Red Hat AMQ Clients 2.11

AMQ JavaScript クライアントの使用

AMQ Clients 2.11 向け

Last Updated: 2023-07-28

Red Hat AMQ Clients 2.11 AMQ JavaScript クライアントの使用

AMQ Clients 2.11 向け

法律上の通知

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

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。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。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。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ート) 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 2.11 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 でサポートされている設定は、Red Hat カスタマーポータルの Red Hat AMQ Supported Configurations 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、コア API が連携する方法を説明します。

表1.1 API の用語

エンティティー	説明
Container	接続の最上位のコンテナー。
Connection	ネットワーク上の 2 つのピア間の通信チャネル。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。
Message	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、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 インストールシステム

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。
- 環境に **npm** コマンドラインツールをインストールする必要があります。詳細は、npm の Web サイトを参照してください。
- 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、デバッグ npm ページ を参照してください。

2.2. RED HAT NPM レジストリーの使用

Red Hat NPM レジストリーからクライアントライブラリーをダウンロードするように NPM 環境を設定します。

手順

1. **npm config set** コマンドを使用して、Red Hat NPM レジストリーを環境に追加します。

```
$ sudo npm config set @redhat:registry https://npm.registry.redhat.com
```

2. **npm install** コマンドを使用して、クライアントをインストールします。

```
$ sudo npm install -g @redhat/rhea@1.0.24-redhat-00004
```

注記

上記の手順は、システム全体のインストール用です。**sudo** や **-g** オプションを指定せずにコマンドを実行して、ローカルインストールを実行できます。

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/@redhat** ディレクトリーを **NODE_PATH** 環境変数に追加します。

Red Hat Enterprise Linux 7

```
$ export NODE_PATH=/opt/rh/rh-nodejs14/root/usr/lib/node_modules/@redhat:$NODE_PATH
```

Red Hat Enterprise Linux 8

```
$ export NODE_PATH=/usr/local/lib/node_modules/@redhat:$NODE_PATH
```

Windows

```
$ set NODE_PATH=%AppData%\Roaming\npm\node_modules\@redhat;%NODE_PATH%
```

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. クライアントをブラウザーにデプロイする

AMQ JavaScript は Web ブラウザー内で実行できます。NPM パッケージには、次の場所にある **rhea.js** という名前のファイルが含まれており、ブラウザーベースのアプリケーションで使用できます。

```
/usr/lib/node_modules/@redhat/rhea/dist/rhea.js
```

次の例のように、**rhea.js** ファイルを Web サーバーによって公開されている場所にコピーし、HTML **<script>** 要素を使用して参照します。

以下に例を示します。ブラウザーでクライアントを実行する

```
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
  <script src="rhea.js"></script>
</head>
<body>
  <script>
    const rhea = require("rhea");
    const container = rhea.create_container();

    container.on("message", (event) => {
        console.log(event.message.body);
    });

    const ws = container.websocket_connect(WebSocket);
    const details = ws("ws://example.net:5673", ["binary", "AMQPWSB10", "amqp"])

    const conn = container.connect({"connection_details": details});
    conn.open_receiver("notifications");
  </script>
</body>
```

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 サンプル 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。

以下に例を示します。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。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。

以下に例を示します。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。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 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 は非同期のイベント駆動型 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 **container** 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。

以下に例を示します。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。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 **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。

以下に例を示します。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。

以下に例を示します。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する場合、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。

以下に例を示します。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 を設定しない場合には、コンテナーが処理されると、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。

以下に例を示します。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。

以下に例を示します。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。

以下に例を示します。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。

以下に例を示します。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。

以下に例を示します。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。

以下に例を示します。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。

以下に例を示します。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。

以下に例を示します。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);
```

以下に例を示します。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。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、**durable** および **expiry_policy** プロパティーを指定して、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は、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 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実行されます。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。

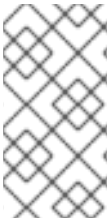
event 引数には、エラーオブジェクトにアクセスするための **error** 属性があります。

以下に例を示します。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。リソースのクリーンアップは、近辺にある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は、一般的な **error** イベントを処理してより具体的なハンドラーを用意しないのが通常です。



注記

再接続が有効になっており、リモートサーバーが **amqp:connection:forced** の条件で接続が切断されると、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。

以下に例を示します。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。多くの場合、このデータは問題の診断時に重要になります。

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。

以下に例を示します。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。

11.2. 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。

以下に例を示します。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。

以下に例を示します。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"
    }
}
```

11.3. 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"**	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
port	string または number	**"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.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

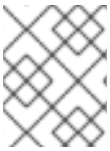
第12章 相互運用性

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12.2 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 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 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、body 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 body 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ている。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. **My Subscriptions** 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、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。

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 例で AMQ ブローカーの使用

AMQ Java Script の例では、名前が **examples**という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

Getting Started with AMQ Brokerの手順に従って、ブロー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、ブローカーインスタンスの場所を **<broker-instance-dir>** と呼びます。

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. 起動時にログに記録された重大なエラーがないか、コンソールの出力を確認してください。ブローカーでは、準備が整うと **Server is now live** とログが記録されます。

```
$ 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
```

プロンプトで質問に Yes または No で回答するように求められます。すべての質問に **N** (いいえ) 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で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
```

Revised on 2022-11-12 21:31:16 +1000