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

This guide provides details on using the Eclipse Vert.x runtime.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>6</td>
</tr>
<tr>
<td>PROVIDING FEEDBACK ON RED HAT DOCUMENTATION</td>
<td>7</td>
</tr>
<tr>
<td>MAKING OPEN SOURCE MORE INCLUSIVE</td>
<td>8</td>
</tr>
<tr>
<td><strong>CHAPTER 1. INTRODUCTION TO APPLICATION DEVELOPMENT WITH ECLIPSE VERT.X</strong></td>
<td>9</td>
</tr>
<tr>
<td>1.1. OVERVIEW OF APPLICATION DEVELOPMENT WITH RED HAT RUNTIMES</td>
<td>9</td>
</tr>
<tr>
<td>1.2. APPLICATION DEVELOPMENT ON RED HAT OPENSSHIFT USING DEVELOPER LAUNCHER</td>
<td>9</td>
</tr>
<tr>
<td>1.3. OVERVIEW OF ECLIPSE VERT.X</td>
<td>10</td>
</tr>
<tr>
<td>1.3.1. Key concepts of Eclipse Vert.x</td>
<td>10</td>
</tr>
<tr>
<td>1.3.2. Supported Architectures by Eclipse Vert.x</td>
<td>11</td>
</tr>
<tr>
<td>1.3.3. Introduction to example applications</td>
<td>11</td>
</tr>
<tr>
<td><strong>CHAPTER 2. CONFIGURING YOUR APPLICATIONS</strong></td>
<td>13</td>
</tr>
<tr>
<td>2.1. CONFIGURING YOUR APPLICATION TO USE ECLIPSE VERT.X</td>
<td>13</td>
</tr>
<tr>
<td><strong>CHAPTER 3. DOWNLOADING AND DEPLOYING APPLICATIONS USING DEVELOPER LAUNCHER</strong></td>
<td>15</td>
</tr>
<tr>
<td>3.1. WORKING WITH DEVELOPER LAUNCHER</td>
<td>15</td>
</tr>
<tr>
<td>3.2. DOWNLOADING THE EXAMPLE APPLICATIONS USING DEVELOPER LAUNCHER</td>
<td>15</td>
</tr>
<tr>
<td>3.3. DEPLOYING AN EXAMPLE APPLICATION ON OPENSSHIFT CONTAINER PLATFORM OR CDK (MINISHIFT)</td>
<td>16</td>
</tr>
<tr>
<td><strong>CHAPTER 4. DEVELOPING AND DEPLOYING ECLIPSE VERT.X RUNTIME APPLICATION</strong></td>
<td>18</td>
</tr>
<tr>
<td>4.1. DEVELOPING ECLIPSE VERT.X APPLICATION</td>
<td>18</td>
</tr>
<tr>
<td>4.2. DEPLOYING ECLIPSE VERT.X APPLICATION TO OPENSSHIFT</td>
<td>21</td>
</tr>
<tr>
<td>4.2.1. Supported Java images for Eclipse Vert.x</td>
<td>21</td>
</tr>
<tr>
<td>4.2.1.1. Images on x86_64 architecture</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1.2. Images on s390x (IBM Z) architecture</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1.3. Images on ppc64le (IBM Power Systems) architecture</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2. Preparing Eclipse Vert.x application for OpenShift deployment</td>
<td>22</td>
</tr>
<tr>
<td>4.2.3. Deploying Eclipse Vert.x application to OpenShift using OpenShift Maven plugin</td>
<td>24</td>
</tr>
<tr>
<td>4.3. DEPLOYING ECLIPSE VERT.X APPLICATION TO STAND-ALONE RED HAT ENTERPRISE LINUX</td>
<td>25</td>
</tr>
<tr>
<td>4.3.1. Preparing Eclipse Vert.x application for stand-alone Red Hat Enterprise Linux deployment</td>
<td>25</td>
</tr>
<tr>
<td>4.3.2. Deploying Eclipse Vert.x application to stand-alone Red Hat Enterprise Linux using jar</td>
<td>26</td>
</tr>
<tr>
<td><strong>CHAPTER 5. DEBUGGING ECLIPSE VERT.X BASED APPLICATION</strong></td>
<td>28</td>
</tr>
<tr>
<td>5.1. REMOTE DEBUGGING</td>
<td>28</td>
</tr>
<tr>
<td>5.1.1. Starting your application locally in debugging mode</td>
<td>28</td>
</tr>
<tr>
<td>5.1.2. Starting your application on OpenShift in debugging mode</td>
<td>28</td>
</tr>
<tr>
<td>5.1.3. Attaching a remote debugger to the application</td>
<td>29</td>
</tr>
<tr>
<td>5.2. DEBUG LOGGING</td>
<td>30</td>
</tr>
<tr>
<td>5.2.1. Configuring logging for your Eclipse Vert.x application using java.util.logging</td>
<td>30</td>
</tr>
<tr>
<td>5.2.2. Adding log output to your Eclipse Vert.x application.</td>
<td>31</td>
</tr>
<tr>
<td>5.2.3. Specifying a custom logging framework for your application</td>
<td>31</td>
</tr>
<tr>
<td>5.2.4. Configuring Netty logging for your Eclipse Vert.x application.</td>
<td>31</td>
</tr>
<tr>
<td>5.2.5. Accessing debug logs on OpenShift</td>
<td>32</td>
</tr>
<tr>
<td><strong>CHAPTER 6. MONITORING YOUR APPLICATION</strong></td>
<td>34</td>
</tr>
<tr>
<td>6.1. ACCESSING JVM METRICS FOR YOUR APPLICATION ON OPENSSHIFT</td>
<td>34</td>
</tr>
<tr>
<td>6.1.1. Accessing JVM metrics using Jolokia on OpenShift</td>
<td>34</td>
</tr>
<tr>
<td>6.2. EXPOSING APPLICATION METRICS USING PROMETHEUS WITH ECLIPSE VERT.X</td>
<td>35</td>
</tr>
<tr>
<td><strong>CHAPTER 7. EXAMPLE APPLICATIONS FOR ECLIPSE VERT.X</strong></td>
<td>39</td>
</tr>
</tbody>
</table>
7.1. REST API LEVEL 0 EXAMPLE FOR ECLIPSE VERT.X

7.1.1. REST API Level 0 design tradeoffs
7.1.2. Deploying the REST API Level 0 example application to OpenShift Online
   7.1.2.1. Deploying the example application using developers.redhat.com/launch
   7.1.2.2. Authenticating the oc CLI client
   7.1.2.3. Deploying the REST API Level 0 example application using the oc CLI client
7.1.3. Deploying the REST API Level 0 example application to Minishift or CDK
   7.1.3.1. Getting the Launcher tool URL and credentials
   7.1.3.2. Deploying the example application using the Launcher tool
   7.1.3.3. Authenticating the oc CLI client
   7.1.3.4. Deploying the REST API Level 0 example application using the oc CLI client
7.1.4. Deploying the REST API Level 0 example application to OpenShift Container Platform
7.1.5. Interacting with the unmodified REST API Level 0 example application for Eclipse Vert.x
7.1.6. Running the REST API Level 0 example application integration tests
7.1.7. REST resources

7.2. EXTERNALIZED CONFIGURATION EXAMPLE FOR ECLIPSE VERT.X

7.2.1. The externalized configuration design pattern
7.2.2. Externalized Configuration design tradeoffs
7.2.3. Deploying the Externalized Configuration example application to OpenShift Online
   7.2.3.1. Deploying the example application using developers.redhat.com/launch
   7.2.3.2. Authenticating the oc CLI client
   7.2.3.3. Deploying the Externalized Configuration example application using the oc CLI client
7.2.4. Deploying the Externalized Configuration example application to Minishift or CDK
   7.2.4.1. Getting the Launcher tool URL and credentials
   7.2.4.2. Deploying the example application using the Launcher tool
   7.2.4.3. Authenticating the oc CLI client
   7.2.4.4. Deploying the Externalized Configuration example application using the oc CLI client
7.2.5. Deploying the Externalized Configuration example application to OpenShift Container Platform
7.2.6. Interacting with the unmodified Externalized Configuration example application for Eclipse Vert.x
7.2.7. Running the Externalized Configuration example application integration tests
7.2.8. Externalized Configuration resources

7.3. RELATIONAL DATABASE BACKEND EXAMPLE FOR ECLIPSE VERT.X

7.3.1. Relational Database Backend design tradeoffs
7.3.2. Deploying the Relational Database Backend example application to OpenShift Online
   7.3.2.1. Deploying the example application using developers.redhat.com/launch
   7.3.2.2. Authenticating the oc CLI client
   7.3.2.3. Deploying the Relational Database Backend example application using the oc CLI client
7.3.3. Deploying the Relational Database Backend example application to Minishift or CDK
   7.3.3.1. Getting the Launcher tool URL and credentials
   7.3.3.2. Deploying the example application using the Launcher tool
   7.3.3.3. Authenticating the oc CLI client
   7.3.3.4. Deploying the Relational Database Backend example application using the oc CLI client
7.3.4. Deploying the Relational Database Backend example application to OpenShift Container Platform
7.3.5. Interacting with the Relational Database Backend API
    Troubleshooting
7.3.6. Running the Relational Database Backend example application integration tests
7.3.7. Relational database resources

7.4. HEALTH CHECK EXAMPLE FOR ECLIPSE VERT.X

7.4.1. Health check concepts
7.4.2. Deploying the Health Check example application to OpenShift Online
   7.4.2.1. Deploying the example application using developers.redhat.com/launch
   7.4.2.2. Authenticating the oc CLI client
   7.4.2.3. Deploying the Health Check example application using the oc CLI client
7.4.3. Health check resources
7.4.3. Deploying the Health Check example application to Minishift or CDK
7.4.3.1. Getting the Launcher tool URL and credentials
7.4.3.2. Deploying the example application using the Launcher tool
7.4.3.3. Authenticating the oc CLI client
7.4.3.4. Deploying the Health Check example application using the oc CLI client
7.4.4. Deploying the Health Check example application to OpenShift Container Platform
7.4.5. Interacting with the unmodified Health Check example application
7.4.6. Running the Health Check example application integration tests
7.4.7. Health check resources

7.5. CIRCUIT BREAKER EXAMPLE FOR ECLIPSE VERT.X
7.5.1. The circuit breaker design pattern
    Circuit breaker implementation
7.5.2. Circuit Breaker design tradeoffs
7.5.3. Deploying the Circuit Breaker example application to OpenShift Online
    7.5.3.1. Deploying the example application using developers.redhat.com/launch
    7.5.3.2. Authenticating the oc CLI client
    7.5.3.3. Deploying the Circuit Breaker example application using the oc CLI client
7.5.4. Deploying the Circuit Breaker example application to Minishift or CDK
    7.5.4.1. Getting the Launcher tool URL and credentials
    7.5.4.2. Deploying the example application using the Launcher tool
    7.5.4.3. Authenticating the oc CLI client
    7.5.4.4. Deploying the Circuit Breaker example application using the oc CLI client
7.5.5. Deploying the Circuit Breaker example application to OpenShift Container Platform
7.5.6. Interacting with the unmodified Eclipse Vert.x Circuit Breaker example application
7.5.7. Running the Circuit Breaker example application integration tests
7.5.8. Using Hystrix Dashboard to monitor the circuit breaker
7.5.9. Circuit breaker resources

7.6. SECURED EXAMPLE APPLICATION FOR ECLIPSE VERT.X
7.6.1. The Secured project structure
7.6.2. Red Hat SSO deployment configuration
7.6.3. Red Hat SSO realm model
    7.6.3.1. Red Hat SSO users
    7.6.3.2. The application clients
7.6.4. Eclipse Vert.x SSO adapter configuration
7.6.5. Deploying the Secured example application to Minishift or CDK
    7.6.5.1. Getting the Launcher tool URL and credentials
    7.6.5.2. Creating the Secured example application using Launcher
    7.6.5.3. Authenticating the oc CLI client
    7.6.5.4. Deploying the Secured example application using the oc CLI client
7.6.6. Deploying the Secured example application to OpenShift Container Platform
    7.6.6.1. Authenticating the oc CLI client
    7.6.6.2. Deploying the Secured example application using the oc CLI client
7.6.7. Authenticating to the Secured example application API endpoint
    7.6.7.1. Getting the Secured example application API endpoint
    7.6.7.2. Authenticating HTTP requests using the command line
    7.6.7.3. Authenticating HTTP requests using the web interface
7.6.8. Running the Eclipse Vert.x Secured example application integration tests
7.6.9. Secured SSO resources

7.7. CACHE EXAMPLE FOR ECLIPSE VERT.X
7.7.1. How caching works and when you need it
7.7.2. Deploying the Cache example application to OpenShift Online
    7.7.2.1. Deploying the example application using developers.redhat.com/launch
    7.7.2.2. Authenticating the oc CLI client
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 Container Development Kit (Minishift)**: A local cloud that you can install and execute on your local machine. This functionality is provided by Red Hat Container Development Kit (CDK) or Minishift.
- **Red Hat CodeReady Studio**: An integrated development environment (IDE) for developing, testing, and deploying applications.

To help you get started with application development, all the runtimes are available with example applications. These example applications are accessible from the Developer Launcher. You can use the examples as templates to create your 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. APPLICATION DEVELOPMENT ON RED HAT OPENSHEET USING DEVELOPER LAUNCHER

You can get started with developing cloud-native applications on OpenShift using Developer Launcher (developers.redhat.com/launch). It is a service provided by Red Hat.

Developer Launcher is a stand-alone project generator. You can use it to build and deploy applications on OpenShift instances, such as, OpenShift Container Platform or Minishift or CDK.
1.3. OVERVIEW OF ECLIPSE VERT.X

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

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.

1.3.1. 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.

### 1.3.2. 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

Different images are supported for different architectures. The example codes in this guide demonstrate the commands for x86_64 architecture. If you are using other architectures, specify the relevant image name in the commands.

Refer to the section [Supported Java images for Eclipse Vert.x](#) for more information about the image names.

### 1.3.3. Introduction to example applications

Examples are working applications that demonstrate how to build cloud native applications and services. They demonstrate prescriptive architectures, design patterns, tools, and best practices that should be used when you develop your applications. The example applications can be used as templates to create your cloud-native microservices. You can update and redeploy these examples using the deployment process explained in this guide.

The examples implement Microservice patterns such as:

- Creating REST APIs
- Interoperating with a database
- Implementing the health check pattern
- Externalizing the configuration of your applications to make them more secure and easier to scale

You can use the examples applications as:

- Working demonstration of the technology
- Learning tool or a sandbox to understand how to develop applications for your project
Starting point for updating or extending your own use case

Each example application is implemented in one or more runtimes. For example, the REST API Level 0 example is available for the following runtimes:

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

The subsequent sections explain the example applications implemented for the Eclipse Vert.x runtime.

You can download and deploy all the example applications on:

- x86_64 architecture - The example applications in this guide demonstrate how to build and deploy example applications on x86_64 architecture.
- s390x architecture - To deploy the example applications on OpenShift environments provisioned on IBM Z infrastructure, specify the relevant IBM Z image name in the commands.
- ppc64le architecture - To deploy the example applications on OpenShift environments provisioned on IBM Power Systems infrastructure, specify the relevant IBM Power Systems image name in the commands.

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

Some of the example applications also require other products, such as Red Hat Data Grid to demonstrate the workflows. In this case, you must also change the image names of these products to their relevant IBM Z or IBM Power Systems image names in the YAML file of the example applications.
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>
4. Include **repositories** and **pluginRepositories** to specify the repositories that contain the artifacts and plugins to build your application:

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

**Additional resources**

- For more information about packaging your Eclipse Vert.x application, see the [Vert.x Maven Plugin documentation](https://vertx.io).
CHAPTER 3. DOWNLOADING AND DEPLOYING APPLICATIONS USING DEVELOPER LAUNCHER

This section shows you how to download and deploy example applications provided with the runtimes. The example applications are available on Developer Launcher.

3.1. WORKING WITH DEVELOPER LAUNCHER

Developer Launcher (developers.redhat.com/launch) runs on OpenShift. When you deploy example applications, the Developer Launcher guides you through the process of:

- Selecting a runtime
- Building and executing the application

Based on your selection, Developer Launcher generates a custom project. You can either download a ZIP version of the project or directly launch the application on an OpenShift Online instance.

When you deploy your application on OpenShift using Developer Launcher, the Source-to-Image (S2I) build process is used. This build process handles all the configuration, build, and deployment steps that are required to run your application on OpenShift.

3.2. DOWNLOADING THE EXAMPLE APPLICATIONS USING DEVELOPER LAUNCHER

Red Hat provides example applications that help you get started with the Eclipse Vert.x runtime. These examples are available on Developer Launcher (developers.redhat.com/launch).

You can download the example applications, build, and deploy them. This section explains how to download example applications.

You can use the example applications as templates to create your own cloud-native applications.

Procedure

1. Go to Developer Launcher (developers.redhat.com/launch).
2. Click Start.
3. Click Deploy an Example Application.
4. Click Select an Example to see the list of example applications available with the runtime.
5. Select a runtime.
6. Select an example application.

NOTE

Some example applications are available for multiple runtimes. If you have not selected a runtime in the previous step, you can select a runtime from the list of available runtimes in the example application.
7. Select the release version for the runtime. You can choose from the community or product releases listed for the runtime.

8. Click Save.

9. Click Download to download the example application.
   A ZIP file containing the source and documentation files is downloaded.

3.3. DEPLOYING AN EXAMPLE APPLICATION ON OPENSHIFT CONTAINER PLATFORM OR CDK (MINISHIFT)

You can deploy the example application to either OpenShift Container Platform or CDK (Minishift). Depending on where you want to deploy your application use the relevant web console for authentication.

Prerequisites

- An example application project created using Developer Launcher.
- If you are deploying your application on OpenShift Container Platform, you must have access to the OpenShift Container Platform web console.
- If you are deploying your application on CDK (Minishift), you must have access to the CDK (Minishift) web console.
- oc command-line client installed.

Procedure

1. Download the example application.

2. You can deploy the example application on OpenShift Container Platform or CDK (Minishift) using the oc command-line client.
   You must authenticate the client using the token provided by the web console. Depending on where you want to deploy your application, use either the OpenShift Container Platform web console or CDK (Minishift) web console. Perform the following steps to get the authenticate the client:
   a. Login to the web console.
   b. Click the question mark icon, which is in the upper-right corner of the web console.
   c. Select Command Line Tools from the list.
   d. Copy the oc login command.
   e. Paste the command in a terminal to authenticate your oc CLI client with your account.

   $ oc login OPENSHIFT_URL --token=MYTOKEN

3. Extract the contents of the ZIP file.

   $ unzip MY_APPLICATION_NAME.zip

4. Create a new project in OpenShift.
5. Navigate to the root directory of `MY_APPLICATION_NAME`.

6. Deploy your example application using Maven.

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

NOTE: Some example applications may require additional setups. To build and deploy the example applications, follow the instructions provided in the `README` file.

7. 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 has the status `Running` after it is fully deployed and started. The pod name of your application may be different. The numeric value in the pod name is incremented for every new build. The letters at the end are generated when the pod is created.

8. After your example application is deployed and started, determine its route.

**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
MY_APP_NAME     8080                                                       MY_APP_NAME
```

The route information of a pod gives you the base URL which you can use to access it. In this example, you can use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.
CHAPTER 4. DEVELOPING AND DEPLOYING ECLIPSE VERT.X RUNTIME APPLICATION

In addition to using an example, you can create a new Eclipse Vert.x application and deploy it to OpenShift or stand-alone Red Hat Enterprise Linux.

4.1. 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 4.0 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) {
   

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

```xml
<?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>4.0.3.redhat-00002</vertx.version>
    <vertx-maven-plugin.version>1.0.24</vertx-maven-plugin.version>
    <vertx.vertx>com.example.MyApp</vertx.vertx>

    <!-- 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>
</project>
```
5. Build the application using Maven from the root directory of the application.
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. To learn how application health monitoring on OpenShift works, try the Health Check example.

### 4.2. 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.

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

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

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

### 4.2.1. 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.

Similar images are available on IBM Z and IBM Power Systems.

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

The following table lists the images supported by Eclipse Vert.x for different architectures. It also provides links to the images available in the Red Hat Ecosystem Catalog. The image pages contain authentication procedures required to access the RHEL 8 images.
4.2.1.1. Images on x86_64 architecture

<table>
<thead>
<tr>
<th>OS</th>
<th>Java</th>
<th>Red Hat Ecosystem Catalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL 7</td>
<td>OpenJDK 8</td>
<td>RHEL 7 with OpenJDK 8</td>
</tr>
<tr>
<td>RHEL 7</td>
<td>OpenJDK 11</td>
<td>RHEL 7 with OpenJDK 11</td>
</tr>
<tr>
<td>RHEL 8</td>
<td>OpenJDK 8</td>
<td>RHEL 8 Universal Base Image with OpenJDK 8</td>
</tr>
<tr>
<td>RHEL 8</td>
<td>OpenJDK 11</td>
<td>RHEL 8 Universal Base Image with OpenJDK 11</td>
</tr>
</tbody>
</table>

**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.

4.2.1.2. Images on s390x (IBM Z) architecture

<table>
<thead>
<tr>
<th>OS</th>
<th>Java</th>
<th>Red Hat Ecosystem Catalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL 8</td>
<td>Eclipse OpenJ9 11</td>
<td>RHEL 8 with Eclipse OpenJ9 11</td>
</tr>
</tbody>
</table>

4.2.1.3. Images on ppc64le (IBM Power Systems) architecture

<table>
<thead>
<tr>
<th>OS</th>
<th>Java</th>
<th>Red Hat Ecosystem Catalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL 8</td>
<td>Eclipse OpenJ9 11</td>
<td>RHEL 8 with Eclipse OpenJ9 11</td>
</tr>
</tbody>
</table>

**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.

4.2.2. 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:

```xml
<!-- 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
       ```xml
       <jkube.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:latest</jkube.generator.from>
       ```
     - RHEL 7 with OpenJDK 11
       ```xml
       <jkube.generator.from>registry.access.redhat.com/openjdk/openjdk-11-rhel7:latest</jkube.generator.from>
       ```
4.2.3. 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**

- **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!
```

### 4.3. 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.

**4.3.1. 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:

```xml
...<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.

   ```bash
   $ mvn clean package
   ```

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

### 4.3.2. 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.

   ```bash
   $ 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 5. DEBUGGING ECLIPSE VERT.X BASED APPLICATION

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

5.1. 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.

5.1.1. 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.

5.1.2. 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`.

5.1.3. 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


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.

5.2. 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.

5.2.1. 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 vertex-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.

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

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

   ```java
   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.

5.2.3. 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:

<table>
<thead>
<tr>
<th>Library</th>
<th>Class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log4J v1</td>
<td>io.vertx.core.logging.Log4jLogDelegateFactory</td>
</tr>
<tr>
<td>Log4J v2</td>
<td>io.vertx.core.logging.Log4j2LogDelegateFactory</td>
</tr>
<tr>
<td>SLF4J</td>
<td>io.vertx.core.logging.SLF4JLogDelegateFactory</td>
</tr>
</tbody>
</table>

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.

5.2.4. 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:

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

### 5.2.5. 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:
   
   ```bash
   $ mvn clean oc:deploy -Popenshift
   ```

2. View the logs:
   
   1. Get the name of the pod with your application:
      
      ```bash
      $ oc get pods
      ```
   2. Start watching the log output:
      
      ```bash
      $ 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, if you had debug logging in the REST API Level 0 example 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 6. MONITORING YOUR APPLICATION

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

6.1. ACCESSING JVM METRICS FOR YOUR APPLICATION ON OPENSHIFT

6.1.1. 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.

Prerequisites

- 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

### 6.2. 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**

```java
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:

```xml
<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](Eclipse Vert.x Micrometer Metrics).
CHAPTER 7. EXAMPLE APPLICATIONS FOR ECLIPSE VERT.X

The Eclipse Vert.x runtime provides example applications. When you start developing applications on OpenShift, you can use the example applications as templates.

You can access these example applications on Developer Launcher.

You can download and deploy all the example applications on:

- x86_64 architecture - The example applications in this guide demonstrate how to build and deploy example applications on x86_64 architecture.
- s390x architecture - To deploy the example applications on OpenShift environments provisioned on IBM Z infrastructure, specify the relevant IBM Z image name in the commands.
- ppc64le architecture - To deploy the example applications on OpenShift environments provisioned on IBM Power Systems infrastructure, specify the relevant IBM Power Systems image name in the commands.

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

Some of the example applications also require other products, such as Red Hat Data Grid to demonstrate the workflows. In this case, you must also change the image names of these products to their relevant IBM Z and IBM Power Systems image names in the YAML file of the example applications.

7.1. REST API LEVEL 0 EXAMPLE FOR ECLIPSE VERT.X

IMPORTANT

The following example is not meant to be run in a production environment.

Example proficiency level: Foundational.

What the REST API Level 0 example does

The REST API Level 0 example shows how to map business operations to a remote procedure call endpoint over HTTP using a REST framework. This corresponds to Level 0 in the Richardson Maturity Model. Creating an HTTP endpoint using REST and its underlying principles to define your API lets you quickly prototype and design the API flexibly.

This example introduces the mechanics of interacting with a remote service using the HTTP protocol. It allows you to:

- Execute an HTTP GET request on the api/greeting endpoint.
- Receive a response in JSON format with a payload consisting of the Hello, World! String.
- Execute an HTTP GET request on the api/greeting endpoint while passing in a String argument. This uses the name request parameter in the query string.
- Receive a response in JSON format with a payload of Hello, $name! with $name replaced by the value of the name parameter passed into the request.

7.1.1. REST API Level 0 design tradeoffs
Table 7.1. Design tradeoffs

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| ● The example application enables fast prototyping.  
● The API Design is flexible.  
● HTTP endpoints allow clients to be language-neutral. | ● As an application or service matures, the REST API Level 0 approach might not scale well. It might not support a clean API design or use cases with database interactions.  
○ Any operations involving shared, mutable state must be integrated with an appropriate backing datastore.  
○ All requests handled by this API design are scoped only to the container servicing the request. Subsequent requests might not be served by the same container. |

7.1.2. Deploying the REST API Level 0 example application to OpenShift Online

Use one of the following options to execute the REST API Level 0 example application on OpenShift Online.

- Use developers.redhat.com/launch

- Use the oc CLI client

Although each method uses the same oc commands to deploy your application, using developers.redhat.com/launch provides an automated deployment workflow that executes the oc commands for you.

7.1.2.1. Deploying the example application using developers.redhat.com/launch

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Red Hat Developer Launcher web interface.

Prerequisites

- An account at OpenShift Online.

Procedure

1. Navigate to the developers.redhat.com/launch URL in a browser.
2. Follow on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.1.2.2. Authenticating the oc CLI client

To work with example applications on OpenShift Online using the oc command-line client, you must authenticate the client using the token provided by the OpenShift Online web interface.

Prerequisites
Procedure

1. Navigate to the OpenShift Online URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the `oc login` command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your `oc` CLI client with your OpenShift Online account.

   $ oc login OPENSHIFT_URL --token=MYTOKEN

7.1.2.3. Deploying the REST API Level 0 example application using the `oc` CLI client

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the command line.

Prerequisites

- The example application created using developers.redhat.com/launch. For more information, see Section 7.1.2.1, “Deploying the example application using developers.redhat.com/launch”.

- The `oc` client authenticated. For more information, see Section 7.1.2.2, “Authenticating the `oc` CLI client”.

Procedure

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.

   $ unzip MY_PROJECT_NAME.zip

2. Create a new project in OpenShift.

   $ oc new-project MY_PROJECT_NAME

3. Navigate to the root directory of your application.

4. Use Maven to start the deployment to OpenShift.

   $ mvn clean oc:deploy -Popenshift

   This command uses the OpenShift Maven plugin to launch the S2I process on OpenShift and to start the pod.
5. 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. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

6. After your example application is deployed and started, determine its route.

**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 the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

7.1.3. Deploying the REST API Level 0 example application to Minishift or CDK

Use one of the following options to execute the REST API Level 0 example application locally on Minishift or CDK:

- **Using Launcher**
- **Using the `oc` CLI client**

Although each method uses the same `oc` commands to deploy your application, using Launcher provides an automated deployment workflow that executes the `oc` commands for you.

7.1.3.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.

2. Check the console output for the URL and user credentials you can use to access the running Launcher:

   **Example Console Output from a Minishift or CDK Startup**
   ```
   ```
7.1.3.2. Deploying the example application using the Launcher tool

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Launcher web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.1.3.1, “Getting the Launcher tool URL and credentials”.

**Procedure**

1. Navigate to the Launcher URL in a browser.

2. Follow the on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.1.3.3. Authenticating the `oc` CLI client

To work with example applications on Minishift or CDK using the `oc` command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.1.3.1, “Getting the Launcher tool URL and credentials”.

**Procedure**

1. Navigate to the Minishift or CDK URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the `oc login` command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your `oc` CLI client with your Minishift or CDK account.
This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the command line.

**Prerequisites**

- The example application created using Launcher tool on a Minishift or CDK. For more information, see Section 7.1.3.2, “Deploying the example application using the Launcher tool”.
- Your Launcher tool URL.
- The `oc` client authenticated. For more information, see Section 7.1.3.3, “Authenticating the `oc` CLI client”.

**Procedure**

1. Clone your project from GitHub.
   ```bash
   $ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
   ```
   Alternatively, if you downloaded a ZIP file of your project, extract it.
   ```bash
   $ unzip MY_PROJECT_NAME.zip
   ```
2. Create a new project in OpenShift.
   ```bash
   $ oc new-project MY_PROJECT_NAME
   ```
3. Navigate to the root directory of your application.
4. Use Maven to start the deployment to OpenShift.
   ```bash
   $ mvn clean oc:deploy -Popenshift
   ```
   This command uses the OpenShift Maven plugin to launch the S2I process on OpenShift and to start the pod.
5. Check the status of your application and ensure your pod is running.
   ```bash
   $ oc get pods -w
   NAME                             READY     STATUS      RESTARTS   AGE
   MY_APP_NAME-1-aaaa               1/1       Running     0          58s
   MY_APP_NAME-s2i-1-build          0/1       Completed   0          2m
   ```
   The `MY_APP_NAME-1-aaaa` pod should have a status of `Running` once it is fully deployed and started. Your specific pod name will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.
6. After your example application is deployed and started, determine its route.
**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 the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

### 7.1.4. Deploying the REST API Level 0 example application to OpenShift Container Platform

The process of creating and deploying example applications to OpenShift Container Platform is similar to OpenShift Online:

**Prerequisites**

- The example application created using [developers.redhat.com/launch](http://developers.redhat.com/launch).

**Procedure**

- Follow the instructions in Section 7.1.2, “Deploying the REST API Level 0 example application to OpenShift Online”, only use the URL and user credentials from the OpenShift Container Platform Web Console.

### 7.1.5. Interacting with the unmodified REST API Level 0 example application for Eclipse Vert.x

The example provides a default HTTP endpoint that accepts GET requests.

**Prerequisites**

- Your application running
- The `curl` binary or a web browser

**Procedure**

1. Use `curl` to execute a `GET` request against the example. You can also use a browser to do this.

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

2. Use `curl` to execute a `GET` request with the `name` URL parameter against the example. You can also use a browser to do this.

   ```
   $ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting?name=Sara
NOTE

From a browser, you can also use a form provided by the example to perform these same interactions. The form is located at the root of the project http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME.

7.1.6. Running the REST API Level 0 example application integration tests

This example application includes a self-contained set of integration tests. When run inside an OpenShift project, the tests:

- Deploy a test instance of the application to the project.
- Execute the individual tests on that instance.
- Remove all instances of the application from the project when the testing is done.

WARNING

Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.

Prerequisites

- The oc client authenticated
- An empty OpenShift project

Procedure

Execute the following command to run the integration tests:

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

7.1.7. REST resources

More background and related information on REST can be found here:

- [Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)]
- [Richardson Maturity Model]
- [JSR 311: JAX-RS: The JavaTM API for RESTful Web Services]
7.2. EXTERNALIZED CONFIGURATION EXAMPLE FOR ECLIPSE VERT.X

**IMPORTANT**

The following example is not meant to be run in a production environment.

Example proficiency level: *Foundational*.

Externalized Configuration provides a basic example of using a ConfigMap to externalize configuration. *ConfigMap* is an object used by OpenShift to inject configuration data as simple key and value pairs into one or more Linux containers while keeping the containers independent of OpenShift.

This example shows you how to:

- Set up and configure a *ConfigMap*.
- Use the configuration provided by the *ConfigMap* within an application.
- Deploy changes to the *ConfigMap* configuration of running applications.

### 7.2.1. The externalized configuration design pattern

Whenever possible, externalize the application configuration and separate it from the application code. This allows the application configuration to change as it moves through different environments, but leaves the code unchanged. Externalizing the configuration also keeps sensitive or internal information out of your code base and version control. Many languages and application servers provide environment variables to support externalizing an application’s configuration.

Microservices architectures and multi-language (polyglot) environments add a layer of complexity to managing an application’s configuration. Applications consist of independent, distributed services, and each can have its own configuration. Keeping all configuration data synchronized and accessible creates a maintenance challenge.

ConfigMaps enable the application configuration to be externalized and used in individual Linux containers and pods on OpenShift. You can create a ConfigMap object in a variety of ways, including using a YAML file, and inject it into the Linux container. ConfigMaps also allow you to group and scale sets of configuration data. This lets you configure a large number of environments beyond the basic *Development*, *Stage*, and *Production*. You can find more information about ConfigMaps in the [OpenShift documentation](#).

### 7.2.2. Externalized Configuration design tradeoffs

**Table 7.2. Design Tradeoffs**
Pros | Cons
--- | ---
- Configuration is separate from deployments | - Adding configuration to environment requires additional step
- Can be updated independently | - Has to be maintained separately
- Can be shared across services | - Requires coordination beyond the scope of a service

7.2.3. Deploying the Externalized Configuration example application to OpenShift Online

Use one of the following options to execute the Externalized Configuration example application on OpenShift Online.

- Use developers.redhat.com/launch
- Use the `oc` CLI client

Although each method uses the same `oc` commands to deploy your application, using developers.redhat.com/launch provides an automated deployment workflow that executes the `oc` commands for you.

7.2.3.1. Deploying the example application using developers.redhat.com/launch

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Red Hat Developer Launcher web interface.

**Prerequisites**

- An account at OpenShift Online.

**Procedure**

1. Navigate to the developers.redhat.com/launch URL in a browser.
2. Follow on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.2.3.2. Authenticating the `oc` CLI client

To work with example applications on OpenShift Online using the `oc` command-line client, you must authenticate the client using the token provided by the OpenShift Online web interface.

**Prerequisites**

- An account at OpenShift Online.

**Procedure**

1. Navigate to the OpenShift Online URL in a browser.
2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the `oc login` command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your `oc` CLI client with your OpenShift Online account.

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

### 7.2.3.3. Deploying the Externalized Configuration example application using the `oc` CLI client

This section shows you how to build your Externalized Configuration example application and deploy it to OpenShift from the command line.

#### Prerequisites

- The example application created using developers.redhat.com/launch. For more information, see Section 7.2.3.1, “Deploying the example application using developers.redhat.com/launch”.

- The `oc` client authenticated. For more information, see Section 7.2.3.2, “Authenticating the `oc` CLI client”.

#### Procedure

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.

   ```bash
   $ unzip MY_PROJECT_NAME.zip
   ```

2. Create a new OpenShift project.

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

3. Assign view access rights to the service account before deploying your example application, so that the application can access the OpenShift API in order to read the contents of the ConfigMap.

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

4. Navigate to the root directory of your application.

5. Deploy your ConfigMap configuration to OpenShift using `app-config.yml`.

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

6. Verify your ConfigMap configuration has been deployed.
7. Use Maven to start the deployment to OpenShift.

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

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

8. 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 its fully deployed and started. You should also wait for your pod to be *ready* before proceeding, which is shown in the **READY** column. For example, **MY_APP_NAME-1-aaaaa** is *ready* when the **READY** column is *1/1*. Your specific pod name will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

9. After your example application is deployed and started, determine its route.

**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 the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

7.2.4. Deploying the Externalized Configuration example application to Minishift or CDK

Use one of the following options to execute the Externalized Configuration example application locally on Minishift or CDK:

- **Using Launcher**
- **Using the `oc` CLI client**
Although each method uses the same `oc` commands to deploy your application, using Launcher provides an automated deployment workflow that executes the `oc` commands for you.

### 7.2.4.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.
2. Check the console output for the URL and user credentials you can use to access the running Launcher:

   **Example Console Output from a Minishift or CDK Startup**

   ```
   -- Removing temporary directory ... OK
   -- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
   https://192.168.42.152:8443
   
   You are logged in as:
   User: developer
   Password: developer
   
   To login as administrator:
   oc login -u system:admin
   ```

### 7.2.4.2. Deploying the example application using the Launcher tool

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Launcher web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.2.4.1, “Getting the Launcher tool URL and credentials”.

**Procedure**

1. Navigate to the Launcher URL in a browser.
2. Follow the on-screen instructions to create and launch your example application in Eclipse Vert.x.

### 7.2.4.3. Authenticating the `oc` CLI client
To work with example applications on Minishift or CDK using the `oc` command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.2.4.1, “Getting the Launcher tool URL and credentials”.

**Procedure**

1. Navigate to the Minishift or CDK URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the `oc login` command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your `oc` CLI client with your Minishift or CDK account.

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

**7.2.4.4. Deploying the Externalized Configuration example application using the `oc` CLI client**

This section shows you how to build your Externalized Configuration example application and deploy it to OpenShift from the command line.

**Prerequisites**

- The example application created using Launcher tool on a Minishift or CDK. For more information, see Section 7.2.4.2, “Deploying the example application using the Launcher tool”.

- Your Launcher tool URL.

- The `oc` client authenticated. For more information, see Section 7.2.4.3, “Authenticating the `oc` CLI client”.

**Procedure**

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.

   ```bash
   $ unzip MY_PROJECT_NAME.zip
   ```

2. Create a new OpenShift project.

   ```bash
   $ oc new-project MY_PROJECT_NAME
   ```
3. Assign view access rights to the service account before deploying your example application, so that the application can access the OpenShift API in order to read the contents of the ConfigMap.

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

4. Navigate to the root directory of your application.

5. Deploy your ConfigMap configuration to OpenShift using `app-config.yml`.

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

6. Verify your ConfigMap configuration has been deployed.

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

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

7. Use Maven to start the deployment to OpenShift.

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

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

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

```bash
$ oc get pods -w
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MY_APP_NAME-1-aaaaa</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>58s</td>
</tr>
<tr>
<td>MY_APP_NAME-s2i-1-build</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>2m</td>
</tr>
</tbody>
</table>

The `MY_APP_NAME-1-aaaaa` pod should have a status of `Running` once its fully deployed and started. You should also wait for your pod to be `ready` before proceeding, which is shown in the `READY` column. For example, `MY_APP_NAME-1-aaaaa` is `ready` when the `READY` column is `1/1`. Your specific pod name will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

9. After your example application is deployed and started, determine its route.

**Example Route Information**

```bash
$ oc get routes
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
<th>SERVICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>MY_APP_NAME</td>
<td>MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MY_APP_NAME</td>
<td>8080</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The route information of a pod gives you the base URL which you use to access it. In the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

### 7.2.5. Deploying the Externalized Configuration example application to OpenShift Container Platform

The process of creating and deploying example applications to OpenShift Container Platform is similar to OpenShift Online:

**Prerequisites**

- The example application created using `developers.redhat.com/launch`.

**Procedure**

- Follow the instructions in Section 7.2.3, "Deploying the Externalized Configuration example application to OpenShift Online", only use the URL and user credentials from the OpenShift Container Platform Web Console.

### 7.2.6. Interacting with the unmodified Externalized Configuration example application for Eclipse Vert.x

The example provides a default HTTP endpoint that accepts GET requests.

**Prerequisites**

- Your application running

  - The `curl` binary or a web browser

**Procedure**

1. Use `curl` to execute a **GET** request against the example. You can also use a browser to do this.

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

2. Update the deployed ConfigMap configuration.

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

   Change the value for the **message** key to **Bonjour, %s from a ConfigMap !** and save the file.

3. Update of the ConfigMap should be read by the application within an acceptable time (a few seconds) without requiring a restart of the application.

4. Execute a **GET** request using `curl` against the example with the updated ConfigMap configuration to see your updated greeting. You can also do this from your browser using the web form provided by the application.

   ```
   $ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
   {"content":"Bonjour, World from a ConfigMap !"}
   ```
7.2.7. Running the Externalized Configuration example application integration tests

This example application includes a self-contained set of integration tests. When run inside an OpenShift project, the tests:

- Deploy a test instance of the application to the project.
- Execute the individual tests on that instance.
- Remove all instances of the application from the project when the testing is done.

**WARNING**

Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.

**Prerequisites**

- The `oc` client authenticated
- An empty OpenShift project
- View access permission assigned to the service account of your example application. This allows your application to read the configuration from the ConfigMap:

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

**Procedure**

Execute the following command to run the integration tests:

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

**7.2.8. Externalized Configuration resources**

More background and related information on Externalized Configuration and ConfigMap can be found here:

- OpenShift ConfigMap Documentation
- Blog Post about ConfigMap in OpenShift
- Externalized Configuration with Eclipse Vert.x
- Externalized Configuration for Spring Boot
- Externalized Configuration for Thorntail
- Externalized Configuration for Node.js
7.3. RELATIONAL DATABASE BACKEND EXAMPLE FOR ECLIPSE VERT.X

IMPORTANT

The following example is not meant to be run in a production environment.

Limitation: Run this example application on a Minishift or CDK. You can also use a manual workflow to deploy this example to OpenShift Online Pro and OpenShift Container Platform. This example is not currently available on OpenShift Online Starter.

Example proficiency level: Foundational.

What the Relational Database Backend example does

The Relational Database Backend example expands on the REST API Level 0 application to provide a basic example of performing create, read, update and delete (CRUD) operations on a PostgreSQL database using a simple HTTP API. CRUD operations are the four basic functions of persistent storage, widely used when developing an HTTP API dealing with a database.

The example also demonstrates the ability of the HTTP application to locate and connect to a database in OpenShift. Each runtime shows how to implement the connectivity solution best suited in the given case. The runtime can choose between options such as using JDBC, JPA, or accessing ORM APIs directly.

The example application exposes an HTTP API, which provides endpoints that allow you to manipulate data by performing CRUD operations over HTTP. The CRUD operations are mapped to HTTP Verbs. The API uses JSON formatting to receive requests and return responses to the user. The user can also use a user interface provided by the example to use the application. Specifically, this example provides an application that allows you to:

- Navigate to the application web interface in your browser. This exposes a simple website allowing you to perform CRUD operations on the data in the my_data database.
- Execute an HTTP GET request on the api/fruits endpoint.
- Receive a response formatted as a JSON array containing the list of all fruits in the database.
- Execute an HTTP GET request on the api/fruits/* endpoint while passing in a valid item ID as an argument.
- Receive a response in JSON format containing the name of the fruit with the given ID. If no item matches the specified ID, the call results in an HTTP error 404.
- Execute an HTTP POST request on the api/fruits endpoint passing in a valid name value to create a new entry in the database.
- Execute an HTTP PUT request on the api/fruits/* endpoint passing in a valid ID and a name as an argument. This updates the name of the item with the given ID to match the name specified in your request.
- Execute an HTTP DELETE request on the api/fruits/* endpoint, passing in a valid ID as an argument. This removes the item with the specified ID from the database and returns an HTTP code 204 (No Content) as a response. If you pass in an invalid ID, the call results in an HTTP error 404.
This example also contains a set of automated integration tests that can be used to verify that the application is fully integrated with the database.

This example does not showcase a fully matured RESTful model (level 3), but it does use compatible HTTP verbs and status, following the recommended HTTP API practices.

### 7.3.1. Relational Database Backend design tradeoffs

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Each runtime determines how to implement the database interactions. One can use a low-level connectivity API such as JDBC, some other can use JPA, and yet another can access ORM APIs directly. Each runtime decides what would be the best way.</td>
<td>● The PostgreSQL database provided with this example application is not backed up with persistent storage. Changes to the database are lost if you stop or redeploy the database pod. To use an external database with your example application’s pod in order to preserve changes, see the Creating an application with a database chapter of the OpenShift Documentation. It is also possible to set up persistent storage with database containers on OpenShift. (For more details about using persistent storage with OpenShift and containers, see the Persistent Storage, Managing Volumes and Persistent Volumes chapters of the OpenShift Documentation).</td>
</tr>
<tr>
<td>● Each runtime determines how the schema is created.</td>
<td></td>
</tr>
</tbody>
</table>

### 7.3.2. Deploying the Relational Database Backend example application to OpenShift Online

Use one of the following options to execute the Relational Database Backend example application on OpenShift Online.

- Use developers.redhat.com/launch
- Use the oc CLI client

Although each method uses the same oc commands to deploy your application, using developers.redhat.com/launch provides an automated deployment workflow that executes the oc commands for you.

#### 7.3.2.1. Deploying the example application using developers.redhat.com/launch

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Red Hat Developer Launcher web interface.

**Prerequisites**

- An account at OpenShift Online.

**Procedure**
1. Navigate to the developers.redhat.com/launch URL in a browser.

2. Follow on-screen instructions to create and launch your example application in Eclipse Vert.x.

### 7.3.2.2. Authenticating the oc CLI client

To work with example applications on OpenShift Online using the oc command-line client, you must authenticate the client using the token provided by the OpenShift Online web interface.

#### Prerequisites

- An account at OpenShift Online.

#### Procedure

1. Navigate to the OpenShift Online URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the `oc login` command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your OpenShift Online account.

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

### 7.3.2.3. Deploying the Relational Database Backend example application using the oc CLI client

This section shows you how to build your Relational Database Backend example application and deploy it to OpenShift from the command line.

#### Prerequisites

- The example application created using developers.redhat.com/launch. For more information, see Section 7.3.2.1, “Deploying the example application using developers.redhat.com/launch”.

- The oc client authenticated. For more information, see Section 7.3.2.2, “Authenticating the oc CLI client”.

#### Procedure

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.

   ```bash
   $ unzip MY_PROJECT_NAME.zip
   ```

2. Create a new OpenShift project.
3. Navigate to the root directory of your application.

4. Deploy the PostgreSQL database to OpenShift. Ensure that you use the following values for user name, password, and database name when creating your database application. The example application is pre-configured to use these values. Using different values prevents your application from integrating with the database.

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

5. Check the status of your database and ensure the pod is running.

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

The **my-database-1-aaaaa** pod should have a status of **Running** and should be indicated as ready once it is fully deployed and started. Your specific pod name will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

6. Use maven to start the deployment to OpenShift.

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

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

7. 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-aaaaaa       1/1      Running    0          58s
MY_APP_NAME-s2i-1-build    0/1      Completed  0          2m
```

Your **MY_APP_NAME-1-aaaaaa** pod should have a status of **Running** and should be indicated as ready once it is fully deployed and started.

8. After your example application is deployed and started, determine its route.

**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 the example above, you would use [http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME](http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME) as the base URL to access the application.
7.3.3. Deploying the Relational Database Backend example application to Minishift or CDK

Use one of the following options to execute the Relational Database Backend example application locally on Minishift or CDK:

- Using Launcher
- Using the `oc` CLI client

Although each method uses the same `oc` commands to deploy your application, using Launcher provides an automated deployment workflow that executes the `oc` commands for you.

7.3.3.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.

2. Check the console output for the URL and user credentials you can use to access the running Launcher:

   **Example Console Output from a Minishift or CDK Startup**

   ```
   ...
   -- Removing temporary directory ... OK
   -- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
   https://192.168.42.152:8443
   
   You are logged in as:
   User:     developer
   Password: developer
   
   To login as administrator:
   oc login -u system:admin
   ```

7.3.3.2. Deploying the example application using the Launcher tool

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Launcher web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.3.3.1, “Getting the Launcher tool URL and credentials”. 

Procedure

1. Navigate to the Launcher URL in a browser.

2. Follow the on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.3.3.3. Authenticating the oc CLI client

To work with example applications on Minishift or CDK using the oc command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

Prerequisites

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.3.3.1, “Getting the Launcher tool URL and credentials”.

Procedure

1. Navigate to the Minishift or CDK URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the oc login command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your Minishift or CDK account.

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

7.3.3.4. Deploying the Relational Database Backend example application using the oc CLI client

This section shows you how to build your Relational Database Backend example application and deploy it to OpenShift from the command line.

Prerequisites

- The example application created using Launcher tool on a Minishift or CDK. For more information, see Section 7.3.3.2, “Deploying the example application using the Launcher tool”.

- Your Launcher tool URL.

- The oc client authenticated. For more information, see Section 7.3.3.3, “Authenticating the oc CLI client”.

Procedure

1. Clone your project from GitHub.

   ```bash
   $ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
   ```
Alternatively, if you downloaded a ZIP file of your project, extract it.

$ unzip MY_PROJECT_NAME.zip

2. Create a new OpenShift project.

$ oc new-project MY_PROJECT_NAME

3. Navigate to the root directory of your application.

4. Deploy the PostgreSQL database to OpenShift. Ensure that you use the following values for user name, password, and database name when creating your database application. The example application is pre-configured to use these values. Using different values prevents your application from integrating with the database.

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

5. Check the status of your database and ensure the pod is running.

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

The my-database-1-aaaaa pod should have a status of Running and should be indicated as ready once it is fully deployed and started. Your specific pod name will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

6. Use maven to start the deployment to OpenShift.

$ mvn clean oc:deploy -Popenshift

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

7. 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 58s

Your MY_APP_NAME-1-aaaaa pod should have a status of Running and should be indicated as ready once it is fully deployed and started.

8. After your example application is deployed and started, determine its route.

Example Route Information

$ oc get routes
NAME HOST/PORT PATH SERVICES PORT TERMINATION
The route information of a pod gives you the base URL which you use to access it. In the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

### 7.3.4. Deploying the Relational Database Backend example application to OpenShift Container Platform

The process of creating and deploying example applications to OpenShift Container Platform is similar to OpenShift Online:

**Prerequisites**

- The example application created using [developers.redhat.com/launch](http://developers.redhat.com/launch).

**Procedure**

- Follow the instructions in Section 7.3.2, “Deploying the Relational Database Backend example application to OpenShift Online”, only use the URL and user credentials from the OpenShift Container Platform Web Console.

### 7.3.5. Interacting with the Relational Database Backend API

When you have finished creating your example application, you can interact with it the following way:

**Prerequisites**

- Your application running
- The `curl` binary or a web browser

**Procedure**

1. Obtain the URL of your application by executing the following command:

   ```bash
   $ oc get route MY_APP_NAME
   ```

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
<th>SERVICES</th>
<th>PORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MY_APP_NAME</td>
<td>MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MY_APP_NAME</td>
<td>8080</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. To access the web interface of the database application, navigate to the application URL in your browser:

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

   Alternatively, you can make requests directly on the `api/fruits/*` endpoint using `curl`:

   List all entries in the database:
Retrieve an entry with a specific ID

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

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

Create a new entry:

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

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

Update an Entry

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

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

- If you receive an HTTP Error code 503 as a response after executing these commands, it means that the application is not ready yet.

7.3.6. Running the Relational Database Backend example application integration tests

This example application includes a self-contained set of integration tests. When run inside an OpenShift project, the tests:

- Deploy a test instance of the application to the project.
- Execute the individual tests on that instance.
- Remove all instances of the application from the project when the testing is done.

**WARNING**

Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.

Prerequisites

- The `oc` client authenticated
- An empty OpenShift project

Procedure

Execute the following command to run the integration tests:

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

7.3.7. Relational database resources

More background and related information on running relational databases in OpenShift, CRUD, HTTP API and REST can be found here:

- HTTP Verbs
- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- The never ending REST API design debase
REST APIs must be Hypertext driven
- Richardson Maturity Model
- JSR 311: JAX-RS: The JavaTM API for RESTful Web Services
- Some Rest with Eclipse Vert.x
- Using the Eclipse Vert.x asynchronous SQL client
- Relational Database Backend for Spring Boot
- Relational Database Backend for Thorntail
- Relational Database Backend for Node.js

7.4. HEALTH CHECK EXAMPLE FOR ECLIPSE VERT.X

IMPORTANT

The following example is not meant to be run in a production environment.

Example proficiency level: Foundational.

When you deploy an application, it is important to know if it is available and if it can start handling incoming requests. Implementing the health check pattern allows you to monitor the health of an application, which includes if an application is available and whether it is able to service requests.

NOTE

If you are not familiar with the health check terminology, see the Section 7.4.1, "Health check concepts" section first.

The purpose of this use case is to demonstrate the health check pattern through the use of probing. Probing is used to report the liveness and readiness of an application. In this use case, you configure an application which exposes an HTTP health endpoint to issue HTTP requests. If the container is alive, according to the liveness probe on the health HTTP endpoint, the management platform receives 200 as return code and no further action is required. If the health HTTP endpoint does not return a response, for example if the thread is blocked, then the application is not considered alive according to the liveness probe. In that case, the platform kills the pod corresponding to that application and recreates a new pod to restart the application.

This use case also allows you to demonstrate and use a readiness probe. In cases where the application is running but is unable to handle requests, such as when the application returns an HTTP 503 response code during restart, this application is not considered ready according to the readiness probe. If the application is not considered ready by the readiness probe, requests are not routed to that application until it is considered ready according to the readiness probe.

7.4.1. Health check concepts

In order to understand the health check pattern, you need to first understand the following concepts:

Liveness
Liveness defines whether an application is running or not. Sometimes a running application moves into an unresponsive or stopped state and needs to be restarted. Checking for liveness helps determine whether or not an application needs to be restarted.

Readiness
Readiness defines whether a running application can service requests. Sometimes a running application moves into an error or broken state where it can no longer service requests. Checking readiness helps determine whether or not requests should continue to be routed to that application.

Fail-over
Fail-over enables failures in servicing requests to be handled gracefully. If an application fails to service a request, that request and future requests can then fail-over or be routed to another application, which is usually a redundant copy of that same application.

Resilience and Stability
Resilience and Stability enable failures in servicing requests to be handled gracefully. If an application fails to service a request due to connection loss, in a resilient system that request can be retried after the connection is re-established.

Probe
A probe is a Kubernetes action that periodically performs diagnostics on a running container.

7.4.2. Deploying the Health Check example application to OpenShift Online

Use one of the following options to execute the Health Check example application on OpenShift Online.

- Use developers.redhat.com/launch
- Use the oc CLI client

Although each method uses the same oc commands to deploy your application, using developers.redhat.com/launch provides an automated deployment workflow that executes the oc commands for you.

7.4.2.1. Deploying the example application using developers.redhat.com/launch

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Red Hat Developer Launcher web interface.

Prerequisites
- An account at OpenShift Online.

Procedure
1. Navigate to the developers.redhat.com/launch URL in a browser.
2. Follow on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.4.2.2. Authenticating the oc CLI client

To work with example applications on OpenShift Online using the oc command-line client, you must authenticate the client using the token provided by the OpenShift Online web interface.

Prerequisites
An account at OpenShift Online.

Procedure

1. Navigate to the OpenShift Online URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the `oc login` command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your `oc` CLI client with your OpenShift Online account.

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

7.4.2.3. Deploying the Health Check example application using the `oc` CLI client

This section shows you how to build your Health Check example application and deploy it to OpenShift from the command line.

Prerequisites

- The example application created using developers.redhat.com/launch. For more information, see Section 7.4.2.1, "Deploying the example application using developers.redhat.com/launch".

- The `oc` client authenticated. For more information, see Section 7.4.2.2, "Authenticating the `oc` CLI client".

Procedure

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.

   ```
   $ unzip MY_PROJECT_NAME.zip
   ```

2. Create a new OpenShift project.

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

3. Navigate to the root directory of your application.

4. Use Maven to start the deployment to OpenShift.

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

   This command uses the OpenShift Maven plugin to launch the S2I process on OpenShift and to start the pod.
5. Check the status of your application and ensure your pod is running.

The `MY_APP_NAME-1-aaaaa` pod should have a status of **Running** once its fully deployed and started. You should also wait for your pod to be ready before proceeding, which is shown in the **READY** column. For example, `MY_APP_NAME-1-aaaaa` is ready when the **READY** column is 1/1. Your specific pod name will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

6. After your example application is deployed and started, determine its route.

**Example Route Information**

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

The route information of a pod gives you the base URL which you use to access it. In the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

### 7.4.3. Deploying the Health Check example application to Minishift or CDK

Use one of the following options to execute the Health Check example application locally on Minishift or CDK:

- **Using Launcher**

- **Using the oc CLI client**

Although each method uses the same `oc` commands to deploy your application, using Launcher provides an automated deployment workflow that executes the `oc` commands for you.

#### 7.4.3.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.

2. Check the console output for the URL and user credentials you can use to access the running Launcher:
Example Console Output from a Minishift or CDK Startup

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

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

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

7.4.3.2. Deploying the example application using the Launcher tool

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Launcher web interface.

Prerequisites

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.4.3.1, “Getting the Launcher tool URL and credentials”.

Procedure

1. Navigate to the Launcher URL in a browser.

2. Follow the on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.4.3.3. Authenticating the oc CLI client

To work with example applications on Minishift or CDK using the oc command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

Prerequisites

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.4.3.1, “Getting the Launcher tool URL and credentials”.

Procedure

1. Navigate to the Minishift or CDK URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.


4. Copy the oc login command.
5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your Minishift or CDK account.

   $ oc login OPENSHIFT_URL --token=MYTOKEN

### 7.4.3.4. Deploying the Health Check example application using the oc CLI client

This section shows you how to build your Health Check example application and deploy it to OpenShift from the command line.

**Prerequisites**

- The example application created using Launcher tool on a Minishift or CDK. For more information, see Section 7.4.3.2, “Deploying the example application using the Launcher tool”.
- Your Launcher tool URL.
- The oc client authenticated. For more information, see Section 7.4.3.3, “Authenticating the oc CLI client”.

**Procedure**

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.

   $ unzip MY_PROJECT_NAME.zip

2. Create a new OpenShift project.

   $ oc new-project MY_PROJECT_NAME

3. Navigate to the root directory of your application.

4. Use Maven to start the deployment to OpenShift.

   $ mvn clean oc:deploy -Popenshift

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

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

   $ oc get pods -w

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MY_APP_NAME-1-aaaaa</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>58s</td>
</tr>
<tr>
<td>MY_APP_NAME-s2i-1-build</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>2m</td>
</tr>
</tbody>
</table>

The **MY_APP_NAME-1-aaaaa** pod should have a status of **Running** once its fully deployed and started. You should also wait for your pod to be ready before proceeding, which is shown in the **READY** column. For example, **MY_APP_NAME-1-aaaaa** is ready when the **READY** column is
6. After your example application is deployed and started, determine its route.

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 the example above, you would use `http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

7.4.4. Deploying the Health Check example application to OpenShift Container Platform

The process of creating and deploying example applications to OpenShift Container Platform is similar to OpenShift Online:

Prerequisites

- The example application created using `developers.redhat.com/launch`.

Procedure

- Follow the instructions in Section 7.4.2, “Deploying the Health Check example application to OpenShift Online”, only use the URL and user credentials from the OpenShift Container Platform Web Console.

7.4.5. Interacting with the unmodified Health Check example application

After you deploy the example application, you will have the `MY_APP_NAME` service running. The `MY_APP_NAME` service exposes the following REST endpoints:

/api/greeting

Returns a JSON containing greeting of `name` parameter (or World as default value).

/api/stop

Forces the service to become unresponsive as means to simulate a failure.

The following steps demonstrate how to verify the service availability and simulate a failure. This failure of an available service causes the OpenShift self-healing capabilities to be trigger on the service.

Alternatively, you can use the web interface to perform these steps.

1. Use `curl` to execute a **GET** request against the `MY_APP_NAME` service. You can also use a browser to do this.

   ```
   $ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
   ```
2. Invoke the `/api/stop` endpoint and verify the availability of the `/api/greeting` endpoint shortly after that. Invoking the `/api/stop` endpoint simulates an internal service failure and triggers the OpenShift self-healing capabilities. When invoking `/api/greeting` after simulating the failure, the service should return a HTTP status **503**.

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/stop
Stopping HTTP server, Bye bye world!
```

(followed by)

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
Not online
```

3. Use `oc get pods -w` to continuously watch the self-healing capabilities in action. While invoking the service failure, you can watch the self-healing capabilities in action on OpenShift console, or with the `oc` client tools. You should see the number of pods in the `<b>READY</b>` state move to zero (0/1) and after a short period (less than one minute) move back up to one (1/1). In addition to that, the `<b>RESTARTS</b>` count increases every time you invoke the service failure.

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

4. Optional: Use the web interface to invoke the service. Alternatively to the interaction using the terminal window, you can use the web interface provided by the service to invoke the different methods and watch the service move through the life cycle phases.

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

5. Optional: Use the web console to view the log output generated by the application at each stage of the self-healing process.

1. Navigate to your project.
2. On the sidebar, click on `Monitoring`.
3. In the upper right-hand corner of the screen, click on `Events` to display the log messages.
4. Optional: Click `View Details` to display a detailed view of the Event log.

The health check application generates the following messages:
<table>
<thead>
<tr>
<th>Message</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unhealthy</td>
<td>Readiness probe failed. This message is expected and indicates that the simulated failure of the /api/greeting endpoint has been detected and the self-healing process starts.</td>
</tr>
<tr>
<td>Killing</td>
<td>The unavailable Docker container running the service is being killed before being re-created.</td>
</tr>
<tr>
<td>Pulling</td>
<td>Downloading the latest version of docker image to re-create the container.</td>
</tr>
<tr>
<td>Pulled</td>
<td>Docker image downloaded successfully.</td>
</tr>
<tr>
<td>Created</td>
<td>Docker container has been successfully created</td>
</tr>
<tr>
<td>Started</td>
<td>Docker container is ready to handle requests</td>
</tr>
</tbody>
</table>

### 7.4.6. Running the Health Check example application integration tests

This example application includes a self-contained set of integration tests. When run inside an OpenShift project, the tests:

- Deploy a test instance of the application to the project.
- Execute the individual tests on that instance.
- Remove all instances of the application from the project when the testing is done.

**WARNING**

Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.

**Prerequisites**

- The `oc` client authenticated
- An empty OpenShift project

**Procedure**

Execute the following command to run the integration tests:
$ mvn clean verify -Popenshift,openshift-it

7.4.7. Health check resources

More background and related information on health checking can be found here:

- Application Health in OpenShift
- Kubernetes Liveness and Readiness Probes
- Health Check for Spring Boot
- Health Check for Thorntail
- Health Check for Node.js

7.5. CIRCUIT BREAKER EXAMPLE FOR ECLIPSE VERT.X

IMPORTANT

The following example is not meant to be run in a production environment.

Limitation: Run this example application on a Minishift or CDK. You can also use a manual workflow to deploy this example to OpenShift Online Pro and OpenShift Container Platform. This example is not currently available on OpenShift Online Starter.

Example proficiency level: Foundational.

The Circuit Breaker example demonstrates a generic pattern for reporting the failure of a service and then limiting access to the failed service until it becomes available to handle requests. This helps prevent cascading failure in other services that depend on the failed services for functionality.

This example shows you how to implement a Circuit Breaker and Fallback pattern in your services.

7.5.1. The circuit breaker design pattern

The Circuit Breaker is a pattern intended to:

- Reduce the impact of network failure and high latency on service architectures where services synchronously invoke other services.
  If one of the services:
    - becomes unavailable due to network failure, or
    - incurs unusually high latency values due to overwhelming traffic,
  other services attempting to call its endpoint may end up exhausting critical resources in an attempt to reach it, rendering themselves unusable.
- Prevent the condition also known as cascading failure, which can render the entire microservice architecture unusable.
- Act as a proxy between a protected function and a remote function, which monitors for failures.
Trip once the failures reach a certain threshold, and all further calls to the circuit breaker return
an error or a predefined fallback response, without the protected call being made at all.

The Circuit Breaker usually also contain an error reporting mechanism that notifies you when the Circuit
Breaker trips.

Circuit breaker implementation

- With the Circuit Breaker pattern implemented, a service client invokes a remote service
  endpoint via a proxy at regular intervals.
- If the calls to the remote service endpoint fail repeatedly and consistently, the Circuit Breaker
  trips, making all calls to the service fail immediately over a set timeout period and returns a
  predefined fallback response.
- When the timeout period expires, a limited number of test calls are allowed to pass through to
  the remote service to determine whether it has healed, or remains unavailable.
  - If the test calls fail, the Circuit Breaker keeps the service unavailable and keeps returning
    the fallback responses to incoming calls.
  - If the test calls succeed, the Circuit Breaker closes, fully enabling traffic to reach the remote
    service again.

7.5.2. Circuit Breaker design tradeoffs

Table 7.4. Design Tradeoffs

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enables a service to handle the failure of other services it invokes.</td>
<td>Optimizing the timeout values can be challenging</td>
</tr>
<tr>
<td></td>
<td>Larger-than-necessary timeout values may generate excessive latency.</td>
</tr>
<tr>
<td></td>
<td>Smaller-than-necessary timeout values may introduce false positives.</td>
</tr>
</tbody>
</table>

7.5.3. Deploying the Circuit Breaker example application to OpenShift Online

Use one of the following options to execute the Circuit Breaker example application on OpenShift Online.

- Use developers.redhat.com/launch
- Use the oc CLI client

Although each method uses the same oc commands to deploy your application, using developers.redhat.com/launch provides an automated deployment workflow that executes the oc commands for you.

7.5.3.1. Deploying the example application using developers.redhat.com/launch
This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Red Hat Developer Launcher web interface.

Prerequisites

- An account at OpenShift Online.

Procedure

1. Navigate to the developers.redhat.com/launch URL in a browser.
2. Follow on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.5.3.2. Authenticating the oc CLI client

To work with example applications on OpenShift Online using the oc command-line client, you must authenticate the client using the token provided by the OpenShift Online web interface.

Prerequisites

- An account at OpenShift Online.

Procedure

1. Navigate to the OpenShift Online URL in a browser.
2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.
4. Copy the oc login command.
5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your OpenShift Online account.

$ oc login OPENSHIFT_URL --token=MYTOKEN

7.5.3.3. Deploying the Circuit Breaker example application using the oc CLI client

This section shows you how to build your Circuit Breaker example application and deploy it to OpenShift from the command line.

Prerequisites

- The example application created using developers.redhat.com/launch. For more information, see Section 7.5.3.1, “Deploying the example application using developers.redhat.com/launch”.
- The oc client authenticated. For more information, see Section 7.5.3.2, “Authenticating the oc CLI client”.

Procedure

1. Clone your project from GitHub.
Alternatively, if you downloaded a ZIP file of your project, extract it.

$ unzip MY_PROJECT_NAME.zip

2. Create a new OpenShift project.

$ oc new-project MY_PROJECT_NAME

3. Navigate to the root directory of your application.

4. Use Maven to start the deployment to OpenShift.

$ mvn clean oc:deploy -Popenshift

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

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

$ oc get pods -w

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MY_APP_NAME-greeting-1-aaaaa</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>17s</td>
</tr>
<tr>
<td>MY_APP_NAME-greeting-1-deploy</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>22s</td>
</tr>
<tr>
<td>MY_APP_NAME-name-1-aaaaa</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>14s</td>
</tr>
<tr>
<td>MY_APP_NAME-name-1-deploy</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>28s</td>
</tr>
</tbody>
</table>

Both the **MY_APP_NAME-greeting-1-aaaaa** and **MY_APP_NAME-name-1-aaaaa** pods should have a status of Running once they are fully deployed and started. You should also wait for your pods to be ready before proceeding, which is shown in the READY column. For example, **MY_APP_NAME-greeting-1-aaaaa** is ready when the READY column is 1/1. Your specific pod names will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

6. After your example application is deployed and started, determine its route.

**Example Route Information**

$ oc get routes

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
<th>SERVICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORT TERMINATION</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MY_APP_NAME-greeting</td>
<td>MY_APP_NAME-greeting-</td>
<td>MY_PROJECT_NAME.OPENSIFT_HOSTNAME</td>
<td>MY_APP_NAME-greeting</td>
</tr>
<tr>
<td>None</td>
<td></td>
<td>MY_APP_NAME-name-</td>
<td>MY_APP_NAME-name-</td>
</tr>
</tbody>
</table>

The route information of a pod gives you the base URL which you use to access it. In the example above, you would use **http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSIFT_HOSTNAME** as the base URL to access the application.
7.5.4. Deploying the Circuit Breaker example application to Minishift or CDK

Use one of the following options to execute the Circuit Breaker example application locally on Minishift or CDK:

- Using Launcher
- Using the `oc` CLI client

Although each method uses the same `oc` commands to deploy your application, using Launcher provides an automated deployment workflow that executes the `oc` commands for you.

7.5.4.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.
2. Check the console output for the URL and user credentials you can use to access the running Launcher:

   **Example Console Output from a Minishift or CDK Startup**

   ```
   ...
   -- Removing temporary directory ... OK
   -- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
   https://192.168.42.152:8443
   
   You are logged in as:
   User:     developer
   Password: developer
   
   To login as administrator:
   oc login -u system:admin
   ```

7.5.4.2. Deploying the example application using the Launcher tool

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Launcher web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.5.4.1, “Getting the Launcher tool URL and credentials”. 
Procedure

1. Navigate to the Launcher URL in a browser.
2. Follow the on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.5.4.3. Authenticating the **oc** CLI client

To work with example applications on Minishift or CDK using the **oc** command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

Prerequisites

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.5.4.1, “Getting the Launcher tool URL and credentials”.

Procedure

1. Navigate to the Minishift or CDK URL in a browser.
2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.
3. Select **Command Line Tools** in the drop-down menu.
4. Copy the **oc login** command.
5. Paste the command in a terminal. The command uses your authentication token to authenticate your **oc** CLI client with your Minishift or CDK account.

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

7.5.4.4. Deploying the Circuit Breaker example application using the **oc** CLI client

This section shows you how to build your Circuit Breaker example application and deploy it to OpenShift from the command line.

Prerequisites

- The example application created using Launcher tool on a Minishift or CDK. For more information, see Section 7.5.4.2, “Deploying the example application using the Launcher tool”.
- Your Launcher tool URL.
- The **oc** client authenticated. For more information, see Section 7.5.4.3, “Authenticating the **oc** CLI client”.

Procedure

1. Clone your project from GitHub.

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```
Alternatively, if you downloaded a ZIP file of your project, extract it.

```
$ unzip MY_PROJECT_NAME.zip
```

2. Create a new OpenShift project.

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

3. Navigate to the root directory of your application.

4. Use Maven to start the deployment to OpenShift.

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

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

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

```
$ oc get pods -w
```

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

Both the `MY_APP_NAME-greeting-1-aaaaa` and `MY_APP_NAME-name-1-aaaaa` pods should have a status of `Running` once they are fully deployed and started. You should also wait for your pods to be ready before proceeding, which is shown in the `READY` column. For example, `MY_APP_NAME-greeting-1-aaaaa` is ready when the `READY` column is `1/1`. Your specific pod names will vary. The number in the middle will increase with each new build. The letters at the end are generated when the pod is created.

6. After your example application is deployed and started, determine its route.

**Example Route Information**

```
$ oc get routes
```

```
NAME                 HOST/PORT                                                     PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting  MY_APP_NAME-greeting- MY_APP_NAME-greeting-   MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name     MY_APP_NAME-name-    MY_APP_NAME-name-   MY_APP_NAME-name     8080
None
```

The route information of a pod gives you the base URL which you use to access it. In the example above, you would use `http://MY_APP_NAME-greeting- MY_APP_NAME-greeting- MY_PROJECT_NAME.OPENSHIFT_HOSTNAME` as the base URL to access the application.

**7.5.5. Deploying the Circuit Breaker example application to OpenShift Container Platform**
The process of creating and deploying example applications to OpenShift Container Platform is similar to OpenShift Online:

**Prerequisites**

- The example application created using developers.redhat.com/launch.

**Procedure**

- Follow the instructions in Section 7.5.3, “Deploying the Circuit Breaker example application to OpenShift Online”, only use the URL and user credentials from the OpenShift Container Platform Web Console.

### 7.5.6. Interacting with the unmodified Eclipse Vert.x Circuit Breaker example application

After you have the Eclipse Vert.x example application deployed, you have the following services running:

**MY_APP_NAME-name**

Exposes the following endpoints:

- the /api/name endpoint, which returns a name when this service is working, and an error when this service is set up to demonstrate failure.
- the /api/state endpoint, which controls the behavior of the /api/name endpoint and determines whether the service works correctly or demonstrates failure.

**MY_APP_NAME-greeting**

Exposes the following endpoints:

- the /api/greeting endpoint that you can call to get a personalized greeting response. When you call the /api/greeting endpoint, it issues a call against the /api/name endpoint of the MY_APP_NAME-name service as part of processing your request. The call made against the /api/name endpoint is protected by the Circuit Breaker.

If the remote endpoint is available, the name service responds with an HTTP code 200 (OK) and you receive the following greeting from the /api/greeting endpoint:

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

If the remote endpoint is unavailable, the name service responds with an HTTP code 500 (Internal server error) and you receive a predefined fallback response from the /api/greeting endpoint:

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

- the /api/cb-state endpoint, which returns the state of the Circuit Breaker. The state can be:
  - open: the circuit breaker is preventing requests from reaching the failed service,
  - closed: the circuit breaker is allowing requests to reach the service.
half-open: the circuit breaker is allowing a request to reach the service. If the request succeeds, the state of the service is reset to closed. If the request fails, the timer is restarted.

The following steps demonstrate how to verify the availability of the service, simulate a failure and receive a fallback response.

1. Use curl to execute a GET request against the MY_APP_NAME-greeting service. You can also use the Invoke button in the web interface to do this.

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

2. To simulate the failure of the MY_APP_NAME-name service you can:
   - use the Toggle button in the web interface.
   - scale the number of replicas of the pod running the MY_APP_NAME-name service down to 0.
   - execute an HTTP PUT request against the /api/state endpoint of the MY_APP_NAME-name service to set its state to fail.

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

3. Invoke the /api/greeting endpoint. When several requests on the /api/name endpoint fail:
   a. the Circuit Breaker opens,
   b. the state indicator in the web interface changes from CLOSED to OPEN,
   c. the Circuit Breaker issues a fallback response when you invoke the /api/greeting endpoint:

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

4. Restore the name MY_APP_NAME-name service to availability. To do this you can:
   - use the Toggle button in the web interface.
   - scale the number of replicas of the pod running the MY_APP_NAME-name service back up to 1.
   - execute an HTTP PUT request against the /api/state endpoint of the MY_APP_NAME-name service to set its state back to ok.

   ```sh
   $ curl -X PUT -H "Content-Type: application/json" -d '{"state": "ok"}'
   http://MY_APP_NAME-name-
   MY_PROJECT_NAME.LOCAL_OPENSHT_HOSTNAME/api/state
   ```
5. Invoke the /api/greeting endpoint again. When several requests on the /api/name endpoint succeed:
   a. the Circuit Breaker closes,
   b. the state indicator in the web interface changes from OPEN to CLOSED,
   c. the Circuit Breaker issues a returns the Hello World! greeting when you invoke the /api/greeting endpoint:

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

7.5.7. Running the Circuit Breaker example application integration tests

This example application includes a self-contained set of integration tests. When run inside an OpenShift project, the tests:

- Deploy a test instance of the application to the project.
- Execute the individual tests on that instance.
- Remove all instances of the application from the project when the testing is done.

**WARNING**

Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.

**Prerequisites**

- The oc client authenticated
- An empty OpenShift project

**Procedure**

Execute the following command to run the integration tests:

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

7.5.8. Using Hystrix Dashboard to monitor the circuit breaker

Hystrix Dashboard lets you easily monitor the health of your services in real time by aggregating Hystrix metrics data from an event stream and displaying them on one screen.

**Prerequisites**
The application deployed

Procedure

1. Log in to your Minishift or CDK cluster.

   $ oc login OPENSHIFT_URL --token=MYTOKEN

2. To access the Web console, use your browser to navigate to your Minishift or CDK URL.

3. Navigate to the project that contains your Circuit Breaker application.

   $ oc project MY_PROJECT_NAME

4. Import the YAML template for the Hystrix Dashboard application. You can do this by clicking Add to Project, then selecting the Import YAML / JSON tab, and copying the contents of the YAML file into the text box. Alternatively, you can execute the following command:


5. Click the Create button to create the Hystrix Dashboard application based on the template. Alternatively, you can execute the following command.

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

6. Wait for the pod containing Hystrix Dashboard to deploy.

7. Obtain the route of your Hystrix Dashboard application.

   $ oc get route hystrix-dashboard

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
<th>SERVICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>hystrix-dashboard</td>
<td>hystrix-dashboard-&lt;all&gt;</td>
<td></td>
<td>hystrix-dashboard</td>
</tr>
<tr>
<td></td>
<td>MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME</td>
<td>hystrix-dashboard</td>
<td>None</td>
</tr>
</tbody>
</table>

8. To access the Dashboard, open the Dashboard application route URL in your browser. Alternatively, you can navigate to the Overview screen in the Web console and click the route URL in the header above the pod containing your Hystrix Dashboard application.

9. To use the Dashboard to monitor the MY_APP_NAME-greeting service, replace the default event stream address with the following address and click the Monitor Stream button.

   http://MY_APP_NAME-greeting/hystrix.stream

Additional resources

- The Hystrix Dashboard wiki page

7.5.9. Circuit breaker resources
Follow the links below for more background information on the design principles behind the Circuit Breaker pattern

- microservices.io: Microservice Patterns: Circuit Breaker
- Martin Fowler: CircuitBreaker
- Circuit Breaker for Spring Boot
- Circuit Breaker for Node.js
- Circuit Breaker for Thorntail

7.6. SECURED EXAMPLE APPLICATION FOR ECLIPSE VERT.X

**IMPORTANT**
The following example is not meant to be run in a production environment.

**Limitation:** Run this example application on a Minishift or CDK. You can also use a manual workflow to deploy this example to OpenShift Online Pro and OpenShift Container Platform. This example is not currently available on OpenShift Online Starter.

Example proficiency level: **Advanced**.

The Secured example application secures a REST endpoint using Red Hat SSO. (This example expands on the REST API Level 0 example).

Red Hat SSO:

- Implements the Open ID Connect protocol which is an extension of the OAuth 2.0 specification.
- Issues access tokens to provide clients with various access rights to secured resources.

Securing an application with SSO enables you to add security to your applications while centralizing the security configuration.

**IMPORTANT**
This example comes with Red Hat SSO pre-configured for demonstration purposes, it does not explain its principles, usage, or configuration. Before using this example, ensure that you are familiar with the basic concepts related to Red Hat SSO.

7.6.1. The Secured project structure

The SSO example contains:

- the sources for the Greeting service, which is the one which we are going to to secure
- a template file (*service.sso.yaml*) to deploy the SSO server
- the Keycloak adapter configuration to secure the service

7.6.2. Red Hat SSO deployment configuration
The `service.sso.yaml` file in this example contains all OpenShift configuration items to deploy a pre-configured Red Hat SSO server. The SSO server configuration has been simplified for the sake of this exercise and does provide an out-of-the-box configuration, with pre-configured users and security settings. The `service.sso.yaml` file also contains very long lines, and some text editors, such as `gedit`, may have issues reading this file.

**WARNING**

It is not recommended to use this SSO configuration in production. Specifically, the simplifications made to the example security configuration impact the ability to use it in a production environment.

**Table 7.5. SSO Example Simplifications**

<table>
<thead>
<tr>
<th>Change</th>
<th>Reason</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default configuration includes both public and private keys in the yaml configuration files.</td>
<td>We did this because the end user can deploy Red Hat SSO module and have it in a usable state without needing to know the internals or how to configure Red Hat SSO.</td>
<td>In production, do not store private keys under source control. They should be added by the server administrator.</td>
</tr>
<tr>
<td>The configured clients accept any callback url.</td>
<td>To avoid having a custom configuration for each runtime, we avoid the callback verification that is required by the OAuth2 specification.</td>
<td>An application-specific callback URL should be provided with a valid domain name.</td>
</tr>
<tr>
<td>Clients do not require SSL/TLS and the secured applications are not exposed over HTTPS.</td>
<td>The examples are simplified by not requiring certificates generated for each runtime.</td>
<td>In production a secure application should use HTTPS rather than plain HTTP.</td>
</tr>
<tr>
<td>The token timeout has been increased to 10 minutes from the default of 1 minute.</td>
<td>Provides a better user experience when working with the command line examples</td>
<td>From a security perspective, the window an attacker would have to guess the access token is extended. It is recommended to keep this window short as it makes it much harder for a potential attacker to guess the current token.</td>
</tr>
</tbody>
</table>

**7.6.3. Red Hat SSO realm model**

The `master` realm is used to secure this example. There are two pre-configured application client definitions that provide a model for command line clients and the secured REST endpoint.
There are also two pre-configured users in the Red Hat SSO master realm that can be used to validate various authentication and authorization outcomes: admin and alice.

7.6.3.1. Red Hat SSO users

The realm model for the secured examples includes two users:

admin

The admin user has a password of admin and is the realm administrator. This user has full access to the Red Hat SSO administration console, but none of the role mappings that are required to access the secured endpoints. You can use this user to illustrate the behavior of an authenticated, but unauthorized user.

alice

The alice user has a password of password and is the canonical application user. This user will demonstrate successful authenticated and authorized access to the secured endpoints. An example representation of the role mappings is provided in this decoded JWT bearer token:

```
{
  "jti": "0073cfaa-7ed6-4326-ac07-c108d34b4f82",
  "exp": 1510162193,
  "nbf": 0,
  "iat": 1510161593,
  "iss": "https://secure-sso-sso.LOCAL_OPENSHIFT_HOSTNAME/auth/realms/master",
  "aud": "demoapp",
  "sub": "c0175ccb-0892-4b31-829f-dda873815fe8",
  "typ": "Bearer",
  "azp": "demoapp",
  "nonce": "90f5f1a-ba44-45ae-a413-50b08b4a242",
  "auth_time": 1510161591,
  "session_state": "98efb95a-b355-43d1-996b-0abcb1304352",
  "acr": "1",
  "client_session": "5962112c-2b19-461e-8aac-84ab512d2a01",
  "allowed-origins": [ "*" ],
  "realm_access": { "roles": [ "example-admin" ] },
  "resource_access": { "secured-example-endpoint": { "roles": [ "example-admin" ] }, "account": { "roles": [ "manage-account", "view-profile" ] } },
  "name": "Alice InChains",
}``
```
“preferred_username”: "alice",  
“given_name”: "Alice",  
“family_name”: "InChains",  
“email”: "alice@keycloak.org" 
}

1. The **iss** field corresponds to the Red Hat SSO realm instance URL that issues the token. This must be configured in the secured endpoint deployments in order for the token to be verified.

2. The **roles** object provides the roles that have been granted to the user at the global realm level. In this case **alice** has been granted the **example-admin** role. We will see that the secured endpoint will look to the realm level for authorized roles.

3. The **resource_access** object contains resource specific role grants. Under this object you will find an object for each of the secured endpoints.

4. The **resource_access.secured-example-endpoint.roles** object contains the roles granted to **alice** for the **secured-example-endpoint** resource.

5. The **preferred_username** field provides the username that was used to generate the access token.

7.6.3.2. The application clients

The OAuth 2.0 specification allows you to define a role for application clients that access secured resources on behalf of resource owners. The **master** realm has the following application clients defined:

- **demoapp**

  This is a **confidential** type client with a client secret that is used to obtain an access token. The token contains grants for the **alice** user which enable **alice** to access the Thorntail, Eclipse Vert.x, Node.js and Spring Boot based REST example application deployments.

- **secured-example-endpoint**

  The **secured-example-endpoint** is a bearer-only type of client that requires a **example-admin** role for accessing the associated resources, specifically the Greeting service.

7.6.4. Eclipse Vert.x SSO adapter configuration

The SSO adapter is the **client side**, or client to the SSO server, component that enforces security on the web resources. In this specific case, it is the greeting service.

Enacting security

```javascript
router.route("/greeting")
  .handler(JWTAuthHandler.create(  
    JWTAuth.create(vertx,  
      new JWTAuthOptions()  
        .addPubSecKey(new PubSecKeyOptions()  
          .setAlgorithm("RS256")  
          .setPublicKey(System.getenv("REALM_PUBLIC_KEY")))  
        .setPermissionsClaimKey("realm_access/roles"));
```
1. Locate the HTTP route to secure.
2. Instantiate a new JWT security handler.
3. The authorization enforcer is created.
4. The configuration to the enforcer.
5. Public key encryption algorithm.
6. PEM format of the realm public key. You can obtain this from the administration console.
7. Where the authorization enforcer should lookup permissions.

The enforcer here is configured using PEM format of the realm public key and specifying the algorithm. And since the enforcer is configured to consume keycloak JWTs, we also need to provide a location for the permission claims in the token.

Below is a JSON file reconstructed from the deployment environment variables, which is used when interacting with the application through web interface.

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

1. The security realm to be used.
2. The address of the Red Hat SSO server (Interpolation at build time)
3. The actual keycloak client configuration.

### 7.6.5. Deploying the Secured example application to Minishift or CDK

#### 7.6.5.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.
2. Check the console output for the URL and user credentials you can use to access the running Launcher:
Example Console Output from a Minishift or CDK Startup

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

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

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

```

7.6.5.2. Creating the Secured example application using Launcher

**Prerequisites**

- The URL and user credentials of your running Launcher instance. For more information, see Section 7.6.5.1, “Getting the Launcher tool URL and credentials”.

**Procedure**

- Navigate to the Launcher URL in a browser.
- Follow the on-screen instructions to create your example in Eclipse Vert.x. When asked about which deployment type, select *I will build and run locally*.
- Follow on-screen instructions. When done, click the **Download as ZIP file** button and store the file on your hard drive.

7.6.5.3. Authenticating the `oc` CLI client

To work with example applications on Minishift or CDK using the `oc` command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

**Prerequisites**

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.6.5.1, “Getting the Launcher tool URL and credentials”.

**Procedure**

1. Navigate to the Minishift or CDK URL in a browser.
2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.
4. Copy the `oc login` command.
5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your Minishift or CDK account.

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

7.6.5.4. Deploying the Secured example application using the oc CLI client

This section shows you how to build your Secured example application and deploy it to OpenShift from the command line.

**Prerequisites**

- The example application created using the Launcher tool on a Minishift or CDK. For more information, see Section 7.6.5.2, “Creating the Secured example application using Launcher”.
- Your Launcher URL.
- The oc client authenticated. For more information, see Section 7.6.5.3, ”Authenticating the oc CLI client”.

**Procedure**

1. Clone your project from GitHub.

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

Alternatively, if you downloaded a ZIP file of your project, extract it.

```bash
$ unzip MY_PROJECT_NAME.zip
```

2. Create a new OpenShift project.

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

3. Navigate to the root directory of your application.

4. Deploy the Red Hat SSO server using the service.sso.yaml file from your example ZIP file:

```bash
$ oc create -f service.sso.yaml
```

5. Use Maven to start the deployment to Minishift or CDK.

```bash
$ mvn clean oc:deploy -Popenshift -DskipTests \
   -DSSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath="{"https://"}{.spec.host}{"/auth
"})
```

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

This process generates the uberjar file as well as the OpenShift resources and deploys them to the current project on your Minishift or CDK server.

7.6.6. Deploying the Secured example application to OpenShift Container Platform
In addition to the Minishift or CDK, you can create and deploy the example on OpenShift Container Platform with only minor differences. The most important difference is that you need to create the example application on Minishift or CDK before you can deploy it with OpenShift Container Platform.

**Prerequisites**
- The example created using Minishift or CDK.

### 7.6.6.1. Authenticating the oc CLI client

To work with example applications on OpenShift Container Platform using the **oc** command-line client, you must authenticate the client using the token provided by the OpenShift Container Platform web interface.

**Prerequisites**
- An account at OpenShift Container Platform.

**Procedure**

1. Navigate to the OpenShift Container Platform URL in a browser.

2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.

3. Select **Command Line Tools** in the drop-down menu.

4. Copy the **oc login** command.

5. Paste the command in a terminal. The command uses your authentication token to authenticate your **oc** CLI client with your OpenShift Container Platform account.

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

### 7.6.6.2. Deploying the Secured example application using the oc CLI client

This section shows you how to build your Secured example application and deploy it to OpenShift from the command line.

**Prerequisites**
- The example application created using the Launcher tool on a Minishift or CDK.
- The **oc** client authenticated. For more information, see Section 7.6.6.1, “Authenticating the oc CLI client”.

**Procedure**

1. Clone your project from GitHub.

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

   Alternatively, if you downloaded a ZIP file of your project, extract it.
$ unzip MY_PROJECT_NAME.zip

2. Create a new OpenShift project.

$ oc new-project MY_PROJECT_NAME

3. Navigate to the root directory of your application.

4. Deploy the Red Hat SSO server using the service.sso.yaml file from your example ZIP file:

$ oc create -f service.sso.yaml

5. Use Maven to start the deployment to OpenShift Container Platform.

$ mvn clean oc:deploy -Popenshift -DskipTests \
   -DSSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}' \ 
   {.spec.host}[/auth
]

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

This process generates the uberjar file as well as the OpenShift resources and deploys them to the current project on your OpenShift Container Platform server.

7.6.7. Authenticating to the Secured example application API endpoint

The Secured example application provides a default HTTP endpoint that accepts GET requests if the caller is authenticated and authorized. The client first authenticates against the Red Hat SSO server and then performs a GET request against the Secured example application using the access token returned by the authentication step.

7.6.7.1. Getting the Secured example application API endpoint

When using a client to interact with the example, you must specify the Secured example application endpoint, which is the PROJECT_ID service.

Prerequisites

- The Secured example application deployed and running.
- The oc client authenticated.

Procedure

1. In a terminal application, execute the oc get routes command.
   A sample output is shown in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Host/Port</th>
<th>Path</th>
<th>Services</th>
<th>Port</th>
<th>Termination</th>
</tr>
</thead>
</table>
### Name | Host/Port | Path | Services | Port | Termination
--- | --- | --- | --- | --- | ---
secure-sso | secure-sso-myproject.LOCAL_OPENSHIFT_HOSTNAME | secure-sso | <all> | passthrough

| PROJECT_ID | PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME | PROJECT_ID | <all> |

| sso | sso-myproject.LOCAL_OPENSHIFT_HOSTNAME | sso | <all> |

In the above example, the example endpoint would be `http://PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME`. PROJECT_ID is based on the name you entered when generating your example using developers.redhat.com/launch or the Launcher tool.

### 7.6.7.2. Authenticating HTTP requests using the command line

Request a token by sending a HTTP POST request to the Red Hat SSO server. In the following example, the jq CLI tool is used to extract the token value from the JSON response.

#### Prerequisites

- The secured example endpoint URL. For more information, see Section 7.6.7.1, “Getting the Secured example application API endpoint”.
- The `jq` command-line tool (optional). To download the tool and for more information, see https://stedolan.github.io/jq/.

#### Procedure

1. Request an access token with `curl`, the credentials, and `<SSO_AUTH_SERVER_URL>` and extract the token from the response with the `jq` command:

   ```bash
   curl -sk -X POST https://<SSO_AUTH_SERVER_URL>/auth/realms/master/protocol/openid-connect/token \
   -d grant_type=password \
   -d username=alice\n   -d password=password \n   -d client_id=demoapp \n   -d client_secret=1daa57a2-b60e-468b-a3ac-25bd2dc2eadc
   ```
The attributes, such as \texttt{username}, \texttt{password}, and \texttt{client_secret} are usually kept secret, but the above command uses the default provided credentials with this example for demonstration purpose.

If you do not want to use \texttt{jq} to extract the token, you can run just the \texttt{curl} command and manually extract the access token.

\begin{verbatim}
NOTE
The \texttt{-sk} option tells curl to ignore failures resulting from self-signed certificates.
Do not use this option in a production environment. On macOS, you must have \texttt{curl} version 7.56.1 or greater installed. It must also be built with OpenSSL.
\end{verbatim}

1. Invoke the Secured service. Attach the access (bearer) token to the HTTP headers:

\begin{verbatim}
$ curl -v -H "Authorization: Bearer \$TOKEN" http://\$SERVICE_HOST/api/greeting

{ "content": "Hello, World!", "id": 2 }
\end{verbatim}

Example 7.2. A sample GET Request Headers with an Access (Bearer) Token

\begin{verbatim}
> GET /api/greeting HTTP/1.1
> Host: \$SERVICE_HOST
> User-Agent: curl/7.51.0
> Accept: */*
> Authorization: Bearer \$TOKEN
\end{verbatim}
<SERVICE_HOST> is the URL of the secured example endpoint. For more information, see Section 7.6.7.1, “Getting the Secured example application API endpoint”.

2. Verify the signature of the access token.
   The access token is a JSON Web Token, so you can decode it using the JWT Debugger:
   a. In a web browser, navigate to the JWT Debugger website.
   b. Select RS256 from the Algorithm drop down menu.

   NOTE
   Make sure the web form has been updated after you made the selection, so it displays the correct RSASHA256(...) information in the Signature section. If it has not, try switching to HS256 and then back to RS256.

   c. Paste the following content in the topmost text box into the VERIFY SIGNATURE section:

   -----BEGIN PUBLIC KEY-----
   MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoETnPmN55xBJjRzN/cs30OzJ9dkr4LVMqjzdTxF0yRts20vFzdh0O9xZucTMbsCOA/ZfSf8K+6yvBxypOSYv75EUDyypmckK1KoptqY5KEBQ1KwhWuP7lWQ0Q/hUwD6ji1QWDFgxfM/h34FvEn/0TJ1xN2P8T12YanwuDZgqsoQb.mxTOkAdnFXIUQcCkiEZ2C41uCrxiS4CEx5OX91aK9HKZV4JZ6vnqMhdDsMdO+UFtxOBYZio+a1jP4W3d7J5fGeIOaXjCOpivKnP2yU2DPdWmDMyYb671DRA+jhOJFKZ5H2mNgE3Il59vdsRwIDAQAB
   -----END PUBLIC KEY-----

   NOTE
   This is the master realm public key from the Red Hat SSO server deployment of the Secured example application.

   d. Paste the token output from the client output into the Encoded box. The Signature Verified sign is displayed on the debugger page.

7.6.7.3. Authenticating HTTP requests using the web interface

In addition to the HTTP API, the secured endpoint also contains a web interface to interact with.

The following procedure is an exercise for you to see how security is enforced, how you authenticate, and how you work with the authentication token.

Prerequisites

- The secured endpoint URL. For more information, see Section 7.6.7.1, “Getting the Secured example application API endpoint”.

Procedure

1. In a web browser, navigate to the endpoint URL.
2. Perform an unauthenticated request:
   a. Click the *Invoke* button.

   **Figure 7.1. Unauthenticated Secured Example Web Interface**

   ![Unauthenticated Secured Example Web Interface](image)

   The service responds with an **HTTP 401 Unauthorized** status code.

   **Figure 7.2. Unauthenticated Error Message**

   ![Unauthenticated Error Message](image)

3. Perform an authenticated request as a user:
   a. Click the *Login* button to authenticate against Red Hat SSO. You will be redirected to the SSO server.
   
   b. Log in as the *Alice* user. You will be redirected back to the web interface.

   **NOTE**

   You can see the access (bearer) token in the command line output at the bottom of the page.
Figure 7.3. Authenticated Secured Example Web Interface (as Alice)

Using the greeting service
The greeting service is a protected endpoint. You will need to login first.

Log in Log out

Greeting service (as alice):
Name: World

Result:
Invoke the service to see the result.

Curl command for the command line:
```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJwYXluYWNvZG9jZSI6MCwiaWRva2tlbiI6MTA5OTY1NTk5NiwiYXNzYWdlXCI6W19odHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5Niwic2Vzc2FnZSI6IiIsImF1dGhvcgiOiJodHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsInJlZiI6eyJ0b2tlbiI6e1wiZmFsc2U6XCJodHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsImF0b25zZXMiOjE0MDAwMTUxMDQ1MjM3ODg1MTQzNCwiaXNzY3R1bGxvdyI6IkkxOTgyMjY0MjI2NDc1MDYzODQzODA5Iiwic3RyaW5ncm91bmQiOiJodHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsImFhZGxlIjoiYXV0b29ufmFtcGFyZWQ6IjIiLCJ0b2tlbiI6e1wiZmFsc2UiOjEwOCJ9.djxZK4cAKo59zX2v51JnM4g книгиdRhT7dEZ7cWpL19oKhJv8mZj1u5XVQ6WRRnGROG2Z3jg5ZbQts2Zn86.trk
```

c. Click **Invoke** again to access the Greeting service.
   Confirm that there is no exception and the JSON response payload is displayed. This means
   the service accepted your access (bearer) token and you are authorized access to the
   Greeting service.

Figure 7.4. The Result of an Authenticated Greeting Request (as Alice)

Using the greeting service
The greeting service is a protected endpoint. You will need to login first.

Log in Log out

Greeting service (as alice):
Name: World

Result:
{"id":1,"content":"Hello, World!"

Curl command for the command line:
```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJwYXluYWNvZG9jZSI6MCwiaWRva2tlbiI6MTA5OTY1NTk5NiwiYXNzYWdlXCI6W19odHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsImF1dGhvcgiOiJodHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsInJlZiI6eyJ0b2tlbiI6e1wiZmFsc2U6XCJodHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsImF0b25zZXMiOjE0MDAwMTUxMDQ1MjM3ODg1MTQzNCwiaXNzY3R1bGxvdyI6IkkxOTgyMjY0MjI2NDc1MDYzODQzODA5Iiwic3RyaW5ncm91bmQiOiJodHRwczovL2ljcmVhdGlvbi5nYW5wYWdlcy5jb20vMXBsdGlzdGFudHMvY3Vyc2lvbml0LWluc3RhbmNlXCI6MTA5OTY1NTk5NiwiY3Vyc2FnZSI6IiIsImFhZGxlIjoiYXV0b29ufmFtcGFyZWQ6IjIiLCJ0b2tlbiI6e1wiZmFsc2UiOjEwOCJ9.djxZK4cAKo59zX2v51JnM4g книгиdRhT7dEZ7cWpL19oKhJv8mZj1u5XVQ6WRRnGROG2Z3jg5ZbQts2Zn86.trk
```

d. Log out.

4. Perform an authenticated request as an administrator:
   a. Click the **Invoke** button.
      Confirm that this sends an unauthenticated request to the Greeting service.
   b. Click the **Login** button and log in as the **admin user**.
5. Click the **Invoke** button.
   The service responds with an **HTTP 403 Forbidden** status code because the **admin** user is not authorized to access the Greeting service.

### Figure 7.6. Unauthorized Error Message

#### 7.6.8. Running the Eclipse Vert.x Secured example application integration tests

This section shows you how to execute the integration tests using a Red Hat SSO test server with a pre-configured realm and example user profiles.

**Prerequisites**

- The **oc** client authenticated.

**Procedure**

```
WARNING
Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.
```
By default, the SSO server is deployed (and destroyed) as part of testing. The steps for executing integration tests are as follows:

1. In a terminal application, navigate to the directory with your project.

2. Execute the integration tests:

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

If you deployed an SSO server beforehand, e.g. by executing `oc create -f service.sso.yaml`, set the system property `skip.sso.init` to `true` when running the tests:

```
mvn clean verify -Popenshift,openshift-it -Dskip.sso.init=true
```

When executed like this, the tests will use the existing SSO server. The tests will not deploy their own SSO server, nor will they destroy the existing one.

### 7.6.9. Secured SSO resources

Follow the links below for additional information on the principles behind the OAuth2 specification and on securing your applications using Red Hat SSO and Keycloak:

- Aaron Parecki: OAuth2 Simplified
- Red Hat SSO 7.1 Documentation
- Keycloak 3.2 Documentation
- Secured for Spring Boot
- Secured for Thorntail
- Secured for Node.js

### 7.7. CACHE EXAMPLE FOR ECLIPSE VERT.X

**IMPORTANT**

The following example is not meant to be run in a production environment.

**Limitation:** Run this example application on a Minishift or CDK. You can also use a manual workflow to deploy this example to OpenShift Online Pro and OpenShift Container Platform. This example is not currently available on OpenShift Online Starter.

**Example proficiency level:** Advanced.

The Cache example demonstrates how to use a cache to increase the response time of applications.

This example shows you how to:

- Deploy a cache to OpenShift.
- Use a cache within an application.
7.7.1. How caching works and when you need it

Caches allow you to store information and access it for a given period of time. You can access information in a cache faster or more reliably than repeatedly calling the original service. A disadvantage of using a cache is that the cached information is not up to date. However, that problem can be reduced by setting an expiration or TTL (time to live) on each value stored in the cache.

Example 7.3. Caching example

Assume you have two applications: service1 and service2:

- Service1 depends on a value from service2.
  - If the value from service2 infrequently changes, service1 could cache the value from service2 for a period of time.
  - Using cached values can also reduce the number of times service2 is called.
- If it takes service1 500 ms to retrieve the value directly from service2, but 100 ms to retrieve the cached value, service1 would save 400 ms by using the cached value for each cached call.
- If service1 would make uncached calls to service2 5 times per second, over 10 seconds, that would be 50 calls.
- If service1 started using a cached value with a TTL of 1 second instead, that would be reduced to 10 calls over 10 seconds.

How the Cache example works

1. The cache, cute name, and greeting services are deployed and exposed.
2. User accesses the web frontend of the greeting service.
3. User invokes the greeting HTTP API using a button on the web frontend.
4. The greeting service depends on a value from the cute name service.
   - The greeting service first checks if that value is stored in the cache service. If it is, then the cached value is returned.
   - If the value is not cached, the greeting service calls the cute name service, returns the value, and stores the value in the cache service with a TTL of 5 seconds.
5. The web front end displays the response from the greeting service as well as the total time of the operation.
6. User invokes the service multiple times to see the difference between cached and uncached operations.
   - Cached operations are significantly faster than uncached operations.
   - User can force the cache to be cleared before the TTL expires.

7.7.2. Deploying the Cache example application to OpenShift Online

Use one of the following options to execute the Cache example application on OpenShift Online.
7.7.2.1. Deploying the example application using developers.redhat.com/launch

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Red Hat Developer Launcher web interface.

Prerequisites

- An account at OpenShift Online.

Procedure

1. Navigate to the developers.redhat.com/launch URL in a browser.
2. Follow on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.7.2.2. Authenticating the oc CLI client

To work with example applications on OpenShift Online using the oc command-line client, you must authenticate the client using the token provided by the OpenShift Online web interface.

Prerequisites

- An account at OpenShift Online.

Procedure

1. Navigate to the OpenShift Online URL in a browser.
2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.
4. Copy the oc login command.
5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your OpenShift Online account.

$ oc login OPENSHIFT_URL --token=MYTOKEN

7.7.2.3. Deploying the Cache example application using the oc CLI client

This section shows you how to build your Cache example application and deploy it to OpenShift from the command line.

Prerequisites
The example application created using developers.redhat.com/launch. For more information, see Section 7.7.2.1, “Deploying the example application using developers.redhat.com/launch”.

The `oc` client authenticated. For more information, see Section 7.7.2.2, “Authenticating the `oc` CLI client”.

**Procedure**

1. Clone your project from GitHub.
   
   ```bash
   $ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
   
   Alternatively, if you downloaded a ZIP file of your project, extract it.
   
   ```bash
   $ unzip MY_PROJECT_NAME.zip
   
2. Create a new project.
   
   ```bash
   $ oc new-project MY_PROJECT_NAME
   
3. Navigate to the root directory of your application.

4. Deploy the cache service.
   
   ```bash
   $ oc apply -f service.cache.yml
   
   **NOTE**

   If you are using an architecture other than x86_64, in the YAML file, update the image name of Red Hat Data Grid to its relevant image name in that architecture. For example, for the s390x or ppc64le architecture, update the image name to its IBM Z or IBM Power Systems image name `registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8`.

5. Use Maven to start the deployment to OpenShift.
   
   ```bash
   $ mvn clean oc:deploy -Popenshift
   
6. Check the status of your application and ensure your pod is running.
   
   ```bash
   $ oc get pods -w
   NAME                             READY     STATUS      RESTARTS   AGE
   cache-server-123456789-aaaaa             1/1       Running     0          8m
   MY_APP_NAME-cutename-1-bbbbbb       1/1       Running     0          4m
   MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
   MY_APP_NAME-greeting-1-ccccccc       1/1       Running     0          3m
   MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
   
   Your 3 pods should have a status of **Running** once they are fully deployed and started.

7. After your example application is deployed and started, determine its route.

   **Example Route Information**
**Example route information**

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

The route information of a pod gives you the base URL which you use to access it. In the example above, you would use `http://MY_APP_NAME-greeting- MY_PROJECT_NAME.OPENSIGHT_HOSTNAME` as the base URL to access the greeting service.

### 7.7.3. Deploying the Cache example application to Minishift or CDK

Use one of the following options to execute the Cache example application locally on Minishift or CDK:

- **Using Launcher**
- **Using the `oc` CLI client**

Although each method uses the same `oc` commands to deploy your application, using Launcher provides an automated deployment workflow that executes the `oc` commands for you.

#### 7.7.3.1. Getting the Launcher tool URL and credentials

You need the Launcher tool URL and user credentials to create and deploy example applications on Minishift or CDK. This information is provided when the Minishift or CDK is started.

**Prerequisites**

- The Launcher tool installed, configured, and running.

**Procedure**

1. Navigate to the console where you started Minishift or CDK.
2. Check the console output for the URL and user credentials you can use to access the running Launcher:

**Example Console Output from a Minishift or CDK Startup**

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

You are logged in as:
  User: developer
```
7.7.3.2. Deploying the example application using the Launcher tool

This section shows you how to build your REST API Level 0 example application and deploy it to OpenShift from the Launcher web interface.

Prerequisites

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.7.3.1, “Getting the Launcher tool URL and credentials”.

Procedure

1. Navigate to the Launcher URL in a browser.
2. Follow the on-screen instructions to create and launch your example application in Eclipse Vert.x.

7.7.3.3. Authenticating the oc CLI client

To work with example applications on Minishift or CDK using the oc command-line client, you must authenticate the client using the token provided by the Minishift or CDK web interface.

Prerequisites

- The URL of your running Launcher instance and the user credentials of your Minishift or CDK. For more information, see Section 7.7.3.1, “Getting the Launcher tool URL and credentials”.

Procedure

1. Navigate to the Minishift or CDK URL in a browser.
2. Click on the question mark icon in the top right-hand corner of the Web console, next to your user name.
4. Copy the oc login command.
5. Paste the command in a terminal. The command uses your authentication token to authenticate your oc CLI client with your Minishift or CDK account.

    $ oc login OPENSHIFT_URL --token=MYTOKEN

7.7.3.4. Deploying the Cache example application using the oc CLI client

This section shows you how to build your Cache example application and deploy it to OpenShift from the command line.
Prerequisites

- The example application created using Launcher tool on a Minishift or CDK. For more information, see Section 7.7.3.2, “Deploying the example application using the Launcher tool”.
- Your Launcher tool URL.
- The `oc` client authenticated. For more information, see Section 7.7.3.3, “Authenticating the `oc` CLI client”.

Procedure

1. Clone your project from GitHub.
   ```
   $ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
   ```
   Alternatively, if you downloaded a ZIP file of your project, extract it.
   ```
   $ unzip MY_PROJECT_NAME.zip
   ```
2. Create a new project.
   ```
   $ oc new-project MY_PROJECT_NAME
   ```
3. Navigate to the root directory of your application.
4. Deploy the cache service.
   ```
   $ oc apply -f service.cache.yml
   ```
   **NOTE**
   If you are using an architecture other than x86_64, in the YAML file, update the image name of Red Hat Data Grid to its relevant image name in that architecture. For example, for the s390x or ppc64le architecture, update the image name to its IBM Z or IBM Power Systems image name `registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8`.
5. Use Maven to start the deployment to OpenShift.
   ```
   $ mvn clean oc:deploy -Popenshift
   ```
6. Check the status of your application and ensure your pod is running.
   ```
   $ oc get pods -w
   NAME                             READY     STATUS      RESTARTS   AGE
   cache-server-123456789-aaaaa       1/1       Running     0         8m
   MY_APP_NAME-cutename-1-bbbbb       1/1       Running     0         4m
   MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0         7m
   MY_APP_NAME-greeting-1-cccccc      1/1       Running     0         3m
   MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0         3m
   ```
Your 3 pods should have a status of **Running** once they are fully deployed and started.

7. After your example application is deployed and started, determine its route.

**Example Route Information**

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

The route information of a pod gives you the base URL which you use to access it. In the example above, you would use http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSSHIFT_HOSTNAME as the base URL to access the greeting service.
```

7.7.4. Deploying the Cache example application to OpenShift Container Platform

The process of creating and deploying example applications to OpenShift Container Platform is similar to OpenShift Online:

**Prerequisites**

- The example application created using developers.redhat.com/launch.

**Procedure**

- Follow the instructions in Section 7.7.2, “Deploying the Cache example application to OpenShift Online”, only use the URL and user credentials from the OpenShift Container Platform Web Console.

7.7.5. Interacting with the unmodified Cache example application

Use the default web interface to interact with the unmodified Cache example application, and see how storing frequently accessed data can shorten the time needed to access your service.

**Prerequisites**

- Your application deployed

**Procedure**

1. Navigate to the **greeting** service using your browser.

2. Click **Invoke the service** once.

   Notice the **duration** value is above **2000**. Also notice the cache state has changed from **No cached value** to **A value is cached**.

3. Wait 5 seconds and notice cache state has changed back to **No cached value**.
The TTL for the cached value is set to 5 seconds. When the TTL expires, the value is no longer cached.

4. Click *Invoke the service* once more to cache the value.

5. Click *Invoke the service* a few more times over the course of a few seconds while cache state is *A value is cached*. Notice a significantly lower *duration* value since it is using a cached value. If you click *Clear the cache*, the cache is emptied.

### 7.7.6. Running the Cache example application integration tests

This example application includes a self-contained set of integration tests. When run inside an OpenShift project, the tests:

- Deploy a test instance of the application to the project.
- Execute the individual tests on that instance.
- Remove all instances of the application from the project when the testing is done.

#### WARNING

Executing integration tests removes all existing instances of the example application from the target OpenShift project. To avoid accidentally removing your example application, ensure that you create and select a separate OpenShift project to execute the tests.

### Prerequisites

- The *oc* client authenticated
- An empty OpenShift project

### Procedure

Execute the following command to run the integration tests:

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

### 7.7.7. Caching resources

More background and related information on caching can be found here:

- [Cache for Spring Boot](#)
- [Cache for Thorntail](#)
- [Cache for Node.js](#)
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:

   ```yaml
   spec:
     template:
       spec:
         containers:
           readinessProbe:
             httpGet:
               path: /path/to/probe
               port: 8080
               scheme: HTTP
               ...   
   
   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
   ```
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 4.0 only supports builder images based on OpenJDK 8 and OpenJDK 11. Oracle JDK and OpenJDK 9 builder images are not supported.

Example pom.xml

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

  
  <property name="jkube.generator.from" value="registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:latest"/>

  
  <build>
    
    
    
    <plugins>
      
      
      
      <plugin>
        
        
        
        <groupId>org.jkube</groupId>
        <artifactId>jkube</artifactId>
        <version>0.19.0</version>
        <configuration>
          
          
          
          <generatorFrom>
            registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:latest
          </generatorFrom>
        </configuration>
      </plugin>
    </plugins>
  </build>

  
  
  
</pom>
```

- 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 Minishift or CDK:

- Setting Up a Nexus Mirror for Maven
APPENDIX F. PROFICIENCY LEVELS

Each available example teaches concepts that require certain minimum knowledge. This requirement varies by example. The minimum requirements and concepts are organized in several levels of proficiency. In addition to the levels described here, you might need additional information specific to each example.

**Foundational**
The examples rated at Foundational proficiency generally require no prior knowledge of the subject matter; they provide general awareness and demonstration of key elements, concepts, and terminology. There are no special requirements except those directly mentioned in the description of the example.

**Advanced**
When using Advanced examples, the assumption is that you are familiar with the common concepts and terminology of the subject area of the example in addition to Kubernetes and OpenShift. You must also be able to perform basic tasks on your own, for example, configuring services and applications, or administering networks. If a service is needed by the example, but configuring it is not in the scope of the example, the assumption is that you have the knowledge to properly configure it, and only the resulting state of the service is described in the documentation.

**Expert**
Expert examples require the highest level of knowledge of the subject matter. You are expected to perform many tasks based on feature-based documentation and manuals, and the documentation is aimed at most complex scenarios.
APPENDIX G. GLOSSARY

G.1. PRODUCT AND PROJECT NAMES

Developer Launcher (developers.redhat.com/launch)

developers.redhat.com/launch called Developer Launcher is a stand-alone getting started experience provided by Red Hat. It helps you get started with cloud-native development on OpenShift. It contains functional example applications that you can download, build, and deploy on OpenShift.

Minishift or CDK

An OpenShift cluster running on your machine using Minishift.

G.2. TERMS SPECIFIC TO DEVELOPER LAUNCHER

Example

An application specification, for example a web service with a REST API. Examples generally do not specify which language or platform they should run on; the description only contains the intended functionality.

Example application

A language-specific implementation of a particular example on a particular runtime. Example applications are listed in an examples catalog. For example, an example application is a web service with a REST API implemented using the Thorntail runtime.

Examples Catalog

A Git repository that contains information about example applications.

Runtime

A platform that executes an example application. For example, Thorntail or Eclipse Vert.x.