Red Hat AMQ 7.2

Using the AMQ Python Client

For Use with AMQ Clients 2.2

Last Updated: 2018-11-15

Red Hat AMQ 7.2 Using the AMQ Python Client

For Use with AMQ Clients 2.2

Legal Notice

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

AMQ Python is a library for developing messaging applications. It enables you to write Python applications that send and receive AMQP messages.

AMQ Python 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 2.2 Release Notes.

AMQ Python is based on the Proton API from Apache Qpid.

1.1. KEY FEATURES

- An event-driven API that simplifies integration with existing applications
- SSL/TLS for secure communication
- Flexible SASL authentication
- Automatic reconnect and failover
- Seamless conversion between AMQP and language-native data types
- Access to all the features and capabilities of AMQP 1.0

1.2. SUPPORTED STANDARDS AND PROTOCOLS

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

- Version 1.0 of the Advanced Message Queueing Protocol (AMQP)
- Versions 1.0, 1.1, and 1.2 of the Transport Layer Security (TLS) protocol, the successor to SSL
- Simple Authentication and Security Layer (SASL) mechanisms supported by Cyrus SASL, including ANONYMOUS, PLAIN, SCRAM, EXTERNAL, and GSSAPI (Kerberos)
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

AMQ Python supports the following OS and language versions:

1. Red Hat Enterprise Linux 6 and 7 with Python 2.6 and 2.7
2. Microsoft Windows Server 2012 R2 with Python 2.7

For more information, see Red Hat AMQ 7 Supported Configurations.

1.4. TERMS AND CONCEPTS

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

Table 1.1. API terms

Entity	Description
Container	A top-level container of connections
Connection	A channel for communication between two peers on a network
Session	A context for sending and receiving messages
Sender	A channel for sending messages to a target
Receiver	A channel for receiving messages from a source
Source	A named point of origin for messages
Target	A named destination for messages
Message	A mutable holder of application data
Delivery	A message transfer

AMQ Python sends and receives *messages*. Messages are transferred between connected peers over *senders* and *receivers*. Senders and receivers are established over *sessions*. Sessions are established over *connections*. Connections are established between two uniquely identified *containers*. Though a connection can have multiple sessions, often this is not needed. The API allows you to ignore sessions unless you require them.

A sending peer creates a sender to send messages. The sender has a *target* that identifies a queue or topic at the remote peer. A receiving peer creates a receiver to receive messages. The receiver has a *source* that identifies a queue or topic at the remote peer.

The sending of a message is called a *delivery*. The message is the content sent, including all metadata such as headers and annotations. The delivery is the protocol exchange associated with the transfer of that content.

To indicate that a delivery is complete, either the sender or the receiver settles it. When the other side learns that it has been settled, it will no longer communicate about that delivery. The receiver can also indicate whether it accepts or rejects the message.

1.5. DOCUMENT CONVENTIONS

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

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

CHAPTER 2. INSTALLATION

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

2.1. PREREQUISITES

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

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

AMQ Python is distributed as a set of RPM packages for Red Hat Enterprise Linux. Follow these steps to install them.

1. Use the **`subscription-manager`** command to subscribe to the required package repositories.

   Red Hat Enterprise Linux 6

   ```
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-6-server-rpms
   ```

   Red Hat Enterprise Linux 7

   ```
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-7-server-rpms
   ```

2. Use the **`yum`** command to install the **`python-qpid-proton`** and **`python-qpid-proton-docs`** packages.

   ```
   $ sudo yum install python-qpid-proton python-qpid-proton-docs
   ```

CHAPTER 3. GETTING STARTED

This chapter guides you through a simple exercise to help you get started using AMQ Python.

3.1. PREPARING THE BROKER

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

Procedure

1. Install the broker.
2. Create a broker instance. Enable anonymous access.
3. Start the broker instance and check the console for any critical errors logged during startup.

```
$ <broker-instance-dir>/bin/artemis run
...
14:43:20,158 INFO
[org.apache.activemq.artemis.integration.bootstrap] AMQ101000:
Starting ActiveMQ Artemis Server
...
15:01:39,686 INFO  [org.apache.activemq.artemis.core.server]
AMQ221020: Started Acceptor at 0.0.0.0:5672 for protocols [AMQP]
...
15:01:39,691 INFO  [org.apache.activemq.artemis.core.server]
AMQ221007: Server is now live
```

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

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

You are prompted to answer a series of questions. For yes or no questions, type **N**. Otherwise, press Enter to accept the default value.

3.2. RUNNING HELLO WORLD

The Hello World example sends a message to the **examples** queue on the broker and then fetches it back. On success it prints **Hello World!** to the console.

Using a new terminal window, change directory to the AMQ Python examples directory and run the **helloworld.py** example.

```
$ cd /usr/share/proton-0.26.0/examples/python/
$ python helloworld.py
Hello World!
```

CHAPTER 4. EXAMPLES

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

See the Qpid Proton Python examples for more sample programs.

4.1. SENDING MESSAGES

This client program connects to a server using **<connection-url>**, creates a sender for target **<address>**, sends a message containing **<message-body>**, closes the connection, and exits.

Example: Sending messages

```
from __future__ import print_function

import sys

from proton import Message
from proton.handlers import MessagingHandler
from proton.reactor import Container

class SendHandler(MessagingHandler):
    def __init__(self, conn_url, address, message_body):
        super(SendHandler, self).__init__()

        self.conn_url = conn_url
        self.address = address
        self.message_body = message_body

    def on_start(self, event):
        conn = event.container.connect(self.conn_url)
        event.container.create_sender(conn, self.address)

    def on_link_opened(self, event):
        print("SEND: Opened sender for target address '{0}'".format
              (event.sender.target.address))

    def on_sendable(self, event):
        message = Message(self.message_body)
        event.sender.send(message)

        print("SEND: Sent message '{0}'".format(message.body))

        event.sender.close()
        event.connection.close()

def main():
    try:
        conn_url, address, message_body = sys.argv[1:4]
    except ValueError:
        sys.exit("Usage: send.py <connection-url> <address> <message-
body>")

    handler = SendHandler(conn_url, address, message_body)
    container = Container(handler)
```

```
    container.run()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Running the example

To run the example program, copy it to a local file and invoke it using the **python** command.

```
$ python send.py amqp://localhost queue1 hello
```

4.2. RECEIVING MESSAGES

This client program connects to a server using **<connection-url>**, creates a receiver for source **<address>**, and receives messages until it is terminated or it reaches **<count>** messages.

Example: Receiving messages

```
from __future__ import print_function

import sys

from proton.handlers import MessagingHandler
from proton.reactor import Container

class ReceiveHandler(MessagingHandler):
    def __init__(self, conn_url, address, desired):
        super(ReceiveHandler, self).__init__()

        self.conn_url = conn_url
        self.address = address
        self.desired = desired
        self.received = 0

    def on_start(self, event):
        conn = event.container.connect(self.conn_url)
        event.container.create_receiver(conn, self.address)

    def on_link_opened(self, event):
        print("RECEIVE: Created receiver for source address '{0}'".format
              (self.address))

    def on_message(self, event):
        message = event.message

        print("RECEIVE: Received message '{0}'".format(message.body))

        self.received += 1

        if self.received == self.desired:
            event.receiver.close()
            event.connection.close()
```

```
def main():
    try:
        conn_url, address = sys.argv[1:3]
    except ValueError:
        sys.exit("Usage: receive.py <connection-url> <address> [<message-
count>]")

    try:
        desired = int(sys.argv[3])
    except (IndexError, ValueError):
        desired = 0

    handler = ReceiveHandler(conn_url, address, desired)
    container = Container(handler)
    container.run()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Running the example

To run the example program, copy it to a local file and invoke it using the **python** command.

```
$ python receive.py amqp://localhost queue1
```

CHAPTER 5. USING THE API

This chapter explains how to use the AMQ Python API to perform common messaging tasks.

5.1. BASIC OPERATION

5.1.1. Handling messaging events

AMQ Python is an asynchronous event-driven API. To define how an application handles events, the user implements callback methods on the **`MessagingHandler`** class. These methods are then called as network activity or timers trigger new events.

Example: Handling messaging events

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        print("The container event loop has started")

    def on_sendable(self, event):
        print("A message can be sent")

    def on_message(self, event):
        print("A message is received")
```

These are only a few common-case events. The full set is documented in the API reference.

The **`event`** argument has attributes for accessing the object the event is regarding. Attributes with no relevance to a particular event are null.

Example: Accessing event objects

```
event.container
event.connection
event.session
event.sender
event.receiver
event.delivery
event.message
```

5.1.2. Creating a container

The container is the top-level API object. It is the entry point for creating connections, and it is responsible for running the main event loop. It is often constructed with a global event handler.

Example: Creating a container

```
handler = ExampleHandler()
container = Container(handler)
container.run()
```

Setting the container identity

Each container instance has a unique identity called the container ID. When AMQ Python makes a connection, it sends the container ID to the remote peer. To set the container ID, pass it to the **`Container`** constructor.

Example: Setting the container identity

```
container = Container(handler, "job-processor-3")
```

If the user does not set the ID, the library will generate a UUID when the container is constucted.

5.2. NETWORK CONNECTIONS

5.2.1. Connection URLs

Connection URLs encode the information used to establish new connections.

Connection URL syntax

```
scheme://host[:port]
```

- *Scheme* - The connection transport, either **`amqp`** for unencrypted TCP or **`amqps`** for TCP with SSL/TLS encryption.
- *Host* - The remote network host. The value can be a hostname or a numeric IP address. IPv6 addresses must be enclosed in square brackets.
- *Port* - The remote network port. This value is optional. The default value is 5672 for the **`amqp`** scheme and 5671 for the **`amqps`** scheme.

Connection URL examples

```
amqps://example.com
amqps://example.net:56720
amqp://127.0.0.1
amqp://[::1]:2000
```

5.2.2. Creating outgoing connections

To connect to a remote server, call the **`Container.connect()`** method with a connection URL. This is typically done inside the **`MessagingHandler.on_start()`** method.

Example: Creating outgoing connections

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        connection = event.container.connect("amqp://example.com")

    def on_connection_opened(self, event):
        print("Connection", **event.connection, "is open")
```

See the Section 5.4, “Security” section for information about creating secure connections.

5.2.3. Configuring reconnect

Reconnect allows a client to recover from lost connections. It is used to ensure that the components in a distributed system reestablish communication after temporary network or component failures.

AMQ Python enables reconnect by default. If a connection is lost or a connection attempt fails, the client will try again after a brief delay. The delay increases exponentially for each new attempt, up to a default maximum of 10 seconds.

To disable reconnect, set the **`reconnect`** connection option to **`False`**.

Example: Disabling reconnect

```
container.connect("amqp://example.com", reconnect=False)
```

To control the delays between connection attempts, define a class implementing the **`reset`** and **`next`** methods and set the **`reconnect`** connection option to an instance of that class.

Example: Configuring reconnect

```
class ExampleReconnect(object):
    def __init__(self):
        self.delay = 0

    def reset(self):
        self.delay = 0

    def next(self):
        if self.delay == 0:
            self.delay = 0.1
        else:
            self.delay = min(10, 2 * self.delay)

        return self.delay

container.connect("amqp://example.com", reconnect=ExampleReconnect())
```

The **`next`** method returns the next delay in seconds. The **`reset`** method is called once before the reconnect process begins.

5.2.4. Configuring failover

AMQ Python allows you to configure multiple connection endpoints. If connecting to one fails, the client attempts to connect to the next in the list. If the list is exhausted, the process starts over.

To specify multiple connection endpoints, set the **`urls`** connection option to a list of connection URLs.

Example: Configuring failover

```
urls = ["amqp://alpha.example.com", "amqp://beta.example.com"]
container.connect(urls=urls)
```

It is an error to use the **`url`** and **`urls`** options at the same time.

5.3. MESSAGE DELIVERY

5.3.1. Sending messages

To send a message, override the **on_sendable** event handler and call the **Sender.send()** method. The **sendable** event fires when the **Sender** has enough credit to send at least one message.

Example: Sending messages

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        sender = event.container.create_sender(conn, "jobs")

    def on_sendable(self, event):
        message = Message(self.message_body)
        event.sender.send(message)
```

5.3.2. Tracking sent messages

When a message is sent, the sender can keep a reference to the **delivery** object representing the transfer. After the message is delivered, the receiver accepts or rejects it. The sender is notified of the outcome for each delivery.

To monitor the outcome of a sent message, override the **on_accepted** and **on_rejected** event handlers and map the delivery state update to the delivery returned from **send()**.

Example: Tracking sent messages

```
def on_sendable(self, event):
    message = Message(self.message_body)
    delivery = event.sender.send(message)

def on_accepted(self, event):
    print("Delivery", event.delivery, "is accepted")

def on_rejected(self, event):
    print("Delivery", event.delivery, "is rejected")
```

5.3.3. Receiving messages

To receive a message, create a receiver and override the **on_message** event handler.

Example: Receiving messages

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        receiver = event.container.create_receiver(conn, "jobs")

    def on_message(self, event):
        print("Received message", event.message, "from", event.receiver)
```

5.3.4. Acknowledging received messages

To explicitly accept or reject a delivery, use the **`Delivery.update()`** method with the **`ACCEPTED`** or **`REJECTED`** state in the **`on_message`** event handler.

Example: Acknowledging received messages

```
def on_message(self, event):
    try:
        process_message(event.message)
        event.delivery.update(ACCEPTED)
    except:
        event.delivery.update(REJECTED)
```

By default, if you do not explicity acknowledge a delivery, then the library accepts it after **`on_message`** returns. To disable this behavior, set the **`auto_accept`** receiver option to false.

5.4. SECURITY

5.4.1. Securing connections with SSL/TLS

AMQ Python uses SSL/TLS to encrypt communication between clients and servers.

To connect to a remote server with SSL/TLS, use a connection URL with the **`amqps`** scheme.

Example: Enabling SSL/TLS

```
container.connect("amqps://example.com")
```

5.4.2. Connecting with a user and password

AMQ Python can authenticate connections with a user and password.

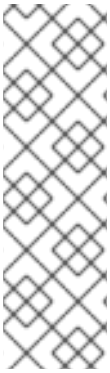
To specify the credentials used for authentication, set the **`user`** and **`password`** options on the **`connect`** method.

Example: Connecting with a user and password

```
container.connect("amqps://example.com", user="alice", password="secret")
```

5.4.3. Configuring SASL authentication

AMQ Python uses the SASL protocol to perform authentication. SASL can use a number of different authentication *mechanisms*. When two network peers connect, they exchange their allowed mechanisms, and the strongest mechanism allowed by both is selected.



NOTE

The client uses Cyrus SASL to perform authentication. Cyrus SASL uses plug-ins to support specific SASL mechanisms. Before you can use a particular SASL mechanism, the relevant plug-in must be installed. For example, you need the **`cyrus-sasl-plain`** plug-in in order to use SASL PLAIN authentication.

To see a list of Cyrus SASL plug-ins in Red Hat Enterprise Linux, use the **`yum search cyrus-sasl`** command. To install a Cyrus SASL plug-in, use the **`yum install PLUG-IN`** command.

By default, AMQ Python allows all of the mechanisms supported by the local SASL library configuration. To restrict the allowed mechanisms and thereby control what mechanisms can be negotiated, use the **`allowed_mechs`** connection option. It takes a string containing a space-separated list of mechanism names.

Example: Configuring SASL authentication

```
container.connect("amqps://example.com", allowed_mechs="ANONYMOUS")
```

This example forces the connection to authenticate using the **`ANONYMOUS`** mechanism even if the server we connect to offers other options. Valid mechanisms include **`ANONYMOUS`**, **`PLAIN`**, **`SCRAM-SHA-256`**, **`SCRAM-SHA-1`**, **`GSSAPI`**, and **`EXTERNAL`**.

AMQ Python enables SASL by default. To disable it, set the **`sasl_enabled`** connection option to false.

Example: Disabling SASL

```
event.container.connect("amqps://example.com", sasl_enabled=False)
```

5.4.4. Authenticating using Kerberos

Kerberos is a network protocol for centrally managed authentication based on the exchange of encrypted tickets. See Using Kerberos for more information.

1. Configure Kerberos in your operating system. See Configuring Kerberos to set up Kerberos on Red Hat Enterprise Linux.
2. Enable the **`GSSAPI`** SASL mechanism in your client application.

   ```
   container.connect("amqps://example.com", allowed_mechs="GSSAPI")
   ```

3. Use the **`kinit`** command to authenticate your user credentials and store the resulting Kerberos ticket.

   ```
   $ kinit USER@REALM
   ```

4. Run the client program.

5.5. MORE INFORMATION

For more information, see the API reference.

CHAPTER 6. INTEROPERABILITY

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

6.1. INTEROPERATING WITH OTHER AMQP CLIENTS

AMQP messages are composed using the AMQP type system. This common format is one of the reasons AMQP clients in different languages are able to interoperate with each other.

When sending messages, AMQ Python automatically converts language-native types to AMQP-encoded data. When receiving messages, the reverse conversion takes place.

NOTE

More information about AMQP types is available at the interactive type reference maintained by the Apache Qpid project.

Table 6.1. AMQP types

AMQP type	Description
`null`	An empty value
`boolean`	A true or false value
`char`	A single Unicode character
`string`	A sequence of Unicode characters
`binary`	A sequence of bytes
`byte`	A signed 8-bit integer
`short`	A signed 16-bit integer
`int`	A signed 32-bit integer
`long`	A signed 64-bit integer
`ubyte`	An unsigned 8-bit integer
`ushort`	An unsigned 16-bit integer
`uint`	An unsigned 32-bit integer
`ulong`	An unsigned 64-bit integer
`float`	A 32-bit floating point number

AMQP type	Description
`double`	A 64-bit floating point number
`array`	A sequence of values of a single type
`list`	A sequence of values of variable type
`map`	A mapping from distinct keys to values
`uuid`	A universally unique identifier
`symbol`	A 7-bit ASCII string from a constrained domain
`timestamp`	An absolute point in time

Table 6.2. AMQ Python types before encoding and after decoding

AMQP type	AMQ Python type before encoding	AMQ Python type after decoding
`null`	`None`	`None`
`boolean`	`bool`	`bool`
`char`	`proton.char`	`unicode`
`string`	`unicode`	`unicode`
`binary`	`bytes`	`bytes`
`byte`	`proton.byte`	`int`
`short`	`proton.short`	`int`
`int`	`proton.int32`	`long`
`long`	`long`	`long`
`ubyte`	`proton.ubyte`	`long`
`ushort`	`proton.ushort`	`long`
`uint`	`proton.uint`	`long`
`ulong`	`proton.ulong`	`long`

AMQP type	AMQ Python type before encoding	AMQ Python type after decoding
`float`	`proton.float32`	`float`
`double`	`float`	`float`
`array`	`proton.Array`	`proton.Array`
`list`	`list`	`list`
`map`	`dict`	`dict`
`symbol`	`proton.symbol`	`str`
`timestamp`	`proton.timestamp`	`long`

Table 6.3. AMQ Python and other AMQ client types (1 of 2)

AMQ Python type before encoding	AMQ C++ type	AMQ JavaScript type
`None`	`nullptr`	`null`
`bool`	`bool`	`boolean`
`proton.char`	`wchar_t`	`number`
`unicode`	`std::string`	`string`
`bytes`	`proton::binary`	`string`
`proton.byte`	`int8_t`	`number`
`proton.short`	`int16_t`	`number`
`proton.int32`	`int32_t`	`number`
`long`	`int64_t`	`number`
`proton.ubyte`	`uint8_t`	`number`
`proton.ushort`	`uint16_t`	`number`
`proton.uint`	`uint32_t`	`number`
`proton.ulong`	`uint64_t`	`number`

AMQ Python type before encoding	AMQ C++ type	AMQ JavaScript type
`proton.float32`	`float`	`number`
`float`	`double`	`number`
`proton.Array`	-	`Array`
`list`	`std::vector`	`Array`
`dict`	`std::map`	`object`
`uuid.UUID`	`proton::uuid`	`number`
`proton.symbol`	`proton::symbol`	`string`
`proton.timestamp`	`proton::timestamp`	`number`

Table 6.4. AMQ Python and other AMQ client types (2 of 2)

AMQ Python type before encoding	AMQ .NET type	AMQ Ruby type
`None`	`null`	`nil`
`bool`	`System.Boolean`	`true, false`
`proton.char`	`System.Char`	`String`
`unicode`	`System.String`	`String`
`bytes`	`System.Byte[]`	`String`
`proton.byte`	`System.SByte`	`Integer`
`proton.short`	`System.Int16`	`Integer`
`proton.int32`	`System.Int32`	`Integer`
`long`	`System.Int64`	`Integer`
`proton.ubyte`	`System.Byte`	`Integer`
`proton.ushort`	`System.UInt16`	`Integer`
`proton.uint`	`System.UInt32`	`Integer`

AMQ Python type before encoding	AMQ .NET type	AMQ Ruby type
`proton.ulong`	`System.UInt64`	`Integer`
`proton.float32`	`System.Single`	`Float`
`float`	`System.Double`	`Float`
`proton.Array`	-	`Array`
`list`	`Amqp.List`	`Array`
`dict`	`Amqp.Map`	`Hash`
`uuid.UUID`	`System.Guid`	-
`proton.symbol`	`Amqp.Symbol`	`Symbol`
`proton.timestamp`	`System.DateTime`	`Time`

6.2. INTEROPERATING WITH AMQ JMS

AMQP defines a standard mapping to the JMS messaging model. This section discusses the various aspects of that mapping. For more information, see the AMQ JMS Interoperability chapter.

JMS message types

AMQ Python provides a single message type whose body type can vary. By contrast, the JMS API uses different message types to represent different kinds of data. The table below indicates how particular body types map to JMS message types.

For more explicit control of the resulting JMS message type, you can set the `x-opt-jms-msg-type` message annotation. See the AMQ JMS Interoperability chapter for more information.

Table 6.5. AMQ Python and JMS message types

AMQ Python body type	JMS message type
`unicode`	`TextMessage`
`None`	`TextMessage`
`bytes`	`BytesMessage`
Any other type	`ObjectMessage`

6.3. CONNECTING TO AMQ BROKER

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

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

6.4. CONNECTING TO AMQ INTERCONNECT

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

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

APPENDIX A. USING YOUR SUBSCRIPTION

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

Accessing your account

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

Activating a subscription

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

Downloading zip and tar files

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

1. 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 **JBOSS INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Registering your system for packages

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

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

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

Revised on 2018-11-15 13:02:23 UTC