Red Hat build of Eclipse Vert.x 4.2

Eclipse Vert.x Runtime Guide

Use Eclipse Vert.x to develop reactive, non-blocking, asynchronous applications that run on OpenShift and on stand-alone RHEL

Last Updated: 2022-06-22

Red Hat build of Eclipse Vert.x 4.2 Eclipse Vert.x Runtime Guide

Use Eclipse Vert.x to develop reactive, non-blocking, asynchronous applications that run on OpenShift and on stand-alone RHEL

Legal Notice

Abstract

This guide provides details on using the Eclipse Vert.x runtime.

Table of Contents

PREFACE

This guide covers concepts as well as practical details needed by developers to use the Eclipse Vert.x runtime.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments.

This section explains how to submit feedback.

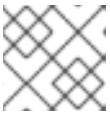
Prerequisites

- You are logged in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in **Multi-page HTML** format.

Procedure

To provide your feedback, perform the following steps:

1. Click the **Feedback** button in the top-right corner of the document to see existing feedback.



NOTE

The feedback feature is enabled only in the **Multi-page HTML** format.

2. Highlight the section of the document where you want to provide feedback.
3. Click the **Add Feedback** pop-up that appears near the highlighted text.
   A text box appears in the feedback section on the right side of the page.
4. Enter your feedback in the text box and click **Submit**.
   A documentation issue is created.
5. To view the issue, click the issue tracker link in the feedback view.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. INTRODUCTION TO APPLICATION DEVELOPMENT WITH ECLIPSE VERT.X

This section explains the basic concepts of application development with Red Hat runtimes. It also provides an overview about the Eclipse Vert.x runtime.

1.1. OVERVIEW OF APPLICATION DEVELOPMENT WITH RED HAT RUNTIMES

Red Hat OpenShift is a container application platform, which provides a collection of cloud-native runtimes. You can use the runtimes to develop, build, and deploy Java or JavaScript applications on OpenShift.

Application development using Red Hat Runtimes for OpenShift includes:

- A collection of runtimes, such as, Eclipse Vert.x, Thorntail, Spring Boot, and so on, designed to run on OpenShift.
- A prescriptive approach to cloud-native development on OpenShift.

OpenShift helps you manage, secure, and automate the deployment and monitoring of your applications. You can break your business problems into smaller microservices and use OpenShift to deploy, monitor, and maintain the microservices. You can implement patterns such as circuit breaker, health check, and service discovery, in your applications.

Cloud-native development takes full advantage of cloud computing.

You can build, deploy, and manage your applications on:

OpenShift Container Platform

A private on-premise cloud by Red Hat.

Red Hat CodeReady Studio

An integrated development environment (IDE) for developing, testing, and deploying applications.

This guide provides detailed information about the Eclipse Vert.x runtime. For more information on other runtimes, see the relevant runtime documentation.

1.2. OVERVIEW OF ECLIPSE VERT.X

Eclipse Vert.x is a toolkit used for creating reactive, non-blocking, and asynchronous applications that run on the Java Virtual Machine (JVM).

Eclipse Vert.x is designed to be cloud-native. It allows applications to use very few threads. This avoids the overhead caused when new threads are created. This enables Eclipse Vert.x applications and services to effectively use their memory as well as CPU quotas in cloud environments.

Using the Eclipse Vert.x runtime in OpenShift makes it simpler and easier to build reactive systems. The OpenShift platform features, such as, rolling updates, service discovery, and canary deployments, are also available. With OpenShift, you can implement microservice patterns, such as externalized configuration, health check, circuit breaker, and failover, in your applications.

1.2.1. Key concepts of Eclipse Vert.x

This section describes some key concepts associated with the Eclipse Vert.x runtime. It also provides a brief overview of reactive systems.

Cloud and Container-Native Applications

Cloud-native applications are typically built using microservices. They are designed to form distributed systems of decoupled components. These components usually run inside containers, on top of clusters that contain a large number of nodes. These applications are expected to be resistant to the failure of individual components, and may be updated without requiring any service downtime. Systems based on cloud-native applications rely on automated deployment, scaling, and administrative and maintenance tasks provided by an underlying cloud platform, such as, OpenShift. Management and administration tasks are carried out at the cluster level using off-the-shelf management and orchestration tools, rather than on the level of individual machines.

Reactive Systems

A reactive system, as defined in the reactive manifesto, is a distributed systems with the following characteristics:

Elastic

The system remains responsive under varying workload, with individual components scaled and load-balanced as necessary to accommodate the differences in workload. Elastic applications deliver the same quality of service regardless of the number of requests they receive at the same time.

Resilient

The system remains responsive even if any of its individual components fail. In the system, the components are isolated from each other. This helps individual components to recover quickly in case of failure. Failure of a single component should never affect the functioning of other components. This prevents cascading failure, where the failure of an isolated component causes other components to become blocked and gradually fail.

Responsive

Responsive systems are designed to always respond to requests in a reasonable amount of time to ensure a consistent quality of service. To maintain responsiveness, the communication channel between the applications must never be blocked.

Message-Driven

The individual components of an application use asynchronous message-passing to communicate with each other. If an event takes place, such as a mouse click or a search query on a service, the service sends a message on the common channel, that is, the event bus. The messages are in turn caught and handled by the respective component.

Reactive Systems are distributed systems. They are designed so that their asynchronous properties can be used for application development.

Reactive Programming

While the concept of reactive systems describes the architecture of a distributed system, reactive programming refers to practices that make applications reactive at the code level. Reactive programming is a development model to write asynchronous and event-driven applications. In reactive applications, the code reacts to events or messages.

There are several implementations of reactive programming. For example, simple implementations using callbacks, complex implementations using Reactive Extensions (Rx), and coroutines.

The Reactive Extensions (Rx) is one of the most mature forms of reactive programming in Java. It uses the *RxJava* library.

1.2.2. Supported Architectures by Eclipse Vert.x

Eclipse Vert.x supports the following architectures:

- x86_64 (AMD64)
- IBM Z (s390x) in the OpenShift environment
- IBM Power Systems (ppc64le) in the OpenShift environment

Refer to the section Supported Java images for Eclipse Vert.x for more information about the image names.

1.2.3. Support for Federal Information Processing Standard (FIPS)

The Federal Information Processing Standards (FIPS) provides guidelines and requirements for improving security and interoperability across computer systems and networks. The FIPS 140-2 and 140-3 series apply to cryptographic modules at both the hardware and software levels.

The Federal Information Processing Standard (FIPS) Publication 140-2 is a computer security standard developed by the U.S. Government and industry working group to validate the quality of cryptographic modules. See the official FIPS publications at NIST Computer Security Resource Center .

Red Hat Enterprise Linux (RHEL) provides an integrated framework to enable FIPS 140-2 compliance system-wide. When operating in the FIPS mode, software packages using cryptographic libraries are self-configured according to the global policy.

To learn about compliance requirements, see the Red Hat Government Standards page.

Red Hat build of Eclipse Vert.x runs on a FIPS enabled RHEL system and uses FIPS certified libraries provided by RHEL.

1.2.3.1. Additional resources

- For more information on how to install RHEL with FIPS mode enabled, see Installing a RHEL 8 system with FIPS mode enabled.
- For more information on how to enable FIPS mode after installing RHEL, see Switching the system to FIPS mode.

CHAPTER 2. CONFIGURING YOUR APPLICATIONS

This section explains how to configure your applications to work with Eclipse Vert.x runtime.

2.1. CONFIGURING YOUR APPLICATION TO USE ECLIPSE VERT.X

When you start configuring your applications to use Eclipse Vert.x, you must reference the Eclipse Vert.x BOM (Bill of Materials) artifact in the **pom.xml** file at the root directory of your application. The BOM is used to set the correct versions of the artifacts.

Prerequisites

- A Maven-based application

Procedure

1. Open the **pom.xml** file, add the **io.vertx:vertx-dependencies** artifact to the **<dependencyManagement>** section. Specify the **type** as **pom** and **scope** as **import**.

```
<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. Include the following properties to track the version of Eclipse Vert.x and the Eclipse Vert.x Maven Plugin you are using.
   Properties can be used to set values that change in every release. For example, versions of product or plugins.

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

3. Specify **vertx-maven-plugin** as the plugin used to package your application:

```
<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. Include **repositories** and **pluginRepositories** to specify the repositories that contain the artifacts and plugins to build your application:

```
<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>
```

Additional resources

- For more information about packaging your Eclipse Vert.x application, see the Vert.x Maven Plugin documentation.

CHAPTER 3. DEVELOPING AND DEPLOYING ECLIPSE VERT.X RUNTIME APPLICATION

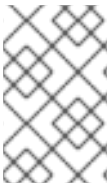
You can create a new Eclipse Vert.x application and deploy it to OpenShift or stand-alone Red Hat Enterprise Linux.

3.1. DEVELOPING ECLIPSE VERT.X APPLICATION

For a basic Eclipse Vert.x application, you need to create the following:

- A Java class containing Eclipse Vert.x methods.
- A **pom.xml** file containing information required by Maven to build the application.

The following procedure creates a simple **Greeting** application that returns "Greetings!" as response.



NOTE

For building and deploying your applications to OpenShift, Eclipse Vert.x 4.2 only supports builder images based on OpenJDK 8 and OpenJDK 11. Oracle JDK and OpenJDK 9 builder images are not supported.

Prerequisites

- OpenJDK 8 or OpenJDK 11 installed.
- Maven installed.

Procedure

1. Create a new directory **myApp**, and navigate to it.

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

   This is the root directory for the application.

2. Create directory structure **src/main/java/com/example/** in the root directory, and navigate to it.

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

3. Create a Java class file **MyApp.java** containing the application code.

```
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. Create a **pom.xml** file in the application root directory **myApp** with the following content:

```
<?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>4.2.7.redhat-00003</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. Build the application using Maven from the root directory of the application.

```
$ mvn vertx:run
```

6. Verify that the application is running.
   Using **curl** or your browser, verify your application is running at **http://localhost:8080**.

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

Additional information

- As a recommended practice, you can configure liveness and readiness probes to enable health monitoring for your application when running on OpenShift.

3.2. DEPLOYING ECLIPSE VERT.X APPLICATION TO OPENSHIFT

To deploy your Eclipse Vert.x application to OpenShift, configure the **pom.xml** file in your application and then use the OpenShift Maven plugin.

NOTE

The Fabric8 Maven plugin is no longer supported. Use the OpenShift Maven plugin to deploy your Eclipse Vert.x applications on OpenShift. For more information, see the section migrating from Fabric8 Maven Plugin to Eclipse JKube .

You can specify a Java image by replacing the **jkube.generator.from** URL in the **pom.xml** file. The images are available in the Red Hat Ecosystem Catalog .

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

For example, the Java image for RHEL 7 with OpenJDK 8 is specified as:

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

3.2.1. Supported Java images for Eclipse Vert.x

Eclipse Vert.x is certified and tested with various Java images that are available for different operating systems. For example, Java images are available for RHEL 7 with OpenJDK 8 or OpenJDK 11.

Eclipse Vert.x introduces support for building and deploying Eclipse Vert.x applications to OpenShift with OCI-compliant Universal Base Images for Red Hat OpenJDK 8 and Red Hat OpenJDK 11 on RHEL 8.

You require Docker or podman authentication to access the RHEL 8 images in the Red Hat Ecosystem Catalog.

The following table lists the container images supported by Eclipse Vert.x for different architectures. These container images are available in the Red Hat Ecosystem Catalog. In the catalog, you can search and download the images listed in the table below. The image pages contain authentication procedures required to access the images.

Table 3.1. OpenJDK images and architectures

JDK (OS)	Architecture supported	Images available in Red Hat Ecosystem Catalog
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, IBM Z, and IBM Power Systems	ubi8/openjdk-11

NOTE

The use of a RHEL 8-based container on a RHEL 7 host, for example with OpenShift 3 or OpenShift 4, has limited support. For more information, see the Red Hat Enterprise Linux Container Compatibility Matrix.

3.2.2. Preparing Eclipse Vert.x application for OpenShift deployment

For deploying your Eclipse Vert.x application to OpenShift, it must contain:

- Launcher profile information in the application's **pom.xml** file.

In the following procedure, a profile with OpenShift Maven plugin is used for building and deploying the application to OpenShift.

Prerequisites

- Maven is installed.
- Docker or podman authentication into Red Hat Ecosystem Catalog to access RHEL 8 images.

Procedure

1. Add the following content to the **pom.xml** file in the application root directory:

```
<!-- 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. Replace the **jkube.generator.from** property in the **pom.xml** file to specify the OpenJDK image that you want to use.

- x86_64 architecture
  - RHEL 7 with OpenJDK 8

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

  - RHEL 7 with OpenJDK 11

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

  - RHEL 8 with OpenJDK 8

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

- x86_64, s390x (IBM Z), and ppc64le (IBM Power Systems) architectures
  - RHEL 8 with OpenJDK 11

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

3.2.3. Deploying Eclipse Vert.x application to OpenShift using OpenShift Maven plugin

To deploy your Eclipse Vert.x application to OpenShift, you must perform the following:

- Log in to your OpenShift instance.
- Deploy the application to the OpenShift instance.

Prerequisites

Prerequisites

- **oc** CLI client installed.
- Maven installed.

Procedure

1. Log in to your OpenShift instance with the **oc** client.

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

2. Create a new project in the OpenShift instance.

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

3. Deploy the application to OpenShift using Maven from the application's root directory. The root directory of an application contains the **pom.xml** file.

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

   This command uses the OpenShift Maven plugin to launch the S2I process on OpenShift and start the pod.

4. Verify the deployment.

   a. Check the status of your application and ensure your pod is running.

```
$ 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
```

   The **MY_APP_NAME-1-aaaaa** pod should have a status of **Running** once it is fully deployed and started.

   Your specific pod name will vary.

   b. Determine the route for the pod.

   Example Route Information

```
$ 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
```

   The route information of a pod gives you the base URL which you use to access it.

   In this example, **http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** is the base URL to access the application.

   c. Verify that your application is running in OpenShift.

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

3.3. DEPLOYING ECLIPSE VERT.X APPLICATION TO STAND-ALONE RED HAT ENTERPRISE LINUX

To deploy your Eclipse Vert.x application to stand-alone Red Hat Enterprise Linux, configure the **pom.xml** file in the application, package it using Maven and deploy using the **java -jar** command.

Prerequisites

- RHEL 7 or RHEL 8 installed.

3.3.1. Preparing Eclipse Vert.x application for stand-alone Red Hat Enterprise Linux deployment

For deploying your Eclipse Vert.x application to stand-alone Red Hat Enterprise Linux, you must first package the application using Maven.

Prerequisites

- Maven installed.

Procedure

1. Add the following content to the **pom.xml** file in the application's root directory:

```
...
<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. Package your application using Maven.

```
$ mvn clean package
```

The resulting JAR file is in the **target** directory.

3.3.2. Deploying Eclipse Vert.x application to stand-alone Red Hat Enterprise Linux using jar

To deploy your Eclipse Vert.x application to stand-alone Red Hat Enterprise Linux, use **java -jar** command.

Prerequisites

- RHEL 7 or RHEL 8 installed.
- OpenJDK 8 or OpenJDK 11 installed.
- A JAR file with the application.

Procedure

1. Deploy the JAR file with the application.

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

2. Verify the deployment.
   Use **curl** or your browser to verify your application is running at **http://localhost:8080**:

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

CHAPTER 4. DEBUGGING ECLIPSE VERT.X BASED APPLICATION

This sections contains information about debugging your Eclipse Vert.x–based application both in local and remote deployments.

4.1. REMOTE DEBUGGING

To remotely debug an application, you must first configure it to start in a debugging mode, and then attach a debugger to it.

4.1.1. Starting your application locally in debugging mode

One of the ways of debugging a Maven-based project is manually launching the application while specifying a debugging port, and subsequently connecting a remote debugger to that port. This method is applicable at least to the following deployments of the application:

- When launching the application manually using the **mvn vertx:debug** goal. This starts the application with debugging enabled.

Prerequisites

- A Maven-based application

Procedure

1. In a console, navigate to the directory with your application.
2. Launch your application and specify the debug port using the **-Ddebug.port** argument:

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

Here, **$PORT_NUMBER** is an unused port number of your choice. Remember this number for the remote debugger configuration.

Use the **-Ddebug.suspend=true** argument to make the application wait until a debugger is attached to start.

4.1.2. Starting your application on OpenShift in debugging mode

To debug your Eclipse Vert.x-based application on OpenShift remotely, you must set the **JAVA_DEBUG** environment variable inside the container to **true** and configure port forwarding so that you can connect to your application from a remote debugger.

Prerequisites

- Your application running on OpenShift.
- The **oc** binary installed.
- The ability to execute the **oc port-forward** command in your target OpenShift environment.

Procedure

1. Using the **oc** command, list the available deployment configurations:

```
$ oc get dc
```

2. Set the **JAVA_DEBUG** environment variable in the deployment configuration of your application to **true**, which configures the JVM to open the port number **5005** for debugging. For example:

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

3. Redeploy the application if it is not set to redeploy automatically on configuration change. For example:

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

4. Configure port forwarding from your local machine to the application pod:

   a. List the currently running pods and find one containing your application:

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

   b. Configure port forwarding:

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

   Here, **$LOCAL_PORT_NUMBER** is an unused port number of your choice on your local machine. Remember this number for the remote debugger configuration.

5. When you are done debugging, unset the **JAVA_DEBUG** environment variable in your application pod. For example:

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

Additional resources

You can also set the **JAVA_DEBUG_PORT** environment variable if you want to change the debug port from the default, which is **5005**.

4.1.3. Attaching a remote debugger to the application

When your application is configured for debugging, attach a remote debugger of your choice to it. In this guide, Red Hat CodeReady Studio is covered, but the procedure is similar when using other programs.

Prerequisites

- The application running either locally or on OpenShift, and configured for debugging.
- The port number that your application is listening on for debugging.
- Red Hat CodeReady Studio installed on your machine. You can download it from the Red Hat CodeReady Studio download page.

Procedure

1. Start Red Hat CodeReady Studio.
2. Create a new debug configuration for your application:
   a. Click **Run→Debug Configurations**.
   b. In the list of configurations, double-click **Remote Java application.** This creates a new remote debugging configuration.
   c. Enter a suitable name for the configuration in the **Name** field.
   d. Enter the path to the directory with your application into the **Project** field. You can use the **Browse...** button for convenience.
   e. Set the **Connection Type** field to *Standard (Socket Attach)* if it is not already.
   f. Set the **Port** field to the port number that your application is listening on for debugging.
   g. Click **Apply.**
3. Start debugging by clicking the **Debug** button in the Debug Configurations window.
   To quickly launch your debug configuration after the first time, click **Run→Debug History** and select the configuration from the list.

Additional resources

- Debug an OpenShift Java Application with JBoss Developer Studio on Red Hat Knowledgebase.
  Red Hat CodeReady Studio was previously called JBoss Developer Studio.
- A Debugging Java Applications On OpenShift and Kubernetes article on OpenShift Blog.

4.2. DEBUG LOGGING

Eclipse Vert.x provides a built-in logging API. The default logging implementation for Eclipse Vert.x uses the **java.util.logging** library that is provided with the Java JDK. Alternatively, Eclipse Vert.x allows you to use a different logging framework, for example, *Log4J* (Eclipse Vert.x supports Log4J v1 and v2) or *SLF4J*.

4.2.1. Configuring logging for your Eclipse Vert.x application using java.util.logging

To configure debug logging for your Eclipse Vert.x application using **java.util.logging**:

- Set the **java.util.logging.config.file** system property in the **application.properties** file. The value of this variable must correspond to the name of your **java.util.logging** configuration file. This ensures that **LogManager** initializes **java.util.logging** at application startup.
- Alternatively, add a **java.util.logging** configuration file with the **vertx-default-jul-logging.properties** name to the classpath of your Maven project. Eclipse Vert.x will use that file to configure **java.util.logging** on application startup.

Eclipse Vert.x allows you to specify a custom logging backend using the **LogDelegateFactory** that provides pre-built implementations for the **Log4J**, **Log4J2** and **SLF4J** libraries. Unlike **java.util.logging**, which is included with Java by default, the other backends require that you specify their respective

libraries as dependencies for your application.

4.2.2. Adding log output to your Eclipse Vert.x application.

1. To add logging to your application, create a **io.vertx.core.logging.Logger**:

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

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

CAUTION

Logging backends use different formats to represent replaceable tokens in parameterized messages. If you rely on parameterized logging methods, you will not be able to switch logging backends without changing your code.

4.2.3. Specifying a custom logging framework for your application

If you do not want Eclipse Vert.x to use **java.util.logging**, configure **io.vertx.core.logging.Logger** to use a different logging framework, for example, **Log4J** or **SLF4J**:

1. Set the value of the **vertx.logger-delegate-factory-class-name** system property to the name of the class that implements the **LogDelegateFactory** interface. Eclipse Vert.x provides the pre-built implementations for the following libraries with their corresponding pre-defined classnames listed below:

Library	Class name
Log4J v1	**io.vertx.core.logging.Log4jLogDelegateFactory**
Log4J v2	**io.vertx.core.logging.Log4j2LogDelegateFactory**
SLF4J	**io.vertx.core.logging.SLF4JLogDelegateFactory**

When implementing logging using a custom library, ensure that the relevant **Log4J** or **SLF4J** jars are included among the dependencies for your application.

CAUTION

The *Log4J* v1 delegate provided with Eclipse Vert.x does not support parameterized messages. The delegates for *Log4J v2* and *SLF4J* both use the **{}** syntax. The **java.util.logging** delegate relies on **java.text.MessageFormat** that uses the **{n}** syntax.

4.2.4. Configuring Netty logging for your Eclipse Vert.x application.

Netty is a library used by VertX to manage asynchronous network communication in applications.

Netty:

- Allows quick and easy development of network applications, such as protocol servers and clients.
- Simplifies and streamlines network programming, such as TCP and UDP socket server development.
- Provides a unified API for managing blocking and non-blocking connections.

Netty does not rely on an external logging configuration using system properties. Instead, it implements a logging configuration based on logging libraries visible to Netty classes in your project. Netty tries to use the libraries in the following order:

1. **SLF4J**
2. **Log4J**
3. **java.util.logging** as a fallback option

You can set **io.netty.util.internal.logging.InternalLoggerFactory** directly to a particular logger by adding the following code at the beginning of the **main** method of your application:

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

4.2.5. Accessing debug logs on OpenShift

Start your application and interact with it to see the debugging statements in OpenShift.

Prerequisites

- The **oc** CLI client installed and authenticated.
- A Maven-based application with debug logging enabled.

Procedure

1. Deploy your application to OpenShift:

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

2. View the logs:
    1. Get the name of the pod with your application:

```
$ oc get pods
```

    2. Start watching the log output:

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

    Keep the terminal window displaying the log output open so that you can watch the log output.

3. Interact with your application:
   For example, the following command is based on an example REST API level 0 application where debug logging is set to log the **message** variable in the **/api/greeting** method:

   1. Get the route of your application:

      ```
      $ oc get routes
      ```

   2. Make an HTTP request on the **/api/greeting** endpoint of your application:

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

4. Return to the window with your pod logs and inspect debug logging messages in the logs.

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

5. To disable debug logging, update your logging configuration file, for example **src/main/resources/vertx-default-jul-logging.properties**, remove the logging configuration for your class and redeploy your application.

CHAPTER 5. MONITORING YOUR APPLICATION

This section contains information about monitoring your Eclipse Vert.x–based application running on OpenShift.

5.1. ACCESSING JVM METRICS FOR YOUR APPLICATION ON OPENSHIFT

5.1.1. Accessing JVM metrics using Jolokia on OpenShift

Jolokia is a built-in lightweight solution for accessing JMX (Java Management Extension) metrics over HTTP on OpenShift. Jolokia allows you to access CPU, storage, and memory usage data collected by JMX over an HTTP bridge. Jolokia uses a REST interface and JSON-formatted message payloads. It is suitable for monitoring cloud applications thanks to its comparably high speed and low resource requirements.

For Java-based applications, the OpenShift Web console provides the integrated *hawt.io* console that collects and displays all relevant metrics output by the JVM running your application.

Prerequistes

- the **oc** client authenticated
- a Java-based application container running in a project on OpenShift
- latest JDK 1.8.0 image

Procedure

1. List the deployment configurations of the pods inside your project and select the one that corresponds to your application.

```
oc get dc
```

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

2. Open the YAML deployment template of the pod running your application for editing.

```
oc edit dc/MY_APP_NAME
```

3. Add the following entry to the **ports** section of the template and save your changes:

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

4. Redeploy the pod running your application.

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

The pod is redeployed with the updated deployment configuration and exposes the port **8778**.

5. Log into the OpenShift Web console.
6. In the sidebar, navigate to *Applications* > *Pods*, and click on the name of the pod running your application.
7. In the pod details screen, click *Open Java Console* to access the hawt.io console.

Additional resources

- hawt.io documentation

5.2. EXPOSING APPLICATION METRICS USING PROMETHEUS WITH ECLIPSE VERT.X

Prometheus connects to a monitored application to collect data; the application does not send metrics to a server.

Prerequisites

- Prometheus server running on your cluster

Procedure

1. Include the **vertx-micrometer** and **vertx-web** dependencies in the **pom.xml** file of your application:

pom.xml

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

2. Starting with version 3.5.4, exposing metrics for Prometheus requires that you configure the Eclipse Vert.x options in a custom **Launcher** class.
In your custom **Launcher** class, override the **beforeStartingVertx** and **afterStartingVertx** methods to configure the metrics engine, for example:

Example CustomLauncher.java file

```
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. Create a custom **Verticle** class and override the **start** method to collect metrics. For example, measure the execution time using the **Timer** class:

Example CustomVertxApp.java file

```
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. Set the **<vertx.verticle>** and **<vertx.launcher>** properties in the **pom.xml** file of your application to point to your custom classes:

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

5. Launch your application:

```
$ mvn vertx:run
```

6. Invoke the traced endpoint several times:

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

7. Wait at least 15 seconds for collection to occur, and see the metrics in Prometheus UI:

    1. Open the Prometheus UI at http://localhost:9090/ and type **hello** into the *Expression* box.

    2. From the suggestions, select for example **application:hello_count** and click *Execute*.

    3. In the table that is displayed, you can see how many times the resource method was invoked.

    4. Alternatively, select **application:hello_time_mean_seconds** to see the mean time of all the invocations.

   Note that all metrics you created are prefixed with **application:**. There are other metrics, automatically exposed by Eclipse Vert.x as the Eclipse MicroProfile Metrics specification requires. Those metrics are prefixed with **base:** and **vendor:** and expose information about the JVM in which the application runs.

Additional resources

- For additional information about using Micrometer metrics with Eclipse Vert.x, see Eclipse Vert.x} Micrometer Metrics.

APPENDIX A. THE SOURCE-TO-IMAGE (S2I) BUILD PROCESS

Source-to-Image (S2I) is a build tool for generating reproducible Docker-formatted container images from online SCM repositories with application sources. With S2I builds, you can easily deliver the latest version of your application into production with shorter build times, decreased resource and network usage, improved security, and a number of other advantages. OpenShift supports multiple build strategies and input sources.

For more information, see the Source-to-Image (S2I) Build chapter of the OpenShift Container Platform documentation.

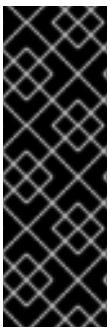
You must provide three elements to the S2I process to assemble the final container image:

- The application sources hosted in an online SCM repository, such as GitHub.
- The S2I Builder image, which serves as the foundation for the assembled image and provides the ecosystem in which your application is running.
- Optionally, you can also provide environment variables and parameters that are used by S2I scripts.

The process injects your application source and dependencies into the Builder image according to instructions specified in the S2I script, and generates a Docker-formatted container image that runs the assembled application. For more information, check the S2I build requirements, build options and how builds work sections of the OpenShift Container Platform documentation.

APPENDIX B. UPDATING THE DEPLOYMENT CONFIGURATION OF AN EXAMPLE APPLICATION

The deployment configuration for an example application contains information related to deploying and running the application in OpenShift, such as route information or readiness probe location. The deployment configuration of an example application is stored in a set of YAML files. For examples that use the OpenShift Maven plugin, the YAML files are located in the **src/main/jkube/** directory. For examples using Nodeshift, the YAML files are located in the **.nodeshift** directory.



IMPORTANT

The deployment configuration files used by the OpenShift Maven plugin and Nodeshift do not have to be full OpenShift resource definitions. Both OpenShift Maven plugin and Nodeshift can take the deployment configuration files and add some missing information to create a full OpenShift resource definition. The resource definitions generated by the OpenShift Maven plugin are available in the **target/classes/META-INF/jkube/** directory. The resource definitions generated by Nodeshift are available in the **tmp/nodeshift/resource/** directory.

Prerequisites

- An existing example project.
- The **oc** CLI client installed.

Procedure

1. Edit an existing YAML file or create an additional YAML file with your configuration update.
   - For example, if your example already has a YAML file with a **readinessProbe** configured, you could change the **path** value to a different available path to check for readiness:

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - If a **readinessProbe** is not configured in an existing YAML file, you can also create a new YAML file in the same directory with the **readinessProbe** configuration.
2. Deploy the updated version of your example using Maven or npm.
3. Verify that your configuration updates show in the deployed version of your example.

```
$ 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
          ...
```

Additional resources

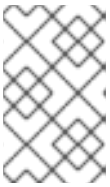
If you updated the configuration of your application directly using the web-based console or the **oc** CLI client, export and add these changes to your YAML file. Use the **oc export all** command to show the configuration of your deployed application.

APPENDIX C. CONFIGURING A JENKINS FREESTYLE PROJECT TO DEPLOY YOUR APPLICATION WITH THE OPENSHIFT MAVEN PLUGIN

Similar to using Maven and the OpenShift Maven plugin from your local host to deploy an application, you can configure Jenkins to use Maven and the OpenShift Maven plugin to deploy an application.

Prerequisites

- Access to an OpenShift cluster.
- The Jenkins container image running on same OpenShift cluster.
- A JDK and Maven installed and configured on your Jenkins server.
- An application configured to use Maven, the OpenShift Maven plugin in the **pom.xml**, and built using a RHEL base image.



NOTE

For building and deploying your applications to OpenShift, Eclipse Vert.x 4.2 only supports builder images based on OpenJDK 8 and OpenJDK 11. Oracle JDK and OpenJDK 9 builder images are not supported.

Example pom.xml

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

- The source of the application available in GitHub.

Procedure

1. Create a new OpenShift project for your application:
   a. Open the OpenShift Web console and log in.
   b. Click *Create Project* to create a new OpenShift project.
   c. Enter the project information and click *Create*.
2. Ensure Jenkins has access to that project.
   For example, if you configured a service account for Jenkins, ensure that account has **edit** access to the project of your application.
3. Create a new freestyle Jenkins project on your Jenkins server:
   a. Click *New Item*.
   b. Enter a name, choose *Freestyle project*, and click *OK*.

c. Under *Source Code Management*, choose *Git* and add the GitHub url of your application.

d. Under *Build*, choose *Add build step* and select **Invoke top-level Maven targets**.

e. Add the following to *Goals*:

```
clean oc:deploy -Popenshift -Djkube.namespace=MY_PROJECT
```

Substitute **MY_PROJECT** with the name of the OpenShift project for your application.

a. Click *Save*.

4. Click *Build Now* from the main page of the Jenkins project to verify your application builds and deploys to the OpenShift project for your application.
You can also verify that your application is deployed by opening the route in the OpenShift project of the application.

Next steps

- Consider adding GITSCM polling or using the **Poll SCM** build trigger. These options enable builds to run every time a new commit is pushed to the GitHub repository.
- Consider adding a build step that executes tests before deploying.

APPENDIX D. ADDITIONAL ECLIPSE VERT.X RESOURCES

- The Reactive Manifesto
- Eclipse Vert.x project
- Vert.x in Action
- 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
- Live Coding Reactive Systems with Eclipse Vert.x and OpenShift

APPENDIX E. APPLICATION DEVELOPMENT RESOURCES

For additional information about application development with OpenShift, see:

- OpenShift Interactive Learning Portal

To reduce network load and shorten the build time of your application, set up a Nexus mirror for Maven on your OpenShift Container Platform:

- Setting Up a Nexus Mirror for Maven