Abstract

This guide describes how to install and configure the client, run hands-on examples, and use your client with other AMQ components.
Table of Contents

CHAPTER 1. OVERVIEW ........................................................................................................... 4
  1.1. KEY FEATURES ............................................................................................................. 4
  1.2. SUPPORTED STANDARDS AND PROTOCOLS ....................................................... 4
  1.3. SUPPORTED CONFIGURATIONS ................................................................................. 4
  1.4. TERMS AND CONCEPTS ............................................................................................. 5
  1.5. DOCUMENT CONVENTIONS ....................................................................................... 6
      The sudo Command ........................................................................................................... 6
      About the Use of File Paths in this Document ............................................................... 6

CHAPTER 2. INSTALLATION .................................................................................................. 7
  2.1. PREREQUISITES ........................................................................................................... 7
  2.2. INSTALLING ON RED HAT ENTERPRISE LINUX ................................................... 7
  2.3. INSTALLING ON MICROSOFT WINDOWS ................................................................... 7
  2.4. CONFIGURING MAVEN ............................................................................................... 7

CHAPTER 3. GETTING STARTED ............................................................................................ 11
  3.1. PREPARING THE BROKER .......................................................................................... 11
  3.2. RUNNING HELLO WORLD .......................................................................................... 11

CHAPTER 4. CONFIGURATION .............................................................................................. 13
  4.1. CONFIGURING A JNDI INITIALCONTEXT .................................................................. 13
      Configuring an InitialContext Using a jndi.properties File ........................................... 13
      Configuring an InitialContext Using System Properties .............................................. 13
      Configuring an InitialContext Programmatically ......................................................... 13
      JNDI Property Syntax ..................................................................................................... 14
  4.2. CONNECTION URIS ..................................................................................................... 14
  4.3. CONNECTION URI OPTIONS ...................................................................................... 15
      4.3.1. JMS Options .......................................................................................................... 15
            Prefetch Policy Options ............................................................................................... 16
            Redelivery Policy Options ......................................................................................... 16
            Message ID Policy Options ......................................................................................... 16
            Presettle Policy Options ............................................................................................ 16
            Deserialization Policy Options .................................................................................. 17
      4.3.2. TCP Transport Options ......................................................................................... 17
      4.3.3. SSL/TLS Transport Options .................................................................................... 18
      4.3.4. AMQP Options ....................................................................................................... 19
      4.3.5. Failover Options ..................................................................................................... 20
      4.3.6. Discovery Options .................................................................................................. 21
  4.4. SECURITY ..................................................................................................................... 22
      4.4.1. Authenticating Using Kerberos .............................................................................. 22
  4.5. LOGGING ...................................................................................................................... 23

CHAPTER 5. EXAMPLES .......................................................................................................... 24
  5.1. CONFIGURING THE JNDI CONTEXT ......................................................................... 24
  5.2. SENDING MESSAGES .................................................................................................. 24
  5.3. RECEIVING MESSAGES .............................................................................................. 26

CHAPTER 6. INTEROPERABILITY ............................................................................................ 29
  6.1. INTEROPERATING WITH OTHER AMQP CLIENTS ................................................... 29
      6.1.1. Sending Messages ................................................................................................. 29
            6.1.1.1. Message Type ................................................................................................ 29
            6.1.1.2. Message Properties ....................................................................................... 30
      6.1.2. Receiving Messages .............................................................................................. 30
6.1.2.1. Message Type
6.1.2.2. Message Properties
6.2. CONNECTING TO AMQ BROKER
6.3. CONNECTING TO AMQ INTERCONNECT

APPENDIX A. USING YOUR SUBSCRIPTION
Accessing Your Account
Activating a Subscription
Downloading Zip and Tar Files
Registering Your System for Packages
CHAPTER 1. OVERVIEW

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

AMQ JMS is part of AMQ Clients, a suite of messaging libraries supporting multiple languages and platforms. See Introducing Red Hat JBoss AMQ 7 for an overview of the clients and other AMQ components. See AMQ Clients 1.2 Release Notes for information about this release.

AMQ JMS is based on the JMS client from Apache Qpid.

1.1. KEY FEATURES

AMQ JMS gives you all of the features of JMS with the power of AMQP.

- JMS 2.0 compatible
- SSL/TLS and SASL for secure communication
- Heartbeating and automatic reconnect for reliable network connections
- Ready for use with OSGi containers
- Pure-Java implementation

1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ JMS supports the following industry-recognized standards and network protocols.

- Version 2.0 of the Java Message Service API
- Version 1.0 of the Advanced Message Queueing Protocol (AMQP)
- Version 1.0 of the AMQP JMS Mapping
- Modern TCP with IPv6

NOTE

The details of distributed transactions (XA) within AMQP are not provided in the 1.0 version of the specification. If your environment requires support for distributed transactions, it is recommended that you use the AMQ Core Protocol JMS.

1.3. SUPPORTED CONFIGURATIONS

AMQ JMS supports the following OS and language versions. See Red Hat JBoss AMQ 7 Supported Configurations for more information.

- Red Hat Enterprise Linux 6 and 7 with the following JDKs.
  - OpenJDK 8
  - Oracle JDK 8
  - IBM JDK 8
1.4. TERMS AND CONCEPTS

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

Table 1.1. API Terms

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactor</td>
<td>An entry point for creating connections</td>
</tr>
<tr>
<td>Connection</td>
<td>A channel for communication between two peers on a network</td>
</tr>
<tr>
<td>Session</td>
<td>A serialized context for producing and consuming messages</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>A channel for sending messages to a destination</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>A channel for receiving messages from a destination</td>
</tr>
<tr>
<td>Destination</td>
<td>A named location for messages, either a queue or a topic</td>
</tr>
<tr>
<td>Queue</td>
<td>A stored sequence of messages</td>
</tr>
<tr>
<td>Topic</td>
<td>A stored sequence of messages for multicast distribution</td>
</tr>
<tr>
<td>Message</td>
<td>A mutable holder of application content</td>
</tr>
</tbody>
</table>

AMQ 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

This document uses the following conventions for the `sudo` command and file paths.

**The `sudo` Command**

In this document, `sudo` is used for any command that requires root privileges. You should always exercise caution when using `sudo`, as any changes can affect the entire system.

For more information about using `sudo`, see The `sudo` Command.

**About the Use of File Paths in this Document**

In this document, all file paths are valid for Linux, UNIX, and similar operating systems (for example, `/home/...`). If you are using Microsoft Windows, you should use the equivalent Microsoft Windows paths (for example, `C:\Users\...`).
CHAPTER 2. INSTALLATION

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

2.1. PREREQUISITES

To begin installation, use your subscription to access AMQ distribution archives and package repositories.

To compile the examples or your own application, make sure you have Apache Maven installed.

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

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 JBoss AMQ Clients entry in the JBOSS INTEGRATION AND AUTOMATION category.


4. Download the AMQ JMS Client zip file.

5. Use the unzip command to extract the file contents into a directory of your choosing.

   $ unzip apache-qpid-jms-VERSION.zip

6. Configure Maven to discover the client.

2.3. INSTALLING ON MICROSOFT WINDOWS

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 JBoss AMQ Clients entry in the JBOSS INTEGRATION AND AUTOMATION category.


4. Download the AMQ JMS Client zip file.

5. Extract the file contents into a directory of your choosing by right-clicking on the zip file and selecting Extract All.

6. Configure Maven to discover the client.

2.4. CONFIGURING MAVEN

The client uses Apache Maven as its build tool.

To build and run the client examples, or to use the client in dependent Maven application builds, you must first configure Maven to discover a repository for the client. To do so, you must update or create the Maven settings.xml file. It is typically located at one of the following locations.
Table 2.1. File Location

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Location of File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td><code>/home/USERNAME/.m2/settings.xml</code></td>
</tr>
<tr>
<td>Windows</td>
<td><code>C:\Users\USERNAME\m2\settings.xml</code></td>
</tr>
</tbody>
</table>

The client can be accessed by configuring Maven in one of two ways depending on your needs:

1. Using the JBoss Enterprise Maven Repository

The AMQ JMS Client 1.2 can be used from the JBoss Enterprise Maven Repository.

The contents of a `settings.xml` configured to use the repository would resemble the following example.

```xml
<settings>
  <profiles>
    <!-- Configure the JBoss GA Maven repository -->
    <profile>
      <id>jboss-ga-repository</id>
      <repositories>
        <repository>
          <id>jboss-ga-repository</id>
          <url>https://maven.repository.redhat.com/ga</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-ga-repository</id>
          <url>http://maven.repository.redhat.com/ga</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
    <!-- Configure the JBoss Early Access Maven repository -->
    <profile>
      <id>jboss-earlyaccess-repository</id>
      <repositories>
        <repository>
          <id>jboss-earlyaccess-repository</id>
          <url>https://maven.repository.redhat.com/earlyaccess/all</url>
        </repository>
      </repositories>
    </profile>
  </profiles>
</settings>
```
2. Using a File-based Maven Repository

Alternatively, you can download the AMQ JMS Client 1.2 Maven Repository zip file from the Red Hat Customer Portal, extract it, and configure Maven to utilize this locally instead.

The contents of a settings.xml configured to use the file-based repository should resemble the following example. Remember to update the URL to the actual path of the extracted repository.

```xml
<settings>
  <profiles>
  <!-- Optionally, make the repository active by default -->
  <activeProfile>jboss-ga-repository</activeProfile>
  <activeProfile>jboss-earlyaccess-repository</activeProfile>
  </activeProfiles>

  <profiles>
  <!-- Configure the File Based Maven repository -->
  <profile>
    <id>amq-jms-client</id>
    <repositories>
      <repository>
        <id>amq-qpid-jms</id>
        <url>file:///path/to/extracted/maven-repository</url>
      </repository>
    </repositories>
  </profile>
  </profiles>

  <repositories>
    <repository>
      <id>amq-qpid-jms</id>
      <url>file:///path/to/extracted/maven-repository</url>
    </repository>
  </repositories>

  <releases>
    <enabled>true</enabled>
  </releases>

  <snapshots>
    <enabled>false</enabled>
  </snapshots>

  <repository/>
  </pluginRepositories>
  </repositories>
  </profile>
  </profiles>
</settings>
```
<repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-amq-repository</id>
      <url>file:///path/to/extracted/maven-repository</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profiles>
<activeProfiles>
  <!-- Optionally, make the repository active by default -->
  <activeProfile>amq-jms-client</activeProfile>
</activeProfiles>
</settings>
CHAPTER 3. GETTING STARTED

This chapter guides you through a simple exercise to help you get started using AMQ JMS. Before starting, make sure you have completed the steps in the Chapter 2, Installation chapter for your environment.

To build the examples, you must first configure Maven to discover the client repository.

3.1. PREPARING THE BROKER

The example programs require a running broker with a queue named queue. Follow these steps to define the queue and start the broker.

1. Install the broker.

2. Create a broker instance. Enable anonymous access.

3. Start the broker instance and check the console for any critical errors logged during startup.

   
   ```
   $ BROKER_INSTANCE_DIR/bin/artemis run
   [...] 
   [...] 
   15:01:39,686 INFO [org.apache.activemq.artemis.core.server] AMQ221020: Started Acceptor at 0.0.0.0:5672 for protocols [AMQP]
   [...] 
   15:01:39,691 INFO [org.apache.activemq.artemis.core.server] AMQ221007: Server is now live
   ```

4. Use the **artemis queue** command to create a queue called **queue**.

   
   ```
   $ BROKER_INSTANCE_DIR/bin/artemis queue create --name queue --auto-create-address --anycast
   ```

   You are prompted to answer a series of questions. For yes|no questions, type N; otherwise, press Enter to accept the default value.

3.2. RUNNING HELLO WORLD

Use Maven to build the examples by running the following command in the **INSTALL_DIR/examples** directory.

```
NOTE

In this example, the addition of dependency:copy-dependencies results in the dependencies being copied into the target/dependency directory.

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

Run the **HelloWorld** example by using one of the following commands.
Linux:   java -cp "target/classes/:target/dependency/**" org.apache.qpid.jms.example.HelloWorld
Windows: java -cp "target\classes\;target\dependency\**" org.apache.qpid.jms.example.HelloWorld

The **HelloWorld** example creates a connection to the broker, creates a **MessageConsumer** and **MessageProducer** for the queue called **queue**, sends a **Hello world!** **TextMessage**, receives it, and prints its contents to the terminal.

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

```
$ java -cp "target/classes/:target/dependency/**"
org.apache.qpid.jms.example.HelloWorld
Hello world!
```

The source code for the example can be found in the **INSTALL_DIR/src/main/java** directory, with the JNDI and logging configuration found in the **INSTALL_DIR/src/main/resources** directory.
CHAPTER 4. CONFIGURATION

This section details various configuration options for the client, such as how to configure and create a JNDI InitialContext, the syntax for its related configuration, and various URI options that can be set when defining a ConnectionFactory.

4.1. CONFIGURING A JNDI INITIALCONTEXT

JMS applications use a JNDI InitialContext obtained from an InitialContextFactory to look up JMS objects such as ConnectionFactory. The client provides an implementation of the InitialContextFactory in the org.apache.qpid.jms.jndi.JmsInitialContextFactory class. You can configure it three different ways.

Configuring an InitialContext Using a jndi.properties File
If you include a file named jndi.properties on the classpath and set the java.naming.factory.initial property value to org.apache.qpid.jms.jndi.JmsInitialContextFactory, the client InitialContextFactory implementation is discovered when the InitialContext object is instantiated.

```java
javax.naming.Context ctx = new javax.naming.InitialContext();
```

The particular ConnectionFactory, Queue, and Topic objects that you want the Context to contain are configured as properties either directly within the jndi.properties file or in a separate file whose path is referenced in jndi.properties using the java.naming.provider.url property. The syntax for these properties is detailed below.

Configuring an InitialContext Using System Properties
If you set the java.naming.factory.initial system property to the value org.apache.qpid.jms.jndi.JmsInitialContextFactory, the client InitialContextFactory implementation is discovered when the InitialContext object is instantiated.

```java
javax.naming.Context ctx = new javax.naming.InitialContext();
```

The particular ConnectionFactory, Queue, and Topic objects that you want the context to contain are configured as properties in a file, the path to which is passed using the java.naming.provider.url system property. The syntax for these properties is detailed below.

Configuring an InitialContext Programmatically
You can configure the InitialContext directly by setting an environment variable on a Hashtable environment object.

```java
Hashtable<Object, Object> env = new Hashtable<Object, Object>();
env.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
javax.naming.Context context = new javax.naming.InitialContext(env);
```

The particular ConnectionFactory, Queue, and Topic objects that you want the context to contain are configured as properties (the syntax for which is detailed below) either directly within the environment Hashtable or in a separate file whose path is referenced using the java.naming.provider.url property within the environment Hashtable.
**JNDI Property Syntax**
The property syntax used in the properties file or environment Hashtable is as follows:

- To define a `ConnectionFactory`, use format `connectionfactory.LOOKUP_NAME = CONNECTION_URI`.
- To define a `Queue`, use format `queue.LOOKUP_NAME = QUEUE_NAME`.
- To define a `Topic` use format `topic.LOOKUP_NAME = TOPIC_NAME`.

For more details about the connection URI, see the next section.

As an example, consider the following properties that define a `ConnectionFactory`, `Queue`, and `Topic`.

```java
connectionfactory.myFactoryLookup = amqp://localhost:5672
queue.myQueueLookup = queueA
topic.myTopicLookup = topicA
```

These objects could then be looked up from a `Context` as follows.

```java
ConnectionFactory factory = (ConnectionFactory) context.lookup("myFactoryLookup");
Queue queue = (Queue) context.lookup("myQueueLookup");
Topic topic = (Topic) context.lookup("myTopicLookup");
```

### 4.2. CONNECTION URIS

A `ConnectionFactory` is configured using a connection URI.

**Connection URI Format**

```
amqp[s]://hostname:port[?option=value[&option2=value...]]
```

The available connection settings are detailed in the Section 4.3, “Connection URI Options” section.

When failover is configured, the client can reconnect to another server automatically if the connection to the current server is lost. Failover URIs start with the prefix `failover:` and contain a comma-separated list of server URIs inside parentheses. Additional options are specified at the end.

**Failover URI Format**

```
failover:(amqp://host1:port[,amqp://host2:port...][?option=value[&option2=value...]])
```

As with the connection URI example, the client can be configured with a number of different settings using the URI in a failover configuration. These settings are detailed below, with the failover options section being of particular interest.

When the `amqps` scheme is used to specify an SSL/TLS connection, the hostname segment from the URI can be used by the JVM’s TLS SNI (Server Name Indication) extension to communicate the desired server hostname during a TLS handshake. The SNI extension is automatically included if a Fully
Qualified Domain Name (for example, "myhost.mydomain") is specified, but not when an unqualified name (for example, "myhost") or a bare IP address is used.

4.3. CONNECTION URI OPTIONS

4.3.1. JMS Options

These options control the behaviour of JMS objects such as Connection, Session, MessageConsumer, and MessageProducer.

- **jms.username** - The username used to authenticate the connection.
- **jms.password** - The password used to authenticate the connection.
- **jms.clientID** - The client ID that is applied to the connection.
- **jms.forceAsyncSend** - If enabled, all messages from a MessageProducer are sent asynchronously. Otherwise, only certain kinds, such as non-persistent messages or those inside a transaction, are sent asynchronously. Default is false.
- **jms.forceSyncSend** - If enabled, all messages from a MessageProducer are sent synchronously. Default is false.
- **jms.forceAsyncAcks** - If enabled, all message acknowledgments are sent asynchronously. Default is false.
- **jms.localMessageExpiry** - If enabled, any expired messages received by a MessageConsumer are filtered out and not delivered. Default is true.
- **jms.localMessagePriority** - If enabled, prefetched messages are reordered locally based on their message priority value. Default is false.
- **jms.validatePropertyName** - If enabled, message property names are required to be valid Java identifiers. Default is true.
- **jms.receiveLocalOnly** - If enabled, calls to receive with a timeout argument will check a consumer's local message buffer only. Otherwise, if the timeout expires, the remote peer is checked to ensure there are really no messages. Default is false.
- **jms.receiveNoWaitLocalOnly** - If enabled, calls to receiveNoWait will check a consumer's local message buffer only. Otherwise, the remote peer is checked to ensure there are really no messages available. Default is false.
- **jms.queuePrefix** - An optional prefix value added to the name of any Queue created from a Session.
- **jms.topicPrefix** - An optional prefix value added to the name of any Topic created from a Session.
- **jms.closeTimeout** - The time in milliseconds for which the client will wait for normal resource closure before returning. Default is 60000 (60 seconds).
- **jms.connectTimeout** - The time in milliseconds for which the client will wait for connection establishment before returning with an error. Default is 15000 (15 seconds).
- **jms.sendTimeout** - The time in milliseconds for which the client will wait for completion of a synchronous message send before returning an error. By default the client will wait indefinitely for a send to complete.

- **jms.requestTimeout** - The time in milliseconds for which the client will wait for completion of various synchronous interactions like opening a producer or consumer (excluding send) with the remote peer before returning an error. By default the client will wait indefinitely for a request to complete.

- **jms.clientIDPrefix** - An optional prefix value used to generate client ID values when a new Connection is created by the ConnectionFactory. Default is ID:.

- **jms.connectionIDPrefix** - An optional prefix value used to generate connection ID values when a new Connection is created by the ConnectionFactory. This connection ID is used when logging some information from the Connection object, so a configurable prefix can make breadcrumbing the logs easier. Default is ID:.

- **jms.populateJMSXUserID** - If enabled, populate the JMSXUserID property for each sent message using the authenticated username from the connection. Default is false.

- **jms.awaitClientId** - If enabled, a connection with no ClientID configured in the URI will wait for a ClientID to be set programmatically, or the connection being used otherwise to signal none can be set, before sending the AMQP connection Open. Default is true.

- **jms.useDaemonThread** - If enabled, a connection will use a daemon thread for its executor, rather than a non-daemon thread. Default is false.

**Prefetch Policy Options**
Prefetch policy determines how many messages each MessageConsumer will fetch from the remote peer and hold in a local "prefetch" buffer.

- **jms.prefetchPolicy.queuePrefetch** - Default is 1000.

- **jms.prefetchPolicy.topicPrefetch** - Default is 1000.

- **jms.prefetchPolicy.queueBrowserPrefetch** - Default is 1000.

- **jms.prefetchPolicy.durableTopicPrefetch** - Default is 1000.

- **jms.prefetchPolicy.all** - Used to set all prefetch values at once.

**Redelivery Policy Options**
Redelivery policy controls how redelivered messages are handled on the client.

- **jms.redeliveryPolicy.maxRedeliveries** - Controls when an incoming message is rejected based on the number of times it has been redelivered. A value of 0 indicates that no message redeliveries are accepted. A value of 5 allows a message to be redelivered five times, and so on. Default is -1, meaning no limit.

**Message ID Policy Options**
Message ID policy controls the data type of the message ID assigned to messages sent from the client.

- **jms.messageIDPolicy.messageIDType** - By default, a generated String value is used for the message ID on outgoing messages. Other available types are UUID, UUID_STRING, and PREFIXED_UUID_STRING.

**Presettle Policy Options**
Presettle policy controls when a producer or consumer instance will be configured to use AMQP presettled messaging semantics.

- **jms.presettlePolicy.presettleAll** - If enabled, all producers and non-transacted consumers created operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleProducers** - If enabled, all producers operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleTopicProducers** - If enabled, any producer that is sending to a Topic or TemporaryTopic destination will operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleQueueProducers** - If enabled, any producer that is sending to a Queue or TemporaryQueue destination will operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleTransactedProducers** - If enabled, any producer that is created in a transacted Session will operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleConsumers** - If enabled, all consumers operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleTopicConsumers** - If enabled, any consumer that is receiving from a Topic or TemporaryTopic destination will operate in presettled mode. Default is false.

- **jms.presettlePolicy.presettleQueueConsumers** - If enabled, any consumer that is receiving from a Queue or TemporaryQueue destination will operate in presettled mode. Default is false.

**Deserialization Policy Options**

Deserialization policy provides a means of controlling which Java types are trusted to be deserialized from the object stream while retrieving the body from an incoming ObjectMessage composed of serialized Java Object content. By default all types are trusted during an attempt to deserialize the body. The default deserialization policy provides URI options that allow specifying a whitelist and a blacklist of Java class or package names.

- **jms.deserializationPolicy.whiteList** - A comma-separated list of class and package names that should be allowed when deserializing the contents of an ObjectMessage, unless overridden by blackList. The names in this list are not pattern values. The exact class or package name must be configured, as in java.util.Map or java.util. Package matches include sub-packages. Default is to allow all.

- **jms.deserializationPolicy.blackList** - A comma-separated list of class and package names that should be rejected when deserializing the contents of a ObjectMessage. The names in this list are not pattern values. The exact class or package name must be configured, as in java.util.Map or java.util. Package matches include sub-packages. Default is to prevent none.

### 4.3.2. TCP Transport Options

When connected to a remote server using plain TCP, the following options specify the behavior of the underlying socket. These options are appended to the connection URI along with any other configuration options.

**Example: A Connection URI with Transport Options**
amqp://localhost:5672?jms.clientID=foo&transport.connectTimeout=30000

The complete set of TCP transport options is listed below.

- `transport.sendBufferSize` - Default is 64k.
- `transport.receiveBufferSize` - Default is 64k.
- `transport.trafficClass` - Default is 0.
- `transport.connectTimeout` - Default is 60 seconds.
- `transport.soTimeout` - Default is -1.
- `transport.soLinger` - Default is -1.
- `transport.tcpKeepAlive` - Default is false.
- `transport.tcpNoDelay` - Default is true.
- `transport.useEpoll` - When available, use the native epoll IO layer instead of the NIO layer. This can improve performance. Default is true.

### 4.3.3. SSL/TLS Transport Options

The SSL/TLS transport is enabled by using the amqps URI scheme. Because the SSL/TLS transport extends the functionality of the TCP-based transport, all of the TCP transport options are valid on an SSL/TLS transport URI.

**Example: A Simple SSL/TLS Connection URI**

amqps://myhost.mydomain:5671

The complete set of SSL/TLS transport options is listed below.

- `transport.keyStoreLocation` - Default is to read from the system property `javax.net.ssl.keyStore`.
- `transport.keyStorePassword` - Default is to read from the system property `javax.net.ssl.keyStorePassword`.
- `transport.trustStoreLocation` - Default is to read from the system property `javax.net.ssl.trustStore`.
- `transport.trustStorePassword` - Default is to read from the system property `javax.net.ssl.trustStorePassword`.
- `transport.keyStoreType` - The type of key store being used. Default is to read from the system property `javax.net.ssl.keyStoreType`. If the property is not set, the default is JKS.
- `transport.trustStoreType` - The type of trust store being used. Default is to read from the system property `javax.net.ssl.trustStoreType`. If the property is not set, the default is JKS.
- `transport.storeType` - Sets both keyStoreType and trustStoreType to the same value. If not set, keyStoreType and trustStoreType will default to the values specified above.
4.3.4. AMQP Options

The following options apply to aspects of behavior related to the AMQP wire protocol.

- **amqp.idleTimeout** - The time in milliseconds after which the connection will be failed if the peer sends no AMQP frames. Default is 60000 (1 minute).

- **amqp.vhost** - The virtual host to connect to. Used to populate the SASL and AMQP hostname fields. Default is the main hostname from the connection URI.

- **amqp.saslLayer** - If enabled, SASL is used when establishing connections. Default is true.

- **amqp.saslMechanisms** - A comma-separated list of SASL mechanisms the client should allow selection of, if offered by the server and usable with the configured credentials. The supported mechanisms are EXTERNAL, SCRAM-SHA-256, SCRAM-SHA-1, CRAM-MD5, PLAIN, ANONYMOUS, and GSSAPI for Kerberos. Default is to allow selection from all mechanisms except GSSAPI, which must be explicitly included here to enable.

- **amqp.maxFrameSize** - The maximum AMQP frame size in bytes allowed by the client. This value will be advertised to the remote peer. Default is 1048576 (1 MiB).

- **amqp.drainTimeout** - The time in milliseconds that the client will wait for a response from the remote peer when a consumer drain request is made. If no response is seen in the allotted timeout period, the link will be considered failed and the associated consumer will be closed. Default is 60000 (1 minute).

- **amqp.allowNonSecureRedirects** - Controls whether the client allows an AMQP redirect to an alternative host over a connection that is not secure when the existing connection is secure, such as redirecting an SSL/TLS connection to a raw TCP connection. Default is false.
4.3.5. Failover Options

Failover URIs start with the prefix `failover:` and contain a comma-separated list of server URIs inside parentheses. Additional options are specified at the end. Options prefixed with `jms.` are applied to the overall failover URI, outside of parentheses, and affect the Connection object for its lifetime.

**Example: A Failover URI with Failover Options**

```
failover:(amqp://host1:5672,amqp://host2:5672)?
jms.clientID=foo&failover.maxReconnectAttempts=20
```

The individual broker details within the parentheses can use the `transport.` or `amqp.` options defined earlier. These are applied as each host is connected to.

**Example: A Failover URI with Per-connection Transport and AMQP Options**

```
failover:(amqp://host1:5672?amqp.option=value,amqp://host2:5672?
transport.option=value)?jms.clientID=foo
```

All of the configuration options for failover are listed below.

- `failover.initialReconnectDelay` - The time in milliseconds the client will wait before the first attempt to reconnect to a remote peer. Default is 0, meaning the first attempt happens immediately.

- `failover.reconnectDelay` - The time in milliseconds between reconnection attempts. If the backoff option is not enabled, this value remains constant. Default is 10.

- `failover.maxReconnectDelay` - The maximum time that the client will wait before attempting to reconnect. This value is only used when the backoff feature is enabled to ensure that the delay does not grow too large. Default is 30 seconds.

- `failover.useReconnectBackOff` - If enabled, the time between reconnection attempts grows based on a configured multiplier. Default is true.

- `failover.reconnectBackOffMultiplier` - The multiplier used to grow the reconnection delay value. Default is 2.0.

- `failover.maxReconnectAttempts` - The number of reconnection attempts allowed before reporting the connection as failed to the client. Default is -1, meaning no limit.

- `failover.startupMaxReconnectAttempts` - For a client that has never connected to a remote peer before, this option controls how many attempts are made to connect before reporting the connection as failed. Default is to use the value of `maxReconnectAttempts`.

- `failover.warnAfterReconnectAttempts` - Controls how often the client will log a message indicating that failover reconnection is being attempted. Default is to log every 10 connection attempts.

- `failover.randomize` - If enabled, the set of failover URIs is randomly shuffled before attempting to connect to one of them. This can help to distribute client connections more evenly across multiple remote peers. Default is false.

- `failover.amqpOpenServerListAction` - Controls how the failover transport behaves when the connection "open" frame from the server provides a list of failover hosts to the client. Valid values are `REPLACE`, `ADD`, or `IGNORE`. If `REPLACE` is configured, all failover URIs other than the
one for the current server are replaced with those provided by the server. If **ADD** is configured, the URIs provided by the server are added to the existing set of failover URIs, with deduplication. If **IGNORE** is configured, any updates from the server are ignored and no changes are made to the set of failover URIs in use. Default is **REPLACE**.

The failover URI also supports defining nested options as a means of specifying AMQP and transport option values applicable to all the individual nested broker URIs. This is accomplished using the same `transport` and `amqp`. URI options outlined earlier for a non-failover broker URI but prefixed with `failover.nested`. For example, to apply the same value for the `amqp.vhost` option to every broker connected to you might have a URI like the following.

**Example: A Failover URI with Shared Transport and AMQP Options**

```
failover:(amqp://host1:5672,amqp://host2:5672)?
jms.clientID=foo&failover.nested.amqp.vhost=myhost
```

### 4.3.6. Discovery Options

The client has an optional discovery module that provides a customized failover layer where the broker URIs to connect to are not given in the initial URI but instead are discovered by interacting with a discovery agent. There are currently two discovery agent implementations: a file watcher that loads URIs from a file and a multicast listener that works with ActiveMQ 5.x brokers that are configured to broadcast their broker addresses for listening clients.

The general set of failover-related options when using discovery are the same as those detailed earlier, with the main prefix changed from `failover` to `discovery`, and with the `nested` prefix used to supply URI options common to all the discovered broker URIs. For example, without the agent URI details, a general discovery URI might look like the following.

**Example: A Discovery URI**

```
discovery:(<agent-uri>)?
discovery.maxReconnectAttempts=20&discovery.discovered.jms.clientID=foo
```

To use the file watcher discovery agent, create an agent URI like the following.

**Example: A Discovery URI Using the File Watcher Agent**

```
discovery:(file:///path/to/monitored-file?updateInterval=60000)
```

The URI options for the file watcher discovery agent are listed below.

- **updateInterval** - The time in milliseconds between checks for file changes. Default is 30000 (30 seconds).

To use the multicast discovery agent with an ActiveMQ 5.x broker, create an agent URI like the following.

**Example: A Discovery URI Using the Multicast Listener Agent**

```
discovery:(multicast://default?group=default)
```
Note that the use of `default` as the host in the multicast agent URI above is a special value that is substituted by the agent with the default `239.255.2.3:6155`. You can change this to specify the actual IP address and port in use with your multicast configuration.

The URI option for the multicast discovery agent is listed below.

- **group** - The multicast group used to listen for updates. Default is `default`.

### 4.4. SECURITY

AMQ JMS has a range of security-related configuration options that can be leveraged according to your application’s needs.

Basic user credentials such as username and password should be passed directly to the `ConnectionFactory` when creating the `Connection` within the application. However, if you are using the no-argument factory method, it is also possible to supply user credentials in the connection URI. For more information, see the Section 4.3.1, “JMS Options” section.

Another common security consideration is use of SSL/TLS. The client connects to servers over an SSL/TLS transport when the `amqps` URI scheme is specified in the connection URI, with various options available to configure behavior. For more information, see the Section 4.3.3, “SSL/TLS Transport Options” section.

In concert with the earlier items, it may be desirable to restrict the client to allow use of only particular SASL mechanisms from those that may be offered by a server, rather than selecting from all it supports. For more information, see the Section 4.3.4, “AMQP Options” section.

Applications calling `getObject()` on a received `ObjectMessage` may wish to restrict the types created during deserialization. Note that message bodies composed using the AMQP type system do not use the `ObjectInputStream` mechanism and therefore do not require this precaution. For more information, see the the section called “Deserialization Policy Options” section.

#### 4.4.1. Authenticating Using Kerberos

The client can be configured to authenticate using Kerberos when used with an appropriately configured server. To enable Kerberos, use the following steps.

1. Configure the client to use the `GSSAPI` mechanism for SASL authentication using the `amqp.saslMechanisms` URI option.

    ```
    amqp://myhost:5672?amqp.saslMechanisms=GSSAPI
    ```

2. Set the `java.security.auth.login.config` system property to the path of a JAAS login configuration file containing appropriate configuration for a Kerberos LoginModule.

    ```
    -Djava.security.auth.login.config=LOGIN_CONFIG_FILE
    ```

    The login configuration file might look like the following example.

    ```
    amqp-jms-client {
        com.sun.security.auth.module.Krb5LoginModule required
        useTicketCache=true;
    };
    ```
The precise configuration used will depend on how you wish the credentials to be established for the connection, and the particular `LoginModule` in use. For details of the Oracle Krb5LoginModule, see the Oracle Krb5LoginModule class reference. For details of the IBM Java 8 Krb5LoginModule, see the IBM Krb5LoginModule class reference.

It is possible to configure a `LoginModule` to establish the credentials to use for the Kerberos process, such as specifying a principal and whether to use an existing ticket cache or keytab. If, however, the `LoginModule` configuration does not provide the means to establish all necessary credentials, it may then request and be passed the username and password values from the client `Connection` object if they were either supplied when creating the `Connection` using the `ConnectionFactory` or previously configured via its URI options.

Note that Kerberos is supported only for authentication purposes. Use SSL/TLS connections for encryption.

The following connection URI options can be used to influence the Kerberos authentication process.

- `sasl.options.configScope` - The name of the login configuration entry used to authenticate. Default is `amqp-jms-client`.
- `sasl.options.protocol` - The protocol value used during the GSSAPI SASL process. Default is `amqp`.
- `sasl.options.serverName` - The `serverName` value used during the GSSAPI SASL process. Default is the server hostname from the connection URI.

Similar to the `amqp` and `transport` options detailed previously, these options must be specified on a per-host basis or as all-host nested options in a failover URI.

4.5. LOGGING

The client uses the SLF4J API, allowing users to select a particular logging implementation based on their needs by supplying an SLF4J binding, such as `slf4j-log4j` in order to use Log4J. More details on SLF4J are available from its website.

The client uses Logger names residing within the `org.apache.qpid.jms` hierarchy, which you can use to configure a logging implementation based on your needs.

When debugging some issues, it can sometimes be useful to enable additional protocol trace logging from the Qpid Proton AMQP 1.0 library. There are two ways to achieve this.

- Set the environment variable (not the Java system property) `PN_TRACE_FRM` to 1. This will cause Proton to emit frame logging to the console.
- Add the option `amqp.traceFrames=true` to your `connection URI` and configure the `org.apache.qpid.jms.provider.amqp.FRAMES` logger to log level `TRACE`. This will add a protocol tracer to Proton and include the output in your logs.
CHAPTER 5. EXAMPLES

This chapter demonstrates the use of AMQ JMS through example programs. To run them, make sure you have completed the steps in the Chapter 2, Installation chapter for your environment and you have a running and configured broker.

See the Qpid JMS examples for more sample programs.

5.1. CONFIGURING THE JNDI CONTEXT

Applications using JMS typically use JNDI to obtain the ConnectionFactory and Destination objects used by the application. This keeps the configuration separate from the program and insulates it from the particular client implementation.

For the purpose of using these examples, a file named jndi.properties should be placed on the classpath to configure the JNDI Context, as detailed previously.

The contents of the jndi.properties file should match what is shown below, which as per the format described previously establishes that the client's InitialContextFactory implementation should be used, configures a ConnectionFactory to connect to a local server, and defines a destination queue named queue.

```property
# Configure the InitialContextFactory class to use
java.naming.factory.initial = org.apache.qpid.jms.jndi.JmsInitialContextFactory

# Configure the ConnectionFactory
connectionfactory.myFactoryLookup = amqp://localhost:5672

# Configure the destination
queue.myDestinationLookup = queue
```

5.2. SENDING MESSAGES

This example first creates a JNDI Context, uses it to look up a ConnectionFactory and Destination, creates and starts a Connection using the factory, and then creates a Session. Then a MessageProducer is created to the Destination, and a message is sent using it. The Connection is then closed, and the program exits.

A runnable variant of this Sender example is in the INSTALL_DIR/examples directory, along with the Hello World example covered previously in Chapter 3, Getting Started.

Example: Sending Messages

```java
package org.jboss.amq.example;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.DeliveryMode;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSError;
import javax.jms.Message;
```
Creates the JNDI Context to look up ConnectionFactory and Destination objects. The configuration is picked up from the jndi.properties file as detailed earlier.

The ConnectionFactory and Destination objects are retrieved from the JNDI Context using their lookup names.
3. The factory is used to create the Connection, which then has an ExceptionListener registered and is then started. The credentials given when creating the connection will typically be taken from an appropriate external configuration source, ensuring they remain separate from the application itself and can be updated independently.

4. A non-transacted, auto-acknowledge Session is created on the Connection.

5. The MessageProducer is created to send messages to the Destination.

6. A TextMessage is created with the given content.

7. The TextMessage is sent. It is sent non-persistent, with default priority and no expiration.

8. The Connection is closed. The Session and MessageProducer are closed implicitly.

Note that this is only an example. A real-world application would typically use a long-lived MessageProducer and send many messages using it over time. Opening and then closing a Connection, Session, and MessageProducer per message is generally not efficient.

5.3. RECEIVING MESSAGES

This example starts by creating a JNDI Context, using it to look up a ConnectionFactory and Destination, creating and starting a Connection using the factory, and then creates a Session. Then a MessageConsumer is created for the Destination, a message is received using it, and its contents are printed to the console. The Connection is then closed and the program exits. The same JNDI configuration is used as in the sending example.

An executable variant of this Receiver example is contained within the examples directory of the client distribution, along with the Hello World example covered previously in Chapter 3, Getting Started.

Example: Receiving Messages

```java
package org.jboss.amq.example;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;

public class Receiver {
    public static void main(String[] args) throws Exception {
        try {
            Context context = new InitialContext();

            ConnectionFactory factory = (ConnectionFactory) context.lookup("myFactoryLookup");
```
Creates the JNDI Context to look up ConnectionFactory and Destination objects. The configuration is picked up from the jndi.properties file as detailed earlier.

The ConnectionFactory and Destination objects are retrieved from the JNDI Context using their lookup names.

The factory is used to create the Connection, which then has an ExceptionListener registered and is then started. The credentials given when creating the connection will typically be taken from an appropriate external configuration source, ensuring they remain separate from the application itself and can be updated independently.

A non-transacted, auto-acknowledge Session is created on the Connection.
The `MessageConsumer` is created to receive messages from the `Destination`.

A call to receive a message is made with a five second timeout.

The result is checked, and if a message was received, its contents are printed, or notice that no message was received. The result is cast explicitly to `TextMessage` as this is what we know the `Sender` sent.

The `Connection` is closed. The `Session` and `MessageConsumer` are closed implicitly.

Note that this is only an example. A real-world application would typically use a long-lived `MessageConsumer` and receive many messages using it over time. Opening and then closing a `Connection`, `Session`, and `MessageConsumer` for each message is generally not efficient.
CHAPTER 6. INTEROPERABILITY

This chapter discusses how to use AMQ JMS in combination with other AMQ components. For an overview of the compatibility of AMQ components, see the product introduction.

6.1. INTEROPERATING WITH OTHER AMQP CLIENTS

AMQP messages are composed using the AMQP type system. Having this common format is one of the reasons AMQP clients in different languages are able to interoperate with each other. This section serves to document behaviour around the AMQP payloads sent and received by the client in relation to the various JMS Message types used, to aid in using the client along with other AMQP clients.

6.1.1. Sending Messages

This section serves to document the different payloads sent by the client when using the various JMS Message types, so as to aid in using other clients to receive them.

6.1.1.1. Message Type

<table>
<thead>
<tr>
<th>JMS Message Type</th>
<th>Description of Transmitted AMQP Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextMessage</td>
<td>A TextMessage will be sent using an amqp-value body section containing a utf8 encoded string of the body text, or null if no body text is set. The message annotation with symbol key of “x-opt-jms-msg-type” will be set to a byte value of 5.</td>
</tr>
<tr>
<td>BytesMessage</td>
<td>A BytesMessage will be sent using a data body section containing the raw bytes from the BytesMessage body, with the properties section content-type field set to the symbol value “application/octet-stream”. The message annotation with symbol key of “x-opt-jms-msg-type” will be set to a byte value of 3.</td>
</tr>
<tr>
<td>MapMessage</td>
<td>A MapMessage body will be sent using an amqp-value body section containing a single map value. Any byte[] values in the MapMessage body will be encoded as binary entries in the map. The message annotation with symbol key of “x-opt-jms-msg-type” will be set to a byte value of 2.</td>
</tr>
<tr>
<td>StreamMessage</td>
<td>A StreamMessage will be sent using an amqp-sequence body section containing the entries in the StreamMessage body. Any byte[] entries in the StreamMessage body will be encoded as binary entries in the sequence. The message annotation with symbol key of “x-opt-jms-msg-type” will be set to a byte value of 4.</td>
</tr>
<tr>
<td>ObjectMessage</td>
<td>An ObjectMessage will be sent using a data body section, containing the bytes from serializing the ObjectMessage body using an ObjectOutputStream, with the properties section content-type field set to the symbol value “application/x-java-serialized-object”. The message annotation with symbol key of “x-opt-jms-msg-type” will be set to a byte value of 1.</td>
</tr>
<tr>
<td>Message</td>
<td>A plain JMS Message has no body, and will be sent as an amqp-value body section containing a null. The message annotation with symbol key of “x-opt-jms-msg-type” will be set to a byte value of 0.</td>
</tr>
</tbody>
</table>
6.1.1.2. Message Properties

JMS messages support setting application properties of various Java types. This section serves to show the mapping of these property types to AMQP typed values in the application-properties section of the sent message. Both JMS and AMQP use string keys for property names.

<table>
<thead>
<tr>
<th>JMS Property Type</th>
<th>AMQP ApplicationProperty Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>byte</td>
<td>byte</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>String</td>
<td>string or null</td>
</tr>
</tbody>
</table>

6.1.2. Receiving Messages

This section serves to document the different payloads received by the client will be mapped to the various JMS Message types, so as to aid in using other clients to send messages for receipt by the JMS client.

6.1.2.1. Message Type

If the the “x-opt-jms-msg-type” message-annotation is present on the received AMQP message, its value is used to determine the JMS message type used to represent it, according to the mapping detailed in the following table. This reflects the reverse process of the mappings discussed for messages sent by the JMS client.

<table>
<thead>
<tr>
<th>AMQP “x-opt-jms-msg-type” message-annotation value (type)</th>
<th>JMS Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (byte)</td>
<td>Message</td>
</tr>
<tr>
<td>1 (byte)</td>
<td>ObjectMessage</td>
</tr>
<tr>
<td>2 (byte)</td>
<td>MapMessage</td>
</tr>
<tr>
<td>3 (byte)</td>
<td>BytesMessage</td>
</tr>
<tr>
<td>4 (byte)</td>
<td>StreamMessage</td>
</tr>
</tbody>
</table>
If the “x-opt-jms-msg-type” message-annotation is not present, the table below details how the message will be mapped to a JMS Message type. Note that the StreamMessage and MapMessage types are only assigned to annotated messages.

<table>
<thead>
<tr>
<th>Description of Received AMQP Message without “x-opt-jms-msg-type” annotation</th>
<th>JMS Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>• An amqp-value body section containing a string or null.</td>
<td>TextMessage</td>
</tr>
<tr>
<td>• A data body section, with the properties section content-type field set to a symbol value representing a common textual media type such as &quot;text/plain&quot;, &quot;application/xml&quot;, or &quot;application/json&quot;.</td>
<td></td>
</tr>
<tr>
<td>• An amqp-value body section containing a binary.</td>
<td>BytesMessage</td>
</tr>
<tr>
<td>• A data body section, with the properties section content-type field either not set, set to symbol value &quot;application/octet-stream&quot;, or set to any value not understood to be associated with another message type.</td>
<td></td>
</tr>
<tr>
<td>• A data body section, with the properties section content-type field set to symbol value &quot;application/x-java-serialized-object&quot;.</td>
<td>ObjectMessage</td>
</tr>
<tr>
<td>• An amqp-value body section containing a value not covered above.</td>
<td></td>
</tr>
<tr>
<td>• An amqp-sequence body section. This will be represented as a List inside the ObjectMessage.</td>
<td></td>
</tr>
</tbody>
</table>

6.1.2.2. Message Properties

This section serves to show the mapping of values in the application-properties section of the received AMQP message to Java types used in the JMS Message.

<table>
<thead>
<tr>
<th>AMQP ApplicationProperty Type</th>
<th>JMS Property Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>byte</td>
<td>byte</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>
6.2. CONNECTING TO AMQ BROKER

AMQ Broker is designed to interoperate with AMQP 1.0 clients. Check the following to ensure the broker is configured for AMQP messaging.

- Port 5672 in the network firewall is open.
- The AMQ Broker AMQP acceptor is enabled. See Configuring Network Access.
- The necessary addresses are configured on the broker. See Addresses, Queues, and Topics.
- The broker is configured to permit access from your client, and the client is configured to send the required credentials. See Broker Security.

6.3. CONNECTING TO AMQ INTERCONNECT

AMQ Interconnect works with any AMQP 1.0 client. Check the following to ensure the components are configured correctly.

- Port 5672 in the network firewall is open.
- The router is configured to permit access from your client, and the client is configured to send the required credentials. See Interconnect Security.
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.

Accessing Your Account

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

Activating a Subscription

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

Downloading Zip and Tar Files
To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Go to access.redhat.com.
2. Navigate to DOWNLOADS.
3. Locate the Red Hat JBoss AMQ entry in the JBOSS INTEGRATION AND AUTOMATION category.
4. Select the desired component type from the drop-down menu on the right side of the entry.
5. Select the Download link for your component.

Registering Your System for Packages
To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using zip or tar files, this step is not required.

1. Go to access.redhat.com.
2. Navigate to Registration Assistant.
3. Select your OS version and continue to the next page.
4. Use the listed command in your system terminal to complete the registration.

To learn more see How to Register and Subscribe a System to the Red Hat Customer Portal.

Revised on 2017-12-15 13:52:27 EST