Red Hat build of Eclipse Vert.x 4.3

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: 2023-04-08

Red Hat build of Eclipse Vert.x 4.3 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

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 (英語のみ)

弊社のドキュメントに関するご意見やご感想をお寄せください。フィードバックをお寄せいただくには、ドキュメントのテキストを強調表示し、コメントを追加できます。

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ている。
- Red Hat カスタマーポータルで、**マルチページ HTML** 形式でドキュメントを表示している。

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 **フィードバック** ボタンをクリックして、既存のフィードバックを確認します。

注記

フィードバック機能は、**マルチページ 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. ハイライトされたテキスト近くに表示される **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。ドキュメントに関する問題が作成されます。
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。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 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 で実行す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モニターリングの管理、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 CodeReady Studio

アプリケーションの開発、テスト、およびデプロイを行う統合開発環境 (IDE)。

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. ECLIPSE VERT.X の概要

Eclipse Vert.x は、Java 仮想マシン (JVM) 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ため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。ローリング更新、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。

1.2.1. 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、メンテナー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。

1.2.2. 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 System (ppc64le)

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。

1.2.3. 連邦情報処理標準 (FIPS) のサポート

FIPS (Federal Information Processing Standards) は、コンピューターシステムやネットワーク間のセキュリティーおよび相互運用性を強化するためのガイドラインと要件を提供します。FIPS 140-2 および 140-3 シリーズは、ハードウェアおよびソフトウェアの両レベルで暗号化モジュールに適用されます。

連邦情報処理標準 (FIPS) 140-2 は、U.S により開発されたコンピューターセキュリティー標準です。暗号化モジュールの品質を検証する政府および業界の作業グループ。NIST Computer Security Resource Center で公式の FIPS の刊行物を参照してください。

Red Hat Enterprise Linux (RHEL) は、FIPS 140-2 コンプライアンスシステム全体を有効にする統合フレームワークを提供します。FIPS モードで操作する場合、暗号化ライブラリーを使用するソフトウェアパッケージはグローバルポリシーに従って自己設定されます。

コンプライアンスの要件については、Red Hat Government Standards ページを参照してください。

Eclipse Vert.x の Red Hat ビルドは、FIPS 対応の RHEL システムで実行され、RHEL が提供する FIPS 認定ライブラリーを使用します。

1.2.3.1. 関連情報

- FIPS モードを有効にして RHEL をインストールする方法は、FIPS モードが有効になっている RHEL 8 システムのインストール を参照してください。
- RHEL をインストールした後に FIPS モードを有効にする方法は、FIPS モードへのシステムの切り替え を参照してください。

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

Eclipse Vert.x を使用するようにアプリケーションを設定する場合は、アプリケーションのルートディレクトリーにある **pom.xml** ファイルの Eclipse Vert.x BOM (Bill of Materials) アーティファクトを参照する必要があります。BOM は、アーティファクトの正しいバージョンを設定するのに使用されます。

前提条件

- Maven ベースのアプリケーション

手順

1. **pom.xml** ファイルを開き、**io.vertx:vertx-dependencies** アーティファクトを **<dependencyManagement>** セクションに追加します。**type** を **pom** として指定し、**scope** を **import** として指定します。

```
<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. 以下のプロパティーを追加して、使用する Eclipse Vert.x および Eclipse Vert.x Maven Plugin のバージョンを追跡します。
プロパティーを使用して、リリースごとに変更する値を設定できます。たとえば、製品またはプラグインのバージョンです。

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

3. アプリケーションのパッケージ化に使用されるプラグインとして **vertx-maven-plugin** を指定します。

```
<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。

第3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

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。

3.1. 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 4.3 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. 新しいディレクトリー **myApp** を作成し、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.Promise;

public class MyApp extends AbstractVerticle {
```

```
  @Override
  public void start(Promise<Void> promise) {
    vertx
      .createHttpServer()
      .requestHandler(r ->
        r.response().end("Greetings!"))
      .listen(8080, result -> {
        if (result.succeeded()) {
          promise.complete();
        } else {
          promise.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>4.3.7.redhat-00002</vertx.version>
  <vertx-maven-plugin.version>1.0.24</vertx-maven-plugin.version>
  <vertx.verticle>com.example.MyApp</vertx.verticle>

  <!-- Specify the JDK builder image used to build your application. -->
  <jkube.generator.from>registry.access.redhat.com/ubi8/openjdk-11</jkube.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!
```

関連情報

- 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ーリングを有効にできます。

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

Eclipse Vert.x アプリケーションを OpenShift 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、OpenShift Maven プラグインを使用します。

注記

Fabric8 Maven プラグインはサポート対象外になりました。OpenShift Maven プラグインを使用して、OpenShift に Eclipse Vert.x アプリケーションをデプロイします。詳細は、migrating from Fabric8 Maven Plugin to Eclipse JKube セクションを参照してください。

pom.xml ファイルの **jkube.generator.from** URL を置き換えて、Java イメージを指定できます。イメージは Red Hat Ecosystem Catalog で利用できます。

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

たとえば、OpenJDK 8 を使用する RHEL 7 の Java イメージは、以下のように指定します。

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

3.2.1. 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。たとえば、Java イメージは、RHEL 7 で OpenJDK 8 または OpenJDK 11 で利用できます。

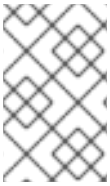
Eclipse Vert.x は、RHEL 8 上の Red Hat OpenJDK 8 および Red Hat OpenJDK 11 用の OCI 準拠の ユニバーサルベースイメージ を使用して、Eclipse Vert.x アプリケーションをビルドして OpenShift にデプロイするためのサポートを導入します。

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 で対応しているコンテナーイメージが記載されています。これらのコンテナーイメージは、Red Hat Ecosystem Catalog で入手できます。カタログでは、下の表に記載されるイメージを検索してダウンロードできます。イメージページには、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。

表3.1 OpenJDK イメージおよびアーキテクチャー

JDK (OS)	アーキテクチャーのサポート	Red Hat Ecosystem Catalog で利用可能なイメージ
OpenJDK8 (RHEL 7)	x86_64	redhat-openjdk-18/openjdk18-openshift
OpenJDK11 (RHEL 7)	x86_64	openjdk/openjdk-11-rhel7
OpenJDK8 (RHEL 8)	x86_64	ubi8/openjdk-8-runtime
OpenJDK11 (RHEL 8)	x86_64、IBMZ、および IBM Power	ubi8/openjdk-11



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 または 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。

3.2.2. 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。

以下の手順では、OpenShift Maven プラグインを使用するプロファイルは、アプリケーションの OpenShift へのビルドおよびデプロイに使用されます。

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

...
```

```
<profiles>
  <profile>
    <id>openshift</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.eclipse.jkube</groupId>
          <artifactId>openshift-maven-plugin</artifactId>
          <version>1.1.1</version>
          <executions>
            <execution>
              <goals>
                <goal>resource</goal>
                <goal>build</goal>
                <goal>apply</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

2. **pom.xml** ファイルの **jkube.generator.from** プロパティーを置き換え、使用する OpenJDK イメージを指定します。

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

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

  - OpenJDK 11 を使用した RHEL 7

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

  - OpenJDK 8 を使用した RHEL 8

```
<jkube.generator.from>registry.access.redhat.com/ubi8/openjdk-8:latest</jkube.generator.from>
```

- x86_64、s390x (IBM Z)、および ppc64le (IBM Power Systems) アーキテクチャー
  - OpenJDK 11 を使用した RHEL 8

```
<jkube.generator.from>registry.access.redhat.com/ubi8/openjdk-11:latest</jkube.generator.from>
```

3.2.3. OpenShift Maven プラグインを使用した Eclipse Vert.x アプリケーションの OpenShift へのデプロイメント

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。
- アプリケーションを OpenShift インスタンスにデプロイします。

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 oc:deploy -Popenshift
```

このコマンドは、OpenShift 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** である必要があります。

特定の Pod 名が異なります。

b. Pod のルートを判別します。

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、アクセスに使用するベース URL を提供します。

この例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** がアプリケーションにアクセスするためのベース URL です。

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!
```

3.3. 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。

3.3.1. 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>1.0.24</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。

3.3.2. 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. 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4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。

4.1. 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。

4.1.1. 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覚えておいてください。

-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。

4.1.2. 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の **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、**$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覚えておいてください。

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。

4.1.3. 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 を起動します。
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

4.2. 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。

4.2.1. 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。

4.2.2. 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。

4.2.3. 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** インターフェイ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。

4.2.4. 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);
```

4.2.5. 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 oc: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、ターミナル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、次のコマンドは、**/api/greeting** メソッドで **message** 変数をログに記録するようにデバッグログが設定されている REST API レベル 0 アプリケーションの例に基づいています。

    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5章 アプリケーションのモニターリング

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ーリングを説明します。

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

5.1.1. 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量データにアクセスできます。Jolokia は、REST インターフェイスおよび 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ー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

- 認証された **oc** クライアント。
- 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. プロジェクト内の Pod のデプロイメント設定をリストし、アプリケーションに対応するものを選択します。

```
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している 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

5.2. 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:** という接頭辞が付けられ、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。

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、リソースおよび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使用して、アプリケーションの最新バージョンを実稼働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 つの要素を指定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、アプリケーションを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および依存関係を 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の 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。たとえば、OpenShift Maven プラグインを使用する例では、YAML ファイルは **src/main/jkube/** ディレクトリーにあります。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

OpenShift Maven プラグインおよび Nodeshift が使用するデプロイメント設定ファイルは完全な OpenShift リソース定義である必要はありません。OpenShift Maven プラグインと Nodeshift はどちらもデプロイメント設定ファイルを取り、不足している情報を追加して完全な OpenShift リソース定義を作成できます。OpenShift Maven プラグインによって生成されたリソースの定義は **target/classes/META-INF/jkube/** ディレクトリーにあります。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 ファイルを編集するか、または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

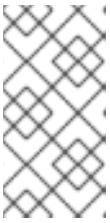
Web ベースのコンソールまたは CLI クライアント **oc** を使用してアプリケーションの設定を直接更新している場合は、これらの変更を YAML ファイルへエクスポートして追加します。**oc export all** コマンドを使用して、デプロイされたアプリケーションの設定を表示します。

付録C OPENSHIFT MAVEN プラグインでアプリケーションをデプロイする JENKINS フリースタイルプロジェクトの設定

ローカルホストの Maven および OpenShift Maven プラグインを使用してアプリケーションをデプロイするのと同様に、Jenkins を設定して Maven および OpenShift Maven プラグインを使用してアプリケーションをデプロイすることができます。

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。
- Maven (**pom.xml** の OpenShift Maven プラグイン) を使用して、Red Hat ベースイメージを使用してビルドするように設定されたアプリケーション。



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 4.3 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>
  ...
  <jkube.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</jkube.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** をクリックして、新しい 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ている場合は、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 targets** を選択します。

e. 以下を **Goals** に追加します。

```
clean oc:deploy -Popenshift -Djkube.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。

a. **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 プロジェクトに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

ネットワーク負荷を軽減し、アプリケーションのビルド時間を短縮するには、OpenShift Container Platform で Maven の Nexus ミラーをセットアップします。

- Maven 用の Nexus ミラーリングの設定