



Red Hat AMQ 7.7

AMQ JavaScript クライアントの使用

AMQ Clients 2.7 向け

Red Hat AMQ 7.7 AMQ JavaScript クライアントの使用

AMQ Clients 2.7 向け

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2023 | You need to change the HOLDER entity in the en-US/Using_the_AMQ_JavaScript_Client.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

本ガイドでは、クライアントをインストールして設定する方法、実例を実行し、他の AMQ コンポーネントでクライアントを使用する方法を説明します。

目次

第1章 概要	4
1.1. 主な特長	4
1.2. サポートされる標準およびプロトコル	4
1.3. サポートされる構成	4
1.4. 用語および概念	5
1.5. 本書の表記慣例	6
sudo コマンド	6
ファイルパス	6
変数テキスト	6
第2章 インストール	7
2.1. 前提条件	7
2.2. RED HAT ENTERPRISE LINUX へのインストールを参照してください。	7
2.3. MICROSOFT WINDOWS へのインストールを参照してください。	8
2.4. ブラウザーで使用するためのライブラリーの準備を参照してください。	8
第3章 スタートガイド	9
3.1. 前提条件	9
3.2. RED HAT ENTERPRISE LINUX での HELLO WORLD の実行	9
3.3. MICROSOFT WINDOWS での HELLO WORLD の実行	9
第4章 例	10
4.1. メッセージの送信	10
サンプルの実行	11
4.2. メッセージの受信	11
サンプルの実行	12
第5章 API の使用	13
5.1. メッセージングイベントの処理	13
5.2. イベント関連のオブジェクトへのアクセス	13
5.3. コンテナの作成	13
5.4. コンテナアイデンティティの設定	14
第6章 ネットワーク接続	15
6.1. 外向き接続の作成	15
6.2. 再接続の設定	15
6.3. フェイルオーバーの設定	16
6.4. 内向き接続の許可	16
第7章 セキュリティー	18
7.1. SSL/TLS を使用した接続のセキュリティー保護	18
7.2. ユーザーとパスワードを使用した接続	18
7.3. SASL 認証の設定	18
第8章 送信者と受信者	19
8.1. オンデマンドでのキューとトピックの作成	19
8.2. 永続サブスクリプションの作成	20
8.3. 共有サブスクリプションの作成	20
第9章 ファイルベースの設定	22
9.1. ファイルの場所	22
9.2. ファイル形式	22
9.3. 設定オプション	23

第10章 相互運用性	24
10.1. 他の AMQP クライアントとの相互運用	24
10.2. AMQ JMS での相互運用	28
JMS メッセージタイプ	28
10.3. AMQ BROKER への接続	29
10.4. AMQ INTERCONNECT への接続	29
付録A サブスクリプションの使用	30
A.1. アカウントへのアクセス	30
A.2. サブスクリプションのアクティベート	30
A.3. リリースファイルのダウンロード	30
A.4. パッケージ用のシステムの登録	30
付録B 例で AMQ ブローカーの使用	32
B.1. ブローカーのインストール	32
B.2. ブローカーの起動	32
B.3. キューの作成	32
B.4. ブローカーの停止	33

第1章 概要

AMQ JavaScript は、メッセージングアプリケーションを開発するためのライブラリーです。また、AMQP メッセージを送受信する JavaScript アプリケーションを作成できます。

AMQ JavaScript は AMQ Clients (複数の言語やプラットフォームをサポートするメッセージングライブラリースイット) に含まれています。クライアントの概要は、[AMQ Clients の概要](#) を参照してください。本リリースに関する詳細は、[AMQ Clients 2.7 リリースノート](#) を参照してください。

AMQ JavaScript は [Rhea](#) メッセージングライブラリーに基づいています。詳細な API ドキュメントは、[AMQ JavaScript API リファレンス](#) を参照してください。

1.1. 主な特長

- 既存のアプリケーションとの統合を簡素化するイベント駆動型の API
- セキュアな通信用の SSL/TLS
- 柔軟な SASL 認証
- 自動再接続およびフェイルオーバー
- AMQP と言語ネイティブのデータ型間のシームレスな変換
- AMQP 1.0 の全機能へのアクセス

1.2. サポートされる標準およびプロトコル

AMQ JavaScript は、以下の業界標準およびネットワークプロトコルをサポートします。

- [Advanced Message Queueing Protocol \(AMQP\)](#) のバージョン 1.0
- SSL の後継である TLS ([Transport Layer Security](#)) プロトコルのバージョン 1.0、1.1、1.2、および 1.3
- [Simple Authentication and Security Layer \(SASL\)](#) メカニズム ANONYMOUS、PLAIN、EXTERNAL
- IPv6 での最新の TCP

1.3. サポートされる構成

AMQ JavaScript は、以下に示す OS と言語のバージョンをサポートしています。詳細は、[Red Hat AMQ 7 Supported Configurations](#) を参照してください。

- Red Hat Enterprise Linux 6 (x86-64 のみ) および 7 (Software Collections の Node.js 6 および 8 を使用)
- Red Hat Enterprise Linux 8 with Node.js 10
- Node.js 10 を搭載した Microsoft Windows 10 Pro
- Node.js 10 を使用する Microsoft Windows Server 2012 R2 および 2016

AMQ JavaScript は、次の AMQ コンポーネントおよびバージョンとの組み合わせでサポートされています。

- AMQ ブローカーのすべてのバージョン
- AMQ Interconnect のすべてのバージョン
- AMQ Online のすべてのバージョン
- A-MQ 6 バージョン 6.2.1 以降

1.4. 用語および概念

本セクションでは、コア API エンティティを紹介し、コア API が連携する方法を説明します。

表1.1 API の用語

エンティティ	説明
Container	接続の最上位のコンテナ。
接続	ネットワーク上の2つのピア間の通信チャンネル。これにはセッションが含まれます。
Session	メッセージの送受信を行うためのコンテキスト。送信者および受信者が含まれます。
sender	メッセージをターゲットに送信するためのチャンネル。これにはターゲットがあります。
receiver	ソースからメッセージを受信するためのチャンネル。これにはソースがあります。
Source	メッセージの名前付きの発信元。
Target	メッセージの名前付き受信先。
メッセージ	情報のアプリケーション固有の部分。
Delivery	メッセージの転送。

AMQ JavaScript は **メッセージ** を送受信します。メッセージは、**senders** と **receivers** を介して、接続されたピアの間で転送されます。送信側および受信側は **セッション** 上で確立されます。セッションは **接続** 上で確立されます。接続は、一意に識別された2つの **コンテナ** 間で確立されます。コネクションには複数のセッションを含めることができますが、多くの場合、必要ありません。API を使用すると、セッションが必要でない限り、セッションを無視できます。

送信ピアは、メッセージ送信用の送信者を作成します。送信側には、リモートピアでキューまたはトピックを識別する **ターゲット** があります。受信ピアは、メッセージ受信用の受信者を作成します。受信側には、リモートピアでキューまたはトピックを識別する **ソース** があります。

メッセージの送信は **配信** と呼ばれます。メッセージとは、送信される内容のことで、ヘッダーやアンノテーションなどのすべてのメタデータが含まれます。配信は、そのコンテンツの移動に関連するプロトコルエクステンションです。

配信が完了したことを示すには、送信側または受信側セットのいずれかが解決します。送信側または受信側が解決されたことを知らせると、その配信の通信ができなくなります。受信側は、メッセージを受諾するか、拒否するかどうかを指定することもできます。

1.5. 本書の表記慣例

sudo コマンド

本書では、root 権限を必要とするすべてのコマンドに対して **sudo** が使用されています。すべての変更がシステム全体に影響する可能性があるため、**sudo** を使用する場合は注意が必要です。**sudo** の詳細は、[sudo コマンドの使用](#) を参照してください。

ファイルパス

本書では、すべてのファイルパスが Linux、UNIX、および同様のオペレーティングシステムで有効です (例: `/home/andrea`)。Microsoft Windows では、同等の Windows パスを使用する必要があります (例: `C:\Users\andrea`)。

変数テキスト

本書では、変数を含むコードブロックが紹介されていますが、これは、お客様の環境に固有の値に置き換える必要があります。可変テキストは矢印の中括弧で囲まれ、斜体の等幅フォントとしてスタイル設定されます。たとえば、以下のコマンドでは `<project-dir>` は実際の環境の値に置き換えます。

```
$ cd <project-dir>
```

第2章 インストール

本章では、環境に AMQ JavaScript をインストールする手順を説明します。

2.1. 前提条件

- AMQ リリースファイルおよびリポジトリにアクセスするには、[サブスクリプション](#) が必要です。
- AMQ JavaScript を使用するには、Node.js を環境にインストールする必要があります。詳細は、[Node.js](#) の Web サイトを参照してください。
- AMQ JavaScript は Node.js **debug** モジュールに依存します。インストール手順は [npm ページ](#) を参照してください。

2.2. RED HAT ENTERPRISE LINUX へのインストールを参照してください。

手順

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **INTEGRATION AND AUTOMATION** カテゴリで **Red Hat AMQ Clients** エントリーを見つけます。
3. **Red Hat AMQ Clients** をクリックします。 **Software Downloads** ページが開きます。
4. **AMQ Clients 2.7.0 JavaScript.zip** ファイルをダウンロードします。
5. **unzip** コマンドを使用して、ファイルの内容を、選択したディレクトリーに展開します。

```
$ unzip amq-clients-2.7.0-javascript.zip
```

.zip ファイルの内容を抽出すると、**amq-clients-2.7.0-javascript** という名前のディレクトリーが作成されます。これはインストールの最上位ディレクトリーであり、本書では **<install-dir>** と呼びます。

インストールされたライブラリーを使用するように環境を設定するには、**node_modules** ディレクトリーを **NODE_PATH** 環境変数に追加します。

```
$ cd amq-clients-2.7.0-javascript
$ export NODE_PATH=$PWD/node_modules:$NODE_PATH
```

新しいコンソールセッションすべてでこの設定を有効にするには **\$HOME/.bashrc** ファイルに **NODE_PATH** を設定します。

インストールをテストするには、次のコマンドを使用します。インストールされたライブラリーを正常にインポートすると、コンソールに **OK** と出力されます。

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

2.3. MICROSOFT WINDOWS へのインストール を参照してください。

手順

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **INTEGRATION AND AUTOMATION** カテゴリで **Red Hat AMQ Clients** エントリーを見つけます。
3. **Red Hat AMQ Clients** をクリックします。 **Software Downloads** ページが開きます。
4. **AMQ Clients 2.7.0 JavaScript.zip** ファイルをダウンロードします。
5. zip ファイルを右クリックし、**Extract All** を選択して、選択したディレクトリーにファイルの内容を展開します。

.zip ファイルの内容を抽出すると、**amq-clients-2.7.0-javascript** という名前のディレクトリーが作成されます。これはインストールの最上位ディレクトリーであり、本書では **<install-dir>** と呼びます。

インストールされたライブラリーを使用するように環境を設定するには、**node_modules** ディレクトリーを **NODE_PATH** 環境変数に追加します。

```
$ cd amq-clients-2.7.0-javascript
$ set NODE_PATH=%cd%\node_modules;%NODE_PATH%
```

2.4. ブラウザーで使用するためのライブラリーの準備 を参照してください。

AMQ JavaScript は Web ブラウザー内で実行できます。ブラウザーと互換性のあるライブラリーのバージョンを作成するには、**npm run browserify** コマンドを使用します。

```
$ cd amq-clients-2.7.0-javascript/node_modules/rhea
$ npm install
$ npm run browserify
```

これにより、ブラウザーベースのアプリケーションで使用できる **rhea.js** という名前のファイルが作成されます。

第3章 スタートガイド

本章では、環境を設定して簡単なメッセージングプログラムを実行する手順を説明します。

3.1. 前提条件

- ご使用の環境の[インストール](#)手順を完了する必要があります。
- インターフェイス **localhost** およびポート **5672** で接続をリッスンする AMQP 1.0 メッセージブローカーが必要です。匿名アクセスを有効にする必要があります。詳細は、[ブローカーの開始](#)を参照してください。
- **examples** という名前のキューが必要です。詳細は、[キューの作成](#)を参照してください。

3.2. RED HAT ENTERPRISE LINUX での HELLO WORLD の実行

Hello World の例では、ブローカーへの接続を作成し、グリーティングを含むメッセージを **examples** キューに送信して、受信しなおします。成功すると、受信したメッセージをコンソールに出力します。

`examples` ディレクトリーに移動し、**helloworld.js** の例を実行します。

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

3.3. MICROSOFT WINDOWS での HELLO WORLD の実行

Hello World の例では、ブローカーへの接続を作成し、グリーティングを含むメッセージを **examples** キューに送信して、受信しなおします。成功すると、受信したメッセージをコンソールに出力します。

`examples` ディレクトリーに移動し、**helloworld.js** の例を実行します。

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

第4章 例

本章では、サンプルプログラムで AMQ JavaScript を使用方法について説明します。

その他の例については、[AMQ JavaScript サンプルスイート](#) を参照してください。

4.1. メッセージの送信

このクライアントプログラムは `<connection-url>` を使用してサーバーに接続し、ターゲット `<address>` の送信者を作成し、`<message-body>` を含むメッセージを送信して接続を切断して終了します。

例: メッセージの送信

```
"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);
```

サンプルの実行

サンプルプログラムを実行するには、サンプルプログラムをローカルファイルにコピーし、**node** コマンドを使用して呼び出します。

```
$ node send.js amqp://localhost queue1 hello
```

4.2. メッセージの受信

このクライアントプログラムは **<connection-url>** を使用してサーバーに接続し、ソース **<address>** の受信側を作成し、終了するか、**<count>** メッセージに到達するまでメッセージを受信します。

例: メッセージの受信

```
"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);
```

サンプルの実行

サンプルプログラムを実行するには、サンプルプログラムをローカルファイルにコピーし、**python** コマンドを使用して呼び出します。

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

第5章 API の使用

詳細は、[AMQ JavaScript API reference](#) および [AMQ JavaScript example suite](#) を参照してください。

5.1. メッセージングイベントの処理

AMQ JavaScript は非同期イベント駆動型 API です。アプリケーションがイベントを処理する方法を定義するには、ユーザーはイベント処理関数をコンテナオブジェクトに登録します。これらの関数は、ネットワークアクティビティとして呼び出され、タイマーが新規イベントをトリガーします。

例: メッセージングイベントの処理

```
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");
});
```

これらはごく一部の一般的なケースイベントのみです。完全なセットは {ClientAmqpJavaScriptApiLink} に記載されています。

5.2. イベント関連のオブジェクトへのアクセス

イベント引数には、イベントが関係するオブジェクトにアクセスするための属性が含まれます。たとえば、**connection_open** イベントはイベント **接続** 属性を設定します。

イベントのプライマリーオブジェクトに加えて、イベントのコンテキストを形成する全オブジェクトも設定されます。特定のイベントに対する関連性のない属性は null です。

例: イベント関連のオブジェクトへのアクセス

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

5.3. コンテナの作成

コンテナは最上位の API オブジェクトです。これは、接続を作成するエントリーポイントであり、メインのイベントループを実行します。多くの場合、これはグローバルイベントハンドラーで構築されます。

例: コンテナの作成

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

5.4. コンテナアイデンティティの設定

各コンテナインスタンスには、コンテナ ID と呼ばれる一意のアイデンティティがあります。AMQ JavaScript がネットワーク接続を作成する場合、コンテナ ID をリモートピアに送信します。コンテナ ID を設定するには、**id** オプションを **create_container** メソッドに渡します。

例: コンテナアイデンティティの設定

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

ユーザーが ID を設定しない場合には、コンテナが処理されると、ライブラリーは UUID を生成します。

第6章 ネットワーク接続

6.1. 外向き接続の作成

リモートサーバーに接続するには、ホストとポートを含む接続オプションを `container.connect()` メソッドに渡します。

例: 外向き接続の作成

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

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

container.connect(opts);
```

デフォルトのホストは `localhost` です。デフォルトのポートは 5672 です。

セキュアな接続の作成については、[7章 セキュリティー](#) セクションを参照してください。

6.2. 再接続の設定

再接続することで、クライアントは失われた接続から復旧できます。これは、一時的なネットワークまたはコンポーネントの障害後に、分散システムのコンポーネントが再確立されるように使用されます。

AMQ JavaScript はデフォルトで再接続を有効にします。接続試行に失敗すると、クライアントは少しの遅延の後に再度試行します。遅延は、デフォルトの最大値 60 秒まで、新しい試行ごとに指数関数的に増加します。

再接続を無効にするには、`reconnect` 接続オプションを `false` に設定します。

例: 再接続の無効化

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

container.connect(opts);
```

次の接続試行までの間の遅延を制御するには、`initial_reconnect_delay` および `max_reconnect_delay` 接続オプションを設定します。遅延オプションはミリ秒単位で指定します。

再接続試行回数を制限するには、`reconnect_limit` オプションを設定します。

例: 再接続の設定例

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

```

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

  container.connect(opts);

```

6.3. フェイルオーバーの設定

AMQ JavaScript を使用すると、代わりの接続エンドポイントをプログラムで設定できます。

複数の接続エンドポイントを指定するには、新しい接続オプションを返す関数を定義し、**connection_details** オプションで関数を渡します。この関数は、接続試行ごとに1回呼び出されます。

例: フェイルオーバーの設定

```

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);

```

この例では、ホストの一覧に対してラウンドロビンフェイルオーバーを繰り返すように実装します。このインターフェイスを使用して、独自のフェイルオーバー動作を実装できます。

6.4. 内向き接続の許可

AMQ JavaScript はインバウンドネットワーク接続を受け入れ、カスタムメッセージングサーバーを構築できます。

接続のリッスンを開始するには、**container.listen()** メソッドを使用して、ローカルホストアドレスとリッスンするポートが含まれるオプションを指定します。

例: 内向き接続の許可

```

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);
```

特別な IP アドレス **0.0.0.0** は、利用可能なすべての IPv4 インターフェイスでリッスンします。すべての IPv6 インターフェイスをリッスンするには **:::0** を使用します。

詳細は、[サーバー receive.js の例](#) を参照してください。

第7章 セキュリティー

7.1. SSL/TLS を使用した接続のセキュリティー保護

AMQ JavaScript は SSL/TLS を使用して、クライアントとサーバー間の通信を暗号化します。

SSL/TLS を使用してリモートサーバーに接続するには、**transport** 接続オプションを **tls** に設定します。

例: SSL/TLS の有効化

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

container.connect(opts);
```



注記

デフォルトでは、クライアントは信頼できない証明書が割り当てられたサーバーへの接続を拒否します。これは、テスト環境などが該当します。証明書の認証を省略するには、**rejectUnauthorized** 接続オプションを **false** に設定します。これにより、接続のセキュリティーが危険にさらされることに注意してください。

7.2. ユーザーとパスワードを使用した接続

AMQ JavaScript は、ユーザーとパスワードによる接続を認証できます。

認証に使用する認証情報を指定するには、**username** と **password** の接続オプションを設定します。

例: ユーザーとパスワードを使用した接続

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

container.connect(opts);
```

7.3. SASL 認証の設定

AMQ JavaScript は SASL プロトコルを使用して認証を実行します。SASL はさまざまな認証メカニズムを使用できます。2つのネットワークピアが接続すると、許可されたメカニズムが交換され、両方で許可されている最も強力なメカニズムが選択されます。

AMQ JavaScript は、ユーザーとパスワード情報があるかどうかによって SASL メカニズムを有効にします。ユーザーとパスワードの両方が指定されている場合は、**PLAIN** が使用されます。ユーザーのみを指定すると、**ANONYMOUS** が使用されます。いずれも指定されていない場合、SASL は無効になります。

第8章 送信者と受信者

クライアントは、送信者と受信者のリンクを使用して、メッセージ配信のチャンネルを表現します。送信者と受信者は一方向であり、送信元はメッセージの発信元に、ターゲットはメッセージの宛先になります。

ソースとターゲットは、多くの場合、メッセージブローカーのキューまたはトピックを参照します。ソースは、サブスクリプションを表すためにも使用されます。

8.1. オンデマンドでのキューとトピックの作成

メッセージサーバーによっては、キューとトピックのオンデマンド作成をサポートします。送信側または受信側が割り当てられている場合、サーバーは送信側ターゲットアドレスまたは受信側ソースアドレスを使用して、アドレスに一致する名前でもキューまたはトピックを作成します。

メッセージサーバーは通常、キュー (1対1のメッセージ配信用) またはトピック (1対多のメッセージ配信用) を作成します。クライアントは、ソースまたはターゲットに **queue** または **topic** 機能を設定してどちらを優先するかを示すことができます。

キューまたはトピックセマンティクスを選択するには、以下の手順に従います。

1. キューとトピックを自動的に作成するようにメッセージサーバーを設定します。多くの場合、これがデフォルト設定になります。
2. 以下の例のように、送信者ターゲットまたは受信者ソースに **queue** または **topic** 機能を設定します。

例: オンデマンドで作成されたキューへの送信

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

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

conn.open_sender(sender_opts);
```

例: オンデマンドで作成されたトピックからの受信

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

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

conn.open_receiver(receiver_opts);
```

詳細は、以下の例を参照してください。

- [queue-send.js](#)
- [queue-receive.js](#)
- [topic-send.js](#)
- [topic-receive.js](#)

8.2. 永続サブスクリプションの作成

永続サブスクリプションは、メッセージの受信側を表すリモートサーバーの状態です。通常、メッセージ受信者は、クライアントが終了すると、破棄されます。ただし、永続サブスクリプションは永続的であるため、クライアントはそれらのサブスクリプションの割り当てを解除してから、後で再度アタッチできます。デタッチ時に受信したすべてのメッセージは、クライアントの再割り当て時に利用できません。

永続サブスクリプションは、クライアントコンテナ ID とレシーバー名を組み合わせることでサブスクリプション ID を形成することで一意に識別されます。これらには、サブスクリプションを回復できるように、安定した値が必要です。

1. 接続コンテナ ID を **client-1** などの安定した値に設定します。

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

2. **sub-1** などの安定した名前で作成し、**durable** および **expiry_policy** プロパティを指定して、受信者のソースが永続化されるように設定します。

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

conn.open_receiver(receiver_opts);
```

サブスクリプションからデタッチするには、**receiver.detach()** メソッドを使用します。サブスクリプションを終了するには、**receiver.close()** メソッドを使用します。

詳細は、[durable-subscribe.js の例](#) を参照してください。

8.3. 共有サブスクリプションの作成

共有サブスクリプションとは、1つ以上のメッセージレシーバーを表すリモートサーバーの状態のことです。このサブスクリプションは共有されているため、複数のクライアントが同じメッセージのストリームから消費できます。

クライアントは、受信者のソースに**shared** 機能を設定して、共有サブスクリプションを設定します。

共有サブスクリプションは、クライアントコンテナ ID とレシーバー名を組み合わせることでサブスクリプション ID を形成することで一意に識別されます。複数のクライアントプロセスで同じサブスクリプションを特定できるように、これらに安定した値を指定する必要があります。**shared** に加えて **global** 機能が設定されている場合、サブスクリプション識別に受信者名だけが使用されます。

永続サブスクリプションを作成するには、以下の手順に従います。

1. 接続コンテナ ID を **client-1** などの安定した値に設定します。

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

2. **sub-1** などの安定した名前で作信者を作成し、**shared** 機能を設定して共有用に受信者のソースを設定します。

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

conn.open_receiver(receiver_opts);
```

サブスクリプションからデタッチするには、**receiver.detach()** メソッドを使用します。サブスクリプションを終了するには、**receiver.close()** メソッドを使用します。

詳細は、[shared-subscribe.js の例](#) を参照してください。

第9章 ファイルベースの設定

AMQ JavaScript は、**connect.json** という名前のローカルファイルからの接続確立に使用される設定オプションを読み取ることができます。これにより、デプロイメント時にアプリケーションで接続を設定できます。

ライブラリーは、接続オプションを指定せずにアプリケーションがコンテナの **connect** メソッドを呼び出すと、ファイルの読み取りを試みます。

9.1. ファイルの場所

設定されている場合には、AMQ JavaScript は **MESSAGING_CONNECT_FILE** 環境変数の値を使用して設定ファイルを検索します。

MESSAGING_CONNECT_FILE が設定されていない場合には、AMQ JavaScript は以下の場所で **connect.json** という名前のファイルを検索します。最初の一致で停止します。

Linux の場合:

1. **\$PWD/connect.json**: **\$PWD** はクライアントプロセスの現在の作業ディレクトリーです。
2. **\$HOME/.config/messaging/connect.json**: **\$HOME** は現在のユーザーのホームディレクトリーに置き換えます。
3. **/etc/messaging/connect.json**

Windows の場合:

1. **%cd%/connect.json**: **%cd%** はクライアントプロセスの現在の作業ディレクトリーです。

connect.json ファイルが見つからない場合、ライブラリーはすべてのオプションにデフォルト値を使用します。

9.2. ファイル形式

connect.json ファイルには JSON データが含まれ、JavaScript コメントの追加サポートが提供されます。

設定属性はすべてオプションであるか、デフォルト値があるため、簡単な例では詳細をいくつか指定するだけで済みます。

例: 簡単な **connect.json** ファイル

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

SASL および SSL/TLS オプションは、**"sasl"** および **"tls"** namespace で入れ子になっています。

例: SASL および SSL/TLS オプションを含む **connect.json** ファイル

```
{
```

```

"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"
}
}

```

9.3. 設定オプション

ドット (.) を含むオプションキーは、namespace にネストされた属性を表します。

表9.1 connect.jsonの設定オプション

キー	値のタイプ	デフォルト値	説明
scheme	string	"amqps"	SSL/TLS のクリアテキストまたは "amqps" の場合は "amqp"
host	string	"localhost"	リモートホストのホスト名または IP アドレス
ポート	文字列または番号	"amqps"	ポート番号またはポートリテラル
user	string	None	認証のユーザー名
password	string	None	認証のパスワード
sasl.mechanism s	リストまたは文字列	none(システムのデフォルト)	有効な SASL メカニズムの JSON リスト。ベア文字列は1つのメカニズムを表します。指定がない場合、クライアントはシステムによって提供されるデフォルトのメカニズムを使用します。
sasl.allow_insecure	boolean	false	クリアテキストパスワードを送信するメカニズムの有効化
tls.cert	string	None	クライアント証明書のファイル名またはデータベース ID
tls.key	string	None	クライアント証明書の秘密鍵のファイル名またはデータベース ID
tls.ca	string	None	CA 証明書のファイル名、ディレクトリー、またはデータベース ID
tls.verify	boolean	true	ホスト名が一致する、有効なサーバー証明書が必要

第10章 相互運用性

本章では、AMQ JavaScript を他の AMQ コンポーネントと組み合わせて使用方法を説明します。AMQ コンポーネントの互換性の概要は、[製品の概要](#) を参照してください。

10.1. 他の AMQP クライアントとの相互運用

AMQP メッセージは [AMQP タイプシステム](#) を使用して設定されます。このような一般的な形式は、異なる言語の AMQP クライアントが相互に対話できる理由の1つです。

メッセージを送信する場合、AMQ JavaScript は自動的に言語ネイティブの型を AMQP でエンコードされたデータに変換します。メッセージの受信時に、リバース変換が行われます。



注記

AMQP タイプの詳細は、Apache Qpid プロジェクトによって維持される [インタラクティブタイプリファレンス](#) を参照してください。

表10.1 AMQP 型

AMQP 型	説明
null	空の値
boolean	true または false の値
char	単一の Unicode 文字
string	Unicode 文字のシーケンス
binary	バイトのシーケンス
byte	署名済み 8 ビット整数
short	署名済み 16 ビット整数
int	署名済み 32 ビット整数
long	署名済み 64 ビット整数
ubyte	署名なしの 8 ビット整数
ushort	署名なしの 16 ビット整数
uint	署名なしの 32 ビット整数
ulong	署名なしの 64 ビット整数
float	32 ビット浮動小数点数

AMQP 型	説明
double	64 ビット浮動小数点数
array	単一型の値シーケンス
list	変数型の値シーケンス
map	異なるキーから値へのマッピング
uuid	ユニバーサル一意識別子
symbol	制限されたドメインからの7ビットの ASCII 文字列
timestamp	絶対的な時点

JavaScript にあるネイティブ型は、AMQP がエンコードできる数よりも少なくなっています。特定の AMQP タイプを含むメッセージを送信するには、**rhea/types.js** モジュールの **wrap_** 関数を使用します。

表10.2 エンコード前およびデコード後における AMQ JavaScript タイプ

AMQP 型	エンコード前の AMQ JavaScript タイプ	デコード後の AMQ JavaScript タイプ
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 型	エンコード前の AMQ JavaScript タイプ	デコード後の AMQ JavaScript タイプ
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

表10.3 AMQ JavaScript およびその他の AMQ クライアントタイプ (1/2)

エンコード前の AMQ JavaScript タイプ	AMQ C++ タイプ	AMQ .NET タイプ
null	nullptr	null
boolean	ブール型	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

エンコード前の AMQ JavaScript タイプ	AMQ C++ タイプ	AMQ .NET タイプ
<code>wrap_uint(number)</code>	<code>uint32_t</code>	<code>System.UInt32</code>
<code>wrap_ulong(number)</code>	<code>uint64_t</code>	<code>System.UInt64</code>
<code>wrap_float(number)</code>	<code>float</code>	<code>System.Single</code>
<code>wrap_double(number)</code>	<code>double</code>	<code>system.Double</code>
<code>wrap_array(Array, code)</code>	-	-
<code>wrap_list(Array)</code>	<code>std::vector</code>	<code>Amqp.List</code>
<code>wrap_map(object)</code>	<code>std::map</code>	<code>Amqp.Map</code>
<code>wrap_uuid(number)</code>	<code>proton::uuid</code>	<code>System.Guid</code>
<code>wrap_symbol(string)</code>	<code>proton::symbol</code>	<code>Amqp.Symbol</code>
<code>wrap_timestamp(number)</code>	<code>proton::timestamp</code>	<code>System.DateTime</code>

表10.4 AMQ JavaScript およびその他の AMQ クライアントタイプ (2/2)

エンコード前の AMQ JavaScript タイプ	AMQ Python タイプ	AMQ Ruby タイプ
<code>null</code>	<code>None</code>	<code>nil</code>
<code>boolean</code>	ブール型	<code>true, false</code>
<code>wrap_char(number)</code>	<code>unicode</code>	<code>String</code>
<code>string</code>	<code>unicode</code>	<code>String</code>
<code>wrap_binary(string)</code>	<code>bytes</code>	<code>String</code>
<code>wrap_byte(number)</code>	<code>int</code>	<code>Integer</code>
<code>wrap_short(number)</code>	<code>int</code>	<code>Integer</code>
<code>wrap_int(number)</code>	<code>long</code>	<code>Integer</code>
<code>wrap_long(number)</code>	<code>long</code>	<code>Integer</code>
<code>wrap_ubyte(number)</code>	<code>long</code>	<code>Integer</code>

エンコード前の AMQ JavaScript タイプ	AMQ Python タイプ	AMQ Ruby タイプ
<code>wrap_ushort(number)</code>	<code>long</code>	<code>Integer</code>
<code>wrap_uint(number)</code>	<code>long</code>	<code>Integer</code>
<code>wrap_ulong(number)</code>	<code>long</code>	<code>Integer</code>
<code>wrap_float(number)</code>	<code>float</code>	<code>Float</code>
<code>wrap_double(number)</code>	<code>float</code>	<code>Float</code>
<code>wrap_array(Array, code)</code>	<code>proton.Array</code>	<code>Array</code>
<code>wrap_list(Array)</code>	<code>list</code>	<code>Array</code>
<code>wrap_map(object)</code>	<code>dict</code>	<code>Hash</code>
<code>wrap_uuid(number)</code>	-	-
<code>wrap_symbol(string)</code>	<code>str</code>	<code>Symbol</code>
<code>wrap_timestamp(number)</code>	<code>long</code>	<code>Time</code>

10.2. AMQ JMS での相互運用

AMQP は JMS メッセージングモデルへの標準マッピングを定義します。本セクションでは、そのマッピングのさまざまな側面について説明します。詳細は、AMQ JMS [Interoperability](#) の章を参照してください。

JMS メッセージタイプ

AMQ JavaScript は、本文タイプが異なる、単一のメッセージを提供します。一方、JMS API は異なるメッセージタイプを使用してさまざまな種類のデータを表します。次の表は、特定の本文タイプが JMS メッセージタイプにどのようにマップされるかを示しています。

作成される JMS メッセージタイプをさらに明示的に制御するには、`x-opt-jms-msg-type` メッセージアノテーションを設定できます。詳細は、AMQ JMS [Interoperability](#) の章を参照してください。

表10.5 AMQ JavaScript および JMS メッセージタイプ

AMQ JavaScript ボディータイプ	JMS メッセージタイプ
<code>string</code>	<code>TextMessage</code>
<code>null</code>	<code>TextMessage</code>
<code>wrap_binary(string)</code>	<code>BytesMessage</code>

AMQ JavaScript ボディタイプ	JMS メッセージタイプ
それ以外のタイプ	ObjectMessage

10.3. AMQ BROKER への接続

AMQ Broker は AMQP 1.0 クライアントと相互運用するために設計されています。以下を確認して、ブローカーが AMQP メッセージング用に設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- AMQ Broker AMQP アクセプターが有効になっている。[デフォルトのアクセプター設定](#) を参照してください。
- 必要なアドレスがブローカーに設定されている。[アドレス](#)、[キュー](#)、[およびトピック](#) を参照してください。
- ブローカーはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。[Broker Security](#) を参照してください。

10.4. AMQ INTERCONNECT への接続

AMQ Interconnect は AMQP 1.0 クライアントであれば機能します。以下をチェックして、コンポーネントが正しく設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- ルーターはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。[ネットワーク接続のセキュリティ保護](#) を参照してください。

付録A サブスクリプションの使用

AMQ は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

A.1. アカウントへのアクセス

手順

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

A.2. サブスクリプションのアクティベート

手順

1. access.redhat.com に移動します。
2. サブスクリプション に移動します。
3. **Activate a subscription** に移動し、16桁のアクティベーション番号を入力します。

A.3. リリースファイルのダウンロード

.zip、.tar.gz およびその他のリリースファイルにアクセスするには、カスタマーポータルを使用してダウンロードする関連ファイルを検索します。RPM パッケージまたは Red Hat Maven リポジトリを使用している場合は、この手順は必要ありません。

手順

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **JBOSS INTEGRATION AND AUTOMATION** カテゴリーの Red Hat AMQ エントリーを見つけます。
3. 必要な AMQ 製品を選択します。 **Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

A.4. パッケージ用のシステムの登録

RPM パッケージを Red Hat Enterprise Linux にインストールするには、システムが登録されている必要があります。ダウンロードしたリリースファイルを使用している場合は、この手順は必要ありません。

手順

1. access.redhat.com に移動します。
2. **Registration Assistant** に移動します。

3. ご使用の OS バージョンを選択し、次のページに進みます。
4. システムの端末に一覧表示されたコマンドを使用して、登録を完了します。

詳細は、[How to Register and Subscribe a System to the Red Hat Customer Portal](#) を参照してください。

B.4. ブローカーの停止

サンプルの実行が終了したら、**artemis stop** コマンドを使用してブローカーを停止します。

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

改訂日時 : 2023-01-28 12:23:13 +1000