Red Hat build of Eclipse Vert.x 4.2

Getting started with Eclipse Vert.x

For use with Eclipse Vert.x 4.2.7

Last Updated: 2022-06-22

Red Hat build of Eclipse Vert.x 4.2 Getting started with Eclipse Vert.x

For use with Eclipse Vert.x 4.2.7

Legal Notice

Abstract

This guide describes how to create a simple Eclipse Vert.x application with Apache Maven.

Table of Contents

CHAPTER 1. PREREQUISITES TO GET STARTED

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

As an application developer, you can use Eclipse Vert.x to create microservices-based applications written in Java that run in OpenShift environments.

This guide shows you how to create, package, run and test a simple Eclipse Vert.x project.

Prerequisites

- OpenJDK 8 or OpenJDK 11 is installed and the **JAVA_HOME** environment variable specifies the location of the Java SDK. Log in to the Red Hat Customer Portal to download Red Hat build of Open JDK from the Software Downloads.
- Apache Maven 3.6.0 or higher is installed. You can download Maven from the Apache Maven Project website.

CHAPTER 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.

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.

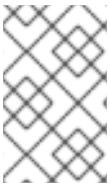
The Reactive Extensions (Rx) is one of the most mature forms of reactive programming in Java. It uses the *RxJava* library.

CHAPTER 3. CREATING AN ECLIPSE VERT.X PROJECT WITH A POM FILE

When you develop a basic Eclipse Vert.x application, you should create the following artifacts. We will create these artifacts in our first **getting-started** Eclipse Vert.x project.

- 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

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

Prerequisites

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

Procedure

1. Create a new directory **getting-started**, and navigate to it.

```
$ mkdir getting-started
$ cd getting-started
```

   This is the root directory for the application.

2. Create a 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.

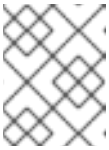
```
package com.example;

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());
                }
            });
    }
}
```

The application starts an HTTP Server on port 8080. When you send a request, it returns **Greetings!** message.

4. Create a **pom.xml** file in the application root directory **getting-started** with the following content:

- In the **<dependencyManagement>** section, add the **io.vertx:vertx-dependencies** artifact.
- Specify the **type** as **pom** and **scope** as **import**.
- In the **<project>** section, under **<properties>**, specify the versions of Eclipse Vert.x and the Eclipse Vert.x Maven Plugin.



NOTE

Properties can be used to set values that change in every release. For example, versions of product or plugins.

- In the **<project>** section, under **<plugin>**, specify **vertx-maven-plugin**. The Eclipse Vert.x Maven Plugin is used to package your application.
- Include **repositories** and **pluginRepositories** to specify the repositories that contain the artifacts and plugins to build your application.
  The **pom.xml** contains the following artifacts:

```
<?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>
```

```
  <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>

  <!-- Test dependencies -->
  <dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-junit5</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.4.0</version>
    <scope>test</scope>
  </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 version of the Maven Surefire plugin. -->
  <build>
```

```
    <plugins>
     <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
     </plugin>
     <plugin>

<!-- Configure your application to be packaged using the Vert.x Maven 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.
   Use **curl** or your browser to verify if your application is running at **http://localhost:8080** and returns **"Greetings!"** as response.

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

CHAPTER 4. TESTING YOUR ECLIPSE VERT.X APPLICATION WITH JUNIT

After you build your Eclipse Vert.x application in the **getting-started** project, test your application with the JUnit 5 framework to ensure that it runs as expected. The following two dependencies in the Eclipse Vert.x **pom.xml** file are used for JUnit 5 testing:

```
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-junit5</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.4.0</version>
    <scope>test</scope>
  </dependency>
```

- The **vertx-junit5** dependency is required for testing. JUnit 5 provides various annotations, such as, **@Test**, **@BeforeEach**, **@DisplayName**, and so on which are used to request asynchronous injection of **Vertx** and **VertxTestContext** instances.
- The **junit-jupiter-engine** dependency is required for execution of tests at runtime.

Prerequisites

- You have built the Eclipse Vert.x **getting-started** project using the **pom.xml** file.

Procedure

1. Open the generated **pom.xml** file and set the version of the Surefire Maven plug-in:

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
</plugin>
```

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

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

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

```
package com.example;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
```

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import io.vertx.core.Vertx;
import io.vertx.core.http.HttpMethod;
import io.vertx.junit5.VertxExtension;
import io.vertx.junit5.VertxTestContext;

@ExtendWith(VertxExtension.class)
class MyAppTest {

  @BeforeEach
  void prepare(Vertx vertx, VertxTestContext testContext) {
    // Deploy the verticle
    vertx.deployVerticle(new MyApp())
      .onSuccess(ok -> testContext.completeNow())
      .onFailure(failure -> testContext.failNow(failure));
  }

  @Test
  @DisplayName("Smoke test: check that the HTTP server responds")
  void smokeTest(Vertx vertx, VertxTestContext testContext) {
    // Issue an HTTP request
    vertx.createHttpClient()
      .request(HttpMethod.GET, 8080, "127.0.0.1", "/")
      .compose(request -> request.send())
      .compose(response -> response.body())
      .onSuccess(body -> testContext.verify(() -> {
        // Check the response
        assertEquals("Greetings!", body.toString());
        testContext.completeNow();
      }))
      .onFailure(failure -> testContext.failNow(failure));
  }
}
```

4. To run the JUnit test on my application using Maven run the following command from the root directory of the application.

```
mvn clean verify
```

You can check the test results in the **target/surefire-reports**. The **com.example.MyAppTest.txt** file contains the test results.

CHAPTER 5. OTHER WAYS TO CREATE ECLIPSE VERT.X PROJECTS

This section shows the different ways in which you can create Eclipse Vert.x projects.

5.1. CREATING A ECLIPSE VERT.X PROJECT ON THE COMMAND LINE

You can use the Eclipse Vert.x Maven plug-in on the command line to create a Eclipse Vert.x project. You can specify the attributes and values on the command line.

Prerequisites

- OpenJDK 8 or OpenJDK 11 is installed.
- Maven 3 or higher is installed.
- A text editor or IDE is available.
- Curl or HTTPie or a browser to perform HTTP requests is available.

Procedure

1. In a command terminal, enter the following command to verify that Maven is using OpenJDK 8 or OpenJDK 11 and the Maven version is 3.6.0 or higher:

```
mvn --version
```

2. If the preceding command does not return OpenJDK 8 or OpenJDK 11, add the path to OpenJDK 8 or OpenJDK 11 to the PATH environment variable and enter the command again.
3. Create a directory and go to the directory location.

```
mkdir getting-started && cd getting-started
```

4. Use the following command to create a new project using the Eclipse Vert.x Maven plug-in.

```
mvn io.reactiverse:vertx-maven-plugin:${vertx-maven-plugin-version}:setup -
DvertxBom=vertx-dependencies \
-DvertxVersion=${vertx_version} \
-DprojectGroupId= ${project_group_id} \
-DprojectArtifactId= ${project_artifact_id} \
-DprojectVersion=${project-version} \
-Dverticle=${verticle_class} \
-Ddependencies=${dependency_names}
```

The following example shows you how you can create an Eclipse Vert.x application using the command explained.

```
mvn io.reactiverse:vertx-maven-plugin:1.0.24:setup -DvertxBom=vertx-dependencies \
-DvertxVersion=4.2.7.redhat-00003 \
-DprojectGroupId=io.vertx.myapp \
-DprojectArtifactId=my-new-project \
-DprojectVersion=1.0-SNAPSHOT \
```

```
-DvertxVersion=4.2.7.redhat-00003 \
-Dverticle=io.vertx.myapp.MainVerticle \
-Ddependencies=web
```

The following table lists the attributes that you can define with the **setup** command:

Attribute	Default Value	Description
vertx_version	The version of Eclipse Vert.x.	The version of Eclipse Vert.x you want to use in your project.
project_group_id	**io.vertx.example**	A unique identifier of your project.
project_artifact_id	**my-vertx-project**	The name of your project and your project directory. If you do not specify the **project_artifact_id**, the Maven plug-in starts the interactive mode. If the directory already exists, the generation fails.
project-version	**1.0-SNAPSHOT**	The version of your project.
verticle_class	**io.vertx.example.MainVerticle**	The new verticle class file created by the verticle parameter.

Attribute	Default Value	Description
dependency_names	Optional parameter	The list of dependencies you want to add to your project separated by comma. You can also use the following syntax to configure the dependencies: **groupId:artifactId:version:classifier** For example: - To inherit the version from BOM use the following syntax: **io.vertx:vertxcodetrans** - To specify dependency use the following syntax: **commons-io:commons-io:2.5** - To specify dependency with a classifier use the following syntax: **io.vertx:vertx-template-engines:3.4.1:shaded**

The command creates an empty Eclipse Vert.x project with the following artifacts in the **getting-started** directory:

- The Maven build descriptor **pom.xml** configured to build and run your application
- Example verticle in the **src/main/java** folder

5. In the **pom.xml** file, specify the repositories that contain the Eclipse Vert.x artifacts to build your application.

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

Alternatively, you can configure the Maven repository to specify the build artifacts in the **settings.xml** file. See the section Configuring the Apache Maven repository for your Eclipse Vert.x projects, for more information.

6. Use the Eclipse Vert.x project as a template to create your own application.

7. Build the application using Maven from the root directory of the application.

```
mvn package
```

8. Run the application using Maven from the root directory of the application.

```
mvn vertx:run
```

5.2. CREATING A ECLIPSE VERT.X PROJECT USING THE COMMUNITY VERT.X STARTER

You can use the community Vert.x starter to create a Eclipse Vert.x project. The starter creates a community project. You will have to convert the community project to a Red Hat build of Eclipse Vert.x project.

Prerequisites

- OpenJDK 8 or OpenJDK 11 is installed.
- Maven 3 or higher is installed.
- A text editor or IDE is available.
- Curl or HTTPie or a browser to perform HTTP requests is available.

Procedure

1. In a command terminal, enter the following command to verify that Maven is using OpenJDK 8 or OpenJDK 11 and the Maven version is 3.6.0 or higher:

```
mvn --version
```

2. If the preceding command does not return OpenJDK 8 or OpenJDK 11, add the path to OpenJDK 8 or OpenJDK 11 to the PATH environment variable and enter the command again.
3. Go to Vert.x Starter.
4. Select the **Version** of Eclipse Vert.x.
5. Select **Java** as the language.
6. Select **Maven** as the build tool.
7. Enter a **Group Id**, which is a unique identifier of your project. For this procedure, keep the default, **com.example**.
8. Enter an **Artifact Id**, which is the name of your project and your project directory. For this procedure, keep the default, **starter**.
9. Specify the dependencies you want to add to your project. For this procedure, add **Vert.x Web** dependency either by typing it in the **Dependencies**text box or select from the list of **Dependencies**.

10. Click **Advanced options** to select the OpenJDK version. For this procedure, keep the default, **JDK 11**.

11. Click **Generate Project**. The **starter.zip** file containing the artifacts for Eclipse Vert.x project is downloaded.

12. Create a directory **getting-started**.

13. Extract the contents of the ZIP file to the **getting-started** folder. The Vert.x Starter creates an Eclipse Vert.x project with the following artifacts:

    - Maven build descriptor **pom.xml** file. The file has configurations to build and run your application.
    - Example verticle in the **src/main/java** folder.
    - Sample test using JUnit 5 in the **src/test/java** folder.
    - Editor configuration to enforce code style.
    - Git configuration to ignore files.

14. To convert the community project to a Red Hat build of Eclipse Vert.x project, replace the following values in **pom.xml** file:

    - **vertx.version** - Specify the Eclipse Vert.x version you want to use. For example, if you want to use Eclipse Vert.x 4.2.7 version, specify the version as 4.2.7.redhat-00003.
    - **vertx-stack-depchain** - Replace this dependency with **vertx-dependencies**.

15. Specify the repositories that contain the Eclipse Vert.x artifacts to build your application in the **pom.xml** file.

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

    Alternatively, you can configure the Maven repository to specify the build artifacts in the **settings.xml** file. See the section Configuring the Apache Maven repository for your Eclipse Vert.x projects, for more information.

16. Use the Eclipse Vert.x project as a template to create your own application.

17. Build the application using Maven from the root directory of the application.

```
mvn package
```

18. Run the application using Maven from the root directory of the application.

```
mvn exec:java
```

19. Verify that the application is running.

Use **curl** or your browser to verify if your application is running at **http://localhost:8888** and returns "Hello from Vert.x!" as response.

```
$ curl http://localhost:8888
Hello from Vert.x!
```

CHAPTER 6. CONFIGURING THE APACHE MAVEN REPOSITORY FOR YOUR ECLIPSE VERT.X PROJECTS

Apache Maven is a distributed build automation tool used in Java application development to create, manage, and build software projects. Maven uses standard configuration files called Project Object Model (POM) files to define projects and manage the build process. POM files describe the module and component dependencies, build order, and targets for the resulting project packaging and output using an XML file. This ensures that the project is built in a correct and uniform manner.

Maven plug-ins

Maven plug-ins are defined parts of a POM file that achieve one or more goals. Eclipse Vert.x applications use the following Maven plug-ins:

- Eclipse Vert.x Maven plug-in (**vertx-maven-plugin**): Enables Maven to create Eclipse Vert.x projects, supports the generation of uber-JAR files, and provides a development mode.
- Maven Surefire plug-in (**maven-surefire-plugin**): Used during the test phase of the build life cycle to execute unit tests on your application. The plug-in generates text and XML files that contain the test reports.

Maven repositories

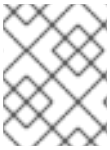
A Maven repository stores Java libraries, plug-ins, and other build artifacts. The default public repository is the Maven 2 Central Repository, but repositories can be private and internal within a company to share common artifacts among development teams. Repositories are also available from third-parties.

With your Eclipse Vert.x projects, you can use:

- Online Maven repository
- Download the Eclipse Vert.x Maven repository

6.1. CONFIGURING THE MAVEN SETTINGS.XML FILE FOR THE ONLINE REPOSITORY

You can use the online Eclipse Vert.x repository with your Eclipse Vert.x Maven project by configuring your user **settings.xml** file. This is the recommended approach. Maven settings used with a repository manager or repository on a shared server provide better control and manageability of projects.

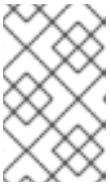


NOTE

When you configure the repository by modifying the Maven **settings.xml** file, the changes apply to all of your Maven projects.

Procedure

1. Open the Maven **~/.m2/settings.xml** file in a text editor or integrated development environment (IDE).



NOTE

If the **settings.xml** file is not available in the **~/.m2/** directory, copy the **settings.xml** file from the **$MAVEN_HOME/.m2/conf/** directory into the **~/.m2/** directory.

2. Add the following lines to the **<profiles>** element of the **settings.xml** file:

```
<!-- Configure the Maven repository -->
<profile>
 <id>red-hat-enterprise-maven-repository</id>
 <repositories>
  <repository>
   <id>red-hat-enterprise-maven-repository</id>
   <url>https://maven.repository.redhat.com/ga/</url>
   <releases>
    <enabled>true</enabled>
   </releases>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </repository>
 </repositories>
 </profile>
```

3. Add the following lines to the **<activeProfiles>** element of the **settings.xml** file and save the file.

```
<activeProfile>red-hat-enterprise-maven-repository</activeProfile>
```

6.2. DOWNLOADING AND CONFIGURING THE ECLIPSE VERT.X MAVEN REPOSITORY

If you do not want to use the online Maven repository, you can download and configure the Eclipse Vert.x Maven repository to create a Eclipse Vert.x application with Maven. The Eclipse Vert.x Maven repository contains many of the requirements that Java developers typically use to build their applications. This procedure describes how to edit the **settings.xml** file to configure the Eclipse Vert.x Maven repository.

NOTE

When you configure the repository by modifying the Maven **settings.xml** file, the changes apply to all of your Maven projects.

Procedure

1. Download the Eclipse Vert.x Maven repository ZIP file from the Software Downloads page of the Red Hat Customer Portal. To download the software, you must log in to the portal.

2. Expand the downloaded archive.

3. Change directory to the **~/.m2/** directory and open the Maven **settings.xml** file in a text editor or integrated development environment (IDE).

4. Add the following lines to the **<profiles>** element of the **settings.xml** file, where ***MAVEN_REPOSITORY*** is the path of the Eclipse Vert.x Maven repository that you downloaded. The format of ***MAVEN_REPOSITORY*** must be **file:**//**$PATH**, for example **file:///home/userX/rhb-vertx-4.1.5.SP1-maven-repository/maven-repository**.

```
<!-- Configure the Maven repository -->
<profile>
 <id>red-hat-enterprise-maven-repository</id>
 <repositories>
  <repository>
   <id>red-hat-enterprise-maven-repository</id>
   <url>MAVEN_REPOSITORY</url>
   <releases>
    <enabled>true</enabled>
   </releases>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
  </repository>
 </repositories>
</profile>
```

5. Add the following lines to the **<activeProfiles>** element of the **settings.xml** file and save the file.

```
<activeProfile>red-hat-enterprise-maven-repository</activeProfile>
```

IMPORTANT

If your Maven repository contains outdated artifacts, you might get one of the following Maven error messages when you build or deploy your project, where ***ARTIFACT_NAME*** is the name of a missing artifact and ***PROJECT_NAME*** is the name of the project you are trying to build:

- **Missing artifact *PROJECT_NAME***
- **[ERROR] Failed to execute goal on project *ARTIFACT_NAME*; Could not resolve dependencies for *PROJECT_NAME***

To resolve the issue, delete the cached version of your local repository located in the ~/**.m2/repository** directory to force a download of the latest Maven artifacts.

CHAPTER 7. ADDITIONAL RESOURCES

- For more information about the Maven Surefire plug-in, see the Apache Maven Project website.
- For information about the JUnit 5 testing framework, see the JUnit 5 website.