Red Hat build of Eclipse Vert.x 3.9

Eclipse Vert.x ランタイムガイド

Eclipse Vert.x を使用して、OpenShift およびスタンドアロン RHEL で実行するリアクティブで非ブロッキングの非同期アプリケーションを開発

Last Updated: 2021-03-23

Red Hat build of Eclipse Vert.x 3.9 Eclipse Vert.x ランタイムガイド

Eclipse Vert.x を使用して、OpenShift およびスタンドアロン RHEL で実行するリアクティブで非ブロッキングの非同期アプリケーションを開発

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、Eclipse Vert.x ランタイムの使用を説明します。

目次

はじめに

本ガイドでは、概念と、開発者が Eclipse Vert.x ランタイムを使用する際に必要となる実用的な詳細情報を説明します。

RED HAT ドキュメントへのフィードバック

ご意見やご意見をお聞かせください。フィードバックを行うには、ドキュメント内のテキストを強調表示し、コメントを追加します。

本セクションでは、フィードバックの送信方法を説明します。

前提条件

- Red Hat カスタマーポータルにログインします。
- Red Hat カスタマーポータルで、**Multi-page HTML** 形式のドキュメントを表示します。

手順

フィードバックを提供するには、以下の手順を実施します。

1. ドキュメントの右上隅にある **Feedback** ボタンをクリックして、既存のフィードバックを表示します。

注記

フィードバック機能は、**Multi-page HTML** 形式でのみ有効です。

2. フィードバックを提供するドキュメントのセクションを強調表示します。
3. 強調表示されているテキストの近くに表示される **Add Feedback** ポップアップをクリックします。
   ページの右側のフィードバックセクションに、テキストボックスが表示されます。
4. テキストボックスにフィードバックを入力し、**Submit** をクリックします。
   ドキュメントの問題が作成されました。
5. この問題を確認するには、フィードバックビューで問題トラッカーをクリックします。

第1章 ECLIPSE VERT.X を使用したアプリケーション開発の概要

このセクションでは、Red Hat ランタイムでのアプリケーション開発の基本概念を説明します。また、Eclipse Vert.x ランタイムの概要も説明します。

1.1. RED HAT RUNTIMES でのアプリケーション開発の概要

Red Hat OpenShift は、クラウドネイティブランタイムのコレクションを提供するコンテナーアプリケーションプラットフォームです。ランタイムを使用して、OpenShift で Java または JavaScript アプリケーションを開発、ビルド、およびデプロイできます。

Red Hat Runtimes for OpenShift を使用したアプリケーション開発には、以下が含まれます。

- OpenShift 上で実行されるように設計された Eclipse Vert.x、Thorntail、Spring Boot などのランタイムのコレクション。
- OpenShift でのクラウドネイティブ開発への規定的なアプローチ。

OpenShift は、アプリケーションのデプロイメントおよび監視の管理、保護、自動化に役立ちます。ビジネス上の問題を小規模なマイクロサービスに分割し、OpenShift を使用してマイクロサービスをデプロイし、監視し、維持することができます。サーキットブレーカー、ヘルスチェック、サービス検出などのパターンをアプリケーションに実装できます。

クラウドネイティブな開発は、クラウドコンピューティングを最大限に活用します。

以下でアプリケーションをビルドし、デプロイし、管理できます。

OpenShift Container Platform

Red Hat のプライベートオンプレミスクラウド。

Red Hat Container Development Kit (Minishift)

ローカルマシンにインストールおよび実行できるローカルクラウド。この機能は、Red Hat Container Development Kit (CDK) または Minishift で提供されます。

Red Hat CodeReady Studio

アプリケーションの開発、テスト、デプロイを行う統合開発環境 (IDE)。

アプリケーション開発を開始できるようにするため、サンプルアプリケーションですべてのランタイムが利用可能になります。これらのサンプルアプリケーションは Developer Launcher からアクセスできます。サンプルをテンプレートとして使用してアプリケーションを作成することができます。

本ガイドでは、Eclipse Vert.x ランタイムに関する詳細情報を提供します。その他のランタイムの詳細は、関連する ランタイムドキュメント を参照してください。

1.2. DEVELOPER LAUNCHER を使用した RED HAT OPENSHIFT でのアプリケーション開発

Developer Launcher (developers.redhat.com/launch) を使用して、OpenShift でのクラウドネイティブアプリケーションの開発を開始することができます。これは、Red Hat が提供するサービスです。

Developer Launcher はスタンドアロンのプロジェクトジェネレーターです。これを使用して、OpenShift Container Platform、Minshift、CDK などの OpenShift インスタンスでアプリケーションをビルドし、デプロイできます。

1.3. ECLIPSE VERT.X の概要

Eclipse Vert.x は、JVM (Java Virtual Machine) で実行される、リアクティブで非ブロッキングの非同期アプリケーションを作成するのに使用されるツールキットです。

Eclipse Vert.x はクラウドネイティブとなるように設計されています。これにより、アプリケーションは非常に少ないスレッドを使用できます。これにより、新規スレッドの作成時に生じるオーバーヘッドを回避します。これにより、Eclipse Vert.x アプリケーションおよびサービスで、クラウド環境でメモリーと CPU クォータが効果的に使用されます。

OpenShift で Eclipse Vert.x ランタイムを使用すると、リアクティブなシステムのビルドが簡単になります。ローリングアップデート、サービス検出、カナリアデプロイメントなどの OpenShift プラットフォーム機能も利用できます。OpenShift では、外部化の設定、ヘルスチェック、サーキットブレーカー、フェイルオーバーなどのマイクロサービスパターンをアプリケーションに実装できます。

1.3.1. Eclipse Vert.x の主な概念

本セクションでは、Eclipse Vert.x ランタイムに関連する主な概念を説明します。また、リアクティブなシステムの概要も説明します。

クラウドネイティブおよびコンテナーネイティブアプリケーション

クラウドネイティブアプリケーションは、通常マイクロサービスを使用してビルドされます。分離コンポーネントの分散システムを形成するように設計されています。これらのコンポーネントは、通常、多数のノードを含むクラスターでコンテナー内で実行されます。これらのアプリケーションは、個々のコンポーネントの障害に対して耐性があることが期待されており、サービスのダウンタイムなく更新できます。クラウドネイティブアプリケーションに基づくシステムは、基盤となるクラウドプラットフォーム (OpenShift など) によって提供される自動デプロイメント、スケーリング、管理タスク、メンテナンスタスクに依存します。管理および管理タスクは、個々のマシンレベルではなく、既成の管理およびオーケストレーションツールを使用してクラスターレベルで実行されます。

リアクティブシステム

reactive manifesto に定義されているリアクティブなシステムは、以下の特徴を持つ分散システムです。

柔軟性

システムは、さまざまなワークロードで応答性を維持し、必要に応じて個々のコンポーネントをスケーリングして負荷分散することで、ワークロードの違いに対応します。Elastic アプリケーションは、同時に受け取る要求の数に関係なく、同じ品質のサービスを提供します。

耐久性

システムが各コンポーネントで障害が発生した場合でも応答し続けます。システムでは、コンポーネントは相互に分離されます。これにより、個々のコンポーネントに障害が発生した場合に迅速に復元するのに役立ちます。1 つのコンポーネントの障害は、他のコンポーネントの機能に影響を与えることはありません。これにより、分離されたコンポーネントの障害により他のコンポーネントがブロックされ、徐々に障害が発生するようなカスケード障害が防止されます。

応答の早さ

応答するシステムは、一貫性のあるサービス品質を確保するために、合理的な時間内に要求に常に応答するように設計されています。応答を維持するには、アプリケーション間の通信チャネルをブロックすることはできません。

メッセージ駆動型

アプリケーションの個々のコンポーネントは、非同期のメッセージパスを使用して相互に通信します。マウスクリックやサービスの検索クエリーなど、イベントが発生した場合、サービスは一般的なチャネル (イベントバス) にメッセージを送信します。メッセージはそれぞれのコンポーネントに

よってキャッチされ、処理されます。

リアクティブシステムは分散システムです。これらは、アプリケーション開発に非同期プロパティーを使用できるように設計されています。

リアクティブプログラミング

リアクティブシステムの概念は、分散システムのアーキテクチャーを記述しますが、リアクティブプログラミングは、アプリケーションをコードレベルでリアクティブにする手法を指します。リアクティブプログラミングは、非同期およびイベント駆動型のアプリケーションを記述する開発モデルです。リアクティブアプリケーションでは、コードはイベントまたはメッセージに反応します。

リアクティブプログラミングにはいくつかの実装があります。たとえば、コールバックを使用した簡単な実装、Reactive Extensions (Rx) を使用した複雑な実装、およびコルーチンを使用した複雑な実装などです。

Reactive Extensions (Rx) は、Java におけるリアクティブプログラミングの最も成熟した形式の 1 つです。これは **RxJava** ライブラリーを使用します。

1.3.2. Eclipse Vert.x でサポートされているアーキテクチャー

Eclipse Vert.x は、以下のアーキテクチャーをサポートします。

- x86_64 (AMD64)
- OpenShift 環境の IBM Z (s390x)
- OpenShift 環境の IBM Power Systems (ppc64le)

アーキテクチャーによって異なるイメージがサポートされています。本ガイドのコード例は、x86_64 アーキテクチャーのコマンドを示しています。他のアーキテクチャーを使用している場合は、コマンドに該当するイメージ名を指定します。

イメージ名の詳細は、「Eclipse Vert.x でサポートされる Java イメージ」 セクションを参照してください。

1.3.3. サンプルアプリケーションの概要

クラウドネイティブのアプリケーションおよびサービスをビルドする方法を実証する作業アプリケーションがあります。これらは、アプリケーションの開発時に使用する必要のある規範的なアーキテクチャー、設計パターン、ツール、およびベストプラクティスを示しています。サンプルアプリケーションは、クラウドネイティブのマイクロサービスを作成するためのテンプレートとして使用できます。本ガイドで説明しているデプロイメントプロセスを使用して、例を更新および再デプロイできます。

この例では、以下のような マイクロサービスパターン を実装します。

- REST API の作成
- データベースの相互運用
- ヘルスチェックパターンの実装
- アプリケーションの設定を外部化してセキュア化し、容易なスケーリング

サンプルアプリケーションは以下のように使用できます。

- テクノロジーのデモの実行

- プロジェクトのアプリケーションの開発方法を理解するツールまたはサンドボックスの学習
- 独自のユースケースを更新または拡張するためのヒント

各サンプルアプリケーションは 1 つ以上のランタイムに実装されます。たとえば、REST API Level 0 のサンプルは以下のランタイムで利用できます。

- Node.js
- Spring Boot
- Eclipse Vert.x
- Thorntail

これ以降のセクションでは、Eclipse Vert.x ランタイムに実装されたサンプルアプリケーションを説明します。

すべてのサンプルアプリケーションを以下にダウンロードおよびデプロイできます。

- x86_64 アーキテクチャー - 本ガイドのアプリケーションの例では、サンプルアプリケーションを x86_64 アーキテクチャーにビルドおよびデプロイする方法を説明します。
- s390x アーキテクチャー - IBM Z インフラストラクチャーでプロビジョニングされた OpenShift 環境にサンプルアプリケーションをデプロイするには、コマンドに関連する IBM Z イメージ名を指定します。
- ppc64le アーキテクチャー - IBM Power Systems インフラストラクチャーにプロビジョニングされた OpenShift 環境にアプリケーションのサンプルをデプロイするには、コマンドで関連する IBM Power Systems イメージ名を指定します。

イメージ名の詳細は、「Eclipse Vert.x でサポートされる Java イメージ」 セクションを参照してください。

サンプルアプリケーションの一部には、ワークフローを実証するために Red Hat Data Grid などの他の製品も必要になります。この場合は、これらの製品のイメージ名を、サンプルアプリケーションの YAML ファイルで関連する IBM Z または IBM Power System のイメージ名に変更する必要もあります。

第2章 アプリケーションの設定

本セクションでは、Eclipse Vert.x ランタイムと連携するようにアプリケーションを設定する方法を説明します。

2.1. ECLIPSE VERT.X を使用するようにアプリケーションを設定

前提条件

- Maven ベースのアプリケーション

手順

1.
```
<project>
 ...
 <dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-dependencies</artifactId>
    <version>${vertx.version}</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
  </dependencies>
 </dependencyManagement>
 ...
</project>
```

2.
```
<project>
 ...
 <properties>
  <vertx.version>${vertx.version}</vertx.version>
  <vertx-maven-plugin.version>${vertx-maven-plugin.version}</vertx-maven-plugin.version>
 </properties>
 ...
</project>
```

3.
```
<project>
 ...
 <build>
  <plugins>
    ...
    <plugin>
      <groupId>io.reactiverse</groupId>
      <artifactId>vertx-maven-plugin</artifactId>
      <version>${vertx-maven-plugin.version}</version>
      <executions>
        <execution>
          <id>vmp</id>
          <goals>
            <goal>initialize</goal>
            <goal>package</goal>
```

```
                </goals>
              </execution>
            </executions>
            <configuration>
              <redeploy>true</redeploy>
            </configuration>
          </plugin>
          ...
        </plugins>
      </build>
      ...
    </project>
```

4. **repositories** および **pluginRepositories** を追加して、アプリケーションをビルドするためのアーティファクトおよびプラグインが含まれるリポジトリーを指定します。

```
<project>
...
  <repositories>
    <repository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </pluginRepository>
  </pluginRepositories>
...
</project>
```

関連情報

- Eclipse Vert.x アプリケーションのパッケージ化に関する詳細は、Vert.x Maven プラグイン のドキュメントを参照してください。

2.2.

注記

手順

1.

url	jdbcUrl

driver_class	driverClassName
user	
パスワード	
castUUID	castUUID

追加情報

- ```
  JsonObject config = new JsonObject()
   .put("url", JDBC_URL)
   // set C3P0 as the JDBC connection pool:
   .put("provider_class", "io.vertx.ext.jdbc.spi.impl.C3P0DataSourceProvider")
   .put("driver_class", "org.postgresql.Driver")
   .put("user", JDBC_USER)
   .put("password", JDBC_PASSWORD)
   .put("castUUID", true);
  ```

- ```
  JsonObject config = new JsonObject()
   .put("jdbcUrl", JDBC_URL)
   .put("driverClassName", "org.postgresql.Driver")
   .put("principal", JDBC_USER)
   .put("credential", JDBC_PASSWORD)
   .put("castUUID", true);
  ```

第3章 DEVELOPER LAUNCHER を使用したアプリケーションのダウンロードおよびデプロイ

このセクションでは、ランタイムで提供されるサンプルアプリケーションをダウンロードおよびデプロイする方法を説明します。アプリケーションのサンプルは Developer Launcher で利用できます。

3.1. DEVELOPER LAUNCHER の使用

Developer Launcher (developers.redhat.com/launch) は OpenShift 上で実行します。アプリケーションのサンプルをデプロイする場合、Developer Launcher は以下のプロセスを説明します。

- ランタイムの選択
- アプリケーションのビルドおよび実行

選択に基づいて、Developer Launcher はカスタムプロジェクトを生成します。プロジェクトの ZIP バージョンをダウンロードするか、または OpenShift Online インスタンスでアプリケーションを直接起動できます。

Developer Launcher を使用してアプリケーションを OpenShift にデプロイする場合は、Source-to-Image (S2I) ビルドプロセスが使用されます。このビルドプロセスは、OpenShift でアプリケーションを実行するために必要なすべての設定、ビルド、およびデプロイメントのステップを処理します。

3.2. DEVELOPER LAUNCHER を使用したサンプルアプリケーションのダウンロード

Red Hat は、Eclipse Vert.x ランタイムを使い始めるのに役立つサンプルアプリケーションを提供します。これらの例は、Developer Launcher (developers.redhat.com/launch) で利用できます。

サンプルアプリケーションをダウンロードして、ビルドし、デプロイできます。本セクションでは、サンプルアプリケーションをダウンロードする方法を説明します。

サンプルアプリケーションをテンプレートとして使用し、独自のクラウドネイティブアプリケーションを作成できます。

手順

1. Developer Launcher (developers.redhat.com/launch) に移動します。
2. **Start** をクリックします。
3. **Deploy an Example Application**をクリックします。
4. **Select an Example** をクリックし、ランタイムで使用できるサンプルアプリケーションの一覧を表示します。
5. ランタイムを選択します。
6. サンプルアプリケーションを選択します。

注記

複数のランタイムで利用できるアプリケーションの例もあります。前の手順でランタイムを選択していない場合は、サンプルアプリケーションで利用できるランタイムの一覧からランタイムを選択できます。

7. ランタイムのリリースバージョンを選択します。ランタイムに一覧表示されているコミュニティーまたは製品リリースから選択できます。
8. **Save** をクリックします。
9. **Download** をクリックして、サンプルアプリケーションをダウンロードします。
ソースおよびドキュメントファイルを含む ZIP ファイルがダウンロードされます。

3.3. OPENSHIFT CONTAINER PLATFORM または CDK (MINISHIFT) へのサンプルアプリケーションのデプロイメント

サンプルアプリケーションを OpenShift Container Platform または CDK (Minishift) のいずれかにデプロイできます。アプリケーションをデプロイする場所に応じて、認証に該当する Web コンソールを使用します。

前提条件

- Developer Launcher を使用して、サンプルアプリケーションプロジェクトを作成している。
- アプリケーションを OpenShift Container Platform にデプロイする場合は、OpenShift Container Platform Web コンソールにアクセスできるようにしている。
- CDK (Minishift) にアプリケーションをデプロイする場合は、CDK (Minishift) Web コンソールにアクセスできるようにしている。
- **OC** コマンドラインクライアントがインストールされている。

手順

1. サンプルアプリケーションをダウンロードします。
2. **oc** コマンドラインクライアントを使用して、サンプルアプリケーションを OpenShift Container Platform または CDK (Minishift) にデプロイできます。
Web コンソールによって提供されるトークンを使用してクライアントを認証する必要があります。アプリケーションをデプロイする場所に応じて、OpenShift Container Platform Web コンソールまたは CDK (Minishift) Web コンソールを使用します。以下の手順を実行してクライアントの認証を取得します。
   a. Web コンソールにログインします。
   b. Web コンソールの右上隅にあるクエスチョンマークアイコンをクリックします。
   c. 一覧から **Command Line Tools** を選択します。
   d. **oc login** コマンドをコピーします。
   e. ターミナルにコマンドを貼り付け、CLI クライアント **oc** をアカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

3. ZIP ファイルの内容を展開します。

```
$ unzip MY_APPLICATION_NAME.zip
```

4. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

5. **MY_APPLICATION_NAME** の root ディレクトリーに移動します。

6. Maven を使用してサンプルアプリケーションをデプロイします。

```
$ mvn clean fabric8:deploy -Popenshift
```

注記: アプリケーションの例によっては、追加の設定が必要になる場合があります。サンプルアプリケーションをビルドおよびデプロイするには、**README** ファイルに記載されている手順に従います。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa         1/1       Running     0          58s
MY_APP_NAME-s2i-1-build     0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は完全にデプロイされ起動すると **Running** ステータスになります。アプリケーションの Pod 名は異なる場合があります。Pod 名の数値は、新しいビルドごとに増加します。末尾の文字は、Pod の作成時に生成されます。

8. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME            HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME     MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME     8080
```

Pod のルート情報は、アクセスできるベース URL を提供します。この例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスできます。

第4章 ECLIPSE VERT.X ランタイムアプリケーションの開発およびデプロイ

例を使用する 以外に、Eclipse Vert.x アプリケーションを新規作成し、OpenShift またはスタンドアロンの Red Hat Enterprise Linux にデプロイできます。

4.1. ECLIPSE VERT.X アプリケーションの開発

基本的な Eclipse Vert.x アプリケーションには、以下を作成する必要があります。

- Eclipse Vert.x メソッドを含む Java クラス。
- アプリケーションをビルドするために Maven が必要とする情報が含まれる **pom.xml** ファイル。

以下の手順では、応答として「Greetings!」を返す単純な **Greeting** アプリケーションを作成します。

注記

アプリケーションをビルドおよび OpenShift にデプロイする場合、Eclipse Vert.x 3.9 は OpenJDK 8 および OpenJDK 11 をベースとしたビルダーイメージのみをサポートします。Oracle JDK および OpenJDK 9 のビルダーイメージはサポートされていません。

前提条件

- OpenJDK 8 または OpenJDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. **myApp** ディレクトリーを新規作成し、そのディレクトリーに移動します。

```
$ mkdir myApp
$ cd myApp
```

これは、アプリケーションのルートディレクトリーです。

2. ルートディレクトリーにディレクトリー構造 **src/main/java/com/example/** を作成し、これに移動します。

```
$ mkdir -p src/main/java/com/example/
$ cd src/main/java/com/example/
```

3. アプリケーションコードを含む Java クラスファイル **MyApp.java** を作成します。

```
package com.example;

import io.vertx.core.AbstractVerticle;
import io.vertx.core.Future;

public class MyApp extends AbstractVerticle {
```

```
    @Override
    public void start(Future<Void> fut) {
        vertx
            .createHttpServer()
            .requestHandler(r ->
                r.response().end("Greetings!"))
            .listen(8080, result -> {
                if (result.succeeded()) {
                    fut.complete();
                } else {
                    fut.fail(result.cause());
                }
            });
    }
}
```

4. 以下の内容を含むアプリケーションルートディレクトリー **myApp** に **pom.xml** ファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>My Application</name>
  <description>Example application using Vert.x</description>

  <properties>
    <vertx.version>3.9.5.redhat-00001</vertx.version>
    <vertx-maven-plugin.version>1.0.20</vertx-maven-plugin.version>
    <vertx.verticle>com.example.MyApp</vertx.verticle>

    <!-- Specify the JDK builder image used to build your application. -->
    <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</fabric8.generator.from>

    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>

  <!-- Import dependencies from the Vert.x BOM. -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>io.vertx</groupId>
        <artifactId>vertx-dependencies</artifactId>
        <version>${vertx.version}</version>
```

```
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<!-- Specify the Vert.x artifacts that your application depends on. -->
<dependencies>
  <dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-core</artifactId>
  </dependency>
  <dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-web</artifactId>
  </dependency>
</dependencies>

<!-- Specify the repositories containing Vert.x artifacts. -->
<repositories>
  <repository>
    <id>redhat-ga</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>

<!-- Specify the repositories containing the plugins used to execute the build of your
application. -->
<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
</pluginRepositories>

<!-- Configure your application to be packaged using the Vert.x Maven Plugin. -->
<build>
  <plugins>
    <plugin>
      <groupId>io.reactiverse</groupId>
      <artifactId>vertx-maven-plugin</artifactId>
      <version>${vertx-maven-plugin.version}</version>
      <executions>
        <execution>
          <id>vmp</id>
          <goals>
            <goal>initialize</goal>
            <goal>package</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

```
    </plugins>
  </build>
</project>
```

5. アプリケーションのルートディレクトリーから Maven を使用してアプリケーションをビルドします。

```
$ mvn vertx:run
```

6. アプリケーションが実行していることを確認します。
curl またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で稼働していることを確認します。

```
$ curl http://localhost:8080
Greetings!
```

追加情報

- 推奨されるプラクティスとして、Liveness プローブおよび Readiness プローブを設定し、OpenShift で実行する際にアプリケーションのヘルスモニタリングを有効にできます。OpenShift でのアプリケーションのヘルスモニタリングの動作を確認するには、Health Check サンプル を試してください。

4.2. ECLIPSE VERT.X アプリケーションの OPENSHIFT へのデプロイメント

イメージは Red Hat Ecosystem Catalog で利用できます。

```
<fabric8.generator.from>IMAGE_NAME</fabric8.generator.from>
```

たとえば、OpenJDK 8 を使用する RHEL 7 の Java イメージは以下のようになります。

```
<fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</fabric8.generator.from>
```

4.2.1. Eclipse Vert.x でサポートされる Java イメージ

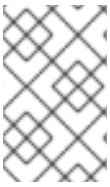
Eclipse Vert.x は、さまざまなオペレーティングシステムで利用可能なさまざまな Java イメージで認定およびテストされています。同様のイメージは、IBM Z および IBM Power Systems で利用できます。

Red Hat Ecosystem Catalog で RHEL 8 イメージにアクセスするには、Docker または Podman 認証が必要です。

以下の表には、さまざまなアーキテクチャーの Eclipse Vert.x で対応しているイメージが記載されています。また、Red Hat Ecosystem Catalog で利用可能なイメージへのリンクも提供します。イメージページには、RHEL 8 イメージへのアクセスに必要な認証手順が含まれています。

4.2.1.1. x86_64 アーキテクチャー上のイメージ

OS	Java	Red Hat Ecosystem Catalog
RHEL 7	OpenJDK 8	OpenJDK 8 を使用した RHEL 7
RHEL 7	OpenJDK 11	OpenJDK 11 を使用した RHEL 7
RHEL 8	OpenJDK 8	OpenJDK 8 を使用した RHEL 8
RHEL 8	OpenJDK 11	OpenJDK 11 を使用した RHEL 8



注記

RHEL 7 ホストでの RHEL 8 ベースのコンテナーの使用 (OpenShift 3 や OpenShift 4 など) は、サポートが限定されています。詳細は、「Red Hat Enterprise Linux Container Compatibility Matrix」を参照してください。

4.2.1.2. s390x (IBM Z) アーキテクチャー上のイメージ

OS	Java	Red Hat Ecosystem Catalog
RHEL 8	Eclipse OpenJ9 11	Eclipse OpenJ9 11 を使用した RHEL 8

4.2.1.3. ppc64le (IBM Power Systems) アーキテクチャー上のイメージ

OS	Java	Red Hat Ecosystem Catalog
RHEL 8	Eclipse OpenJ9 11	Eclipse OpenJ9 11 を使用した RHEL 8

注記

RHEL 7 ホストでの RHEL 8 ベースのコンテナーの使用 (OpenShift 3 や OpenShift 4 など) は、サポートが限定されています。詳細は、「Red Hat Enterprise Linux Container Compatibility Matrix」を参照してください。

4.2.2. OpenShift デプロイメント用の Eclipse Vert.x アプリケーションの準備

Eclipse Vert.x アプリケーションを OpenShift にデプロイするには、以下を含める必要があります。

- アプリケーションの **pom.xml** ファイルにあるランチャープロファイル情報。
-

前提条件

- Maven がインストールされている。

- Red Hat Ecosystem Catalog での Docker または Podman 認証による RHEL 8 イメージへのアクセスができる。

手順

1. 以下の内容を、アプリケーションのルートディレクトリーの **pom.xml** ファイルに追加します。

```
<!-- Specify the JDK builder image used to build your application. -->
<properties>
  <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</fabric8.generator.from>
</properties>

...

<profiles>
   <profile>
     <id>openshift</id>
     <build>
       <plugins>
         <plugin>
           <groupId>io.fabric8</groupId>
           <artifactId>fabric8-maven-plugin</artifactId>
           <version>4.4.1</version>
           <executions>
             <execution>
               <goals>
                 <goal>resource</goal>
                 <goal>build</goal>
               </goals>
             </execution>
           </executions>
         </plugin>
       </plugins>
     </build>
   </profile>
</profiles>
```

2. - x86_64 アーキテクチャー
     - OpenJDK 8 を使用した RHEL 7

       ```
       <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
       openshift:latest</fabric8.generator.from>
       ```

     - OpenJDK 11 を使用した RHEL 7

       ```
       <fabric8.generator.from>registry.access.redhat.com/openjdk/openjdk-11-
       rhel7:latest</fabric8.generator.from>
       ```

     - OpenJDK 8 を使用した RHEL 8

       ```
       <fabric8.generator.from>registry.redhat.io/openjdk/openjdk-8-
       rhel8:latest</fabric8.generator.from>
       ```

- OpenJDK 11 を使用した RHEL 8

```
<fabric8.generator.from>registry.redhat.io/openjdk/openjdk-11-rhel8:latest</fabric8.generator.from>
```

- s390x (IBM Z) アーキテクチャー
  - Eclipse OpenJ9 11 を使用した RHEL 8

```
<fabric8.generator.from>registry.access.redhat.com/openj9/openj9-11-rhel8:latest</fabric8.generator.from>
```

- ppc64le (IBM Power Systems) アーキテクチャー
  - Eclipse OpenJ9 11 を使用した RHEL 8

```
<{fabric8}.generator.from>registry.access.redhat.com/openj9/openj9-11-rhel8:latest</{fabric8}.generator.from>
```

3.
```
spec:
  template:
    spec:
      containers:
        - name: vertx
          env:
            - name: KUBERNETES_NAMESPACE
              valueFrom:
                fieldRef:
                  apiVersion: template.openshift.io/v1
                  fieldPath: metadata.namespace
            - name: JAVA_OPTIONS
              value: '-Dvertx.cacheDirBase=/tmp -Dvertx.jgroups.config=default'
```

4.2.3.

Eclipse Vert.x アプリケーションを OpenShift にデプロイするには、以下を実行する必要があります。

- OpenShift インスタンスにログインします。
- OpenShift インスタンスにアプリケーションをデプロイします。

前提条件

- CLI クライアント **oc** がインストールされている。
- Maven がインストールされている。

手順

1. **oc** クライアントを使用して OpenShift インスタンスにログインします。

```
$ oc login ...
```

2. OpenShift インスタンスで新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションのルートディレクトリーから Maven を使用してアプリケーションを OpenShift にデプロイします。アプリケーションのルートディレクトリーには **pom.xml** ファイルが含まれます。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

4. デプロイメントを確認します。

a. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa         1/1       Running     0          58s
MY_APP_NAME-s2i-1-build     0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが**Running** になるはずです。

特定の Pod 名が異なります。

b. Pod のルートを確認します。

ルート情報の例

```
$ oc get routes
NAME            HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME     MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME     MY_APP_NAME     8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。

この例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスできます。

c. OpenShift でアプリケーションが実行していることを確認します。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
Greetings!
```

4.3. スタンドアロンの RED HAT ENTERPRISE LINUX への ECLIPSE VERT.X アプリケーションのデプロイメント

Eclipse Vert.x アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイするには、アプリケーションで **pom.xml** ファイルを設定し、Maven を使用してパッケージ化し、**java -jar** コマンドを使用してデプロイします。

前提条件

- RHEL 7 または RHEL 8 がインストールされている。

4.3.1. スタンドアロンの Red Hat Enterprise Linux デプロイメントのための Eclipse Vert.x アプリケーションの準備

Eclipse Vert.x アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイするには、最初に Maven を使用してアプリケーションをパッケージ化する必要があります。

前提条件

- Maven がインストールされている。

手順

1. 以下の内容をアプリケーションの root ディレクトリーの **pom.xml** ファイルに追加します。

```
...
<build>
  <plugins>
    <plugin>
      <groupId>io.reactiverse</groupId>
      <artifactId>vertx-maven-plugin</artifactId>
      <version>${vertx-maven-plugin.version}</version>
      <executions>
        <execution>
          <id>vmp</id>
          <goals>
            <goal>initialize</goal>
            <goal>package</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
...
```

2. Maven を使用してアプリケーションをパッケージ化します。

```
$ mvn clean package
```

作成される JAR ファイルは **target** ディレクトリーに置かれます。

4.3.2. jar を使用してスタンドアロンの Red Hat Enterprise Linux への Eclipse Vert.x アプリケーションのデプロイメント

Eclipse Vert.x アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイするには、**java -jar** コマンドを使用します。

前提条件

- RHEL 7 または RHEL 8 がインストールされている。
- OpenJDK 8 または OpenJDK 11 がインストールされている。

- アプリケーションが含まれる JAR ファイル。

手順

1. そのアプリケーションで JAR ファイルをデプロイします。

```
$ java -jar my-app-fat.jar
```

2. デプロイメントを確認します。
curl またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で稼働していることを確認します。

```
$ curl http://localhost:8080
```

第5章 ECLIPSE VERT.X ベースのアプリケーションのデバッグ

本セクションでは、ローカルデプロイメントとリモートデプロイメントの両方で、Eclipse Vert.x ベースのアプリケーションのデバッグを説明します。

5.1. リモートのデバッグ

アプリケーションをリモートでデバッグするには、まずデバッグモードで開始するように設定してから、デバッガーを割り当てる必要があります。

5.1.1. デバッグモードでのアプリケーションをローカルで起動

Maven ベースのプロジェクトのデバッグ方法の 1 つが、デバッグポートを指定している間にアプリケーションを手動で起動し、その後にリモートデバッガーをそのポートに接続することです。この方法は、少なくとも以下のアプリケーションのデプロイメントに適用できます。

- **mvn vertx:debug** ゴールを使用してアプリケーションを手動で起動する場合。これにより、デバッグが有効な状態でアプリケーションが開始します。

前提条件

- Maven ベースのアプリケーション

手順

1. コンソールで、アプリケーションでディレクトリーに移動します。
2. アプリケーションを起動し、**-Ddebug.port** 引数を使用してデバッグポートを指定します。

```
$ mvn vertx:debug -Ddebug.port=$PORT_NUMBER
```

ここで、**$PORT_NUMBER** は未使用のポート番号です。リモートデバッガー設定のこの番号をメモします。

-Ddebug.suspend=true 引数を使用して、デバッガーが起動するまでアプリケーションを待機します。

5.1.2. デバッグモードでの OpenShift でのアプリケーションの起動

OpenShift で Eclipse Vert.x ベースのアプリケーションをリモートでデバッグするには、コンテナー内で **JAVA_DEBUG** 環境変数を **true** に設定し、リモートデバッガーからアプリケーションに接続できるようにポート転送を設定する必要があります。

前提条件

- アプリケーションが OpenShift で実行している。
- **oc** バイナリーがインストールされている。
- ターゲット OpenShift 環境で **oc port-forward** コマンドを実行する機能。

手順

1. **oc** コマンドを使用して、利用可能なデプロイメント設定を一覧表示します。

```
$ oc get dc
```

2. アプリケーションのデプロイメント設定で **JAVA_DEBUG** 環境変数を **true** に設定します。これにより、JVM がデバッグ用にポート番号 **5005** を開くように設定されます。以下は例になります。

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG=true
```

3. 設定変更時に自動的に再デプロイするように設定されていない場合は、アプリケーションを再デプロイします。以下は例になります。

```
$ oc rollout latest dc/MY_APP_NAME
```

4. ローカルマシンからアプリケーション Pod へのポート転送を設定します。

   a. 現在実行中の Pod を一覧表示し、アプリケーションが含まれる Pod を検索します。

```
$ oc get pod
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-3-1xrsp         0/1       Running     0          6s
...
```

   b. ポート転送を設定します。

```
$ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5005
```

   ここでは、**$LOCAL_PORT_NUMBER** はローカルマシンで選択した未使用のポート番号になります。リモートデバッガー設定のこの番号をメモします。

5. デバッグが完了したら、アプリケーション Pod の **JAVA_DEBUG** 環境変数の設定を解除します。以下は例になります。

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG-
```

関連情報

デバッグポートをデフォルト (**5005**) から変更する場合は、**JAVA_DEBUG_PORT** 環境変数を設定することもできます。

5.1.3. アプリケーションへのリモートデバッガーの割り当て

デバッグ用にアプリケーションが設定されている場合は、選択したリモートデバッガーを割り当てます。本ガイドでは、Red Hat CodeReady Studio について説明していますが、他のプログラムを使用する場合も手順は同じようになります。

前提条件

- ローカルまたは OpenShift 上で実行し、デバッグ用に設定されたアプリケーション。
- アプリケーションがデバッグをリッスンしているポート番号。
- Red Hat CodeReady Studio がマシンにインストールされている。Red Hat CodeReady Studio ダウンロードページ からダウンロードできます。

手順

1. Red Hat CodeReady Studio を開始します。
2. アプリケーションの新規デバッグ設定を作成します。
   a. **Run→Debug Configurations** をクリックします。
   b. 設定のリストで、**Remote Java アプリケーション** をダブルクリックします。これにより、新しいリモートデバッグ設定が作成されます。
   c. **Name** フィールドに設定に適した名前を入力します。
   d. アプリケーションが含まれるディレクトリーへのパスを **Project** フィールドに入力します。便宜上、**Browse...** ボタンを使用できます。
   e. **Connection Type** フィールドを **Standard (Socket Attach)** に設定していない場合は、これを設定します。
   f. **Port** フィールドを、アプリケーションがデバッグをリッスンしているポート番号に設定します。
   g. **Apply** をクリックします。
3. Debug Configurations ウィンドウの **Debug** ボタンをクリックして、デバッグを開始します。初めてデバッグ設定を迅速に起動するには **Run→Debug History** をクリックし、一覧から設定を選択します。

関連情報

- Red Hat ナレッジベースの「Debug an OpenShift Java Application with JBoss Developer Studio」
  Red Hat CodeReady Studio はこれまで JBoss Developer Studio と呼ばれていました。
- OpenShift ブログの記事 「Debugging Java Applications On OpenShift and Kubernetes」

5.2. デバッグログ

Eclipse Vert.x は組み込みロギング API を提供します。Eclipse Vert.x のデフォルトのロギング実装は、Java JDK で提供される **java.util.logging** ライブラリーを使用します。Eclipse Vert.x では、**Log4J** (Eclipse Vert.x は Log4J v1 および v2 をサポート)、**SLF4J** などの異なるロギングフレームワークを使用できます。

5.2.1. java.util.logging を使用した Eclipse Vert.x アプリケーションのロギング設定

java.util.logging を使用して Eclipse Vert.x アプリケーションのデバッグロギングを設定するには、以下を実行します。

- **application.properties** ファイルに **java.util.logging.config.file** システムプロパティーを設定します。この変数の値は、**java.util.logging** 設定ファイル の名前に対応する必要があります。これにより、アプリケーションの起動時に **LogManager** が **java.util.logging** を初期化できるようになります。
- **vertx-default-jul-logging.properties** 名を持つ **java.util.logging** 設定ファイルを Maven プロジェクトのクラスパスに追加します。Eclipse Vert.x はこのファイルを使用して、アプリケーションの起動時に **java.util.logging** を設定します。

Eclipse Vert.x では、**Log4J** ライブラリー、**Log4J2** ライブラリー、および **SLF4J** ライブラリーに事前実装を提供する **LogDelegateFactory** を使用してカスタムのロギングバックエンドを指定できます。デフォルトで Java に含まれる **java.util.logging** とは異なり、他のバックエンドでは各ライブラリーをアプリケーションの依存関係として指定する必要があります。

5.2.2. Eclipse Vert.x アプリケーションにログ出力を追加

1. ロギングをアプリケーションに追加するには、**io.vertx.core.logging.Logger** を作成します。

```
Logger logger = LoggerFactory.getLogger(className);

logger.info("something happened");
logger.error("oops!", exception);
logger.debug("debug message");
logger.warn("warning");
```

注意

ロギングバックエンドは異なる形式を使用して、パラメーター化されたメッセージで置き換え可能なトークンを表します。パラメーター化されたロギングメソッドに依存すると、コードを変更せずにロギングバックエンドを切り替えることができません。

5.2.3. アプリケーションのカスタムロギングフレームワークの指定

Eclipse Vert.x で **java.util.logging** を使用しない場合は、**io.vertx.core.logging.Logger** を設定し、別のロギングフレームワークを使用するように設定します (例: **Log4J** または **SLF4J**)。

1. **vertx.logger-delegate-factory-class-name** システムプロパティーの値を **LogDelegateFactory** インターフェースを実装するクラスの名前に設定します。Eclipse Vert.x は、以下のライブラリーに事前ビルドされた実装を提供し、以下のように対応する事前定義されたクラス名を提供します。

ライブラリー	クラス名
Log4J v1	**io.vertx.core.logging.Log4jLogDelegateFactory**
Log4J v2	**io.vertx.core.logging.Log4j2LogDelegateFactory**
SLF4J	**io.vertx.core.logging.SLF4JLogDelegateFactory**

カスタムライブラリーを使用してロギングを実装する場合は、関連する **Log4J** jar または **SLF4J** jar がアプリケーションの依存関係に含まれていることを確認します。

注意

Eclipse Vert.x で提供される **Log4J** v1 委譲は、パラメーター化されたメッセージに対応していません。**Log4J v2** および **SLF4J** の委譲はどちらも **{}** 構文を使用します。**java.util.logging** 委譲は、**{n}** 構文を使用する **java.text.MessageFormat** に依存します。

5.2.4. Eclipse Vert.x アプリケーション用の Netty ロギングの設定

Netty は、アプリケーション内の非同期ネットワーク通信を管理する VertX によって使用されるライブラリーです。

Netty:

- プロトコルサーバーやクライアントなど、ネットワークアプリケーションを迅速かつ簡単に開発できます。
- TCP や UDP ソケットサーバーの開発などのネットワークプログラミングを簡素化し、整備します。
- ブロックおよび非ブロッキング接続を管理する統合 API を提供します。

Netty は、システムプロパティーを使用して外部ロギング設定に依存しません。代わりに、プロジェクトの Netty クラスに表示されるロギングライブラリーに基づいてロギング設定を実装します。Netty は、以下の順序でライブラリーの使用を試みます。

1. **SLF4J**
2. **Log4J**
3. フォールバックオプションとしての **java.util.logging**

アプリケーションの **main** メソッドの冒頭に以下のコードを追加すると、**io.netty.util.internal.logging.InternalLoggerFactory** を直接特定のロガーに直接設定できます。

```
// Force logging to Log4j
InternalLoggerFactory.setDefaultFactory(Log4JLoggerFactory.INSTANCE);
```

5.2.5. OpenShift でのデバッグログへのアクセス

アプリケーションを起動し、これと対話して、OpenShift のデバッグステートメントを確認します。

前提条件

- CLI クライアント **oc** がインストールされ、認証されている。
- デバッグロギングが有効になっている Maven ベースのアプリケーション。

手順

1. アプリケーションを OpenShift にデプロイします。

```
$ mvn clean fabric8:deploy -Popenshift
```

2. ログを表示します。
    1. アプリケーションと共に Pod の名前を取得します。

```
$ oc get pods
```

    2. ログ出力の監視を開始します。

```
$ oc logs -f pod/MY_APP_NAME-2-aaaaa
```

ログ出力を確認できるように、端末ウィンドウにログ出力が表示されます。

3. アプリケーションと対話します。
たとえば、REST API Level 0 のサンプル にデバッグロギングがあり、**/api/greeting** メソッドで **message** 変数をログに記録します。

    1. アプリケーションのルートを取得します。

    ```
    $ oc get routes
    ```

    2. アプリケーションの **/api/greeting** エンドポイントで HTTP 要求を作成します。

    ```
    $ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
    ```

4. Pod ログのあるウィンドウに戻り、ログでデバッグロギングメッセージを検査します。

```
...
Feb 11, 2017 10:23:42 AM io.openshift.MY_APP_NAME
INFO: Greeting: Hello, Sarah
...
```

5. デバッグロギングを無効にするには、ロギング設定ファイル (例: **src/main/resources/vertx-default-jul-logging.properties**) を更新し、クラスのロギング設定を削除し、アプリケーションを再デプロイします。

第6章 アプリケーションのモニタリング

本セクションでは、OpenShift 上で実行する Eclipse Vert.x ベースのアプリケーションのモニタリングを説明します。

6.1. OPENSHIFT でのアプリケーションの JVM メトリクスへのアクセス

6.1.1. OpenShift で Jolokia を使用した JVM メトリクスへのアクセス

Jolokia は、OpenShift 上の HTTP (Java Management Extension) メトリクスにアクセスするための組み込みの軽量ソリューションです。Jolokia を使用すると、HTTP ブリッジ上で JMX によって収集される CPU、ストレージ、およびメモリー使用状況データにアクセスできます。Jolokia は REST インターフェースおよび JSON 形式のメッセージペイロードを使用します。これは、非常に高速で、リソース要件が低いため、クラウドアプリケーションのモニタリングに適しています。

Java ベースのアプリケーションの場合、OpenShift Web コンソールは、アプリケーションを実行している JVM によって関連するすべてのメトリクス出力を収集し、表示する統合 **hawt.io コンソール** を提供します。

前提条件

- **oc** クライアントが認証されている。
- OpenShift のプロジェクトで実行している Java ベースのアプリケーションコンテナー
- 最新の JDK 1.8.0 イメージ

手順

1. プロジェクト内で Pod のデプロイメント設定を一覧表示し、アプリケーションに対応する Pod のデプロイメント設定を選択します。

```
oc get dc
```

```
NAME        REVISION   DESIRED   CURRENT   TRIGGERED BY
MY_APP_NAME   2          1         1         config,image(my-app:6)
...
```

2. アプリケーションを実行する Pod の YAML デプロイメントテンプレートを開いて編集します。

```
oc edit dc/MY_APP_NAME
```

3. 以下のエントリーをテンプレートの **ports** セクションに追加し、変更を保存します。

```
...
spec:
  ...
  ports:
  - containerPort: 8778
    name: jolokia
    protocol: TCP
  ...
...
```

4. アプリケーションを実行する Pod を再デプロイします。

```
oc rollout latest dc/MY_APP_NAME
```

Pod は更新されたデプロイメント設定で再デプロイされ、ポート **8778** を公開します。

5. OpenShift Web コンソールにログインします。
6. サイドバーで、**Applications** > **Pods** に移動し、アプリケーションを実行する Pod の名前をクリックします。
7. Pod の詳細画面で **Open Java Console** をクリックし、hawt.io コンソールにアクセスします。

関連情報

- hawt.io ドキュメント

6.2. ECLIPSE VERT.X での PROMETHEUS を使用したアプリケーションメトリクスの公開

Prometheus は監視されたアプリケーションに接続してデータを収集します。アプリケーションはメトリクスをサーバーに送信しません。

前提条件

- Prometheus サーバーがクラスターで実行している。

手順

1. アプリケーションの **pom.xml** ファイルに **vertx-micrometer** 依存関係および **vertx-web** 依存関係を含めます。

pom.xml

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-micrometer-metrics</artifactId>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web</artifactId>
</dependency>
```

2. バージョン 3.5.4 以降、Prometheus のメトリクスを公開するには、カスタム **Launcher** クラスで Eclipse Vert.x オプションを設定する必要があります。
カスタム **Launcher** クラスの **beforeStartingVertx** メソッドおよび **afterStartingVertx** メソッドを上書きして、メトリクスエンジンを設定します。以下に例を示します。

CustomLauncher.java ファイルの例

```
package org.acme;

import io.micrometer.core.instrument.Meter;
import io.micrometer.core.instrument.config.MeterFilter;
```

```
import io.micrometer.core.instrument.distribution.DistributionStatisticConfig;
import io.micrometer.prometheus.PrometheusMeterRegistry;
import io.vertx.core.Vertx;
import io.vertx.core.VertxOptions;
import io.vertx.core.http.HttpServerOptions;
import io.vertx.micrometer.MicrometerMetricsOptions;
import io.vertx.micrometer.VertxPrometheusOptions;
import io.vertx.micrometer.backends.BackendRegistries;

public class CustomLauncher extends Launcher {

  @Override
  public void beforeStartingVertx(VertxOptions options) {
    options.setMetricsOptions(new MicrometerMetricsOptions()
      .setPrometheusOptions(new VertxPrometheusOptions().setEnabled(true)
        .setStartEmbeddedServer(true)
        .setEmbeddedServerOptions(new HttpServerOptions().setPort(8081))
        .setEmbeddedServerEndpoint("/metrics"))
      .setEnabled(true));
  }

  @Override
  public void afterStartingVertx(Vertx vertx) {
    PrometheusMeterRegistry registry = (PrometheusMeterRegistry)
BackendRegistries.getDefaultNow();
    registry.config().meterFilter(
      new MeterFilter() {
        @Override
        public DistributionStatisticConfig configure(Meter.Id id, DistributionStatisticConfig config)
{
          return DistributionStatisticConfig.builder()
            .percentilesHistogram(true)
            .build()
            .merge(config);
        }
    });
}
```

3. カスタムの **Verticle** クラスを作成し、メトリクスを収集するために **start** メソッドを上書きします。たとえば、**Timer** クラスを使用して実行時間を測定します。

CustomVertxApp.java ファイルの例

```
package org.acme;

import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.Timer;
import io.vertx.core.AbstractVerticle;
import io.vertx.core.Vertx;
import io.vertx.core.VertxOptions;
import io.vertx.core.http.HttpServerOptions;
import io.vertx.micrometer.backends.BackendRegistries;

public class CustomVertxApp extends AbstractVerticle {

  @Override
```

```
  public void start() {
    MeterRegistry registry = BackendRegistries.getDefaultNow();
    Timer timer = Timer
      .builder("my.timer")
      .description("a description of what this timer does")
      .register(registry);

    vertx.setPeriodic(1000, l -> {
      timer.record(() -> {

        // Do something

      });
    });
  }
}
```

4. アプリケーションの **pom.xml** ファイルの **<vertx.verticle>** プロパティーおよび **<vertx.launcher>** プロパティーを設定して、カスタムクラスを参照します。

```
<properties>
  ...
  <vertx.verticle>org.acme.CustomVertxApp</vertx.verticle>
  <vertx.launcher>org.acme.CustomLauncher</vertx.launcher>
  ...
</properties>
```

5. アプリケーションを起動します。

```
$ mvn vertx:run
```

6. トレースされたエンドポイントを複数回呼び出します。

```
$ curl http://localhost:8080/
Hello
```

7. コレクションが発生するまで 15 秒以上待機し、Prometheus UI でメトリクスを確認します。

    1. http://localhost:9090/ で Prometheus UI を開き、 **Expression** ボックスに **hello** と入力します。

    2. 提案から、たとえば **application:hello_count** を選択し、**Execute** をクリックします。

    3. 表示される表で、リソースメソッドが呼び出された回数を確認できます。

    4. **application:hello_time_mean_seconds** を選択して、すべての呼び出しの平均時間を確認します。

    作成したすべてのメトリクスの前には **application:** があることに注意してください。Eclipse MicroProfile Metrics 仕様が要求するように、Eclipse Vert.x によって自動的に公開される他のメトリックがあります。これらのメトリクスには **base:** および **vendor:** というプレフィックスが付けられ、アプリケーションが実行する JVM に関する情報を公開します。

関連情報

- Eclipse Vert.x で Micrometer メトリクスを使用する方法は、「Eclipse Vert.x} Micrometer Metrics」を参照してください。

第7章 ECLIPSE VERT.X のアプリケーションの例

Eclipse Vert.x ランタイムは、サンプルアプリケーションを提供します。OpenShift でアプリケーションの開発を開始すると、サンプルアプリケーションをテンプレートとして使用できます。

これらのサンプルアプリケーションは Developer Launcher でアクセスできます。

すべてのサンプルアプリケーションを以下にダウンロードおよびデプロイできます。

- x86_64 アーキテクチャー - 本ガイドのアプリケーションの例では、サンプルアプリケーションを x86_64 アーキテクチャーにビルドおよびデプロイする方法を説明します。
- s390x アーキテクチャー - IBM Z インフラストラクチャーでプロビジョニングされた OpenShift 環境にサンプルアプリケーションをデプロイするには、コマンドに関連する IBM Z イメージ名を指定します。
- ppc64le アーキテクチャー - IBM Power Systems インフラストラクチャーにプロビジョニングされた OpenShift 環境にアプリケーションのサンプルをデプロイするには、コマンドで関連する IBM Power Systems イメージ名を指定します。

イメージ名の詳細は、「Eclipse Vert.x でサポートされる Java イメージ」 セクションを参照してください。

サンプルアプリケーションの一部には、ワークフローを実証するために Red Hat Data Grid などの他の製品も必要になります。この場合は、これらの製品のイメージ名を、サンプルアプリケーションの YAML ファイルで関連する IBM Z および IBM Power System のイメージ名に変更する必要もあります。

7.1. ECLIPSE VERT.X の REST API LEVEL 0 サンプル

重要

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

REST API Level 0 サンプルでできること

REST API Level 0 サンプルでは、REST フレームワークを使用して、HTTP 経由でビジネスオペレーションをリモートプロシージャコールエンドポイントにマッピングする方法が示されています。これは、Richardson Maturity Model の Level 0 に対応します。REST およびその基本的な原則を使用して HTTP エンドポイントを作成すると、API を柔軟にプロトタイプおよび設計することができます。

この例では、HTTP プロトコルを使用してリモートサービスと対話するためのメカニズムが導入されました。これにより、以下が可能になります。

- **api/greeting** エンドポイントで HTTP **GET** 要求を実行します。
- **Hello, World!** で構成されるペイロードを使用して JSON 形式でレスポンスを受け取ります。文字列。
- String 引数を渡し、**api/greeting** エンドポイントで HTTP **GET** 要求を実行します。これにより、クエリー文字列に **name** 要求パラメーターが使用されます。
- **Hello, $name!** のペイロードを含む JSON 形式の応答を受信します。**$name** は、要求に渡された **name** パラメーターの値に置き換えられます。

7.1.1. REST API Level 0 設計トレードオフ

表7.1 設計トレードオフ

良い点	悪い点

• アプリケーション例では、高速なプロトタイプを有効にします。
• API Design は柔軟性があります。
• HTTP エンドポイントにより、クライアントは言語に依存しません。
	• アプリケーションまたはサービスが成熟するにつれて、REST API Level 0 のアプローチは適切にスケーリングされない可能性があります。クリーンな API 設計や、データベースの対話に関するユースケースをサポートしない場合があります。
	  ◦ 共有された変更可能な状態を含むすべての操作は、適切なバッキングデータストアと統合する必要があります。
	  ◦ この API 設計で処理されるすべての要求は、要求に対応するコンテナーにのみスコープ指定されます。これ以降の要求は、同じコンテナーで処理されない可能性があります。

7.1.2. REST API Level 0 サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で REST API Level 0 サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

7.1.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェースから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.1.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.1.2.3. CLI クライアント oc を使用した REST API Level 0 サンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                       READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa            1/1       Running     0          58s
MY_APP_NAME-s2i-1-build        0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.1.3. REST API Level 0 サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、REST API Level 0 サンプルアプリケーションを Minishift または CDK でローカルに実行します。

-
- CLI クライアント **oc** の使用

7.1.3.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。

2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

7.1.3.2.

前提条件

- 詳細は、「」を参照してください。

手順

1.

2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.1.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.1.3.4. CLI クライアント oc を使用した REST API Level 0 サンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

   このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                       READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa            1/1       Running     0          58s
MY_APP_NAME-s2i-1-build        0/1       Completed   0          2m
```

   MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

   ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                   PATH      SERVICES
```

```
PORT      TERMINATION
MY_APP_NAME        MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.1.4. REST API Level 0 サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「REST API Level 0 サンプルアプリケーションの OpenShift Online へのデプロイメント」 の手順に従って、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

7.1.5. Eclipse Vert.x の未変更の REST API Level 0 サンプルアプリケーションとの対話

この例では、GET 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。

前提条件

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

手順

1. **curl** を使用して、サンプルに **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{
  "content" : "Hello, World!"
}
```

2. **curl** を使用して、例に対して URL パラメーター **name** を付けて **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting?name=Sarah
{
  "content" : "Hello, Sarah!"
}
```

注記

ブラウザーから、例で提供されているフォームを使用して、これらの同じ対話を実行することもできます。フォームは、プロジェクト **http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** の root にあります。

7.1.6. REST API Level 0 のサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

7.1.7. REST リソース

REST の背景および関連情報は、以下を参照してください。

- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- Richardson Maturity Model
- JSR 311: JAX-RS: The JavaTM API for RESTful Web Services
- Some Rest with Eclipse Vert.x
- Spring Boot の REST API Level 0 の例

- Thorntail の REST API Level 0 の例
- REST API Level 0 for Node.js

7.2. ECLIPSE VERT.X の外部化設定の例

重要

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

外部化された設定は、ConfigMap を使用して設定を外部化する基本的な例を提供します。**ConfigMap** は、コンテナーを OpenShift に依存しないようにする一方で、単純なキーと値のペアとして設定データを 1 つ以上の Linux コンテナーに挿入するために OpenShift で使用されるオブジェクトです。

この例では、以下の方法を示しています。

- **ConfigMap** をセットアップし、設定します。
- アプリケーション内で **ConfigMap** によって提供される設定を使用します。
- 実行中のアプリケーションの **ConfigMap** 設定に変更をデプロイします。

7.2.1. 外部化された設定の設計パターン

可能な限り、アプリケーション設定を外部化し、アプリケーションコードから分離させます。これにより、異なる環境を通過する際にアプリケーション設定を変更できますが、コードは変更されません。構成を外部化すると、機密情報や内部情報がコードベースやバージョン管理から除外されます。多くの言語およびアプリケーションサーバーは、アプリケーション設定の外部化をサポートする環境変数を提供します。

マイクロサービスアーキテクチャーおよび多言語 (polyglot) 環境は、アプリケーションの設定を管理する複雑な層を追加します。アプリケーションは独立した分散サービスで構成され、それぞれ独自の設定を持つことができます。すべての設定データを同期し、アクセス可能な状態に維持すると、メンテナンスの課題が発生します。

ConfigMap により、アプリケーション設定を外部化でき、OpenShift 上の個々の Linux コンテナーおよび Pod で使用できます。YAML ファイルの使用を含むさまざまな方法で ConfigMap オブジェクトを作成し、これを Linux コンテナーに挿入できます。ConfigMap を使用すると、設定データのグループ化およびスケーリングが可能です。これにより、基本的な **開発**、**ステージ**、および **実稼働** 以外の多くの環境を設定できます。ConfigMap の詳細は、OpenShift ドキュメント を参照してください。

7.2.2. 外部化設定設計のトレードオフ

表7.2 設計のトレードオフ

良い点	悪い点
• 設定がデプロイメントとは分離される • 個別に更新可能 • サービス間で共有できる	• 環境への設定の追加には追加のステップが必要 • 別々に維持する必要がある • サービスのスコープを超える調整が必要

7.2.3. 外部化設定のサンプルアプリケーションの OpenShift Online へのデプロイメント

以下のいずれかのオプションを使用して、OpenShift Online で外部化設定アプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

7.2.3.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェースから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.2.3.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。

2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.2.3.3. CLI クライアント oc を使用した Externalized Configuration アプリケーションのデプロイメント

このセクションでは、外部化設定のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. サンプルアプリケーションをデプロイする前に、サービスアカウントへの view アクセス権を割り当て、ConfigMap の内容を読み取るためにアプリケーションが OpenShift API にアクセスできるようにします。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

4. アプリケーションの root ディレクトリーに移動します。
5. **app-config.yml** を使用して ConfigMap 設定を OpenShift にデプロイします。

```
$ oc create configmap app-config --from-file=app-config.yml
```

6. ConfigMap 設定がデプロイされていることを確認します。

```
$ oc get configmap app-config -o yaml

apiVersion: template.openshift.io/v1
data:
  app-config.yml: |-
      message : "Hello, %s from a ConfigMap !"
      level : INFO
...
```

7. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

8. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                              READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa               1/1       Running     0          58s
MY_APP_NAME-s2i-1-build           0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。また、続行する前に Pod が **準備状態** になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、 **MY_APP_NAME-1-aaaaa** は **READY** 列が **1/1** の場合に **準備状態** になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

9. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.2.4. 外部化設定アプリケーションの Minishift または CDK へのデプロイメント

以下のいずれかのオプションを使用して、Minishift または CDK で外部設定サンプルアプリケーションをローカルで実行します。

-
- CLI クライアント **oc** の使用

7.2.4.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

7.2.4.2.

前提条件

- 詳細は、「」を参照してください。

手順

1.
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.2.4.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。

2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.2.4.4. CLI クライアント oc を使用した Externalized Configuration アプリケーションのデプロイメント

このセクションでは、外部化設定のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. サンプルアプリケーションをデプロイする前に、サービスアカウントへの view アクセス権を割り当て、ConfigMap の内容を読み取るためにアプリケーションが OpenShift API にアクセスできるようにします。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

4. アプリケーションの root ディレクトリーに移動します。
5. **app-config.yml** を使用して ConfigMap 設定を OpenShift にデプロイします。

```
$ oc create configmap app-config --from-file=app-config.yml
```

6. ConfigMap 設定がデプロイされていることを確認します。

```
$ oc get configmap app-config -o yaml

apiVersion: template.openshift.io/v1
data:
  app-config.yml: |-
      message : "Hello, %s from a ConfigMap !"
      level : INFO
...
```

7. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

8. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                              READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa               1/1       Running     0          58s
MY_APP_NAME-s2i-1-build           0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。また、続行する前に Pod が **準備状態** になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、 **MY_APP_NAME-1-aaaaa** は **READY** 列が **1/1** の場合に **準備状態** になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

9. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.2.5. 外部設定サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「外部化設定のサンプルアプリケーションの OpenShift Online へのデプロイメント」の手順に従って、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

7.2.6. Eclipse Vert.x の未変更の外部化設定サンプルアプリケーションとの対話

この例では、GET 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。

前提条件

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

手順

1. **curl** を使用して、サンプルに **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World from a ConfigMap !"}
```

2. デプロイされた ConfigMap 設定を更新します。

```
$ oc edit configmap app-config
```

message キーの値を **Bonjour, %s from a ConfigMap !** に変更し、ファイルを保存します。

3. ConfigMap の更新は、アプリケーションの再起動を必要とせずに許容可能な時間 (数秒) 内でアプリケーションによって読み取る必要があります。

4. 更新された ConfigMap 構成を使用した例に対して **curl** を使用して **GET** 要求を実行し、更新されたグリーティングを確認します。また、アプリケーションが提供する Web フォームを使用して、ブラウザーから実行することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Bonjour, World from a ConfigMap !"}
```

7.2.7. 外部化設定のサンプルアプリケーションの統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。
- サンプルアプリケーションのサービスアカウントに割り当てられたアクセスパーミッションを表示します。これにより、アプリケーションは ConfigMap から設定を読み取ることができます。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

7.2.8. 外部化設定リソース

外部化設定および ConfigMap の背景および関連情報は、以下を参照してください。

- OpenShift ConfigMap ドキュメント
- ブログ投稿 OpenShift での ConfigMap に関する記事
- Eclipse Vert.x を使用した外部化設定
- Spring Boot の外部化設定の例
- Externalized Configuration for Thorntail
- Externalized Configuration for Node.js

7.3. ECLIPSE VERT.X のリレーショナルデータベースバックエンドのサンプル

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: Foundational

リレーショナルデータベースバックエンドのサンプル

リレーショナルデータベースバックエンドのサンプルは、REST API Level 0 アプリケーションを拡張して、単純な HTTP API を使用して PostgreSQL データベースで **作成**、**読み取り**、**更新**、**削除** (**CRUD**) 操作を実行する基本的な例を提供します。**CRUD** 操作は永続ストレージの 4 つの基本的な機能であり、データベースを処理する HTTP API の開発時に広く使用されます。

また、この例では、HTTP アプリケーションが OpenShift のデータベースを特定し、接続する機能も示しています。各ランタイムは、指定のケースで最も適した接続ソリューションを実装する方法を示しています。ランタイムは、API の **JDBC**、**JPA** などを使用、または **ORM** に直接アクセスするなどのオプションを選択できます。

アプリケーションのサンプルは HTTP API を公開し、HTTP で **CRUD** 操作を実行してデータを操作できるようにするエンドポイントを提供します。**CRUD** 操作は HTTP **Verbs** にマップされます。API は JSON フォーマットを使用して要求を受け取りし、ユーザーに応答を返します。また、ユーザーは、サンプルが提供するユーザーインターフェースを使用して、アプリケーションを使用することもできます。具体的には、この例では以下を可能にするアプリケーションを提供します。

- ブラウザーでアプリケーション Web インターフェースに移動します。これにより、**my_data** データベースのデータで **CRUD** 操作を実行する簡単な Web サイトが公開されます。
- **api/fruits** エンドポイントで HTTP **GET** 要求を実行します。
- データベース内のすべての fruits のリストが含まれる JSON 配列としてフォーマットされたレスポンスを受け取ります。
- 有効なアイテム ID を引数として渡しながら、**api/fruits/*** エンドポイントで HTTP **GET** 要求を実行します。
- 指定した ID を持つ fruit の名前が含まれる JSON 形式で応答を受け取ります。指定された ID に項目がない場合は、呼び出しにより HTTP エラー 404 が発生します。
- **api/fruits** エンドポイントで HTTP **POST** 要求を実行し、有効な **name** 値を渡してデータベースの新規エントリーを作成します。
- 有効な ID と名前を引数として渡して、**api/fruits/*** エンドポイントで HTTP **PUT** 要求を実行します。これにより、要求に指定された名前に一致するように、指定の ID を持つ項目の名前が更新されます。
- **api/fruits/*** エンドポイントで HTTP **DELETE** 要求を実行し、有効な ID を引数として渡します。これにより、指定された ID の項目がデータベースから削除され、応答として HTTP コード **204** (No Content) を返します。無効な ID を渡すと、呼び出しにより HTTP エラー **404** が発生します。

この例には、アプリケーションがデータベースと完全に統合されていることを検証するために使用できる自動化された 統合テスト のセットも含まれています。

この例では、完全に成熟した RESTful モデル (レベル3) を紹介していませんが、推奨される HTTP API プラクティスに従って、互換性のある HTTP 動詞とステータスを使用しています。

7.3.1. リレーショナルデータベースバックエンドの設計トレードオフ

表7.3 設計のトレードオフ

良い点	悪い点
• 各ランタイムは、データベースの対話の実装方法を決定します。1つは JDBC などの低レベルの接続 API を使用し、他の接続では JPA を使用できますが、別の接続は ORM API に直接アクセスできます。各ランタイムは、最適な方法を決定します。 • 各ランタイムはスキーマの作成方法を決定します。	• このサンプルアプリケーションで提供される PostgreSQL データベースは永続ストレージではバックアップされていません。データベース Pod を停止または再デプロイすると、データベースへの変更は失われます。変更を保持するために、サンプルアプリケーションの Pod で外部データベースを使用するには、OpenShift ドキュメントの「Creating an application with a database」を参照してください。また、OpenShift 上のデータベースコンテナーで永続ストレージを設定することもできます。OpenShift およびコンテナーで永続ストレージを使用する方法は、OpenShift ドキュメントの「Persistent Storage」、「Managing Volumes」および「Persistent Volumes」の章を参照してください。

7.3.2. リレーショナルデータベースバックエンドのサンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online でリレーショナルデータベースバックエンドアプリケーションのサンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

7.3.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェースから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.3.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.3.2.3. CLI クライアント oc を使用したリレーショナルデータベースバックエンドのサンプルアプリケーションのデプロイメント

このセクションでは、リレーショナルデータベースバックエンドのサンプルアプリケーションを構築し、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. PostgreSQL データベースを OpenShift にデプロイします。データベースアプリケーションの作成時に、ユーザー名、パスワード、およびデータベース名に以下の値を使用するようにしてください。これらの値を使用するサンプルアプリケーションが事前設定されています。異なる値を使用すると、アプリケーションがデータベースと統合できなくなります。

```
$ oc new-app -e POSTGRESQL_USER=luke -ePOSTGRESQL_PASSWORD=secret -
ePOSTGRESQL_DATABASE=my_data registry.access.redhat.com/rhscl/postgresql-10-rhel7
--name=my-database
```

5. データベースのステータスを確認し、Pod が実行中であることを確認します。

```
$ oc get pods -w
my-database-1-aaaaa   1/1       Running   0         45s
my-database-1-deploy   0/1       Completed   0         53s
```

my-database-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。

8. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                   PATH      SERVICES          PORT
TERMINATION
MY_APP_NAME   MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME   8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.3.3. リレーショナルデータベースバックエンドのサンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK でローカルでリレーショナルデータベースバックエンドのサンプルアプリケーションを実行します。

-
- CLI クライアント **oc** の使用

7.3.3.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

7.3.3.2.

前提条件

- 詳細は、「」を参照してください。

手順

1.
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.3.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.3.3.4. CLI クライアント oc を使用したリレーショナルデータベースバックエンドのサンプルアプリケーションのデプロイメント

このセクションでは、リレーショナルデータベースバックエンドのサンプルアプリケーションを構築し、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. PostgreSQL データベースを OpenShift にデプロイします。データベースアプリケーションの作成時に、ユーザー名、パスワード、およびデータベース名に以下の値を使用するようにしてください。これらの値を使用するサンプルアプリケーションが事前設定されています。異なる値を使用すると、アプリケーションがデータベースと統合できなくなります。

```
$ oc new-app -e POSTGRESQL_USER=luke -ePOSTGRESQL_PASSWORD=secret -
ePOSTGRESQL_DATABASE=my_data registry.access.redhat.com/rhscl/postgresql-10-rhel7
--name=my-database
```

5. データベースのステータスを確認し、Pod が実行中であることを確認します。

```
$ oc get pods -w
my-database-1-aaaaa   1/1       Running   0         45s
my-database-1-deploy   0/1       Completed   0         53s
```

my-database-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。

8. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                PATH      SERVICES          PORT
TERMINATION
MY_APP_NAME   MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME   8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.3.4. リレーショナルデータベースバックエンドのサンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「リレーショナルデータベースバックエンドのサンプルアプリケーションの OpenShift Online へのデプロイメント」の手順に従って、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

7.3.5. リレーショナルデータベースバックエンド API との対話

サンプルアプリケーションの作成が完了したら、以下のように対話できます。

前提条件

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

手順

1. 以下のコマンドを実行して、アプリケーションの URL を取得します。

```
$ oc get route MY_APP_NAME
```

```
NAME            HOST/PORT                                        PATH      SERVICES        PORT      TERMINATION
MY_APP_NAME     MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME             MY_APP_NAME     8080
```

2. データベースアプリケーションの Web インターフェースにアクセスするには、ブラウザーで **アプリケーション URL** に移動します。

```
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
```

また、**curl** を使用して **api/fruits/*** エンドポイントで要求を直接作成できます。

データベースのエントリーの一覧表示

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits
```

```
[ {
  "id" : 1,
  "name" : "Apple",
  "stock" : 10
}, {
  "id" : 2,
  "name" : "Orange",
  "stock" : 10
}, {
  "id" : 3,
  "name" : "Pear",
  "stock" : 10
} ]
```

特定の ID のあるエントリーの取得

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/3
```

```
{
  "id" : 3,
  "name" : "Pear",
  "stock" : 10
}
```

エントリーの新規作成

```
$ curl -H "Content-Type: application/json" -X POST -d '{"name":"Peach","stock":1}'
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits
```

```
{
  "id" : 4,
  "name" : "Peach",
  "stock" : 1
}
```

エントリーの更新

```
$ curl -H "Content-Type: application/json" -X PUT -d '{"name":"Apple","stock":100}'
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/1
```

```
{
  "id" : 1,
  "name" : "Apple",
  "stock" : 100
}
```

エントリーの削除

```
$ curl -X DELETE http://MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/1
```

トラブルシューティング

- これらのコマンドを実行後に HTTP エラーコード **503** を応答として受け取った場合は、アプリケーションが準備状態にないことを意味します。

7.3.6. リレーショナルデータベースバックエンドのサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

7.3.7. リレーショナルデータベースリソース

OpenShift、CRUD、HTTP API、および REST でのリレーショナルデータベースの実行に関する背景および関連情報は、以下を参照してください。

- HTTP Verbs
- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- REST API 設計のディベースが終了しない
- REST APIs must be Hypertext driven
- Richardson Maturity Model
- JSR 311: JAX-RS: The JavaTM API for RESTful Web Services
- Some Rest with Eclipse Vert.x
- Using the Eclipse Vert.x asynchronous SQL client
- Spring Boot のリレーショナルデータベースバックエンドの例
- Thorntail のリレーショナルデータベースバックエンドの例
- Relational Database Backend for Node.js

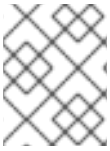
7.4. ECLIPSE VERT.X のヘルスチェックの例

重要

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: Foundational

アプリケーションをデプロイする場合は、アプリケーションが利用可能かどうかを確認し、受信した要求の処理を開始することが重要です。**ヘルスチェック** パターンを実装すると、アプリケーションが利用できるかどうかや要求に対応できるかどうかなど、アプリケーションの健全性を監視できます。



注記

先に進む前に、「ヘルスチェックの概念」セクションでヘルスチェックという用語の説明を参照してください。

このユースケースの目的は、プローブを使用してヘルスチェックパターンを実証することです。プロービングは、アプリケーションの Liveness および Readiness を報告するために使用されます。このユースケースでは、HTTP **health** エンドポイントを公開して HTTP 要求を発行するアプリケーションを設定します。コンテナーが動作している場合は、HTTP エンドポイント **health** の Liveness プローブによると、管理プラットフォームは **200** を戻りコードとして受け取り、それ以上のアクションは必要ありません。HTTP エンドポイント **health** が応答を返さない場合、たとえばスレッドがブロックされている場合は、Liveness プローブにより、アプリケーションが稼働しているとは見なされません。この場合、プラットフォームはそのアプリケーションに対応する Pod を強制終了し、アプリケーションを再起動するために新規 Pod を再作成します。

このユースケースでは、Readiness プローブを実証し、使用することもできます。アプリケーションが実行していても要求を処理できない場合 (再起動中にアプリケーションが HTTP 応答コード **503** を返す場合など)、このアプリケーションは Readiness プローブにより準備ができていないと見なされます。Readiness プローブによってアプリケーションが準備状態にあるとみなされないと、要求は Readiness プローブにより準備が整っているとみなされるまで、要求はそのアプリケーションにルーティングされません。

7.4.1. ヘルスチェックの概念

ヘルスチェックパターンを理解するには、まず以下の概念を理解する必要があります。

Liveness

Liveness は、アプリケーションが実行しているかどうかを定義します。実行中のアプリケーションが応答しない状態または停止状態に移行する可能性があるため、再起動する必要があります。Liveness の確認は、アプリケーションを再起動する必要があるかどうかを判断するのに役立ちます。

Readines

Readiness は、実行中のアプリケーションが要求を処理できるかどうかを定義します。実行中のアプリケーションがエラーまたは破損状態に切り替わり、要求にサービスを提供することができません。Readiness を確認すると、要求が引き続きそのアプリケーションにルーティングされるべきかどうかが判断されます。

フェイルオーバー

フェイルオーバーにより、サービス要求の失敗が適切に処理されるようになります。アプリケーションが要求のサービスに失敗した場合、その要求と今後の要求は **フェイルオーバー** または他のアプリケーションにルーティングできます (通常、同じアプリケーションの冗長コピーです)。

耐障害性および安定性

耐障害性および安定性により、要求処理の失敗を適切に処理できます。接続の損失によりアプリケーションが要求を処理できない場合、耐久性のあるシステムでは、接続が再確立された後にその要求を再試行できます。

プローブ

プローブは実行中のコンテナーで定期的に実行する Kubernetes の動作です。

7.4.2. Health Check サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Health Check のサンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

7.4.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェースから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.4.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。

4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.4.2.3. CLI クライアント oc を使用した Health Check サンプルアプリケーションのデプロイメント

このセクションでは、Health Check のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

   このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                    READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa           1/1       Running     0          58s
MY_APP_NAME-s2i-1-build       0/1       Completed   0          2m
```

   MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。また、続行する前に Pod が準備状態になるまで待機する必要があります。こ

れは **READY** 列に表示されます。たとえば、**MY_APP_NAME-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.4.3. Health Check のサンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK で Health Check サンプルアプリケーションをローカルで実行します。

-
- CLI クライアント **oc** の使用

7.4.3.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer
```

```
To login as administrator:
    oc login -u system:admin
```

7.4.3.2.

前提条件

- 詳細は、「」を参照してください。

手順

1.

2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.4.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.4.3.4. CLI クライアント oc を使用した Health Check サンプルアプリケーションのデプロイメント

このセクションでは、Health Check のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-

- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa             1/1       Running     0          58s
MY_APP_NAME-s2i-1-build         0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.4.4. Health Check サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「Health Check サンプルアプリケーションの OpenShift Online へのデプロイメント」 の手順に従って、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

7.4.5. 未変更の Health Check サンプルアプリケーションとの対話

サンプルアプリケーションをデプロイすると、**MY_APP_NAME** サービスが実行します。**MY_APP_NAME** サービスは、以下の REST エンドポイントを公開します。

/api/greeting
: **name** パラメーター (または World をデフォルト値として) のグリーティングが含まれる JSON を返します。

/api/stop
: 障害をシミュレーションする手段として、サービスが強制的に応答しなくなります。

以下の手順は、サービスの可用性を確認し、障害をシミュレートする方法を示しています。この利用可能なサービスの障害により、そのサービスで OpenShift の自己修復機能が実行します。

Web インターフェースを使用して、これらの手順を実施することができます。

1. **curl** を使用して、**MY_APP_NAME** サービスに対して **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
```

```
{"content":"Hello, World!"}
```

2. **/api/stop** エンドポイントを呼び出して、その直後に **/api/greeting** エンドポイントの可用性を確認します。
/api/stop エンドポイントを呼び出すと、内部サービス障害をシミュレートし、OpenShift の自己修復機能をトリガーします。障害のシミュレート後に **/api/greeting** を呼び出すと、サービスは HTTP ステータス **503** を返すはずです。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/stop
```

```
Stopping HTTP server, Bye bye world !
```

(続いて、以下を行います)

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
```

```
Not online
```

3. **oc get pods -w** を使用して、動作中の自己修復機能を継続的に監視します。
サービス障害の呼び出し中に、OpenShift コンソールまたは **oc** クライアントツールで機能する自己修復機能を確認できます。**READY** 状態の Pod 数がゼロ (**0**/**1**) になり、その後短時間 (1 分未満) で 1 (**1**/**1**) に戻ることが確認できるはずです。さらに、サービスの失敗を呼び出すたびに **RESTARTS** カウントが増加します。

```
$ oc get pods -w
NAME                    READY     STATUS    RESTARTS   AGE
MY_APP_NAME-1-26iy7   0/1       Running   5          18m
MY_APP_NAME-1-26iy7   1/1       Running   5         19m
```

4. 必要に応じて、Web インターフェースを使用してサービスを呼び出します。
端末ウィンドウを使用して対話を行うと、サービスによって提供される Web インターフェースを使用して、異なるメソッドを起動し、サービスがライフサイクルフェーズを通過するのを監視できます。

```
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
```

5. 必要に応じて、Web コンソールを使用して、自己修復プロセスの各段階でアプリケーションによって生成されたログ出力を表示します。
    1. プロジェクトに移動します。
    2. サイドバーで **Monitoring** をクリックします。
    3. 画面右上にある **Events** をクリックし、ログメッセージを表示します。
    4. 必要に応じて、**View Details** をクリックし、Event ログの詳細なビューを表示します。

    ヘルスチェックアプリケーションは以下のメッセージを生成します。

メッセージ	Status
Unhealthy	readiness プローブが失敗しました。このメッセージは予想され、**/api/greeting** エンドポイントのシミュレートされた失敗が検出され、自己修復プロセスが開始することを示します。
Killing	サービスを実行している利用不可の Docker コンテナーは、再作成前に強制終了されています。
Pulling	最新バージョンの Docker イメージをダウンロードして、コンテナーを再作成します。
Pulled	Docker イメージが正常にダウンロードされました。
Created	Docker コンテナーが正常に作成されました。

メッセージ	Status
Started	Docker コンテナーが要求を処理する準備が整っている。

7.4.6. Health Check のサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

7.4.7. ヘルスチェックリソース

ヘルスチェックの背景および関連情報は、以下を参照してください。

- OpenShift のアプリケーションの正常性
- Kubernetes Liveness and Readiness Probes
- Spring Boot のヘルスチェックの例
- Thorntail のヘルスチェックの例

- Health Check for Node.js

7.5. ECLIPSE VERT.X の CIRCUIT BREAKER の例

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Foundational**

Circuit Breaker 例は、サービスの障害を報告し、要求の処理に使用できるまで、失敗したサービスへのアクセスを制限する一般的なパターンを示しています。これにより、障害が発生したサービスの機能に依存する他のサービスでのカスケード障害を防ぐことができます。

この例では、サービスに Circuit Breaker および Fallback パターンを実装する方法を表しています。

7.5.1. サーキットブレーカー設計パターン

Circuit Breaker は、次のことも目的としたパターンです。

- ネットワーク障害の影響を低減し、サービスが他のサービスを同期的に呼び出すサービスアーキテクチャーにおける高いレイテンシーを削減します。
  いずれかのサービスが以下の状態では:
  - ネットワーク障害により利用できない
  - 圧倒的なトラフィックが原因で、レイテンシーが異常に高くなる

  エンドポイントを呼び出そうとする他のサービスは、エンドポイントに到達しようとして重要なリソースを使い果たし、使用できなくなる可能性があります。
- マイクロサービスアーキテクチャー全体が使用できなくなる可能性があるカスケード障害とも呼ばれる状態を防止します。
- 障害を監視する保護機能とリモート機能の間のプロキシーとして機能します。
- 障害が特定のしきい値に達すると作動し、サーキットブレーカーへのそれ以降のすべての呼び出しは、保護された呼び出しがまったく行われずに、エラーまたは事前定義されたフォールバック応答を返します。

Circuit Breaker には通常、Circuit Breaker が作動した時に通知するエラー報告メカニズムも含まれます。

Circuit Breaker の実装

- Circuit Breaker パターンが実装されると、サービスクライアントは一定間隔でプロキシー経由でリモートサービスエンドポイントを呼び出します。
- リモートサービスエンドポイントへの呼び出しが繰り返し一貫して失敗すると、Circuit Breaker が作動し、そのサービスへのすべての呼び出しが、設定されたタイムアウト期間中に即座に失敗し、事前定義されたフォールバック応答を返します。

- タイムアウト期間が終了すると、限られた数のテストコールがリモートサービスを通過し、リモートサービスが回復したか、使用できないままであるかを判断できます。
  - テストの呼び出しに失敗した場合、Circuit Breaker はサービスを利用できない状態を保ち、受信呼び出しへのフォールバック応答を返し続けます。
  - テストに成功すると、Circuit Breaker が閉じ、トラフィックが再度リモートサービスに到達できるようにします。

7.5.2. Circuit Breaker 設計のトレードオフ

表7.4 設計のトレードオフ

良い点	悪い点

• サービスが、自身が呼び出す他のサービスの障害を処理できるようになる。	• タイムアウト値の最適化が困難な場合がある。
	  ◦ タイムアウト値を必要以上に大きくすると、レイテンシーが過度に生成される可能性があります。
	  ◦ タイムアウトを必要な値より小さくすると、誤検知が発生する可能性があります。

7.5.3. Circuit Breaker サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Circuit Breaker サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

7.5.3.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェースから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。

2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.5.3.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.5.3.3. CLI クライアント oc を使用した Circuit Breaker サンプルアプリケーションのデプロイメント

このセクションでは、Circuit Breaker サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-greeting-1-aaaaa     1/1       Running   0          17s
MY_APP_NAME-greeting-1-deploy    0/1       Completed 0          22s
MY_APP_NAME-name-1-aaaaa         1/1       Running   0          14s
MY_APP_NAME-name-1-deploy        0/1       Completed 0          28s
```

MY_APP_NAME-greeting-1-aaaaa Pod および **MY_APP_NAME-name-1-aaaaa** Pod の両方には、完全にデプロイおよび起動すると **Running** のステータスが必要になります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-greeting-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name       MY_APP_NAME-name-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-name       8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.5.4. Circuit Breaker サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のいずれかのオプションを使用して、Minishift または CDK で Circuit Breaker サンプルアプリケーションをローカルで実行します。

-
- CLI クライアント **oc** の使用

7.5.4.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

7.5.4.2.

前提条件

- 詳細は、「」を参照してください。

手順

1.
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.5.4.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。

2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.5.4.4. CLI クライアント oc を使用した Circuit Breaker サンプルアプリケーションのデプロイメント

このセクションでは、Circuit Breaker サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

   このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-greeting-1-aaaaa     1/1       Running   0          17s
```

```
MY_APP_NAME-greeting-1-deploy    0/1       Completed 0          22s
MY_APP_NAME-name-1-aaaaa         1/1       Running   0          14s
MY_APP_NAME-name-1-deploy        0/1       Completed 0          28s
```

MY_APP_NAME-greeting-1-aaaaa Pod および **MY_APP_NAME-name-1-aaaaa** Pod の両方には、完全にデプロイおよび起動すると **Running** のステータスが必要になります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-greeting-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name       MY_APP_NAME-name-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-name       8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

7.5.5. Circuit Breaker サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「Circuit Breaker サンプルアプリケーションの OpenShift Online へのデプロイメント」 の手順に従って、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

7.5.6. 未変更の Eclipse Vert.x Circuit Breaker サンプルアプリケーションとの対話

Eclipse Vert.x のサンプルアプリケーションをデプロイした後に、以下のサービスが実行されます。

MY_APP_NAME-name

以下のエンドポイントを公開します。

- このサービスの稼働時に名前を返す **/api/name** エンドポイントと、このサービスが失敗を実証するように設定されたときにエラーが返されます。

- **/api/state** エンドポイント: **/api/name** エンドポイントの動作を制御し、サービスが正しく機能するか、または失敗を実証するかどうかを判断します。

MY_APP_NAME-greeting

以下のエンドポイントを公開します。

- パーソナライズされたグリーティング応答を取得するために呼び出す **/api/greeting** エンドポイント。
  /api/greeting エンドポイントを呼び出すと、要求の処理の一環として、**MY_APP_NAME-name** サービスの **/api/name** エンドポイントに対して呼び出しを実行します。**/api/name** エンドポイントに対する呼び出しは、Circuit Breaker によって保護されます。

  リモートエンドポイントが利用可能な場合、**name** サービスは HTTP コード **200** (**OK**) で応答し、**/api/greeting** エンドポイントから以下の応答を受け取ります。

  ```
  {"content":"Hello, World!"}
  ```

  リモートエンドポイントが利用できない場合、**name** サービスは HTTP コード **500** (**Internal server error**) で応答し、**/api/greeting** エンドポイントから事前定義されたフォールバック応答を受け取ります。

  ```
  {"content":"Hello, Fallback!"}
  ```

- Circuit Breaker の状態を返す **/api/cb-state** エンドポイント。状態は次のいずれかです。
  - open: サーキットブレーカーにより、失敗したサービスに要求が到達できないようになります。
  - closed: サーキットブレーカーにより、要求がサービスに到達できるようになります。
  - half-open: サーキットブレーカーにより、要求がサービスに到達できるようになります。要求に成功すると、サービスの状態はクローズにリセットされます。要求に失敗すると、タイマーが再起動します。

以下の手順は、サービスの可用性を確認し、障害をシミュレートしてフォールバック応答を受け取る方法を示しています。

1. **curl** を使用して、**MY_APP_NAME-greeting** サービスに対して **GET** 要求を実行します。これを行うには、Web インターフェースの **Invoke** ボタンを使用することもできます。

   ```
   $ curl http://MY_APP_NAME-greeting-
   MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
   {"content":"Hello, World!"}
   ```

2. **MY_APP_NAME-name** サービスの失敗をシミュレートするには、以下を行います。
   - Web インターフェースで **トグル** ボタンを使用します。
   - **MY_APP_NAME-name** サービスを実行している Pod のレプリカ数を 0 にスケーリングします。
   - **MY_APP_NAME-name** サービスの **/api/state** エンドポイントに対して HTTP **PUT** 要求を実行し、その状態を **fail** に設定します。

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"state": "fail"}'
http://MY_APP_NAME-name-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/state
```

3. **/api/greeting** エンドポイントを呼び出します。**/api/name** エンドポイントで複数の要求が失敗する場合:

   a. Circuit Breaker が開きます。

   b. Web インターフェースの状態インジケーターが **CLOSED** から **OPEN** に変わります。

   c. **/api/greeting** エンドポイントを呼び出すと、Circuit Breaker はフォールバック応答を実行します。

```
$ curl http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, Fallback!"}
```

4. 名前 **MY_APP_NAME-name** サービスを、可用性に戻します。これを行うには、以下を行います。

   - Web インターフェースで **トグル** ボタンを使用します。
   - **MY_APP_NAME-name** サービスを実行する Pod のレプリカ数を 1 にスケーリングします。
   - **MY_APP_NAME-name** サービスの **/api/state** エンドポイントに対して HTTP **PUT** 要求を実行し、その状態を **OK** に戻します。

```
$ curl -X PUT -H "Content-Type: application/json" -d '{"state": "ok"}'
http://MY_APP_NAME-name-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/state
```

5. **/api/greeting** エンドポイントを再度呼び出します。**/api/name** エンドポイントでの複数の要求が正常に実行される場合:

   a. Circuit Breaker が閉じます。

   b. Web インターフェースの状態インジケーターが **OPEN** から **CLOSED** に変わります。

   c. Circuit Breaker は、**/api/greeting** エンドポイントを呼び出す際に **Hello World!** グリーティングを返します。

```
$ curl http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World!"}
```

7.5.7. Circuit Breaker サンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。

- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

7.5.8. Hystrix Dashboard を使用したサーキットブレーカーの監視

Histrix Dashboard を使用すると、イベントストリームから Histrix メトリクスデータを集計し、それを 1 つの画面で表示することで、サービスの正常性を簡単にリアルタイムで監視できます。

前提条件

- アプリケーションがデプロイされている。

手順

1. Minishift または CDK クラスターにログインします。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

2. Web コンソールにアクセスするには、ブラウザーを使用して Minishift または CDK URL に移動します。
3. Circuit Breaker アプリケーションが含まれるプロジェクトに移動します。

```
$ oc project MY_PROJECT_NAME
```

4. Hystrix Dashboard アプリケーションの YAML テンプレート をインポートします。そのためには、**Add to Project** をクリックしてから **Import YAML / JSON** タブを選択し、YAML ファイルの内容をテキストボックスにコピーします。または、次のコマンドを実行できます。

```
$ oc create -f https://raw.githubusercontent.com/snowdrop/openshift-
templates/master/hystrix-dashboard/hystrix-dashboard.yml
```

5. **Create** ボタンをクリックして、テンプレートに基づいて Histrix Dashboard アプリケーションを作成します。または、次のコマンドを実行できます。

```
$ oc new-app --template=hystrix-dashboard
```

6. Hystrix Dashboard が含まれる Pod がデプロイされるまで待機します。

7. Hystrix Dashboard アプリケーションのルートを取得します。

```
$ oc get route hystrix-dashboard
NAME                HOST/PORT                                                   PATH      SERVICES
PORT      TERMINATION   WILDCARD
hystrix-dashboard   hystrix-dashboard-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME                hystrix-dashboard
<all>                 None
```

8. Dashboard にアクセスするには、ブラウザーで Dashboard アプリケーションルート URL を開きます。Web コンソールの **Overview** 画面に移動し、Hystrix Dashboard アプリケーションが含まれる Pod の上にあるヘッダーのルート URL をクリックします。

9. Dashboard を使用して **MY_APP_NAME-greeting** サービスを監視するには、デフォルトのイベントストリームアドレスを以下のアドレスに置き換え、**Monitor Stream** ボタンをクリックします。

```
http://MY_APP_NAME-greeting/hystrix.stream
```

関連情報

- Histrix Dashboard wiki ページ

7.5.9. サーキットブレーカーリソース

Circuit Breaker パターンの背後にある設計原則に関する背景情報については、以下のリンクを参照してください。

- microservices.io: Microservice Patterns: Circuit Breaker
- Martin Fowler: CircuitBreaker
- Spring Boot のサーキットブレーカーの例
- Circuit Breaker for Node.js
- Circuit Breaker for Thorntail

7.6. ECLIPSE VERT.X の SECURED サンプルアプリケーション

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

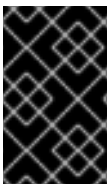
上達度レベルの例: **Advanced**

Secured サンプルアプリケーションは、Red Hat SSO を使用して REST エンドポイントを保護します。この例では、REST API Level 0 の例に展開されます。

Red Hat SSO:

- OAuth 2.0 仕様のエクステンションである Open ID Connect プロトコルを実装します。
- アクセストークンを発行し、セキュアなリソースに対するさまざまなアクセス権限をクライアントに提供します。

SSO でアプリケーションをセキュアにすると、セキュリティー設定を一元化しつつ、アプリケーションにセキュリティーを追加することができます。



重要

この例では、デモ目的で Red Hat SSO が事前設定されたので、その原則、使用方法、または設定を説明しません。この例を使用する前に、Red Hat SSO に関する基本的な概念を理解していることを確認してください。

7.6.1. Secured プロジェクト構造

SSO のサンプルには以下が含まれます。

- 保護するつもりである Greeting サービスのソース
- SSO サーバーをデプロイするテンプレートファイル (**service.sso.yaml**)
- サービスのセキュリティーを保護するように Keycloak アダプターの設定

7.6.2. Red Hat SSO デプロイメントの設定

この例の **service.sso.yaml** ファイルには、事前設定された Red Hat SSO サーバーをデプロイするすべての OpenShift 設定項目が含まれます。SSO サーバーの構成は、このサンプルのために簡略化されており、ユーザーとセキュリティー設定が事前に構成された、すぐに使用できる構成を提供します。**service.sso.yaml** ファイルには非常に長い行が含まれ、gedit などの一部のテキストエディターでは、このファイルの読み取りに問題がある場合があります。

警告

実稼働環境では、この SSO 設定を使用することは推奨されません。具体的には、セキュリティー設定の例に追加された単純化は、実稼働環境での使用に影響を及ぼします。

表7.5 SSO サンプルの簡略化

変更	理由	推奨事項
デフォルト設定には、**yaml 設定ファイルに公開鍵と秘密鍵** の両方が含まれます。	これを行ったのは、エンドユーザーが Red Hat SSO モジュールをデプロイして、内部や Red Hat SSO の構成方法を知らなくても使用可能な状態にすることができるためです。	実稼働環境では、秘密鍵をソース制御に保存しないでください。サーバー管理者が追加する必要があります。
設定済みの **クライアントがコールバック URL を受け入れます。**	各ランタイムにカスタム設定を使用しないように、OAuth2 仕様で必要なコールバックの検証を回避します。	アプリケーション固有のコールバック URL には、有効なドメイン名を指定する必要があります。
クライアントには SSL/TLS が必要ありません。また、セキュアなアプリケーションは HTTPS 上で公開されません。	この例は、ランタイムごとに証明書を生成する必要がないことで単純化されます。	実稼働環境では、セキュアなアプリケーションは単純な HTTP ではなく HTTPS を使用する必要があります。
トークンのタイムアウトがデフォルトの 1 分から 10 分に増えました。	コマンドラインの例を使用した場合のユーザーエクスペリエンスを向上します。	セキュリティーの観点から、攻撃者はアクセストークンが拡張されていると推測する必要があるウィンドウ。潜在的な攻撃者が現在のトークンを推測するのがより困難になるため、このウィンドウを短くすることが推奨されます。

7.6.3. Red Hat SSO レルムモデル

master レルムは、この例のセキュリティーを保護するために使用されます。コマンドラインクライアントとセキュアな REST エンドポイントのモデルを提供する事前設定されたアプリケーションクライアント定義は 2 つあります。

また、Red Hat SSO の **master** レルムには、**admin** および **alice** のさまざまな認証および認可の結果の検証に使用できる 2 つの事前設定されたユーザーも存在します。

7.6.3.1. Red Hat SSO ユーザー

セキュリティーが保護された例のレルムモデルには、ユーザーが 2 つ含まれます。

admin

admin ユーザーのパスワードは **admin** で、レルム管理者です。このユーザーは Red Hat SSO 管理コンソールに完全アクセスできますが、セキュアなエンドポイントへのアクセスに必要なロールマッピングはありません。このユーザーを使用して、認証されていて認可されていないユーザーの動作を説明できます。

alice

alice ユーザーには **password** のパスワードがあり、正規のアプリケーションユーザーです。このユーザーは、セキュアなエンドポイントへの認証および認可が成功したアクセスを実証します。ロールマッピングの例は、デコードされた JWT ベアラートークンにあります。

```
{
  "jti": "0073cfaa-7ed6-4326-ac07-c108d34b4f82",
```

```
  "exp": 1510162193,
  "nbf": 0,
  "iat": 1510161593,
  "iss": "https://secure-sso-sso.LOCAL_OPENSHIFT_HOSTNAME/auth/realms/master", (1)
  "aud": "demoapp",
  "sub": "c0175ccb-0892-4b31-829f-dda873815fe8",
  "typ": "Bearer",
  "azp": "demoapp",
  "nonce": "90ff5d1a-ba44-45ae-a413-50b08bf4a242",
  "auth_time": 1510161591,
  "session_state": "98efb95a-b355-43d1-996b-0abcb1304352",
  "acr": "1",
  "client_session": "5962112c-2b19-461e-8aac-84ab512d2a01",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [ (2)
      "example-admin"
    ]
  },
  "resource_access": { (3)
    "secured-example-endpoint": {
      "roles": [
        "example-admin" (4)
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "view-profile"
      ]
    }
  },
  "name": "Alice InChains",
  "preferred_username": "alice", (5)
  "given_name": "Alice",
  "family_name": "InChains",
  "email": "alice@keycloak.org"
}
```


(1) **iss** フィールドは、トークンを発行する Red Hat SSO レルムインスタンス URL に対応します。トークンを検証するには、セキュアなエンドポイントデプロイメントで設定する必要があります。

(2) **roles** オブジェクトは、グローバルレルムレベルでユーザーに付与されたロールを提供します。この例では、**alice** には **example-admin** ロールが付与されています。セキュリティーが保護されたエンドポイントが認可されたロールのレルムレベルを検索していることを確認できます。

(3) **resource_access** オブジェクトには、リソースの固有のロール付与が含まれます。このオブジェクトの下には、セキュアな各エンドポイントのオブジェクトを見つけます。

(4) **resource_access.secured-example-endpoint.roles** オブジェクトには、**secured-example-endpoint** リソースの **alice** に付与されるロールが含まれます。

5 **preferred_username** フィールドは、アクセストークンの生成に使用したユーザー名を提供します。

7.6.3.2. アプリケーションクライアント

OAuth 2.0 仕様では、リソースの所有者の代わりにセキュアなリソースにアクセスするアプリケーションクライアントのロールを定義することができます。**master** レルムには、以下のアプリケーションクライアントが定義されています。

demoapp

これは、アクセストークンの取得に使用されるクライアントシークレットを持つ **confidential** タイプのクライアントです。トークンには **alice** ユーザーの権限が含まれ、**alice** が Thorntail、Eclipse Vert.x、Node.js、および Spring Boot ベースの REST サンプルアプリケーションデプロイメントにアクセスできるようにします。

secured-example-endpoint

secured-example-endpoint は、関連するリソース (特に Greeting サービス) にアクセスするために **example-admin** ロールを必要とするベアラーのみのクライアントタイプです。

7.6.4. Eclipse Vert.x SSO アダプター設定

SSO アダプターは、**クライアント側**、または SSO サーバーのクライアントで、Web リソースのセキュリティーを強制するコンポーネントです。ここでは、これは greeting サービスです。

セキュリティーの確率

```
router.route("/greeting")                                    1
  .handler(JWTAuthHandler.create(                              2
    JWTAuth.create(vertx,                                      3
      new JWTAuthOptions()                                     4
          .addPubSecKey(new PubSecKeyOptions()
            .setAlgorithm("RS256")                             5
            .setPublicKey(System.getenv("REALM_PUBLIC_KEY"))) 6
          .setPermissionsClaimKey("realm_access/roles"))));  7
```

1 セキュリティーを保護する HTTP ルートを見つけます。

2 新しい JWT セキュリティーハンドラーをインスタンス化します。

3 認可エンフォーサーが作成されます。

4 エンフォーサーの設定。

5 公開鍵暗号化アルゴリズム。

6 レルム公開鍵の PEM 形式。これは管理コンソールから取得できます。

7 認可エンフォーサーは、権限を検索する必要がある場所です。

ここでのエンフォーサーは、レルム公開鍵の PEM 形式を使用して設定され、アルゴリズムを指定します。さらに、エンフォーサーは keycloak JWT を消費するように設定されているため、トークンにパーミッション要求の場所を提供する必要もあります。

以下は、デプロイメント環境変数から再構築された JSON ファイルです。このファイルは、Web インターフェースを介してアプリケーションと対話する際に使用されます。

```
JsonObject keycloakJson = new JsonObject()
  .put("realm", System.getenv("REALM")) 1
  .put("auth-server-url", System.getenv("SSO_AUTH_SERVER_URL")) 2
  .put("ssl-required", "external")
  .put("resource", System.getenv("CLIENT_ID")) 3
  .put("credentials", new JsonObject()
    .put("secret", System.getenv("SECRET")));
```

1 使用されるセキュリティーレルム。

3 実際の keycloak **クライアント** の設定。

2 Red Hat SSO サーバーのアドレス (ビルド時の挿入)。

7.6.5. Secured サンプルアプリケーションの Minishift または CDK へのデプロイメント

7.6.5.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

7.6.5.2.

前提条件

- 詳細は、「」を参照してください。

手順

-
- 画面上の指示に従って Eclipse Vert.x でサンプルを作成します。デプロイメントタイプについて尋ねられたら、**ローカルに構築して実行** します。
- 画面の指示に従ってください。
  完了したら、**Download as ZIP file**ボタンをクリックし、ファイルをハードドライブに保存します。

7.6.5.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.6.5.4. CLI クライアント oc を使用した Secured サンプルアプリケーションのデプロイメント

このセクションでは、Secured サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

-

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. サンプルの ZIP ファイルから **service.sso.yaml** ファイルを使用して Red Hat SSO サーバーをデプロイします。

```
$ oc create -f service.sso.yaml
```

5. Maven を使用して、Minishift または CDK にデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift -DskipTests \
    -DSSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}
{.spec.host}{"/auth\n"}')
```

このコマンドは、Fabric8 Maven Plugin を使用して Minishift または CDK で S2I プロセス を起動し、Pod を起動します。

このプロセスは、uberjar ファイルと OpenShift リソースを生成し、それらを Minishift または CDK サーバーの現在のプロジェクトにデプロイします。

7.6.6. Secured サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

Minishift または CDK の他に、マイナーな相違点のみが OpenShift Container Platform にサンプルを作成し、デプロイできます。最も重要な相違点は、OpenShift Container Platform でデプロイする前に、Minishift または CDK にサンプルアプリケーションを作成する必要があります。

前提条件

- Minishift または CDK を使用して作成された例

7.6.6.1. CLI クライアント oc の認証

コマンドラインクライアント **oc** を使用して OpenShift Container Platform のサンプルアプリケーションを使用するには、OpenShift Container Platform Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- Red Hat OpenShift Container Platform のアカウント

手順

1. ブラウザーで OpenShift Container Platform URL に移動します。

2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Container Platform アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.6.6.2. CLI クライアント oc を使用した Secured サンプルアプリケーションのデプロイメント

このセクションでは、Secured サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. サンプルの ZIP ファイルから **service.sso.yaml** ファイルを使用して Red Hat SSO サーバーをデプロイします。

```
$ oc create -f service.sso.yaml
```

5. Maven を使用して、OpenShift Container Platform へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift -DskipTests \
    -DSSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}
{.spec.host}{"/auth\n"}')
```

   このコマンドは、Fabric8 Maven プラグインを使用して OpenShift Container Platform で S2I プロセス を起動し、Pod を起動します。

このプロセスは、uberjar ファイルと OpenShift リソースを生成し、それらを OpenShift Container Platform サーバーの現在のプロジェクトにデプロイします。

7.6.7. Secured サンプルアプリケーション API エンドポイントへの認証

Secured サンプルアプリケーションは、呼び出し元が認証および認可されている場合に **GET** 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。クライアントは最初に Red Hat SSO サーバーに対して認証を行い、認証手順によって返されるアクセストークンを使用して Secured サンプルアプリケーションに対して **GET** 要求を実行します。

7.6.7.1. Secured サンプルアプリケーション API エンドポイントの取得

クライアントを使用して例を操作する場合は、**PROJECT_ID** サービスである Secured サンプルアプリケーションのエンドポイントを指定する必要があります。

前提条件

- Secured サンプルアプリケーションがデプロイされ、実行します。
- 認証された **oc** クライアント。

手順

1. 端末アプリケーションで、**oc get routes** コマンドを実行します。
   以下の表の出力例を以下に示します。

例7.1 セキュアなエンドポイントの一覧

Name	ホスト/ポート	パス	サービス	ポート	終了
secure-sso	secure-sso-myproject.LOCAL_OPENSHIFT_HOSTNAME		secure-sso	(すべて)	passthrough
PROJECT_ID	PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME		PROJECT_ID	(すべて)	
sso	sso-myproject.LOCAL_OPENSHIFT_HOSTNAME		sso	(すべて)	

上記の例では、エンドポイントは **http://PROJECT_ID-**

myproject.LOCAL_OPENSHIFT_HOSTNAME になります。**PROJECT_ID** は、developers.redhat.com/launch または Fabric8 Launcher ツールを使用してサンプルを生成する際に入力した名前に基づいています。

7.6.7.2. コマンドラインで HTTP 要求の認証

HTTP POST 要求を Red Hat SSO サーバーに送信してトークンを要求します。以下の例では、CLI ツール jq を使用して JSON 応答からトークン値を抽出します。

前提条件

- セキュアなサンプルエンドポイント URL。詳細は、「Secured サンプルアプリケーション API エンドポイントの取得」を参照してください。
- **jq** コマンドラインツール (任意)。ツールのダウンロードと、詳細な情報は、https://stedolan.github.io/jq/ を参照してください。

手順

1. **curl**、認証情報、**<SSO_AUTH_SERVER_URL>** でアクセストークンを要求し、**jq** コマンドを使用して応答からトークンを展開します。

```
curl -sk -X POST https://<SSO_AUTH_SERVER_URL>/auth/realms/master/protocol/openid-connect/token \
  -d grant_type=password \
  -d username=alice\
  -d password=password \
  -d client_id=demoapp \
  -d client_secret=1daa57a2-b60e-468b-a3ac-25bd2dc2eadc \
  | jq -r '.access_token'

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJRek1nbXhZMUhrQnpxTnR0SnkwMm5jNTNtMGNiWDQxV1hNSTU1MFo4MGVBIn0.eyJqdGkiOiI0NDA3YTliNC04YWRhLTRlMTctODQ2ZS03YjI5MjMyN2RmYTIiLCJleHAiOjE1MDc3OTM3ODcsIm5iZiI6MCwiaWF0IjoxNTA3NzkzNzI3LCJpc3MiOiJodHRwczovL3NlY3VyZS1zc28tc3NvLWRlbW8uYXBwcy5jYWZlLWJhYmUub3JnL2F1dGgvcmVhbG1zL21hc3RlciIsImF1ZCI6ImRlbW9hcHAiLCJzdWIiOiJjMDE3NWNjYi0wODkyLTRiMzEtODI5Zi1kZGE4NzM4MTVmZTgiLCJ0eXAiOiJCZWFyZXIiLCJhenAiOiJkZW1vYXBwIiwiYXV0aF90aW1lIjowLCJzZXNzaW9uX3N0YXRlIjoiMDFjOTkzNGQtNmZmOS00NWYzLWJkNWUtMTU4NDI5ZDZjNDczIiwiYWNyIjoiMSIsImNsaWVudF9zZXNzaW9uIjoiMzM3Yzk0MTYtYTdlZS00ZWUzLThjZWQtODhlODI0MGJjNTAyIiwiYWxsb3dlZC1vcmlnaW5zIjpbIioiXSwicmVhbG1fYWNjZXNzIjp7InJvbGVzIjpbImJvb3N0ZXItYWRtaW4iXX0sInJlc291cmNlX2FjY2VzcyI6eyJzZWN1cmVkLWJvb3N0ZXItZW5kcG9pbnQiOnsicm9sZXMiOlsiYm9vc3Rlci1hZG1pbiJdfSwiYWNjb3VudCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsInZpZXctcHJvZmlsZSJdfX0sIm5hbWUiOiJBbGljZSBJbkNoYWlucyIsInByZWZlcnJlZF91c2VybmFtZSI6ImFsaWNlIiwiZ2l2ZW5fbmFtZSI6IkFsaWNlIiwiZmFtaWx5X25hbWUiOiJJbkNoYWlucyIsImVtYWlsIjoiYWxpY2VAa2V5Y2xvYWsub3JnIn0.mjmZe37enHpigJv0BGuIitOj-kfMLPNwYzNd3n0Ax4Nga7KpnfytGyuPSvR4KAG8rzkfBNN9klPYdy7pJEeYlfmnFUkM4EDrZYgn4qZAznP1Wzy1RfVRdUFi0-GqFTMPb37o5HRldZZ09QljX_j3GHnoMGXRtYW9RZN4eKkYkcz9hRwgfJoTy2CuwFqeJwZYUyXifrfA-JoTr0UmSUed-0NMksGrtJjjPggUGS-qOn6OgKcmN2vaVAQlxW32y53JqUXctfLQ6DhJzIMYTmOfIIPy0sgG1mG7sovQhw1xTg0vTjdx8zQ-EJcexkj7IivRevRZsslKgqRFWs67jQAFQA
```

<SSO_AUTH_SERVER_URL> は **secure-sso** サービスの URL です。

通常、**username**、**password**、**client_secret** などの属性は、通常、秘密になりますが、上記のコマンドはこの例で提供されているデフォルトの認証情報をデモの目的で使用します。

jq を使用してトークンを抽出しない場合は、**curl** コマンドのみを実行し、アクセストークンを手動で抽出できます。

注記

-sk オプションは curl に対し、自己署名証明書の失敗を無視します。実稼働環境ではこのオプションを使用しないでください。macOS では、**curl** バージョン **7.56.1** 以降がインストールされている必要があります。また、OpenSSL でビルドする必要があります。

1. Secured サービスを起動します。アクセス (ベアラー) トークンを HTTP ヘッダーに割り当てます。

```
$ curl -v -H "Authorization: Bearer <TOKEN>" http://<SERVICE_HOST>/api/greeting

{
   "content": "Hello, World!",
   "id": 2
}
```

例7.2 Access (Bearer) トークンが含まれるGET 要求ヘッダーのサンプル

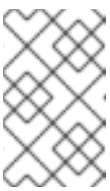
```
> GET /api/greeting HTTP/1.1
> Host: <SERVICE_HOST>
> User-Agent: curl/7.51.0
> Accept: */*
> Authorization: Bearer <TOKEN>
```

<SERVICE_HOST> は、Secured サンプルエンドポイントの URL です。詳細は、「Secured サンプルアプリケーション API エンドポイントの取得」を参照してください。

2. アクセストークンの署名を確認します。
アクセストークンは JSON Web トークン であるため、JWT デバッガー を使用してデコードできます。

   a. Web ブラウザーで、JWT デバッガー の Web サイトに移動します。

   b. **Algorithm** ドロップダウンメニューから **RS256** を選択します。



注記

選択後に Web フォームが更新され、正しい RSASHA256(...) 情報が Signature セクションに表示されることを確認します。そうでない場合は、HS256 に切り替えてから RS256 に戻ります。

   c. 上部のテキストボックスの **VERIFY SIGNATURE** セクションに以下のコンテンツを貼り付けます。

```
-----BEGIN PUBLIC KEY-----
```

```
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoETnPmN55xBJjRzN/cs30OzJ
9olkteLVNRjzdTxFOyRtS2ovDfzdhhO9XzUcTMbIsCOAZtSt8K+6yvBXypOSYvI75EUdypm
kcK1KoptqY5KEBQ1KwhWuP7IWQ0fshUwD6jI1QWDfGxfM/h34FvEn/0tJ71xN2P8TI2Yan
wuDZgosdobx/PAvlGREBGuk4BgmexTOkAdnFxIUQcCkiEZ2C41uCrxiS4CEe5OX91aK9
HKZV4ZJX6vnqMHmdDnsMdO+UFtxOBYZio+a1jP4W3d7J5fGeiOaXjQCOpivKnP2yU2D
PdWmDMyVb67l8DRA+jh0OJFKZ5H2fNgE3II59vdsRwIDAQAB
-----END PUBLIC KEY-----
```

注記

これは、Secured サンプルアプリケーションの Red Hat SSO サーバーデプロイメントからのマスターレルム公開鍵です。

d. クライアント出力から **Encoded** されたボックスに **token** 出力を貼り付けます。**Signature Verified** 記号がデバッガーページに表示されます。

7.6.7.3. Web インターフェースを使用した HTTP 要求の認証

セキュアなエンドポイントには、HTTP API の他に、対話する Web インターフェースも含まれます。

以下の手順は、セキュリティーの実施方法、認証方法、および認証トークンの使用方法を確認するための演習です。

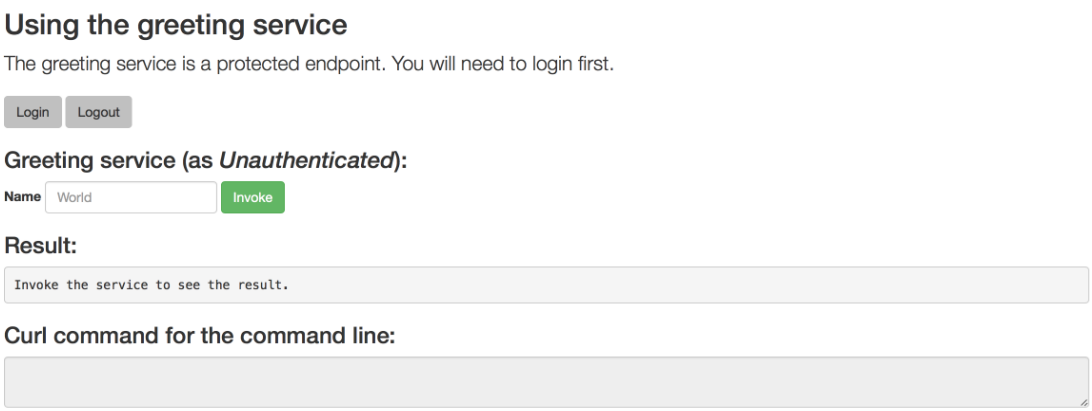
前提条件

- セキュリティーが保護されたエンドポイント URL。詳細は、「Secured サンプルアプリケーション API エンドポイントの取得」を参照してください。

手順

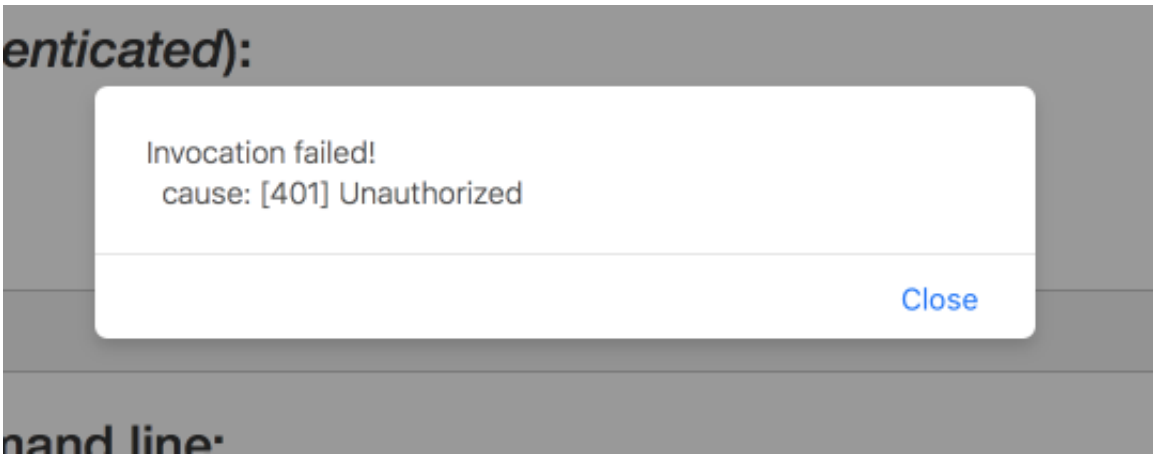
1. Web ブラウザーで、エンドポイント URL に移動します。
2. 認証されていない要求を実行します。
   a. **Invoke** ボタンをクリックします。

図7.1 認証されていないセキュアなサンプル Web インターフェース



サービスは、**HTTP 401 Unauthorized** ステータスコードで応答します。

図7.2 未認証のエラーメッセージ



3. 認証された要求をユーザーとして実行します。

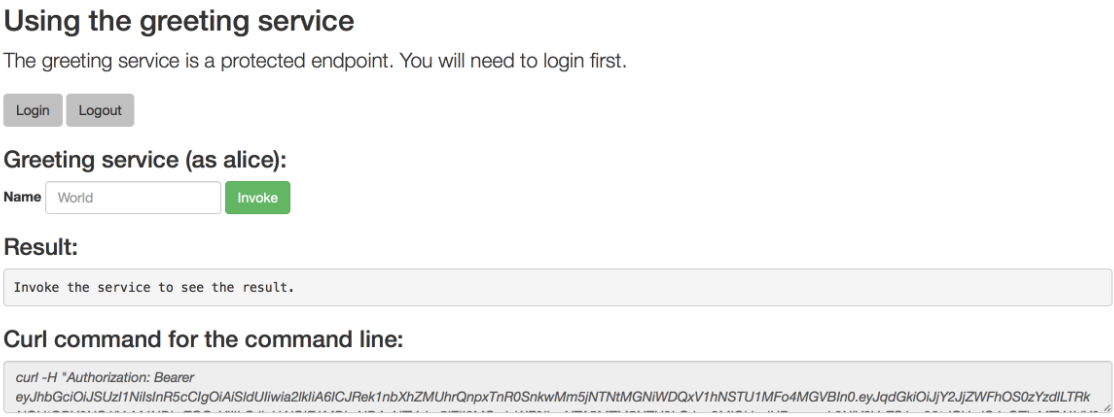
   a. **Login** ボタンをクリックして Red Hat SSO に対して認証を行います。SSO サーバーにリダイレクトされます。

   b. Alice ユーザー としてログインします。Web インターフェースにリダイレクトされます。

注記

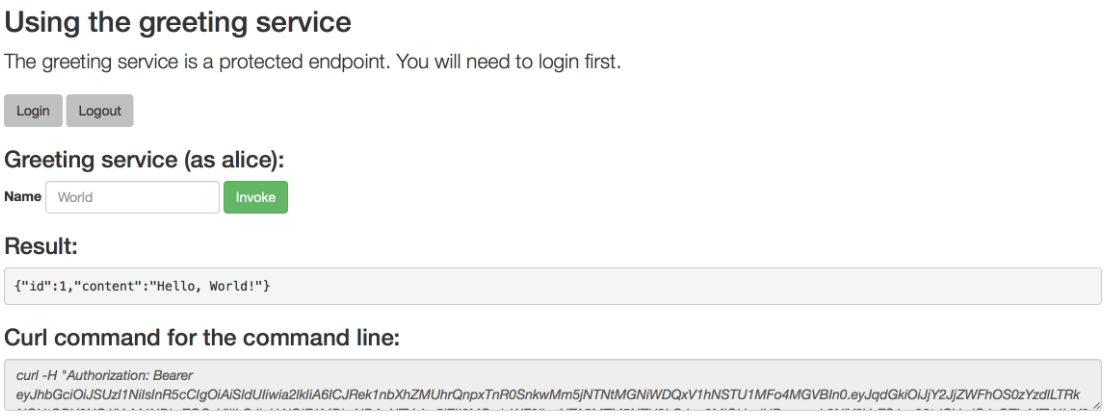
ページ下部のコマンドライン出力に、アクセス (ベアラー)トークンが表示されます。

図7.3 (Alice として) 認証されたセキュアなサンプル Web インターフェース



   c. 再度 **Invoke** をクリックして Greeting サービスにアクセスします。
   例外がなく、JSON 応答ペイロードが表示されていることを確認します。つまり、サービスはアクセス (ベアラー) トークンを許可し、Greeting サービスへのアクセスが認可されていることを意味します。

図7.4 (Alice として) 認証されたグリーティング要求の結果



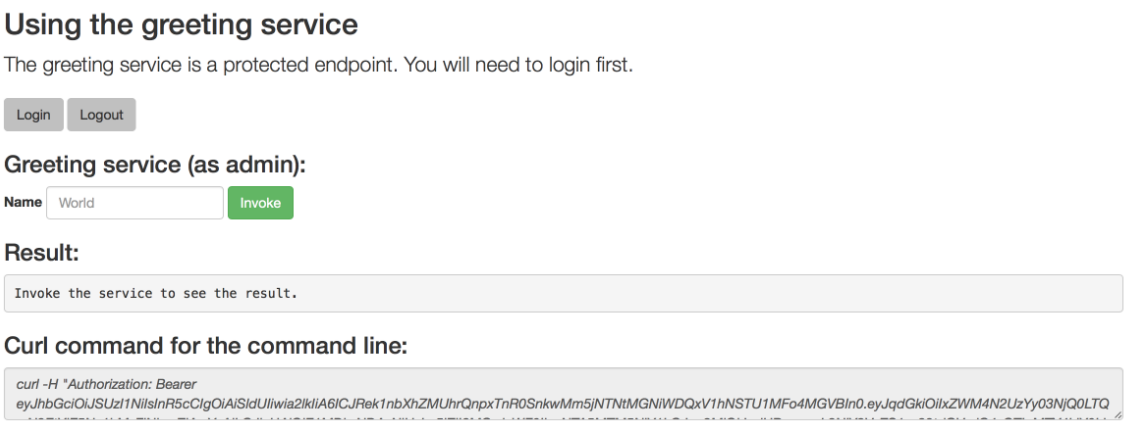
d. ログアウトします。

4. 認証された要求を管理者として実行します。

   a. **Invoke** ボタンをクリックします。
   これにより、認証されていない要求が Greeting サービスに送信されていることを確認します。

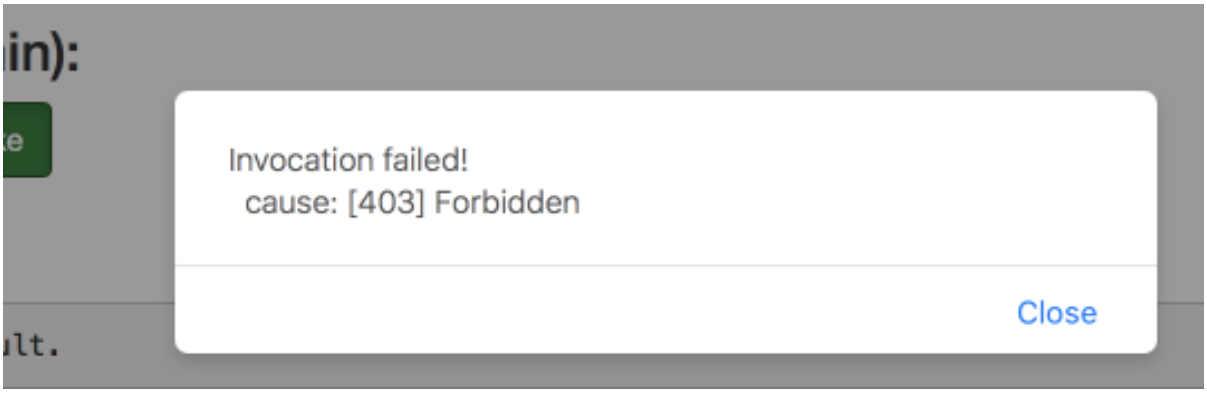
   b. **Login** ボタンをクリックして admin ユーザー としてログインします。

図7.5 (admin として) 認証されたセキュアな Web インターフェースの例



5. **Invoke** ボタンをクリックします。
   このサービスは、**admin** ユーザーが Greeting サービスへのアクセスが許可されていないため、**HTTP 403 Forbidden** ステータスコードで応答します。

図7.6 承認されていないエラーメッセージ



7.6.8. Eclipse Vert.x の Secured サンプルアプリケーション統合テストの実行

本セクションでは、事前設定されたレルムとサンプルユーザープロファイルを使用して、Red Hat SSO テストサーバーを使用して統合テストを実行する方法を説明します。

前提条件

- 認証された **oc** クライアント。

手順

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

デフォルトでは、SSO サーバーはテストの一環としてデプロイ (および破棄) されます。統合テストを実行するステップは次のとおりです。

1. 端末アプリケーションで、プロジェクトのディレクトリーに移動します。
2. インテグレーションテストを実行します。

```
mvn clean verify -Popenshift,openshift-it
```

事前に SSO サーバーをデプロイしている場合 (たとえば **oc create -f service.sso.yaml** を実行して SSO サーバーをデプロイしている場合) は、テストの実行中にシステムプロパティー **skip.sso.init** を **true** に設定します。

```
mvn clean verify -Popenshift,openshift-it -Dskip.sso.init=true
```

このように実行されると、テストは既存の SSO サーバーを使用します。テストは独自の SSO サーバーをデプロイせず、既存の SSO サーバーを破棄しません。

7.6.9. セキュアな SSO リソース

OAuth2 仕様の背後にある原則に関する追加情報や、Red Hat SSO および Keycloak を使用してアプリケーションのセキュリティーを保護する方法については、以下のリンクを参照してください。

- Aaron Parecki: OAuth2 Simplified
- Red Hat SSO 7.1 ドキュメント
- Keycloak 3.2 のドキュメント
- Spring Boot のセキュアなサンプルアプリケーション

- Thorntail のセキュアなアプリケーションの例
- Secured for Node.js

7.7. ECLIPSE VERT.X のキャッシュの例

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: Advanced

キャッシュサンプルは、キャッシュを使用してアプリケーションの応答時間を長くする方法を示しています。

この例では、以下の方法を示しています。

- キャッシュを OpenShift にデプロイします。
- アプリケーション内でキャッシュを使用します。

7.7.1. キャッシュの仕組みおよび必要なタイミング

キャッシュを使用すると、特定の期間情報を保存し、アクセスすることができます。元のサービスを繰り返し呼び出すよりも、キャッシュの情報により迅速またはより確実にアクセスすることができます。キャッシュを使用する欠点は、キャッシュされた情報が最新ではないことです。ただし、キャッシュに保存されている各値に **有効期限** または TTL (time to live) に設定すると、この問題を軽減できます。

例7.3 キャッシュの例

service1 および **service2** の 2 つのアプリケーションがあるとします。

- **Service1** は **service2** からの値によって異なります。
  - **service2** からの値が頻繁に変更されると、**service1** は一定期間 **service2** からの値をキャッシュする可能性があります。
  - キャッシュされた値を使用すると、**service2** が呼び出される回数を減らすことができます。
- **service1** が **service2** から直接値を取得するのに 500 ミリ秒かかり、キャッシュされた値を取得するのに 100 ミリ秒かかる場合、**service1** はキャッシュされた各呼び出しに対してキャッシュされた値を使用することで 400 ミリ秒短くすることができます。
- **service1** が **service2** にキャッシュされていない呼び出しを 1 秒あたり 5 回、10 秒以上行うとすると、呼び出しは 50 になります。
- 代わりに **service1** が 1 秒の TTL でキャッシュされた値の使用を開始した場合は、10 秒で 10 コールに削減されます。

キャッシュサンプルの仕組み

1. **cache** サービス、**cute name** サービス、および **greeting** サービスがデプロイされ、公開されます。
2. ユーザーは **greeting** サービスの Web フロントエンドにアクセスします。
3. ユーザーは、Web フロントエンドのボタンを使用して **greeting** HTTP API を呼び出します。
4. **greeting** サービスは、**cute name** サービスの値によって異なります。
   - **greeting** サービスは、最初にその値が **cache** サービスに保存されているかどうかを確認します。これが存在する場合は、キャッシュされた値が返されます。
   - 値がキャッシュされていない場合、**greeting** サービスは **cute name** サービスを呼び出し、値を返し、その値を 5 秒の TTL で **cache** サービスに保存します。
5. Web フロントエンドは、**greeting** サービスからの応答と、操作の合計時間を表示します。
6. ユーザーはサービスを複数回呼び出し、キャッシュされた操作とキャッシュされていない操作の違いを確認します。
   - キャッシュされた操作は、キャッシュされていない操作よりもはるかに高速です。
   - ユーザーは、TTL の有効期限が切れる前にキャッシュを強制的に消去することができます。

7.7.2. キャッシュサンプルアプリケーションの OpenShift Online へのデプロイ

以下のオプションのいずれかを使用して、OpenShift Online で Cache サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

7.7.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェースから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.7.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online Web インターフェースによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.7.2.3. CLI クライアント oc を使用したキャッシュサンプルアプリケーションのデプロイメント

このセクションでは、キャッシュサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は、「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

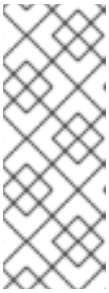
```
$ unzip MY_PROJECT_NAME.zip
```

2. 新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. キャッシュサービスをデプロイします。

```
$ oc apply -f service.cache.yml
```



注記

x86_64 以外のアーキテクチャーを使用している場合は、YAML ファイルで Red Hat Data Grid のイメージ名を、そのアーキテクチャー内の関連するイメージ名に更新します。たとえば、s390x または ppc64le アーキテクチャーの場合は、イメージ名を IBM Z または IBM Power Systems のイメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新します。

5. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

6. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
cache-server-123456789-aaaaa            1/1       Running     0          8m
MY_APP_NAME-cutename-1-bbbbb       1/1       Running     0          4m
MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
MY_APP_NAME-greeting-1-ccccc       1/1       Running     0          3m
MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
```

3 つの Pod が完全にデプロイされて起動すると、ステータスは **Running** でなければなりません。

7. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-cutename   MY_APP_NAME-cutename-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-cutename   8080
None
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-greeting   8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用して greeting サービスにアクセスします。

7.7.3. Cache サンプルアプリケーションの Minishift または CDK へのデプロイ

以下のオプションのいずれかを使用して、Minishift または CDK でキャッシュサンプルアプリケーションをローカルで実行します。

-
- CLI クライアント **oc** の使用

7.7.3.1.

この情報は、Minishift または CDK の開始時に提供されます。

前提条件

-

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. **Minishift または CDK 起動時のコンソール出力の例**

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

7.7.3.2.

前提条件

- 詳細は、「」を参照してください。

手順

1.
2. 画面上の指示に従って、Eclipse Vert.x でアプリケーションのサンプルを作成して起動します。

7.7.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェースが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 詳細は、「」を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

7.7.3.4. CLI クライアント oc を使用したキャッシュサンプルアプリケーションのデプロイメント

このセクションでは、キャッシュサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- 詳細は、「」を参照してください。
-
- 認証された **oc** クライアント。詳細は、「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. キャッシュサービスをデプロイします。

```
$ oc apply -f service.cache.yml
```

注記

x86_64 以外のアーキテクチャーを使用している場合は、YAML ファイルで Red Hat Data Grid のイメージ名を、そのアーキテクチャー内の関連するイメージ名に更新します。たとえば、s390x または ppc64le アーキテクチャーの場合は、イメージ名を IBM Z または IBM Power Systems のイメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新します。

5. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

6. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
cache-server-123456789-aaaaa            1/1       Running     0          8m
MY_APP_NAME-cutename-1-bbbbb       1/1       Running     0          4m
MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
MY_APP_NAME-greeting-1-ccccc       1/1       Running     0          3m
MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
```

3 つの Pod が完全にデプロイされて起動すると、ステータスは **Running** でなければなりません。

7. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-cutename   MY_APP_NAME-cutename-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-cutename   8080
None
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用して greeting サービスにアクセスします。

7.7.4. キャッシュサンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「キャッシュサンプルアプリケーションの OpenShift Online へのデプロイ」 の手順に従って、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

7.7.5. 未変更の Cache サンプルアプリケーションとの対話

デフォルトの Web インターフェースを使用して、未変更の Cache サンプルアプリケーションを操作し、頻繁にアクセスされるデータの保存方法により、サービスへのアクセスに必要な時間を短縮できます。

前提条件

- アプリケーションがデプロイされている。

手順

1. ブラウザーを使用して **greeting** サービスに移動します。
2. **サービスの起動** を一度クリックします。
   duration の値が **2000** を超えることに注意してください。また、キャッシュ状態が **No cached value** から **A value is cached** になっていることに注意してください。
3. 5 秒間待機して、キャッシュ状態が **No cached value** に戻されるのを確認します。
   キャッシュされた値の TTL は 5 秒に設定されています。TTL の期限が切れると、値はキャッシュされなくなります。
4. **Invoke the service** を一度クリックして、値をキャッシュします。
5. キャッシュ状態が **A value is cached** になっている数秒の間に **Invoke the service** をさらに数回クリックします。
   キャッシュされた値を使用しているため、**duration** の値が大幅に低いことに注意してください。**Clear the cache** をクリックすると、キャッシュは空になります。

7.7.6. キャッシュサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

7.7.7. リソースのキャッシュ

キャッシュに関する背景や詳細情報は、以下を参照してください。

- Spring Boot のキャッシュの例
- Thorntail のキャッシュ
- Node.js のキャッシュ

付録A SOURCE-TO-IMAGE (S2I) ビルドプロセス

Source-to-Image (S2I) は、アプリケーションソースのあるオンライン SCM リポジトリーから再現可能な Docker 形式のコンテナーイメージを生成するビルドツールです。S2I ビルドを使用すると、ビルド時間を短縮し、リソースとネットワークの使用量を減らし、セキュリティーを改善し、その他の多くの利点を活用して、アプリケーションの最新バージョンを本番環境に簡単に配信できます。OpenShift は、複数の ビルドストラテジーおよび入力ソース をサポートします。

詳細は、OpenShift Container Platform ドキュメントの「Source-to-Image (S2I) ビルド」の章を参照してください。

最終的なコンテナーイメージをアセンブルするには、S2I プロセスに 3 つの要素を提供する必要があります。

- GitHub などのオンライン SCM リポジトリーでホストされるアプリケーションソース。
- S2I Builder イメージ。アセンブルされたイメージの基盤となり、アプリケーションが実行しているエコシステムを提供します。
- 必要に応じて、S2I スクリプト によって使用される環境変数およびパラメーターを指定することもできます。

このプロセスは、S2I スクリプトで指定された指示に従ってアプリケーションソースと依存関係を Builder イメージに挿入し、アセンブルされたアプリケーションを実行する Docker 形式のコンテナーイメージを生成します。詳細は、OpenShift Container Platform ドキュメントの「S2I build requirements」、「build options」、および「how builds work」を参照してください。

付録B サンプルアプリケーションのデプロイメント設定の更新

サンプルアプリケーションのデプロイメント設定には、ルート情報や Readiness プローブの場所など、OpenShift でのアプリケーションのデプロイおよび実行に関連する情報が含まれます。サンプルアプリケーションのデプロイメント設定は YAML ファイルのセットに保存されます。Nodeshift を使用する例として、YAML ファイルは **.nodeshift** ディレクトリーにあります。

重要

Nodeshift によって生成されるリソース定義は **tmp/nodeshift/resource/** ディレクトリーにあります。

前提条件

- 既存のサンプルプロジェクト。
- CLI クライアント **oc** がインストールされている。

手順

1. 既存の YAML ファイルを編集したり、設定を更新して追加の YAML ファイルを作成します。
   - たとえば、サンプルに **readinessProbe** が設定された YAML ファイルがすでにある場合は、**path** の値を別の利用可能なパスに変更し、Readiness の有無を確認することができます。

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - **readinessProbe** が既存の YAML ファイルで設定されていない場合は、**readinessProbe** 設定を使用して新規 YAML ファイルを同じディレクトリーに作成することもできます。
2. Maven または npm を使用して、サンプルの更新バージョンをデプロイします。
3. 設定更新が、デプロイ済みのサンプルに表示されることを確認します。

```
$ oc export all --as-template='my-template'

apiVersion: template.openshift.io/v1
kind: Template
metadata:
  creationTimestamp: null
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
```

```
spec:
  ...
  template:
    ...
    spec:
      containers:
        ...
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /path/to/different/probe
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 60
          periodSeconds: 30
          successThreshold: 1
          timeoutSeconds: 1
        ...
```

関連情報

Web ベースのコンソール、または CLI クライアント **oc** を使用してアプリケーションの設定を直接更新した場合は、これらの変更を YAML ファイルへエクスポートして追加します。**oc export all** コマンドを使用して、デプロイされたアプリケーションの設定を表示します。

付録C

前提条件

- OpenShift クラスターへのアクセス。
- 同じ OpenShift クラスターで実行している Jenkins コンテナーイメージ。
- Jenkins サーバーに JDK および Maven がインストールされ、設定されている。
- 

  注記

  アプリケーションをビルドおよび OpenShift にデプロイする場合、Eclipse Vert.x 3.9 は OpenJDK 8 および OpenJDK 11 をベースとしたビルダーイメージのみをサポートします。Oracle JDK および OpenJDK 9 のビルダーイメージはサポートされていません。

  pom.xml の例

  ```
  <properties>
    ...
    <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
  openshift:latest</fabric8.generator.from>
  </properties>
  ```

- GitHub で利用可能なアプリケーションのソース。

手順

1. アプリケーションの新規 OpenShift プロジェクトを作成します。
   a. OpenShift Web コンソールを開き、ログインします。
   b. **Create Project** をクリックし、新規 OpenShift プロジェクトを作成します。
   c. プロジェクト情報を入力し、**Create** をクリックします。
2. Jenkins がそのプロジェクトにアクセスできるようにします。
   たとえば、Jenkins のサービスアカウントを設定した場合は、アカウントに、アプリケーションのプロジェクトへの **edit** アクセスがあることを確認してください。
3. Jenkins サーバーで新しい フリースタイルの Jenkins プロジェクト を作成します。
   a. **New Item** をクリックします。
   b. 名前を入力し、**Freestyle プロジェクト** を選択して **OK** をクリックします。
   c. **Source Code Management** で **Git** を選択し、アプリケーションの GitHub URL を追加します。
   d. **Build** で **Add build step** を選択し、**Invoke top-level Maven target** を選択します。
   e. 以下を **Goals** に追加します。

      ```
      clean fabric8:deploy -Popenshift -Dfabric8.namespace=MY_PROJECT
      ```

- **MY_PROJECT** をアプリケーションの OpenShift プロジェクトの名前に置き換えます。

f. **Save** をクリックします。

4. Jenkins プロジェクトのメインページから **Build Now** をクリックし、アプリケーションの OpenShift プロジェクトへのアプリケーションのビルドおよびデプロイを確認します。
アプリケーションの OpenShift プロジェクトでルートを開いて、アプリケーションがデプロイされていることを確認することもできます。

次のステップ

- GITSCM ポーリング を追加すること、または the **Poll SCM** ビルドトリガー を使用することを検討してください。これらのオプションにより、新規コミットが GitHub リポジトリーにプッシュされるたびにビルドを実行できます。
- デプロイ前にテストを実行するビルドステップを追加することを検討してください。

付録D 追加の ECLIPSE VERT.X リソース

- リアクティブマニフェスト
- Eclipse Vert.x プロジェクト
- 動作中の Vert.x
- Eclipse Vert.x for Reactive Programming
- Building Reactive Microservices in Java
- Eclipse Vert.x Cheat Sheet for Developers
- Vert.x - From zero to (micro)-hero
- Red Hat Summit 2017 Talk - Reactive Programming with Eclipse Vert.x
- Red Hat Summit 2017 Breakout Session - Reactive Systems with Eclipse Vert.x and Red Hat OpenShift
- Eclipse Vert.x および OpenShift でのライブコーディングリアクティブシステム

付録E アプリケーション開発リソース

OpenShift でのアプリケーション開発に関する詳細は、以下を参照してください。

- OpenShift インタラクティブラーニングポータル

ネットワークの負荷を削減し、アプリケーションのビルド時間を短縮するには、Minishift または CDK に Maven の Nexus ミラーを設定します。

- Maven 用の Nexus ミラーリングの設定

付録F 上達度レベル

使用できる各例は、特定の最小知識を必要とする概念について言及しています。この要件は例によって異なります。最小要件と概念は、いくつかの上達度レベルで構成されています。ここで説明するレベルの他に、各例に固有の追加情報が必要になる場合があります。

Foundational

Foundational とされている例は、通常、主題に関する事前の知識を必要としません。この例では、主要な要素、概念、および用語の一般的な認識とデモンストレーションを提供します。この例の説明で直接言及されているものを除き、特別な要件はありません。

Advanced

Advanced サンプルを使用する場合は、Kubernetes および OpenShift の他に、例の主題の領域の一般的な概念および用語について理解していることを前提としています。サービスやアプリケーションの設定、ネットワークの管理など、独自の基本的なタスクを実行できるようにする必要があります。この例ではサービスが必要で、設定が例の範囲に含まれていない場合には、適切に設定する知識があり、サービスの結果として生じる状態のみがドキュメントに記載されていることを前提とします。

Expert

Expert サンプルでは、主題について最も高いレベルの知識が必要です。機能ベースのドキュメントとマニュアルに基づいて多くのタスクを実行することが期待されており、ドキュメントは最も複雑なシナリオを対象としています。

付録G 用語

G.1. 製品およびプロジェクト名

Developer Launcher (developers.redhat.com/launch)

Developer Launcher (developers.redhat.com/launch) は、Red Hat が提供するスタンドアロンの入門エクスペリエンスです。これは、OpenShift でのクラウドネイティブ開発の開始に役立ちます。これには、OpenShift にダウンロード、ビルド、およびデプロイできる機能のサンプルアプリケーションが含まれます。

Minishift または CDK

Minishift を使用してマシンで実行している OpenShift クラスター。

G.2. DEVELOPER LAUNCHER に固有の用語

例

アプリケーション仕様 (例: **REST API を持つ Web サービス**)。
この例では、通常、実行する言語やプラットフォームを指定していません。説明には意図した機能のみが含まれています。

サンプルアプリケーション

特定の ランタイム における特定の サンプル の言語固有の実装。サンプルアプリケーションは、サンプルカタログ に記載されています。
たとえば、サンプルアプリケーションは Thorntail ランタイムを使用して実装される REST API を持つ Web サービスです。

サンプルカタログ

サンプルアプリケーションに関する情報が含まれる Git リポジトリー。

ランタイム

アプリケーションの例 を実行するプラットフォーム。たとえば、Thorntail または Eclipse Vert.x などです。