Red Hat AMQ 2021.Q3

Using the AMQ JavaScript Client

For Use with AMQ Clients 2.10

Last Updated: 2021-08-24

Red Hat AMQ 2021.Q3 Using the AMQ JavaScript Client

For Use with AMQ Clients 2.10

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

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

AMQ JavaScript 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 2.10 Release Notes .

AMQ JavaScript is based on the Rhea messaging library. For detailed API documentation, see the AMQ JavaScript API reference.

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 JavaScript supports the following industry-recognized standards and network protocols:

- Version 1.0 of the Advanced Message Queueing Protocol (AMQP)
- 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 ANONYMOUS, PLAIN, and EXTERNAL
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

Refer to Red Hat AMQ 7 Supported Configurations on the Red Hat Customer Portal for current information regarding AMQ JavaScript 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.

Entity	Description
Connection	A channel for communication between two peers on a network. It contains sessions.
Session	A context for sending and receiving messages. It contains senders and receivers.
Sender	A channel for sending messages to a target. It has a target.
Receiver	A channel for receiving messages from a source. It has a source.
Source	A named point of origin for messages.
Target	A named destination for messages.
Message	An application-specific piece of information.
Delivery	A message transfer.

AMQ JavaScript 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

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 JavaScript in your environment.

2.1. PREREQUISITES

- You must have a subscription to access AMQ release files and repositories.
- To use AMQ JavaScript, you must install Node.js in your environment. See the Node.js website for more information.
- AMQ JavaScript depends on the Node.js **debug** module. See the npm page for installation instructions.

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

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 Clients** entry in the **INTEGRATION AND AUTOMATION** category.
3. Click **Red Hat AMQ Clients** The **Software Downloads** page opens.
4. Download the **AMQ Clients 2.10.0 JavaScript**.zip file.
5. Use the **unzip** command to extract the file contents into a directory of your choosing.

```
$ unzip amq-clients-2.10.0-javascript.zip
```

When you extract the contents of the .zip file, a directory named **amq-clients-2.10.0-javascript** is created. This is the top-level directory of the installation and is referred to as ***<install-dir>*** throughout this document.

To configure your environment to use the installed library, add the **node_modules** directory to the **NODE_PATH** environment variable.

```
$ cd amq-clients-2.10.0-javascript
$ export NODE_PATH=$PWD/node_modules:$NODE_PATH
```

To make this configuration take effect for all new console sessions, set **NODE_PATH** in your **$HOME/.bashrc** file.

To test your installation, use the following command. It prints **OK** to the console if it successfully imports the installed library.

```
$ node -e 'require("rhea")' && echo OK
OK
```

2.3. INSTALLING ON MICROSOFT WINDOWS

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 Clients** entry in the **INTEGRATION AND AUTOMATION** category.
3. Click **Red Hat AMQ Clients** The **Software Downloads** page opens.
4. Download the **AMQ Clients 2.10.0 JavaScript**.zip file.
5. Extract the file contents into a directory of your choosing by right-clicking on the zip file and selecting **Extract All**.

When you extract the contents of the .zip file, a directory named **amq-clients-2.10.0-javascript** is created. This is the top-level directory of the installation and is referred to as ***<install-dir>*** throughout this document.

To configure your environment to use the installed library, add the **node_modules** directory to the **NODE_PATH** environment variable.

```
$ cd amq-clients-2.10.0-javascript
$ set NODE_PATH=%cd%\node_modules;%NODE_PATH%
```

2.4. PREPARING THE LIBRARY FOR USE IN BROWSERS

AMQ JavaScript can run inside a web browser. To create a browser-compatible version of the library, use the **npm run browserify** command.

```
$ cd amq-clients-2.10.0-javascript/node_modules/rhea
$ npm install
$ npm run browserify
```

This produces a file named **rhea.js** that can be used in browser-based applications.

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

- You must complete the installation procedure for your environment.
- You must have an AMQP 1.0 message broker listening for connections on interface **localhost** and port **5672**. It must have anonymous access enabled. For more information, see Starting the broker.
- You must have a queue named **examples**. For more information, see Creating a queue.

3.2. RUNNING HELLO WORLD ON RED HAT ENTERPRISE LINUX

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

Change to the examples directory and run the **helloworld.js** example.

```
$ cd <install-dir>/node_modules/rhea/examples
$ node helloworld.js
Hello World!
```

3.3. RUNNING HELLO WORLD ON MICROSOFT WINDOWS

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

Change to the examples directory and run the **helloworld.js** example.

```
> cd <install-dir>/node_modules/rhea/examples
> node helloworld.js
Hello World!
```

CHAPTER 4. EXAMPLES

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

For more examples, see the AMQ JavaScript example suite and the Rhea examples.

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

```
"use strict";

var rhea = require("rhea");
var url = require("url");

if (process.argv.length !== 5) {
    console.error("Usage: send.js <connection-url> <address> <message-body>");
    process.exit(1);
}

var conn_url = url.parse(process.argv[2]);
var address = process.argv[3];
var message_body = process.argv[4];

var container = rhea.create_container();

container.on("sender_open", function (event) {
    console.log("SEND: Opened sender for target address '" +
                event.sender.target.address + "'");
});

container.on("sendable", function (event) {
    var message = {
        body: message_body
    };

    event.sender.send(message);

    console.log("SEND: Sent message '" + message.body + "'");

    event.sender.close();
    event.connection.close();
});

var opts = {
    host: conn_url.hostname,
    port: conn_url.port || 5672,
    // To connect with a user and password:
    // username: "<username>",
    // password: "<password>",
};
```

```
var conn = container.connect(opts);
conn.open_sender(address);
```

Running the example

To run the example program, copy it to a local file and invoke it using the **node** command. For more information, see Chapter 3, *Getting started*.

```
$ node send.js 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

```
"use strict";

var rhea = require("rhea");
var url = require("url");

if (process.argv.length !== 4 && process.argv.length !== 5) {
    console.error("Usage: receive.js <connection-url> <address> [<message-count>]");
    process.exit(1);
}

var conn_url = url.parse(process.argv[2]);
var address = process.argv[3];
var desired = 0;
var received = 0;

if (process.argv.length === 5) {
    desired = parseInt(process.argv[4]);
}

var container = rhea.create_container();

container.on("receiver_open", function (event) {
    console.log("RECEIVE: Opened receiver for source address '" +
                event.receiver.source.address + "'");
});

container.on("message", function (event) {
    var message = event.message;

    console.log("RECEIVE: Received message '" + message.body + "'");

    received++;

    if (received == desired) {
        event.receiver.close();
        event.connection.close();
    }
```

```
});

var opts = {
    host: conn_url.hostname,
    port: conn_url.port || 5672,
    // To connect with a user and password:
    // username: "<username>",
    // password: "<password>",
};

var conn = container.connect(opts);
conn.open_receiver(address);
```

Running the example

To run the example program, copy it to a local file and invoke it using the **python** command. For more information, see Chapter 3, *Getting started*.

```
$ node receive.js amqp://localhost queue1
```

CHAPTER 5. USING THE API

For more information, see the AMQ JavaScript API reference and AMQ JavaScript example suite .

5.1. HANDLING MESSAGING EVENTS

AMQ JavaScript is an asynchronous event-driven API. To define how the application handles events, the user registers event-handling functions on the **container** object. These functions are then called as network activity or timers trigger new events.

Example: Handling messaging events

```
var rhea = require("rhea");
var container = rhea.create_container();

container.on("sendable", function (event) {
    console.log("A message can be sent");
});

container.on("message", function (event) {
    console.log("A message is received");
});
```

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

5.2. ACCESSING EVENT-RELATED OBJECTS

The **event** argument has attributes for accessing the object the event is regarding. For example, the **connection_open** event sets the event **connection** attribute.

In addition to the primary object for the event, all objects that form the context for the event are set as well. Attributes with no relevance to a particular event are null.

Example: Accessing event-related objects

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

5.3. 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

```
var rhea = require("rhea");
var container = rhea.create_container();
```

5.4. SETTING THE CONTAINER IDENTITY

Each container instance has a unique identity called the container ID. When AMQ JavaScript makes a network connection, it sends the container ID to the remote peer. To set the container ID, pass the **id** option to the **create_container** method.

Example: Setting the container identity

```
var container = rhea.create_container({id: "job-processor-3"});
```

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

CHAPTER 6. NETWORK CONNECTIONS

6.1. CREATING OUTGOING CONNECTIONS

To connect to a remote server, pass connection options containing the host and port to the **container.connect()** method.

Example: Creating outgoing connections

```
container.on("connection_open", function (event) {
    console.log("Connection " + event.connection + " is open");
});

var opts = {
    host: "example.com",
    port: 5672
};

container.connect(opts);
```

The default host is **localhost**. The default port is 5672.

For information about creating secure connections, Chapter 7, *Security*.

6.2. 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 JavaScript enables reconnect by default. If 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 60 seconds.

To disable reconnect, set the **reconnect** connection option to **false**.

Example: Disabling reconnect

```
var opts = {
    host: "example.com",
    reconnect: false
};

container.connect(opts);
```

To control the delays between connection attempts, set the **initial_reconnect_delay** and **max_reconnect_delay** connection options. Delay options are specified in milliseconds.

To limit the number of reconnect attempts, set the **reconnect_limit** option.

Example: Configuring reconnect

```
var opts = {
    host: "example.com",
```

```
    initial_reconnect_delay: 100,
    max_reconnect_delay: 60 * 1000,
    reconnect_limit: 10
};

container.connect(opts);
```

6.3. CONFIGURING FAILOVER

AMQ JavaScript allows you to configure alternate connection endpoints programatically.

To specify multiple connection endpoints, define a function that returns new connection options and pass the function in the **connection_details** option. The function is called once for each connection attempt.

Example: Configuring failover

```
var hosts = ["alpha.example.com", "beta.example.com"];
var index = -1;

function failover_fn() {
    index += 1;

    if (index == hosts.length) index = 0;

    return {host: hosts[index].hostname};
};

var opts = {
    host: "example.com",
    connection_details: failover_fn
}

container.connect(opts);
```

This example implements repeating round-robin failover for a list of hosts. You can use this interface to implement your own failover behavior.

6.4. ACCEPTING INCOMING CONNECTIONS

AMQ JavaScript can accept inbound network connections, enabling you to build custom messaging servers.

To start listening for connections, use the **container.listen()** method with options containing the local host address and port to listen on.

Example: Accepting incoming connections

```
container.on("connection_open", function (event) {
    console.log("New incoming connection " + event.connection);
});

var opts = {
    host: "0.0.0.0",
```

```
    port: 5672
};

container.listen(opts);
```

The special IP address **0.0.0.0** listens on all available IPv4 interfaces. To listen on all IPv6 interfaces, use **[::0]**.

For more information, see the server receive.js example.

CHAPTER 7. SECURITY

7.1. SECURING CONNECTIONS WITH SSL/TLS

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

To connect to a remote server with SSL/TLS, set the **transport** connection option to **tls**.

Example: Enabling SSL/TLS

```
var opts = {
    host: "example.com",
    port: 5671,
    transport: "tls"
};

container.connect(opts);
```

NOTE

By default, the client will reject connections to servers with untrusted certificates. This is sometimes the case in test environments. To bypass certificate authorization, set the **rejectUnauthorized** connection option to **false**. Be aware that this compromises the security of your connection.

7.2. CONNECTING WITH A USER AND PASSWORD

AMQ JavaScript can authenticate connections with a user and password.

To specify the credentials used for authentication, set the **username** and **password** connection options.

Example: Connecting with a user and password

```
var opts = {
    host: "example.com",
    username: "alice",
    password: "secret"
};

container.connect(opts);
```

7.3. CONFIGURING SASL AUTHENTICATION

AMQ JavaScript 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.

AMQ JavaScript enables SASL mechanisms based on the presence of user and password information. If the user and password are both specified, **PLAIN** is used. If only a user is specified, **ANONYMOUS** is used. If neither is specified, SASL is disabled.

CHAPTER 8. SENDERS AND RECEIVERS

The client uses sender and receiver links to represent channels for delivering messages. Senders and receivers are unidirectional, with a source end for the message origin, and a target end for the message destination.

Sources and targets often point to queues or topics on a message broker. Sources are also used to represent subscriptions.

8.1. CREATING QUEUES AND TOPICS ON DEMAND

Some message servers support on-demand creation of queues and topics. When a sender or receiver is attached, the server uses the sender target address or the receiver source address to create a queue or topic with a name matching the address.

The message server typically defaults to creating either a queue (for one-to-one message delivery) or a topic (for one-to-many message delivery). The client can indicate which it prefers by setting the **queue** or **topic** capability on the source or target.

To select queue or topic semantics, follow these steps:

1. Configure your message server for automatic creation of queues and topics. This is often the default configuration.

2. Set either the **queue** or **topic** capability on your sender target or receiver source, as in the examples below.

Example: Sending to a queue created on demand

```
var conn = container.connect({host: "example.com"});

var sender_opts = {
    target: {
        address: "jobs",
        capabilities: ["queue"]
    }
}

conn.open_sender(sender_opts);
```

Example: Receiving from a topic created on demand

```
var conn = container.connect({host: "example.com"});

var receiver_opts = {
    source: {
        address: "notifications",
        capabilities: ["topic"]
    }
}

conn.open_receiver(receiver_opts);
```

For more details, see the following examples:

- queue-send.js
- queue-receive.js
- topic-send.js
- topic-receive.js

8.2. CREATING DURABLE SUBSCRIPTIONS

A durable subscription is a piece of state on the remote server representing a message receiver. Ordinarily, message receivers are discarded when a client closes. However, because durable subscriptions are persistent, clients can detach from them and then re-attach later. Any messages received while detached are available when the client re-attaches.

Durable subscriptions are uniquely identified by combining the client container ID and receiver name to form a subscription ID. These must have stable values so that the subscription can be recovered.

1. Set the connection container ID to a stable value, such as **client-1**:

```
var container = rhea.create_container({id: "client-1"});
```

2. Create a receiver with a stable name, such as **sub-1**, and configure the receiver source for durability by setting the **durable** and **expiry_policy** properties:

```
var receiver_opts = {
    source: {
        address: "notifications",
        name: "sub-1",
        durable: 2,
        expiry_policy: "never"
    }
}

conn.open_receiver(receiver_opts);
```

To detach from a subscription, use the **receiver.detach()** method. To terminate the subscription, use the **receiver.close()** method.

For more information, see the durable-subscribe.js example.

8.3. CREATING SHARED SUBSCRIPTIONS

A shared subscription is a piece of state on the remote server representing one or more message receivers. Because it is shared, multiple clients can consume from the same stream of messages.

The client configures a shared subscription by setting the **shared** capability on the receiver source.

Shared subscriptions are uniquely identified by combining the client container ID and receiver name to form a subscription ID. These must have stable values so that multiple client processes can locate the same subscription. If the **global** capability is set in addition to **shared**, the receiver name alone is used to identify the subscription.

To create a durable subscription, follow these steps:

1. Set the connection container ID to a stable value, such as **client-1**:

```
var container = rhea.create_container({id: "client-1"});
```

2. Create a receiver with a stable name, such as **sub-1**, and configure the receiver source for sharing by setting the **shared** capability:

```
var receiver_opts = {
    source: {
        address: "notifications",
        name: "sub-1",
        capabilities: ["shared"]
    }
}

conn.open_receiver(receiver_opts);
```

To detach from a subscription, use the **receiver.detach()** method. To terminate the subscription, use the **receiver.close()** method.

For more information, see the shared-subscribe.js example.

CHAPTER 9. ERROR HANDLING

Errors in AMQ JavaScript can be handled by intercepting named events corresponding to AMQP protocol or connection errors.

9.1. HANDLING CONNECTION AND PROTOCOL ERRORS

You can handle protocol-level errors by intercepting the following events:

- **connection_error**
- **session_error**
- **sender_error**
- **receiver_error**
- **protocol_error**
- **error**

These events are fired whenever there is an error condition with the specific object that is in the event. After calling the error handler, the corresponding ***<object>*_close** handler is also called.

The **event** argument has an **error** attribute for accessing the error object.

Example: Handling errors

```
container.on("error", function (event) {
    console.log("An error!", event.error);
});
```

NOTE

Because the close handlers are called in the event of any error, only the error itself needs to be handled within the error handler. Resource cleanup can be managed by close handlers. If there is no error handling that is specific to a particular object, it is typical to handle the general **error** event and not have a more specific handler.

NOTE

When reconnect is enabled and the remote server closes a connection with the **amqp:connection:forced** condition, the client does not treat it as an error and thus does not fire the **connection_error** event. The client instead begins the reconnection process.

CHAPTER 10. LOGGING

10.1. CONFIGURING LOGGING

AMQ JavaScript uses the JavaScript debug module to implement logging.

For example, to enable detailed client logging, set the **DEBUG** environment variable to **rhea***:

Example: Enabling detailed logging

```
$ export DEBUG=rhea*
$ <your-client-program>
```

10.2. ENABLING PROTOCOL LOGGING

The client can log AMQP protocol frames to the console. This data is often critical when diagnosing problems.

To enable protocol logging, set the **DEBUG** environment variable to **rhea:frames**:

Example: Enabling protocol logging

```
$ export DEBUG=rhea:frames
$ <your-client-program>
```

CHAPTER 11. FILE-BASED CONFIGURATION

AMQ JavaScript can read the configuration options used to establish connections from a local file named **connect.json**. This enables you to configure connections in your application at the time of deployment.

The library attempts to read the file when the application calls the container **connect** method without supplying any connection options.

11.1. FILE LOCATIONS

If set, AMQ JavaScript uses the value of the **MESSAGING_CONNECT_FILE** environment variable to locate the configuration file.

If **MESSAGING_CONNECT_FILE** is not set, AMQ JavaScript searches for a file named **connect.json** at the following locations and in the order shown. It stops at the first match it encounters.

On Linux:

1. **$PWD/connect.json**, where **$PWD** is the current working directory of the client process
2. **$HOME/.config/messaging/connect.json**, where **$HOME** is the current user home directory
3. **/etc/messaging/connect.json**

On Windows:

1. **%cd%/connect.json**, where **%cd%** is the current working directory of the client process

If no **connect.json** file is found, the library uses default values for all options.

11.2. THE FILE FORMAT

The **connect.json** file contains JSON data, with additional support for JavaScript comments.

All of the configuration attributes are optional or have default values, so a simple example need only provide a few details:

Example: A simple connect.json file

```
{
   "host": "example.com",
   "user": "alice",
   "password": "secret"
}
```

SASL and SSL/TLS options are nested under **"sasl"** and **"tls"** namespaces:

Example: A connect.json file with SASL and SSL/TLS options

```
{
   "host": "example.com",
   "user": "ortega",
   "password": "secret",
```

```
    "sasl": {
        "mechanisms": ["SCRAM-SHA-1", "SCRAM-SHA-256"]
    },
    "tls": {
        "cert": "/home/ortega/cert.pem",
        "key": "/home/ortega/key.pem"
    }
}
```

11.3. CONFIGURATION OPTIONS

The option keys containing a dot (.) represent attributes nested inside a namespace.

Table 11.1. Configuration options in connect.json

Key	Value type	Default value	Description
scheme	string	**"amqps"**	**"amqp"** for cleartext or **"amqps"** for SSL/TLS
host	string	**"localhost"**	The hostname or IP address of the remote host
port	string or number	**"amqps"**	A port number or port literal
user	string	*None*	The user name for authentication
password	string	*None*	The password for authentication
sasl.mechanisms	list or string	*None (system defaults)*	A JSON list of enabled SASL mechanisms. A bare string represents one mechanism. If none are specified, the client uses the default mechanisms provided by the system.
sasl.allow_insecure	boolean	**false**	Enable mechanisms that send cleartext passwords
tls.cert	string	*None*	The filename or database ID of the client certificate
tls.key	string	*None*	The filename or database ID of the private key for the client certificate
tls.ca	string	*None*	The filename, directory, or database ID of the CA certificate
tls.verify	boolean	**true**	Require a valid server certificate with a matching hostname

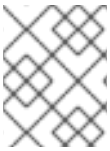
CHAPTER 12. INTEROPERABILITY

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

12.1. 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 JavaScript 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 12.1. 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

JavaScript has fewer native types than AMQP can encode. To send messages containing specific AMQP types, use the **wrap_** functions from the **rhea/types.js** module.

Table 12.2. AMQ JavaScript types before encoding and after decoding

AMQP type	AMQ JavaScript type before encoding	AMQ JavaScript type after decoding
null	**null**	**null**
boolean	**boolean**	**boolean**
char	**wrap_char(number)**	**number**
string	**string**	**string**
binary	**wrap_binary(string)**	**string**
byte	**wrap_byte(number)**	**number**
short	**wrap_short(number)**	**number**
int	**wrap_int(number)**	**number**
long	**wrap_long(number)**	**number**
ubyte	**wrap_ubyte(number)**	**number**
ushort	**wrap_ushort(number)**	**number**
uint	**wrap_uint(number)**	**number**

AMQP type	AMQ JavaScript type before encoding	AMQ JavaScript type after decoding
ulong	wrap_ulong(number)	number
float	wrap_float(number)	number
double	wrap_double(number)	number
array	wrap_array(Array, code)	Array
list	wrap_list(Array)	Array
map	wrap_map(object)	object
uuid	wrap_uuid(number)	number
symbol	wrap_symbol(string)	string
timestamp	wrap_timestamp(number)	number

Table 12.3. AMQ JavaScript and other AMQ client types (1 of 2)

AMQ JavaScript type before encoding	AMQ C++ type	AMQ .NET type
null	nullptr	null
boolean	bool	System.Boolean
wrap_char(number)	wchar_t	System.Char
string	std::string	System.String
wrap_binary(string)	proton::binary	System.Byte[]
wrap_byte(number)	int8_t	System.SByte
wrap_short(number)	int16_t	System.Int16
wrap_int(number)	int32_t	System.Int32
wrap_long(number)	int64_t	System.Int64
wrap_ubyte(number)	uint8_t	System.Byte
wrap_ushort(number)	uint16_t	System.UInt16

AMQ JavaScript type before encoding	AMQ C++ type	AMQ .NET type
wrap_uint(number)	**uint32_t**	**System.UInt32**
wrap_ulong(number)	**uint64_t**	**System.UInt64**
wrap_float(number)	**float**	**System.Single**
wrap_double(number)	**double**	**System.Double**
wrap_array(Array, code)	-	-
wrap_list(Array)	**std::vector**	**Amqp.List**
wrap_map(object)	**std::map**	**Amqp.Map**
wrap_uuid(number)	**proton::uuid**	**System.Guid**
wrap_symbol(string)	**proton::symbol**	**Amqp.Symbol**
wrap_timestamp(number)	**proton::timestamp**	**System.DateTime**

Table 12.4. AMQ JavaScript and other AMQ client types (2 of 2)

AMQ JavaScript type before encoding	AMQ Python type	AMQ Ruby type
null	**None**	**nil**
boolean	**bool**	**true, false**
wrap_char(number)	**unicode**	**String**
string	**unicode**	**String**
wrap_binary(string)	**bytes**	**String**
wrap_byte(number)	**int**	**Integer**
wrap_short(number)	**int**	**Integer**
wrap_int(number)	**long**	**Integer**
wrap_long(number)	**long**	**Integer**
wrap_ubyte(number)	**long**	**Integer**

AMQ JavaScript type before encoding	AMQ Python type	AMQ Ruby type
wrap_ushort(number)	**long**	**Integer**
wrap_uint(number)	**long**	**Integer**
wrap_ulong(number)	**long**	**Integer**
wrap_float(number)	**float**	**Float**
wrap_double(number)	**float**	**Float**
wrap_array(Array, code)	**proton.Array**	**Array**
wrap_list(Array)	**list**	**Array**
wrap_map(object)	**dict**	**Hash**
wrap_uuid(number)	-	-
wrap_symbol(string)	**str**	**Symbol**
wrap_timestamp(number)	**long**	**Time**

12.2. 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 JavaScript 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 12.5. AMQ JavaScript and JMS message types

AMQ JavaScript body type	JMS message type
string	**TextMessage**
null	**TextMessage**
wrap_binary(string)	**BytesMessage**

AMQ JavaScript body type	JMS message type
Any other type	**ObjectMessage**

12.3. 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 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.

12.4. 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 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 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 AMQ BROKER WITH THE EXAMPLES

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

B.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>***.

B.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
...
```

B.3. CREATING A QUEUE

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

```
$ <broker-instance-dir>/bin/artemis queue create --name examples --address examples --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.

B.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 2021-08-24 14:26:52 UTC