Red Hat AMQ 7.6

Using the AMQ JMS Client

For Use with AMQ Clients 2.7
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
- 1.1. KEY FEATURES .................................................. 4
- 1.2. SUPPORTED STANDARDS AND PROTOCOLS ................. 4
- 1.3. SUPPORTED CONFIGURATIONS ............................... 5
- 1.4. TERMS AND CONCEPTS ........................................ 5
- 1.5. DOCUMENT CONVENTIONS ..................................... 6
  - The sudo command .............................................. 6
  - File paths ...................................................... 6
  - Variable text .................................................. 7

## CHAPTER 2. INSTALLATION
- 2.1. PREREQUISITES .................................................. 8
- 2.2. USING THE RED HAT MAVEN REPOSITORY .................... 8
- 2.3. INSTALLING A LOCAL MAVEN REPOSITORY ................. 8
- 2.4. INSTALLING THE EXAMPLES ................................ 9

## CHAPTER 3. GETTING STARTED
- 3.1. PREREQUISITES .................................................. 10
- 3.2. RUNNING HELLO WORLD ..................................... 10

## CHAPTER 4. CONFIGURATION
- 4.1. CONFIGURING THE INITIAL CONTEXT FACTORY ............. 11
  - Using a jndi.properties file .................................. 11
  - Using a system property ...................................... 11
- 4.2. CONFIGURING THE CONNECTION FACTORY .................. 11
- 4.3. CONNECTION URIS .............................................. 12
- 4.4. JMS OPTIONS .................................................... 12
  - Prefetch policy options ...................................... 14
  - Redelivery policy options ................................... 14
  - Message ID policy options .................................... 15
  - Presettle policy options ...................................... 15
  - Deserialization policy options ............................... 15
- 4.5. TCP OPTIONS .................................................... 16
- 4.6. SSL/TLS OPTIONS ............................................. 16
- 4.7. AMQP OPTIONS .................................................. 18
- 4.8. FAILOVER OPTIONS ........................................... 18
- 4.9. DISCOVERY OPTIONS ......................................... 20
- 4.10. CONFIGURING JNDI RESOURCES ............................ 21
  - 4.10.1. Configuring queue and topic names .................... 21
  - 4.10.2. Setting JNDI properties programatically .......... 21
  - 4.10.3. Variable expansion in JNDI properties ............ 22

## CHAPTER 5. EXAMPLES
- 5.1. CONFIGURING THE JNDI CONTEXT ............................ 23
- 5.2. SENDING MESSAGES .......................................... 23
- 5.3. RECEIVING MESSAGES ....................................... 25

## CHAPTER 6. SECURITY
- 6.1. AUTHENTICATING USING KERBEROS ......................... 27
- 6.2. ENABLING OPENSSL SUPPORT ................................ 28

## CHAPTER 7. MESSAGE DELIVERY
- 7.1. HANDLING UNACKNOWLEDGED DELIVERIES .................... 29
Non-transacted producer with an unacknowledged delivery 29
Transacted producer with an uncommitted transaction 29
Transacted producer with a pending commit 29
Non-transacted consumer with an unacknowledged delivery 29
Transacted consumer with an uncommitted transaction 29
Transacted consumer with a pending commit 29
7.2. EXTENDED SESSION ACKNOWLEDGMENT MODES 30
Individual acknowledge 30
No acknowledge 30

CHAPTER 8. LOGGING AND TRACING 31
8.1. ENABLING PROTOCOL LOGGING 31
8.2. ENABLING DISTRIBUTED TRACING 31

CHAPTER 9. INTEROPERABILITY 33
9.1. INTEROPERATING WITH OTHER AMQP CLIENTS 33
9.1.1. Sending messages 33
9.1.1.1. Message type 33
9.1.1.2. Message properties 34
9.1.2. Receiving messages 34
9.1.2.1. Message type 34
9.1.2.2. Message properties 35
9.2. CONNECTING TO AMQ BROKER 36
9.3. CONNECTING TO AMQ INTERCONNECT 36

APPENDIX A. USING YOUR SUBSCRIPTION 37
A.1. ACCESSING YOUR ACCOUNT 37
A.2. ACTIVATING A SUBSCRIPTION 37
A.3. DOWNLOADING RELEASE FILES 37
A.4. REGISTERING YOUR SYSTEM FOR PACKAGES 37

APPENDIX B. USING RED HAT MAVEN REPOSITORIES 39
B.1. USING THE ONLINE REPOSITORY 39
Adding the repository to your Maven settings 39
Adding the repository to your POM file 40
B.2. USING A LOCAL REPOSITORY 40

APPENDIX C. USING AMQ BROKER WITH THE EXAMPLES 42
C.1. INSTALLING THE BROKER 42
C.2. STARTING THE BROKER 42
C.3. CREATING A QUEUE 42
C.4. STOPPING THE BROKER 42
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. For an overview of the clients, see AMQ Clients Overview. For information about this release, see AMQ Clients 2.7 Release Notes.

AMQ JMS is based on the JMS implementation from Apache Qpid. 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
- Flexible SASL authentication
- Automatic reconnect and failover
- Ready for use with OSGi containers
- Pure-Java implementation
- Distributed tracing based on the OpenTracing standard

**IMPORTANT**

Distributed tracing in AMQ Clients is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

**NOTE**

AMQ JMS does not currently support distributed transactions (XA). If your application requires distributed transactions, it is recommended that you use the AMQ Core Protocol JMS client.

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
• Versions 1.0, 1.1, 1.2, and 1.3 of the Transport Layer Security (TLS) protocol, the successor to SSL
• Simple Authentication and Security Layer (SASL) mechanisms including ANONYMOUS, PLAIN, SCRAM, EXTERNAL, and GSSAPI (Kerberos)
• Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

AMQ JMS supports the OS and language versions listed below. For more information, see Red Hat AMQ 7 Supported Configurations.

• Red Hat Enterprise Linux 7 and 8 with the following JDKs:
  ○ OpenJDK 8 and 11
  ○ Oracle JDK 8
  ○ IBM JDK 8
• Red Hat Enterprise Linux 6 with the following JDKs:
  ○ OpenJDK 8
  ○ Oracle JDK 8
• IBM AIX 7.1 with IBM JDK 8
• Microsoft Windows 10 Pro with Oracle JDK 8
• Microsoft Windows Server 2012 R2 and 2016 with Oracle JDK 8
• Oracle Solaris 10 and 11 with Oracle JDK 8

AMQ JMS is supported in combination with the following AMQ components and versions:

• All versions of AMQ Broker
• All versions of AMQ Interconnect
• All versions of AMQ Online
• A-MQ 6 versions 6.2.1 and newer

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>ConnectionFactory</td>
<td>An entry point for creating connections.</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

**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 JMS in your environment.

2.1. PREREQUISITES

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

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

2. Add the library dependency to your POM file.

   ```xml
   <dependency>
   <groupId>org.apache.qpid</groupId>
   <artifactId>qpid-jms-client</artifactId>
   <version>0.51.0.redhat-00002</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 JMS can be installed to your local filesystem as a file-based Maven repository.

Procedure

1. Use your subscription to download the AMQ Clients 2.7.0 JMS 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-clients-2.7.0-jms-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 Clients 2.7.0 JMS.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-clients-2.7.0-jms.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-clients-2.7.0-jms 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 queue. For more information, see Creating a queue.

3.2. RUNNING HELLO WORLD

The Hello World example creates a connection to the broker, sends a message containing a greeting to the queue queue, and receives it back. On success, it prints the received message to the console.

Procedure

1. Use Maven to build the examples by running the following command in the <install-dir>/examples 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.qpid.jms.example.HelloWorld

   On Windows:

   > java -cp "target/classes:target/dependency/*" org.apache.qpid.jms.example.HelloWorld

   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 is in the <install-dir>/examples/src/main/java directory. The JNDI and logging configuration is in the <install-dir>/examples/src/main/resources directory.
CHAPTER 4. CONFIGURATION

This chapter describes the process for binding the AMQ 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. Some of the options are also exposed as corresponding set and get methods on the ConnectionFactory implementation object.

4.1. CONFIGURING THE INITIAL CONTEXT FACTORY

JMS applications use a JNDI InitialContext object obtained from an InitialContextFactory to look up JMS objects such as the connection factory. AMQ JMS provides an implementation of the InitialContextFactory in the org.apache.qpid.jms.jndi.JmsInitialContextFactory class.

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

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

To find an implementation, JNDI must be configured in your environment. There are two main ways of achieving this, using a jndi.properties file or using a system property.

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
java.naming.factory.initial = org.apache.qpid.jms.jndi.JmsInitialContextFactory
```

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.qpid.jms.jndi.JmsInitialContextFactory ...
```

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.<factory-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**

```java
ConnectionFactory app1 = amqp://example.net:5672?jms.clientID=backend
```

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

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

### 4.3. CONNECTION URIS

A connection factory is configured using a connection URI in the following format:

**The connection URI format**

```text
amqp[s]://<host>[:<port>][?<option>=<value>[&<option>=<value>]]
```

For example, the following is a connection URI that connects to host example.net at port 5672 and sets the client ID to backend:

**Example: A connection URI**

```text
amqp://example.net:5672?jms.clientID=backend
```

The available connection options are described in the sections following this one.

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

**The failover URI format**

```text
failover:(amqp[s]://<host>[:<port>][.amqp[s]://<host>[:<port>...]][?<option>=<value>[&<option>=<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 Section 4.8, "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.

### 4.4. JMS OPTIONS

These options control the behaviour of JMS objects such as Connection, Session, MessageConsumer, and MessageProducer.
jms.username
   The user name 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. It is disabled by default.

jms.forceSyncSend
   If enabled, all messages from a MessageProducer are sent synchronously. It is disabled by default.

jms.forceAsyncAcks
   If enabled, all message acknowledgments are sent asynchronously. It is disabled by default.

jms.localMessageExpiry
   If enabled, any expired messages received by a MessageConsumer are filtered out and not delivered. It is enabled by default.

jms.localMessagePriority
   If enabled, prefetched messages are reordered locally based on their message priority value. It is disabled by default.

jms.validatePropertyNames
   If enabled, message property names are required to be valid Java identifiers. It is enabled by default.

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. It is disabled by default.

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. It is disabled by default.

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. The 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. The 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 operations before returning an error. By default the client will wait indefinitely for a request to complete.
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`. The 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. The default is `ID:`.

**jms.populateJMSXUserID**
If enabled, populate the `JMSXUserID` property for each sent message using the authenticated user name from the connection. It is disabled by default.

**jms.awaitClientID**
If enabled, a connection with no client ID configured in the URI will wait for a client ID to be set programmatically, or the connection being used otherwise to signal none can be set, before sending the AMQP connection "open". It is enabled by default.

**jms.useDaemonThread**
If enabled, a connection will use a daemon thread for its executor, rather than a non-daemon thread. It is disabled by default.

**jms.tracing**
The name of a tracing provider. Supported values are `opentracing` and `noop`. The default is `noop`.

**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**
The default is 1000.

**jms.prefetchPolicy.topicPrefetch**
The default is 1000.

**jms.prefetchPolicy.queueBrowserPrefetch**
The default is 1000.

**jms.prefetchPolicy.durableTopicPrefetch**
The default is 1000.

**jms.prefetchPolicy.all**
This can be used to set all prefetch values at once.

The value of prefetch can affect the distribution of messages to multiple consumers on a queue or shared subscription. A higher value can result in larger batches sent at once to each consumer. To achieve more even round-robin distribution, use a lower value.

**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. The default is -1, meaning no limit.

**jms.redeliveryPolicy.outcome**

Controls the outcome applied to a message once it has exceeded the configured maxRedeliveries value. Supported values are: ACCEPTED, REJECTED, RELEASED, MODIFIED_FAILED and MODIFIED_FAILED_UNDELIVERABLE. The default value is MODIFIED_FAILED_UNDELIVERABLE.

**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. It is disabled by default.

**jms.presettlePolicy.presettleProducers**

If enabled, all producers operate in presettled mode. It is disabled by default.

**jms.presettlePolicy.presettleTopicProducers**

If enabled, any producer that is sending to a Topic or TemporaryTopic destination will operate in presettled mode. It is disabled by default.

**jms.presettlePolicy.presettleQueueProducers**

If enabled, any producer that is sending to a Queue or TemporaryQueue destination will operate in presettled mode. It is disabled by default.

**jms.presettlePolicy.presettleTransactedProducers**

If enabled, any producer that is created in a transacted Session will operate in presettled mode. It is disabled by default.

**jms.presettlePolicy.presettleConsumers**

If enabled, all consumers operate in presettled mode. It is disabled by default.

**jms.presettlePolicy.presettleTopicConsumers**

If enabled, any consumer that is receiving from a Topic or TemporaryTopic destination will operate in presettled mode. It is disabled by default.

**jms.presettlePolicy.presettleQueueConsumers**

If enabled, any consumer that is receiving from a Queue or TemporaryQueue destination will operate in presettled mode. It is disabled by default.

**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. The 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. The default is to prevent none.

### 4.5. TCP 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**

```text
amqp://localhost:5672?jms.clientID=foo&transport.connectTimeout=30000
```

The complete set of TCP transport options is listed below.

- **transport.sendBufferSize**
  - The send buffer size in bytes. The default is 65536 (64 KiB).
- **transport.receiveBufferSize**
  - The receive buffer size in bytes. The default is 65536 (64 KiB).
- **transport.trafficClass**
  - The default is 0.
- **transport.connectTimeout**
  - The default is 60 seconds.
- **transport.soTimeout**
  - The default is -1.
- **transport.soLinger**
  - The default is -1.
- **transport.tcpKeepAlive**
  - The default is false.
- **transport.tcpNoDelay**
  - If enabled, do not delay and buffer TCP sends. It is enabled by default.
- **transport.useEpoll**
  - When available, use the native epoll IO layer instead of the NIO layer. This can improve performance. It is enabled by default.

### 4.6. SSL/TLS 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**
  - The path to the SSL/TLS key store. If unset, the value of the `javax.net.ssl.keyStore` system property is used.

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

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

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

- **transport.keyStoreType**
  - If unset, the value of the `javax.net.ssl.keyStoreType` system property is used. If the system property is unset, the default is **JKS**.

- **transport.trustStoreType**
  - If unset, the value of the `javax.net.ssl.trustStoreType` system property is used. If the system property is unset, the default is **JKS**.

- **transport.storeType**
  - Sets both `keyStoreType` and `trustStoreType` to the same value. If unset, `keyStoreType` and `trustStoreType` default to the values specified above.

- **transport.contextProtocol**
  - The protocol argument used when getting an SSLContext. The default is **TLS**, or **TLSv1.2** if using OpenSSL.

- **transport.enabledCipherSuites**
  - A comma-separated list of cipher suites to enable. If unset, the context-default ciphers are used. Any disabled ciphers are removed from this list.

- **transport.disabledCipherSuites**
  - A comma-separated list of cipher suites to disable. Ciphers listed here are removed from the enabled ciphers.

- **transport.enabledProtocols**
  - A comma-separated list of protocols to enable. If unset, the context-default protocols are used. Any disabled protocols are removed from this list.

- **transport.disabledProtocols**
  - A comma-separated list of protocols to disable. Protocols listed here are removed from the enabled protocol list. The default is **SSLv2Hello,SSLv3**.

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

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

**transport.keyAlias**
The alias to use when selecting a key pair from the key store if required to send a client certificate to the server.

**transport.useOpenSSL**
If enabled, use native OpenSSL libraries for SSL/TLS connections if available. It is disabled by default.
For more information, see Section 6.2, “Enabling OpenSSL support”.

### 4.7. 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 is failed if the peer sends no AMQP frames. The default is 60000 (1 minute).

**amqp.vhost**
The virtual host to connect to. This is used to populate the SASL and AMQP hostname fields. The default is the main hostname from the connection URI.

**amqp.saslLayer**
If enabled, SASL is used when establishing connections. It is enabled by default.

**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. The 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. The 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. The default is 60000 (1 minute).

**amqp.allowNonSecureRedirects**
If enabled, allow AMQP redirects to alternative hosts when the existing connection is secure and the alternative connection is not. For example, if enabled this would permit redirecting an SSL/TLS connection to a raw TCP connection. It is disabled by default.

### 4.8. FAILOVER OPTIONS

Failover URIs start with the prefix `failover:` and contain a comma-separated list of server URIs inside.
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**

```plaintext
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**

```plaintext
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. The 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. The 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. The default is 30 seconds.

- **failover.useReconnectBackOff**
  If enabled, the time between reconnection attempts grows based on a configured multiplier. It is enabled by default.

- **failover.reconnectBackOffMultiplier**
  The multiplier used to grow the reconnection delay value. The default is 2.0.

- **failover.maxReconnectAttempts**
  The number of reconnection attempts allowed before reporting the connection as failed to the client. The 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. If unset, the value of **maxReconnectAttempts** is used.

- **failover.warnAfterReconnectAttempts**
  The number of failed reconnection attempts until a warning is logged. The default is 10.

- **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. It is disabled by default.

- **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. The 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
```

#### 4.9. 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. The 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. The default is default.

### 4.10. CONFIGURING JNDI RESOURCES

#### 4.10.1. 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 define corresponding system properties.

**The JNDI property format for queues and topics**

```
queue.<queue-lookup-name> = <queue-name>
topic.<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");
```

#### 4.10.2. Setting JNDI properties programatically

As an alternative to using a jndi.properties file or system properties to configure JNDI, you can define
properties programatically using the JNDI initial context API.

**Example: Setting JNDI properties programatically**

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

env.put("java.naming.factory.initial", "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
env.put("connectionFactory.app1", "amqp://example.net:5672?jms.clientID=backend");
env.put("queue.jobs", "app1/work-items");
env.put("topic.notifications", "app1/updates");

InitialContext context = new InitialContext(env);
```
4.10.3. Variable expansion in JNDI properties

JNDI property values can contain variables of the form `${<variable-name>}`. The library resolves the variable value by searching in order in the following locations:

- Java system properties
- OS environment variables
- The JNDI properties file or environment Hashtable

For example, on Linux `${HOME}` resolves to the HOME environment variable, the current user’s home directory.

A default value can be supplied using the syntax `${<variable-name>:-<default-value>}`. If no value for `<variable-name>` is found, the default value is used instead.
CHAPTER 5. EXAMPLES

This chapter demonstrates the use of AMQ JMS through example programs.

For more examples, see the JMS example suite.

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 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.

```java
# 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.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
```
public class Sender {
    public static void main(String[] args) throws Exception {
        try {
            Context context = new InitialContext();  
            ConnectionFactory factory = (ConnectionFactory) context.lookup("myFactoryLookup");
            Destination destination = (Destination) context.lookup("myDestinationLookup");

            Connection connection = factory.createConnection("<username>", "<password>");
            connection.setExceptionListener(new MyExceptionListener());
            connection.start();

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

            MessageProducer messageProducer = session.createProducer(destination);

            TextMessage message = session.createTextMessage("Message Text!");
            messageProducer.send(message, DeliveryMode.NON_PERSISTENT, Message.DEFAULT_PRIORITY, Message.DEFAULT_TIME_TO_LIVE);

            connection.close();
        } catch (Exception exp) {
            System.out.println("Caught exception, exiting.");
            exp.printStackTrace(System.out);
            System.exit(1);
        }
    }

    private static class MyExceptionListener implements ExceptionListener {
        @Override
        public void onException(JMSException exception) {
            System.out.println("Connection ExceptionListener fired, exiting.");
            exception.printStackTrace(System.out);
            System.exit(1);
        }
    }
}

1. Creates the JNDI Context to look up ConnectionFactory and Destination objects. The configuration is picked up from the jndi.properties file as detailed earlier.

2. 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.
The TextMessage is sent. It is sent non-persistent, with default priority and no expiration.

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");
            Destination destination = (Destination) context.lookup("myDestinationLookup");

            Connection connection = factory.createConnection("<username>", "<password>");
            connection.setExceptionListener(new MyExceptionListener());
            connection.start();

            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer messageConsumer = session.createConsumer(destination);
            Message message = messageConsumer.receive(5000);

            if (message == null) {
                // Handle case when message is null
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```
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. 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 4.4, “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 4.6, “SSL/TLS 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 4.7, “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 section called “Deserialization policy options” section.

6.1. 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. The default is `amqp-jms-client`.

`sasl.options.protocol`
- The protocol value used during the GSSAPI SASL process. The default is `amqp`.

`sasl.options.serverName`
- The `serverName` value used during the GSSAPI SASL process. The 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.

### 6.2. ENABLING OPENSSL SUPPORT

SSL/TLS connections can be configured to use a native OpenSSL implementation for improved performance. To use OpenSSL, the `transport.useOpenSSL` option must be enabled, and an OpenSSL support library must be available on the classpath.

To use the system-installed OpenSSL libraries on Red Hat Enterprise Linux, install the `openssl` and `apr` RPM packages and add the following dependency to your POM file:

#### Example: Adding native OpenSSL support

```xml
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-tcnative</artifactId>
  <version>2.0.29.Final-redhat-00001</version>
</dependency>
```

A list of OpenSSL library implementations is available from the Netty project.
CHAPTER 7. MESSAGE DELIVERY

7.1. HANDLING UNACKNOWLEDGED DELIVERIES

Messaging systems use message acknowledgment to track if the goal of sending a message is truly accomplished.

When a message is sent, there is a period of time after the message is sent and before it is acknowledged (the message is "in flight"). If the network connection is lost during that time, the status of the message delivery is unknown, and the delivery might require special handling in application code to ensure its completion.

The sections below describe the conditions for message delivery when connections fail.

Non-transacted producer with an unacknowledged delivery
If a message is in flight, it is sent again after reconnect, provided a send timeout is not set and has not elapsed.

No user action is required.

Transacted producer with an uncommitted transaction
If a message is in flight, it is sent again after reconnect. If the send is the first in a new transaction, then sending continues as normal after reconnect. If there are previous sends in the transaction, then the transaction is considered failed, and any subsequent commit operation throws a TransactionRolledBackException.

To ensure delivery, the user must resend any messages belonging to a failed transaction.

Transacted producer with a pending commit
If a commit is in flight, then the transaction is considered failed, and any subsequent commit operation throws a TransactionRolledBackException.

To ensure delivery, the user must resend any messages belonging to a failed transaction.

Non-transacted consumer with an unacknowledged delivery
If a message is received but not yet acknowledged, then acknowledging the message produces no error but results in no action by the client.

Because the received message is not acknowledged, the producer might resend it. To avoid duplicates, the user must filter out duplicate messages by message ID.

Transacted consumer with an uncommitted transaction
If an active transaction is not yet committed, it is considered failed, and any pending acknowledgments are dropped. Any subsequent commit operation throws a TransactionRolledBackException.

The producer might resend the messages belonging to the transaction. To avoid duplicates, the user must filter out duplicate messages by message ID.

Transacted consumer with a pending commit
If a commit is in flight, then the transaction is considered failed. Any subsequent commit operation throws a TransactionRolledBackException.

The producer might resend the messages belonging to the transaction. To avoid duplicates, the user must filter out duplicate messages by message ID.
7.2. EXTENDED SESSION ACKNOWLEDGMENT MODES

The client supports two additional session acknowledgement modes beyond those defined in the JMS specification.

**Individual acknowledge**
In this mode, messages must be acknowledged individually by the application using the `Message.acknowledge()` method used when the session is in `CLIENT_ACKNOWLEDGE` mode. Unlike with `CLIENT_ACKNOWLEDGE` mode, only the target message is acknowledged. All other delivered messages remain unacknowledged. The integer value used to activate this mode is 101.

```java
connection.createSession(false, 101);
```

**No acknowledge**
In this mode, messages are accepted at the server before being dispatched to the client, and no acknowledgment is performed by the client. The client supports two integer values to activate this mode, 100 and 257.

```java
connection.createSession(false, 100);
```
CHAPTER 8. LOGGING AND TRACING

The client uses the SLF4J API, enabling users to select a particular logging implementation based on their needs. For example, users can provide the slf4j-log4j binding to select the Log4J implementation. 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.

8.1. ENABLING PROTOCOL LOGGING

When debugging, it is sometimes 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.

You can also configure the client to emit low-level tracing of input and output bytes. To enable this, add the option transport.traceBytes=true to your connection URI and configure the org.apache.qpid.jms.transports.netty.NettyTcpTransport logger to log level DEBUG.

8.2. ENABLING DISTRIBUTED TRACING

The client offers distributed tracing based on the Jaeger implementation of the OpenTracing standard. Use the following steps to enable tracing in your application:

Procedure

1. Add the Jaeger client dependency to your POM file.

   <dependency>
   <groupId>io.jaegertracing</groupId>
   <artifactId>jaeger-client</artifactId>
   <version>${jaeger-version}</version>
   </dependency>

   ${jaeger-version} must be 1.0.0 or later.

2. Add the jms.tracing option to your connection URI. Set the value to opentracing.

   Example: A connection URI with tracing enabled

   amqps://example.net?jms.tracing=opentracing

3. Register the global tracer.

   Example: Global tracer registration

   import io.jaegertracing.Configuration;
   import io.opentracing.Tracer;
import io.opentracing.util.GlobalTracer;

public class Example {
    public static void main(String[] args) {
        Tracer tracer = Configuration.fromEnv("<service-name>").getTracer();
        GlobalTracer.registerIfAbsent(tracer);

        // ...
    }
}

4. Configure your environment for tracing.

Example: Tracing configuration

$ export JAEGGER_SAMPLER_TYPE=const
$ export JAEGGER_SAMPLER_PARAM=1
$ java -jar example.jar net.example.Example

The configuration shown here is for demonstration purposes. For more information about Jaeger configuration, see Configuration via Environment and Jaeger Sampling.

To view the traces your application captures, use the Jaeger Getting Started to run the Jaeger infrastructure and console.
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.

9.1. 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.

9.1.1. 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.

9.1.1.1. 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, omnil 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>
9.1.1.2. 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 application 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>
<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><code>string or null</code></td>
</tr>
</tbody>
</table>

9.1.2. 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.

9.1.2.1. 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 <strong>amqp-value</strong> body section containing a <strong>string</strong> or <strong>null</strong>.</td>
<td><strong>TextMessage</strong></td>
</tr>
<tr>
<td>A <strong>data</strong> body section, with the <strong>properties</strong> section <strong>content-type</strong> field set to a <strong>symbol</strong> value representing a common textual media type such as &quot;text/plain&quot;, &quot;application/xml&quot;, or &quot;application/json&quot;.</td>
<td><strong>TextMessage</strong></td>
</tr>
<tr>
<td>An <strong>amqp-value</strong> body section containing a <strong>binary</strong>.</td>
<td><strong>BytesMessage</strong></td>
</tr>
<tr>
<td>A <strong>data</strong> body section, with the <strong>properties</strong> section <strong>content-type</strong> field either not set, set to <strong>symbol</strong> value &quot;application/octet-stream&quot;, or set to any value not understood to be associated with another message type.</td>
<td><strong>BytesMessage</strong></td>
</tr>
<tr>
<td>A <strong>data</strong> body section, with the <strong>properties</strong> section <strong>content-type</strong> field set to <strong>symbol</strong> value &quot;application/x-java-serialized-object&quot;.</td>
<td><strong>ObjectMessage</strong></td>
</tr>
<tr>
<td>An <strong>amqp-value</strong> body section containing a value not covered above.</td>
<td><strong>ObjectMessage</strong></td>
</tr>
<tr>
<td>An <strong>amqp-sequence</strong> body section. This will be represented as a List inside the ObjectMessage.</td>
<td><strong>ObjectMessage</strong></td>
</tr>
</tbody>
</table>

### 9.1.2.2. 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 application property Type</th>
<th>JMS property type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean</code></td>
<td><code>boolean</code></td>
</tr>
<tr>
<td><code>byte</code></td>
<td><code>byte</code></td>
</tr>
<tr>
<td><code>short</code></td>
<td><code>short</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>int</code></td>
</tr>
</tbody>
</table>

---

CHAPTER 9. INTEROPERABILITY
<table>
<thead>
<tr>
<th>AMQP application property Type</th>
<th>JMS property type</th>
</tr>
</thead>
<tbody>
<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</td>
</tr>
<tr>
<td>null</td>
<td>String</td>
</tr>
</tbody>
</table>

### 9.2. 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 Default acceptor settings.
- 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.

### 9.3. 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 Securing network connections.
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.

A.4. REGISTERING YOUR SYSTEM FOR PACKAGES

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using downloaded release files, this step is not required.

Procedure

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.

For more information, see How to Register and Subscribe a System to the Red Hat Customer Portal.
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**

   ```xml
   <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>
   ```
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**

```xml
<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:

```xml
<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**
<table>
<thead>
<tr>
<th>Operating system</th>
<th>Filesystem path</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux or UNIX</td>
<td>/home/alice/maven-repository</td>
<td>file:/home/alice/maven-repository</td>
</tr>
<tr>
<td>Windows</td>
<td>C:\repos\red-hat</td>
<td>file:C:\repos\red-hat</td>
</tr>
</tbody>
</table>
APPENDIX C. USING AMQ BROKER WITH THE EXAMPLES

The AMQ JMS examples require a running message broker with a queue named queue. 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 queue.

   $ <broker-instance-dir>/bin/artemis queue create --name queue --address queue --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 2020-06-16 17:52:01 UTC