Red Hat AMQ 2021.Q3

AMQ JavaScript クライアントの使用

AMQ Clients 2.10 向け

Last Updated: 2023-02-23

Red Hat AMQ 2021.Q3 AMQ JavaScript クライアントの使用

AMQ Clients 2.10 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

概要

本ガイドでは、クライアントのインストールや設定、実例の実行、他の AMQ コンポーネントでのクライアントの使用方法について説明します。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。これは大規模な取り組みであるため、これらの変更は今後の複数のリリースで段階的に実施されます。詳細は、Red Hat CTO である Chris Wright のメッセージをご覧ください。

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。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、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ートである AMQ Clients の一部です。クライアントの概要は、「AMQ Clients Overview」を参照してください。本リリースに関する詳細は、『AMQ Clients 2.10 Release Notes』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 reference 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 と言語ネイティブデータ型間のシームレスな変換
- AMQP 1.0 のすべての機能と機能へのアクセス

1.2. サポート対象の標準およびプロトコル

AMQ JavaScrip 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) 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、および EXTERNAL
- IPv6 での最新の TCP

1.3. サポートされる構成

AMQ JavaScript でサポートされている設定については、Red Hat カスタマーポータルの「Red Hat AMQ 7 でサポートされる構成」を参照してください。

1.4. 用語および概念

本セクションでは、コア API エンティティーを紹介し、それらが一緒に操作する方法を説明します。

表1.1 API の用語

エンティティー	説明
コンテナー	接続の最上位のコンテナー。
接続	ネットワーク上の 2 つのピア間の通信用のチャネル。これにはセッションが含まれます。
セッション	メッセージの送受信を行うためのコンテキスト。送信者およびレシーバーが含まれます。
送信	メッセージをターゲットに送信するためのチャネル。これにはターゲットがあります。
受信	ソースからメッセージを受信するためのチャネル。ソースがあります。
Source	メッセージに対する名前付きポイント。
ターゲット	メッセージの名前付き宛先。
メッセージ	アプリケーション固有の情報部分。
配信	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、**送信側**と**受信側**を介して接続されたピア間で転送されます。送信側およびレシーバーは **セッション** 上で確立されます。セッションは**コネクション**を介して確立されます。接続は、一意に識別された 2 つの**コンテナー**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、これは必要ありません。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、メッセージを送信するために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、メッセージを受信するための受信側を作成します。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。メッセージは送信される内容で、ヘッダーやア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チェンジです。

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です。これが設定されていることを知らせると、その配信に関する通信はなくなります。受信側は、メッセージを受諾または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 権限を必要とするコマンドには **sudo** が使用されています。何らか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

本書には、実際の環境に固有の値に置き換える必要がある変数を含むコードブロックが含まれています。変数テキストは中括弧で囲まれ、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、以下の例では、**<project-dir>** を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ー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. 「INSTALLING ON RED HAT ENTERPRISE LINUX」 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** カテゴリーで **Red Hat AMQ Client** 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 2.10.0 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-2.10.0-javascript.zip
```

.zip ファイルの内容を展開すると、**amq-clients-2.10.0-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、本書全体で **<install-dir>** と呼ばれます。

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-2.10.0-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した場合は、**OK** をコンソールに出力します。

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

2.3. 「INSTALLING ON MICROSOFT WINDOWS」 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** カテゴリーで **Red Hat AMQ Client** 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 2.10.0 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 ファイルの内容を展開すると、**amq-clients-2.10.0-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、本書全体で **<install-dir>** と呼ばれます。

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

2.4. 「PREPARING THE LIBRARY FOR USE IN BROWSERS」 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-2.10.0-javascript/node_modules/rhea
$ npm install
$ npm run browserify
```

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

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

- お使いの環境の インストール 手順を完了する必要があります。
- インターフェース **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、グリーティングが含まれるメッセージを **examples** キューに送信し、それを受け取ります。成功すると、受け取っ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、グリーティングが含まれるメッセージを **examples** キューに送信し、それを受け取ります。成功すると、受け取っ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 を使用する方法について説明します。

その他の例は、AMQ JavaScript サンプルスイート および Rhea の例 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ます。ターゲット **<address>** の送信側は **<message-body>** が含まれるメッセージを送信し、接続を閉じて終了します。

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、これをローカルファイルにコピーし、**node** コマンドを使用してこれを呼び出します。詳細は、3章*スタートガイド*を参照してください。

```
$ 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>** のレシーバーを作成し、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、これをローカルファイルにコピーし、**python** コマンドを使用してこれを呼び出します。詳細は、*3章スタートガイド* を参照してください。

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

第5章 API の使用

詳細は、AMQ JavaScript API リファレンス および AMQ JavaScript サンプルスイート 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、ユーザーが **container** オブジェクトのイベント処理機能を登録します。これらの機能は、ネットワークアクティビティーとして呼び出され、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");
});
```

これらはいくつかの一般的なケースイベントのみです。完全セットは AMQ JavaScript API リファレンス に文書化されています。

5.2. イベント関連のオブジェクトへのアクセス

event 引数には、イベントが関連するオブジェクトにアクセスするための属性があります。たとえば、**connection_open** イベントはイベント **connection** 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すべての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. コンテナーの作成

コンテナーはトップレベルの 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: コンテナーの作成

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

5.4. コンテナーアイデンティティーの設定

各コンテナーインスタンスには、コンテナー ID と呼ばれる一意のアイデンティティーがあります。AMQ JavaScript がネットワーク接続を作成したら、コンテナー ID をリモートピアに送信します。コンテナー 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: コンテナーアイデンティティーの設定

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

ユーザーが ID を設定しないと、コンテナーが構成されると、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: 送信接続の作成

```
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関する詳細は、「*7章 セキュリティー*」を参照してください。

6.2. 再接続の設定

再接続すると、クライアントが失われた接続から回復できます。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た状態にするために使用されます。

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、クライアントは若干時間が経ってから再度試行します。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);
```

接続試行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

```
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、代替の接続エンドポイントをプログラム的に設定できます。

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ことを実装します。このインターフェースを使用すると、独自のフェイルオーバー動作を実装できます。

6.4. 受信接続の許可

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、ローカルホストアドレスおよびリッスンするポートが含まれるオプションで **container.listen()** メソッドを使用します。

例: 受信接続の許可

```
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 インターフェースでリッスンします。すべての IPv6 インターフェー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を持つサーバーへの接続を拒否します。これは、テスト環境で発生する場合があります。証明書の承認をバイパスするには、**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損なわ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。

認証に使用するクレデンシャルを指定するには、**username** および **password** 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 は多くの異なる 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に接続すると、許可されるメカニズムを交換し、両方で許可され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の有無に基づいて 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。ユーザーのみが指定されている場合には、**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 送信者およびレシーバー

クライアントは送信側と受信側のリンクを使用して、メッセージの配信にチャネルを表します。送信者と受信側は一方向で、メッセージの送信元がソースエンドでメッセージの送信先がターゲットエンドになります。

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

一部のメッセージサーバーは、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。送信側またはレシーバーが割り当てられている場合、サーバーは送信側の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を持つ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の場合) 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** 機能を設定することで、希望のものを指定できます。

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. キューとトピックの自動作成のために、メッセージサーバーを設定します。多くの場合、これはデフォルト設定になります。

2. 以下の例のように、送信側のターゲットまたは受信側ソースに **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: 要求時に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、クライアントが閉じられると、メッセージ受信側は破棄されます。ただし、永続サブスクリプションは永続的であるため、クライアントはこれらのサブスクリプションの割り当てを解除してから、後で再度アタッチすることができます。デタッチ中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す。

永続サブスクリプションは、クライアントコンテナー ID とレシーバー名を組み合わせて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。サブスクリプションが回復できるようにするには、これらの値に安定した値が必要です。

1. 接続コンテナー 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 receiver を作成し、 **durable** および **expiry_policy** プロパティーを設定して、持続性を確保するためにレシーバーソースを設定します。

```
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。共有されているので、複数のクライアントは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、レシーバーソースに **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ー ID とレシーバー名を組み合わせて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見つけることができるように、これらの値が安定している必要があります。**shared** に加えて **global** 機能が設定されている場合、レシーバー名のみを使用してサブスクリプションを特定します。

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. 接続コンテナー 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受信側を作成し、**shared** 機能を設定して共有用の受信側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 エラー処理

AMQ JavaScript のエラーは、AMQP プロトコルまたは接続エラーに対応する名前付きイベントをインターセプトすることで処理できます。

9.1. 接続およびプロトコルエラーの処理

以下のイベントをインターセプトして、プロトコルレベルのエラーを処理できます。

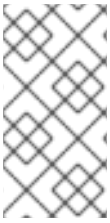
- **connection_error**
- **session_error**
- **sender_error**
- **receiver_error**
- **protocol_error**
- **error**

これらのイベントは、イベントにある特定のオブジェクトにエラー状態が生じるたびに呼び出されます。エラーハンドラーを呼び出すと、対応する **<object>_close** ハンドラーも呼び出されます。

event 引数は、エラーオブジェクトにアクセスするための **error** 属性を持ちます。

例: エラーの処理

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



注記

クローズハンドラーはエラー発生時に呼び出されるため、エラーハンドラー内でのみ処理する必要があります。リソースクリーンアップは、クローズハンドラーで管理できます。特定のオブジェクトに固有のエラー処理がない場合、一般的な **error** イベントを処理することが一般的で、より具体的なハンドラーはありません。

注記

再接続が有効になり、リモートサーバーが **amqp:connection:forced** 条件で接続を閉じると、クライアントはこれをエラーとして処理しないため、**connection_error** イベントは実行されません。代わりに、クライアントが再接続プロセスを開始します。

第10章 ロギング

10.1. ロギングの設定

AMQ JavaScript は JavaScript デバッグモジュール を使用してロギングを実装します。

たとえば、詳細なクライアントロギングを有効にするには、**DEBUG** 環境変数を **rhea*** に設定します。

例: 詳細なロギングの有効化

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

10.2. プロトコルロギングの有効化

クライアントは AMQP プロトコルフレームをコンソールに記録できます。通常、このデータは問題を診断する際に重要です。

プロトコルロギングを有効にするには、**DEBUG** 環境変数を **rhea:frames** に設定します。

例: プロトコルロギングの有効化

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

第11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接続を確立するために使用される設定オプションを読み取りできます。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ーの **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。

11.1. ファイルの場所

設定された場合、AMQ JavaScript は **MESSAGING_CONNECT_FILE** 環境変数の値を使用して設定ファイルを見つけます。

MESSAGING_CONNECT_FILE が設定されていない場合、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. **$PWD/connect.json**。ここでの **$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**。ここでの **%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。

11.2. ファイルフォーマット

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。

すべての設定属性は任意で、またはデフォルト値を持っているため、簡単な例にはいくつかの詳細のみが必要になります。

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 オプションを持つ 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"
    }
}
```

11.3. SSL 設定オプション

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

表11.1 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"**	クリアテキストの **"amqp"** SSL/TLS の **"amqps"** の場合
host	string	**"localhost"**	リモートホストのホスト名または IP アドレス
port	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	**なし**	認証のユーザー名
password	string	**なし**	認証のパスワード
sasl.mechanisms	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。指定のない場合は、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	**なし**	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	**なし**	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	**なし**	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

キー	値のタイプ	デフォルト値	説明
tls.verify	boolean	**true**	一致するホスト名を持つ有効なサーバー証明書が必要

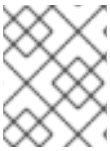
第12章 相互運用性

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する方法を説明します。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。

12.1. 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 を使用して構成されます。この一般的な形式を使用するのは、異なる言語の AMQP クライアントが、相互運用できることが理由です。

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 エンコード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 のタイプに関する詳細は、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12.1 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	バイト数のシーケンス
byte	署名済み 8 ビットの整数
short	署名済み 16 ビット整数
int	署名付き 32 ビット整数
long	署名済み 64 ビット整数
ubyte	署名なし 8 ビット整数
ushort	未署名の 16 ビット整数
uint	署名のない 32 ビット整数
ulong	未署名の 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	単一タイプの値シーケンス
list	変数タイプの値シーケンス
map	異なるキーから値へのマッピング
uuid	ユニバーサルー意識別子
symbol	制限されたドメインからの 7 ビットの ASCII 文字列
timestamp	絶対ポイント (時間単位)

JavaScript がエンコードできるネイティブタイプは、AMQP よりも少ないです。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12.2 エンコードおよびデコードの後における 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 AMQJavaScript 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 AMQJavaScript 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12.3 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	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

エンコード前の AMQ JavaScript タイプ	AMQ C++ タイプ	AMQ .NET タイプ
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**

表12.4 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 のタイプ	AMQ Ruby タイプ
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**

エンコード前の AMQ JavaScript タイプ	AMQ Python のタイプ	AMQ Ruby タイプ
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. 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的なマッピングを定義します。本項では、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 相互運用性」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。以下の表は、特定のボディ型が JMS メッセージタイプにマッピングする方法を示しています。

結果として生成される JMS メッセージタイプの明示的な制御を行うために、**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 相互運用性」の章を参照してください。

表12.5 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
string	**TextMessage**
null	**TextMessage**
wrap_binary(string)	**BytesMessage**

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

12.3. 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。以下をチェックして、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ます。「デフォルトのアクセプター設定」を参照してください。
- 必要なアドレスはブローカーで設定されます。「Addresses, Queues, and Topics」を参照してください。
- ブローカーはクライアントからアクセスを許可するよう設定され、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。

12.4. AMQ INTERCONNECT への接続

AMQ Interconnect は AMQP 1.0 クライアントと動作します。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アクセスを許可するよう設定され、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ー保護」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 リポジトリーを使用している場合、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. **INTEGRATION AND AUTOMATION** カテゴリーで **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. パッケージを受信するためのシステムの登録

この製品の RPM パッケージを Red Hat Enterprise Linux にインストールするには、お使いのシステムを登録する必要があります。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。

システムを登録する方法は、以下のリソースを参照してください。

- Red Hat Enterprise Linux 7 - システム登録およびサブスクリプション管理
- Red Hat Enterprise Linux 8 - システム登録およびサブスクリプション管理

付録B サンプルでの AMQ BROKER の使用

AMQ JavaScript のサンプルでは、**examples** という名前のキューが含まれる実行中のメッセージブローカーが必要です。以下の手順に従って、ブローカーをインストールして起動し、キューを定義します。

B.1. ブローカーのインストール

『**AMQ Broker の使用**』の説明に従い ブロッカーをインストール して、ブローカーインスタンスを作成 します。匿名アクセスを有効にします。

以下の手順では、**<broker-instance-dir>** としてブローカーインスタンスの場所を参照します。

B.2. ブローカーの起動

手順

1. **artemis run** コマンドを使用してブローカーを起動します。

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

2. コンソールの出力で、起動時にログに記録される重要なエラーの有無を確認します。ブローカーは、準備が整う際に **Server is now live** をログに記録します。

```
$ 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. キューの作成

新しいターミナルで、**artemis queue** コマンドを使用して **examples** という名前のキューを作成します。

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

yes または no の質問への回答を求めるプロンプトが表示されます。そのすべてに no (**N**) と回答します。

キューが作成されると、ブローカーはサンプルプログラムと使用できるようになります。

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: 2021-08-29 15:56:26 +1000