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

**MAKING OPEN SOURCE MORE INCLUSIVE**  ................................................................. 4

**CHAPTER 1. OVERVIEW**  ......................................................................................... 5

  1.1. KEY FEATURES  .................................................................................. 5
  1.2. SUPPORTED STANDARDS AND PROTOCOLS  .................................. 5
  1.3. SUPPORTED CONFIGURATIONS  ...................................................... 5
  1.4. TERMS AND CONCEPTS  .................................................................. 6
  1.5. DOCUMENT CONVENTIONS  ............................................................... 7

  - The sudo command
  - File paths
  - Variable text

**CHAPTER 2. INSTALLATION**  .................................................................................. 8

  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**  ........................................................................ 10

  3.1. PREREQUISITES  .................................................................................. 10
  3.2. RUNNING YOUR FIRST EXAMPLE  ....................................................... 10

**CHAPTER 4. CONFIGURATION**  ............................................................................ 11

  4.1. CONFIGURING THE JNDI INITIAL CONTEXT  ................................ 11

    - Using a jndi.properties file
    - Using a system property
    - Using the initial context API
  4.2. CONFIGURING THE CONNECTION FACTORY  ..................................... 12
  4.3. CONNECTION URIS  ............................................................................. 12

    - Failover URIs
  4.4. CONFIGURING QUEUE AND TOPIC NAMES  ...................................... 13

**CHAPTER 5. CONFIGURATION OPTIONS**  ............................................................ 14

  5.1. JMS OPTIONS  ....................................................................................... 14

    - Prefetch policy options
    - Redelivery policy options
  5.2. TCP OPTIONS  .......................................................................................... 15
  5.3. SSL/TLS OPTIONS  ................................................................................ 16
  5.4. OPENWIRE OPTIONS  .......................................................................... 17
  5.5. FAILOVER OPTIONS  .............................................................................. 17

**CHAPTER 6. MESSAGE DELIVERY**  ...................................................................... 19

  6.1. WRITING TO A STREAMED LARGE MESSAGE  .................................. 19
  6.2. READING FROM A STREAMED LARGE MESSAGE  ............................ 19

**APPENDIX A. USING YOUR SUBSCRIPTION**  ..................................................... 20

  A.1. ACCESSING YOUR ACCOUNT  ............................................................. 20
  A.2. ACTIVATING A SUBSCRIPTION  ........................................................... 20
  A.3. DOWNLOADING RELEASE FILES  ....................................................... 20
  A.4. REGISTERING YOUR SYSTEM FOR PACKAGES  ............................ 20

**APPENDIX B. USING RED HAT MAVEN REPOSITORIES**  ................................... 22

  B.1. USING THE ONLINE REPOSITORY  ....................................................... 22
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
CHAPTER 1. OVERVIEW

AMQ OpenWire JMS is a Java Message Service (JMS) 1.1 client for use in messaging applications that send and receive OpenWire messages.

AMQ OpenWire 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 2.9 Release Notes.

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

1.1. KEY FEATURES

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

1.2. SUPPORTED STANDARDS AND PROTOCOLS

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

- Version 1.1 of the Java Message Service API.
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

AMQ OpenWire 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
- 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 OpenWire JMS is supported in combination with the following AMQ components and versions:
The latest version of AMQ Broker
- 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>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactory</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. It contains</td>
</tr>
<tr>
<td></td>
<td>sessions.</td>
</tr>
<tr>
<td>Session</td>
<td>A context for producing and consuming messages. It contains message</td>
</tr>
<tr>
<td></td>
<td>producers and consumers.</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>A channel for sending messages to a destination. It has a target destination.</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>A channel for receiving messages from a destination. It has a source</td>
</tr>
<tr>
<td></td>
<td>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>An application-specific piece of information.</td>
</tr>
</tbody>
</table>

AMQ OpenWire 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:

```bash
$ cd <project-dir>
```
CHAPTER 2. INSTALLATION

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

2.1. PREREQUISITES

- You must have a subscription to access AMQ release files and repositories.
- To build programs with AMQ OpenWire JMS, you must install Apache Maven.
- To use AMQ OpenWire 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>
   
   <dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>activemq-client</artifactId>
   <version>5.11.0.redhat-630416</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 OpenWire JMS can be installed to your local filesystem as a file-based Maven repository.

Procedure

1. Use your subscription to download the AMQ Broker 7.9.0 Maven repository.zip file.

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

   ```
   $ unzip amq-broker-7.9.0-maven-repository.zip
   
   On Windows, right-click the .zip file and select Extract All.
3. Configure Maven to use the repository in the maven-repository directory inside the extracted install directory. For more information, see Section B.2, "Using a local repository".

2.4. INSTALLING THE EXAMPLES

Procedure

1. Use your subscription to download the AMQ Broker 7.9.0.zip file.

2. Extract the file contents into a directory of your choosing.
   On Linux or UNIX, use the unzip command to extract the file contents.
   ```
   $ unzip amq-broker-7.9.0.zip
   ```
   On Windows, right-click the .zip file and select Extract All.

When you extract the contents of the .zip file, a directory named amq-broker-7.9.0 is created. This is the top-level directory of the installation and is referred to as <install-dir> throughout this document.
CHAPTER 3. GETTING STARTED

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

3.1. PREREQUISITES

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

3.2. RUNNING YOUR FIRST EXAMPLE

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

Procedure

1. Use Maven to build the examples by running the following command in the `<install-dir>/examples/protocols/openwire/queue` directory.

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

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

2. Use the `java` command to run the example.
   - On Linux or UNIX:
     ```
     $ java -cp "target/classes:target/dependency/*"
     org.apache.activemq.artemis.jms.example.QueueExample
     ```
   - On Windows:
     ```
     > java -cp "target\classes;target\dependency\*"
     org.apache.activemq.artemis.jms.example.QueueExample
     ```

Running it on Linux results in the following output:

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

The source code for the example is in the `<install-dir>/examples/protocols/openwire/queue/src` directory. Additional examples are available in the `<install-dir>/examples/protocols/openwire` directory.
CHAPTER 4. CONFIGURATION

This chapter describes the process for binding the AMQ OpenWire 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.

For more information about configuring AMQ OpenWire JMS, see the [ActiveMQ user guide](#).

4.1. CONFIGURING THE JNDI INITIAL CONTEXT

JMS applications use a JNDI InitialContext object obtained from an InitialContextFactory to look up JMS objects such as the connection factory. AMQ OpenWire JMS provides an implementation of the InitialContextFactory in the org.apache.activemq.jndi.ActiveMQInitialContextFactory 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 three ways of achieving this: using a jndi.properties file, using a system property, or using the initial context API.

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

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

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

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

Using a system property
Set the `java.naming.factory.initial` system property.

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

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

Using the initial context API
Use the JNDI initial context API to set properties programatically.

**Example: Setting JNDI properties programatically**

```java
Hashtable<Object, Object> env = new Hashtable<>();
env.put("java.naming.factory.initial", "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
InitialContext context = new InitialContext(env);
```
4.2. CONFIGURING THE CONNECTION FACTORY

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

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

**The JNDI property format for connection factories**

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

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

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

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

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

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

4.3. CONNECTION URIS

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

**The connection URI format**

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

The scheme is `tcp` for unencrypted connections and `ssl` for SSL/TLS connections.

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

**Example: A connection URI**

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

**Failover URIs**

URIs used for reconnect and failover can contain multiple connection URIs. They take the following form:

**The failover URI format**

```
failover:(<connection-uri>[,.<connection-uri>])[?<option>=<value>[&<option>=<value>...]]
```
Transport options prefixed with nested are applied to each connection URI in the list.

### 4.4. CONFIGURING QUEUE AND TOPIC NAMES

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

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

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

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

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

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

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

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

```
Queue queue = (Queue) context.lookup("jobs");
Topic topic = (Topic) context.lookup("notifications");
```
CHAPTER 5. CONFIGURATION OPTIONS

This chapter lists the available configuration options for AMQ OpenWire JMS.

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

5.1. JMS OPTIONS

- **jms.username**: The user name the client uses to authenticate the connection.
- **jms.password**: The password the client uses to authenticate the connection.
- **jms.clientID**: The client ID that the client applies to the connection.
- **jms.closeTimeout**: The timeout in milliseconds for JMS close operations. The default is 15000 (15 seconds).
- **jms.connectResponseTimeout**: The timeout in milliseconds for JMS connect operations. The default is 0, meaning no timeout.
- **jms.sendTimeout**: The timeout in milliseconds for JMS send operations. The default is 0, meaning no timeout.
- **jms.checkForDuplicates**: If enabled, ignore duplicate messages. It is enabled by default.
- **jms.disableTimeStampsByDefault**: If enabled, do not timestamp messages. It is disabled by default.
- **jms.useAsyncSend**: If enabled, send messages without waiting for acknowledgment. It is disabled by default.
- **jms.alwaysSyncSend**: If enabled, send waits for acknowledgment in all delivery modes. It is disabled by default.
- **jms.useCompression**: If enabled, compress message bodies. It is disabled by default.
- **jms.useRetroactiveConsumer**: If enabled, non-durable subscribers can receive messages that were published before the subscription started. It is disabled by default.

**Prefetch policy options**

Prefetch policy determines how many messages each `MessageConsumer` fetches from the remote peer and holds in a local "prefetch" buffer.

- **jms.prefetchPolicy.queuePrefetch**: The number of messages to prefetch for queues. The default is 1000.
- **jms.prefetchPolicy.queueBrowserPrefetch**: The number of messages to prefetch for queue browsers. The default is 500.
- **jms.prefetchPolicy.topicPrefetch**: The number of messages to prefetch for non-durable topics. The default is 32766.
jms.prefetchPolicy.durableTopicPrefetch
The number of messages to prefetch for durable topics. The default is 100.

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. A higher value can result in larger batches sent at once to each consumer. To achieve more even round-robin distribution when consumers operate at different rates, use a lower value.

Redelivery policy options
Redelivery policy controls how redelivered messages are handled on the client.

jms.redeliveryPolicy.maximumRedeliveries
The number of times redelivery is attempted before the message is sent to the dead letter queue. The default is 6. -1 means no limit.

jms.redeliveryPolicy.redeliveryDelay
The time in milliseconds between redelivery attempts. This is used if initialRedeliveryDelay is 0. The default is 1000 (1 second).

jms.redeliveryPolicy.initialRedeliveryDelay
The time in milliseconds before the first redelivery attempt. The default is 1000 (1 second).

jms.redeliveryPolicy.maximumRedeliveryDelay
The maximum time in milliseconds between redelivery attempts. This is used if useExponentialBackOff is enabled. The default is 1000 (1 second). -1 means no limit.

jms.redeliveryPolicy.useExponentialBackOff
If enabled, increase redelivery delay with each subsequent attempt. It is disabled by default.

jms.redeliveryPolicy.backOffMultiplier
The multiplier for increasing the redelivery delay. The default is 5.

jms.redeliveryPolicy.useCollisionAvoidance
If enabled, adjust the redelivery delay slightly up or down to avoid collisions. It is disabled by default.

jms.redeliveryPolicy.collisionAvoidanceFactor
The multiplier for adjusting the redelivery delay. The default is 0.15.

nonBlockingRedelivery
If enabled, allow out of order redelivery, to avoid head-of-line blocking. It is disabled by default.

5.2. TCP OPTIONS

closeAsync
If enabled, close the socket in a separate thread. It is enabled by default.

connectionTimeout
The timeout in milliseconds for TCP connect operations. The default is 30000 (30 seconds). 0 means no timeout.

dynamicManagement
If enabled, allow JMX management of the transport logger. It is disabled by default.

ioBufferSize
The I/O buffer size in bytes. The default is 8192 (8 KiB).

jmxPort
The port for JMX management. The default is 1099.

** keepAlive **
If enabled, use TCP keepalive. This is distinct from the keepalive mechanism based on \KeepAliveInfo\ messages. It is disabled by default.

** logWriterName **
The name of the org.apache.activemq.transport.LogWriter implementation. Name-to-class mappings are stored in the resources/META-INF/services/org/apache/activemq/transport/logwriters directory. The default is default.

** soLinger **
The socket linger option. The default is 0.

** soTimeout **
The timeout in milliseconds for socket read operations. The default is 0, meaning no timeout.

** soWriteTimeout **
The timeout in milliseconds for socket write operations. The default is 0, meaning no timeout.

** startLogging **
If enabled, and the trace option is also enabled, log transport startup events. It is enabled by default.

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

** threadName **
If set, the name assigned to the transport thread. The remote address is appended to the name. It is unset by default.

** trace **
If enabled, log transport events to log4j.logger.org.apache.activemq.transport.TransportLogger. It is disabled by default.

** useInactivityMonitor **
If enabled, time out connections that fail to send KeepAliveInfo messages. It is enabled by default.

** useKeepAlive **
If enabled, periodically send KeepAliveInfo messages to prevent the connection from timing out. It is enabled by default.

** useLocalHost **
If enabled, make local connections using the name localhost instead of the current hostname. It is disabled by default.

### 5.3. SSL/TLS OPTIONS

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

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

** socket.keyStoreType **
The string name of the trust store type. The default is the value of java.security.KeyStore.getDefaultType().

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

**socket.trustStorePassword**

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

**socket.trustStoreType**

The string name of the trust store type. The default is the value of `java.security.KeyStore.getDefaultType()`.

**socket.enabledCipherSuites**

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

**socket.enabledProtocols**

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

### 5.4. OPENWIRE OPTIONS

**wireFormat.cacheEnabled**

If enabled, avoid excessive marshalling and bandwidth consumption by caching frequently used values. It is enabled by default.

**wireFormat.cacheSize**

The number of cache entries. The cache is per connection. The default is 1024.

**wireFormat.maxInactivityDuration**

The maximum time in milliseconds before a connection with no activity is considered dead. The default is 30000 (30 seconds).

**wireFormat.maxInactivityDurationInitialDelay**

The initial delay in milliseconds before inactivity checking begins. Note that **Initial** is misspelled. The default is 10000 (10 seconds).

**wireFormat.maxFrameSize**

The maximum frame size in bytes. The default is the value of `java.lang.Long.MAX_VALUE`.

**wireFormat.sizePrefixDisabled**

If set true, do not prefix packets with their size. It is false by default.

**wireFormat.stackTraceEnabled**

If enabled, send stack traces from exceptions on the server to the client. It is enabled by default.

**wireFormat.tcpNoDelayEnabled**

If enabled, tell the server to activate `TCP_NODELAY`. It is enabled by default.

**wireFormat.tightEncodingEnabled**

If enabled, optimize for smaller encoding on the wire. This increases CPU usage. It is enabled by default.

### 5.5. FAILOVER OPTIONS

**maxReconnectAttempts**

The number of reconnect attempts allowed before reporting the connection as failed. The default is -1, meaning no limit. 0 disables reconnect.

**maxReconnectDelay**
The maximum time in milliseconds between the second and subsequent reconnect attempts. The default is 30000 (30 seconds).

**randomize**

If enabled, randomly select one of the failover endpoints. It is enabled by default.

**reconnectDelayExponent**

The multiplier for increasing the reconnect delay backoff. The default is 2.0.

**useExponentialBackOff**

If enabled, increase the reconnect delay with each subsequent attempt. It is enabled by default.

**timeout**

The timeout in milliseconds for send operations waiting for reconnect. The default is -1, meaning no timeout.
CHAPTER 6. MESSAGE DELIVERY

6.1. WRITING TO A STREAMED LARGE MESSAGE

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

Example: Writing to a streamed large message

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

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

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

6.2. READING FROM A STREAMED LARGE MESSAGE

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

Example: Reading from a streamed large message

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

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

for (int pos = 0; pos < message.getBodyLength(); pos += buffer.length) {
    numRead = message.readBytes(buffer);
    outputStream.write(buffer, 0, numRead);
}
```
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 for this product 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 about registering your system, see one of the following resources:

- Red Hat Enterprise Linux 7 - Registering the system and managing subscriptions
- Red Hat Enterprise Linux 8 - Registering the system and managing subscriptions
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>
       </pluginRepository>
     </pluginRepositories>
   </profile>
   </profiles>
   </settings>
   ```
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 OpenWire JMS examples require a running message broker with a queue named exampleQueue. Use the procedures below to install and start the broker and define the queue.

C.1. INSTALLING THE BROKER

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

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

C.2. STARTING THE BROKER

Procedure

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

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

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

   $ example-broker/bin/artemis run

   Red Hat AMQ <version>

   ...
   2020-06-03 12:12:12,336 INFO [org.apache.activemq.artemis.core.server] AMQ221007: Server is now live
   ...

C.3. CREATING A QUEUE

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

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

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

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

C.4. STOPPING THE BROKER
When you are done running the examples, use the **artemis stop** command to stop the broker.

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

*Revised on 2021-05-07 10:16:37 UTC*