*, regulating the rate of consumption ensures that a consumer never consumes messages at a rate faster than configuration allows.

NOTE

Rate-limited flow control can be used in conjunction with window-based flow control. Rate-limited flow control affects only how many messages a client can consume per second and not how many messages are in its buffer. With a slow rate limit and a high window-based limit, the internal buffer of the client fills up with messages quickly.

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 the rate to -1 disables rate-limited flow control. The default value is -1.

The examples below show a client that limits the rate of consuming messages to 10 messages per second.

Procedure

- If the client uses JNDI to instantiate its connection factory, include the **consumerMaxRate** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616?consumerMaxRate=10
```

- If the client does not use JNDI to instantiate its connection factory, pass the value to **ActiveMQConnectionFactory.setConsumerMaxRate()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerMaxRate(10);
```

Additional resources

See the **consumer-rate-limit** example in ***<install-dir>*/examples/standard** for a working example of how to limit the consumer rate.

8.6. SETTING THE RATE OF MESSAGE PRODUCTION

AMQ Core Protocol JMS can also limit the rate at which a producer sends messages. The producer rate is specified in units of messages per second. Setting it to -1, the default, disables rate-limited flow control.

The examples below show how to set the rate of sending messages when the producer is using AMQ Core Protocol JMS. Each example sets the maximum rate to 10 messages per second.

Procedure

- If the client uses JNDI to instantiate its connection factory, include the **producerMaxRate** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:61616?producerMaxRate=10
```

- If the client does not use JNDI to instantiate its connection factory, pass the value to **ActiveMQConnectionFactory.setProducerMaxRate()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setProducerMaxRate(10);
```

Additional resources

See the **producer-rate-limit** example in ***<install-dir>*/examples/standard** for a working example of how to limit a the rate of sending messages.

APPENDIX A. USING YOUR SUBSCRIPTION

AMQ is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

A.1. ACCESSING YOUR ACCOUNT

Procedure

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

A.2. ACTIVATING A SUBSCRIPTION

Procedure

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

A.3. DOWNLOADING RELEASE FILES

To access .zip, .tar.gz, and other release files, use the customer portal to find the relevant files for download. If you are using RPM packages or the Red Hat Maven repository, this step is not required.

Procedure

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **Red Hat AMQ** entries in the **INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

APPENDIX B. USING RED HAT MAVEN REPOSITORIES

This section describes how to use Red Hat-provided Maven repositories in your software.

B.1. USING THE ONLINE REPOSITORY

Red Hat maintains a central Maven repository for use with your Maven-based projects. For more information, see the repository welcome page.

There are two ways to configure Maven to use the Red Hat repository:

- Add the repository to your Maven settings
- Add the repository to your POM file

Adding the repository to your Maven settings

This method of configuration applies to all Maven projects owned by your user, as long as your POM file does not override the repository configuration and the included profile is enabled.

Procedure

1. Locate the Maven **settings.xml** file. It is usually inside the **.m2** directory in the user home directory. If the file does not exist, use a text editor to create it.
   On Linux or UNIX:

   ```
   /home/<username>/.m2/settings.xml
   ```

   On Windows:

   ```
   C:\Users\<username>\.m2\settings.xml
   ```

2. Add a new profile containing the Red Hat repository to the **profiles** element of the **settings.xml** file, as in the following example:

   Example: A Maven settings.xml file containing the Red Hat repository

```
<settings>
  <profiles>
    <profile>
      <id>red-hat</id>
      <repositories>
        <repository>
          <id>red-hat-ga</id>
          <url>https://maven.repository.redhat.com/ga</url>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>red-hat-ga</id>
          <url>https://maven.repository.redhat.com/ga</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
```

```
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
 </profile>
</profiles>
<activeProfiles>
 <activeProfile>red-hat</activeProfile>
</activeProfiles>
</settings>
```

For more information about Maven configuration, see the Maven settings reference.

Adding the repository to your POM file

To configure a repository directly in your project, add a new entry to the **repositories** element of your POM file, as in the following example:

Example: A Maven pom.xml file containing the Red Hat repository

```
<project>
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.example</groupId>
 <artifactId>example-app</artifactId>
 <version>1.0.0</version>

 <repositories>
  <repository>
   <id>red-hat-ga</id>
   <url>https://maven.repository.redhat.com/ga</url>
  </repository>
 </repositories>
</project>
```

For more information about POM file configuration, see the Maven POM reference.

B.2. USING A LOCAL REPOSITORY

Red Hat provides file-based Maven repositories for some of its components. These are delivered as downloadable archives that you can extract to your local filesystem.

To configure Maven to use a locally extracted repository, apply the following XML in your Maven settings or POM file:

```
<repository>
 <id>red-hat-local</id>
 <url>${repository-url}</url>
</repository>
```

${repository-url} must be a file URL containing the local filesystem path of the extracted repository.

Table B.1. Example URLs for local Maven repositories

Operating system	Filesystem path	URL
Linux or UNIX	**/home/alice/maven-repository**	**file:/home/alice/maven-repository**
Windows	**C:\repos\red-hat**	**file:C:\repos\red-hat**

APPENDIX C. USING AMQ BROKER WITH THE EXAMPLES

The AMQ Core Protocol JMS examples require a running message broker with a queue named **exampleQueue**. Use the procedures below to install and start the broker and define the queue.

C.1. INSTALLING THE BROKER

Follow the instructions in *Getting Started with AMQ Broker* to install the broker and create a broker instance. Enable anonymous access.

The following procedures refer to the location of the broker instance as ***<broker-instance-dir>***.

C.2. STARTING THE BROKER

Procedure

1. Use the **artemis run** command to start the broker.

```
$ <broker-instance-dir>/bin/artemis run
```

2. Check the console output for any critical errors logged during startup. The broker logs **Server is now live** when it is ready.

```
$ example-broker/bin/artemis run

        __  __  ____    ____           _
       /\   |  \/  |/ __ \  |  _ \         | |
      /  \  | \  / | |  | | | |_) |_ __ ___ | | _____ _ __
     / /\ \ | |\/| | |  | | |  _ <| '__/ _ \| |/ / _ \ '__|
    / ____ \| |  | | |__| | | |_) | | | (_) |   <  __/ |
   /_/    \_\_|  |_|\____\_\ |____/|_|  \___/|_|\_\___|_|

    Red Hat AMQ <version>

2020-06-03 12:12:11,807 INFO  [org.apache.activemq.artemis.integration.bootstrap]
AMQ101000: Starting ActiveMQ Artemis Server
...
2020-06-03 12:12:12,336 INFO  [org.apache.activemq.artemis.core.server] AMQ221007:
Server is now live
...
```

C.3. CREATING A QUEUE

In a new terminal, use the **artemis queue** command to create a queue named **exampleQueue**.

```
$ <broker-instance-dir>/bin/artemis queue create --name exampleQueue --address exampleQueue --auto-create-address --anycast
```

You are prompted to answer a series of yes or no questions. Answer **N** for no to all of them.

Once the queue is created, the broker is ready for use with the example programs.

C.4. STOPPING THE BROKER

When you are done running the examples, use the **artemis stop** command to stop the broker.

```
$ <broker-instance-dir>/bin/artemis stop
```

Revised on 2024-01-22 11:05:17 UTC