



Red Hat OpenShift Development II: Creating Microservices with Red Hat OpenShift Application Runtimes



RHOAR 1.0 DO292
Red Hat OpenShift Development II: Creating Microservices
with Red Hat OpenShift Application Runtimes
Edition 120200501
Publication date 20180923

Authors: Jim Rigsbee, Fernando Lozano, Zach Guterman, Ricardo Jun Taniguchi,
Ravi Srinivasan
Editor: Seth Kenlon

Copyright © 2018 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2018 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but
not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of
Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat,
Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details
contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send
email to training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, Hibernate, Fedora, the Infinity logo, and RHCE are
trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a registered trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or
other countries.

The OpenStack® word mark and the Square O Design, together or apart, are trademarks or registered trademarks
of OpenStack Foundation in the United States and other countries, and are used with the OpenStack Foundation's
permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the
OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Sajith Sugathan, Heather Charles, Dave Sacco, Rob Locke, Achyut Madhusudhan,
Rudolf Kastl, George Hacker

Document Conventions	vii
Introduction	ix
Red Hat OpenShift Development II: Creating Microservices with Red Hat OpenShift	ix
Application Runtimes	ix
Orientation to the Classroom Environment	xi
Internationalization	xiv
1. Deploying Microservices to an OpenShift Cluster	1
Describing Red Hat OpenShift Application Runtimes	2
Quiz: Describing Red Hat OpenShift Application Runtimes	8
Describing Microservice Architecture Patterns	12
Quiz: Describing Microservice Architecture Patterns	17
Deploying Microservices with the Fabric8 Maven Plug-in	21
Guided Exercise: Deploying Microservices with the Fabric8 Maven Plug-in	25
Lab: Deploying Microservices to an OpenShift Cluster	33
Summary	41
2. Deploying Microservices with the WildFly Swarm Runtime	43
Developing an Application with the WildFly Swarm Runtime	44
Guided Exercise: Developing an Application with the WildFly Swarm Runtime	52
Configuring a Maven Project for WildFly Swarm	58
Guided Exercise: Configuring a Maven Project for WildFly Swarm	66
Describing the Coolstore Microservice Application	73
Quiz: Describing the Coolstore Microservice Application	76
Guided Exercise: Deploying Microservices with the WildFly Swarm Runtime	80
Summary	99
3. Developing Microservices with the Vert.x Runtime	101
Developing an Application with the Vert.x Runtime	102
Guided Exercise: Developing an Application with the Vert.x Runtime	116
Configuring a Maven Project for Vert.x	125
Guided Exercise: Configuring a Maven Project for Vert.x	131
Guided Exercise: Developing Microservices with the Vert.x Runtime	139
Summary	169
4. Developing Microservices with the Spring Boot Runtime	171
Developing an Application with the Spring Boot Runtime	172
Guided Exercise: Developing an Application with the Spring Boot Runtime	178
Configuring a Maven Project for Spring Boot	185
Guided Exercise: Configuring a Maven Project for Spring Boot	192
Guided Exercise: Developing Microservices with the Spring Boot Runtime	203
Summary	220
5. Developing an API Gateway	221
Developing an API Gateway for Microservices	222
Guided Exercise: Developing an API Gateway for Microservices	231
Lab: Developing an API Gateway	238
Summary	258
6. Implementing Fault Tolerance with Hystrix	259
Creating a Fault Tolerant Microservice	260
Guided Exercise: Making a Microservice Fault Tolerant	268
Deploying Hystrix Dashboard and Turbine	277
Guided Exercise: Deploying Hystrix Turbine	282
Guided Exercise: Implementing Fault Tolerance with Hystrix	291
Summary	305
A. Managing Git Branches	307

Managing Git Branches	308
B. Working With Red Hat JBoss Developer Studio	313
Working with Red Hat JBoss Developer Studio	314

Document Conventions



References

"References" describe where to find external documentation relevant to a subject.



Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.

Introduction

Red Hat OpenShift Development II: Creating Microservices with Red Hat OpenShift Application Runtimes

Microservices are an important component of modern application architectures. Red Hat has introduced solutions for the creation and deployment of microservices through its developer programs and Red Hat OpenShift Application Runtimes.

Red Hat OpenShift Development II: Creating Microservices with Red Hat OpenShift Application Runtimes (DO292) teaches students about three of these runtimes: WildFly Swarm, Vert.x, and Spring Boot. Students will develop multiple microservices using the three runtimes and deploy them on a Red Hat OpenShift Container Platform cluster.

Course Objectives

- Deploying Microservices to an OpenShift Cluster.
- Developing Microservices with WildFly Swarm, Vert.x and Spring Boot Runtimes.

Audience

- Software Developers
- Software Architects

Prerequisites

- *Red Hat Application Development I: Implementing Microservice Architectures (JB283)* or equivalent experience with microservice architecture.
- RHCSA or higher is helpful for navigation and usage of the command line, but not required.
- Earning the *Red Hat Certified Specialist in Containerized Application Development (EX288)*, attending *Red Hat OpenShift Development I: Containerizing Applications (DO288)*, or experience developing and deploying containerized applications to an OpenShift cluster.

The course hands-on activities also assume that the student is familiar with the **git** command, local and remote Git repository branches, and the *Red Hat JBoss Developer Studio* integrated development environment (IDE). If you are not familiar with these tools, or you need to refresh your knowledge, please refer to the appendices.

Orientation to the Classroom Environment

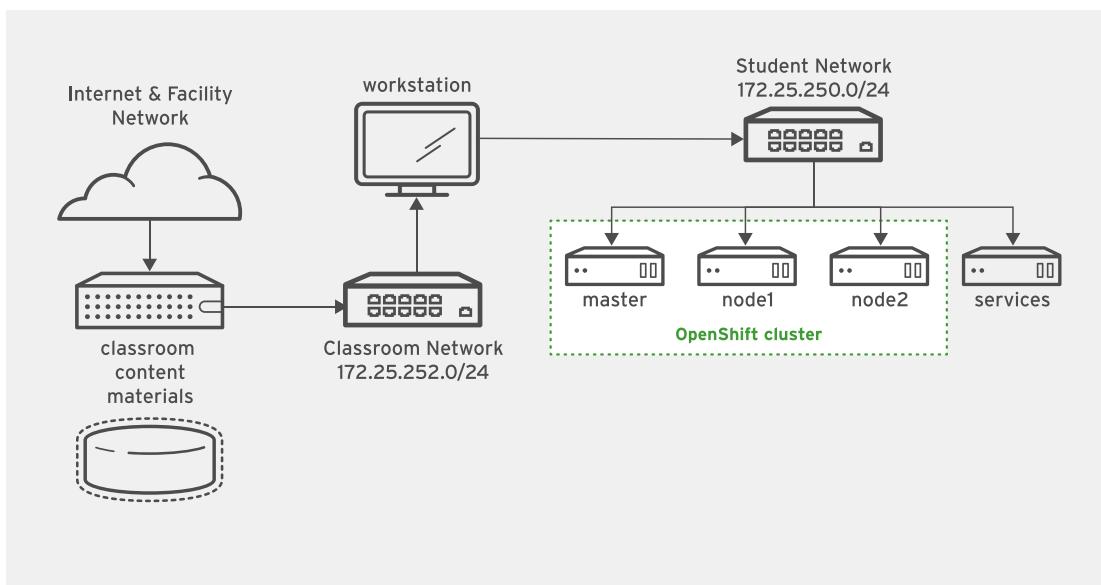


Figure 0.1: Classroom environment

In this course, the main computer system used for hands-on learning activities is **workstation**. Four other machines will also be used by students for these activities. These are **master**, **node1**, **node2**, and **services**. All four of these systems are in the **lab.example.com** DNS domain.

All student computer systems have a standard user account, **student**, which has the password **student**. The **root** password on all student systems is **redhat**.

Classroom Machines

Machine name	IP addresses	Role
workstation.lab.example.com	172.25.250.254	Graphical workstation used for system administration
master.lab.example.com	172.25.250.10	Master of the OpenShift cluster
node1.lab.example.com	172.25.250.11	Node in the OpenShift cluster
node2.lab.example.com	172.25.250.12	Node in the OpenShift cluster
services.lab.example.com	172.25.250.13	Provides supporting services such as: container image registry, nexus repository, and Git server

One additional function of **workstation** is that it acts as a router between the network that connects student machines and the classroom network. If **workstation** is down, other student machines are only able to access systems on the student network.

There are several systems in the classroom that provide supporting services. Two servers, **content.example.com** and **materials.example.com** are sources for software and lab materials used in hands-on activities. Information on how to use these servers will be provided in the instructions for those activities.

Controlling Your Station

The top of the console describes the state of your machine.

Machine States

State	Description
none	Your machine has not yet been started. When started, your machine will boot into a newly initialized state (the disk will have been reset).
starting	Your machine is in the process of booting.
running	Your machine is running and available (or, when booting, soon will be.)
stopping	Your machine is in the process of shutting down.
stopped	Your machine is completely shut down. Upon starting, your machine will boot into the same state as when it was shut down (the disk will have been preserved).
impaired	A network connection to your machine cannot be made. Typically this state is reached when a student has corrupted networking or firewall rules. If the condition persists after a machine reset, or is intermittent, please open a support case.

Depending on the state of your machine, a selection of the following actions will be available to you.

Machine Actions

Action	Description
Start Station	Start ("power on") the machine.
Stop Station	Stop ("power off") the machine, preserving the contents of its disk.
Reset Station	Stop ("power off") the machine, resetting the disk to its initial state. Caution: Any work generated on the disk will be lost.
Refresh	Refresh the page will re-probe the machine state.

The Station Timer

Your Red Hat Online Learning enrollment entitles you to a certain amount of computer time. In order to help you conserve your time, the machines have an associated timer, minutes when your machine is started.

The timer operates as a "dead man's switch," which decrements as your machine is running. If the timer is winding down to 0, you may choose to increase the timer.

Internationalization

Language Support

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

Per-user Language Selection

Users may prefer to use a different language for their desktop environment than the system-wide default. They may also want to set their account to use a different keyboard layout or input method.

Language Settings

In the GNOME desktop environment, the user may be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the Region & Language application.

Run the command **gnome-control-center region**, or from the top bar, select **(User) → Settings**. In the window that opens, select Region & Language. The user can click the **Language** box and select their preferred language from the list that appears. This will also update the **Formats** setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including **gnome-terminal**, started inside it. However, they do not apply to that account if accessed through an **ssh** login from a remote system or a local text console (such as **tty2**).



Note

A user can make their shell environment use the same **LANG** setting as their graphical environment, even when they log in through a text console or over **ssh**. One way to do this is to place code similar to the following in the user's **~/.bashrc** file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
    | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, or other languages with a non-Latin character set may not display properly on local text consoles.

Individual commands can be made to use another language by setting the **LANG** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date  
jeu. avril 24 17:55:01 CDT 2014
```

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to check the current value of **LANG** and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 automatically uses the IBus input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The Region & Language application can also be used to enable alternative input methods. In the Region & Language application's window, the **Input Sources** box shows what input methods are currently available. By default, **English (US)** may be the only available method. Highlight **English (US)** and click the **keyboard** icon to see the current keyboard layout.

To add another input method, click the **+** button at the bottom left of the **Input Sources** window. An **Add an Input Source** window will open. Select your language, and then your preferred input method or keyboard layout.

Once more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese **Japanese (Kana Kanji)** input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may find also this useful. For example, under **English (United States)** is the keyboard layout **English (international AltGr dead keys)**, which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary-shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



Note

Any Unicode character can be entered in the GNOME desktop environment if the user knows the character's Unicode code point, by typing **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl+Shift+U**, then **03bb**, then **Enter**.

System-wide Default Language Settings

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, *root* can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it will display the current system-wide locale settings.

To set the system-wide language, run the command `localectl set-locale LANG=locale`, where `locale` is the appropriate `$LANG` from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in `/etc/locale.conf`.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from Region & Language and clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the login screen will also adjust the system-wide default language setting stored in the `/etc/locale.conf` configuration file.



Important

Local text consoles such as `tty2` are more limited in the fonts that they can display than `gnome-terminal` and `ssh` sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through `localectl` for both local text virtual consoles and the X11 graphical environment. See the `localectl(1)`, `kbd(4)`, and `vconsole.conf(5)` man pages for more information.

Language Packs

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available langpacks, run `yum langavailable`. To view the list of langpacks currently installed on the system, run `yum langlist`. To add an additional langpack to the system, run `yum langinstall code`, where `code` is the code in square brackets after the language name in the output of `yum langavailable`.



References

`locale(7)`, `localectl(1)`, `kbd(4)`, `locale.conf(5)`, `vconsole.conf(5)`, `unicode(7)`, `utf-8(7)`, and `yum-langpacks(8)` man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in `localectl` can be found in the file `/usr/share/X11/xkb/rules/base.lst`.

Language Codes Reference

Language Codes

Language	\$LANG value
English (US)	en_US.utf8
Assamese	as_IN.utf8

Language	\$LANG value
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8

Chapter 1

Deploying Microservices to an OpenShift Cluster

Goal

Deploy an application based on a microservice architecture to an OpenShift cluster.

Objectives

- Describe the features of a Red Hat OpenShift Application Runtimes subscription.
- Describe the major patterns implemented in microservice architectures.
- Deploy a microservice to an OpenShift cluster using the Maven Fabric8 plug-in.

Sections

- Describing Red Hat OpenShift Application Runtimes (and Quiz)
- Describing Microservice Architecture Patterns (and Quiz)
- Deploying Microservices with the Maven Fabric8 Plug-in (and Guided Exercise)

Lab

Deploying Microservices to an OpenShift Cluster

Describing Red Hat OpenShift Application Runtimes

Objective

After completing this section, students should be able to describe the features of a Red Hat OpenShift Application Runtimes subscription.

Introducing Red Hat OpenShift Application Runtimes

Red Hat OpenShift Application Runtimes (RHOAR) is Red Hat's development platform for cloud-native and microservices applications. RHOAR provides an opinionated approach for developing microservices applications that targets OpenShift as the deployment platform.

Red Hat OpenShift Application Runtimes supports multiple runtimes, languages, frameworks, and architectures. It offers the choice and flexibility to pick the right frameworks and runtimes for the right job. Applications developed with RHOAR can run on any cloud infrastructure where Red Hat OpenShift Container Platform can run, offering freedom from vendor lock-in.

Sometimes developers are overwhelmed by the number of choices offered by any particular runtime to implement each microservices pattern. RHOAR guides a developer to implement each pattern in a way that takes advantage of the OpenShift infrastructure.

A Red Hat OpenShift Application Runtimes subscription provides:

- Access to Red Hat-built and supported binaries for selected microservices development frameworks and runtimes.
- Access to Red Hat-built and supported binaries for integration modules that replaces or enhances a framework's implementation of a microservices pattern to use OpenShift features.
- Developer support for writing applications using selected microservices development frameworks, runtimes, integration modules, and integration with selected external services, such as database servers.
- Production support for deploying applications using selected microservices development frameworks, runtimes, integration modules, and integrations on a supported OpenShift cluster.

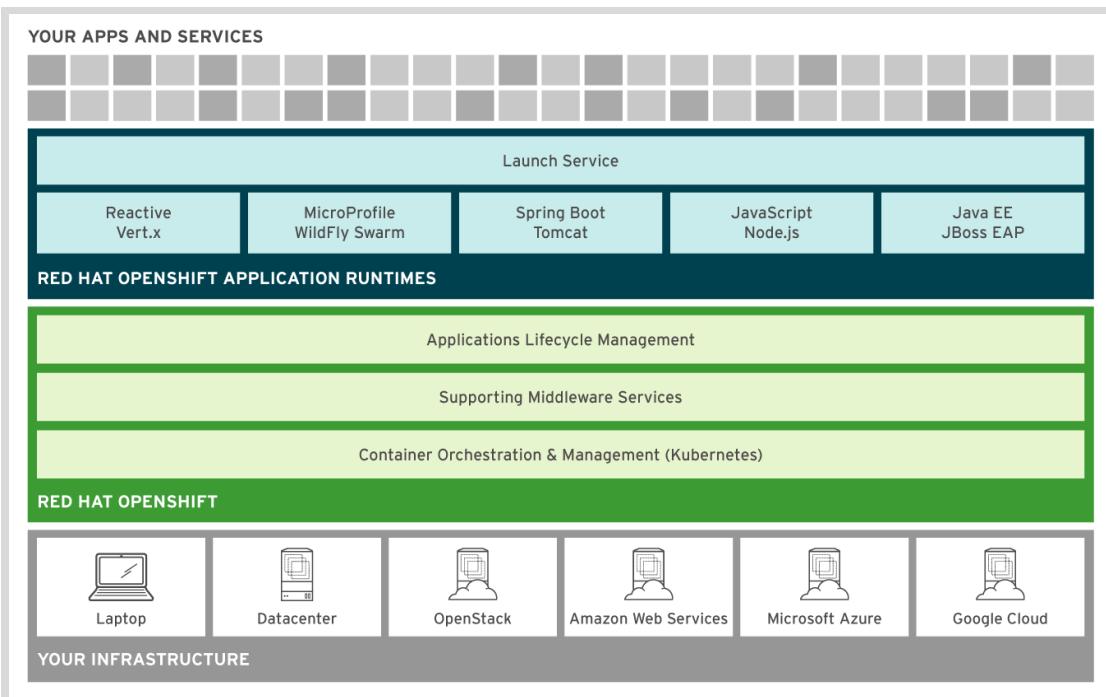


Figure 1.1: Red Hat OpenShift Application Runtimes architecture

The production environment still requires a subscription to Red Hat OpenShift Container Platform or another supported member of the OpenShift product family, such as Red Hat OpenShift Container Dedicated.

There are two subscription levels for RHOAR:

- Red Hat OpenShift Application Runtimes (RHOAR) includes supported frameworks and runtimes for cloud-native application development and deployment.
- Red Hat OpenShift Application Runtimes Plus (RHOAR Plus) adds Red Hat JBoss EAP entitlements for organizations that are migrating monoliths to microservices.

All Red Hat JBoss EAP subscriptions also include RHOAR Plus entitlements so the Red Hat JBoss EAP for OpenShift subscription is now equivalent to the RHOAR Plus subscription.

Working with Red Hat OpenShift Application Runtimes

A developer works with Red Hat OpenShift Application Runtimes by integrating a number of software packages into the development workflow:

- Development frameworks and runtimes
- Container images
- Container orchestration engine
- Build tool and IDE
- The launcher tool

Development Frameworks and Runtimes

Red Hat supports the following development frameworks and runtimes as part of the Red Hat OpenShift Application Runtimes subscription:

WildFly Swarm

An implementation of the MicroProfile standard that builds cloud-native applications using both existing and new Java EE APIs.

Eclipse Vert.x

A reactive, low-latency development framework based on asynchronous I/O and event streams.

Spring Boot

A cloud-native development framework based on the popular Spring Framework and auto configuration.

Netflix OSS libraries

A set of libraries that implement common microservices patterns, among them the Hystrix circuit-breaker and the Ribbon inter-process communication library.

For each framework, a very specific subset of modules, releases, and integrations are supported or tested by Red Hat. The Red Hat OpenShift Application Runtimes Release Notes document provides a list of modules and integrations with support status.

Container Images

Container images for cloud-native Java applications based on RHOAR use the OpenJDK 8 Image for Java Applications container image, from the Red Hat Container Catalog, as their parent image.

The OpenJDK 8 Image for Java Applications container image provides a Red Hat Enterprise Linux 7 (RHEL 7) Operating system distribution plus the certified build of OpenJDK 8 for RHEL 7. This is the same container image as the Red Hat Java S2I for OpenShift container image from the Red Hat JBoss Middleware for OpenShift product.

The development frameworks and their dependencies are added to the application layer as regular Maven dependencies. There is no need for specialized parent container images for each Java framework and runtime.

Container Orchestration Engine

The Red Hat OpenShift Application Runtimes subscription supports deploying applications to OpenShift only, in any of its supported incarnations: Red Hat OpenShift Container Platform, Red Hat OpenShift Dedicated, and Red Hat OpenShift Online.

Developers also have two options to run an all-in-one OpenShift cluster on their local workstation:

- The **oc cluster up** command runs all OpenShift services in a single container, and requires a Linux developer workstation.
- The Red Hat Container Development Kit (CDK) runs all OpenShift services in a virtual machine. The CDK supports Linux, Windows, and MacOS workstations.

Whatever the flavor of OpenShift a developer chooses, a local installation of the OpenShift Command-Line Interface (CLI) and access credentials to an OpenShift cluster are required.

Build Tool and IDE

Developers working with Red Hat OpenShift Application Runtimes are entitled, but not required, to use the Red Hat JBoss Developer Studio (JBDS) Integrated Development Environment (IDE), based on the Eclipse Platform.

JBDS provides a number of features that help work with the RHOAR frameworks and also with OpenShift cluster, including the CDK.

Java developers usually automate application builds using the Maven tool. Red Hat OpenShift Application Runtimes is tested with a plug-in from the Fabric8 project: the *Fabric8 Maven Plug-in* (FMP), which provides a number of custom Maven goals to automate building application container images and deploying them on OpenShift clusters.

The Launcher Tool

Red Hat OpenShift Application Runtimes also supports another component from the Fabric8 project: the Launcher tool, an interactive web-based application that generates sample applications to illustrate RHOAR features.

The *Fabric8 Launcher Tool* can either be executed online, as part of the Red Hat Developer Program web site, or deployed into an OpenShift cluster. It generates complete sample projects based on a choice of a framework and a mission. Missions are typical microservice development tasks, such as:

- REST API Level 0
- Externalized configuration
- Health check

The developer also chooses whether to deploy the generated application on OpenShift Online, or to download the source code to deploy manually on a local OpenShift cluster.

Choosing Between Supported, Tested, and Community Modules

Library dependencies, and other Maven modules provided by the Red Hat OpenShift Application Runtimes, are classified into a number categories:

- Supported
- Tested, or Certified
- Community, or Unsupported

Supported modules are provided by Maven repositories Red Hat managed by Red Hat, while the other categories are provided by third party repositories, including public ones, such as Maven Central, and company-sponsored ones.

Each release of RHOAR is composed of a different mix of supported, tested, and community modules. The support status of each module may impact your organization decision to use it (or not) for a given application.

A summary of supported and unsupported modules for the first release of Red Hat OpenShift Application Runtimes 1.0 follows:

- WildFly Swarm **7.1.0.redhat-77**: Red Hat supports the core, Web Profile, and Microservices modules, plus a few other essential modules, such as Metrics. Additional modules from WildFly Swarm **2018.3.3**, such as Arquillian and Swagger, are tested or unsupported.
- Vert.x **3.5.1.redhat-003**: Red Hat supports essential modules, such as core, codegen, web, and config. Other Vert.x modules not included in the Red Hat distribution, such as Apache Kafka and Groovy, are unsupported.

- Spring Boot **1.5.12.RELEASE**: Red Hat supports Spring Boot modules to integrate with Kubernetes and OpenShift APIs, and with an embedded JBoss Web Server (Apache Tomcat). Red Hat also supports some modules, such as REST APIs, using Apache CXF, and JPA persistence using Hibernate.

Important modules such Spring DI, Spring Boot bootstrap, and Netflix OSS services, are only certified. A number of other popular modules, such as Spring Web and Spring MVC, are unsupported, sometimes because they fill the same niche as a supported module.

Spring Boot and Spring Framework do not follow a consistent version number scheme, and a number of third-party modules are considered part of a Spring distribution but keep their own Maven artifact IDs and version strings.

Supported modules

Many developers are used to working with community modules from popular open source projects, and never think about the importance of using supported modules.

Supported modules from RHOAR provide similar benefits compared to supported packages from RHEL 7 and APIs from JBoss EAP, among them:

- Stability: Red Hat provides updates including bug and security fixes without breaking backwards-compatibility. These fixes are backported from upstream projects when required. Community projects usually apply updates only to the latest release, which may not be backward-compatible with the release currently in use by an application.
- Integration: Red Hat tests a number of components with each release of RHOAR, offering guarantees that not only multiple modules work together inside the same application, but also that they work with a number of external services such as databases and messaging services.

A supported module has a release string that includes a **.redhat-###** suffix. The prefix identifies the community release it is based upon. Multiple **.redhat-###** suffixes for the same community release mean that either bug fixes were backported, or Red Hat added an emergency fix that is not accepted by the upstream yet.

Tested or Certified Modules

A third party vendor or open source project provides these modules and controls the naming convention for the version string. Either the vendor or Red Hat follow a formal process to validate that each module is compatible with RHOAR.

Red Hat provides support for tested modules in commercially reasonable basis, and works with the vendor to develop bug fixes and security fixes, but each vendor controls its own policies regarding frequency of updates and backwards compatibility.

Community or Unsupported Modules

Red Hat performs a basic set of unit and integration tests for these modules, and provides no assurances regarding bug and security fixes.

Many developers assume that "unsupported" means "does not work". Actually it means "use at your own risk". It might even be that the vendor of an unsupported module does a good job of providing bug and security fixes that are compatible with RHOAR, but this needs to be evaluated on a case-by-case basis.

Implementing Microservices Patterns with RHOAR

A key point of microservices architectures is to implement patterns such as load balancing, circuit breakers, externalized configurations, health, and metrics. Each framework provides its own way to implement them, and sometimes a framework provides multiple ways to implement a given pattern.

There are also popular libraries dedicated to one or just a few microservices patterns, such as the popular Netflix OSS libraries. Many frameworks provide, as an option, the ability to use these libraries as a replacement for their own implementation of the same pattern.

Red Hat OpenShift Application Runtimes chooses the best implementation of a pattern for each framework, and supports it. Sometimes this implementation comes from the framework itself, sometimes from Netflix OSS. But many times RHOAR provides a different implementation because that particular pattern is already provided by the OpenShift platform, and does not need to be fully implemented by the framework.

A few examples of using OpenShift to implement microservices patterns are:

- Service discovery using Kubernetes services
- Externalized configuration using OpenShift configuration maps
- TLS and PKI certificates using Kubernetes secrets
- Authentication using Keycloak
- Health using OpenShift readiness and liveness probes.

Sometimes these require a completely new framework module that uses OpenShift REST APIs. Other times, there is nothing to change anywhere in the framework because OpenShift implements the pattern transparently to the microservices application.



References

JBoss Enterprise Maven Repository

<https://access.redhat.com/maven-repository>

OpenJDK 8 Image for Java Applications in the Red Hat Container Catalog.

<https://access.redhat.com/containers/#/registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift>

Online version of the Fabric8 Launcher tool at the Red Hat Developer Program web site

<https://developers.redhat.com/launch/>

Further information about supported and tested Maven artifact releases is available in the *Red Hat OpenShift Application Runtimes 1.0 Release Notes* at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/red_hat_openshift_application_runtimes_release_notes/

► Quiz

Describing Red Hat OpenShift Application Runtimes

Choose the correct answers to the following questions:

- 1. Which of the following two components are fully supported as part of a Red Hat OpenShift Application Runtimes (RHOAR) subscription? (Choose two.)
- a. Multi-node OpenShift clusters to deploy and test an application designed as a set of microservices.
 - b. Selected community libraries that implement microservices patterns.
 - c. Integration between selected microservices frameworks and OpenShift features such as secrets.
 - d. Red Hat-maintained builds of popular microservices frameworks.
 - e. Selected third-party and community builds of popular microservices frameworks.
 - f. Infrastructure services such as CI/CD tools and container registries.
- 2. A serious vulnerability was found in a supported module from RHOAR. One of your production applications requires the module, which is maintained by an open source project. The community is working on a security fix only for their latest release, which made changes incompatible with the release included in RHOAR. Given this scenario, which of the following actions fixes the vulnerability to the production application with minimal impact:
- a. Open a support case and ask Red Hat to provide a hot fix for the current RHOAR release. When the fix is released, rebuild the application to include it.
 - b. Open a support case and ask Red Hat to provide a hot fix for the latest community release, then rewrite the application to use the latest community release.
 - c. Just rewrite the application to use the latest community release, because Red Hat is not expected to provide hot fixes for open source components developed by third parties.
 - d. Open a support case and ask Red Hat to add the latest community release to a RHOAR update, then rewrite the application to use the latest community release.

- 3. The Spring Boot framework supports multiple approaches to implement a REST API endpoint. Some of them are not supported or tested in RHOAR. The development team is divided on which approach to take because each team member has experience with a different one, and none of them has previous experience deploying to OpenShift. Given this scenario, which recommendation would you give to the development team to make the best use of both internal and external resources?
- Poll the developers to find the approach that most developers consider the best one and recommend that approach, despite its support status in RHOAR.
 - Research the CVE history of each approach and recommend the one with the lowest number of issues, regardless of its support status in RHOAR.
 - Research which approach is more popular among the Spring framework community, because it should be easier to find help with this one.
 - Research which approach is supported under RHOAR and ask the development team to use it.
- 4. Which of the following four are supported deployment targets for an application built using RHOAR? (Choose four.)
- A container engine under a Red Hat Enterprise Linux Atomic Host server.
 - An OpenShift cluster running under a VM created using the Red Hat Container Development Kit (CDK).
 - An OpenShift Online account.
 - A Red Hat OpenShift Container Platform cluster running on bare metal Red Hat Enterprise Linux servers.
 - An OpenShift Origin cluster running in bare metal Red Hat Enterprise Linux servers.
 - A Red Hat OpenShift Container Platform cluster running in Red Hat Enterprise Linux servers from a public cloud provider.
 - A Kubernetes cluster from a public cloud provider container service.
- 5. The release string for Spring Boot version tested with the first release of Red Hat OpenShift Application Runtimes is 1.5.12.RELEASE. What does this release string indicate about the level of support status that the Spring Boot framework receives directly from Red Hat?
- Red Hat will provide emergency hot fixes for core Spring Boot components whenever a security issue is found.
 - Red Hat is not able to deliver hot fixes for Spring Boot core components, but will collaborate with the Spring Boot community and its vendor to develop and deliver fixes for core Spring Boot components whenever a bug or a security issue is found.
 - Red Hat will not work with the Spring Boot community to develop or deliver fixes for core Spring Boot components. Customers use Spring Boot at their own risk.

► Solution

Describing Red Hat OpenShift Application Runtimes

Choose the correct answers to the following questions:

- 1. Which of the following two components are fully supported as part of a Red Hat OpenShift Application Runtimes (RHOAR) subscription? (Choose two.)
- a. Multi-node OpenShift clusters to deploy and test an application designed as a set of microservices.
 - b. Selected community libraries that implement microservices patterns.
 - c. Integration between selected microservices frameworks and OpenShift features such as secrets.
 - d. Red Hat-maintained builds of popular microservices frameworks.
 - e. Selected third-party and community builds of popular microservices frameworks.
 - f. Infrastructure services such as CI/CD tools and container registries.
- 2. A serious vulnerability was found in a supported module from RHOAR. One of your production applications requires the module, which is maintained by an open source project. The community is working on a security fix only for their latest release, which made changes incompatible with the release included in RHOAR. Given this scenario, which of the following actions fixes the vulnerability to the production application with minimal impact:
- a. Open a support case and ask Red Hat to provide a hot fix for the current RHOAR release. When the fix is released, rebuild the application to include it.
 - b. Open a support case and ask Red Hat to provide a hot fix for the latest community release, then rewrite the application to use the latest community release.
 - c. Just rewrite the application to use the latest community release, because Red Hat is not expected to provide hot fixes for open source components developed by third parties.
 - d. Open a support case and ask Red Hat to add the latest community release to a RHOAR update, then rewrite the application to use the latest community release.

- **3. The Spring Boot framework supports multiple approaches to implement a REST API endpoint. Some of them are not supported or tested in RHOAR. The development team is divided on which approach to take because each team member has experience with a different one, and none of them has previous experience deploying to OpenShift. Given this scenario, which recommendation would you give to the development team to make the best use of both internal and external resources?**
- Poll the developers to find the approach that most developers consider the best one and recommend that approach, despite its support status in RHOAR.
 - Research the CVE history of each approach and recommend the one with the lowest number of issues, regardless of its support status in RHOAR.
 - Research which approach is more popular among the Spring framework community, because it should be easier to find help with this one.
 - Research which approach is supported under RHOAR and ask the development team to use it.
- **4. Which of the following four are supported deployment targets for an application built using RHOAR? (Choose four.)**
- A container engine under a Red Hat Enterprise Linux Atomic Host server.
 - An OpenShift cluster running under a VM created using the Red Hat Container Development Kit (CDK).
 - An OpenShift Online account.
 - A Red Hat OpenShift Container Platform cluster running on bare metal Red Hat Enterprise Linux servers.
 - An OpenShift Origin cluster running in bare metal Red Hat Enterprise Linux servers.
 - A Red Hat OpenShift Container Platform cluster running in Red Hat Enterprise Linux servers from a public cloud provider.
 - A Kubernetes cluster from a public cloud provider container service.
- **5. The release string for Spring Boot version tested with the first release of Red Hat OpenShift Application Runtimes is 1.5.12.RELEASE. What does this release string indicate about the level of support status that the Spring Boot framework receives directly from Red Hat?**
- Red Hat will provide emergency hot fixes for core Spring Boot components whenever a security issue is found.
 - Red Hat is not able to deliver hot fixes for Spring Boot core components, but will collaborate with the Spring Boot community and its vendor to develop and deliver fixes for core Spring Boot components whenever a bug or a security issue is found.
 - Red Hat will not work with the Spring Boot community to develop or deliver fixes for core Spring Boot components. Customers use Spring Boot at their own risk.

Describing Microservice Architecture Patterns

Objective

After completing this section, students should be able to describe the major patterns implemented in microservice architectures.

Introducing Microservice Architectures

The defining characteristic of a *Microservice Architecture* (MSA) environment is that modular services are deployed individually and each can be replaced independent of other services or other instances of the same service. It is not about size: each microservice should implement a well-defined, highly cohesive business context, following Domain-Driven Design techniques.

Each service is developed and deployed as an independent unit, by an autonomous team. Each team could follow a different release strategy and use different frameworks and runtimes.

Another defining characteristic of an MSA is that the inherent complexities of distributed systems are considered from the start instead of as an afterthought. Each service team owns nonfunctional concerns, such as network reliability, tracing, and monitoring. Unlike the old Services-Oriented Architecture (SOA) world, an MSA team does not assume that a central piece of infrastructure handles these nonfunctional components.

Microservice Architectures are enabled by containerization, cloud platforms, Continuous Integration/Continuous Deployment (CI/CD), and lightweight application runtimes, such as WildFly Swarm, Vert.x, and Node.js.

A number of patterns have surfaced as recommended approaches for implementing an MSA, among them:

- Inter-process communication
- Service discovery
- Fault tolerance: bulkhead and circuit breaker
- API gateway
- Distributed tracing
- Aggregated logging
- Security

Another popular description of an MSA is the *Twelve-Factor* methodology. See the references at the end of this section for more information about Microservice Architectures.

Inter-Process Communication

Each microservice runs as a distinct process, usually as a distinct container. Microservices communicate using a standard Inter-Process Communication (IPC) mechanism, which is usually one of:

- REST over HTTP: a popular but synchronous request/response-based

- AMQP, MQTT: message-driven, asynchronous protocols that can implement publish/subscribe, fire-and-forget, and request/reply semantics

A microservice IPC mechanism should be independent of the runtime, so microservices implemented using different frameworks can communicate transparently. This usually means using JSON encoding to format messages.

Service Discovery

In a traditional distributed system deployment, services run at fixed, well-known locations (hosts and ports). But in a cloud-based environment, host names and IP address changes dynamically.

A related issue is that scalability usually requires running multiple concurrent instances of the same microservice. Client applications need to load-balance requests to these instances, and also need to detect when an instance dies, or a new instance is up.

A number of libraries, such as Netflix OSS Eureka and Ribbon, implement the service discovery pattern, but they usually work only with other services that use the same library. Each of these libraries usually requires a special service registry to keep track of available instances of each microservice.

OpenShift provides implementations of the service discovery pattern as the service and the *route* resources, which work for any framework and runtime. Microservices running inside the same OpenShift cluster can communicate using services that provide load-balancing, dynamic discovery of new instances, and a stable IP address based on a well-known DNS host name internal to the cluster.

Microservices running outside of the OpenShift cluster can communicate using routes, which provides the same benefits of services and are tied to a public DNS host name.

Fault Tolerance

Among the patterns related to fault tolerance are the *circuit breaker* and the *bulkhead* patterns. Some MSA libraries, such as the Netflix OSS Hystrix library, implement both patterns.

Circuit Breaker

A circuit breaker avoids cascading failure when a client invokes a dependent service that is overloaded or unresponsive. A circuit breaker wraps a service call and monitors all invocations to detect failures and timeouts.

When the circuit breaker detects failures, it opens the circuit, so no invocations are made to the service, until it is found to be healthy again. In the mean time, the circuit breaker can serve a fallback response, such as a previously cached value, so that the client is not stopped because of an unhealthy service.

Even when there is no fallback response, making network invocations to a service that is unhealthy increases load on the network, which impacts unrelated services and may force the service to take longer to recover.

Bulkhead

A bulkhead isolates service dependencies from each other and limits concurrent accesses to each of them. A bulkhead wraps a service call in a semaphore or thread pool, and when the semaphore is acquired, or the pool is full, service calls are queued.

Bulkhead can also serve a fallback response instead of queuing the call, or serve a fallback response when the queue length is above a threshold.

API Gateway

An API gateway, in an MSA environment, is not a mechanism of policy enforcement or message transformation, but a distributed mechanism of providing a custom view of one or more HTTP APIs. One API gateway is customized for a particular set of services and clients, and different applications usually employ different API gateways.

Example scenarios for using an API gateway are:

- Aggregate data from many services to present a unified view for a web browser-based application.
- Bridge different message transport protocols, such as HTTP and AMQP.
- Expose an older version of a service API, but delegate requests to a newer version of the same service.
- Authenticate clients using different security mechanisms.

There are multiple ways to implement an API gateway, for example: custom-coded services, which are no different than any other microservice, and specialized integration toolkits, such as Fuse Integration Services and Apache Camel.

Distributed Tracing

An end-user or client application request to an MSA environment can span multiple services. Debugging and profiling this request cannot be done using traditional techniques. You cannot isolate a single process to observe and troubleshoot it. Monitoring individual services gives no hint about which originating request provoked which invocation.

The distributed tracing pattern assigns a unique ID to each request. This ID is included in all dependent service invocations, and these services also include the same ID when they make further service calls. This way, it is possible to trace a call graph from originating requests to all dependent requests.

Each dependent service also adds a span ID, which should be somewhat related to the span ID from the previous service request. This way, multiple service invocations from the same originating request can be ordered in time and space.

Note that the request ID is the same for all services in the call graph. The span ID is different for each service in the call graph.

Applications log the request and span IDs, and may also provide additional data, such as start and stop timestamps, and relevant business data. These logs are collected or sent to a central aggregator for storage and visualization.

One popular standard for distributed tracing is the OpenTracing API, and a popular implementation of this standard is the Jaeger project.

Aggregated Logging

Most Java developers are used to generating logs using a standard API. Traditional applications rely on local storage to save these logs, but containerized applications are ephemeral, so when the container stops, all logs are lost.

Many organizations keep centralized log storage facilities for traditional applications. These centralized logs usually record only security and audit trails, and application logs are left on the local server where the application runs.

With cloud and container platforms, aggregating application logs has become an imperative. Without aggregated application logs, there is no information for troubleshooting applications.

Containerized applications are expected to send all log events to standard output and error streams, where they are collected by the container engine. OpenShift provides a logging subsystem that collects logs from the container engines in all cluster nodes and stores them in a central repository. The OpenShift logging subsystem supports not only retrieval but also custom queries on the logs.

To make better use of the OpenShift log query capabilities, an application is expected to generate structured logs, usually as JSON-formatted messages, instead of plain text lines. Most popular Java logging frameworks support custom formatting of structured logs.

Security

Maintaining identity and verifying access control role is a challenge in distributed environments because there is no user session concept to rely upon.

Distributed identity and access control is solved using a number of approaches, for example:

- Single Sign-On (SSO) systems
- Distributed sessions
- Client-side tokens

All of these approaches require a central authentication and authorization server to store and validate user credentials and access roles. Each service needs to invoke that central server for every request, which may become a single point of failure and also a performance bottleneck.

Client-side token solutions are designed to reduce these concerns by generating cryptographically signed tokens, based on public/private key pairs. These tokens can be forwarded to subsequent service invocations and validated by each dependent service without accessing the central authentication server. Each token has a time to live to avoid the need of a session construct and of sending log off requests to a central server. Access rules are also embedded into the client-side token, so each service can make authorization decisions on its own.

The OpenID Connect and JSON Web Token (JWT) standards are popular implementations of client-side tokens. The OpenShift Master API and the Red Hat Single Sign-on (RHSSO) product, based on the Keycloak open source project, both support OpenID Connect and JWT.

It is also common to implement an API gateway to handle client-side tokens on behalf of a set of microservices. Without this, each service must handle conditions such as the need to refresh expired tokens. The API gateway also offloads each service from making authorization decisions.



References

Microservices - a definition of this new architectural term

<https://martinfowler.com/articles/microservices.html>

The Twelve-Factor App web site

<https://12factor.net/>

Microservices Primer: A Short Overview

<https://leanpub.com/microservices-primer/read>

The wikipedia entry about Domain-Driven Design

https://en.wikipedia.org/wiki/Domain-driven_design

Further information is available in the Reference Architecture *Microservice*

Architecture - Building Microservices with JBoss EAP 7 at

https://access.redhat.com/documentation/en-us/reference_architectures/2017/html/microservice_architecture/

► Quiz

Describing Microservice Architecture Patterns

Choose the correct answers to the following questions:

- ▶ 1. Which three of the following are infrastructure enablers of a Microservice Architecture? (Choose three.)
 - a. Containers
 - b. Virtual machines
 - c. Cloud platforms
 - d. Java EE application servers
 - e. Asynchronous and reactive frameworks
 - f. Lightweight application runtimes such as WildFly Swarm

- ▶ 2. Which two of the following are defining characteristics of a Microservice Architecture? (Choose two.)
 - a. Usage of the Model-View-Controller design pattern
 - b. Individual deployment of modular services
 - c. Service implementation with a low number of lines of code or low number of classes
 - d. Nonfunctional concerns, such as network reliability, tracing, and monitoring, are handled early
 - e. Usage of patterns such as distributed tracing, aggregated logging, and convention-over-configuration

- ▶ 3. Which two of the following are common Inter-Process Communication mechanisms employed by Microservice Architectures? (Choose two.)
 - a. AMQP
 - b. HTTP
 - c. IIOP
 - d. SOAP
 - e. Unix pipes

- ▶ 4. Which of the following is a microservices pattern concerned with fault tolerance?
 - a. Service discovery
 - b. Distributed tracing
 - c. Circuit breaker
 - d. API gateway

- **5. Which feature of an API token is adequate for a Microservice Architecture, because it makes each service capable of validating the token without invoking a central authentication server?**
- a. Usage of HTTP headers to transport the token
 - b. Usage of a cryptographic hash that prevents reuse of the token
 - c. Validation is delegated to an API gateway
 - d. Usage of a cryptographic signature generated using a private key
- **6. Which of the following microservices pattern is implemented by the OpenShift infrastructure?**
- a. Service discovery
 - b. Distributed tracing
 - c. Circuit breaker
 - d. API gateway

► Solution

Describing Microservice Architecture Patterns

Choose the correct answers to the following questions:

- ▶ 1. Which three of the following are infrastructure enablers of a Microservice Architecture? (Choose three.)
 - a. Containers
 - b. Virtual machines
 - c. Cloud platforms
 - d. Java EE application servers
 - e. Asynchronous and reactive frameworks
 - f. Lightweight application runtimes such as WildFly Swarm

- ▶ 2. Which two of the following are defining characteristics of a Microservice Architecture? (Choose two.)
 - a. Usage of the Model-View-Controller design pattern
 - b. Individual deployment of modular services
 - c. Service implementation with a low number of lines of code or low number of classes
 - d. Nonfunctional concerns, such as network reliability, tracing, and monitoring, are handled early
 - e. Usage of patterns such as distributed tracing, aggregated logging, and convention-over-configuration

- ▶ 3. Which two of the following are common Inter-Process Communication mechanisms employed by Microservice Architectures? (Choose two.)
 - a. AMQP
 - b. HTTP
 - c. IIOP
 - d. SOAP
 - e. Unix pipes

- ▶ 4. Which of the following is a microservices pattern concerned with fault tolerance?
 - a. Service discovery
 - b. Distributed tracing
 - c. Circuit breaker
 - d. API gateway

- **5. Which feature of an API token is adequate for a Microservice Architecture, because it makes each service capable of validating the token without invoking a central authentication server?**
- a. Usage of HTTP headers to transport the token
 - b. Usage of a cryptographic hash that prevents reuse of the token
 - c. Validation is delegated to an API gateway
 - d. Usage of a cryptographic signature generated using a private key
- **6. Which of the following microservices pattern is implemented by the OpenShift infrastructure?**
- a. Service discovery
 - b. Distributed tracing
 - c. Circuit breaker
 - d. API gateway

Deploying Microservices with the Fabric8 Maven Plug-in

Objective

After completing this section, students should be able to deploy a microservice to an OpenShift cluster using the Fabric8 Maven Plug-in.

The Fabric8 Maven Plug-in

The Fabric8 Maven Plug-in is used to build and deploy Java applications on Kubernetes and OpenShift.

The plug-in focuses on three main tasks:

- Building container images
- Creating OpenShift and Kubernetes resources
- Deploying applications on Kubernetes and OpenShift

Enabling the Fabric8 Maven Plug-in

You can enable the Fabric8 Plug-in in the **plugins** sections of your Project Object Model (POM) file for your project. Assuming you want to enable version **3.5.38** of the plug-in in your project, add the following to the **pom.xml** file:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.38</version>
</plugin>
```

You can also enable the plug-in by running the **setup** goal of the plug-in with the **mvn** command:

```
mvn io.fabric8:fabric8-maven-plugin:3.5.38:setup
```

Run the command from the root of your project folder where your **pom.xml** file is located. The **setup** goal automatically adds the requisite XML configuration to the **plugins** section of the **pom.xml** file.

Configuring the Fabric8 Maven Plug-in

The plug-in uses a convention-over-configuration approach by default. This makes it easy to get up and running quickly.

The default plug-in settings may not be suitable for applications based on the RHOAR runtimes. You need to customize the plug-in configuration and adapt it to suit your choice of RHOAR runtime. The plug-in provides *Generators*, which helps generate container images for applications targeting RHOAR runtimes, such as Spring Boot, WildFly Swarm, and Vert.x.

Complementing the concept of generators are *Enrichers*, used to customize the OpenShift resources generated by the plug-in.

For example, if you are deploying a microservice based on WildFly Swarm, which is typically packaged as a Web Archive (WAR) file, your **pom.xml** file for the project can be customized in this way:

```

<properties>
...
<artifactId>myservice</artifactId>
<packaging>war</packaging>
...
<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
<fabric8.generator.fromMode>istag</fabric8.generator.fromMode> ①
<fabric8.generator.from>redhat-openjdk18-openshift</fabric8.generator.from> ②
...
</properties>
...
<plugins>
<plugin>
<groupId>io.fabric8</groupId>
<artifactId>fabric8-maven-plugin</artifactId>
<version>${version.fabric8.plugin}</version>
<configuration>
<generator>
<includes>
<include>wildfly-swarm</include> ③
</includes>
<excludes>
<exclude>webapp</exclude> ④
</excludes>
</generator>
</configuration>
<executions>
<execution>
<id>fmp</id>
<goals> ⑤
<goal>resource</goal>
<goal>build</goal>
</goals>
</execution>
</executions>
</plugin>

```

- ① Declares that the plug-in should use OpenShift image stream tags to identify the S2I builder image for building the container image.
- ② The image stream tag identifying the S2I builder image that will be used to build the container image. In this scenario, you are using the official OpenJDK 1.8 image from the Red Hat Container Catalog.
- ③ Generator configuration making the plug-in use the **wildfly-swarm** generator instead of the default. This is needed because the fabric8 plug-in categorizes the project as a regular web application due to the packaging type being set to WAR, and tries to build the container image using the **webapp** generator.
- ④ Exclude the **webapp** generator and force the build to use the **wildfly-swarm** generator.

- 5 Bind the **fabric8:resource** and **fabric8:build** goals to the standard life-cycle phases of Maven. Binding these goals makes it easy for the developer to run a single goal, **fabric8:deploy**, which builds the application, generates OpenShift resources, builds the container image, and deploys the application to an OpenShift cluster in one go.

Similarly, other generators are available for Spring Boot and Vert.x runtimes. Refer to the fabric8 Maven plug-in documentation for more details.

Deploying Applications to an OpenShift Cluster

The fabric8 Maven plug-in provides a number of goals to build and deploy applications on an OpenShift cluster. Before running some of these goals, you must have logged in to an OpenShift cluster. The plug-in creates resources under the currently selected OpenShift project.

fabric8:build

This goal builds container images of the application. The plug-in auto-detects if the target platform is OpenShift, and builds images using the S2I approach.

For OpenShift, the plug-in builds container images using the *binary source* build method. In this method, the application binary is built outside the OpenShift cluster, and then the container image is created and streamed into the OpenShift cluster.

fabric8:resource

This goal generates Kubernetes and OpenShift resource descriptors. You can run this goal and examine the generated resource descriptor files, which are in YAML format. A developer can provide *resource fragments*, which are YAML template files that can be enriched by the plug-in (using *Enrichers*). These resource fragments must be provided in the **src/main/fabric8** directory of your project.

fabric8:apply

This goal applies the generated resource files that are generated by the **fabric8:resource** goal to the OpenShift cluster. You can also run the **fabric8:resource-apply** goal which combines both resource generation and applying it to the cluster as a single atomic action.

fabric8:deploy

The **fabric8:deploy** goal is a shortcut that runs the **fabric8:resource**, **fabric8:build**, and **fabric8:apply** goals sequentially. This results in your application being rebuilt, a new container image being created, and the resources being applied to the OpenShift cluster with a single command.

Refer to the Fabric8 Maven Plug-in documentation for a detailed list of all the goals provided by the plug-in.

Demonstration: Deploying to OpenShift with the Fabric8 Maven Plug-in

1. Create a new OpenShift project called **worlD** to deploy the microservice.
2. Review the Maven configuration for the microservice.
3. Deploy the application to the OpenShift cluster. Run the **fabric8:deploy** goal to deploy the microservice to the OpenShift cluster.
4. Use the OpenShift web console to verify that the microservice is deployed on the OpenShift cluster.
5. Test the REST API exposed by the microservice.

6. Clean up. Delete the project to conclude the demonstration.



References

fabric8 Maven plug-in goals

<https://maven.fabric8.io/#goals>

fabric8 Maven plug-in generators

<https://maven.fabric8.io/#generators>

► Guided Exercise

Deploying Microservices with the Fabric8 Maven Plug-in

In this exercise, you will deploy a simple microservice to an OpenShift cluster using the Fabric8 Maven Plug-in.

Outcomes

You should be able to:

- Generate OpenShift resource files for a microservice using the **fabric8:resource** goal
- Build a container image for a microservice using the **fabric8:build** goal
- Deploy a microservice to an OpenShift cluster using the **fabric8:apply** goal

To perform this exercise, you need access to:

- A running OpenShift cluster
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-hola-deploy** branch in the classroom Git repository, containing the source code for the microservice (**hello-microservices**)
- The **org.wildfly.swarm:bom**, and **io.fabric8:fabric8-maven-plugin** Maven artifacts in the classroom Nexus proxy server

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab hola-deploy setup
```

- 1. Create a new project called **hello** in OpenShift to deploy the microservice.

- 1.1. Log in to OpenShift as the **developer** user:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
```

**Note**

If you are logging in to OpenShift cluster for the first time, you will see a warning message about insecure certificates. Accept the insecure connection by typing **y** at the prompt, and press **Enter** to log in.

- 1.2. Create a new project for the application:

```
[student@workstation ~]$ oc new-project hello
```

- 2. Switch to the **do292-hola-deploy** branch in the **hello-microservices** Git repository

- 2.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the student user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 2.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.

- 2.3. Switch to the **do292-hola-deploy** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout do292-hola-deploy
...
Switched to a new branch 'do292-hola-deploy'
```

- 3. Review the Maven configuration for the microservice.

- 3.1. The Hola microservice is written in Java and uses the WildFly Swarm runtime. The parent Maven Project Object Model (POM) file, which is common to all the microservices in the project, is located at the Git repository root folder.

Open the **~/hello-microservices/pom.xml** file in a text editor and briefly review it.

The parent POM file consists of common properties used by different microservices in this project.

```
<properties>
...
<version.wildfly.swarm>7.1.0.redhat-77</version.wildfly.swarm>
...
<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
<fabric8.generator.fromMode>istag</fabric8.generator.fromMode>
<fabric8.generator.from>redhat-openjdk18-openshift</fabric8.generator.from>
```

```
...
</properties>
...
```

- 3.2. The Hola project POM is located at the root folder of the Hola Maven project.

Open the `~/hello-microservices/hola/pom.xml` file in a text editor and briefly review it.

The packaging format for the Hola microservice is declared as **war**, and the microservice inherits all the Maven configuration from the parent POM file.

```
...
<parent>
  <groupId>com.redhat.training.msa</groupId>
  <artifactId>msa-parent</artifactId>
  <version>1.0</version>
</parent>

<artifactId>hola</artifactId>
<packaging>war</packaging>
<name>Red Hat Training MSA: hola</name>
<description>Spanish Hello microservice</description>
...
```

- 3.3. The project POM declares the Bill of Materials (BOM) for WildFly Swarm.

```
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>bom</artifactId>
      <version>${version.wildfly.swarm}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  ...

```

- 3.4. Observe the fabric8 Maven plug-in configuration in the **plugins** section.

The **<configuration>** section within the **<plugin>** definition configures the plug-in. The Fabric8 Maven Plug-in is configured to use the **wildfly-swarm** generator.

```
...
<plugins>
  <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>${version.fabric8.plugin}</version>
    <configuration>
      <generator>
        <includes>
          <include>wildfly-swarm</include>
        </includes>
      </generator>
    </configuration>
  </plugin>

```

```

<excludes>
  <exclude>webapp</exclude>
</excludes>
</generator>
</configuration>
<executions>
  <execution>
    <id>fmp</id>
    <goals>
      <goal>resource</goal>
      <goal>build</goal>
    </goals>
  </execution>
</executions>
</plugin>
...

```

▶ 4. Build and package the application.

- 4.1. Build a Fat JAR for the microservice containing the application and all its dependent JAR files.

Run the **package** goal of Maven to build the Fat JAR for the microservice. Make sure you run the **package** goal from the **hola** project folder and not from the Git repository folder:

```

[student@workstation hello-microservice]$ cd hola
[student@workstation hola]$ mvn clean package
...
[INFO] Building Red Hat Training MSA: hola 1.0
...
[INFO] Repackaging .war: /home/student/hello-microservices/hola/target/hola.war
...
[INFO] BUILD SUCCESS
...

```

It may take a while to download all the Maven dependencies from the classroom Nexus server.

- 4.2. Verify that a WAR file and a fat JAR for the microservice is created in the **target** directory:

```

[student@workstation hola]$ ls -lah target/*.jar
-rw-rw-r--. 1 student student 57M Mar  7 08:05 target/hola-swarm.jar
-rw-rw-r--. 1 student student 61K Mar  7 08:05 target/hola.war

```

▶ 5. Generate the OpenShift resources required to deploy the microservice. Review the resource files generated by the Fabric8 Maven Plug-in.

- 5.1. Run the **fabric8:resource** goal to generate the OpenShift resource files:

```

[student@workstation hola]$ mvn fabric8:resource
...
[INFO] --- fabric8-maven-plugin:3.5.38:resource (default-cli) @ hola ---

```

```
[INFO] F8: Running in OpenShift mode
...
[INFO] F8: validating /home/student/hello-microservices/hola/target/classes/META-INF/fabric8/openshift/hola-svc.yml resource
[INFO] F8: validating /home/student/hello-microservices/hola/target/classes/META-INF/fabric8/openshift/hola-deploymentconfig.yml resource
[INFO] F8: validating /home/student/hello-microservices/hola/target/classes/META-INF/fabric8/openshift/hola-route.yml resource
...
[INFO] BUILD SUCCESS
...
```

- 5.2. Briefly review the generated resource YAML files for the service, deployment configuration, and the route in a text editor.

```
[student@workstation hola]$ cat \
    target/classes/META-INF/fabric8/openshift/hola-svc.yml
[student@workstation hola]$ cat \
    target/classes/META-INF/fabric8/openshift/hola-deploymentconfig.yml
[student@workstation hola]$ cat \
    target/classes/META-INF/fabric8/openshift/hola-route.yml
```

► 6. Build the application container image.

- 6.1. Use the **fabric8:build** goal to build the container image using the binary S2I build method.

```
[student@workstation hola]$ mvn fabric8:build
...
[INFO] Building Red Hat Training MSA: hola 1.0
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (default-cli) @ hola ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator wildfly-swarm
[INFO] F8: wildfly-swarm: Using ImageStreamTag 'redhat-openjdk18-openshift:latest'
as builder image
...
[INFO] F8: Creating ImageStream hola
...
[INFO] F8: Starting S2I Java Build .....
...
[INFO] F8: Pushed 6/6 layers, 100% complete
[INFO] F8: Push successful
...
[INFO] BUILD SUCCESS
...
```

 **Note**

You may see the following messages in the output. These can be safely ignored.

```
[INFO] Current reconnect backoff is 2000 milliseconds (T1)
...
[INFO] Current reconnect backoff is 4000 milliseconds (T2)
...
[INFO] Current reconnect backoff is 8000 milliseconds (T3)
...
[INFO] Current reconnect backoff is 16000 milliseconds (T4)
...
Exception in reconnect
java.util.concurrent.RejectedExecutionException: ...
    at java.util.concurrent.ThreadPoolExecutor
$AbortPolicy.rejectedExecution(ThreadPoolE
xecutor.java...)
```

6.2. Verify that build and image stream resources for the microservice are created:

```
[student@workstation hola]$ oc get builds
NAME      TYPE      FROM      STATUS      STARTED      DURATION
hola-s2i-1  Source   Binary    Complete   About a minute ago  18s

[student@workstation hola]$ oc get is
NAME      DOCKER REPO      TAGS ...
hola      docker-registry.default.svc:5000/hello/hola  1.0 ...
```

► 7. Deploy the application to the OpenShift cluster.

Run the **fabric8:apply** goal to deploy the microservice to the OpenShift cluster:

```
[student@workstation hola]$ mvn fabric8:apply
...
[INFO] --- fabric8-maven-plugin:3.5.38:apply (default-cli) @ hola ---
[INFO] F8: Using OpenShift at https://master.lab.example.com:443/ in namespace
hello...
[INFO] OpenShift platform detected
[INFO] Using project: hello
[INFO] Creating a Service...
...
[INFO] Creating a DeploymentConfig...
...
[INFO] Creating Route...
...
[INFO] BUILD SUCCESS
...
```

► 8. Use the OpenShift web console to verify that the microservice is deployed on the OpenShift cluster.

- 8.1. Log in to the OpenShift web console. From the workstation VM, open a web browser and access the OpenShift web console at <https://master.lab.example.com>. Log in as **developer**, using **redhat** as the password.

**Note**

If you are logging in to the OpenShift web console for the first time, you will see a warning about insecure certificates. Accept the insecure certificate and add an exception for this website.

- 8.2. Click on **hello** in the **My Projects** pane on the right to open the **Overview** page for the project.

The screenshot shows the 'My Projects' section of the OpenShift web console. At the top, it says 'My Projects' and has a blue button '+ Create Project'. Below that, it displays '1 of 1 Projects' and a 'View All' link. A single project card is shown for 'hello', which was created by developer 2 hours ago. To the right of the project name is a three-dot menu icon.

- 8.3. Click the arrow to the left of the **hola** deployment to expand the deployment details, and verify that you can see a donut chart (blue colored circle with the number "1" inside it) showing that a single pod of the microservice is running.

The screenshot shows the 'Deployment' details page for the 'hola' deployment. At the top, it says 'APPLICATION hola' and provides a link 'http://hola-hello.apps.lab.example.com'. The deployment is labeled 'DEPLOYMENT hola, #1'. Under 'CONTAINER: WILDFLY-SWARM', it lists the image as 'hello/hola f7dff7c 278.8 MiB', the build as 'hola-s21, #1', the source as 'Binary', and ports as '8080/TCP (http) and 2 others'. To the right, there is a large blue donut chart with the number '1' in the center, indicating one pod is running. There are also up and down arrows next to the chart.

► 9. Test the hola microservice.

- 9.1. Note the route URL in the **Overview** page near the top right corner of the OpenShift web console. Copy the URL for the next step.
- 9.2. From the terminal window on the workstation VM, use the **curl** command to test the microservice. Ensure that you append the path **/api/hola** to the route URL:

```
[student@workstation hola]$ curl \
  http://hola-hello.apps.lab.example.com/api/hola
Hola de hola-hello.apps.lab.example.com
```

► **10.** Clean up.

- 10.1. If you wish to start over this exercise, delete the **/home/student/hello-microservices** directory, delete all OpenShift resources in the project, and restart from the beginning:

```
[student@workstation ~]$ oc delete all -l app=hola
buildconfig "hola-s2i" deleted
imagestream "hola" deleted
deploymentconfig "hola" deleted
route "hola" deleted
pod "hola-1-m2pzb" deleted
service "hola" deleted
```

Verify that you do not see any pods in the **Overview** page of the OpenShift web console, and that the service, deployment configuration, and route resources for the **hola** project are deleted.

- 10.2. If you want to move to the next exercise, delete the **hello** project in OpenShift.

```
[student@workstation ~]$ oc delete project hello
```

This concludes the guided exercise.

► Lab

Deploying Microservices to an OpenShift Cluster

Performance Checklist

In this lab, you will deploy a microservice to an OpenShift cluster using the fabric8 Maven plug-in.

Outcomes

You should be able to configure the Fabric8 Maven Plug-in for a project, and deploy a simple Java-based microservice to an OpenShift cluster.

The Aloha microservice runs on the WildFly Swarm runtime. It provides a simple HTTP API that responds to requests by printing a message in Hawaiian.

You must deploy the microservice according to the following requirements:

- The microservice should be deployed in an OpenShift project called **hello**.
- The HTTP API for the microservice should be accessible at the URL:
`http://aloha-hello.apps.lab.example.com/api/aloha`.
- The Git repository that contains the source code for the microservice is available at:
`http://services.lab.example.com/hello-microservices`.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

To perform this exercise, you need access to:

- A running OpenShift cluster
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-aloha-deploy-begin** branch in the classroom Git repository containing the source code for the microservice (**hello-microservices**)
- The **org.wildfly.swarm:bom**, and **io.fabric8:fabric8-maven-plugin** Maven artifacts in the classroom Nexus server

Run the following command on the **workstation** VM to validate the prerequisites:

```
[student@workstation ~]$ lab aloha-deploy setup
```

You can compare your source code changes while performing this lab with the solution branch **do292-aloha-deploy-solution** of the **hello-microservices** Git repository.

1. Log in to the OpenShift cluster as the **developer** user, with a password of **redhat**. Create the project in OpenShift.
2. Switch to the **do292-aloha-deploy-begin** branch in the **hello-microservices** Git repository.
3. Configure the version number for the **org.wildfly.swarm:bom** dependency, and the version number of the Fabric8 Maven Plug-in in the parent POM file for the microservice as follows:

Property	Value
version.wildfly.swarm	7.1.0.redhat-77
version.fabric8.plugin	3.5.38

4. Edit the Maven POM file for the Aloha microservice and do the following:
 - Enable the **wildfly-swarm** generator, and disable the default **webapp** generator.
 - Bind the goals of the Fabric8 Maven Plug-in to the standard Maven life-cycle phases so that OpenShift resources are generated and the container images are created whenever the project is built.
5. Deploy the microservice to the OpenShift cluster.
6. Log in to the OpenShift web console and verify that the microservice is deployed on the OpenShift cluster.
7. Test the microservice by using the **curl** command. Verify that you see a greeting in Hawaiian along with the host name where the microservice is running.
8. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation aloha]$ lab aloha-deploy grade
```

9. Clean up:
 - 9.1. Commit your changes to your local Git repository:

```
[student@workstation coolstore]$ git commit -a -m \
"Finished the lab."
```

- 9.2. Delete the project in OpenShift.

```
[student@workstation aloha]$ oc delete project hello
```

This concludes the lab.

► Solution

Deploying Microservices to an OpenShift Cluster

Performance Checklist

In this lab, you will deploy a microservice to an OpenShift cluster using the fabric8 Maven plug-in.

Outcomes

You should be able to configure the Fabric8 Maven Plug-in for a project, and deploy a simple Java-based microservice to an OpenShift cluster.

The Aloha microservice runs on the WildFly Swarm runtime. It provides a simple HTTP API that responds to requests by printing a message in Hawaiian.

You must deploy the microservice according to the following requirements:

- The microservice should be deployed in an OpenShift project called **hello**.
- The HTTP API for the microservice should be accessible at the URL:
`http://aloha-hello.apps.lab.example.com/api/aloha`.
- The Git repository that contains the source code for the microservice is available at:
`http://services.lab.example.com/hello-microservices`.

If you need help using Git and JBoss Developer Studio, refer to ??? and ???.

To perform this exercise, you need access to:

- A running OpenShift cluster
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-aloha-deploy-begin** branch in the classroom Git repository containing the source code for the microservice (**hello-microservices**)
- The **org.wildfly.swarm:bom**, and **io.fabric8:fabric8-maven-plugin** Maven artifacts in the classroom Nexus server

Run the following command on the **workstation** VM to validate the prerequisites:

```
[student@workstation ~]$ lab aloha-deploy setup
```

You can compare your source code changes while performing this lab with the solution branch **do292-aloha-deploy-solution** of the **hello-microservices** Git repository.

1. Log in to the OpenShift cluster as the **developer** user, with a password of **redhat**. Create the project in OpenShift.

- 1.1. Log in to OpenShift as the **developer** user:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
```

- 1.2. Create a new project for the application:

```
[student@workstation ~]$ oc new-project hello
```

2. Switch to the **do292-aloha-deploy-begin** branch in the **hello-microservices** Git repository.

- 2.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the student user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 2.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.

- 2.3. Switch to the **do292-aloha-deploy-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout do292-aloha-deploy-begin
...
Switched to a new branch 'do292-aloha-deploy-begin'
```

3. Configure the version number for the **org.wildfly.swarm:bom** dependency, and the version number of the Fabric8 Maven Plug-in in the parent POM file for the microservice as follows:

Property	Value
version.wildfly.swarm	7.1.0.redhat-77
version.fabric8.plugin	3.5.38

Edit the parent POM file by opening the **~/hello-microservices/pom.xml** file in a text editor. Edit the **<properties>** section, changing the following properties:

```
...
<properties>
...
<version.wildfly.swarm>7.1.0.redhat-77</version.wildfly.swarm>
```

```

...
<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
...
</properties>
...

```

Do not make any other changes to the parent POM file.

4. Edit the Maven POM file for the Aloha microservice and do the following:

- Enable the **wildfly-swarm** generator, and disable the default **webapp** generator.
- Bind the goals of the Fabric8 Maven Plug-in to the standard Maven life-cycle phases so that OpenShift resources are generated and the container images are created whenever the project is built.

Edit the Maven POM file for the aloha microservice by opening the **~/hello-microservices/aloha/pom.xml** file in a text editor.

Configure the fabric8 Maven plug-in like the following:

```

...
<plugins>
...
<plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>${version.fabric8.plugin}</version>
    <configuration>
        <generator>
            <includes>
                <include>wildfly-swarm</include>
            </includes>
            <excludes>
                <exclude>webapp</exclude>
            </excludes>
        </generator>
    </configuration>
    <executions>
        <execution>
            <id>fmp</id>
            <goals>
                <goal>resource</goal>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
</plugin>
...

```

5. Deploy the microservice to the OpenShift cluster.

Run the **fabric8:deploy** goal to deploy the microservice to the OpenShift cluster. Make sure you run the **fabric8:deploy** goal from the **aloha** Maven folder and not from the Git repository project folder:

```
[student@workstation hello-microservice]$ cd aloha
[student@workstation aloha]$ mvn fabric8:deploy
...
[INFO] Building Red Hat Training MSA: aloha 1.0
...
[INFO] >>> fabric8-maven-plugin:3.5.38:deploy (default-cli) > install @ aloha >>>
...
[INFO] --- fabric8-maven-plugin:3.5.38:resource (fmp) @ aloha ---
[INFO] F8: Running in OpenShift mode
[INFO] F8: Using docker image name of namespace: hello
[INFO] F8: Running generator wildfly-swarm
[INFO] F8: wildfly-swarm: Using ImageStreamTag 'redhat-openjdk18-openshift:latest'
as builder image
...
[INFO] --- maven-war-plugin:2.5:war (default-war) @ aloha ---
...
[INFO] Building war: /home/student/hello-microservices/aloha/target/aloha.war
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ aloha ---
...
[INFO] Repackaged .war: /home/student/hello-microservices/aloha/target/aloha.war
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ aloha ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator wildfly-swarm
[INFO] F8: wildfly-swarm: Using ImageStreamTag 'redhat-openjdk18-openshift:latest'
as builder image
...
[INFO] F8: Starting S2I Java Build .....
...
[INFO] F8: Pushing image docker-registry.default.svc:5000/hello/aloha:1.0 ...
[INFO] F8: Pushed 1/6 layers, 17% complete
...
[INFO] F8: Pushed 6/6 layers, 100% complete
[INFO] F8: Push successful
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ aloha ---
[INFO] F8: Using OpenShift at https://master.lab.example.com:443/ in namespace
hello...
[INFO] OpenShift platform detected
```

```
[INFO] Using project: hello
...
[INFO] BUILD SUCCESS
...
```

**Note**

You may see the following messages in the output. These can be safely ignored.

```
...
[INFO] Current reconnect backoff is 8000 milliseconds (T3)
...
[INFO] Current reconnect backoff is 16000 milliseconds (T4)
...
[ERROR] Exception in reconnect
java.util.concurrent.RejectedExecutionException: ...
at java.util.concurrent.ThreadPoolExecutor
$AbortPolicy.rejectedExecution(ThreadPoolE
xecutor.java...)
...
```

6. Log in to the OpenShift web console and verify that the microservice is deployed on the OpenShift cluster.
 - 6.1. Log in to the OpenShift web console. From the workstation VM, open a web browser and access the OpenShift web console at <https://master.lab.example.com>. Log in as **developer**, using **redhat** as the password.
 - 6.2. Click on **hello** in the **My Projects** pane on the right to open the **Overview** page for the project.
 - 6.3. Click the arrow to the left of the **aloha** deployment to expand the deployment details, and verify that you can see a donut chart showing that a single pod of the microservice is running.
7. Test the microservice by using the **curl** command. Verify that you see a greeting in Hawaiian along with the host name where the microservice is running.
From the terminal window on the workstation VM, use the **curl** command to test the microservice:

```
[student@workstation aloha]$ curl \
  http://aloha-hello.apps.lab.example.com/api/aloha
Aloha mai aloha-hello.apps.lab.example.com
```

8. Grade your work.
Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation aloha]$ lab aloha-deploy grade
```
9. Clean up:
 - 9.1. Commit your changes to your local Git repository:

```
[student@workstation coolstore]$ git commit -a -m \  
"Finished the lab."
```

9.2. Delete the project in OpenShift.

```
[student@workstation aloha]$ oc delete project hello
```

This concludes the lab.

Summary

In this chapter, you learned:

- Microservice Architectures (MSA) consist of modular services that are deployed individually. They are developed and maintained by independent teams that consider complexities of distributed systems from the start.
- RHOAR requires application container images to be based on a certified parent container image.
- A number of patterns are recommended for implementing an MSA, among them: service discovery, fault tolerance, and API gateway.
- Microservice patterns such as distributed tracing, logging, and security based on client-side tokens enable debugging, profiling, and securing MSA applications.
- The Fabric8 Maven Plug-in (FMP) enables a project POM to build application container images, create OpenShift resources, and manage OpenShift build and deployment processes.
- The FMP performs, by default, a local maven build and feeds the final artifact (a fat JAR) to an OpenShift binary Source-to-Image (S2I) build.

Chapter 2

Deploying Microservices with the WildFly Swarm Runtime

Goal

Develop and deploy a microservice using the WildFly Swarm runtime.

Objectives

- Describe the WildFly Swarm runtime and develop an application in WildFly Swarm.
- Configure a Maven project to build a WildFly Swarm application and deploy on an OpenShift cluster.
- Describe the components and architecture of the Coolstore Microservice application.

Sections

- Developing an Application with the WildFly Swarm Runtime (and Guided Exercise)
- Configuring a Maven Project for WildFly Swarm (and Guided Exercise)
- Describing the Coolstore Microservice Application (and Quiz)

Lab

Deploying Microservices with the WildFly Swarm Runtime

Developing an Application with the WildFly Swarm Runtime

Objectives

After completing this section, students should be able to describe the WildFly Swarm runtime and develop an application using WildFly Swarm.

Describing WildFly Swarm

WildFly Swarm is a framework for developing modern cloud-native applications using Java. It is based on the popular WildFly application server and is capable of producing small, stand-alone, REST-based, back-end microservices that can be independently developed, deployed, managed, scaled and monitored on platforms such as the OpenShift Container Platform.

Unlike standard Java EE application servers, which are monolithic in nature and contain all the APIs and frameworks in a single runtime, WildFly Swarm promotes the concept of "*just enough app server*". You can selectively pick and choose which APIs and frameworks are required for your microservice and run the entire application as a stand-alone executable JAR file.

The WildFly Swarm runtime packaged in the Red Hat OpenShift Application Runtime (RHOAR) subscription enables you to develop and deploy microservices on the OpenShift Container Platform with the following advantages:

- Rolling updates to the deployed microservices with minimal to zero downtime.
- Automatic service discovery of other microservices deployed in the OpenShift cluster.
- Externalized application configuration using OpenShift configuration maps.
- Automatic health checks for deployed microservices using OpenShift readiness and liveness probes.

The MicroProfile Specification and WildFly Swarm

The MicroProfile specification is a joint venture between the Eclipse Foundation and many large vendors including Red Hat to define a baseline platform definition that optimizes Java for a microservices-based architecture. The intention is to provide a framework supporting many of the most common design patterns already in use by Java developers building microservices across the industry.

WildFly Swarm supports numerous components from the MicroProfile specification that you can include in your application. These components can be added to your Maven POM file as dependencies to include support for the MicroProfile components in your microservices application.

WildFly Swarm continues to rapidly adopt APIs from the MicroProfile specification for microservices concerns like security, circuit breakers, distributed tracing, metrics, configuration management and more.

Describing WildFly Swarm Fractions

A *fraction* is a component within the WildFly Swarm framework that provides a certain unique capability that applications can use. Usually fractions provide implementation of standard Java EE

APIs like CDI, JAX-RS, JPA and more. Fractions can also provide other higher level functionality like support for single sign-on (SSO), microservices patterns like circuit breaker, caching, logging, monitoring, and more.

As a developer, you consume fractions using Maven dependencies declared in your project's Maven POM file. The functionality provided by the fraction is then packaged along with the application in the executable JAR file produced by the Maven build process.

Developing Applications with WildFly Swarm

To start developing applications using WildFly Swarm, you need a JDK version 8 or newer, and Apache Maven 3.3 or newer installed.

To develop a simple REST-based microservice that exposes a single HTTP GET endpoint, start with a class containing JAX-RS annotations as follows:

```
package com.redhat.example;

import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

@Path("/") ①
public class HelloResource {

    @GET ②
    @Produces("text/plain") ③
    @Path("/hello") ④
    public Response hello() {
        return Response.ok("Hello World!").build();
    }
}
```

- ① Root path for all endpoints declared in this resource class.
- ② The HTTP request type; in this case, HTTP GET.
- ③ The MIME type of the response from this method.
- ④ The relative path for this method.

JAX-RS based applications need to identify the root resource URI path for all endpoints exposed by a microservice. This is done by using the `@ApplicationPath` annotation. Create a class definition like the following:

```
package com.redhat.example;

import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;

@ApplicationPath("/api") ①
public class RestApplication extends Application {
```

- ① All endpoints exposed by the microservice are relative to the `/api` path

The above two class definitions result in the `hello()` method being invoked for HTTP GET requests to the path `/api/hello`.

Once you have implemented the code for the REST API, you must create a Maven POM file for the microservice listing all dependencies required to build and package it. The detailed Maven POM file configuration is covered in a subsequent section of this chapter.

Packaging the microservice using Maven creates a self-contained, stand-alone executable JAR file called a *fat JAR*, or *uber JAR*. The fat JAR produced by Maven has the string `-swarm` suffixed to it. If for example, your Maven `project.artifactId` is named `hello`, the resulting fat JAR is built as `hello-swarm.jar` in the `target` folder of your project.

You can run the fat JAR to start your microservice:

```
$ java -jar target/hello-swarm.jar
```

Once your microservice is running, you can test the application using the `curl` command:

```
$ curl http://localhost:8080/api/hello
Hello World!
```

Invoking REST APIs from WildFly Swarm

You can invoke REST APIs from WildFly Swarm using the JAX-RS client API. For example, to invoke a REST API with a HTTP GET endpoint available at `http://someservice/api/service1`:

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient(); ①

WebTarget target = client
    .target("http://someservice") ②
    .path("/api")
    .path("/service1"); ③

Response response = target.request(MediaType.TEXT_PLAIN).get(); ④

// Process response
...
```

- ① Instantiate the JAX-RS client instance.
- ② The Base URI for the REST API call.
- ③ The path of the REST resource that the client invokes. In this case, `/api/service1`.
- ④ Make a HTTP GET request and get a plain text response.

Testing WildFly Swarm Applications

You can unit test your WildFly Swarm microservices using the *Arquillian* test framework. Arquillian provides a test harness that you can use to launch an embedded WildFly Swarm container and execute test code both from within the running application.

WildFly Swarm provides an **arquillian** fraction for testing applications. Add the dependency to your project Maven POM file in **test** scope.

A unit test class to test a simple JAX-RS microservice can be created as follows. Note that the import statements have been omitted for brevity:

```

...
@RunWith(Arquillian.class) ❶
public class HelloTest {

    private Client client; ❷

    @Deployment ❸
    public static Archive<?> createDeployment() {

        WebArchive archive = ShrinkWrap
            .create(WebArchive.class, "test.war")
            .addPackages(true,
RestApplication.class.getPackage());
        return archive;
    }

    @CreateSwarm ❹
    public static Swarm newContainer() throws Exception {
        Properties properties = new Properties();
        properties.put("swarm.http.port", 8080);
        return new Swarm(properties);
    }

    @Before ❺
    public void before() throws Exception {
        client = ClientBuilder.newClient();
    }

    @After ❻
    public void after() throws Exception {
        client.close();
    }

    @Test
    @RunAsClient ❼
    public void testHello() throws Exception {
        WebTarget target = client
            .target("http://localhost:8080")
            .path("/api")
            .path("/hello"); ❽
        Response response = target.request(MediaType.TEXT_PLAIN).get();
        String text = response.readEntity(String.class);
        assertThat(text, containsString("Hello World")); ❾
    }
}

```

- ❶ Use the Arquillian test runner instead of the default JUnit runner.

- ② Reference the JAX-RS client class to invoke the REST API under test.
- ③ The method annotated with `@Deployment` creates the WAR file to be deployed in the embedded WildFly Swarm container. You can control which classes and resources should be part of the deployable WAR.
- ④ Bootstrap an embedded WildFly Swarm container for running tests. You can customize the properties of the WildFly Swarm instance (for example, bindings to network interfaces, ports, and more) for your tests.
- ⑤ Before running each test, Instantiate a JAX-RS client instance.
- ⑥ After running each test, destroy the JAX-RS client instance.
- ⑦ Run the test method as a headless client against the microservice being tested.
- ⑧ Invoke the `/api/hello` endpoint.
- ⑨ Assert that the response contains the expected string using the Hamcrest API, which provides a granular and flexible way to define assertions.

Testing REST APIs using the REST Assured Framework

As an alternative to using the JAX-RS client API to invoke REST APIs, you can also use the REST Assured framework to test microservices using a less verbose syntax. For example, to test a simple HTTP GET endpoint `http://localhost:8080/api/hello`:

```
@Test
@RunAsClient
public void testHello() throws Exception {
    given().when().get("/api/hello").then().body(containsString("Hello World"));
}
```

There is no need to explicitly list the full host name of the endpoint in the `get()` method, since REST Assured assumes that the service being tested is running on the localhost at port 8080. For invoking other endpoints outside localhost, provide the full URL to the `get()` method.

Configuring WildFly Swarm Applications

You can configure various aspects of the WildFly Swarm runtime and fractions, as well as manage the configuration of applications deployed on it in numerous ways. For many cases, reasonable defaults are already applied, so you do not have to make any changes unless you explicitly want to.

This section briefly discusses the most common options.

Setting System Properties Using the Command Line

You can set properties by passing command-line arguments to the JVM when executing the fat JAR. This is useful for temporarily changing the configuration for a single run of your application.

For example:

```
$ java -Dswarm.bind.address=127.0.0.1 \
-Djava.net.preferIPv4Stack=true \
-jar myapp-swarm.jar
```

Setting System Properties Using YAML Files

YAML is the preferred method for configuration of your application. This method provides grouping of environment-specific configurations (dev, stage, qa, prod, and more), which you can

then selectively enable when executing the application. These environment-specific groupings are called *stages* or *profiles* in WildFly Swarm parlance.

If the packaged fat JAR contains a file named **project-defaults.yml** inside it, then that file represents the defaults applied over the absolute defaults that WildFly Swarm provides. You can provide your own configuration in a file named **project-name.yml** inside the fat JAR, and pass the **-S** option during startup. You should place the YAML file in the **src/main/resources** folder of your Maven project.

For example, if you create a new profile called **test** and add configuration items in the file named **project-test.yml** in your fat JAR, you should start the application as follows:

```
$ java -jar myapp-swarm.jar -Stest
```

Within the YAML files, you can declare both system properties related to the WildFly Swarm runtime, as well as custom application configuration properties using YAML format as follows:

```
# A project defaults file
swarm:
  bind:
    address: 127.0.0.1
com:
  mycompany:
    myapp:
      config:
        property1: value1
        property2: value2
```

Note how the indentation corresponds to the dotted notation representation of the property. That is, WildFly Swarm parses the YAML data into three properties:

- **-Dswarm.bind.address=127.0.0.1**
- **-Dcom.mycompany.myapp.config.property1=value1**
- **-Dcom.mycompany.myapp.config.property2=value2**

The applications running on WildFly Swarm can directly reference these system properties using JDK APIs, or these properties can be injected into classes using the **@ConfigProperty** annotation of the MicroProfile Config API:

```
@Inject
@ConfigProperty(name = "com.mycompany.myapp.config.property1")
private String prop1;
```

Configuring JDBC Datasources using YAML Configuration

For applications that require access to a database, you can provide the datasource configuration in the YAML configuration file.

For example, to create a datasource named **testDS** that connects to a PostgreSQL database, create an entry in the **swarm.datasources.data-sources** namespace as follows:

```

swarm:
  datasources:
    data-sources:
      testDS:
        driver-name: postgresql
        connection-url: jdbc:postgresql://localhost:5432/testdb
        user-name: test
        password: test
  
```

You can provide any system property that is valid for your JDBC driver, and WildFly Swarm ensures that the datasource is created at startup with the provided properties.

Monitoring WildFly Swarm Applications

WildFly Swarm provides a *monitor* fraction that exposes a number of default REST endpoints providing runtime status of the host where the instance is running. The **monitor** fraction also provides access to the health check API, which allows a developer to provide custom health check methods in Java code, and exposes these as REST end points, which can be used for monitoring.

To add health checks to your WildFly Swarm microservice, add the **monitor** fraction as a dependency in your Maven POM file.

You can register methods to the health check API using the **@Health** attribute. The method should return a **HealthStatus** object which is transformed into JSON by the health check API:

```

@Path("/")
public class HealthCheckResource {

  @GET
  @Health
  @Path("/status")
  public HealthStatus check() {
    return HealthStatus.named("server-state").up();
  }
  
```

You can register your own named attributes to the **HealthStatus** object, and each of these can contain arbitrary key-value pairs of information relevant to clients consuming the health check endpoints.

The Health Check API transforms the **HealthStatus** above to the following JSON payload:

```
{
  "id": "server-state",
  "result": "UP"
}
```

You can declare more than one health check method within an application. The methods annotated with **@Health** are included in the **/health** endpoint response, which represents the combined outcome of all the health checks methods in the application:

```
{
  "checks": [
    {
      "id": "server-state",
      "result": "UP"
    }
  ],
  "outcome": "UP"
}
```

If any one of the health check methods responds with a status of **DOWN**, the overall status is set to **DOWN** and the **/health** endpoint responds with a HTTP status code of 503 (service not available).

Newer release of the WildFly Swarm runtime also support the MicroProfile Health Check API as an alternative to the Monitor fraction.



References

Further information about supported and tested WildFly Swarm fractions and libraries is available in the *Red Hat OpenShift Application Runtimes 1.0 Release Notes* at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/red_hat_openshift_application_runtimes_release_notes/

Further information about configuring a Maven POM for WildFly Swarm is available at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/wildfly_swarm_runtime_guide/#building_your_application

WildFly Swarm open source project

<http://wildfly-swarm.io/>

Eclipse MicroProfile

<https://micropattern.io/>

Building RESTful Web Services with JAX-RS

<https://javaee.github.io/tutorial/jaxrs.html>

Accessing REST Resources with the JAX-RS Client API

<https://javaee.github.io/tutorial/jaxrs-client.html>

REST Assured framework

<http://rest-assured.io/>

Hamcrest framework

<http://hamcrest.org/JavaHamcrest/>

► Guided Exercise

Developing an Application with the WildFly Swarm Runtime

In this exercise, you will develop a simple microservice using the WildFly Swarm runtime.

Outcomes

You should be able to:

- Implement an HTTP API endpoint using JAX-RS.
- Configure the Maven WildFly Swarm plug-in for the **hola** microservice.
- Package and run the **hola** microservice as a fat JAR.

To perform this exercise, you need access to:

- The **do292-hola-local-*** branches in the classroom Git repository, containing the source code for the **hola** microservice as part of the **hello-microservice/hola** Maven project.
- The **org.wildfly.swarm:bom**, **io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts on the classroom Nexus proxy server.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab hola-local setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-hola-local-solution** of the **hello-microservices** Git repository.

- 1. Switch to the **do292-hola-local-begin** branch in the **hello-microservices** Git repository and import the **hola** project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-hola-local-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
  do292-hola-local-begin
...
Switched to a new branch 'do292-hola-local-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/hello-microservices/hola** folder as a Maven project.
- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-hola-local-begin**.
- ▶ 2. Inspect the Maven POM file for the microservice.
Open the **pom.xml** file from the **hola** project. Switch to the **pom.xml** tab.
Note the reference to the parent POM file, which provides some system properties. These properties are used to define the version strings for Maven artifacts required to build the microservice.
Note the reference to the Bill of Materials (BOM) dependencies for the WildFly Swarm distribution and the MicroProfile API.
Note the use of the artifacts in the **org.wildfly.swarm** group as dependencies for building the microservice.

- ▶ 3. Inspect the REST API implementation for the microservice.

- 3.1. Inspect the JAX-RS application class.
Open the **JaxRsActivator** class from the **com.redhat.training.msa.hola.rest** package.
Note that this class is annotated with the **@ApplicationPath**. This class is complete. Do not make any changes to it.
- 3.2. Inspect the class that defines the microservice's HTTP API.
Open the **HolaResource** class from the **com.redhat.training.msa.hola.rest** package.
Note that the class is annotated with the **@Path** annotation, and the CDI scope is set to **@ApplicationScoped**.
Note that two String properties (**a** and **b**) are injected into this class using the **@ConfigProperty** annotation. Note that the property **a** has an empty string as the default value, whereas the property **b** has a default value of **Dos**, in case the property value is not set explicitly.
Inspect the incomplete **hola()** method. It prints a welcome message in Spanish along with the host name where the microservice is currently running, and also prints the value of the two String properties.
- 3.3. Inspect the WildFly Swarm application configuration file.
Open the **project-defaults.yml** file in the **/src/main/resources** folder.

Note the values of the properties **a** and **b** in the file.

► **4.** Run the unit tests for the REST API.

4.1. Inspect the unit test class for the REST API.

Open the **HolaResourceTest** class from the **com.redhat.training.msa.hola.rest** package.

Note that the test class uses the **Arquillian** JUnit runner class.

Review the **newContainer()** method, which bootstraps a WildFly Swarm container as part of each test.

Review the **createWebArchive()** method, which creates a WAR file containing the microservice along with other test artifacts and deploys the WAR file to the bootstrapped WildFly Swarm container before running the tests.

Review the code in the **testHola()** method. It uses the REST assured testing library to call the REST API exposed by the microservice and verifies that the response contains the expected response in Spanish.

4.2. Run the **HolaResourceTest** class as a JUnit test.

Ignore arquillian related errors in the JBDS **Console** view. If you are running the test for the first time, it may take some time to download all the dependencies from the classroom nexus proxy and run the test.

The **JUnit** view shows that the test run fails with an assertion error stating that the response does not contain the expected string.

► **5.** Complete the REST API implementation for the microservice to make the unit tests pass.

5.1. Complete the REST API implementation.

Open the **HolaResource** class from the **com.redhat.training.msa.hola.rest** package.

Add the following annotations to the **hola()** method so that:

- This method is invoked for HTTP GET requests to the **/api/hola** path.
- The method produces a plain text response for REST clients.

```
...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
...
@GET
@Path("/hola")
@Produces("text/plain")
public String hola() {
    ...
}
```

Save your changes to the **HolaResource.java** file.

5.2. Run the **HolaResourceTest** class as a JUnit test.

The **JUnit** view shows that the test now passes.

► **6.** Package the microservice as a Fat JAR and test the REST API.

- Configure the WildFly Swarm Maven plug-in in the project POM file.

Open the **pom.xml** file from the **hola** project. Switch to the **pom.xml** tab.

Uncomment the WildFly Swarm Maven plug-in in the **<plugins>** section of the POM file as follows:

```
<plugin>
<groupId>org.wildfly.swarm</groupId>
<artifactId>wildfly-swarm-plugin</artifactId>
<version>${version.wildfly.swarm}</version>
<configuration>
  <useUberJar>true</useUberJar>
</configuration>
<executions>
  <execution>
    <goals>
      <goal>package</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

- Package the microservice as a Fat JAR. Switch to a terminal window and run the following commands:

```
[student@workstation hello-microservices]$ cd hola
[student@workstation hola]$ mvn clean package -DskipTests
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ hola ---
...
[INFO] BUILD SUCCESS
...
```

- Run the Fat JAR and test the microservice.

```
[student@workstation hola]$ java -jar target/hola-swarm.jar
...
2018-05-26 10:49:12,018 INFO [org.jboss.as] (MSC service thread 1-8) WFLYSRV0049: WildFly Swarm 7.1.0.redhat-77 (WildFly Core 3.0.12.Final-redhat-1) starting
...
2018-05-26 10:49:18,250 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Swarm 7.1.0.redhat-77 (WildFly Core 3.0.12.Final-redhat-1) started in 6377ms
...
2018-05-26 10:50:40,182 INFO [org.wildfly.swarm] (main) WFSWARM99999: WildFly Swarm is Ready
```

- Test the microservice's HTTP API

Open another terminal window. Use the **curl** command to invoke the HTTP API. Use **localhost:8080** as the host name and port, and the **/api/hola** resource URL:

```
[student@workstation hola]$ curl -si http://localhost:8080/api/hola
HTTP/1.1 200 OK
...
Content-Type: text/plain; charset=UTF-8
...
Hola de localhost
Value in property 'a' => Uno
Value in property 'b' => Dos
```

Note that the value of **a** is equal to the value set in the **project-defaults.yml** file, whereas the value of **b** is **Dos** even though it is set to null in the **project-defaults.yml** file. The value of **b** is set to the **defaultValue** provided in the source code.

- 6.5. Stop the microservice.

Switch to the terminal running the microservice and press **Ctrl+C**

► 7. Override the properties using command-line flags and test the application.

- 7.1. Override the values of the **a** and **b** properties:

```
[student@workstation hola]$ java -Dcom.redhat.hola.config.a=Tres \
-Dcom.redhat.hola.config.b=Cuatro \
-jar target/hola-swarm.jar
...
2018-05-26 11:27:40,182 INFO [org.wildfly.swarm] (main) WFSWARM99999: WildFly
Swarm is Ready
```

- 7.2. Test the microservice and verify that new values for the properties are displayed:

```
[student@workstation hola]$ curl -si http://localhost:8080/api/hola
...
Hola de localhost
Value in property 'a' => Tres
Value in property 'b' => Cuatro
```

The values provided on the command line override the values provided in the **project-defaults.yml** file.

- 7.3. Stop the microservice.

Switch to the terminal running the microservice and press **Ctrl+C**

► 8. Clean up.

- 8.1. Commit your changes to your local Git repository:

```
[student@workstation hola]$ git commit -a -m \
"Finished the exercise."
```

- 8.2. Remove the **hola** project from the IDE workspace.

- 8.3. If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation hola]$ git reset --hard \
origin/do292-hola-local-begin
```

This concludes the guided exercise.

Configuring a Maven Project for WildFly Swarm

Objectives

After completing this section, students should be able to configure a Maven project to build a WildFly Swarm application and deploy it on an OpenShift cluster.

Configuring Maven Dependencies for WildFly Swarm Applications

To simplify managing versions of individual fractions needed for building a microservice on WildFly Swarm, Red Hat provides a Bill of Materials (BOM) artifact for Maven, which groups a set of well tested, stable fractions. You can then refer to fractions needed to build your applications without explicitly referencing their versions.

Add a `<dependencyManagement>` section in your Maven POM file to import the BOM as follows:

```
...
<properties>
  ...
  <version.wildfly.swarm>7.1.0.redhat-77</version.wildfly.swarm>
</properties>
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>bom</artifactId>
      <version>${version.wildfly.swarm}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  <dependencies>
<dependencyManagement>
  ...

```

If your application needs to use the MicroProfile APIs, then import the BOM for Eclipse MicroProfile:

```
...
<properties>
  ...
  <version.microprofile>1.2</version.microprofile>
</properties>
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.microprofile</groupId>

```

```

<artifactId>microprofile</artifactId>
<version>${version.microprofile}</version>
<type>pom</type>
<scope>provided</scope>
</dependency>
<dependencies>
<dependencyManagement>
...

```

Once the BOMs are imported, declare all the fractions you need for the microservice as Maven dependencies:

```

...
<dependencies>
<dependency>
<groupId>org.wildfly.swarm</groupId>
<artifactId>cdi</artifactId>
</dependency>
<dependency>
<groupId>org.wildfly.swarm</groupId>
<artifactId>jaxrs</artifactId>
</dependency>
<dependency>
<groupId>org.wildfly.swarm</groupId>
<artifactId>microprofile-config</artifactId>
</dependency>
...
<dependencies>
...

```

Configuring the Maven WildFly Swarm Plug-in

The Maven WildFly Swarm plug-in builds your microservice and packages it as an executable fat JAR.

Configure the WildFly Swarm plug-in in your Maven POM file as follows in the **<plugins>** section:

```

...
<plugin>
<groupId>org.wildfly.swarm</groupId>
<artifactId>wildfly-swarm-plugin</artifactId>
<version>${version.wildfly.swarm}</version>
<configuration>
<useUberJar>true</useUberJar>
</configuration>
<executions>
<execution>
<goals>
<goal>package</goal>
</goals>
</execution>

```

```

</executions>
</plugin>
...

```

You can then execute the Maven **package** goal to build the fat JAR:

```
$ mvn clean package
```

Apart from running the fat JAR directly on the command line, you can also run the microservice using the **run** goal of the Maven WildFly Swarm plug-in:

```
$ mvn wildfly-swarm:run
```

Configuring the Fabric8 Maven Plug-in (FMP) for WildFly Swarm

Configure the FMP in the **<plugins>** section of the Maven POM file as follows:

```

...
<artifactId>myservice</artifactId>
<packaging>war</packaging>
...
<properties>
    <version.fabric8.plugin>3.5.38</version.fabric8.plugin>
    <fabric8.generator.fromMode>istag</fabric8.generator.fromMode> ①
    <fabric8.generator.from>redhat-openjdk18-openshift</fabric8.generator.from> ②
    ...
</properties>
...
<plugins>
    ...
    <plugin>
        <groupId>io.fabric8</groupId>
        <artifactId>fabric8-maven-plugin</artifactId>
        <version>${version.fabric8.plugin}</version>
        <configuration>
            <generator>
                <includes>
                    <include>wildfly-swarm</include> ③
                </includes>
                <excludes>
                    <exclude>webapp</exclude> ④
                </excludes>
            </generator>
        </configuration>
        <executions>
            <execution>
                <id>fmp</id>
                <goals> ⑤
                    <goal>resource</goal>
                    <goal>build</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>

```

```

    </execution>
  </executions>
</plugin>
</plugins>
...

```

- ➊ Declares that the plug-in should use OpenShift image stream tags to identify the S2I builder image for building the container image.
- ➋ The image stream tag identifying the S2I builder image used to build the container image. In this case, you are using the official OpenJDK 1.8 image from the Red Hat Container Catalog.
- ➌ Generator configuration making the plug-in use the **wildfly-swarm** generator instead of the default. This is needed because the fabric8 plug-in categorizes the project as a regular web application due to the packaging type being set to WAR, and tries to build the container image using the **webapp** generator.
- ➍ Exclude the **webapp** generator and force the build to use the **wildfly-swarm** generator.
- ➎ Bind the **fabric8:resource** and **fabric8:build** goals to the standard life-cycle phases of Maven. Binding these goals makes it easy for the developer to run a single goal, **fabric8:deploy**, which builds the application, generates OpenShift resources, builds the container image, and deploys the application to an OpenShift cluster in one go.

Once the FMP is configured, create a project in OpenShift and run the **fabric8:deploy** goal to deploy the microservice to the OpenShift cluster:

```
$ mvn fabric8:deploy
```

Customizing OpenShift Resources with FMP

The OpenShift resources created by the Fabric8 Maven plug-in can be customized with YAML *resource fragments*, which are located in the **src/main/fabric8** directory of the Maven project. FMP merges the attributes defined in these resource fragments with defaults for each OpenShift resource type.

The fabric8 maven plug-in examines each resource fragment file in the **src/main/fabric8** directory, and looks for the **kind** attribute which indicates the type of resource it needs to create.

If there is no explicit **kind** attribute provided in the resource fragment file, it checks the file name of the resource fragment and creates the resources accordingly. Default file names for the most common resource types are listed in the table below. A full list of resource types and the corresponding file name mapping is provided in the references for this section:

Resource	Filename
DeploymentConfig	dc.yml or deploymentconfig.yml
Route	route.yml
Service	service.yml
ConfigMap	cm.yml or configmap.yml
Template	template.yml

The following section briefly discusses some of the OpenShift resources that can be customized using YAML resource fragments.

Customizing OpenShift Configuration Maps

It is generally a good practice to externalize application configuration for cloud-native applications, and avoid bundling configuration details within the container image. OpenShift provides the Configuration Map resource as a way to store configuration data.

The configuration map resource can be provided as a YAML resource fragment file. For example, the WildFly Swarm **project-defaults.yml** file can be provided as a configuration map:

```
---
apiVersion: v1
kind: ConfigMap
data:
  project-defaults.yml: |
    com:
      mycompany:
        myapp:
          config:
            var1: 100
            var2: value2

metadata:
  name: myconfigmap
```

Given the resource fragment above, the FMP creates a configuration map resource called **myappconfig**, which can then be used by applications running the OpenShift cluster. Note that the **data** attribute can contain multiple key-value pairs separated by a colon. The pipe (|) symbol next to the **project-defaults.yml** key indicates a multi-line value that is to be treated as a single value.

Customizing OpenShift Deployment Configuration

An example deployment configuration resource fragment that uses the configuration map resource defined previously, as well as customizing resource limits for the application looks like the following:

```
spec:
  replicas: 1
  template:
    spec:
      containers:
        - env:
            - name: JAVA_OPTIONS
              value: "-Dswarm.project.stage.file=file:///app/config/project-
defaults.yml" ①
        resources:
          limits: ②
            memory: 1024Mi
            cpu: 500m
        volumeMounts: ③
          - name: swarm-config
```

```

    mountPath: /app/config
  volumes:
    - configMap:
        name: myconfigmap ④
        name: swarm-config

```

- ➊ The data in the configuration map is mounted as a file inside the application container in the **/app/config** folder. The name of the file is the key name in the configuration map, that is **project-defaults.yml**.

The file is passed as an argument to the **swarm.project.stage.file** system property, which is read by WildFly Swarm at startup. Note that the property is set on the **JAVA_OPTIONS** environment variable, which is passed in as arguments when starting the fat JAR in the container.

- ➋ Set resource limits for the application, in this case memory and CPU limits are set.
- ➌ Refers to the volume name and the location where it should be mounted in the container.
- ➍ Refers to the name of the configuration map that should be mounted as a volume in the container.

Customizing OpenShift Routes

You can customize the default route URL generated by the FMP by providing a route resource fragment as follows:

```

spec:
  port:
    targetPort: 8080
  to:
    kind: Service
    name: myservice
  host: mycustomroute.example.com

```

The **host** attribute provides the custom route URL for the service.

Health Checks for WildFly Swarm Applications on OpenShift

You can use the health check API to integrate with the OpenShift probes feature. In OpenShift, the health of an application is assessed using probes, which is an action executed periodically by the OpenShift cluster to perform diagnostics on running containers.

OpenShift provides readiness and liveness probes for containers. These probes are configured in the deployment configuration resource fragment for the microservice:

```

spec:
  template:
    spec:
      containers:
        - readinessProbe:
            httpGet:
              path: /health
              port: 8080
              scheme: HTTP
            ...
        livenessProbe:

```

```

httpGet:
  path: /health
  port: 8080
  scheme: HTTP
  ...

```

The fragment above assumes that the `/health` endpoint exposes the health data for the application.

Demonstration: Customizing OpenShift Resources with the Fabric8 Maven Plug-in

1. Briefly review the POM file of the **hola** microservice Maven project. Note the reference to a parent POM file. Open the parent POM file.
2. In the parent POM file, note the system properties for the Fabric8 Maven plug-in and the WildFly Swarm Maven plug-in.
3. Switch to the **hola** Maven project POM file, and note the reference to the WildFly Swarm Bill of Materials (BOM), and the Eclipse MicroProfile API artifacts in the `<dependencyManagement>` section.
4. Note the configuration of the Fabric8 Maven plug-in to configure the WildFly Swarm generator, and the WildFly Swarm Maven plug-in in the `<plugins>` section.
5. Review the resource fragment files provided in the `src/main/fabric8` folder of the project.
6. Log in to OpenShift as the **developer** user and create the **mydemo** project.
7. Use the Fabric8 Maven Plug-in to deploy the microservice to the OpenShift cluster:

```
$ mvn clean fabric8:deploy
```

8. Inspect the generated resource YAML files, which combines the default configuration and the resource fragment files in the `src/main/fabric8` folder of the project.
9. Inspect the default route URL generated by the FMP:

```
$ oc get route
NAME      HOST/PORT          ...
hola     hola-mydemo.apps.lab.example.com ...
```

10. Wait for the microservice pod to be ready and running, and then test the microservice API using the **curl** command:

```
$ curl -si http://hola-mydemo.apps.lab.example.com/api/hola
```

11. Edit the route resource fragment file in the `src/main/fabric8` folder. Add a **host** attribute and set the value to **bueno.apps.lab.example.com**.
12. Instead of redeploying the entire microservice, you can recreate the route resource using the **fabric8:resource-apply** goal. However, you first need to delete the existing route resource before recreating it:

```
$ oc delete route hola
$ mvn fabric8:resource-apply
```

13. Inspect the route resource YAML file generated by the FMP after you added the **host** attribute. Verify that a **host** attribute with the new route URL is seen in the generated resource YAML file.
14. Verify that the new route URL is **bueno.apps.lab.example.com**:

```
$ oc get route
NAME      HOST/PORT      ...
hola     bueno.apps.lab.example.com ...
```

15. Test the microservice using the new route:

```
$ curl -si http://bueno.apps.lab.example.com/api/hola
```

This concludes the demonstration.



References

Fabric8 Maven Plug-in goals

<https://maven.fabric8.io/#goals>

Fabric8 Maven Plug-in resource fragments

<http://maven.fabric8.io/#resource-fragments>

Fabric8 Maven Plug-in resource types and default file names mapping

<https://github.com/fabric8io/fabric8-maven-plugin/blob/master/core/src/main/java/io/fabric8/maven/core/util/KubernetesResourceUtil.java#L234-L280>

WildFly Swarm Maven plug-in

<https://wildfly-swarm.gitbooks.io/wildfly-swarm-users-guide/content/v/1.0.0.Alpha6/maven-plugin.html>

More information about WildFly Swarm Bill of Materials (BOM) is available at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/wildfly_swarm_runtime_guide/index#using-a-bom

More information about WildFly Swarm health checks is available at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/wildfly_swarm_runtime_guide/index#mission-health-check-wf-swarm

More information about externalizing WildFly Swarm configuration is available at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/wildfly_swarm_runtime_guide/index#mission-configmap-wf-swarm

► Guided Exercise

Configuring a Maven Project for WildFly Swarm

In this exercise, you will configure a Maven project for the **hola** microservice and deploy it to an OpenShift cluster.

Outcomes

You should be able to :

- Configure the Fabric8 Maven plug-in (FMP) for the **hola** microservice.
- Customize the route resource fragment in the **hello-microservices** project to generate a custom route URL for the **hola** microservice.
- Configure the resource fragment for a configuration map that stores configuration data for the **hola** microservice.

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-fmp-resources-*** branches in the classroom Git repository, containing the source code for the **hola** microservice as part of the **hello-microservice/hola** Maven project.
- The **org.wildfly.swarm:bom**, **io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts on the classroom Nexus proxy server.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab fmp-resources setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-fmp-resources-solution** of the **hello-microservices** Git repository.

- 1. Switch to the **do292-fmp-resources-begin** branch in the **hello-microservices** Git repository and import the **hola** project into JBoss Developer Studio.
- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-fmp-resources-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
do292-fmp-resources-begin
...
Switched to a new branch 'do292-fmp-resources-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/hello-microservices/hola** folder as a Maven project.
- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-fmp-resources-begin**.

▶ 2. Configure the Fabric8 Maven plug-in for the microservice.

- 2.1. Inspect the Maven POM file for the microservice.

Open the **pom.xml** file from the **hola** project. Switch to the **pom.xml** tab.

Note the reference to the parent POM file, which provides some system properties. These properties are used to define the version strings for Maven artifacts required to build the microservice.

Note the reference to the Bill of Materials (BOM) dependencies for the WildFly Swarm distribution and the MicroProfile API.

Note the use of the artifacts in the **org.wildfly.swarm** group as dependencies for building the microservice.

Note the WildFly Swarm plug-in configuration in the **<plugins>** section of the POM file. It is configured to package the microservice as a fat JAR.

- 2.2. Add the Fabric8 Maven Plug-in (FMP) to the Maven life cycle.

At the end of the **pom.xml** file, before the closing **</plugins>** element, there is a reference to the FMP. The code for the **<plugin>** element is commented out in the POM file. Uncomment this section to enable the FMP for the microservice.

Use the listing that follows as a reference for adding FMP to the Maven life cycle:

```

...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <generator>
      <includes>
        <include>wildfly-swarm</include>
      </includes>
      <excludes>
        <exclude>webapp</exclude>
      </excludes>
    </generator>
  </configuration>
</plugin>
...

```

► 3. Inspect the REST API implementation for the microservice.

3.1. Inspect the **HolaResource** class.

Open the **HolaResource** class from the
com.redhat.training.msa.hola.rest package.

This class implements the REST API for the microservice using JAX-RS.

Note that two String properties (**a** and **b**) are injected into this class using the
@ConfigProperty annotation. In an earlier exercise, you provided values for these
properties in a YAML file packaged inside the application, and also overrode the
values using command line arguments when starting the fat JAR for the microservice.

In this exercise, you will provide values for these properties using OpenShift
configuration maps.

Inspect the **hola()** method. It prints a welcome message in Spanish along with the
host name where the microservice is currently running, and also prints the value of the
two String properties.

► 4. Customize the route and configuration map resource fragment files to set a custom route
URL, and configure application properties respectively.

Inspect the resource fragment files provided in the **/src/main/fabric8** folder of the
project. There are four resource fragment files in this folder.

4.1. Set the microservice configuration in the configuration map resource fragment.

The `cm.yml` file declares a configuration map named `app-config`. It contains properties to configure WildFly Swarm in the `swarm.*` namespace. Two application properties, `a` and `b` under the `com.redhat.hola.config` are also declared.

Edit this file and set values for the **a** and **b** properties as follows:

```
---  
apiVersion: v1  
kind: ConfigMap  
data:  
    project-defaults.yml: |  
    ...  
        com:  
            redhat:  
                hola:  
                    config:  
                        a: Tres  
                        b: Cuatro  
metadata:  
    name: app-config
```

4.2. Inspect the deployment configuration resource fragment.

The **deployment.yml** file customizes the OpenShift deployment configuration. Note how the **JAVA_OPTIONS** environment variable is augmented with a system property (**-Dswarm.project.stage.file**) that indicates the path to the **project-defaults.yml** file that is read by WildFly Swarm at startup:

This file is complete. Do not make any changes to this file.

```
spec:  
  containers:  
    - env:  
        - name: JAVA_OPTIONS  
          value: "-Dswarm.project.stage.file=file:///app/config/project-  
defaults.yml"
```

Notice also the references to the configuration map named **app-config** from the previous step, and how it is mounted as a file inside the container at the **/app/config** path:

```
        . . .
        volumeMounts:
          - name: config
            mountPath: /app/config
      volumes:
        - configMap:
            name: app-config
            name: config
```

4.3. Configure a custom route URL in the route resource fragment.

The **route.yml** file creates a route resource that points to the **hola** service.

Edit this file and add a **host** attribute as follows:

```
...
  to:
    kind: Service
    name: ${project.artifactId}
  host: hola.apps.lab.example.com
```

Note

Ensure that the **host** attribute is indented to align with the **to** attribute.

- 4.4. Inspect the service definition resource fragment for the microservice.

The **service.yml** file declares that the OpenShift service is available on port 8080 (the **port** attribute), and that requests are routed to port 8080 (**targetPort**) in the containers for the **hola** microservice. Do not make any changes to this file.

- 5. Deploy the microservice to the OpenShift cluster.

- 5.1. Log in to OpenShift and create the **hola** OpenShift project:

```
[student@workstation hello-microservices]$ oc login -u developer -p redhat \
  https://master.lab.example.com
Login successful.
...
[student@workstation hello-microservices]$ oc new-project hola
Now using project "hola" on server "https://master.lab.example.com:443".
...
```

- 5.2. Deploy the microservice to the OpenShift cluster:

```
[student@workstation hello-microservices]$ cd hola
[student@workstation hola]$ mvn clean fabric8:deploy
[INFO] OpenShift platform detected
...
[INFO] Creating a Service from openshift.yml namespace hola name hola
...
[INFO] Creating a ConfigMap from openshift.yml namespace hola name app-config
...
[INFO] Creating a DeploymentConfig from openshift.yml namespace hola name hola
...
[INFO] Creating Route hola:hola host: hola.apps.lab.example.com
...
[INFO] BUILD SUCCESS
```

- 6. Verify the OpenShift resources created by the FMP and test the microservice.

- 6.1. In the terminal window from the **hola** folder, view the generated route resource YAML file:

```
[student@workstation hola]$ cat \
target/classes/META-INF/fabric8/openshift/hola-route.yml
```

Verify that a **host** attribute is seen as follows:

```
...
apiVersion: v1
kind: Route
...
spec:
  host: hola.apps.lab.example.com
  port:
  ...
...
```

- 6.2. Verify that a custom route matching the route URL provided in the route resource fragment is created:

```
[student@workstation hola]$ oc get route
NAME      HOST/PORT      ...
hola     hola.apps.lab.example.com ...
```

- 6.3. Verify that a configuration map called **app-config** is created:

```
[student@workstation hola]$ oc describe cm app-config
Name: app-config
...
Data
=====
project-defaults.yml:
-----
...
com:
  redhat:
    hola:
      config:
        a: Tres
        b: Cuatro
...
```

- 6.4. Test the microservice using the custom route URL:

```
[student@workstation hola]$ curl -si http://hola.apps.lab.example.com/api/hola
HTTP/1.1 200 OK
...
Hola de hola.apps.lab.example.com
Value in property 'a' => Tres
Value in property 'b' => Cuatro
```

► 7. Clean up.

- 7.1. Delete the **hola** OpenShift project.

```
[student@workstation hola]$ oc delete project hola  
project "hola" deleted
```

7.2. Commit your changes to your local Git repository:

```
[student@workstation hola]$ git commit -a -m \  
"Finished the exercise."
```

7.3. Remove the **hola** project from the IDE workspace.

7.4. If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation hola]$ git reset --hard \  
origin/do292-fmp-resources-begin
```

This concludes the guided exercise.

Describing the Coolstore Microservice Application

Objectives

After completing this section, students should be able to describe the components and architecture of the Coolstore microservice application.

Describing the Coolstore Application

Coolstore is an online shopping application composed of a number of microservices, implemented using different Java frameworks and runtimes supported by Red Hat OpenShift Application Runtimes product.

Throughout this course, you will implement parts of the Coolstore application, one microservice at a time and, by the end of this course, you will be able to run a complete but simplified version of the application on an OpenShift cluster.

The Coolstore application you will see in this course is composed of five microservices and two databases.

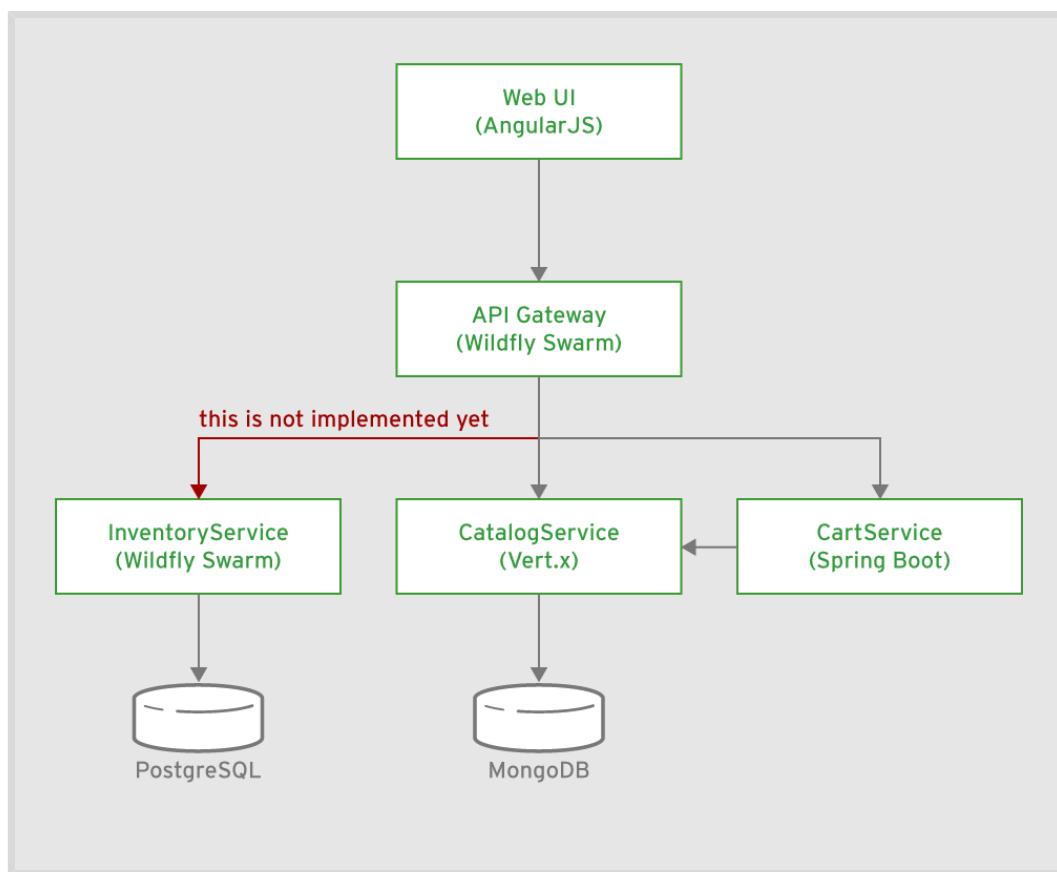


Figure 2.1: Coolstore application microservices and containers

A larger, more full-featured version of the Coolstore application is available from JBoss Demo Central in GitHub. This version of the Coolstore application not only includes more microservices, but also uses additional Red Hat products, such as Fuse.

The main components of the Coolstore application are three back-end microservices that implement the Coolstore application business logic:

Inventory Service

The Coolstore Inventory Service manages the product inventory, and stores inventory data, such as quantity in stock, in a PostgreSQL database. The Inventory Service is developed using WildFly Swarm and the Microprofile specification.

Catalog Service

The Coolstore Catalog Service manages the product catalog, and stores product information, such as the product price, in a MongoDB database. The Catalog Service is developed using the Vert.x framework.

Cart Service

The Coolstore Cart Service manages a visitor shopping cart, and currently stores all data, such as quantity to order for a product, in memory. The Cart Service also calculates the shipping costs. The Cart Service is developed using the Spring Boot framework.

Besides these three back-end microservices, the Coolstore application also includes two front-end microservices, which together provide a web interface for the online store:

Web UI

The Coolstore UI application provides a rich web interface to navigate the product catalog and add products to the shopping cart. The Coolstore UI is developed using the Angular JavaScript client framework. The Coolstore UI application also embeds a web server based on Node.js to serve the static HTML, CSS, and JavaScript files to the user's web browser.

API Gateway

The Coolstore API Gateway isolates the front-end Web UI from changes in back-end services APIs, and adds *Cross Origin Resource Sharing* (CORS) headers required by rich web user interfaces. The API Gateway is developed using WildFly Swarm and the Microprofile specification.

The Coolstore UI includes menus and links for a few features that are not yet implemented, such as user authentication and checkout the shopping cart. You should ignore them.

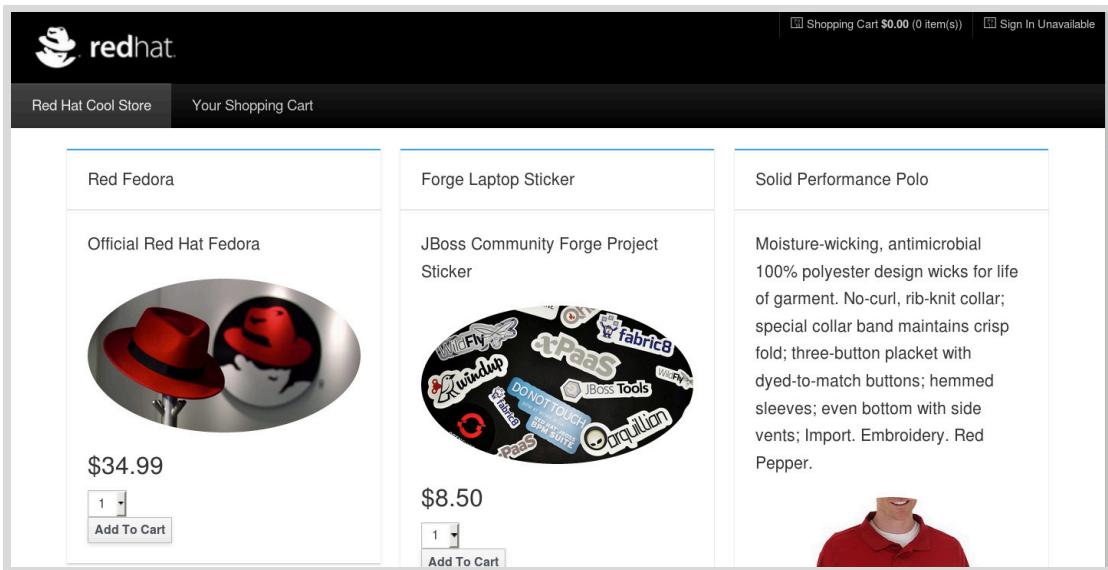


Figure 2.2: Coolstore application user interface

The Coolstore back-end microservices may depend on each other, for example: the Cart Service requests current product price information from the Catalog Service. Not all such dependencies are implemented right now, for example: checking out a shopping cart would involve getting product inventory data, but the current Cart Service does not invoke the Inventory Service.

All components of the Coolstore application, including the PostgreSQL and MongoDB databases, run as containers on an OpenShift cluster. Each microservice project on the classroom Git server includes both Java source code and OpenShift resource fragment files, managed by the Fabric8 Maven Plug-in.

Each microservice also includes complete OpenShift template resource files in YAML format to deploy their database pods.

Each database is initialized by deployment hooks, which are embedded into the OpenShift templates. The database templates also assume that persistent volumes are available. The classroom environment has these volumes already provisioned.



References

The complete Coolstore application on GitHub

<https://github.com/jbossdemocentral/coolstore-microservice>

Mozilla Developers Network article about Cross-Origin Resource Sharing (CORS)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

► Quiz

Describing the Coolstore Microservice Application

Choose the correct answers to the following questions:

- ▶ 1. Why does the Coolstore Web UI require Cross Origin Resource Sharing (CORS) headers, provided by the API Gateway?
 - a. Because non-Java frameworks, such as AngularJS, need these headers to function.
 - b. Because the Web UI static files and the microservices HTTP API end points are served by different web servers.
 - c. Because AngularJS is only able to process Ajax calls that include these headers.
 - d. Because the Web UI does not invoke each microservice HTTP API directly, but invokes them indirectly through the API Gateway.
- ▶ 2. When the checkout function is implemented, which microservice would become a client of the Inventory Service?
 - a. Coolstore UI
 - b. API Gateway
 - c. Catalog Service
 - d. Cart Service
- ▶ 3. Which two of the Coolstore application microservices are developed using the MicroProfile specification? (Choose two.)
 - a. Coolstore UI
 - b. API Gateway
 - c. Inventory Service
 - d. Catalog Service
 - e. Cart Service
- ▶ 4. Which two of the following new functions, when added to the Coolstore application, would require changes to the API Gateway? (Choose two.)
 - a. New shopping cart total cost calculation logic.
 - b. New microservice to manage user profiles.
 - c. New mobile user interface for shoppers.
 - d. New product details page layout.
 - e. New caching layer for product catalog data.

► **5. Which of the following Coolstore microservices would lose all its data if its pod were terminated and recreated by OpenShift?**

- a. Coolstore UI
- b. API Gateway
- c. Inventory Service
- d. Catalog Service
- e. Cart Service

► Solution

Describing the Coolstore Microservice Application

Choose the correct answers to the following questions:

- ▶ **1. Why does the Coolstore Web UI require Cross Origin Resource Sharing (CORS) headers, provided by the API Gateway?**
 - a. Because non-Java frameworks, such as AngularJS, need these headers to function.
 - b. Because the Web UI static files and the microservices HTTP API end points are served by different web servers.
 - c. Because AngularJS is only able to process Ajax calls that include these headers.
 - d. Because the Web UI does not invoke each microservice HTTP API directly, but invokes them indirectly through the API Gateway.
- ▶ **2. When the checkout function is implemented, which microservice would become a client of the Inventory Service?**
 - a. Coolstore UI
 - b. API Gateway
 - c. Catalog Service
 - d. Cart Service
- ▶ **3. Which two of the Coolstore application microservices are developed using the MicroProfile specification? (Choose two.)**
 - a. Coolstore UI
 - b. API Gateway
 - c. Inventory Service
 - d. Catalog Service
 - e. Cart Service
- ▶ **4. Which two of the following new functions, when added to the Coolstore application, would require changes to the API Gateway? (Choose two.)**
 - a. New shopping cart total cost calculation logic.
 - b. New microservice to manage user profiles.
 - c. New mobile user interface for shoppers.
 - d. New product details page layout.
 - e. New caching layer for product catalog data.

► **5. Which of the following Coolstore microservices would lose all its data if its pod were terminated and recreated by OpenShift?**

- a. Coolstore UI
- b. API Gateway
- c. Inventory Service
- d. Catalog Service
- e. Cart Service

► Guided Exercise

Deploying Microservices with the WildFly Swarm Runtime

In this exercise, you will complete the Inventory microservice of the Coolstore application using the Wildfly Swarm runtime.

Outcomes

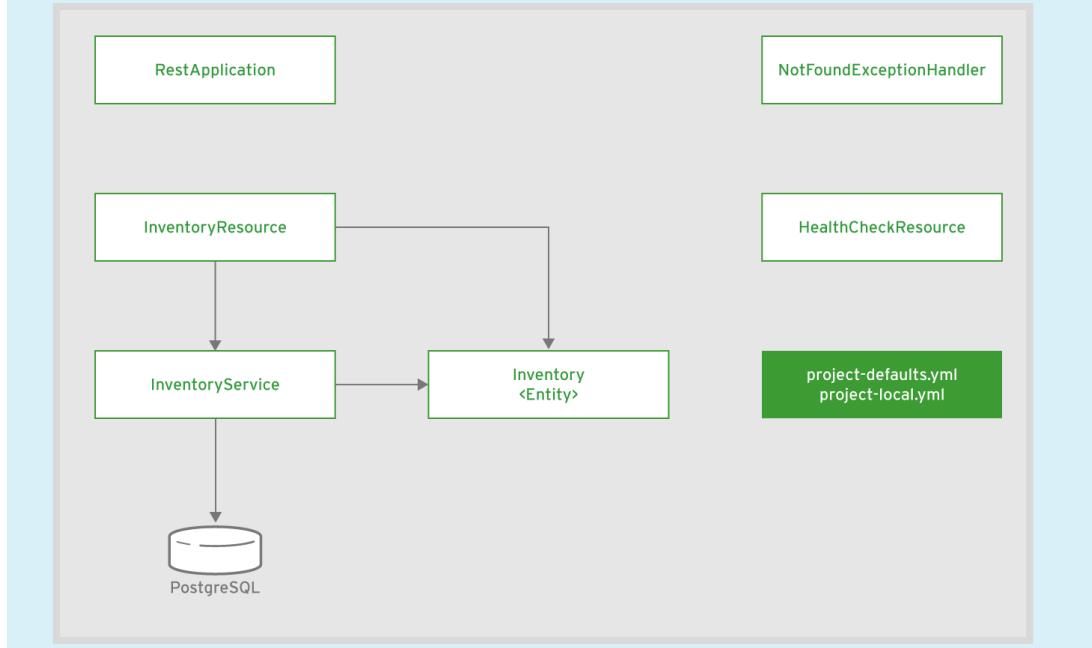
You should be able to:

- Implement and test the service layer code that connects to a PostgreSQL database and fetches Inventory information.
- Implement and test a REST API that delegates calls to the service layer.
- Implement and test REST API endpoints for health checks.
- Fetch application configuration using OpenShift configuration maps.
- Deploy the microservice to OpenShift.

For each outcome, there is a corresponding Git branch that provides the completed code up to that point in the exercise. If you are unable to complete the steps and need help, you can browse the solution code for each branch using the classroom Gitweb server. You can also switch to any of the intermediate branches and continue from that point, skipping previous steps.

The Inventory Microservice

The following diagram shows a high-level view of the inventory microservice implementation.



The Inventory microservice is composed of the **InventoryService** class, which fetches inventory data from a PostgreSQL database using the Java Persistence API (JPA).

The **InventoryResource** class implements the REST API for the microservices using the Java API for RESTful Web Services (JAX-RS) API, and delegates requests to the **InventoryService** class.

Both the **InventoryService** and the **InventoryResource** class use the **Inventory** bean class to encapsulate inventory data. The **Inventory** class is annotated to function as a JPA entity.

The **HealthCheckResource** class implements the REST API for health checks. It signals whether the microservice is in a healthy or unhealthy state.

To perform this exercise, you need access to:

- A running OpenShift cluster
- At least one available persistent volume for PostgreSQL database data
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-inventory-lab-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **coolstore/inventory-service** Maven project
- The **org.wildfly.swarm:bom**, **io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts in the classroom Nexus proxy server

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab inventory-deploy setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-inventory-lab-solution** of the **coolstore** Git repository.

► 1. Switch to the **do292-inventory-lab-begin** branch in the **coolstore** Git repository and import the project into JBoss Developer Studio.

1.1. If you have not done so yet, clone the **coolstore** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **coolstore** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/coolstore
Cloning into 'coolstore'...
...
```

- 1.2. If you already have a clone of the **coolstore** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-inventory-lab-begin** branch:

```
[student@workstation ~]$ cd ~/coolstore
[student@workstation coolstore]$ git checkout do292-inventory-lab-begin
...
Switched to a new branch 'do292-inventory-lab-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/coolstore/inventory-service** folder as a Maven project.



Warning

During the import process, the IDE will show an error dialog saying **An internal error occurred during: "Importing Maven projects". java.lang.NullPointerException**. Click **OK** to dismiss the dialog and continue. You will fix the errors in the project as you proceed with the exercise.

- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-inventory-lab-begin**.
 - 1.6. Ignore errors in the starter project.
JBoss Developer Studio fails to build the project because there are incomplete files in the project. You will progressively add code throughout this exercise and build the project successfully.
- ▶ 2. Review the initial state of the **inventory-service** project. Most of the classes are incomplete and you cannot run the unit tests, or deploy the microservice to an OpenShift cluster, at this point.
- 2.1. Inspect the Maven POM file for the inventory microservice.
Open the **pom.xml** file from the **inventory-service** project. Switch to the **pom.xml** tab.
You see a number of errors in the **pom.xml** file indicating missing dependencies.
You will modify this file as you progress through the exercise. For now, note the reference to a parent POM file providing values for system properties. These properties are used to define version strings for Maven artifacts required to build the microservice.
 - 2.2. Notice that the required Wildfly Swarm fractions (JPA, CDI, JAX-RS, and others) needed for this microservice and the database drivers are declared as Maven dependencies.
 - 2.3. Notice that the **org.wildfly.swarm:arquillian** dependency is declared in **test** scope for running unit tests in this microservice.
 - 2.4. Notice the Wildfly Swarm Maven plug-in declaration in the **plugins** section of the POM file. This is used to build and package a runnable JAR file with all dependencies bundled within it (a fat JAR).

- 3. Complete the project POM file so that the project builds successfully. You will not run unit tests to validate the changes you make while performing this step. The unit tests from Step 6 verifies both this step and the **InventoryService** implementation.



Note

If you want to catch up to Step 6, stash your changes, check out the branch named **do292-inventory-lab-service**, and continue from Step 7.

- 3.1. Add the Wildfly Swarm Bill of Materials (BOM) dependencies to the Maven POM file.

Open the **pom.xml** file from the **inventory-service** project. Switch to the **pom.xml** tab.

Right after the **<properties>** definition, add the **org.wildfly.swarm:bom** dependency.

Use the listing that follows as a reference for adding the dependency:

```
...
<version.postgresql>9.4.1207</version.postgresql>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>bom</artifactId>
      <version>${version.wildfly.swarm}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
...
```

- 3.2. Save your changes to the **pom.xml** file and update your project configuration. JBoss Developer Studio will take some time to rebuild the project and the errors in the project POM file should disappear.

- 4. Inspect the persistence layer code for the microservice. The microservice uses the Java Persistence API (JPA) to store inventory data in a PostgreSQL database.

- 4.1. Inspect the **Inventory** entity class.

Open the **Inventory** class from the **com.redhat.coolstore.inventory.model** package.

There is no need to make any changes to this class.

- 4.2. Inspect the persistence configuration for the microservice.

Open the **persistence.xml** file in the **/src/main/resources/META-INF** folder. Open the **persistence.xml** file. Click the **Source** tab to view the raw XML source.

A persistence unit named **primary** is declared in the file, which points to a datasource called **java:jboss/datasources/InventoryDS**.

- 5. Complete the **InventoryService** class.

The **InventoryService** class implements all logic required to retrieve inventory data from a PostgreSQL database using the Java Persistence API (JPA). The **InventoryServiceTest** class tests this functionality using an embedded H2 database that is populated as part of the unit tests.

- 5.1. Add the appropriate CDI scope for this class.

Open the **InventoryService** class from the **com.redhat.coolstore.inventory.service** package.

Add the **@ApplicationScoped** annotation to the **InventoryService** class:

```
...
import javax.enterprise.context.ApplicationScoped;
...
@ApplicationScoped
public class InventoryService {
...
}
```

- 5.2. Inject the persistence unit into the **InventoryService** class.

Add the **@PersistenceContext** annotation to the **InventoryService** class, and pass the persistence unit name to the **unitName** argument:

```
...
import javax.persistence.PersistenceContext;
...
public class InventoryService {

    @PersistenceContext(unitName = "primary")
    private EntityManager em;
...
}
```

- 5.3. Complete the **getInventory** method.

Use the injected entity manager to look up an inventory object with the given item ID and return the object as follows:

```
...
private EntityManager em;

public Inventory getInventory(String itemId) {
    Inventory inventory = em.find(Inventory.class, itemId);
    return inventory;
}
...
```

Remove the call to **return null** at the end of the **getInventory()** method.

- 6. Run the unit test for the **InventoryService** class.

- 6.1. Inspect the test cases for the **InventoryService** class

Open the **InventoryServiceTest** class from the **com.redhat.coolstore.inventory.service** package.

The test methods are complete. Do not make any changes to this class.

- 6.2. Note that the **InventoryServiceTest** class uses the **Arquillian** JUnit runner class.
Review the **newContainer()** method that bootstraps a Wildfly Swarm container as part of each test.
- Review the **createDeployment()** method that creates a WAR file containing the **InventoryService** class, along with other test artifacts and deploys the WAR file to the bootstrapped Wildfly Swarm container before running the tests.

- 6.3. Review the **getInventory()** test method.

The test invokes the **getInventory()** method on the **InventoryService** class and performs a number of assertions to verify the returned inventory object.

- 6.4. Review the **getNonExistingInventory()** test method.

The test invokes the **getInventory()** method on the **InventoryService** class with an invalid item ID as the argument, and asserts that the returned inventory object is null.

- 6.5. Run the **InventoryServiceTest** class as a JUnit test.

The **JUnit** view shows that both tests passed. It may take some time to resolve all test dependencies before the unit tests are executed.

If you see test failures, review your changes for this step and repeat the test until you get both tests to pass. Use the **Console** view in the bottom to troubleshoot test failures.

▶ 7. Implement the REST API for the microservice.

The **InventoryResource** class uses the JAX-RS API to implement a REST API for retrieving inventory objects. It delegates each request to the **InventoryService** class, and returns data in JSON format.

The **RestApiTest** class provides unit tests for the REST API exposed by the inventory microservice. Not all test methods are expected to pass at the end of this step. The tests related to the health probes pass only after you implement the health monitoring methods, later in this exercise.

Do not run unit tests to validate the changes you make while performing this step. The unit tests from Step 8 verifies this step.

Note

If you want to skip Step 7 and Step 8, stash your changes, check out the branch named **do292-inventory-lab-api** and continue from Step 9.

- 7.1. Inspect the **RestApplication** class.

Open the **RestApplication** class from the **com.redhat.coolstore.inventory** package.

The **RestApplication** class is annotated with **@ApplicationPath** to indicate the path that serves as the root URI for all resources in this microservice. This class is complete. Do not make any changes to it.

- 7.2. Inspect the incomplete **InventoryResource** class.

Open the **InventoryResource** class from the **com.redhat.coolstore.inventory.rest** package.

The **InventoryResource** class defines a single **getInventory()** method that takes an item ID as argument, and returns the inventory object corresponding to the input item ID in JSON format.

- 7.3. Add annotations to the **InventoryResource** class to handle calls to the `/inventory` path.

Since multiple clients could invoke methods on this class concurrently, this class should be declared in request scope:

```
...
import javax.enterprise.context.RequestScoped;
import javax.ws.rs.Path;
...
@Path("/inventory")
@RequestScoped
public class InventoryResource {
...
}
```

- 7.4. Add annotations to the **getInventory()** method. It should accept a single string argument called **itemId** containing the item ID that you want to look up in the inventory database.

Add the following annotations to the **getInventory()** method to handle HTTP GET requests to the `/inventory/{itemId}` path. The response should be an inventory object in JSON format. Also add the **@PathParam** annotation to the **itemId** argument:

```
...
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rsPathParam;
import javax.ws.rs.core.MediaType;
...
@GET
@Path("/{itemId}")
@Produces(MediaType.APPLICATION_JSON)
public Inventory getInventory(@PathParam("itemId") String itemId) {
...
}
```

- 7.5. Add code to delegate requests to the **InventoryService** class.

Invoke the **getInventory()** method on the **InventoryService** class and pass the **itemId** as input:

```
...
public Inventory getInventory(@PathParam("itemId") String itemId) {
    Inventory inventory = inventoryService.getInventory(itemId);
...
}
```

- 7.6. Add code to handle scenarios where invalid item ID is provided as input.

Continue editing the **getInventory()** method in the **InventoryResource** class. If the inventory item is not found in the database, throw a **NotFoundException** exception, else return the valid inventory object:

```

...
import javax.ws.rs.NotFoundException;
...
Inventory inventory = inventoryService.getInventory(itemId);

if (inventory == null) {
    throw new NotFoundException();
} else {
    return inventory;
}
...

```

Remove the `return null` statement at the end of the `getInventory()` method.

If you want to understand how `NotFoundException` exceptions are handled, look at the code in the `NotFoundExceptionHandler` class, found in the `com.redhat.coolstore.inventory.rest` package.

- 8. Complete the unit test for the `getInventory()` method of the `InventoryResource` class.

- 8.1. Inspect the test case for the `InventoryResource` class.

Open the `RestApiTest` class from the `com.redhat.coolstore.inventory` package.

Note that the `RestApiTest` class uses the `Arquillian` JUnit runner class.

Review the code in the `newContainer()` and `createDeployment()` bootstrap methods, which are similar to the ones in the `InventoryServiceTest` class.

Review the `before()` and `after()` methods, which instantiate a JAX-RS client instance before tests are executed, and destroys it when test execution completes.

- 8.2. Inspect the `testGetInventoryWhenItemIdDoesNotExist()` test method.

The test is provided fully complete. It instantiates a JAX-RS client request to the `/inventory/{itemId}` path and passes in an invalid item id as the argument. It then uses the Hamcrest library to assert that the HTTP status code of the response is **404 - not found**.

- 8.3. Complete the `testGetInventory()` test method using the code in the `testGetInventoryWhenItemIdDoesNotExist()` method as a reference.

Instantiate a JAX-RS client request to the `/inventory/{itemId}` path and pass in a value of **123456** as the item ID. The code to parse the JSON response as an object, and the required assertions are already provided.

Replace the `WebTarget target = null;` line in the `testGetInventory()` method with the following code:

```

...
    @Test
    @RunAsClient
    public void testGetInventory() throws Exception {
        WebTarget target = client.target("http://localhost:" + port)
            .path("/inventory")
            .path("/123456");
    ...
}
...

```

Save your changes to the **RestApiTest.java** file.

- 8.4. Run the **RestApiTest** class as a JUnit test.

The **JUnit** view shows that three out of five tests passed:

- **testError**
- **testGetInventory**
- **testGetInventoryWhenItemIdDoesNotExist**

The **JUnit** view also shows that other two tests failed:

- **testHealthCheckCombined**
- **testHealthCheckStatus**

These failures are expected at this point in the exercise because you have not yet implemented health checks for the microservice. If you get a different outcome, review your changes during this step and repeat the tests until you get the correct tests to pass.

- 9. Implement health checks for the microservice.

The **HealthCheckResource** class implements health checking for the microservice, and returns data in JSON format.

The **RestApiTest** class provides unit tests for the health checks in the microservice.

You will not run unit tests to validate the changes you make while performing this step. The unit tests from Step 10 verifies this step.



Note

If you want to skip Step 9 and Step 10, stash your changes, check out the branch named **do292-inventory-lab-health** and continue from Step 11.

- 9.1. Inspect the incomplete **HealthCheckResource** class.

Open the **HealthCheckResource** class from the **com.redhat.coolstore.inventory.rest** package.

Add annotations to the **HealthCheckResource** class to register the **check()** method to the Wildfly Swarm health check API. The **check()** method provides an endpoint called **/status** that responds to HTTP GET requests, and returns an **HealthStatus** object that indicates that the microservice is healthy:

```
...
import org.wildfly.swarm.health.Health;
...
@GET
@Path("/status")
@Health
public HealthStatus check() {
...
}
```

9.2. Add code to return an **HealthStatus** object in the **check()** method.

Return a **HealthStatus** instance having an attribute called **server-state** with a status of **UP**:

```
...
public HealthStatus check() {
    return HealthStatus.named("server-state").up();
...
}
```

Remove the **return null** statement at the end of the **check()** method.

▶ 10. Complete the unit tests for the health checks.

10.1. Inspect the **testHealthCheckCombined()** method.

Open the **RestApiTest** class from the **com.redhat.coolstore.inventory** package.

The **testHealthCheckCombined()** test instantiates a JAX-RS client request to the **/health** path, and gets a response from the microservice in JSON format. It then asserts the following:

- The status of the HTTP response is equal to 200
- The **outcome** attribute is equal to **UP**
- An array called **checks** exists and is of size 1
- The value of the **id** attribute in the **checks** array is equal to **server-state**
- The value of the **result** attribute in the **checks** array is equal to **UP**

Use the **testHealthCheckCombined()** method code as a reference to complete the **testHealthCheckStatus()** method

10.2. Complete the **testHealthCheckStatus()** test method.

Instantiate a JAX-RS client request to the **/status** path. The code to parse the JSON response and verify the values is already provided:

Replace the **WebTarget target = null;** line in the **testHealthCheckStatus()** method with the following code:

```
...
    @Test
    @RunAsClient
    public void testHealthCheckStatus() throws Exception {
        WebTarget target = client.target("http://localhost:"+ port)
            .path("/status");
        ...
    }
...
}
```

10.3. Run the completed **RestApiTest** class as a JUnit test.

The **JUnit** view shows that five out of five tests passed.

If you get test failures or a different outcome, review your changes during this step and repeat the tests until you get the correct tests to pass.

10.4. Optional: Perform a manual test of the health check API.

The code changes for the inventory microservice is now complete. To avoid having to provide and initialize an external PostgreSQL database, you can package and run the application using the **run** goal of the Wildfly Swarm plug-in, and invoke only the health endpoints. The **run** goal starts Wildfly Swarm using the **local** profile, which uses an embedded H2 database for the microservice instead of PostgreSQL.

Open a terminal window and run the **wildfly-swarm:run** goal:

```
[student@workstation ~]$ cd ~/coolstore/inventory-service
[student@workstation catalog-service]$ mvn clean wildfly-swarm:run -DskipTests
...
...WFLYSRV0049: WildFly Swarm 7.1.0.redhat-77 (WildFly Core 3.0.12.Final-redhat-1)
starting
...
...[org.wildfly.swarm.monitor.health] (main) Adding /health endpoint delegate: /
status
...
...WFLYSRV0010: Deployed "inventory-service-1.0.war" (runtime-name : "inventory-
service-1.0.war")
...WFSWARM99999: WildFly Swarm is Ready
...
```

Then open another terminal window and invoke the health endpoints using the **curl** command:

```
[student@workstation ~]$ curl -si http://localhost:8080/status
HTTP/1.1 200 OK
...
>{"id":"server-state","result":"UP"}

[student@workstation ~]$ curl -si http://localhost:8080/health
HTTP/1.1 200 OK
...
Content-Type: application/json
...
>{"checks": [
```

```
{"id":"server-state","result":"UP"}],  
"outcome": "UP"  
}
```

Switch back to the first terminal and press **Ctrl+C** to stop the application.

► 11. Deploy the Inventory microservice to OpenShift and test the microservice API.

11.1. Log in to OpenShift and create a new project.

From a terminal window, log in to OpenShift as the **developer** user and create the **inventory-service** project:

```
[student@workstation ~]$ oc login -u developer -p redhat \  
https://master.lab.example.com  
Login successful.  
...  
[student@workstation ~]$ oc new-project inventory-service  
Now using project "inventory-service" on server "https://  
master.lab.example.com:443".  
...
```

11.2. Inspect the template for the PostgreSQL database.

Enter the **inventory-service** folder inside the local **coolstore** Git repository and inspect the provided template for the PostgreSQL database.

The template relies on persistent volumes already provisioned for you in the classroom environment, and defines parameters for the database user name and password.

```
[student@workstation ~]$ cd ~/coolstore/inventory-service  
[student@workstation inventory-service]$ less \  
ocp/inventory-service-postgresql-persistent.yaml  
...  
parameters:  
...  
- description: Inventory Service database user name  
from: user[a-zA-Z0-9]{3}  
generate: expression  
name: INVENTORY_DB_USERNAME  
required: true  
- description: Inventory Service database user password  
from: '[a-zA-Z0-9]{8}'  
generate: expression  
name: INVENTORY_DB_PASSWORD  
required: true  
- description: Inventory Service database name  
name: INVENTORY_DB_NAME  
required: true  
value: inventorydb
```

11.3. Deploy a containerized PostgreSQL database.

Use the **oc new-app** command to deploy the database container. Use **jboss** as the value for the database user and password parameters:

```
[student@workstation inventory-service]$ oc new-app -f \
  ocp/inventory-service-postgresql-persistent.yaml \
  -p INVENTORY_DB_USERNAME=jboss -p INVENTORY_DB_PASSWORD=jboss
...
--> Success
...
--> Creating resources ...
service "inventory-postgresql" created
deploymentconfig "inventory-postgresql" created
persistentvolumeclaim "inventory-postgresql-pv" created
...
```

11.4. Verify that the PostgreSQL database is operational.

Verify that the persistent volume claim is bound:

```
[student@workstation inventory-service]$ oc get pvc
NAME           STATUS    VOLUME   CAPACITY  ACCESSMODES  ...
inventory-postgresql-pv  Bound     vol01    1Gi       RWO         ...
```

It may take some time for the database pod to start and be fully operational. Wait until the database pod is ready and in running state:

```
[student@workstation inventory-service]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
inventory-postgresql-1-dfpsd  1/1     Running   0          1m
```

Verify that sample inventory data has been populated in the database:

```
[student@workstation inventory-service]$ oc rsh inventory-postgresql-1-dfpsd
sh-4.2$ psql -d inventorydb -c 'select * from product_inventory'
 itemid | link | location | quantity
-----+-----+-----+-----+
 329299 | http://maps.google.com/?q=Raleigh | Raleigh | 736
 329199 | http://maps.google.com/?q=Raleigh | Raleigh | 512
 165613 | http://maps.google.com/?q=Raleigh | Raleigh | 256
 165614 | http://maps.google.com/?q=Raleigh | Raleigh | 29
 165954 | http://maps.google.com/?q=Raleigh | Raleigh | 87
 444434 | http://maps.google.com/?q=Raleigh | Raleigh | 443
 444435 | http://maps.google.com/?q=Raleigh | Raleigh | 600
 444436 | http://maps.google.com/?q=Tokyo | Tokyo | 230
(8 rows)
```

Type **exit** to exit from the database container shell.



Note

If you need to redeploy the PostgreSQL database, for example because you used incorrect parameter values, then you must re-provision the persistent volume. Perform the clean up Step 16 and restart from Step 11

11.5. Inspect the configuration map resource fragment for the inventory microservice.

Open the **configmap.yml** file in the **/src/main/fabric8** folder.

Do not make any changes to this file. Note the details to connect to the PostgreSQL database for the microservice.

- 11.6. Inspect the deployment configuration resource fragment for the inventory microservice.

Open the **deployment.yml** file in the **/src/main/fabric8** folder.

Do not make any changes to this file. Note the reference to the configuration map created in the previous step:

```
...
volumes:
- configMap:
  name: app-config
  name: config
...
```

The configuration map resource is mounted as a file in the **/app/config** directory in the container:

```
...
volumeMounts:
- name: config
  mountPath: /app/config
...
```

Finally, the configuration file is passed as an argument to the **swarm.project.stage.file** system property:

```
...
- name: JAVA_OPTIONS
  value: "-Dswarm.project.stage.file=file:///app/config/project-defaults.yml"
...
```

- 11.7. Add the Fabric8 Maven Plug-in (FMP) to the Maven life cycle.

Open the **pom.xml** file from the **inventory-service** project. Switch to the **pom.xml** tab.

At the end of the **pom.xml** file, before the closing **</plugins>** element, there is a reference to the FMP. The code for the **<execution>** element is commented out in the POM file. You only need to uncomment the **<execution>** element.

Use the listing that follows as a reference adding FMP to the Maven life cycle:

```
...
<plugins>
...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
  <executions>
    <execution>
```

```

<id>fmp</id>
<goals>
  <goal>resource</goal>
  <goal>build</goal>
</goals>
</execution>
</executions>
<configuration>
</configuration>
</plugin>
...

```

- 11.8. Configure the Fabric8 Maven Plug-in (FMP) to use the Wildfly Swarm generator.

Inside the **<configuration>** element of the FMP plug-in, add a **<generator>** with a **<configuration>** that references the **<wildfly-swarm>** generator.

Use the listing that follows as a reference for adding the generator:

```

...
<plugins>
...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <generator>
      <includes>
        <include>wildfly-swarm</include>
      </includes>
      <excludes>
        <exclude>webapp</exclude>
      </excludes>
    </generator>
  </configuration>
</plugin>
...

```

- 12. Deploy the microservice to the OpenShift cluster. Test and verify that the Inventory microservice is operational.

- 12.1. Deploy the Inventory microservice to the OpenShift cluster.

Use the **fabric8:deploy** Maven goal. Skip tests to have a faster deployment. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation inventory-service]$ mvn fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:resource (default) @ inventory-service ---
...
[INFO] --- maven-war-plugin:2.5:war (default-war) @ inventory-service ---
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ inventory-
service ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (default) @ inventory-service ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ inventory-service
...
...
[INFO] BUILD SUCCESS
...
```

- 12.2. Verify that the FMP created a configuration map that contains the expected database connection parameters:

```
[student@workstation inventory-service]$ oc describe configmap app-config
...
Data
=====
project-defaults.yml:
-----
swarm:
  datasources:
    data-sources:
      InventoryDS:
        driver-name: postgresql
        connection-url: jdbc:postgresql://inventory-postgresql:5432/inventorydb
        user-name: jboss
        password: jboss
...
```

- 12.3. Wait until the Inventory microservice pod is ready and running:

NAME	READY	STATUS	RESTARTS	AGE
inventory-postgresql-1-dfpsd	1/1	Running	0	4h
inventory-service-1-d4jn6	1/1	Running	0	3m
inventory-service-s2i-1-build	0/1	Completed	0	4m

Verify that the microservice logs resemble the following:

```
[student@workstation inventory-service]$ oc logs inventory-service-1-d4jn6
...
INFO [org.jboss.as.connector.subsystems.datasources] (MSC service thread 1-1)
WFLYJCA0001: Bound data source [java:jboss/datasources/InventoryDS]
...
```

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Swarm  
7.1.0.redhat-77 (WildFly Core 3.0.12.Final-redhat-1) started in 8589ms...  
...  
INFO [org.wildfly.swarm.runtime.deployer] (main) deploying inventory-  
service-1.0.war  
...  
INFO [org.jboss.as.jpa] (ServerService Thread Pool -- 16) WFLYJPA0010: Starting  
Persistence Unit (phase 1 of 2) Service 'inventory-service-1.0.war#primary'  
...  
INFO [org.wildfly.swarm] (main) WFSWARM99999: WildFly Swarm is Ready  
...
```

12.4. Invoke the HTTP API to fetch data for inventory item with ID **329299**.

Get the host name of the route that was created by FMP:

```
[student@workstation inventory-service]$ oc get route  
NAME           HOST/PORT  
inventory-service  inventory-service.apps.lab.example.com  ...
```

Use the **curl** command to invoke the **/inventory/329299** endpoint:

```
[student@workstation inventory-service]$ curl -si \  
http://inventory-service.apps.lab.example.com/inventory/329299  
HTTP/1.1 200 OK  
Content-type: application/json  
...  
{"itemId": "329299", "location": "Raleigh", "quantity": 736, "link": "http://  
maps.google.com/?q=Raleigh"}
```

► 13. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were successful:

```
[student@workstation inventory-service]$ lab inventory-deploy grade
```

► 14. Commit your changes to your local Git repository:

```
[student@workstation coolstore]$ git commit -a -m \  
"Finished the exercise."
```

► 15. Remove the **inventory-service** project from the IDE workspace.

► 16. **Optional:** Clean up to redo the exercise from scratch.



Important

Later during this course, you will need the Inventory microservice to be fully operational. Do not delete the **inventory-service** project in OpenShift unless you really want to start over this exercise.

16.1. Perform this and the following steps only if you wish to start over this exercise.

Reset your local repository to the remote branch:

```
[student@workstation coolstore]$ git reset --hard \
origin/do292-inventory-lab-begin
```

Delete the **inventory-service** project in OpenShift.

```
[student@workstation ~]$ oc delete project inventory-service
project "inventory-service" deleted
```

16.2. Delete the persistent volume for the database data.

Log in to OpenShift as the **admin** user and delete only the persistent volume that is in the **Released** state:

```
[student@workstation ~]$ oc login -u admin -p redhat \
https://master.lab.example.com
Login successful.

[student@workstation ~]$ oc get pv
NAME      CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS    ...
registry-volume  10Gi       RWX          Retain        Bound    ...
vol01     1Gi        RWO          Retain        Released ...
vol02      1Gi        RWO          Retain        Available ...
[student@workstation ~]$ oc delete pv vol01
persistentvolume "vol01" deleted
```

16.3. Clean the persistent volume NFS share.

Open an SSH connection to the **services** VM, as the **root** user, and remove the contents of the volume folder under **/var(exports**. Do not remove the folder.

```
[student@workstation ~]$ ssh root@services rm -rf /var(exports/vol01/*
```

Verify that the persistent volume NFS share is empty:

```
[student@workstation ~]$ ssh root@services ls -la /var(exports/vol01
total 0
drwxrwxrwx. 2 nfsnobody nfsnobody 6 Abr 9 18:23 .
drwxr-xr-x. 4 root      root      32 Mar 21 15:52 ..
```

16.4. Recreate the persistent volume.

Use one of the resource definition files in the **coolstore/inventory-service** folder:

```
[student@workstation ~]$ oc create -f ~/coolstore/inventory-service/vol01-pv.yaml
persistentvolume "vol01" created
```

Verify that the volume is in the **Available** state:

```
[student@workstation ~]$ oc get pv
NAME          CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS    ...
registry-volume  10Gi      RWX         Retain        Bound     ...
vol01        1Gi       RWO         Retain        Available ...
vol02          1Gi      RWO         Retain        Available ...
...
```

This concludes the guided exercise.

Summary

In this chapter, you learned:

- WildFly Swarm is a runtime for deploying modern Java-based, cloud-native microservices. Red Hat provides an enterprise distribution of WildFly Swarm as part of the Red Hat OpenShift Application Runtimes subscription.
- WildFly Swarm supports most of the Java EE APIs through the concept of fractions, which are unique components providing features on top of the WildFly Swarm core. Fractions are managed as Maven dependencies in your project. WildFly Swarm is rapidly adopting many APIs from the MicroProfile specification.
- The standard method to package WildFly Swarm applications is to build a fat JAR file containing the application as well as all its dependencies.
- WildFly Swarm can be configured in many ways: using command-line parameters, using YAML files embedded in the fat JAR, and using OpenShift configuration maps.
- You can provide your own custom health checks in WildFly Swarm applications. These health checks can be integrated with the OpenShift probes features to provide health monitoring of applications deployed in OpenShift.
- To deploy a WildFly Swarm microservice on OpenShift, use the Fabric8 Maven Plug-in (FMP) together with the WildFly Swarm Maven Plug-in.
- You can customize OpenShift resources by providing resource fragment files in your project source code tree. The FMP merges these resource fragments with defaults for each OpenShift resource type and creates the resources on OpenShift.

Chapter 3

Developing Microservices with the Vert.x Runtime

Goal

Develop and deploy a microservice using the Vert.x runtime.

Objectives

- Describe the Vert.x runtime and develop an application in Vert.x.
- Configure a Maven project to build a Vert.x application and deploy on OpenShift.

Sections

- Developing an Application with the Vert.x Runtime (and Guided Exercise)
- Configuring a Maven Project for Vert.x (and Guided Exercise)

Lab

Developing Microservices with the Vert.x Runtime

Developing an Application with the Vert.x Runtime

Objectives

After completing this section, students should be able to describe the Vert.x runtime and develop an application in Vert.x.

Describing Vert.x

The Vert.x open source project, maintained by the Eclipse Foundation and sponsored by Red Hat, provides a toolkit for building reactive applications in the Java VM. Vert.x is not tied to the Java programming language; it supports other JVM languages such as Groovy and Scala. Vert.x not only supports multiple programming languages, but provides idiomatic APIs for each of them.

Unlike some popular Java frameworks, such as Apache Struts or the Spring Framework, Vert.x does not impose a packaging and project structure onto an application. Vert.x provides a loosely coupled set of libraries that can be used to serve a wide variety of scenarios. High-concurrency, low-latency, and I/O-intensive applications make the best use of Vert.x features.

The core Vert.x library defines the fundamental APIs for writing asynchronous networked applications, and the remaining Vert.x libraries provide modules for database access, logging, authentication, and so on. You should avoid using most standard Java and Java EE APIs with Vert.x, because most of them are synchronous, blocking APIs. Fortunately, Vert.x provides modules that help solve most common and many advanced tasks, such as database access and REST APIs.

Its asynchronous, unblocking design allows Vert.x to deal with several concurrent network connections using less system resources, but with higher scalability compared to synchronous API designs such as the Java Servlets API and the **java.net** package.

Introducing Reactive and Asynchronous Execution Models

Vert.x provides tools to build *reactive systems*. Reactive systems are composed of components designed to observe a data stream and react to values from the data stream.

Components in a reactive system interact using asynchronous message passing. These interactions provide reactive applications with important properties:

- Responsiveness: reactive systems respond in an acceptable time, even under load.
- Elasticity: reactive systems scale horizontally by adding more instances of each component. Each component is scaled independently of other components.
- Resilience: reactive systems are able to handle and recover from failure in a given component, because other components are not blocked by the failed one. A reactive system can reroute messages to a different component or to another instance of the same component.

Reactive systems are a natural fit for functional programming, and a typical Vert.x application makes heavy use of Java 8 features such as lambda expressions and the streams API.

Vert.x APIs are event-driven. Vert.x runs an event loop that handles events as they come in, and calls event handler methods. If event handler methods perform their work quickly and not to

block, the event loop can process a large number of events in a short time, and avoid the context-switching overhead common on multithreaded designs.

To make use of multiple cores, Vert.x runs multiple event loops in parallel, by default two event loops per CPU core. This way, Vert.x makes efficient uses of multicore and hyperthreaded processors.

If there is no way to avoid either a long or blocking operation, Vert.x manages a working thread pool to execute those tasks outside the event loop. The working thread performs the operation and calls an event handler when the operation is completed, bringing the result back to the event loop.

Describing Vert.x Event Handler Methods

The basic building block of Vert.x asynchronous APIs are the **Handler**, **AsyncResult**, and **Future** interfaces.

The **io.vertx.core.Handler<E>** interface defines a generic event handler:

```
@FunctionalInterface
public interface Handler<E> {
    void handle(E event);
}
```

The **handler** method is a callback method invoked each time an event happens. The generic type **<E>** contains the event data and also the result of the asynchronous operation.

Most Vert.x APIs encapsulate the result of an asynchronous operation using the **io.vertx.core.AsyncResult<T>** interface. An **AsyncResult<T>** object allows the caller to: query whether the asynchronous operation was successful or if it failed, to get the result of a successful invocation, and to query the cause of an error. The generic type **<T>** from the **AsyncResult<T>** interface is the event data **<E>** from the **Handler<E>** interface.

The **io.vertx.core.Future<T>** interface extends the **AsyncResult<T>** interface to represent an operation that may or may not have occurred yet. It allows the caller to query whether the asynchronous operation has completed or not. It can invoke alternative event handlers in case of a failure.

The **Future<T>** interface also provides static helper methods to create **Future<T>** instances and populate them with result values and error causes.



Note

A Vert.x **Future<T>** is not the same as a Java 8 **java.util.concurrent.Future<V>**.

The **io.vertx.core.CompositeFuture** interface aggregates **Future<T>** objects to make multiple asynchronous calls easier to manage. A **CompositeFuture** object can wait for all of its **Future<T>** objects to complete, and allows the caller to query whether any of them failed.

Writing Vert.x Asynchronous Methods

Suppose you have a traditional synchronous method that takes an integer argument, and returns a String:

```
String myMethod(int arg) {
    String value;
    // ... perform some logic to determine value from arg...
    return value;
}
```

A synchronous call to that method looks like:

```
String value = obj.myMethod(123);
logger.info("value is " + value);
// ... do something with the returned value ...
```

The following listing shows how the previous example would be defined as a Vert.x asynchronous method:

```
void myMethod(int arg, Handler<AsyncResult<String>> resultHandler) {
    String value;
    // ... perform some logic to determine value from arg ...
    resultHandler.handle(Future.succeededFuture(value));
}
```

To call the asynchronous method, the caller needs to provide an event handler; in this example, a **Handler<AsyncResult<String>>** object. The easiest way to provide the event handler is by using a Java 8 lambda expression:

```
obj.myMethod(123, ar -> {
    String value = ar.result();
    logger.info("value is " + value);
    // ... do something with the returned value ...
});
```

The previous example assumes that the asynchronous method never fails, and unconditionally takes its return value. If an asynchronous method can fail, it should not throw an exception. Java's **try..catch** statement is not compatible with asynchronous calls.

The following listing shows how an asynchronous method would signal failure:

```
void myMethod(int arg, Handler<AsyncResult<String>> resultHandler) {
    String value;
    // ... perform some logic to determine value from arg ...
    if (errorFound)
        resultHandler.handle(Future.failedFuture("Error message"));
    else
        resultHandler.handle(Future.succeededFuture(value));
}
```

The following listing shows how the caller would detect and log the failure:

```
obj.myMethod(123, ar -> {
    if (ar.failed()) {
        logger.error("failed", ar.cause());
```

```

    }
    else {
        String value = ar.result();
        logger.info("value is " + value);
        // ... do something with the returned value ...
    }
});
```

Developing Verticles

Developing an entire application, or even a small microservice, using only asynchronous methods quickly leads to code that is very hard to understand and maintain. Vert.x provides two constructs that help organize an application into manageable pieces: *verticles* and the *event bus*.

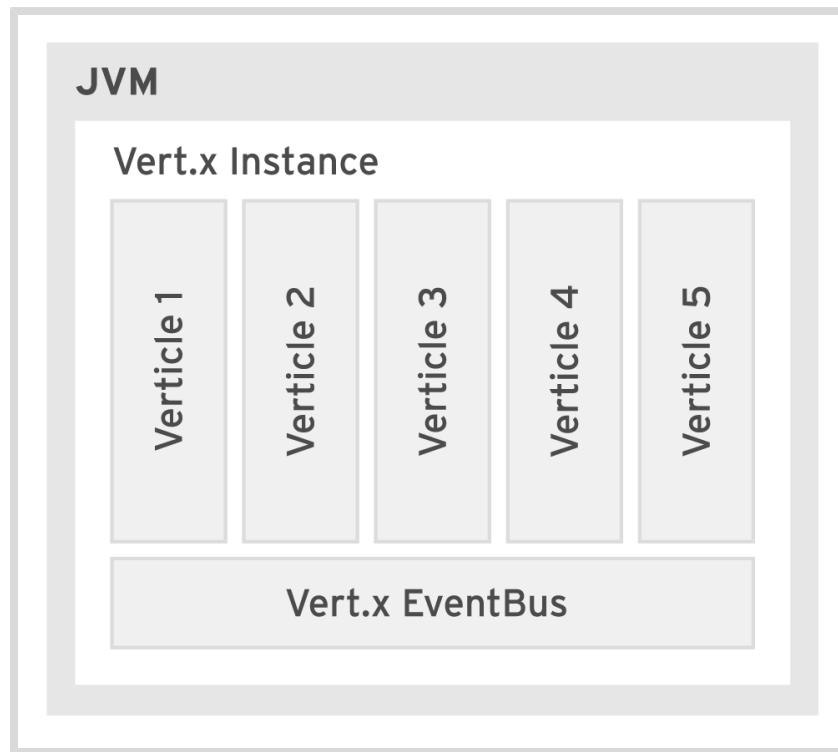


Figure 3.1: Verticles and the event bus

A verticle is the unit of deployment in Vert.x. A Vert.x application usually begins with a main verticle that deploys other verticles. Verticles communicate with other verticles using the event bus.

Some verticles just perform initialization tasks, such as initializing a database, then finish. Other verticles run until terminated by the operating system; for example, a verticle might accept requests from network clients, or a verticle could register a database access service to the event bus.

A verticle instance has a life cycle composed of start and stop operations, and always runs in the same thread to avoid the cost of synchronization primitives. You can deploy multiple instances of the same verticle to take advantage of multiple threads. Because verticles do not communicate directly, multiple verticles can share the same thread and the same event loop.

All verticles inside the same Java VM runs under an **io.vertx.core.Vertx** instance, which provides access to features such as network client and servers, shared configuration data, and the event bus.

The easiest way to implement a verticle is to extend the **io.vertx.core.AbstractVerticle** class:

```
public class MyVerticle extends AbstractVerticle {

    @Override
    public void start() throws Exception {
        // register hanlders for events of interest to the verticle
    }

    @Override
    public void stop() throws Exception {
        // clean up when the verticle is to be undeployed
    }
}
```

The **start** and **stop** methods have versions that accept **Future<T>** arguments. These versions allow a verticle to signal any failures during its initialization or shutdown.

To start another verticle, call the **Vertx.deploy** method. Use either the **vertx** attribute or the **getVertx()** method of the **.AbstractVerticle** class to get the current **Vertx** instance:

```
vertx.deployVerticle(new MyOtherVerticle());
```

The **Vertx.deploy** method has versions that accepts argument such as configuration data and event handlers. Using event handlers allows a verticle to deploy multiple other verticles in parallel, and wait until all of them are started before accepting events.

Calling Blocking Code from a Verticle

If a verticle needs to perform a long-running or potentially blocking operation, it should invoke the **Vertx.executeBlocking** method, which takes two handlers as arguments:

- The first handler encapsulates the long-running or blocking operation. The handler is submitted to a worker thread so that it does not keep the verticle's event loop busy.
- The second handler gets the result from the first handler. It runs on the current verticle's event loop.

Connecting Services to the Event Bus

The Vert.x event bus allow a verticle to exchange messages with other verticles. The event bus provides an API similar to other popular messaging APIs, such as the Java Messaging Service (JMS).

Messages are sent to destinations identified by an address which can be any string. Messages have headers and a body that can use any data format, though JSON is the most common, using the **io.vertx.core.json.JsonObject** class. Usual messaging patterns, such as point-to-point and publish/subscribe are also supported.

The Vert.x Clustering module extends the event bus to support communication between verticles running in different Java VMs. Other Vert.x modules extend the event bus to non-Vert.x applications, including non-Java applications, such as JavaScript code running in a web browser, or any platform that can use standard messaging protocols, such as the *Advanced Message Queue Protocol* (AMQP).

One way to use the event bus is to invoke the **Vertx.eventBus** method to get a reference to the **EventBus** object of the current Vert.x instance. An easier way is to use event bus services.

Vert.x event bus services encapsulates sending and receiving messages using the event bus as regular Vert.x asynchronous calls.

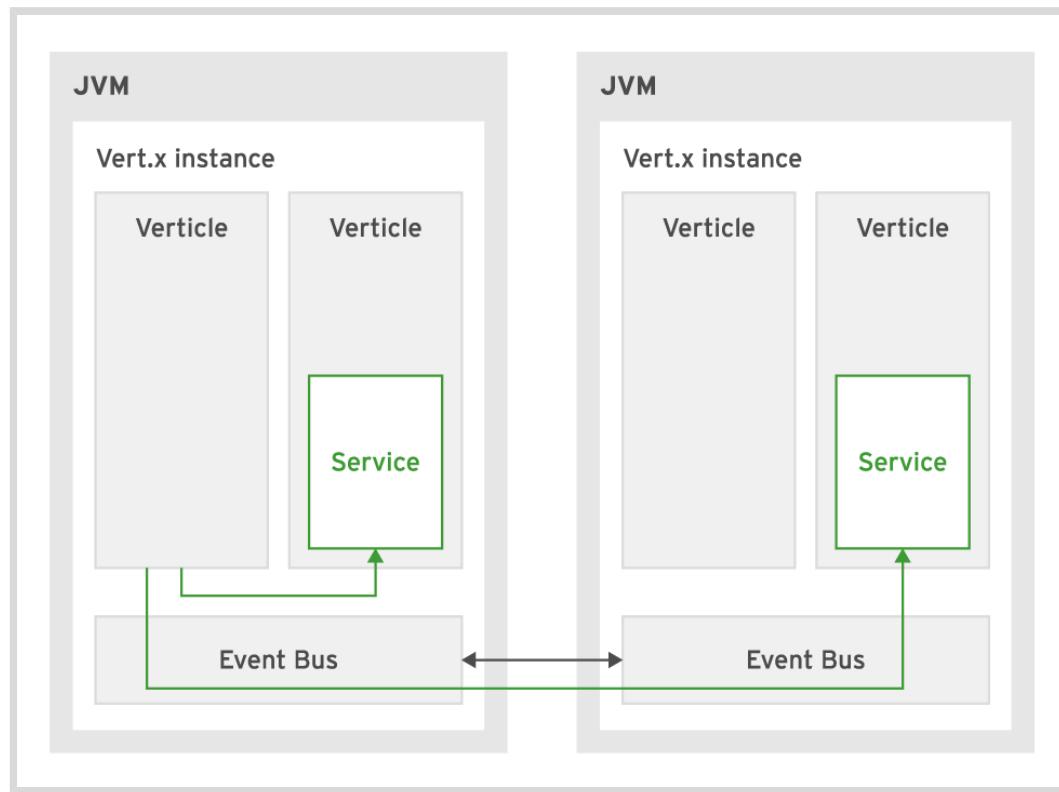


Figure 3.2: Event bus services

Writing an Event Bus Service

A Vert.x event bus service is composed of:

- A Java interface that describes the messages accepted by the service. Each method of the interface represents one kind of message.
- A Java class that implements the service interface and is registered to the event bus to receive messages.
- A proxy class that is generated by the Vert.x code generation module and marshals event data into event bus messages, then sends the messages on behalf of the caller.

An event bus service interface is annotated with the **io.vertx.codegen.annotations.ProxyGen** annotation:

```
@ProxyGen
public interface MyService {
    void myMethod(int arg, Handler<AsyncResult<String>> resultHandler);
}
```

All packages that define a **@ProxyGen** interface are annotated with the **io.vertx.codegen.annotations.ModuleGen** annotation. Create a **package-info.java** file inside the package folder with the annotation:

```
@io.vertx.codegen.annotations.ModuleGen(
    groupPackage="com.example.myservice.mypackage",
    name="myservice")
package com.example.myservice.mypackage;
```

If the service interface method signatures use any Java class beyond **String** and primitive wrappers, such as **Integer** and **Boolean**, each of these classes needs to be annotated with the **io.vertx.codegen.annotations.DataObject** annotation, for example:

```
@DataObject
public interface MyDataType {
    // ... getters and setters ...
}
```

All packages that define a **@DataObject** class also need to be annotated with **@ModuleGen**.

The implementation class of an event bus service is just a regular Java class:

```
public class MyServiceImpl implements MyService
{
    void myMethod(int arg, Handler<AsyncResult<String>> resultHandler) {
        // ... do something with arg and resultHandler ...
    }
}
```

Registering and Invoking an Event Bus Service

Using an event bus service involves two actions, usually performed by different verticles:

- Register a service implementation class to the event bus, associated with a destination address.
- Create an instance of the generated proxy class, associated with the same destination address.

To register a service implementation with the event bus, a verticle calls the **ProxyHelper.registerService** method with the following arguments:

- The service interface class object
- The Vert.x instance
- An instance of the implementation class
- The destination address to receive the messages

Continuing with the previous example, to register the **MyServiceImpl** class to receive messages from the **MyServiceAddress** address, a verticle calls:

```
ProxyHelper.registerService(MyService.class, vertx, new MyServiceImpl(),
    "MyServiceAddress");
```

The event bus service runs as part of the event bus of the verticle that registers the service.

A verticle invokes an event bus service using the generated proxy class, which is **ServiceInterfaceNameVertxEBProxy**. For example, to invoke the **MyService** event bus service and send messages the **MyServiceAddress** address, a verticle calls:

```
MyService proxy = new MyServiceVertxEBProxy(vertx, "MyServiceAddress");
proxy.myMethod(123, ar -> {
    //... do something with the AsyncResult ar ...
});
```

It is a common practice in the Vert.x community to define, in a event bus service interface, static methods to create both the service implementation class and the generated proxy class.

Unlike other popular Java frameworks, such as Hibernate and Context and Dependency Injection (CDI), Vert.x does not employ byte code manipulation techniques. Vert.x relies instead on static code generation, with the help of Maven and Java annotation processors.

Implementing HTTP APIs with the Web Module

The Vert.x Core Module provides the **io.vertx.core.http.HttpServer**, which provides low-level HTTP and WebSocket protocol APIs.

To create an **HttpServer** object, call the **Vertx.createHttpServer** method. Once you have an **HttpServer** object, call the **HttpServer.listen** method to accept HTTP requests.

The **listen** method has many overloaded versions, for example one that accepts a handler to signal success or failure to start the TCP server socket:

```
public void start(Future<Void> future) {
    HttpServer server = vertx.createHttpServer();
    // ... set a handler to process HTTP requests ...
    server.listen(8080, result -> {
        if (result.succeeded()) {
            logger.info("Started HTTP server");
            future.complete();
        } else {
            logger.error("Cannot start the HTTP server: " + result.cause());
            future.fail(result.cause());
        }
    });
}
```

Most Vert.x applications also use the Vert.x Web Module, which is inspired by the Node.js Express and the Ruby Sinatra frameworks. The Web Module offers features to deal with sessions, cross-origin resource sharing (CORS), authentication based on JSON Web Tokens (JWT), and other requirements of modern HTTP APIs.

The main components of the Vert.x Web Module are the **io.vertx.ext.web.Router** and the **io.vertx.ext.web.Route** interfaces.

A **Router** object describes an HTTP API composed of multiple endpoints, or *routes*, and a **Route** object describes a single end point.

To create a **Router** object, call the **router** static factory method:

```
Router router = Router.router(vertx);
```

One way to create a **Route** object is to call one of the **Router** methods whose name resembles an HTTP method name, for example: **get** and **post**, and then pass the resource URL. A colon (**:**) in the resource URL denotes a path element that can take any value.

A **Route** object accepts multiple event handlers. Each handler method receives an **io.vertx.ext.web.RoutingContext** argument, used to get the HTTP request data such as cookies and headers, and to set the HTTP response data.

The last handler method should call the **RoutingContext.end()** method to signal that the HTTP response is completed and can be returned to the client.

The following example creates a **Route** for the **/order/:orderId** resource URL, with one handler that returns the received **orderId**, using the HTTP GET method,

```
router.get("/order/:orderId").handler(rc -> {
    String id = rc.request().getParam("orderId");
    rc.response().end("Order: " + id);
});
```

Because a **Route** handler is usually a larger piece of code, it is common practice to define it as a method reference instead of using a lambda expression, for example:

```
public void start() throws Exception {
    ...
    router.get("/order/:orderId").handler(this::getOrder)
    ...
}

private void getOrder(RoutingContext rc) {
    String id = rc.request().getParam("orderId");
    rc.response().end("Order: " + id);
}
```

The following example illustrates a common scenario using two handlers for the same route: the first handler uses the **io.vertx.ext.web.handler.BodyHandler** object to retrieve the HTTP POST request body and store its data on the **RoutingContext**; and the second handler uses that data to perform some business logic.

```
router.post("/order")
    .handler(BodyHandler.create())
    .handler(this::acceptNewOrder);
```

When all routes are set up, set the **Router.accept** method as the request handler for the **HttpServer** object:

```
Router router = Router.router(vertx);
//... set routes ...
HttpServer server = vertx.createHttpServer();
server.handler(router::accept);
server.listen();
```

Because most methods of the **HttpServer** interface are designed to return the **HttpServer** instance, its methods can be concatenated in a fluent coding style:

```
Router router = Router.router(vertx);
//... set routes ...
vertx.createHttpServer()
    .handler(router::accept)
    .listen();
```

The fluent coding style was also used on the previous examples of setting up routes.

Invoking HTTP APIs with Vert.x

The Vert.x Core Module provides the **io.vertx.core.http.HttpClient** interface, which provides methods such as **get** and **post** to describe and submit HTTP requests. These methods take the host name, port, and request URL as discrete arguments, and return an **io.vertx.core.http.HttpClientRequest** object.

To submit the HTTP request call the **HttpClientRequest.end** method.

To process the HTTP response, you require two handlers:

- A handler for the HTTP request, which takes an **io.vertx.core.http.HttpClientResponse** object. The **HttpClientResponse** object provides access to the HTTP response data, such as headers and cookies.
- A handler for the HTTP request response body, which takes an **io.vertx.core.buffer.Buffer** object. The **Buffer** object provides methods to stream and decode the HTTP response data.

To set up the handler for the response body, call the **HttpClientResponse.bodyHandler** method.

The following example creates an **HttpClient** object, using the **Vertx.createHttpClient** method, then submits an HTTP GET request to `http://localhost:8080/order/123`:

```
vertx.createHttpClient().get(8080, "localhost", "/order/123", response -> {
    response.bodyHandler(body -> {
        logger.log("got: " + body.toString());
    });
}).end();
```

Running Vert.x Applications

A Vert.x application usually does not create its **Vertx** instance, and does not deploy its first verticle: these operations are handled by the Vert.x runtime.

If you install a Vert.x distribution, you have access to the **vertx** command. The **vertx** command provides a few convenience operations for developer, for example the **run** verb. It initializes the Java VM class path, initializes Vert.x runtime, and deploys the startup verticle given as an argument:

```
$ vertx run com.example.MyVerticle
```

The **vertx run** command offers command-line switches to define configuration data, clustering behavior, and other runtime properties for the Vert.x runtime.

If you rely on Maven to download Vert.x libraries, and never install a Vert.x distribution, you can rely on the *Vert.x Maven Plug-in* to package your application as a fat JAR and define the Vert.x runtime properties.

The Vert.x Maven Plug-in also sets up a main class that starts the Vert.x runtime and deploys a startup verticle for your application. The generated JAR file can be executed directly with the **java -jar** command.

Later in this book you will learn how to configure a Maven project POM to use the Vert.x Maven Plug-in.

Testing Vert.x Applications

The Vert.x Unit module provides a number of useful helpers to write unit tests for Vert.x asynchronous calls, and also integration with the JUnit test framework.

JUnit tests leverage the Vert.x Unit module by using the **io.vertx.ext.unit.junit.VertxUnitRunner** test runner and adding a **io.vertx.ext.unit.TestContext** argument to each test method:

```
@RunWith(VertxUnitRunner.class)
public class MyVerticleTest {

    @Test
    public void testSomething(TestContext context) throws Exception {
        // ... test logic ...
    }
}
```

The test class needs to perform tasks such as initializing the Vert.x runtime, providing Vert.x configurations, deploying the verticles under test, and registering event bus services. It also needs to register an exception handler with the current **TestContext** object:

```
private Vertx vertx;

@Before
public void setUp(TestContext context) throws Exception {
    vertx = Vertx.vertx();
    vertx.exceptionHandler(context.exceptionHandler());
    vertx.deployVerticle(new MyVerticle(), new DeploymentOptions(),
        context.asyncAssertSuccess());
}

@After
public void tearDown(TestContext context) throws IOException {
    vertx.close(context.asyncAssertSuccess());
}
```

Most test methods should set up an execution timeout, and use an **io.vertx.ext.unit.Async** object to signal all asynchronous calls from the test run are completed:

```
@Test(timeout=3000)
public void testSomething(TestContext context) {
    Async async = context.async();
```

```

        makeAsyncCall(args, result -> {
            // ... asserts ...
            async.complete();
        });
    }
}

```

The **TestContext** interface also provides a number of convenience assert methods. You are not required to use them because the Vert.x Unit module is compatible with popular assertion libraries such as Hamcrest and RestAssured.

Implementing Health Probes with the Vert.x Health Check Module

Red Hat recommends that applications targeting OpenShift define liveness and readiness health probes. These are typically implemented as HTTP API endpoints, for example:

```

router.get("/readiness").handler(rc -> {
    if (myService.isInitialized())
        rc.response().end("OK");
    else
        rc.fail(500);
});

```

The Vert.x Health Check module provides the **io.vertx.ext.healthchecks.HealthCheckHandler** interface. A **HealthCheckHandler** can potentially aggregate a number of health check procedures into its JSON response. Each health check procedure returns an instance of **io.vertx.ext.healthchecks.Status**.

After you register the health check procedures to a **HealthCheckHandler** object, assign this object as the handler for a route, for example:

```

HealthCheckHandler healthHandler = HealthCheckHandler.create(vertx)
healthHandler.register("Database", this::verifyDatabase());
healthHandler.register("Cache", this::verifyCache());
router.get("/liveness").handler(healthHandler);

...

private void verifyDatabase(Future<Status> status) {
    databaseService.testQuery(ar -> {
        if (ar.succeeded())
            future.complete(Status.OK());
        else
            future.complete(Status.KO());
    });
}

private void verifyCache(Future<Status> status) {
...
}

```

Configuring a Vert.x Application

The Vert.x runtime manages its configurations as JSON data, stored in a Vert.x **JsonObject** instance. This configuration is used by the current **Vertx** instance and other Vert.x components, such as verticles, the event bus, and the **HttpServer**, to provide runtime parameters such as clustering behavior and a TCP port to listen for network connections.

Most Vert.x components provide helper classes, such as **io.vertx.core.DeploymentOptions** and **io.vertx.core.http.HttpServerOptions**, that encapsulated the configuration data for easier access, and provide default configuration values.

All Vert.x components are supposed to ignore any configuration data they do not understand, so the configuration data for all components can be managed by the **Vertx** instance.

The **vertx** command and the Vert.x Maven Plug-in take an optional text file containing JSON configuration data. The Vert.x Config module adds support for alternative sources and configuration data formats, which are retrieved, converted to JSON data, and stored in the current **Vertx** instance.

The Vert.x Config module is composed of two main components:

- A configuration store that defines a location and format for configuration data.
- A configuration retriever that uses a set of configuration stores to fetch, and consolidate the configuration data.

You do not create configuration stores directly. You describe a set of configuration stores using one or more **io.vertx.config.ConfigStoreOptions** objects and add them to an **io.vertx.config.ConfigRetrieverOptions** object.

Then you create a **io.vertx.config.ConfigRetriever** object by calling the **ConfigRetriever.create** static method and passing the **ConfigRetrieverOptions** object as an argument.

Finally, you call the **ConfigRetriever.getConfig** asynchronous method to retrieve the configuration data and store it in the current **Vertx** instance. It is advisable to call **getConfig** before deploying any verticle and registering any service to the event bus.

The following example sets up a **ConfigRetriever** object to fetch configuration data from an OpenShift configuration map:

```
ConfigStoreOptions configStore = new ConfigStoreOptions();
configStore.setType("configmap") ①
    .setFormat("yaml") ②
    .setConfig(new JsonObject()
        .put("name", "myconfig") ③
        .put("key", "config.yaml") ④
    );
ConfigRetrieverOptions retrieverOptions = new ConfigRetrieverOptions();
retrieverOptions.addStore(configStore); ⑤
ConfigRetriever retriever = ConfigRetriever.create(vertx, retrieverOptions); ⑥
retriever.getConfig(ar -> { ⑦
    // ... do something such as deploy verticles ...
});
```

- 1 Sets the configuration store type to configuration map.
- 2 Sets the configuration data format to YAML.
- 3 Sets the configuration map resource name.
- 4 Sets the configuration map key that stores the configuration data.
- 5 Adds the configuration store to a configuration retriever.
- 6 Creates the configuration retriever object.
- 7 Reads the configuration data.

A **ConfigRetriever** object also allows an application to react to configuration changes.

Depending on the duration and complexity of your application initialization procedure, it may be more efficient to restart the application than to react to configuration changes. This is specially true with containerized microservices running under OpenShift, where you can rely on deployment configuration features.



References

A Gentle Guide to Asynchronous Programming with Eclipse Vert.x for Java developers

<https://vertx.io/docs/guide-for-java-devs/>

Building Reactive Microservices in Java

<https://developers.redhat.com/promotions/building-reactive-microservices-in-java/>

The Vert.x community web site

<https://vertx.io>

The Reactive Manifesto

<https://www.reactivemanifesto.org>

Lambda Expressions

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

► Guided Exercise

Developing an Application with the Vert.x Runtime

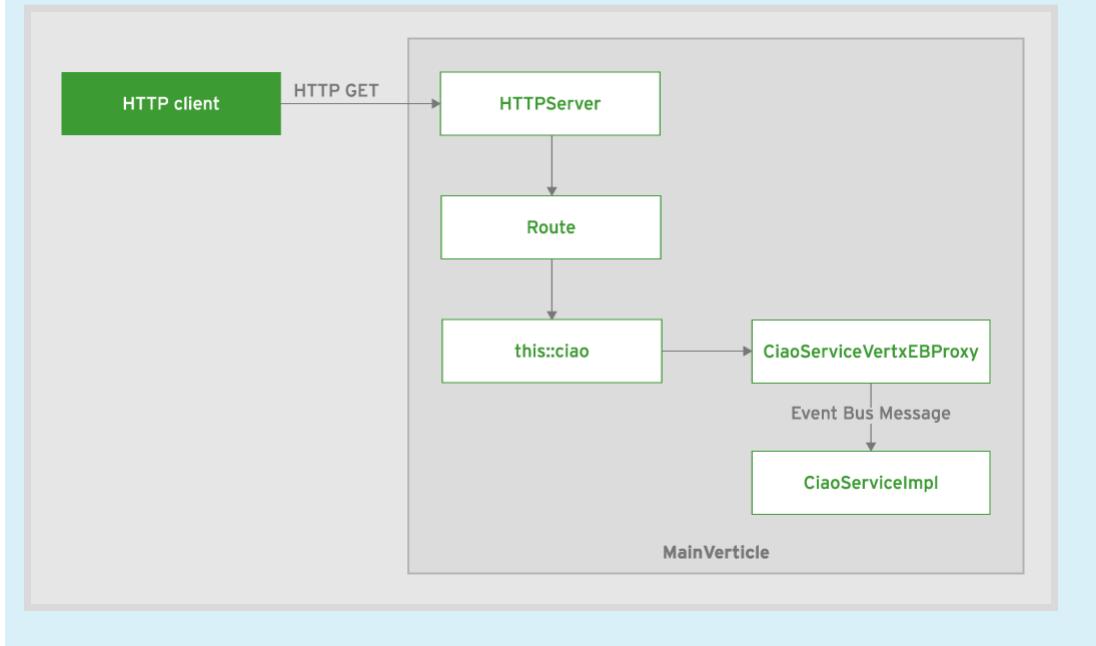
In this exercise, you will develop a simple microservice using Vert.x.

Outcomes

You should be able to:

- Complete an asynchronous method using Vert.x handlers.
- Connect a service to the Vert.x event bus.
- Implement an HTTP API that delegates to the event bus service.

The following diagram illustrates the design of the exercise. Vert.x instantiates the **MainVerticle**, and an HTTP server on start up and registers the routes declared in the **Router** object. Once a request is received for the **/api/ciao** endpoint, it handles the request by forwarding it to the **ciao** method of the **CiaoService**.



To perform this exercise, you need access to:

- The **do292-ciao-microservice-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **hello-microservice/ciao** Maven project.
- The **io.vertx:vertx-dependencies**, **io.fabric8:fabric8-maven-plugin**, and **io.fabric8:vertx-maven-plugin** Maven artifacts on the classroom Nexus proxy server.

If you need help to use Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab ciao-microservice setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-ciao-microservice-solution** of the **hello-microservices** Git repository.

- 1. Switch to the **do292-ciao-microservice-begin** branch in the **hello-microservices** Git repository and import the **ciao** project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-ciao-microservice-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
  do292-ciao-microservice-begin
...
Switched to a new branch 'do292-ciao-microservice-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/hello-microservices/ciao** folder as a Maven project.
- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-ciao-microservice-begin**.

- 2. Review the initial state of the **ciao** project.

The project is composed of three source files. During this exercise, you will complete the two files that define Java classes, and make no change to the file that defines a Java interface.

MainVerticle.java

Implements the microservice's REST API and invokes the event bus service.

CiaoService.java

Interface of the event bus service.

CiaoServiceImpl.java

Implementation of the event bus service.

The project provides three unit test source files. During this exercise you will make no change to any of the unit tests, except to uncomment a few lines:

MainVerticleTest.java

Test the microservice's REST API by making HTTP requests and verifying the response text.

CiaoServiceEBProxyTest.java

Test the event bus service, using the event bus to dispatch service calls.

CiaoServiceTest.java

Test the event bus service implementation making direct asynchronous calls.

▶ **3.** Inspect the project's Maven POM file.

Open the **pom.xml** file from the **ciao** project. Switch to the **pom.xml** tab.

You will not change the project POM during this exercise. For now, note the reference to a parent POM file, which provides values for system properties. These properties are used to define the version strings for Maven artifacts required to build the microservice.

Note the use of the **io.vertx:vertx-dependencies** imported POM to list supported dependencies for the Vert.x framework distribution provided by Red Hat OpenShift Application Runtimes. Note also the Maven plug-ins configuration. These will be explained in the next section.

▶ **4.** Complete the asynchronous method in the **CiaoServiceImpl** class.4.1. Inspect the unit tests for the **CiaoServiceImpl** class.

Open the **CiaoServiceTest** class from the **com.redhat.training.msa.ciao.service** package.

The **ciaoImplTest** method invokes the **CiaoServiceImpl.ciao** method, using the **CiaoService** interface, and asserts the returned string includes the expected words.

4.2. Run the **CiaoServiceTest** class as a JUnit test.

The **JUnit** view shows that the test run starts, and after a few seconds the test run stops with a timeout.

4.3. Inspect the incomplete **CiaoServiceImpl** class.

Open the **CiaoServiceImpl** class from the **com.redhat.training.msa.ciao.service** package.

Note that the **ciao** method builds its return value in the **msg** local variable. Do not change this line.

The incomplete **ciao** method does not return the value in the **msg** local variable to the caller.

4.4. Complete the asynchronous **ciao** method to return its value using the **Handler** argument as a callback.

Invoke the callback method, using the **Handler** argument, and pass a **Future** object to signal success. Pass the result string **msg** to the **Future.succeededFuture** method.

Use the following listing as a reference for the changes to the asynchronous method:

```

    ...
    public void ciao(String host, String nome,
        Handler<AsyncResult<String>> resultHandler) {
        String msg = "Ciao " + nome + ", da " + host +"\n";
        resultHandler.handle(Future.succeededFuture(msg));
    }
    ...

```

- 4.5. Run the **CiaoServiceTest** class as a JUnit test.

The **JUnit** view shows that the test run pass. If not, review your changes during the previous steps, make fixes, and try again.

▶ 5. Connect the **CiaoServiceImpl** class to the event bus

- 5.1. Inspect the unit tests from the **CiaoServiceEBProxyTest** class.

Open the **CiaoServiceEBProxyTest** class from the **com.redhat.training.msa.ciao.service** package and inspect the **ciaoProxyTest** method.

The **ciaoProxyTest** method invokes the **CiaoService.ciao** method, using an event bus proxy, and asserts the returned string includes the expected words. The code that registers the **CiaoServiceImpl** to the event bus is commented out to avoid build errors. The code references the **CiaoServiceEBProxy** event bus proxy class that you need to generate.

- 5.2. Annotate the **CiaoService** interface for the event bus.

Open the **CiaoService** class from the **com.redhat.training.msa.ciao.service** package.

Add the **@ProxyGen** annotation to the interface. If you press **Ctrl+Space** to trigger the IDE auto-completion feature, it adds the correct **import** statement to the source code.

Use the listing that follows as a reference for changing the interface:

```

...
import io.vertx.codegen.annotations.ProxyGen;
...
@ProxyGen
public interface CiaoService {
...

```

- 5.3. Annotate the **com.redhat.training.msa.ciao.service** package for code generation.

Create the **package-info.java** file in the **com.redhat.training.msa.ciao.service** package.

Add the **package** declaration and the **@ModuleGen** annotation. You can rely on the IDE auto-completion feature to reduce typing.

The **@ModuleGen** annotation requires two arguments. Pass the complete package name to the **groupPackage** argument, and **ciao-service** to the **name** argument.

Use the following listing as a reference for the complete contents of the **package-info.java** file:

```

@io.vertx.codegen.annotations.ModuleGen(
    groupPackage="com.redhat.training.msa.ciao.service",
    name="ciao-service")
package com.redhat.training.msa.ciao.service;

```

- 5.4. Run the Vert.x code generation plug-in.

Run a Maven build with the **clean package -DskipTests** arguments.

The **Console** view shows that Maven invoked the Vert.x code generator as part of the build:

```

...
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ catalog-service
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 4 source files to /home/student/hello-microservices/ciao/target/
classes
...
INFO: Loaded service_proxies code generator
Apr 16, 2018 6:50:05 PM io.vertx.codegen.CodeGenProcessor lambda$process$6
INFO: Generated model com.redhat.training.msa.ciao.service.CiaoService:
  com.redhat.training.msa.ciao.service.CiaoServiceVertxProxyHandler
Apr 16, 2018 6:50:05 PM io.vertx.codegen.CodeGenProcessor lambda$process$6
INFO: Generated model com.redhat.training.msa.ciao.service.CiaoService:
  com.redhat.training.msa.ciao.service.CiaoServiceVertxEBProxy...

```

- 5.5. Configure the project build path.

Refresh the project in the IDE and add the **/src/main/generated** folder to the project's build path. Refer to the *Adding a Source Folder to the Build Path* section in *Appendix B: Working with Red Hat JBoss Developer Studio* for detailed steps.

Wait while the IDE rebuilds the project.

- 5.6. Uncomment the code in the **CiaoServiceEBProxyTest** class that instantiates the event bus proxy.

Open the **CiaoServiceEBProxyTest** class from the **com.redhat.training.msa.ciao.service** package.

Inside the **ciaoProxyTest** method, uncomment the three lines after the declaration of the **async** local variable, and delete the line that initializes the **proxy** variable with a **null** value.

Use the listing that follows as a reference for the changes to the **ciaoProxyTest** method:

```

...
@Test(timeout=3000)
public void ciaoProxyTest(TestContext testContext) {
    final Async async = testContext.async();

    CiaoService serviceImpl = new CiaoServiceImpl();
    ProxyHelper.registerService(CiaoService.class, vertx, serviceImpl,
        ADDRESS);
    CiaoService proxy = new CiaoServiceVertxEBProxy(vertx, ADDRESS);

```

```
final String host = "corleone.example.com";
...
```

Ignore warnings about the **ProxyHelper** class being deprecated.

- 5.7. Run the **CiaoServiceEBProxyTest** class as a JUnit test.

The **JUnit** view shows that the test pass. If not, review your changes during the previous steps, make fixes, and try again.

▶ 6. Connect the verticle to the event bus service.

The same code from the **CiaoServiceEBProxyTest** class that connects the **CiaoServiceImpl** class to the event bus must be added to the **MainVerticle** class.

- 6.1. Inspect the incomplete **MainVerticle** class.

Open the **MainVerticle** class from the **com.redhat.training.msa.ciao** package.

The **start** method invokes the **registerCiaoService** method to connect the **CiaoServiceImpl** class to the event bus. Scroll down the **MainVerticle.java** file, to the **registerCiaoService** method, and note it is empty.

- 6.2. Complete the **registerCiaoService** method.

Create an instance of the **CiaoServiceImpl** class and register that instance to the event bus. Then create an instance of the **CiaoServiceEBProxy** class and save that instance in the **ebProxy** variable.

Use the following listing as a guide to the completed **registerCiaoService** method:

```
...
import com.redhat.training.msa.ciao.service.CiaoServiceVertxEBProxy;
...

private void registerCiaoService() {
    LOG.info("Registering ciao service to the event bus...");

    CiaoService serviceImpl = new CiaoServiceImpl();
    ProxyHelper.registerService(CiaoService.class, vertx, serviceImpl,
        ADDRESS);
    ebProxy = new CiaoServiceVertxEBProxy(vertx, ADDRESS);
}
```

- 6.3. Inspect the incomplete **ciao** method.

The **ciao** method is a handler for the Vert.x **Router** class and already contains the code to fetch the arguments to invoke the **CiaoService.ciao** method. You need to add the call, using the event bus service.

- 6.4. Complete the **ciao** method.

Use the **ebProxy** variable to call the **CiaoService.ciao** method. Also pass to the service call a handler that takes the resulting String and uses it as the **RoutingContext** response.

Use the following as a guide to the completed **ciao** method:

```

...
    private void ciao(RoutingContext rc) {
        String nome = rc.request().getParam("nome");
        String host = rc.request().host();

        LOG.info("Got API request for nome = '" + nome + "' ...");

        ebProxy.ciao(host, nome, ar -> {
            rc.response().end(ar.result());
        });
    }
...

```

Do not close the file the **MainVerticle** class. You will make more changes to it during the next step.

▶ 7. Implement the microservice's REST API.

7.1. Add a route to the microservice API entry point.

Find the **start** method. Right after the line that creates the **Router** object, add a route using the HTTP GET request method to produce plain text output, and invoke the **ciao** handler method.

Use the following as a guide to the changes to the **start** method:

```

...
public void start(Future<Void> future) {

    LOG.info("Welcome to Vertx. Starting Ciao service...");

    registerCiaoService();

    Router router = Router.router(vertx);
    router.get("/api/ciao/:nome")
        .produces("application/text")
        .handler(this::ciao);

    vertx.createHttpServer()
...

```

7.2. Configure the Router object as the request handler of the **HttpServer** object.

inside the **start** method, right after the call to the **createHttpServer** method, add a call to **requestHandler**, passing the **Router** object **accept** method as argument.

Use the following listing as a guide for the change to the **start** method:

```

...
public void start(Future<Void> future) {

    ...

    vertx.createHttpServer()

```

```

    .requestHandler(router::accept)
    .listen(8080, result -> {
        if (result.succeeded()) {
            future.complete();
        } else {
            future.fail(result.cause());
        }
    });
...

```

7.3. Inspect the test for the **MainVerticle** class.

Open the **MainVerticleTest** class from the **com.redhat.training.msa.ciao** package.

The **ciaoTest** method uses the Vert.x **HttpClient** object to send an HTTP GET request to **/api/ciao/Luigi**, and asserts that the plain test data returned includes the expected substrings.

7.4. Run the **MainVerticleTest** class as a JUnit test.

The **JUnit** view shows that the test passes. If not, review your changes during the previous steps, make fixes, and try again.

► 8. Test the microservice from the command line.

8.1. Package the microservice as a Fat JAR.

Open a terminal window and enter the **~/hello-microservice/ciao** folder. Then use the Maven package goal to generate the Fat JAR file. Skip the tests for faster execution of the **mvn** command:

```

[student@workstation ~]$ cd ~/hello-microservices/ciao
[student@workstation ciao]$ mvn clean package -DskipTests
...
[INFO] BUILD SUCCESS
...

```

8.2. Start the microservice.

Run the microservice using the **java** command and the Fat JAR file:

```

[student@workstation ciao]$ java -jar target/ciao-1.0.jar
abr 16, 2018 8:04:01 PM com.redhat.training.msa.ciao.MainVerticle start
INFO: Welcome to Vertx. Starting Ciao service...
abr 16, 2018 8:04:01 PM com.redhat.training.msa.ciao.MainVerticle
registerCiaoService
INFO: Registering ciao service to the event bus...
abr 16, 2018 8:04:01 PM io.netty.util.internal.logging.Slf4JLogger warn
WARNING: Failed to find a usable hardware address from the network interfaces;
using random bytes: ab:f3:23:7c:4c:70:53:df
abr 16, 2018 8:04:01 PM io.vertx.core.impl.launcher.commands.VertxIsolatedDeployer
INFO: Succeeded in deploying verticle

```

8.3. Send an HTTP request to the microservice.

Open another terminal window, and use the curl command to send a request to the microservice. Use **localhost:8080** as the host name and port, and the **/api/ciao/Fredo** resource URL:

```
[student@workstation ciao]$ curl -si http://localhost:8080/api/ciao/Fredo
HTTP/1.1 200 OK
Content-Length: 30

Ciao Fredo, da localhost:8080
```

Switch to the terminal running the microservice and press **Ctrl+C**

► **9.** Clean up.

- 9.1. Commit your changes to your local Git repository:

```
[student@workstation ciao]$ git add .
[student@workstation ciao]$ git commit -m \
"Finished the exercise."
```

- 9.2. Remove the **ciao** project from the IDE workspace.

- 9.3. If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation coolstore]$ git reset --hard \
origin/do292-ciao-microservice-begin
```

This concludes the guided exercise.

Configuring a Maven Project for Vert.x

Objectives

After completing this section, students should be able to configure a Maven project to build a Vert.x application and deploy on OpenShift.

Using Vert.x Dependencies from Red Hat OpenShift Application Runtimes

Red Hat provides, as part of the Red Hat OpenShift Application Runtimes (RHOAR) subscription, an enterprise distribution of the Vert.x runtime. To receive product support, Vert.x developers should download the Maven artifacts from RHOAR using the JBoss Enterprise Maven Repository.

The set of Vert.x libraries that are supported and tested with RHOAR may not include all upstream Vert.x modules. Refer to the *Release Notes* for your RHOAR release to get a list of these libraries, and their versions.

At the time this book was written, the latest supported release of Vert.x provided by RHOAR is: **3.5.1.redhat-003**.

The Vert.x Maven Plug-ins and the Fabric8 Maven Plug-in are not included in the Vert.x distribution from RHOAR. A developer must download these from the standard community Maven repositories. Refer to the *Release Notes* for your RHOAR release to get the versions of those plug-ins that were tested for compatibility with RHOAR.

To configure a Maven project to use Vert.x from RHOAR, first define system properties for the component releases in your project's POM:

```
<properties>
  <version.fabric8.plugin>3.5.38</version.fabric8.plugin>
  <version.vertx>3.5.1.redhat-003</version.vertx>
  <version.vertx-maven-plugin>1.0.7</version.vertx-maven-plugin>
  <!-- ... other system properties .. -->
</properties>
```

Then include the Vert.x distribution bill-of-materials:

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

And finally, include the Vert.x modules required by your application, for example:

```
<dependencies>
  <dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-core</artifactId>
  </dependency>
  <dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-web</artifactId>
  </dependency>
  <!-- ... other dependencies ... -->
</dependencies>
```

Configuring the Vert.x Maven Plug-in

The Vert.x Maven Plug-in packages a Vert.x application as a Fat JAR that embeds all dependencies and requires only a Java VM to run. To enable this plug-in, add the **io.fabric8:vertx-maven-plugin** Maven Artifact to the **<build>** section of your project's POM:

```
<build>
  <plugins>
    <plugin>
      <groupId>io.fabric8</groupId>
      <artifactId>vertx-maven-plugin</artifactId>
      <version>${version.vertx-maven-plugin}</version>
      <executions>
        <execution>
          <id>vmp</id>
          <goals>
            <goal>package</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!-- ... other plugins ... -->
  <plugins>
<build>
```

The Vert.x Maven Plug-in needs a property to specify the main verticle for the application:

```
<properties>
  <vertx.verticle>com.example.myapp.MyMainVerticle</vertx.verticle>
  <!-- ... other system properties ... -->
</properties>
```

Configuring the Vert.x Annotation Processor

If your application uses Vert.x event bus services, it also needs the annotation processor that generates the proxy classes and the marshaling code. Add to the project the **io.vertx:vertx-codegen** and **io.vertx:vertx-service-proxy** dependencies, and add

the **io.vertx.codegen.CodeGenProcessor** annotation processor to the Maven Compiler Plug-in configuration:

```
<dependencies>
    <dependency>
        <groupId>io.vertx</groupId>
        <artifactId>vertx-codegen</artifactId>
    </dependency>
    <dependency>
        <groupId>io.vertx</groupId>
        <artifactId>vertx-service-proxy</artifactId>
    </dependency>
    <!-- ... other dependencies ... -->
</dependencies>

<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${version.compiler.plugin}</version>
            <configuration>
                <annotationProcessors>
                    <annotationProcessor>
                        io.vertx.codegen.CodeGenProcessor
                    </annotationProcessor>
                </annotationProcessors>
                <generatedSourcesDirectory>
                    ${project.basedir}/src/main/generated
                </generatedSourcesDirectory>
            </configuration>
        </plugin>
        <!-- ... other plugins .. -->
    <plugins>
<build>
```

You also need to configure the Maven Clean Plug-in to remove the Java source files generated by the annotation processor in the **src/main/generated** folder:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-clean-plugin</artifactId>
            <version>${version.clean.plugin}</version>
            <configuration>
                <filesets>
                    <fileset>
                        <directory>src/main/generated</directory>
                    </fileset>
                </filesets>
            </configuration>
        </plugin>
    <plugins>
<build>
```

```
<!-- ... other plugins .. -->
<plugins>
<build>
```

The Java source files generated in the **src/main/generated** folder should not be managed by a version control system.

Configuring the Vert.x Generator of the Fabric8 Maven Plug-in

For many applications, the Fabric8 Maven Plug-in (FMP) is able to detect a Vert.x application project and configure the correct generator with default values. Just in case, add an explicit reference to the Vert.x generator on the FMP configuration:

```
<build>
<plugins>
<plugin>
<groupId>io.fabric8</groupId>
<artifactId>fabric8-maven-plugin</artifactId>
<version>${version.fabric8-maven-plugin}</version>
<!-- .... <executions> section omitted ... -->
<configuration>
<generator>
<config>
<vertx>
</vertx>
</config>
</generator>
</configuration>
</plugin>
<!-- ... other plugins .. -->
<plugins>
<build>
```

Vert.x applications are expected to use the OpenJDK builder image from Red Hat to be eligible for product support. Add the following properties to your POM:

```
<properties>
<fabric8.generator.fromMode>istag</fabric8.generator.fromMode>
<fabric8.generator.from>redhat-openjdk18-openshift</fabric8.generator.from>
<!-- ... other system properties .. -->
</properties>
```

Note that, except for the addition of the Vert.x generator, the configuration for the FMP is the same as for other Java runtimes supported by RHOAR.

Demonstration: Specifying Vert.x Dependencies

1. Open the POM file of the Ciao microservice Maven project.
2. Note the reference to a parent POM file, and open the parent POM file.
3. In the parent POM file, note the following system properties (they are not contiguous in the file):

```
<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
...
<version.vertx>3.5.1.redhat-003</version.vertx>
<version.vertx-maven-plugin>1.0.7</version.vertx-maven-plugin>
```

- In the parent POM file, note the reference to the **io.vertx:vertx-dependencies** artifact:

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

- Back to the **ciao** Maven project POM file, note the references to Vert.x libraries, for example:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-core</artifactId>
</dependency>
```

- Still in **ciao** Maven project POM file, note the configuration of the Fabric8 Maven plug-in that specify the Vert.x generator:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  ...
  <configuration>
    <generator>
      <config>
        <vertx>
        </vertx>
      </config>
    </generator>
  </configuration>
</plugin>
```

- Log in to OpenShift as the **developer** user and create the **ciao** OpenShift project.

- Use Maven and the Fabric8 Maven Plug-in to deploy the project on OpenShift:

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

- Inspect the route resource fragment file to see the host name to access the microservice.
- Wait for the microservice pod to be ready and running, and test the microservice API using the **curl** command:

```
$ curl -si http://ciao.apps.lab.example.com/api/ciao/Carmela
```



References

JBoss Enterprise Maven Repository

<https://access.redhat.com/maven-repository>

Vert.x Service Proxy

<https://vertx.io/docs/vertx-service-proxy/java/>

The Fabric8 Maven Plug-in Vert.x Generator

<http://maven.fabric8.io/#generator-vertx>

Further information about supported and tested Vert.x libraries is available in the

Red Hat OpenShift Application Runtimes 1.0 Release Notes at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/red_hat_openshift_application_runtimes_release_notes/

Further information about configuring a Maven POM for Vert.x is available in the

Runtime Details chapter of the *Eclipse Vert.x Runtime Guide* for Red Hat OpenShift

Application Runtimes 1.0 at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/eclipse_vert.x_runtime_guide/

► Guided Exercise

Configuring a Maven Project for Vert.x

In this exercise, you will configure the Maven POM for a simple microservice that uses Vert.x and Red Hat OpenShift Application Runtimes (RHOAR).

Outcomes

You should be able to configure the Fabric8 maven Plug-in (FMP) to deploy the microservice to an OpenShift instance.

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-ciao-deploy-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **hello-microservice/ciao** Maven project.
- The **io.vertx:vertx-dependencies**, **io.fabric8:fabric8-maven-plugin**, and **io.fabric8:vertx-maven-plugin** Maven artifacts in the classroom Nexus proxy server.

If you need help to use Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab ciao-deploy setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-ciao-deploy-solution** of the **hello-microservices** Git repository.

- 1. Switch to the **do292-ciao-deploy-begin** branch in the **hello-microservices** Git repository.

You do not need to import the **ciao** project into JBoss Developer Studio because you will make no changes to Java code.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

12. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
13. Switch to the **do292-ciao-deploy-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
  do292-ciao-deploy-begin
...
Switched to a new branch 'do292-ciao-deploy-begin'
```

► 2. Inspect the starter **ciao** project.

- 2.1. Inspect the parent POM.

Open the **~/hello-microservices/pom.xml** file using any text editor, and note the system properties that define the versions of the Fabric8 Maven Plug-in, the Vert.x framework, and the Vert.x Maven Plug-in:

```
...
<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
...
<version.vertx>3.5.1.redhat-003</version.vertx>
<version.vertx-maven-plugin>1.0.7</version.vertx-maven-plugin>
...
```

You can close the **~/hello-microservices/pom.xml** file when done, because you will make no changes to it.

- 2.2. Inspect the incomplete **ciao** Maven project POM.

Open the **~/hello-microservices/ciao/pom.xml** file using any text editor, and note:

- The reference to the parent POM:

```
...
<parent>
  <groupId>com.redhat.training.msa</groupId>
  <artifactId>msa-parent</artifactId>
  <version>1.0</version>
</parent>
...
```

- The **vertx.verticle** property:

```
...
<properties>
  <vertx.verticle>com.redhat.training.msa.ciao.MainVerticle</vertx.verticle>
</properties>
...
```

- The **io.vertx:vertx-dependencies** imported POM, that brings in the tested dependencies for Vert.x with RHOAR:

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

- The Vert.x libraries:

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

- The Maven Compiler Plug-in with the Vert.x annotation processor:

```
.....
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>${version.compiler.plugin}</version>
  <configuration>
    <annotationProcessors>
      <annotationProcessor>
        io.vertx.codegen.CodeGenProcessor
      </annotationProcessor>
    </annotationProcessors>
    <generatedSourcesDirectory>
      ${project.basedir}/src/main/generated
    </generatedSourcesDirectory>
  </configuration>
</plugin>
....
```

- The Fabric8 Vert.x Plug-in:

```
.....
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>vertx-maven-plugin</artifactId>
  <version>${version.vertx-maven-plugin}</version>
  <executions>
    <execution>
```

```

<id>vmp</id>
<goals>
  <goal>package</goal>
</goals>
</execution>
</executions>
</plugin>
...

```

- The incomplete Fabric8 Maven Plug-in (FMP):

```

...
<plugin>
<groupId>io.fabric8</groupId>
<artifactId>fabric8-maven-plugin</artifactId>
<version>${version.fabric8.plugin}</version>
<executions>
</executions>
<configuration>
</configuration>
</plugin>
...

```

Leave the `~/hello-microservices/ciao/pom.xml` file open, because you will make changes to it during the next steps.

► 3. Verify that the project pass unit tests but does not deploy on OpenShift.

- Package the Ciao microservice to run its unit tests and prove that its code is completed.

Enter the `~/hello-microservices/ciao` folder and run the **package** Maven goal. The unit test should pass, and a Fat JAR is generated:

```

[student@workstation hello-microservices]$ cd ciao
[student@workstation ciao]$ mvn clean package
...
INFO: Loaded service_proxies code generator
apr 17, 2018 4:31:51 PM io.vertx.codegen.CodeGenProcessor lambda$process$6
INFO: Generated model com.redhat.training.msa.ciao.service.CiaoService:
  com.redhat.training.msa.ciao.service.CiaoServiceVertxProxyHandler
apr 17, 2018 4:31:51 PM io.vertx.codegen.CodeGenProcessor lambda$process$6
INFO: Generated model com.redhat.training.msa.ciao.service.CiaoService:
  com.redhat.training.msa.ciao.service.CiaoServiceVertxEBProxy
...
Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ ciao ---
[INFO] Building jar: /home/student/hello-microservices/ciao/target/ciao-1.0.jar
[INFO]
[INFO] --- vertx-maven-plugin:1.0.7:package (vmp) @ ciao ---

```

```
...
[INFO] BUILD SUCCESS
...
```

- 3.2. Log in to OpenShift and create the **ciao** OpenShift project:

```
[student@workstation ciao]$ oc login -u developer -p redhat \
  https://master.lab.example.com
Login successful.
...
[student@workstation ciao]$ oc new-project ciao
Now using project "ciao" on server "https://master.lab.example.com:443".
...
```

- 3.3. Try to deploy the Ciao microservice to OpenShift to prove that the FMP configuration is incomplete.

Run the **fabric8:deploy** Maven goal, and skip tests. The build is successful, but it creates no OpenShift resources:

```
[student@workstation ciao]$ mvn fabric8:deploy -DskipTests
...
[INFO] <<< fabric8-maven-plugin:3.5.38:deploy (default-cli) < install @ ciao <<<
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ ciao ---
[WARNING] F8: No such generated manifest file /home/student/github/hello-
microservices/ciao/target/classes/META-INF/fabric8/openshift.yml for this project
so ignoring
[INFO] -----
[INFO] BUILD SUCCESS
...
```

- 3.4. Verify that the **ciao** OpenShift project is still empty.

Run the **oc get all** command. It shows that there are no resources inside the OpenShift project:

```
[student@workstation ciao]$ oc get all
No resources found.
```

▶ 4. Complete the Fabric8 Maven Plug-in configuration to deploy a Vert.x application.

- 4.1. Add the Fabric8 Maven Plug-in (FMP) to the standard Maven life cycle phases.

Complete the **<executions>** element inside the FMP definition in the **~/hello-microservices/ciao/pom.xml** file:

```
...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  <executions>
    <execution>
```

```

<id>fmp</id>
<goals>
  <goal>resource</goal>
  <goal>build</goal>
</goals>
</execution>
</executions>
<configuration>
</configuration>
</plugin>
...

```

4.2. Configure the Fabric8 Maven Plug-in (FMP) to use the Vert.x generator

Complete the **<configuration>** element inside the FMP definition in the **~/hello-microservices/ciao/pom.xml** file:

```

...
<plugin>
<groupId>io.fabric8</groupId>
<artifactId>fabric8-maven-plugin</artifactId>
<version>${version.fabric8.plugin}</version>
<executions>
  <execution>
    <id>fmp</id>
    <goals>
      <goal>resource</goal>
      <goal>build</goal>
    </goals>
  </execution>
</executions>
<configuration>
  <generator>
    <config>
      <vertx>
        </vertx>
    </config>
  </generator>
</configuration>
</plugin>
...

```

Save your changes to the **~/hello-microservices/ciao/pom.xml** file before you continue to the next step.

► 5. Deploy the Ciao microservice to OpenShift.

5.1. Generate the resources required to deploy the microservice on OpenShift.

Run the **fabric8:deploy** Maven goal, and skip tests. The build is successful, and a few OpenShift resources are created:

```

[student@workstation ciao]$ mvn fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ ciao ---

```

```
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator vertx
...
[INFO] F8: Creating BuildServiceConfig ciao-s2i for Source build
[INFO] F8: Creating ImageStream ciao
[INFO] F8: Starting Build ciao-s2i
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ ciao ---
...
[INFO] Creating a Service from openshift.yml namespace test name ciao
...
[INFO] Creating a DeploymentConfig from openshift.yml namespace test name ciao
...
[INFO] Creating Route test:ciao host: ciao.apps.lab.example.com
...
[INFO] BUILD SUCCESS
...
```

5.2. Verify that the OpenShift resources are created.

Run the **oc status** command. Your output may be a little different if your pods are not yet ready:

```
[student@workstation ciao]$ oc status
In project ciao on server https://master.lab.example.com:443

http://ciao.apps.lab.example.com to pod port 8080 (svc/ciao)
dc/ciao deploys istag/ciao:1.0 <-
  bc/ciao-s2i source builds uploaded code on openshift/redhat-openjdk18-
  openshift:latest
    deployment #1 deployed 2 minutes ago - 1 pod
...
```

▶ 6. Test the Ciao microservice.

6.1. Wait for the microservice pod to be ready and running.

Repeat the **oc get pod** command until you get output similar to the following:

```
[student@workstation ciao]$ oc get pod
NAME          READY   STATUS    ...
ciao-1-5k5sw  1/1     Running   ...
ciao-s2i-1-build  0/1     Completed ...
```

6.2. Get the host name to access the microservice.

Use the **oc get route** command to get the host name of the route generated by the Fabric8 Maven Plug-in (FMP):

```
[student@workstation ciao]$ oc get route
NAME      HOST/PORT           PATH      SERVICES  PORT  ...
ciao      ciao.apps.lab.example.com  ciao      8080    ...
```

6.3. Invoke the Ciao microservice API.

Use the host name from the previous step and the **curl** command to access the **/api/ciao/Mario** resource URL:

```
[student@workstation ciao]$ curl -si \
  http://ciao.apps.lab.example.com/api/ciao/Mario
HTTP/1.1 200 OK
...
Ciao Mario, da ciao.apps.lab.example.com
```

► 7. Cleanup.

- 7.1. Delete the **ciao** OpenShift project.

```
[student@workstation ciao]$ oc delete project ciao
project "ciao" deleted
```

- 7.2. Commit your changes to your local Git repository:

```
[student@workstation hello-microservices]$ git commit -a -m \
"Finished the exercise."
```

- 7.3. If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation hello-microservices]$ git reset --hard \
origin/do292-bonjour-deploy-begin
```

This concludes the guided exercise.

► Guided Exercise

Developing Microservices with the Vert.x Runtime

In this exercise, you will complete the Catalog microservice of the Coolstore application, using the Vert.x runtime.

Outcomes

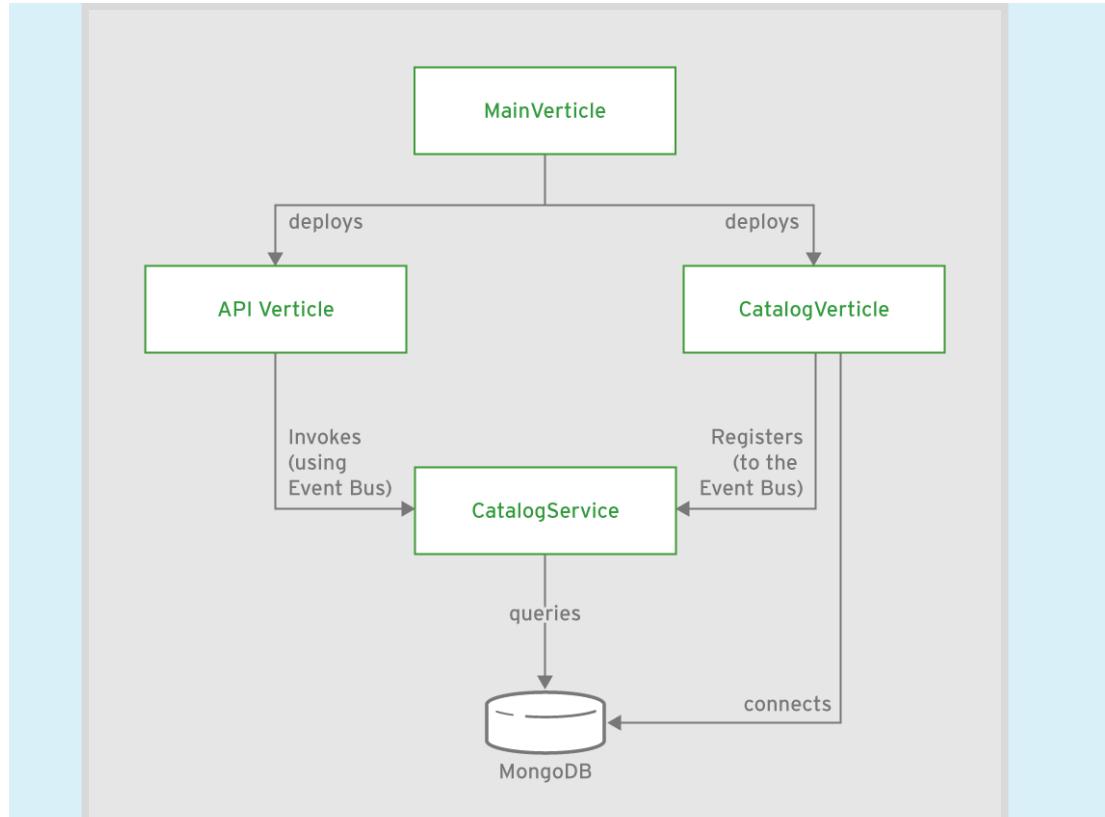
You should be able to:

- Connect a service to the Vert.x event bus.
- Implement verticles to initialize the application and connect to a database.
- Implement an HTTP API server.
- Implement a unit test for asynchronous calls.
- Implement an HTTP API client.
- Implement HTTP API endpoints for health checks.
- Fetch configuration from OpenShift configuration maps.
- Deploy the microservice to OpenShift.

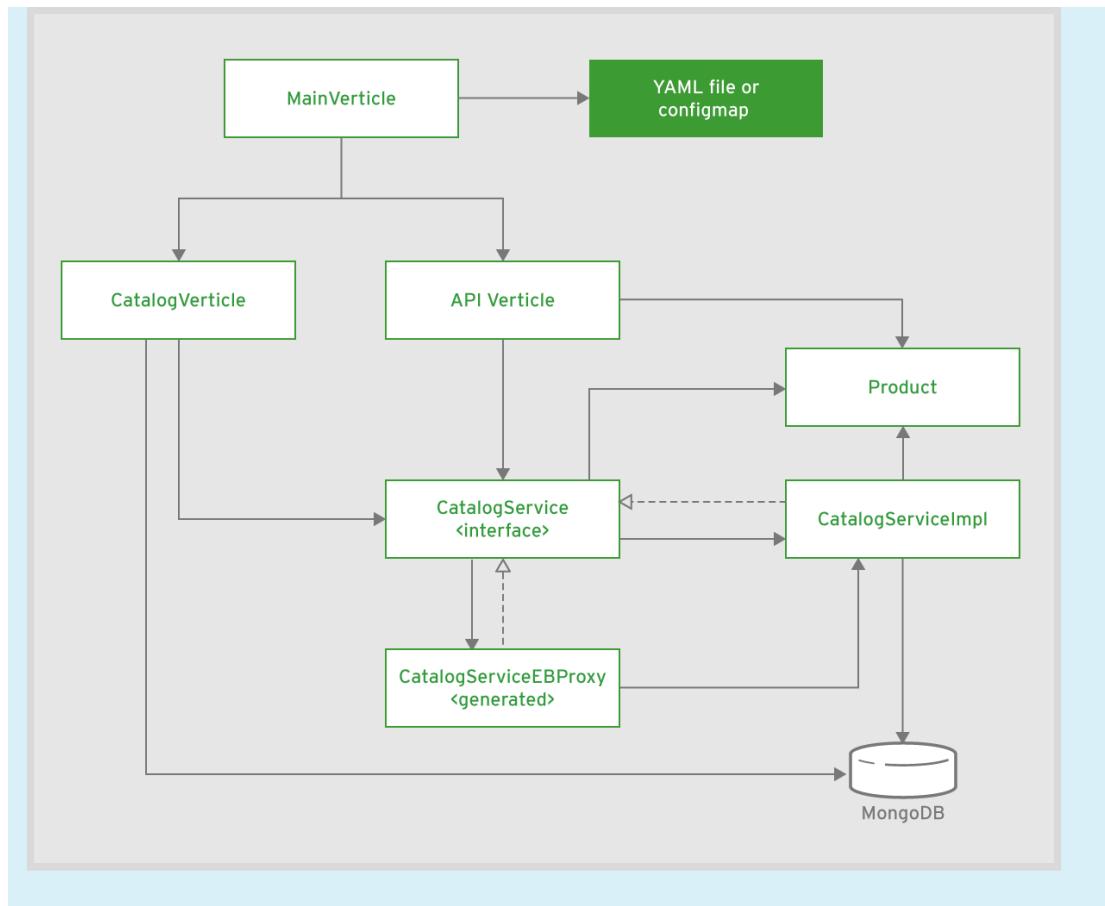
For each outcome, there is a branch that provides the code completed up to that point in the exercise. If you feel that you are taking too much time to complete a step, or you feel that you are stuck and need help, you can browse the solution code for each branch using the classroom GitWeb server. You can also switch to any of the intermediate branches and continue from that point, skipping previous steps.

The Catalog Microservice

The Catalog microservice is composed of three Vert.x verticles and one Vert.x event bus service, as you can see in the following diagram:



The CatalogService is actually composed of one Java interface and a number of support classes. One of the support classes is generated by the Vert.x Maven Plug-in. The following UML class diagram illustrates the relationships between all classes in the Catalog microservice:



To perform this exercise, you need access to:

- A running OpenShift cluster.
- At least one available persistent volume for MongoDB database data.
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-catalog-lab-*** branches in the classroom Git repository, containing the source code for the microservice as part of the coolstore/catalog-service Maven project.
- The **io.vertx:vertx-dependencies**, **io.fabric8:fabric8-maven-plugin**, and **io.fabric8:vertx-maven-plugin** Maven artifacts in the classroom Nexus proxy server.
- The embedded MongoDB server, for unit tests.

If you need help to use Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab catalog-deploy setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-catalog-lab-solution** of the **coolstore** Git repository.

- 1. Switch to the **do292-catalog-lab-begin** branch in the **coolstore** Git repository and import the project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **coolstore** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **coolstore** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/coolstore
Cloning into 'coolstore'...
...
```

- 1.2. If you already have a clone of the **coolstore** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-catalog-lab-begin** branch:

```
[student@workstation ~]$ cd ~/coolstore
[student@workstation coolstore]$ git checkout do292-catalog-lab-begin
...
Switched to a new branch 'do292-catalog-lab-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/coolstore/catalog-service** folder as a Maven project.
- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-catalog-lab-begin**.
- 1.6. The IDE builds the new **catalog-service** project and finds no errors.

- 2. Review the initial state of the **catalog-service** project.

The **CatalogServiceImpl** class implements all logic required to retrieve and store product data in a MongoDB database. The **CatalogServiceTest** class tests this functionality using an embedded MongoDB database that is populated as part of the unit tests.

All other classes in the project are incomplete, and the corresponding tests fail at this point.

- 2.1. Inspect the Maven POM file.

Open the **pom.xml** file from the **catalog-service** project. Switch to the **pom.xml** tab.

You will change this file during this exercise. For now, note the reference to a parent POM file that provides values for system properties. These properties are used to define the version strings for Maven artifacts required to build the microservice.

Also note the use of the POM imported by **io.vertx:vertx-dependencies** to list supported dependencies for the Vert.x framework distribution provided by Red Hat OpenShift Application Runtimes.

- 2.2. Inspect the **CatalogServiceImpl** class.

Open the **CatalogServiceImpl** class from the **com.redhat.coolstore.catalog.vertx.service** package.

There is no need to make any changes to this class, but note how most methods call **resultHandler.handle** and the **Future** interface to signal success or failure of the method.

- 2.3. Inspect the unit tests for the **CatalogService** interface.

Open the **CatalogServiceTest** class from the **com.redhat.coolstore.catalog.vertx.service** package.

There is no need to make any changes to this class. You will complete the **testGetProducts** method later during this exercise.

- 2.4. Run the **CatalogServiceTest** class as a JUnit test.

The **JUnit** view shows that all tests passed, except for **testGetProducts**.

At this point, if you try any other test case in the **catalog-service** project, all their test methods either fail or throw errors. This is expected.

▶ 3. Connect the **CatalogService** interface to the event bus.

Do not run unit tests to validate the changes you make while performing this step. The unit tests in the next step verifies both this step and the next step.



Note

If you want to skip Step 3 and Step 4, check out the branch named **do292-catalog-lab-service**, stash your changes, and continue from Step 5.

Before you continue from Step 5, you also need to update the IDE project properties to include the **/src/main/fabric8** folder in the build path. Perform Step 3.7 and Step 3.8.

- 3.1. Add Vert.x code generator dependencies.

Open the **pom.xml** file from the **catalog-service** project. Switch to the **pom.xml** tab.

Right after the **io.vertx:vertx-web** dependency, add the **io.vertx:vertx-codegen** and **io.vertx:vertx-service-proxy** dependencies.

Use the listing that follows as a reference for adding the dependencies:

```
...
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-web</artifactId>
</dependency>
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-codegen</artifactId>
</dependency>
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-service-proxy</artifactId>
</dependency>
<dependency>
    <groupId>io.vertx</groupId>
```

```

<artifactId>vertx-mongo-client</artifactId>
</dependency>
...

```

- 3.2. Add the Vert.x code generator to the Maven build.

Add the **io.vertx.codegen.CodeGenProcessor** annotation processor to the **maven-compiler-plugin** configuration.

Use the listing that follows as a reference for changing the plug-in configuration:

```

...
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>${version.compiler.plugin}</version>
  <configuration>
    <annotationProcessors>
      <annotationProcessor>
        io.vertx.codegen.CodeGenProcessor
      </annotationProcessor>
    </annotationProcessors>
    <generatedSourcesDirectory>
      ${project.basedir}/src/main/generated
    </generatedSourcesDirectory>
  </configuration>
</plugin>
...

```

Update your IDE project configuration after making changes to the POM.

- 3.3. Annotate the **Product** class for the event bus.

Open the **Product** class from the **com.redhat.coolstore.catalog.model** package.

Add the **@DataObject** annotation to the class. If you trigger the IDE auto-completion feature, the IDE adds the correct **import** statement to the source code.

Use the listing that follows as a reference for changing the class:

```

...
import io.vertx.codegen.annotations.DataObject;
...
@DataObject
public class Product implements Serializable {
...

```

- 3.4. Annotate the **com.redhat.coolstore.catalog.model** package for code generation.

Create the **package-info.java** file in the **com.redhat.coolstore.catalog.model** package.

Add the **package** declaration and the **@ModuleGen** annotation. If you trigger the IDE auto-completion feature, the IDE uses the correct package name for the annotation.

The **@ModuleGen** annotation requires two arguments. Pass the complete package name to the **groupPackage** argument, and **coolstore-catalog-model** to the **name** argument.

Use the listing that follows as a reference for the complete contents of the **package-info.java** file:

```
@io.vertx.codegen.annotations.ModuleGen(
  groupPackage="com.redhat.coolstore.catalog.model",
  name="coolstore-catalog-model")
package com.redhat.coolstore.catalog.model;
```

3.5. Annotate the **CatalogService** interface for the event bus.

Open the **CatalogService** class from the **com.redhat.coolstore.catalog.vertx.service** package.

Add the **@ProxyGen** annotation to the interface. Use the listing that follows as a reference for changing the interface:

```
...
import io.vertx.codegen.annotations.ProxyGen;
...
@ProxyGen
public interface CatalogService {
  ...
}
```

3.6. Annotate the **com.redhat.coolstore.catalog.vertx.service** package for code generation.

Create a **package-info.java** file in the **com.redhat.coolstore.catalog.vertx.service** package. Pass **coolstore-catalog-service** as the value for the **name** argument of the **@ModuleGen** annotation.

Use the listing that follows as a reference for the complete contents of the **package-info.java** file:

```
@io.vertx.codegen.annotations.ModuleGen(
  groupPackage="com.redhat.coolstore.catalog.vertx.service",
  name="coolstore-catalog-service")
package com.redhat.coolstore.catalog.vertx.service;
```

3.7. Run the Vert.x code generation plug-in.

Run a Maven build with the **clean package -DskipTests** arguments.

The **Console** view shows that Maven invoked the Vert.x code generator as part of the build:

```
...
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ catalog-service
...
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 8 source files to /home/student/coolstore/catalog-service/target/
classes
...
INFO: Loaded service_proxies code generator
Apr 02, 2018 6:43:19 PM io.vertx.codegen.CodeGenProcessor lambda$process$6
```

```

INFO: Generated model
com.redhat.coolstore.catalog.vertx.service.CatalogService:
com.redhat.coolstore.catalog.vertx.service.CatalogServiceVertxEBProxy
Apr 02, 2018 6:43:19 PM io.vertx.codegen.CodeGenProcessor lambda$process$6
INFO: Generated model
com.redhat.coolstore.catalog.vertx.service.CatalogService:
com.redhat.coolstore.catalog.vertx.service.CatalogServiceVertxProxyHandler
...

```

3.8. Configure the project build path.

Refresh the project in the IDE and add the **/src/main/generated** folder to the project's build path.

Wait while the IDE rebuilds the project.

3.9. Complete the helper methods in the **CatalogService** interface to instantiate the service implementation and the service proxy.

Open the **CatalogService** class from the **com.redhat.coolstore.catalog.vertx.service** package.

Notice that the static methods **create** and **createProxy** return **null**. Change the first method to return a new instance of the **CatalogServiceImpl** class, and the second method to return a new instance of the generated **CatalogServiceVertxEBProxy** class.

Use the listing that follows as a reference for changing the interface:

```

...
    static CatalogService create(Vertx vertx, JsonObject config,
        MongoClient client) {
        return new CatalogServiceImpl(vertx, config, client);
    }
...
    static CatalogService createProxy(Vertx vertx) {
        return new CatalogServiceVertxEBProxy(vertx, ADDRESS);
    }
...

```

▶ 4. Implement the verticle for the **CatalogService** interface.

The **CatalogVerticle** class connects to the MongoDB database and registers the **CatalogService** interface to the event bus.

The **CatalogVerticleTest** class deploys the verticle and invokes the **CatalogService** interface using the event bus, thus testing the changes from this step and also from the previous step.

4.1. Inspect the incomplete code in the **CatalogVerticle** class.

Open the **CatalogVerticle** class from the **com.redhat.coolstore.catalog.vertx.service** package.

The **start** method already includes the code to connect to the MongoDB database, but it misses the code to register the **CatalogService** interface to the event bus.

4.2. Create a service object that implements the **CatalogService** interface, and connects the service object to the MongoDB database.

Use the helper static method from the **CatalogService** interface to instantiate the service object and pass the connection to the MongoDB database as an argument.

Use the following listing as a guide for the changes to the **CatalogVerticle** class:

```
...
@Override
public void start(Future<Void> startFuture) throws Exception {
    client = MongoClient.createShared(vertx, config());
    CatalogService service = CatalogService.create(getVertx(), config(),
        client);
    startFuture.complete();
}
...
```

- 4.3. Register the service object to the event bus.

Use the Vert.x **ProxyHelper** class to register the service object to the event bus.

Ignore warnings about the **ProxyHelper** class being deprecated.

Use the following listing as a guide for the changes to the **CatalogVerticle** class:

```
...
import io.vertx.serviceproxy.ProxyHelper;

...
@Override
public void start(Future<Void> startFuture) throws Exception {
    client = MongoClient.createShared(vertx, config());
    CatalogService service = CatalogService.create(vertx, config(),
        client);
    ProxyHelper.registerService(CatalogService.class, vertx, service,
        CatalogService.ADDRESS);
    startFuture.complete();
}
...
```

- 4.4. Inspect the unit test for the **CatalogVerticle** class.

Open the **CatalogVerticleTest** class from the **com.redhat.coolstore.catalog.vertx.service** package.

There is no need to make any changes to this class. It already includes the code required to deploy an **CatalogVerticle** instance and invoke the **CatalogService** interface using the event bus.

- 4.5. Run the **CatalogVerticleTest** class as a JUnit test.

The **JUnit** view shows that the single test method passed. If not, review your changes during this and the previous step, and repeat the test until it pass.

- ▶ 5. Implement the HTTP API of the microservice.

The **ApiVerticle** class starts the Vert.x HTTP server and defines routes that delegates each request to the **CatalogService** interface using the event bus.

The **ApiVerticleTest** class provide unit tests for the **ApiVerticle** class. Not all test methods are expected to pass at the end of this step: you will complete one of them in a later step of this exercise, and the tests related to health probes will pass only you after implement the health feature later in this exercise.

**Note**

If you want to skip Step 5, check out the branch named **do292-catalog-lab-api**, stash your changes, and continue from Step 6.

Before you continue from Step 6, you also need to update the IDE project properties to include the **/src/main/fabric8** folder in the build path. Perform Step 3.7 and Step 3.8.

5.1. Inspect the incomplete **ApiVerticle** class.

Open the **ApiVerticle** class from the **com.redhat.coolstore.catalog.api** package.

Note that:

- The constructor receives an instance of **CatalogService** interface. This instance is expected to be an event bus proxy.
- The **start** method defines a **Router** object that is passed to an **HttpServer** object. You need to add routes to the **Router** object, which corresponds to each HTTP API request, and register handler methods for each route.
- The **getProduct** handler methods is already completed for you to use as a model to complete the remaining handler methods.

5.2. Complete the **getProducts** handler method:

- Call the **CatalogService.getProducts** method and convert the returned **List<Product>** object to a **JSONArray** object.
- Set the **JSONArray** object as the request response, and set the **Content-type** header to **application/json**.
- Fail the **RoutingContext** if the call to **CatalogService.getProducts** failed.

Use the following listing as a guide for the completed **getProducts** method.

```
...
private void getProducts(RoutingContext rc) {
    catalogService.getProducts(ar -> {
        if (ar.succeeded()) {
            List<Product> products = ar.result();
            JSONArray json = new JSONArray();
            products.stream()
                .map(p -> p.toJson())
                .forEach(p -> json.add(p));
            rc.response()
                .putHeader("Content-type", "application/json")
                .end(json.encodePrettily());
        }
        else {
            rc.fail(ar.cause());
        }
    })
}
```

```

    });
}
...

```

5.3. Complete the **addProduct** handler method:

- Get the request body as a **JsonObject** and convert it to a **Product** object.
- Call the **CatalogService.addProducts** method, passing the **Product** object.
- Return an HTTP 201 status code if the call to **CatalogService.addProduct** was successful.
- Fail the **RoutingContext** if the call to **CatalogService.addProduct** failed.

Use the following listing as a guide for the completed **addProduct** method:

```

...
private void addProduct(RoutingContext rc) {
    JsonObject json = rc.getBodyAsJson();
    catalogService.addProduct(new Product(json), ar -> {
        if (ar.succeeded()) {
            rc.response().setStatusCode(201).end();
        }
        else {
            rc.fail(ar.cause());
        }
    });
}
...

```

5.4. Define the routes for each HTTP API request.

The API endpoints are:

- GET **/products** to return all products
- GET **/product/{:itemId}** to return a single product
- POST **/product** to add a single product

For each API endpoint, set the appropriate handler method.

You also need to add a **BodyHandler** to the **/product** endpoint, else the **addProduct** handler method does not get the POST request data.

Use the following listing as a guide for the changes to the **start** method:

```

...
@Override
public void start(Future<Void> startFuture) throws Exception {

    Router router = Router.router(vertx);
    router.get("/products").handler(this::getProducts);
    router.get("/product/:itemId").handler(this::getProduct);
    router.route("/product").handler(BodyHandler.create());
    router.post("/product").handler(this::addProduct);
}

```

```

        vertx.createHttpServer().listen(
            config().getInteger("catalog.http.port", 8080),
            result -> {
                if (result.succeeded()) {
                    startFuture.complete();
                }
                else {
                    startFuture.fail(result.cause());
                }
            });
        ...
    }
...

```

- 5.5. Set the **Router** object **accept** method as the request handler for the **HttpServer** object.

Use the following listing as a guide for the changes to the **start** method:

```

...
@Override
public void start(Future<Void> startFuture) throws Exception {

    Router router = Router.router(vertx);
    router.get("/products").handler(this::getProducts);
    router.get("/product/:itemId").handler(this::getProduct);
    router.route("/product").handler(BodyHandler.create());
    router.post("/product").handler(this::addProduct);

    vertx.createHttpServer().requestHandler(router::accept).listen(
        config().getInteger("catalog.http.port", 8080),
        result -> {
            if (result.succeeded()) {
                startFuture.complete();
            }
            else {
                startFuture.fail(result.cause());
            }
        });
}
...

```

- 5.6. Inspect the unit test for the **ApiVerticle** class.

Open the **ApiVerticleTest** class from the **com.redhat.coolstore.catalog.api** package.

There is no need to make any changes to the **ApiVerticleTest** class. Not all test methods are expected to pass at this point. They depend on code you will implement later during this exercise.

The **ApiVerticleTest** class deploys an **ApiVerticle** instance and invokes the HTTP API using the Vert.x **HttpClient** class. The **ApiVerticleTest** class also mocks calls to the **CatalogService** interface, so neither the Vert.x event bus nor a MongoDB database are exercised during the unit tests.

- 5.7. Run the **ApiVerticleTest** class as a JUnit test.

The **JUnit** view shows that three test methods passed:

- **testGetProducts**
- **testAddProduct**
- **testGetNonExistingProduct**

The **JUnit** view also shows that the remaining three test methods failed:

- **testGetProduct**
- **testLiveness**
- **testHealthReadiness**

These failures are expected at this point in the exercise. If you get a different outcome, review your changes during this step and repeat the tests until you get the correct tests to pass.

▶ 6. Complete a unit test for asynchronous code.

The **CatalogServiceTest** class initializes an embedded MongoDB database to test the **CatalogServiceImpl** class, which implements the Vert.x service described by the **CatalogService** interface.

The **testGetProducts** test method was left incomplete, and it is your task to complete it. The **testGetProducts** test method exercises the **CatalogService.getProducts** method that was provided ready to run in the beginning of this exercise.



Note

If you want to skip Step 6, check out the branch named **do292-catalog-lab-test**, stash your changes, and continue from Step 7.

Before you continue from Step 7, you also need to update the IDE project properties to include the **/src/main/fabric8** folder in the build path. Perform Step 3.7 and Step 3.8.

6.1. Inspect the **CatalogServiceTest** class.

Open the **CatalogServiceTest** class from the **com.redhat.coolstore.catalog.vertx.service** package.

Note that the **CatalogServiceTest** class uses the **VertxUnitRunner** JUnit runner class and optionally feeds each test method with a **TestContext** argument.

Review the **setup** and **tearDown** methods. They bootstrap and shutdown the Vert.x runtime and an embedded MongoDB database for use by the test methods.

Also note the use of the **TestContext.async** method to get an **Async** object. The **Async** object allows a procedural JUnit test method to wait until all asynchronous calls are finished before moving on.

6.2. Inspect the **testGetProduct** method.

The **testGetProduct** test method performs the following high-level tasks:

- Insert test data into the embedded MongoDB database.
- Wait for the asynchronous calls to the MongoDB database to be completed.

- Create a new **CatalogServiceImpl** service object.
- Invoke the service method.
- Use the **org.hamcrest** static methods to verify the asynchronous result from the service object.
- Alert the **TestContext** of the successful or failed completion of the asynchronous calls.

Use the **testGetProduct** test method code as a reference to complete the **testGetProducts** test method.

6.3. Prepare test data for the **testGetProducts** test method.

Create two **JsonObject** instances with product data, and save them into the **products** collection of the MongoDB database, using the **MongoClient.save** method. Don't forget to signal the **TestContext** about the completion of the asynchronous calls to MongoDB, using the **Async** object.

Use the following listing as a reference to the code to add to the **testGetProducts** method:

```
...
@Test
public void testGetProducts(TestContext context) throws Exception {
    Async saveAsync = context.async(2);

    String itemId1 = "111111";
    JsonObject json1 = new JsonObject()
        .put("itemId", itemId1)
        .put("name", "productName1")
        .put("desc", "productDescription1")
        .put("price", new Double(100.0));
    mongoClient.save("products", json1, ar -> {
        if (ar.failed()) {
            context.fail();
        }
        saveAsync.countDown();
    });

    String itemId2 = "222222";
    JsonObject json2 = new JsonObject()
        .put("itemId", itemId2)
        .put("name", "productName2")
        .put("desc", "productDescription2")
        .put("price", new Double(100.0));
    mongoClient.save("products", json2, ar -> {
        if (ar.failed()) {
            context.fail();
        }
        saveAsync.countDown();
    });

    saveAsync.await();
}
```

```
...
}
```

6.4. Verify the data returned by the **CatalogServiceImpl** class.

Assert that the returned value is not **null**, that it contains the expected number of products, and that the **itemId** of the returned products match the test data, and remove the call to **fail** in the end of the **testGetProducts** method.

The **org.hamcrest** packages provide a number of useful helpers for these asserts, such as **allOf** and **hasItem**.

Use the following listing as a reference to the code to add to the **testGetProducts** method:

```
...
@Test
public void testGetProducts(TestContext context) throws Exception {
    ...
    service.getProducts(ar -> {
        if (ar.failed()) {
            context.fail(ar.cause().getMessage());
        }
        else {
            assertThat(ar.result(), notNullValue());
            assertThat(ar.result().size(), equalTo(2));
            Set<String> itemIds = ar.result().stream().map(
                p -> p.getItemId()).collect(Collectors.toSet());
            assertThat(itemIds.size(), equalTo(2));
            assertThat(itemIds, allOf(hasItem(itemId1), hasItem(itemId2)));

            async.complete();
        }
    });
    ...
}
```

6.5. Run the completed **CatalogServiceTest** class as a JUnit test.

The **JUnit** view shows that all five test methods passed.

If your test method fails, review your changes during this step and repeat the test until you get all tests to pass.

► 7. Implement an HTTP API client.

To practice the Vert.x way of calling an external HTTP API, complete the **testGetProduct** test method of the **ApiVerticleTest** class.

The **ApiVerticleTest** class mocks the **CatalogService** interface and does not use the event bus. You have to complete the configuration of the mock objects used by the test method.

Note

If you want to skip Step 7, check out the branch named **do292-catalog-lab-client**, stash your changes, and continue from Step 8.

Before you continue from Step 8, you also need to update the IDE project properties to include the **/src/main/fabric8** folder in the build path. Perform Step 3.7 and Step 3.8.

7.1. Inspect the **ApiVerticleTest** class.

Open the **ApiVerticleTest** class from the **com.redhat.coolstore.catalog.api** package.

Note that the **ApiVerticleTest** class is similar to the **CatalogServiceTest** class in the use of the **VertxUnitRunner** test runner, **TestContext** and **Async** objects.

Review the **setUp** and **tearDown** methods. They bootstrap and shutdown the Vert.x runtime. The **setUp** method also sets the Vert.x configuration with unused local TCP port, and creates a mock for the **CatalogServiceTest** class using Mockito. Finally the **setUp** method deploys the **ApiVerticle** class.

7.2. Inspect the **testGetProducts** method.

The **testGetProducts** test method performs the following high-level tasks:

- Creates a **List<Product>** object to use as a return value for the mocked **CatalogService** interface.
- Uses the Mockito's static **doAnswer** method to mock a call to **CatalogService.getProducts** with any arguments.
- Returns the **List<Product>** object using a Vert.x asynchronous result handler.
- Creates a Vert.x **HttpClient** object to submit an HTTP GET request.
- Sets a body handler in the **HttpClientResponse** to fetch the JSON data returned by the HTTP API request.
- Converts the JSON data to **Product** objects.
- Uses the **org.hamcrest** asserts to verify that the returned product data has the expected number of objects, with the correct **itemIds**.

Use the **testGetProducts** method code as a reference to complete the **testGetProduct** method.

7.3. Configure the mock for the **testGetProduct** method.

The incomplete **testGetProduct** method already creates a **Product** object for the mocked **CatalogService** interface, and has a skeleton call to Mockito's **doAnswer**.

Complete the configuration of the mock object to return the **Product** object using the asynchronous result handler.

Use the following listing as a reference to the code to add to the **testGetProduct** method:

```

    ...
    public void testGetProduct(TestContext context) throws Exception {
        String itemId1 = "111111";
        JsonObject json1 = new JsonObject()
            .put("itemId", itemId1)
            .put("name", "productName1")
            .put("desc", "productDescription1")
            .put("price", new Double(100.0));
        Product product = new Product(json1);

        doAnswer(new Answer<Void>() {
            public Void answer(InvocationOnMock invocation){
                Handler<AsyncResult<Product>> handler =
                    invocation.getArgument(1);
                handler.handle(Future.succeededFuture(product));
                return null;
            }
        }).when(catalogService).getProduct(any(),any());

        verify(catalogService).getProduct(any(),any());
        Async async = context.async();
        fail("Not implemented yet");
    }
    ...

```

7.4. Invoke the HTTP API endpoint.

Create a Vert.x **HttpClient** object that sends a GET request to fetch the product whose **itemId** matches the mocked **CatalogService** interface call. The URL path of this request is **/product/111111**. Use the TCP port found by the **setUp** method.

Remember to signal completion of the asynchronous calls, invoke the **TestContext** exception handler, and to remove the call to **fail**. Note that the existing calls to the Mockito's **verify** method and to the **async.complete()** method need to be moved to inside the HTTP response handler code you are adding.

Use the following listing as a reference to the code to add to the **testGetProduct** method. Beware that the new code is added around the existing code.

```

    ...
    public void testGetProduct(TestContext context) throws Exception {
        ...
        Async async = context.async();

        vertx.createHttpClient().get(port, "localhost", "/product/" + itemId1,
            response -> {

            verify(catalogService).getProduct(any(),any());
            async.complete();

        }).exceptionHandler(context.exceptionHandler()).end();
    }
    ...

```

7.5. Fetch the JSON data returned by the HTTP API endpoint.

Assert that the HTTP response has a successful HTTP status 200 and that it is of the correct content type.

Use a **bodyHandler** to get JSON data returned by the **HttpClient** and convert it to a **Product** object.

Use the following listing as a reference to the code to add to the **testGetProduct** method:

```
...
public void testGetProduct(HandlerContext context) throws Exception {
    ...
    Async async = context.async();

    vertx.createHttpClient().get(port, "localhost", "/product/" + itemId1
        response -> {
            assertThat(response.statusCode(), equalTo(200));
            assertThat(response.headers().get("Content-Type"),
                equalTo("application/json"));
            response.bodyHandler(body -> {
                JsonObject json = body.toJsonObject();
                Product productResult = new Product(json);

                verify(catalogService).getProduct(any(),any());
                async.complete();
            });
        }).exceptionHandler(context.exceptionHandler()).end();
    }
...
}
```

7.6. Assert that the returned **Product** object has the correct attributes.

Assert that the product data is the expected one, from the mocked **CatalogService** interface call, using the **org.hamcrest** static methods, and remove the call to **fail** in the end of the **testGetProduct** method

Use the following listing as a reference to the code to add to the **testGetProduct** method:

```
...
public void testGetProduct(HandlerContext context) throws Exception {
    ...
    Async async = context.async();

    vertx.createHttpClient().get(port, "localhost", "/product/" + itemId1
        response -> {
            assertThat(response.statusCode(), equalTo(200));
            assertThat(response.headers().get("Content-Type"),
                equalTo("application/json"));
            response.bodyHandler(body -> {
                JsonObject json = body.toJsonObject();
                Product productResult = new Product(json);

                assertThat(productResult, notNullValue());
                assertThat(productResult.getItemId(), equalTo("111111"));
            });
        }).exceptionHandler(context.exceptionHandler()).end();
    }
...
```

```

        assertThat(productResult.getPrice(), equalTo(100.0));

        verify(catalogService).getProduct(any(), any());
        async.complete();
    });

}).exceptionHandler(context.exceptionHandler()).end();
}
...

```

7.7. Run the completed **ApiVerticleTest** class as a JUnit test.

The **JUnit** view shows that four of the six test methods passed. Only the following tests fail:

- **testLiveness**
- **testHealthReadiness**

If you get a different output, review your changes during this step and repeat the test until you get the correct mix of passed and failed tests.

► 8. Implement health probes.

Change the **ApiVerticle** class to add HTTP API endpoints for health probes. You do not need to implement extensive health checks, your probes can always return a healthy status.

The **ApiVerticleTest** provides unit tests to verify your health probes.



Note

If you want to skip Step 8, check out the branch named **do292-catalog-lab-health**, stash your changes, and continue from Step 9.

Before you continue from Step 9, you also need to update the IDE project properties to include the **/src/main/fabric8** folder in the build path. Perform Step 3.7 and Step 3.8.

8.1. Add the Vert.x health check dependencies to the project.

Open the **pom.xml** file from the **catalog-service** project. Switch to the **pom.xml** tab.

Right after the **io.vertx:vertx-mongo-client** dependency, add the **io.vertx:vertx-health-check** dependency.

Use the listing that follows as a reference for adding the dependency:

```

...
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-mongo-client</artifactId>
</dependency>
<dependency>
    <groupId>io.vertx</groupId>
    <artifactId>vertx-health-check</artifactId>
</dependency>
<dependency>
    <groupId>junit</groupId>

```

```

<artifactId>junit</artifactId>
<version>${junit.version}</version>
<scope>junit</scope>
</dependency>
...

```

8.2. Inspect the incomplete **ApiVerticle** class.

Open the **ApiVerticle** class from the **com.redhat.coolstore.catalog.api** package.

Recall that the **start** method defines a **Router** object, which registers a route and a handler to each HTTP API request.

8.3. Add a route and a handler for the readiness probe.

Add a route for the **/health/readiness** HTTP API entry point, using the HTTP GET method, that returns the string "OK".

This simplistic probe may be adequate for a number of applications, because if the microservice can answer a fixed string, this means the microservice is ready to server HTTP API requests, so it is "ready".

Use the following listing as a guide for the code to add to the **start** method:

```

...
public void start(Future<Void> startFuture) throws Exception {
    Router router = Router.router(vertx);
    ...
    router.post("/product").handler(this::addProduct);

    router.get("/health/readiness").handler(requestHandler -> {
        requestHandler.response().end("OK");
    });

    vertx.createHttpServer().requestHandler(router::accept).listen(
    ...
}
...

```

8.4. Add a Vert.x **health** method to the **ApiVerticle** class that delegates the **ping** method to the **CatalogService** interface.

The **health** method returns either the **Status.OK** or the **Status.KO** constant defined in the **io.vertx.ext.healthchecks.Status** interface, based on the return of the **CatalogService.ping** method.

Use the following listing as a guide for the complete contents of the **health** method of the **ApiVerticle** class:

```

...
import io.vertx.ext.healthchecks.Status;
...

private void health(Future<io.vertx.ext.healthchecks.Status> future) {
    catalogService.ping(ar -> {
        if (ar.succeeded()) {
            if (!future.isComplete()) {
                future.complete(Status.OK());
            }
        }
    });
}

```

```

        else {
            future.complete(Status.OK());
        }
    });
}
...

```

8.5. Add a route and a handler for the liveness probe.

Register the **health** method in a Vert.x **io.vertx.ext.healthchecks.HealthCheckHandler** object. Use the static create method to create object.

Add a route for the **/health/liveness** HTTP API entry point, using the HTTP GET method, that invokes the Vert.x **HealthCheckHandler** object.

Use the following listing as a guide for the code to add to the **start** method:

```

...
import io.vertx.ext.healthchecks.HealthCheckHandler;
...

public void start(Future<Void> startFuture) throws Exception {
    Router router = Router.router(vertx);
    ...

    router.post("/product").handler(this::addProduct);
    router.get("/health/readiness").handler(requestHandler -> {
        requestHandler.response().end("OK");
    });
    HealthCheckHandler healthCheckHandler =
        HealthCheckHandler.create(vertx).register(
            "health", f -> health(f));
    router.get("/health/liveness").handler(healthCheckHandler);

    vertx.createHttpServer().requestHandler(router::accept).listen(
        ...
    }
...

```

8.6. Inspect the **ping** method of the **CatalogServiceImpl** class.

Open the **CatalogServiceImpl** class from the **com.redhat.coolstore.catalog.vertx.service** package.

The listing that follows shows the provided implementations of the **ping** method, which unconditionally returns success:

```

...
public void ping(Handler<AsyncResult<String>> resultHandler) {
    resultHandler.handle(Future.succeededFuture("OK"));
}
...

```

8.7. Inspect the unit tests for the health probes.

Open the **ApiVerticleTest** class from the **com.redhat.coolstore.catalog.api** package.

The **testHealthReadiness** test method just verifies whether the readiness HTTP API endpoint returned a success HTTP status code. The **testLiveness** verifies the expected response from a Vert.x **HealthCheckHandler** object.

- 8.8. Run the unit tests for the health probes.

Run the **ApiVerticleTest** class as a JUnit test.

The **JUnit** view shows that all test methods passed:

If you get a failed test, review your changes during this step and repeat the tests until all tests pass.

- 8.9. Optional: Perform a manual test of the health probes.

You are done with all code changes to the Catalog microservice. Now would be a good time to run the complete application, because so far you have only executed unit tests.

Another reason to perform a test of the full application is the fact that the **ApiVerticleTest** class mocked the **CatalogService** object. A test of the full application would make the **ApiVerticle** class call the **CatalogServiceImpl** class using the event bus, something that none of the unit tests do.

To save the time required to provide and initialize a local MongoDB database, and also to provide the application with configuration to access the database, you can package and run the application Fat JAR, and invoke only the health probes.

First, open a terminal window and use Maven to package the application as a Fat JAR, and use the **java** command to run the application. Ignore the errors to connect to MongoDB:

```
[student@workstation ~]$ cd ~/coolstore/catalog-service
[student@workstation catalog-service]$ mvn clean package -DskipTests
...
[student@workstation catalog-service]$ java -jar \
  target/catalog-service-1.0.0-SNAPSHOT.jar
...
com.mongodb.MongoSocketOpenException: Exception opening socket
...
INFO: Succeeded in deploying verticle
...
```

Open another terminal window and invoke the health endpoints using the **curl** command:

```
[student@workstation ~]$ curl -si http://localhost:8080/health/readiness
HTTP/1.1 200 OK
...
OK
[student@workstation ~]$ curl -si http://localhost:8080/health/liveness
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
...
>{"checks": [{"id": "health", "status": "UP"}], "outcome": "UP"}
```

Switch back to the first terminal and press **Ctrl+C** to stop the application.

- ▶ 9. Complete the **MainVerticle** to fetch microservice configurations from OpenShift configuration maps.

You will test the completed Catalog microservice during the next step, running under OpenShift.



Note

If you want to skip Step 9, check out the branch named **do292-catalog-lab-solution**, stash your changes, and continue from Step 10.

Before you continue from Step 10, you also need to update the IDE project properties to include the **/src/main/fabric8** folder in the build path. Perform Step 3.7 and Step 3.8.

- 9.1. Add the Vert.x configuration dependencies to the project.

Open the **pom.xml** file from the **catalog-service** project. Switch to the **pom.xml** tab.

Right after the **io.vertx:vertx-config-yaml** dependency, add the **io.vertx:vertx-config-kubernetes** dependency.

Use the listing that follows as a reference for adding the dependency:

```
...
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-config</artifactId>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-config-yaml</artifactId>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-config-kubernetes-configmap</artifactId>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-health-check</artifactId>
</dependency>
...

```

- 9.2. Add the Fabric8 Maven Plug-in (FMP) to the Maven life cycle.

At the end of the **pom.xml** file, before the closing **</plugins>** element, there is a reference to the FMP. The code for the **<execution>** element is commented out in the POM file because it would force every build to generate OpenShift resource files, slowing down the previous steps. You only need to uncomment the **<execution>** element.

Use the listing that follows as a reference adding FMP to the Maven life cycle:

```
...
<plugin>
  <groupId>io.fabric8</groupId>

```

```

<artifactId>fabric8-maven-plugin</artifactId>
<version>${version.fabric8-maven-plugin}</version>
<executions>
  <execution>
    <id>fmp</id>
    <goals>
      <goal>resource</goal>
      <goal>build</goal>
    </goals>
  </execution>
</executions>
<configuration>
</configuration>
</plugin>
...
  
```

- Configure the Fabric8 Maven Plug-in (FMP) to use the Vert.x generator.

Inside the **<configuration>** element of the FMP plug-in, add a **<generator>** with a **<config>** that references the **<vertx>** generator.

Use the listing that follows as a reference for adding the generator:

```

...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <generator>
      <config>
        <vertx>
        </vertx>
      </config>
    </generator>
  </configuration>
</plugin>
  
```

- Review the Vert.x properties in the project's POM.

At the beginning of the **pom.xml** file, **vertx.verticle** property references the **com.redhat.coolstore.catalog.MainVerticle** class as the startup verticle for the application.

- Since you have changed the Maven POM for the project, update the project configuration in the IDE. Refer to Appendix B, *Working With Red Hat JBoss Developer Studio* for detailed instructions on updating the project configuration in the IDE.

9.6. Inspect the incomplete **MainVerticle** class.

Open the **MainVerticle** class from the **com.redhat.coolstore.catalog.verticle** package.

Note that there are two versions of the **start** method. The shorter one just invokes the **deployVerticles** method, and the longer one is commented out. The longer version of the **start** method retrieves Vert.x configuration before invoking the **deployVerticles** method.

You do not need to change the **stop** and **deployVerticles** methods. The **deployVerticles** method deploys the **CatalogVerticle** and the **ApiVerticle** using the configuration passed as an argument.

9.7. Switch to the longer version of the **start** method.

Delete the shorter version of **start** method, and uncomment the incomplete, longer **start** method.

9.8. Complete the longer version of the **start** method to retrieve Vert.x configurations from an OpenShift configuration map.

Use a Vert.x **ConfigStoreOptions** object to configure the framework to fetch configurations from the OpenShift configuration map resource named **app-config**, using the YAML format.

Create a **ConfigRetrieverOptions** object that references the **ConfigStoreOptions** object, and then a **ConfigRetriever** object that references the **ConfigStoreOptions** object.

Finally, if the **ConfigRetriever** object successfully loads the configuration, invoke the **deployVerticles** method, passing the configuration as an argument.

Use the listing that follows as a reference for the changes to the longer version of the **start** method:

```
...
public void start(Future<Void> startFuture) throws Exception {
    ConfigStoreOptions appStore = new ConfigStoreOptions();
    appStore.setType("configmap").setFormat("yaml");
    appStore.setConfig(new JsonObject()
        .put("name", "app-config")
        .put("key", "app-config.yaml"));

    ConfigRetrieverOptions configRetrieverOptions =
        new ConfigRetrieverOptions();
    configRetrieverOptions.addStore(appStore);

    ConfigRetriever retriever = ConfigRetriever.create(
        vertx, configRetrieverOptions);
    retriever.getConfig(handler -> {
        if (handler.succeeded()) {
            deployVerticles(handler.result(), startFuture);
        } else {
            startFuture.fail(handler.cause());
        }
    });
}
...
```

► 10. Deploy the Catalog microservice to OpenShift and test the microservice API.

10.1. Log in to OpenShift and create a new project.

From a terminal window, log in to OpenShift as the **developer** user and create the **catalog-service** project:

```
[student@workstation ~]$ oc login -u developer -p redhat \
  https://master.lab.example.com
Login successful.
...
[student@workstation ~]$ oc new-project catalog-service
Now using project "catalog-service" on server "https://
master.lab.example.com:443".
...
```

10.2. Inspect the template for a Mongo DB database.

Enter the **catalog-service** folder inside the local **coolstore** Git repository and inspect the provided template for MongoDB.

The template relies on persistent volumes that are already provisioned for you in the classroom environment, and defines parameters for the MongoDB user and password.

```
[student@workstation ~]$ cd ~/coolstore/catalog-service
[student@workstation catalog-service]$ less \
  ocp/coolstore-catalog-mongodb-persistent.yaml
...
parameters:
- description: Catalog Service database user name
  from: user[a-zA-Z0-9]{3}
  generate: expression
  name: CATALOG_DB_USERNAME
  required: true
- description: Catalog Service database user password
  from: '[a-zA-Z0-9]{8}'
  generate: expression
  name: CATALOG_DB_PASSWORD
  required: true
- description: Catalog Service database name
  name: CATALOG_DATABASE
  required: true
  value: catalogdb
```

10.3. Deploy a containerized MongoDB database.

Use the **oc new-app** command and **mongodb** for the database user and password parameters:

```
[student@workstation catalog-service]$ oc new-app -f \
  ocp/coolstore-catalog-mongodb-persistent.yaml \
  -p CATALOG_DB_USERNAME=mongo -p CATALOG_DB_PASSWORD=mongo
...
--> Creating resources ...
  service "catalog-mongodb" created
  deploymentconfig "catalog-mongodb" created
```

```
persistentvolumeclaim "mongodb-data-pv" created
--> Success
...
```

10.4. Verify that the MongoDB database is operational.

Verify that the persistent volume claim is bound. The volume name you see may be different:

```
[student@workstation catalog-service]$ oc get pvc
NAME           STATUS    VOLUME   CAPACITY  ACCESSMODES ...
mongodb-data-pv  Bound     vol01    1Gi       RWO        ...
```

Wait until the database pod is ready and running:

```
[student@workstation catalog-service]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
catalog-mongodb-1-86m8k  1/1     Running   0          1m
```

Verify the database logs:

```
[student@workstation catalog-service]$ oc logs catalog-mongodb-1-86m8k
...
2018-04-05T18:46:03.919+0000 I NETWORK  [initandlisten] waiting for connections on
port 27017
...
```



Note

If you need to redeploy the MongoDB database, for example because you used incorrect parameter values, you need to reprovision the persistent volume. Perform the cleanup Step 14 and restart from Step 10

10.5. Authorize pods to access OpenShift resources.

Authorize access from pods to the OpenShift Master API by assigning the **view** role to the **default** service account:

```
[student@workstation catalog-service]$ oc policy add-role-to-user \
  view -z default
role "view" added: "default"
```

Verify that the **default** service account has the **view** role:

```
[student@workstation catalog-service]$ oc get rolebinding view
NAME      ROLE      USERS      GROUPS      SERVICE ACCOUNTS      SUBJECTS
view      /view                  default
```

10.6. Deploy the Catalog microservice.

Use the **fabric8:deploy** Maven goal. Skip tests to have a faster deployment. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation catalog-service]$ mvn fabric8:deploy -DskipTests
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ catalog-service ---
...
[INFO] --- vertx-maven-plugin:1.0.7:package (vmp) @ catalog-service ---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ catalog-service ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ catalog-service ---
...
[INFO] BUILD SUCCESS
...
```

- 10.7. Verify that the FMP created a configuration map that contains the expected database connection parameters:

```
[student@workstation catalog-service]$ oc describe configmap app-config
...
Data
=====
app-config.yaml:
-----
catalog.http.port: 8080
connection_string: mongodb://catalog-mongodb:27017
db_name: catalogdb
username: mongo
password: mongo
```

- 10.8. Verify that the Coolstore microservice is operational.

Wait until the Coolstore microservice pod is ready and running:

```
[student@workstation catalog-service]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
catalog-mongodb-1-86m8k   1/1     Running   0          1h
catalog-service-1-4w6lh    1/1     Running   0          6m
catalog-service-s2i-1-build 0/1     Completed  0          6m
```

Verify the microservice logs:

```
[student@workstation catalog-service]$ oc logs catalog-service-1-4w6lh
...
INFO: Succeeded in deploying verticle
...
INFO: Opened connection [connectionId{localValue:1, serverValue:711}] to catalog-
mongodb:27017
...
```

- 10.9. Invoke the HTTP API to fetch data for product 444435.

Get the host name of the route that was created by FMP:

```
[student@workstation catalog-service]$ oc get route
NAME          HOST/PORT
catalog-service  catalog-service.apps.lab.example.com  ...
```

Use the **curl** command to invoke the **/product/444435** resource URL:

```
[student@workstation catalog-service]$ curl -si \
  http://catalog-service.apps.lab.example.com/product/444435
HTTP/1.1 200 OK
Content-type: application/json
...
{
  "itemId" : "444435",
  "name" : "Oculus Rift",
  ...
}
```

► **11.** Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation aloha]$ lab catalog-deploy grade
```

► **12.** Commit your changes to your local Git repository:

```
[student@workstation coolstore]$ git add .
[student@workstation coolstore]$ git commit -m \
"Finished the Vert.x lab."
```

► **13.** Remove the **catalog-service** project from the IDE workspace.

► **14. Optional:** Clean up to redo the exercise from scratch.



Important

Later during this course, you will need the Catalog microservice to be fully operational. Do not delete the **catalog-service** project in OpenShift unless you really want to start over this exercise.

14.1. Perform this and the following steps only if you wish to start over this exercise.

Reset your local repository to the remote branch:

```
[student@workstation coolstore]$ git reset --hard \
  origin/do292-catalog-lab-begin
```

Delete the **catalog-service** project in OpenShift.

```
[student@workstation ~]$ oc delete project catalog-service
project "catalog-service" deleted
```

14.2. Delete the persistent volume for the database data.

Log in to OpenShift as the **admin** user and delete only the persistent volume that is in the **Released** state:

```
[student@workstation ~]$ oc login -u admin -p redhat \
  https://master.lab.example.com
Login successful.

...
[student@workstation ~]$ oc get pv
NAME      CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS    ...
registry-volume  10Gi        RWX          Retain       Bound    ...
vol01     1Gi         RWO          Retain       Released ...
vol02      1Gi         RWO          Retain       Available ...
[student@workstation ~]$ oc delete pv vol01
persistentvolume "vol01" deleted
```

14.3. Clean the persistent volume NFS share.

Open an SSH connection to the **services** VM, as the **root** user, and remove the contents of the volume folder under **/var(exports**. Do not remove the folder.

```
[student@workstation ~]$ ssh root@services rm -rf /var(exports/vol01/*
```

Verify that the persistent volume NFS share is empty:

```
[student@workstation ~]$ ssh root@services ls -la /var(exports/vol01
total 0
drwxrwxrwx. 2 nfsnobody nfsnobody 6 Abr 9 18:23 .
drwxr-xr-x. 4 root      root      32 Mar 21 15:52 ..
```

14.4. Recreate the persistent volume.

Use one of the resource definition files in the **coolstore/catalog-service** folder:

```
[student@workstation ~]$ oc create -f ~/coolstore/catalog-service/vol01-pv.yaml
persistentvolume "vol01" created
```

Verify that the volume is in the **Available** state:

```
[student@workstation ~]$ oc get pv
NAME      CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS    ...
registry-volume  10Gi        RWX          Retain       Bound    ...
vol01     1Gi         RWO          Retain       Available ...
vol02      1Gi         RWO          Retain       Available ...
```

This concludes the guided exercise.

Summary

In this chapter, you learned:

- Vert.x is an open source set of libraries maintained by the Eclipse Foundation for building highly scalable and low latency Reactive Systems. Red Hat supports an enterprise distribution of Vert.x as part of the Red Hat OpenShift Application Runtimes subscription.
- Vert.x APIs are based on asynchronous calls using event handlers, asynchronous results, and futures. Event handlers should run quickly and not block.
- A verticle is the unit of deployment in Vert.x. A verticle instance runs in a single thread to avoid the overhead of synchronization. Verticles communicate using the Vert.x event bus. The event bus supports clustered architectures and communication with non-Vert.x and non-Java applications.
- Vert.x event bus services allows programming interactions using the event bus as regular asynchronous calls. Event bus services employ static code generation, performed by a Maven annotation processor.
- Vert.x modules provide asynchronous and fluent APIs for database access, web services, REST API consumers, configuration, health checks, unit tests, and many other tasks. The Red Hat OpenShift Application Runtimes subscription supports a large number of Vert.x modules.
- The Vert.x Config and Health Check modules implement essential microservices features and provide integration with OpenShift features.
- The Vert.x Unit module provides support for writing JUnit tests that make asynchronous calls.
- To deploy a Vert.x microservice on OpenShift, use the Fabric8 Maven Plug-in together with the Vert.x Maven Plug-in.

Chapter 4

Developing Microservices with the Spring Boot Runtime

Goal

Describe the Spring Boot runtime and develop an application using Spring Boot

Objectives

- Describe the Spring Boot runtime and develop an application using Spring Boot.
- Configure a Maven project to build a Spring Boot application and deploy it on OpenShift.

Sections

- Developing an Application with the Spring Boot Runtime (and Guided Exercise)
- Configuring a Maven Project for Spring Boot (and Guided Exercise)

Lab

Developing Microservices with the Spring Boot Runtime

Developing an Application with the Spring Boot Runtime

Objectives

After completing this section, students should be able to describe the Spring Boot runtime and develop an application using Spring Boot.

Introducing the Spring Boot Framework

The *Spring Boot* framework makes it easy to create standalone applications based on the Spring framework. While traditional Spring framework applications are meant to be deployed into a web container such as Apache Tomcat, or an application server such as JBoss EAP, Spring Boot applications usually embed the web container or rely on other mechanisms, such as a messaging client or a reactive library, that sources events to the application.

Spring Boot builds upon the *Spring framework*, created as an alternative to the Enterprise Java Beans (EJB) programming models of early Enterprise Java standards, before EJB 3 and CDI existed. The Spring framework provides an *inversion of control (IoC)* container that supports the *dependency injection (DI)* programming model.

The Spring framework comprises a large ecosystem of libraries maintained by its corporate sponsor Pivotal and the larger community. Some of these libraries support standard Enterprise Java APIs, such as JPA, JMS, and JAX-RS, while others provide competing APIs, such as Spring Web. The Spring framework supports multiple implementations of the same APIs and does not prescribe an application architecture.

Spring Boot also relies heavily on the *Spring Cloud* libraries that implement common microservice architecture patterns. Spring Cloud started as a project to integrate Netflix Open Source Software (Netflix OSS) libraries into the Spring framework ecosystem and later grew to support alternative implementations of microservice architecture patterns.

Working with Spring Framework Annotations for Dependency Injection

Originally, the Spring framework employed XML configuration files to set up both application components, named *managed beans*, and infrastructure components.

Later releases of the Spring framework favor an annotation-based approach to dependency injection. To define a class as a managed bean, use the **@Component** annotation:

```
@org.springframework.stereotype.Component  
public class MyManagedBean {  
    ...
```

To inject a managed bean into a consumer class, use the **@Autowired** annotation:

```

@Component
public class AnotherManagedBean {

    @org.springframework.beans.factory.annotation.Autowired
    private MyManagedBean bean;
    ...
}

```

If you require custom logic to instantiate and initialize a managed bean, apply the **@Bean** annotation to a method, and annotate its class as **@Configuration**:

```

@org.springframework.context.annotation.Configuration
public class MyManagedBeanFactory {

    @org.springframework.context.annotation.Bean
    private MyManagedBean getManagedBean {
        MyManagedBean bean = new MyManagedBean();
        // ... set some bean attributes ...
        return bean;
    }
    ...
}

```

You can also use the Spring framework annotation-based configuration to inject the value of a Java system property using the **@Value** annotation. The following listing illustrates how to inject the value of the **com.example.my.configuration.parameter** system property with a default value of **defaultParameterValue**:

```

@Component
public class MyManagedBean {

    @org.springframework.beans.factory.annotation.Value(
        "${com.example.my.configuration.parameter:defaultParameterValue}"
    )
    String myConfigurationParameter;
    ...
}

```

There are many ways to provide system properties for a Spring Boot application. The most common ones are:

- Using the **java** command with the **-D** option
- Adding the **application.properties** file as an application resource in the default package

Describing Spring Boot Auto-configuration

Despite annotation-based dependency injection, a typical Spring framework XML configuration can still become very large, because of the need to configure infrastructure components such as web containers, data sources, and transaction managers.

Spring Boot favors convention over configuration, and automatically configures infrastructure components required by application managed beans, based on the dependencies available on the class path.

Spring Boot also makes use of Java system properties to customize its default configurations, so there are few instances where a developer needs to provide either XML configuration files or **@Configuration** classes.

Describing Spring Boot Starters

Spring Boot starters are Maven artifacts that provide default dependency graphs for common scenarios using Spring framework libraries. For example, using libraries for web applications, including web services, requires a web container.

Spring Boot starters are organized around features and may include one or more **@Configuration** classes to configure that feature. Among the starters provided by Spring Boot are:

- **spring-boot-starter-web**: starter to use Spring MVC with Apache Tomcat as the embedded web container
- **spring-boot-starter-data-jpa**: starter to use Spring Data JPA with Hibernate and JDBC data sources
- **spring-boot-starter-activemq**: starter to use JMS messaging with Apache ActiveMQ

A typical application uses more than one feature and so requires multiple starter dependencies in its POM configuration file.

Defining a Spring Boot Application Main Class

A Spring Boot application that is not deployed into an application server must provide a class that is annotated as **@SpringBootApplication** and defines a static **main** method that invokes the **SpringApplication.run** static method:

```
@org.springframework.boot.autoconfigure.SpringBootApplication
public class Application {

    public static void main(String[] args) {
        org.springframework.context.ApplicationContext ctx =
            org.springframework.boot.SpringApplication.run(
                Application.class, args);
    }
}
```

The **@Application** class is usually also a **@Configuration** class, but it is not required to be.

Defining HTTP API Endpoints with JAX-RS

Spring Boot applications can use many different approaches to implement HTTP APIs, including REST services. The most popular ones are:

- Spring Web MVC, a web application framework originally created to be an alternative to Apache Struts and Java Server Faces (JSF)
- JAX-RS, the standard Enterprise Java API for REST web services

A JAX-RS endpoint in a Spring Boot application looks exactly the same as in an Enterprise Java application or a MicroProfile application using WildFly Swarm. The only difference is the need to annotate the JAX-RS resource class as a **@Component**:

```

@Path("/api")
@Component
public class MyHttpApi {
    ...
}

```

The application POM needs to provide dependencies for a JAX-RS implementation and a web container. While Spring Boot comes with no suitable starter, most open source projects that provide a JAX-RS API provide a starter, such as the **org.apache.cxf:cxf-spring-boot-starter-jaxrs** Maven dependency from Apache CXF.

If your application uses JAX-RS and you want to consume or produce JSON data, you also need to add a JSON data binding library, such as Jackson, to your application. For example, to configure Jackson as the JSON data bind provider for JAX-RS, put the following code inside a **@Configuration** class, and add the **com.fasterxml.jackson.jaxrs:jackson-jaxrs-json-provider** dependency to your project's POM:

```

@Bean
public JacksonJsonProvider jsonProvider(ObjectMapper objectMapper) {
    JacksonJaxbJsonProvider provider = new JacksonJaxbJsonProvider();
    provider.setMapper(objectMapper);
    return provider;
}

```

Consuming HTTP APIs with RestTemplate

Spring Boot applications can use many different approaches to consume HTTP APIs and REST services. The most popular one is the **RestTemplate** class from Spring Web. The **RestTemplate** class provides a fluent API to make an HTTP request, and marshal the request response into a Java object.

The following listing illustrates how to consume data from the **http://localhost:8080/userProfile/{profileId}** endpoint as a **UserProfile** object:

```

RestTemplate restTemplate = new RestTemplate();
ResponseEntity<UserProfile> entity = restTemplate.getForEntity(
    "http://localhost:8080/userProfile/1234568", UserProfile.class);
UserProfile profile = entity.getBody();

```

To consume JSON data using **RestTemplate**, your application must include a JSON marshaller, for example the one provided by Jackson as the **com.fasterxml.jackson.core:jackson-databind** Maven dependency. In this case, there is no need to provide a **@Configuration** class.

To get the raw text data from the request response, just use the **String** class as the type parameter for the **ResponseEntity** generic class.

Implementing Health Probes Using the Spring Boot Actuator

The Spring Boot *Actuator* is a library that adds monitoring capabilities to a Spring Boot application. It can provide information such as performance metrics, environment settings, and health indicators using either HTTP or JMX.

Spring Boot Actuator itself uses Spring Web MVC, but your application is not required to. You can invoke methods from the Spring Boot Actuator managed beans and use their return data as the response for JAX-RS endpoints.

The **HealthEndpoint** managed bean aggregates health information from multiple providers. Each provider is a managed bean that implements the **HealthAggregator** interface. Simple applications do not need to provide a health provider and can rely on the providers that are built in Spring Boot, such as the **DiskSpaceHealthIndicator**. See the references at the end of this section for more information about coding and using Spring Boot Actuator health providers.

The following listing invokes the Spring Boot health actuator endpoint from a JAX-RS resource method and returns its **Health** return object as JSON data:

```
@Autowired
org.springframework.boot.actuate.endpoint.HealthEndpoint health;

@GET
@Path("/health")
@Produces(MediaType.APPLICATION_JSON)
public org.springframework.boot.actuate.health.Health getHealth() {
    return health.invoke();
}
```

The **HealthEndpoint.invoke** method returns a **Health** object that aggregates the health status of all registered health providers. The following listing shows sample output. Yours may be different because of the Spring Boot release and other dependencies embedded into your application:

```
{
    "status": "UP", ①
    "diskSpace": { ②
        "status": "UP",
        "total": 107361468416,
        "free": 100138319872,
        "threshold": 10485760
    }
}
```

- ① Overall health status. If any provider returns an status different than **UP**, the overall status is not **UP**.
- ② Status from the **DiskSpaceHealthIndicator** health provider.

Testing Spring Boot Applications

The **org.springframework.boot:spring-boot-starter-test** dependency is a Spring Boot starter that provides features to ease writing unit and integration tests for Spring Boot applications. It provides, among other features:

- A JUnit test runner that initializes the Spring framework IoC container
- An annotation that triggers Spring Boot automatic configuration, and connects to your application **@Configuration** classes
- A test profiles for a test case to provide, for example, an in-memory database different from the production database

The following listing illustrates how to declare a test case for a Spring Boot application:

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

@ActiveProfiles("test")
@RunWith(SpringRunner.class)
@SpringBootTest(
    webEnvironment = SpringApplication.WebEnvironment.RANDOM_PORT,
    classes = com.example.myapp.MyApplication.class
)
public class BonjourResourceTest {
    ...
}
```

Packaging and Running a Spring Boot Application

Spring Boot provides a Maven plug-in that packages the application and its dependencies in a Fat JAR archive. To execute the JAR generated by the plug-in, use the **java -jar** command. The plug-in configuration is described in the next section.



References

Spring framework web site

<https://spring.io/projects/spring-framework>

Spring Boot framework web site

<https://spring.io/projects/spring-boot>

Spring Cloud framework web site

<https://projects.spring.io/spring-cloud/>

Spring Boot Actuator reference documentation

<https://docs.spring.io/spring-boot/docs/1.5.14.BUILD-SNAPSHOT/reference/htmlsingle/#production-ready>

Spring framework documentation guide for RestTemplate

<https://spring.io/guides/gs/consuming-rest/>

► Guided Exercise

Developing an Application with the Spring Boot Runtime

In this exercise, you will develop a REST microservice with Spring Boot and JAX-RS.

Outcomes

You should be able to:

- Bootstrap Apache CXF as the JAX-RS implementation for a Spring Boot application.
- Wire managed beans and configuration data using Spring framework dependency injection.
- Implement an HTTP API client using Spring Web.
- Invoke the Spring Boot Actuator to provide health status.

To perform this exercise, you need access to:

- The **do292-bonjour-microservice-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **hello-microservices/bonjour** Maven project.
- The **org.springframework.boot:spring-boot-dependencies**, **org.apache.cxf:cxf-spring-boot-starter-jaxrs**, and **org.springframework.boot:spring-boot-actuator** Maven artifacts in the classroom Nexus proxy server.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab bonjour-microservice setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-bonjour-microservice-solution** of the **hello-microservices** Git repository.

- 1. Switch to the **do292-bonjour-microservice-begin** branch in the **hello-microservices** Git repository and import the **bonjour** project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-bonjour-microservice-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
do292-bonjour-microservice-begin
...
Switched to a new branch 'do292-bonjour-microservice-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/hello-microservices/bonjour** folder as a Maven project.
- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-bonjour-microservice-begin**.
- ▶ 2. Review the initial state of the **bonjour** project.

The project is composed of a number of source files and a Maven POM file. The main files define the Bonjour microservice with an HTTP API endpoint and a Spring framework managed bean:

BonjourResource.java

Class that defines the microservice's HTTP API and invokes the **BonjourService** managed bean

BonjourService.java

Interface that defines the Spring framework managed bean

BonjourServiceImpl.java

Class that implements the Spring framework managed bean

During this exercise you will complete the two files that define Java classes, and make no change to the file that defines a Java interface.

The **bonjour** project also provides a number of auxiliary files:

Application.java

Class that performs Spring framework initialization and also provides Spring framework configuration

BonjourHealthResource.java

Class that implements a health probe accessible as an HTTP API endpoint

BonjourResourceTest.java

Test the microservice's HTTP API by making HTTP requests and verifying the response text

During this exercise you will complete the health probe, and make no change to the remaining two auxiliary files.

2.1. Inspect the project's Maven POM file.

Open the **pom.xml** file from the **bonjour** project. Switch to the **pom.xml** tab.

Do not change the project POM during this exercise. The dependencies and plug-in configurations are explained in the next section.

2.2. Inspect the project main class.

Open the **Application** class from the **com.redhat.training.msa.hello** package.

Note that the class is annotated as both **@SpringBootApplication** and provides a **main** method that creates a Spring framework **ApplicationContext** object.

The class is also annotated as **@Configuration** and has a factory method annotated as **@Bean** that provides a **JacksonJsonProvider** managed bean as the JAX-RS marshaller for JSON.

► 3. Complete the Spring framework managed bean.

3.1. Annotate the **BonjourServiceImpl** class as a managed bean.

Open the **BonjourServiceImpl** class from the **com.redhat.training.msa.hello** package.

Add the **@Component** annotation to the class. Use the following listing as a reference:

```
...
import org.springframework.stereotype.Component;

@Component
public class BonjourServiceImpl implements BonjourService {
...
```

3.2. Annotate the **showLocation** attribute as configuration data.

Add the **@Value** annotation to the attribute, with an argument that refers to the **com.redhat.training.msa.hello.show-location** system property and a default value of **true**. Use the following listing as a reference:

```
...
import org.springframework.beans.factory.annotation.Value;

@Component
public class BonjourServiceImpl implements BonjourService {

    @Value("${com.redhat.training.msa.hello.show-location:true}")
    private boolean showLocation;
...
```

► 4. Complete the JAX-RS resource class.

4.1. Inspect the incomplete HTTP API resource class.

Open the **BonjourResource** class from the **com.redhat.training.msa.hello** package.

The JAX-RS annotations in the class bind the `/api/bonjour` resource URI to the `bonjour` method, using the HTTP GET method, and returning plain text data.

You need to complete the code to wire the `BonjourService` managed bean.

4.2. Annotate the `BonjourResource` class as a managed bean.

Add the `@Component` annotation to the class. Use the following listing as a reference:

```
...
import org.springframework.stereotype.Component;

@Path("/api")
@Component
public class BonjourResource {
...
}
```

4.3. Inject the `BonjourService` managed bean.

Add the `@Autowired` annotation to the `service` attribute. Use the following listing as a reference:

```
...
import org.springframework.beans.factory.annotation.Autowired;

@Path("/api")
@Component
public class BonjourResource {

    @Autowired
    private BonjourService service;
...
}
```

▶ 5. Run unit tests for the Bonjour HTTP API.

The test case does not use mocks, and tests both the `BonjourResource` and the `BonjourServiceImpl` classes.

5.1. Inspect the test case.

Open the `BonjourResourceTest` class from the `com.redhat.training.msa.hola` package.

The class define two methods:

- `invokeHello` that tests the Bonjour microservice's HTTP API.
- `invokeHealthCheck` that tests the Bonjour microservice's health probe.

Do not change the test methods.

Both test methods use the `RestTemplate` class from the Spring Web library to make HTTP API requests as a regular consumer of the microservice's API.

The test case uses the `@RunWith` and `@SpringBootTest` annotations to bootstrap the Spring framework and the Spring Boot runtime to perform the tests.

5.2. Run the `BonjourResourceTest` class as a JUnit test.

The **JUnit** view shows that the **invokeHello** test passed and the **invokeHealthCheck** failed. If you get a different output, review the changes you made since the beginning of this exercise.

► 6. Complete the microservice's health probe.

- 6.1. Inspect the health probe class.

Open the **BonjourHealthResource** class from the **com.redhat.training.msa.hello** package.

The JAX-RS annotations in the class bind the **/health** resource URI to the **getHealth** method, and return JSON data.

You need to complete the **getHealth** method implementation.

- 6.2. Invoke the Spring Boot Actuator **HealthEndpoint**.

Replace the existing **return null** statement with a statement that returns the result of **invoke** method from the **HealthEndpoint** managed bean. Use the following listing as a reference:

```
...
@GET
@Path("/health")
@Produces(MediaType.APPLICATION_JSON)
public Health getHealth() {
    return healthEndpoint.invoke();
}
...
```

- 6.3. Run the test case.

Run the **BonjourResourceTest** class from the **com.redhat.training.msa.hola** package as a JUnit test.

The **JUnit** view shows that the two tests passed. If you get a different output, review the changes you made since the previous test run.

► 7. Test the microservice from the command line.

- 7.1. Package the microservice as a Fat JAR.

Open a terminal window and enter the **~/hello-microservices/bonjour** folder. Then use the Maven **package** goal to generate the Fat JAR file. Skip tests for faster execution of the **mvn** command:

```
[student@workstation ~]$ cd ~/hello-microservices/bonjour
[student@workstation bonjour]$ mvn clean package -DskipTests
...
[INFO] BUILD SUCCESS
...
```

- 7.2. Start the microservice using default configuration properties.

Run the microservice using the **java** command and the Fat JAR file:

```
[student@workstation bonjour]$ java -jar target/bonjour-1.0.jar
...
2018-06-11 11:24:57.955 INFO 4080 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2018-06-11 11:24:57.958 INFO 4080 --- [           main]
c.redhat.training.msa.hello.Application : Started Application in 2.889 seconds
(JVM running for 3.241)
```

7.3. Send an HTTP request to the microservice.

Open another terminal window, and use the **curl** command to send an HTTP GET request to the microservice. Use **localhost:8080** as the host name and port, and the **/api/bonjour** resource URI:

```
[student@workstation ~]$ curl -si http://localhost:8080/api/bonjour
HTTP/1.1 200
...
Bonjour de localhost
```

Note that the output includes the HTTP server name.

7.4. Send an HTTP request to the microservice's health probe.

Use the **curl** command to send an HTTP GET request to the microservice's **/health** resource URI:

```
[student@workstation ~]$ curl -si http://localhost:8080/health
HTTP/1.1 200
...
{"status":"UP","diskSpace":
 {"status":"UP","total":107361468416,"free":100075352064,"threshold":10485760}}
[student@workstation hello-microservices]$
```

Switch to the terminal running the microservice and press **Ctrl+C** to stop the microservice.

7.5. Start the microservice setting configuration properties.

Run the microservice using the **java** with the **-D** option to provide a value of **false** to the **com.redhat.training.msa.hello.show-location** system property:

```
[student@workstation bonjour]$ java \
-Dcom.redhat.training.msa.hello.show-location=false \
-jar target/bonjour-1.0.jar
...
2018-06-11 11:29:51.123 INFO 4080 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2018-06-11 11:29:51.126 INFO 4080 --- [           main]
c.redhat.training.msa.hello.Application : Started Application in 2.889 seconds
(JVM running for 3.241)
```

7.6. Send an HTTP request to the microservice.

Run the same **curl** command again:

```
[student@workstation ~]$ curl -si http://localhost:8080/api/bonjour
HTTP/1.1 200
...
Bonjour
```

Note that the output does not include the HTTP server name.

Switch to the terminal running the microservice and press **Ctrl+C** to stop the microservice.

► **8.** Clean up.

- 8.1. Commit your changes to your local Git repository:

```
[student@workstation hello-microservices]$ git commit -a -m \
"Finished the exercise."
```

- 8.2. Remove the **bonjour** project from the IDE workspace.

- 8.3. If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation hello-microservices]$ git reset --hard \
origin/do292-bonjour-microservice-begin
```

This concludes the guided exercise.

Configuring a Maven Project for Spring Boot

Objectives

After completing this section, students should be able to configure a Maven project to build a Spring Boot application and deploy it on OpenShift.

Identifying Spring Boot Libraries Supported and Tested with Red Hat OpenShift Application Runtimes

Unlike the WildFly Swarm and Vert.x runtimes, Red Hat does not provide an enterprise distribution of Spring Boot in the JBoss Enterprise Maven repository. Red Hat certifies community Spring Boot releases and a subset of its dependencies from the community repositories as compatible with the Red Hat OpenShift Application Runtimes (RHOAR).

Red Hat does provide support according to the RHOAR subscription service-level agreement (SLA) for a few libraries where Red Hat maintains an active participation in the upstream development, and so is able to influence feature development plans and provide timely fixes to critical issues, for example:

- JPA using Hibernate
- JAX-RS using Apache CXF
- Web containers using Apache Tomcat
- Spring Cloud integration with Kubernetes

These artifacts must be downloaded from the JBoss Enterprise Maven repository to be eligible for production support.

For other Spring Boot libraries and dependencies certified for RHOAR, Red Hat only provides best effort support, because fixes depend on the upstream community, which is not tied to the RHOAR subscription SLA terms.

Spring Boot developers targeting OpenShift as the production environment are advised to evaluate the risk involved in using libraries from Spring Boot that are not tested for compatibility with RHOAR. The complete list of libraries that are either supported or certified are available in the product Release Notes.

Defining Dependencies for a Spring Boot Application

Red Hat recommends that the POM for a Spring Boot application uses the bill-of-materials (BOM) provided by the Spring Boot community as an imported POM. The following examples assume the POM defines system properties for all version strings, to ease maintenance:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${version.spring-boot}</version>
```

```

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

```

Most applications require at least the Spring Boot starter:

```

<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
...

```

If you use JAX-RS to implement an HTTP API, add the starter from the Apache CXF community. Notice that it is not part of the Spring Boot BOM, so you need to specify the Apache CXF starter version string. The following listing also includes a dependency for the Jackson JAX-RS JSON provider which is required by Apache CXF to produce or consume JSON data:

```

...
<dependency>
<groupId>org.apache.cxf</groupId>
<artifactId>cxf-spring-boot-starter-jaxrs</artifactId>
<version>${version.spring-boot.cxf-starter}</version>
<exclusions>
<exclusion>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>com.fasterxml.jackson.jaxrs</groupId>
<artifactId>jackson-jaxrs-json-provider</artifactId>
</dependency>
<dependencies>

```

The Apache CXF starter has a transitive dependency on the Sprint Web MVC library, but most applications that use JAX-RS do not actually require Spring Web MVC. The **<exclusions>** element in the previous listing makes sure your Fat JAR is not inflated with unnecessary dependencies.

If your application uses the Spring Web **RestTemplate** class, add dependencies for the Spring Web and Jackson libraries:

```

...
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
</dependency>
<dependency>
<groupId>com.fasterxml.jackson.core</groupId>

```

```

<artifactId>jackson-databind</artifactId>
</dependency>
<dependencies>
```

Any other Spring Boot, Spring Cloud, or Spring framework library you require needs to be added as an explicit dependencies, unless it is provided by another starter already defined inside your project's POM.

Configuring the Spring Boot Maven Plug-in

The following configuration binds the Spring Boot Maven Plug-in to the Maven life cycle so the **mvn package** command generates a Fat JAR:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${version.spring-boot}</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  ...

```

Configuring the Spring Boot Generator for the Fabric8 Maven Plug-in

Most Spring Boot applications do not require specific configurations for the Fabric8 Maven Plug-in. The Spring Boot generator picks the image stream name from system properties:

```

<properties>
  ...
  <fabric8.generator.fromMode>istag</fabric8.generator.fromMode>
  <fabric8.generator.from>redhat-openjdk18-openshift</fabric8.generator.from>
</properties>

<plugin>
  ...
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <phase>package</phase>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>

```

```

    </goals>
  </execution>
</executions>
<configuration>
  <generator>
    <config>
      <spring-boot>
        </spring-boot>
      </config>
    </generator>
  </configuration>
</plugin>

```

Fetching Configuration Data from OpenShift Configuration Maps

To make the Spring framework configuration features, such as the `@Value` annotation, fetch configuration data from OpenShift configuration maps, add the `org.springframework.cloud:spring-cloud-starter-kubernetes-config` dependency to your POM:

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
  <version>${version.spring-cloud.kubernetes}</version>
</dependency>

```

The configuration map name is provided by the application name defined by the `spring.application.name` system property. The following listing defines the application name inside `application.properties` file:

```
spring.application.name=myapp
```

Add the `view` role to your application service account, so its pods are authorized to invoke the OpenShift Master API to read the configuration map resource:

```
$ oc policy add-role-to-user view -z default
```

Demonstration: Specifying Spring Boot Dependencies

1. Open the POM file of the Bonjour microservice Maven project.
2. Note the reference to a parent POM file, and open the parent POM file.
3. In the parent POM file, note the following system properties (they are not contiguous in the file):

```

<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
...
<version.spring-boot>1.5.12.RELEASE</version.spring-boot>
<version.spring-boot.cxf-starter>3.1.12.redhat-1</version.spring-boot.cxf-starter>
<version.spring-cloud.kubernetes>0.2.0.RELEASE</version.spring-cloud.kubernetes>

```

4. In the **bonjour** Maven project POM file, note the reference to the **org.springframework.boot:spring-boot-dependencies** artifact:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${version.spring-boot}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

5. In **bonjour** Maven project POM file, note the references to Spring Boot starters and libraries, for example:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
...
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator</artifactId>
</dependency>
```

6. Note the reference to the Spring Cloud Apache CXF starter and the Jackson JAX-RS JSON provider:

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfr-spring-boot-starter-jaxrs</artifactId>
  <version>${version.spring-boot.cxf-starter}</version>
  <exclusions>
    <exclusion>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.jaxrs</groupId>
  <artifactId>jackson-jaxrs-json-provider</artifactId>
</dependency>
```

7. Note the reference to the Spring Cloud library that supports fetching configuration from Kubernetes configuration maps:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
  <version>${version.spring-cloud.kubernetes}</version>
</dependency>
```

8. Note the Spring Boot Maven Plug-in that packages the application as a Fat JAR:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <version>${version.spring-boot}</version>
  <executions>
    <execution>
      <goals>
        <goal>repackage</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

9. Finally, note the configuration of the Fabric8 Maven plug-in that specifies the Spring Boot generator:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  ...
  <configuration>
    <generator>
      <config>
        <spring-boot>
          </spring-boot>
        </config>
    </generator>
  </configuration>
</plugin>
```

10. Log in to OpenShift as the **developer** user and create the **bonjour** OpenShift project.
11. Give the project's default service account rights to access OpenShift resources.

```
$ oc policy add-role-to-user view -z default
```

12. Use Maven and the Fabric8 Maven Plug-in to deploy the project on OpenShift:

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

13. Inspect the route resource fragment file to see the host name to access the microservice.

14. Wait for the microservice pod to be ready and running, and test the microservice's API using the **curl** command:

```
$ curl -si http://bonjour.apps.lab.example.com/api/bonjour
```

This concludes this demonstration



References

JBoss Enterprise Maven Repository

<https://access.redhat.com/maven-repository>

Spring Cloud Kubernetes

<https://github.com/spring-cloud-incubator/spring-cloud-kubernetes/>

Further information about Spring Boot libraries is available in the *Tested and Verified Maven Artifacts Provided with Spring Boot* section of the *Spring Boot* chapter in the *Red Hat OpenShift Application Runtimes 1 Release Notes* at

https://access.redhat.com/documentation/en-us/red_hat_openshift_application_runtimes/1/html-single/red_hat_openshift_application_runtimes_release_notes/#maven-artifacts-spring-boot-tested-verified

► Guided Exercise

Configuring a Maven Project for Spring Boot

In this exercise, you will configure the Maven POM to deploy a simple microservice that uses Spring Boot with Red Hat OpenShift Application Runtimes (RHOAR).

Outcomes

You should be able to:

- Configure the project dependencies so Spring Boot gets configuration data from an OpenShift configuration map.
- Configure the Fabric8 Maven Plug-in (FMP) to deploy the microservice on OpenShift.

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-bonjour-deploy-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **hello-microservices/bonjour** Maven project.
- The **org.springframework.boot:spring-boot-dependencies**, **org.apache.cxf:cxf-spring-boot-starter-jaxrs**, and **org.springframework.boot:spring-boot-actuator** Maven artifacts in the classroom Nexus proxy server.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab bonjour-deploy setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-bonjour-deploy-solution** of the **hello-microservices** Git repository.

► 1. Switch to the **do292-bonjour-deploy-begin** branch in the **hello-microservices**

Git repository and import the **bonjour** project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-bonjour-deploy-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
do292-bonjour-deploy-begin
...
Switched to a new branch 'do292-bonjour-deploy-begin'
```

▶ 2. Inspect the starter **bonjour** project.

- 2.1. Inspect the parent POM.

Open the **~/hello-microservices/pom.xml** file using any text editor, and note the system properties that define the image stream name and the versions of the required dependencies and Maven plug-ins:

```
...
<version.fabric8.plugin>3.5.38</version.fabric8.plugin>
<fabric8.generator.fromMode>istag</fabric8.generator.fromMode>
<fabric8.generator.from>redhat-openjdk18-openshift</fabric8.generator.from>
...
<version.spring-boot>1.5.12.RELEASE</version.spring-boot>
<version.spring-boot.cxf-starter>3.1.12.redhat-1</version.spring-boot.cxf-
starter>
<version.spring-cloud.kubernetes>0.2.0.RELEASE</version.spring-
cloud.kubernetes>
...
```

Do not make any change and close the **~/hello-microservices/pom.xml** file.

- 2.2. Inspect the incomplete **bonjour** Maven project POM.

Open the **~/hello-microservices/bonjour/pom.xml** file using any text editor, and note:

- The reference to the parent POM:

```
...
<parent>
  <groupId>com.redhat.training.msa</groupId>
  <artifactId>msa-parent</artifactId>
  <version>1.0</version>
</parent>
...
```

- The **org.springframework.boot:spring-boot-dependencies** imported POM, referred in the **<dependencyManagement>** element. It has all dependencies for Spring Boot tested for the Red Hat OpenShift Application Runtimes:

```
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${version.spring-boot}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
...
```

- The Spring Boot starters:

```
...
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
...
```

- The Apache CXF Spring Cloud starter for JAX-RS and its Jackson JAX-RS dependency:

```
...
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxfr-spring-boot-starter-jaxrs</artifactId>
  <version>${version.spring-boot.cxf-starter}</version>
  <exclusions>
    <exclusion>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
```

```
<groupId>com.fasterxml.jackson.jaxrs</groupId>
<artifactId>jackson-jaxrs-json-provider</artifactId>
</dependency>
...
```

- The Spring Boot and Spring framework libraries required by the microservice, for example:

```
...
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator</artifactId>
</dependency>

...
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
</dependency>
...
```

- The Spring Boot Maven Plug-in:

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${version.spring-boot}</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  ...

```

- The incomplete Fabric8 Maven Plug-in (FMP) configuration:

```
...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  <executions>
  </executions>

```

```

<configuration>
</configuration>
</plugin>
...

```

Leave the `~/hello-microservices/bonjour/pom.xml` file open, because you will make changes to it during the next steps.

2.3. Review the configuration map resource fragment.

Open the `~/hello-microservices/bonjour/src/main/fabric8/configmap.yaml` file using any text editor, and verify that the `com.redhat.training.msa.hello.show-location` system property is set to `false`.

```

data:
  com.redhat.training.msa.hello.show-location: false
metadata:
  name: bonjour

```

2.4. Review the deployment configuration resource fragment.

Open the `~/hello-microservices/bonjour/src/main/fabric8/deployment.yml` file using any text editor, and note that the `/health` endpoint is used by both the liveness and readiness probes:

```

...
  containers:
    - livenessProbe:
        failureThreshold: 2
        httpGet:
          path: "/health"
          port: 8080
          scheme: HTTP
        initialDelaySeconds: 12
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 1
      readinessProbe:
        failureThreshold: 3
        httpGet:
          path: "/health"
          port: 8080
          scheme: HTTP
        initialDelaySeconds: 12
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 1
...

```

► 3. Verify that the project passes unit tests but does not deploy on OpenShift.

3.1. Package the Bonjour microservice to run its unit tests and prove that its code is complete.

Enter the **~/hello-microservices/bonjour** folder and run the **package** Maven goal. All unit test pass, and a Fat JAR is generated:

```
[student@workstation hello-microservices]$ cd bonjour
[student@workstation bonjour]$ mvn clean package
...
Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ bonjour ---
[INFO] Building jar: /home/student/github/hello-microservices/bonjour/target/
bonjour-1.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ bonjour
---
[INFO] -----
[INFO] BUILD SUCCESS
...
```

3.2. Log in to OpenShift and create the **bonjour** project:

```
[student@workstation bonjour]$ oc login -u developer -p redhat \
https://master.lab.example.com
Login successful.

...
[student@workstation bonjour]$ oc new-project bonjour
Now using project "bonjour" on server "https://master.lab.example.com:443".
...
```

3.3. Try to deploy the Bonjour microservice to OpenShift to prove that the FMP configuration is incomplete.

Run the **fabric8:deploy** Maven goal, and skip tests. The build is successful, but it creates no OpenShift resources:

```
[student@workstation bonjour]$ mvn fabric8:deploy -DskipTests
...
[INFO] <<< fabric8-maven-plugin:3.5.38:deploy (default-cli) < install @ bonjour
<<<
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ bonjour ---
[WARNING] F8: No such generated manifest file /home/student/github/hello-
microservices/bonjour/target/classes/META-INF/fabric8/openshift.yml for this
project so ignoring
[INFO] -----
[INFO] BUILD SUCCESS
...
```

3.4. Verify that the **bonjour** OpenShift project is still empty.

Run the **oc get all** command. It shows that there are no resources inside the OpenShift project:

```
[student@workstation bonjour]$ oc get all  
No resources found.
```

► 4. Complete the FMP configuration to deploy a Spring Boot application.

4.1. Add the Spring Cloud Kubernetes starter dependency.

Right after the **spring-boot-actuator** dependency, add the **spring-cloud-starter-kubernetes-config** dependency to the **~/hello-microservices/bonjour/pom.xml** file:

```
...  
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-actuator</artifactId>  
</dependency>  
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-kubernetes-config</artifactId>  
  <version>${version.spring-cloud.kubernetes}</version>  
</dependency>  
...
```

4.2. Add the Fabric8 Maven Plug-in (FMP) to the standard Maven life cycle phases.

Complete the **<executions>** element inside the FMP definition in the **~/hello-microservices/bonjour/pom.xml** file:

```
...  
<plugin>  
  <groupId>io.fabric8</groupId>  
  <artifactId>fabric8-maven-plugin</artifactId>  
  <version>${version.fabric8.plugin}</version>  
  <executions>  
    <execution>  
      <id>fmp</id>  
      <goals>  
        <goal>resource</goal>  
        <goal>build</goal>  
      </goals>  
    </execution>  
  </executions>  
  <configuration>  
  </configuration>  
</plugin>  
...
```

4.3. Configure the Fabric8 Maven Plug-in (FMP) to use the Spring Boot generator.

Complete the **<configuration>** element inside the FMP definition in the **~/hello-microservices/bonjour/pom.xml** file:

```

...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8.plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <generator>
      <config>
        <spring-boot>
        </spring-boot>
      </config>
    </generator>
  </configuration>
</plugin>
...

```

Save your changes to the `~/hello-microservices/bonjour/pom.xml` file before you continue to the next step.

► 5. Deploy the Bonjour microservice to OpenShift.

5.1. Authorize pods to access OpenShift resources.

Authorize access from pods to the OpenShift Master API by assigning the `view` role to the `default` service account:

```
[student@workstation bonjour]$ oc policy add-role-to-user \
  view -z default
role "view" added: "default"
```

5.2. Verify that pods are authorized to access OpenShift resources in the current project.

Verify that the `default` service account has the `view` role:

```
[student@workstation bonjour]$ oc get rolebinding view
NAME      ROLE      USERS      GROUPS      SERVICE ACCOUNTS      SUBJECTS
view      /view                  default
```

5.3. Generate the resources and deploy the microservice on OpenShift.

Run the `fabric8:deploy` Maven goal, and skip tests. The build is successful, and creates a few OpenShift resources:

```
[student@workstation bonjour]$ mvn fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ bonjour ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator springt-boot
...
[INFO] F8: Creating BuildServiceConfig bonjour-s2i for Source build
[INFO] F8: Creating ImageStream bonjour
[INFO] F8: Starting Build bonjour-s2i
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ bonjour ---
...
[INFO] Creating a Service from openshift.yml namespace bonjour name bonjour
...
[INFO] Creating a ConfigMap from openshift.yml namespace bonjour name bonjour
...
[INFO] Creating a DeploymentConfig from openshift.yml namespace bonjour name
bonjour
...
[INFO] Creating Route bonjour:bonjour host: bonjour.apps.lab.example.com
...
[INFO] BUILD SUCCESS
...
```

You can safely ignore any timeout warnings or errors caused by a **RejectedExecutionException**.

- 5.4. Verify that the OpenShift resources are created.

Run the **oc status** command. Your output may be a little different if your pods are not yet ready:

```
[student@workstation bonjour]$ oc status
In project bonjour on server https://master.lab.example.com:443

http://bonjour.apps.lab.example.com to pod port 8080 (svc/bonjour)
dc/bonjour deploys istag/bonjour:1.0 <-
bc/bonjour-s2i source builds uploaded code on openshift/redhat-openjdk18-
openshift:latest
deployment #1 deployed 2 minutes ago - 1 pod
...
```

- 5.5. Verify that an OpenShift configuration map was created with correct configuration data.

Run the **oc describe cm** command to verify that the system property is set to **false**.

```
[student@workstation bonjour]$ oc describe cm bonjour
...
Data
=====
com.redhat.training.msa.hello.show-location:
-----
false
...
```

► 6. Test the Bonjour microservice.

6.1. Wait for the microservice pod to be ready and running.

Repeat the **oc get pod** command until you get output similar to the following:

```
[student@workstation bonjour]$ oc get pod
NAME          READY   STATUS    ...
bonjour-1-5k5sw   1/1     Running   ...
bonjour-s2i-1-build   0/1     Completed  ...
```

6.2. Get the host name to access the microservice.

Use the **oc get route** command to get the host name of the route generated by the Fabric8 Maven Plug-in (FMP):

```
[student@workstation bonjour]$ oc get route
NAME        HOST/PORT           PATH      SERVICES    PORT    ...
bonjour     bonjour.apps.lab.example.com  bonjour    8080    ...
```

6.3. Invoke the Bonjour microservice API.

Use the host name from the previous step and the **curl** command to access the **/api/bonjour** resource URL:

```
[student@workstation bonjour]$ curl -si \
http://bonjour.apps.lab.example.com/api/bonjour
HTTP/1.1 200 OK
...
Bonjour
```

6.4. Invoke the Bonjour health endpoint.

Use the host name from the previous step and the **curl** command to access the **/health** resource URL and note that the Spring Cloud Kubernetes libraries add a health provider with information about the Kubernetes pod and its service account:

```
[student@workstation bonjour]$ curl -si \
http://bonjour.apps.lab.example.com/health
HTTP/1.1 200 OK
...
{"status": "UP", "diskSpace": {"status": "UP", "total": 10718543872, "free": 10179674112, "threshold": 10485760}, "refreshScope": {"status": "UP"}, "kubernetes": {}}
```

```
{"status":"UP","inside":true,"namespace":"bonjour","podName":"bonjour-1-5k5sw",
"podIp":"10.128.0.195","serviceAccount":"default","nodeName":"node2.lab.example
.com","hostIp":"172.25.250.12"}}
```

► 7. Clean up.

- 7.1. Delete the **bonjour** OpenShift project.

```
[student@workstation bonjour]$ oc delete project bonjour
project "bonjour" deleted
```

- 7.2. Commit your changes to your local Git repository:

```
[student@workstation bonjour]$ git commit -a -m \
"Finished the exercise."
```

- 7.3. Remove the **bonjour** project from the IDE workspace.

- 7.4. If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation bonjour]$ git reset --hard \
origin/do292-bonjour-deploy-begin
```

This concludes the guided exercise.

► Guided Exercise

Developing Microservices with the Spring Boot Runtime

In this exercise, you will complete the Cart microservice of the Coolstore application using the Spring Boot runtime.

Outcomes

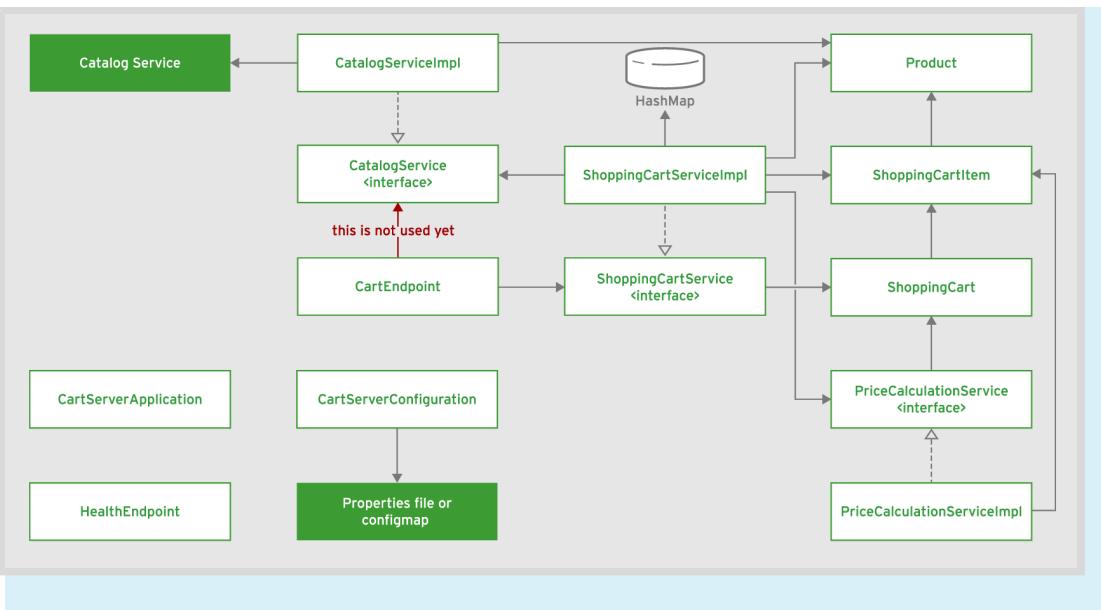
You should be able to:

- Wire managed beans using Spring Framework annotations.
- Implement an HTTP API client using Spring Web.
- Implement an HTTP API server using JAX-RS.
- Implement a unit test using REST Assured.
- Implement HTTP API endpoints for health checks using Spring Boot Actuator.
- Initialize configuration properties from OpenShift configuration maps.
- Deploy the microservice on OpenShift.

For each outcome, there is a branch that provides the code completed up to that point in the exercise. If you feel that you are taking too much time to complete a step, or you feel that you are stuck and need help, you can browse the solution code for each branch using the classroom GitWeb server. You can also switch to any of the intermediate branches and continue from that point, skipping previous steps.

The Cart Microservice

The Cart microservice is composed of a number of Spring Framework managed beans and plain old Java objects (POJOs):



To perform this exercise, you need access to:

- A running OpenShift cluster.
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-cart-lab-*** branches in the classroom Git repository, containing the source code for the microservice as part of the `coolstore/cart-service` Maven project.
- The **org.springframework.boot:spring-boot-dependencies**, **org.apache.cxf:cxf-spring-boot-starter-jaxrs**, and **org.springframework.cloud:spring-cloud-starter-kubernetes-config** Maven artifacts in the classroom Nexus proxy server.
- The Catalog microservice deployed and running in OpenShift. You need to perform Guided Exercise: Developing Microservices with the Vert.x Runtime starting from Step 10 before you are able to perform this guided exercise.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab cart-deploy setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-cart-lab-solution** of the **coolstore** Git repository.

- ▶ 1. Switch to the **do292-cart-lab-begin** branch in the **coolstore** Git repository and import the project into JBoss Developer Studio.
 - 1.1. If you have not done so yet, clone the **coolstore** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **coolstore** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/coolstore
Cloning into 'coolstore'...
...
```

- 1.2. If you already have a clone of the **coolstore** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-cart-lab-begin** branch:

```
[student@workstation ~]$ cd ~/coolstore
[student@workstation coolstore]$ git checkout do292-cart-lab-begin
...
Switched to a new branch 'do292-cart-lab-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/coolstore/cart-service** folder as a Maven project.
- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-cart-lab-begin**.
- 1.6. Ignore errors in the starter project.

The IDE builds the new **cart-service** project and reports syntax errors in the **com.redhat.coolstore.cart** and the **com.redhat.coolstore.cart.rest** packages. Ignore them for now. You will fix these errors as you proceed with this exercise.

- 2. Review the initial state of the **cart-service** project and wire managed beans into the **ShoppingCartServiceImpl** class.

The **ShoppingCartServiceImpl** class implements all logic required to manage shopping carts in memory. It uses the classes in the **com.redhat.coolstore.cart.model** package as a data model.

The **PriceCalculationServiceImpl** class calculates the total cost of the shopping cart.



Note

If you want to skip Step 2 and Step 3, stash your changes, check out the branch named **do292-cart-lab-client**, and continue from Step 4.

- 2.1. Inspect the Maven POM file.

Open the **pom.xml** file from the **cart-service** project. Switch to the **pom.xml** tab.

You will change this file during this exercise. For now, note the reference to a parent POM file that provides values for system properties. These properties are used to define the version strings for Maven artifacts required to build the microservice.

Also note the use of the POM imported by **org.springframework.boot:spring-boot-dependencies** to list supported

dependencies for the Spring Boot framework libraries and dependencies tested for compatibility with the Red Hat OpenShift Application Runtimes.

2.2. Inspect the **PriceCalculationServiceImpl** class.

Open the **PriceCalculationServiceImpl** class from the **com.redhat.coolstore.cart.service** package.

There is no need to make any changes to this class. It provides the **priceShoppingCart** method that calculates shipping costs for the shopping cart based on the total cost of the items in the shopping cart.

2.3. Inspect the unit tests for the **PriceCalculationServiceImpl** class.

Open the **PriceCalculationServiceImplTest** class from the **com.redhat.coolstore.cart.service** package.

There is no need to make any changes to this class. It is a standard JUnit test class using a few static methods from the **org.hamcrest** package as a convenience. This test class has no dependencies on the Spring framework.

2.4. Run the **PriceCalculationServiceImplTest** class as a JUnit test and ignore the errors in the workspace.

The **JUnit** view shows that all three tests passed.

2.5. Complete the **ShoppingCartServiceImpl** class.

Open the **ShoppingCartServiceImpl** class from the **com.redhat.coolstore.cart.service** package.

Add the **@Autowired** annotation to inject managed beans that implement the **CatalogService** and **PriceCalculationService** interfaces. Use the following listing as a reference for the changes to the **ShoppingCartServiceImpl.java** file:

```
...
import org.springframework.beans.factory.annotation.Autowired;
...

@Component
public class ShoppingCartServiceImpl implements ShoppingCartService {

    ...
    @Autowired
    private CatalogService catalogService;

    @Autowired
    private PriceCalculationService priceCalculationService;
    ...
}
```

2.6. Inspect the unit tests for the **ShoppingCartServiceImpl** class.

Open the **ShoppingCartServiceImplTest** class from the **com.redhat.coolstore.cart.service** package.

There is no need to make any changes to this class. The **ShoppingCartServiceImplTest** class uses the Mockito framework to mock implementations of the **CatalogService** and the **PriceCalculationService** interfaces, so all tests work even if you forget to complete the **ShoppingCartServiceImpl** class in the previous step.

The **ShoppingCartServiceImplTest** class also uses the Spring Framework **ReflectionTestUtils** utility class to inject the mocked services into the **ShoppingCartServiceImpl** object under test. So the tests for **ShoppingCartServiceImpl** work even if its dependent services are not working. This highlights the importance of designing and performing integration tests in addition to unit tests.

- 2.7. Run the **ShoppingCartServiceImplTest** class as a JUnit test and ignore the errors in the workspace.

The **JUnit** view shows that all tests passed.

► 3. Implement an HTTP API consumer.

The **CatalogServiceImpl** class encapsulates calls to the Catalog service to get product data such as unit price.

- 3.1. Configure the **CatalogServiceImpl** class as a Spring managed bean.

Open the **CatalogServiceImpl** class from the **com.redhat.coolstore.cart.service** package.

Add the **@Component** annotation to the class. Use the following listing as a reference:

```
...
import org.springframework.stereotype.Component;

@Component
public class CatalogServiceImpl implements CatalogService {
...
```

- 3.2. Inject the configuration value for the Catalog microservice URL.

Add the **@Value** annotation to the **catalogServiceUrl** attribute, referencing the **catalog.service.url** system property. Use the following listing as a reference:

```
...
import org.springframework.beans.factory.annotation.Value;

@Component
public class CatalogServiceImpl implements CatalogService {

    @Value("${catalog.service.url}")
    private String catalogServiceUrl;
```

- 3.3. Complete the **getProduct** method.

Instantiate a **RestTemplate** object that gets a **Product** object from the **/product/{id}** API entry point, and return the **Product** object. Catch the **HttpClientErrorException** exception, and return **null** in case the exception indicates HTTP status 404, otherwise throw the exception again.

Finally, remove the **return null** statement at the end of the method. Use the following listing as a reference:

```
...
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
```

```

import org.springframework.web.client.HttpClientErrorException;
...
@Override
public Product getProduct(String itemId) {
    RestTemplate restTemplate = new RestTemplate();
    try {
        ResponseEntity<Product> entity = restTemplate.getForEntity(
            catalogServiceUrl + "/product/" + itemId, Product.class);
        return entity.getBody();
    }
    catch (HttpClientErrorException ex) {
        if (ex.getRawStatusCode() == 404)
            return null;
        else
            throw (ex);
    }
}
...

```

3.4. Add the Jackson Databind dependency.

Open the **pom.xml** file from the **cart-service** project. Switch to the **pom.xml** tab. Just after the **org.springframework:spring-web** dependency, add the **com.fasterxml.jackson.core:jackson-databind** required by Spring **RestTemplate**. Use the following listing as a reference:

```

...
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
...

```

3.5. Inspect the unit tests for the **CatalogServiceImpl** class.

Open the **CatalogServiceImplTest** class from the **com.redhat.coolstore.cart.service** package.

There is no need to make any changes to this class. The **CatalogServiceImplTest** class tests the **CatalogServiceImpl** class without requiring a running Catalog microservice. It uses the WireMock framework to mock the microservice HTTP replies. It also uses Spring Framework **ReflectionTestUtils** utility class to inject the configuration value that points to the mocked Catalog microservice.

3.6. Run the **CatalogServiceImplTest** class as a JUnit test and ignore the errors in the workspace.

The **JUnit** view shows that all tests passed. If you get a different outcome, review the changes you made during this step and the previous one.

► 4. Implement the microservice's HTTP API.

The **CartEndpoint** class provides the entry points to add and remove products from the shopping cart. It uses JAX-RS and delegates all requests to the **ShoppingCartService** managed bean.



Note

If you want to skip Step 4, stash your changes, check out the branch named **do292-cart-lab-api**, and continue from Step 5.

- 4.1. Add the Apache CXF dependency to the POM.

Open the **pom.xml** file from the **cart-service** project. Switch to the **pom.xml** tab.

After the **com.fasterxml.jackson.core:jackson-databind** dependency, uncomment the **org.apache.cxf:cxf-spring-boot-starter-jaxrs** dependency. Note that the code that is commented out already excludes the Spring Web MVC dependency.

Also uncomment the **com.fasterxml.jackson.jaxrs:jackson-jaxrs-json-provider** dependency. Use the following listing as a reference:

```
...
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-spring-boot-starter-jaxrs</artifactId>
    <version>${version.spring-boot.cxf-starter}</version>
    <exclusions>
        <exclusion>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
    <artifactId>jackson-jaxrs-json-provider</artifactId>
</dependency>
...

```

When you save these changes to the project's POM, the IDE rebuilds the project and clears all errors in the project.

- 4.2. Inspect the JAX-RS configuration class.

Open the **CartServiceConfiguration** class from the **com.redhat.coolstore.cart** package.

You do not need to make any change to this file. It configures a **JacksonJaxbJsonProvider** object as the JAX-RS JSON marshaller.

- 4.3. Inspect the incomplete JAX-RS resource class.

Open the **CartEndpoint** class from the **com.redhat.coolstore.cart.rest** package.

Notice that the **CartEndpoint** class already contains JAX-RS annotations, and most of its methods are already configured as HTTP API end points.

4.4. Complete the **addToCart** method.

Annotate this method to answer to HTTP POST requests to the `/{{cartId}}/{{itemId}}/{{quantity}}` URI and return a JSON reply. Annotate the method parameters as appropriate. Invoke the **addToCart** method from the **ShoppingCartService** managed bean and use the return value as the JSON reply. Use the following listing as a reference:

```
...
@POST
@Path("/{cartId}/{itemId}/{quantity}")
@Produces(MediaType.APPLICATION_JSON)
public ShoppingCart addToCart(@PathParam("cartId") String cartId,
    @PathParam("itemId") String itemId,
    @PathParam("quantity") int quantity) {
    return shoppingCartService.addToCart(cartId, itemId, quantity);
}
...
```

4.5. Take a quick look at the unit tests for the **CartEndpoint** class.

Open the **CartEndpointTest** class from the `com.redhat.coolstore.cart.rest` package.

Do not make any changes to this class. Ignore, for now, that not all test methods are complete at this point. The **CartEndpointTest** class test is similar to previous tests in its use of the WireMock framework and the Spring Framework **ReflectionTestUtils** utility class.

Unlike previous tests, the **CartEndpointTest** class uses the REST Assured library to make HTTP requests to the **CartEndpoint** class and validate the responses.

4.6. Run the **CartEndpointTest** class as a JUnit test.

The **JUnit** view shows that four of five tests passed, and only **removeAllInstancesOfItemFromCart** failed. You will implement this test method during the next step. If you get a different outcome, review the changes you made during this step and the previous one.

► 5. Implement a unit test.

One of the tests for the **CartEndpoint** class is incomplete. It is your job to complete the **removeAllInstancesOfItemFromCart** method.



Note

If you want to skip Step 5, stash your changes, check out the branch named **do292-cart-lab-test**, and continue from Step 6.

5.1. Take a closer look to the **CartEndpointTest** class.

Open the **CartEndpointTest** class from the `com.redhat.coolstore.cart.rest` package.

Note that the **CartEndpointTest** class is annotated with `@RunWith`, `@SpringBootTest`, and `@ActiveProfiles`. It is coded like an integration

tests because it relies on Spring framework dependency injection to have the **CartEndpoint** managed bean injected with a **ShoppingCartServiceImpl** managed bean and its dependencies. Only external services are mocked.

5.2. Prepare test data for the **removeAllInstancesOfItemFromCart** method.

Use the **removeSomeInstancesOfItemFromCart** method as a reference for this and the next steps.

Use the REST Assured API to add two units of a product to a shopping cart, using an HTTP POST request. Use the following listing as a reference:

```
...
    @Test
    @DirtiesContext
    public void removeAllInstancesOfItemFromCart() throws Exception {

        given()
            .post("/{cartId}/{itemId}/{quantity}",
                  "456789", "111111", new Integer(2));
        fail("Not implemented yet.");
    }
...

```

5.3. Invoke the HTTP DELETE operation from the Cart microservice.

Use the REST Assured API to remove, using an HTTP DELETE request, all units of the product you just added from the same shopping cart. Use the following listing as a reference:

```
...
    @Test
    @DirtiesContext
    public void removeAllInstancesOfItemFromCart() throws Exception {

        given()
            .post("/{cartId}/{itemId}/{quantity}",
                  "456789", "111111", new Integer(2));
        given()
            .delete("/{cartId}/{itemId}/{quantity}",
                  "456789", "111111", new Integer(2))
        fail("Not implemented yet.");
    }
...

```

5.4. Verify that the HTTP DELETE request response is an empty shopping cart.

Use the REST Assured API to assert that the response from the HTTP DELETE request meets the following criteria:

- The HTTP status code is 200.
- The content-type is JSON.
- The JSON response contains an ID equal of the shopping cart from previous HTTP requests.

- The total price of the shopping cart is 0.
- The shopping cart has 0 items.

Finally, remove the call to **fail**. Use the following listing as a reference:

```
...
    @Test
    @DirtiesContext
    public void removeAllInstancesOfItemFromCart() throws Exception {

        given()
            .post("/{cartId}/{itemId}/{quantity}",
                  "456789", "111111", new Integer(2));
        given()
            .delete("/{cartId}/{itemId}/{quantity}",
                  "456789", "111111", new Integer(2))
            .then()
            .assertThat()
            .statusCode(200)
            .contentType(MediaType.APPLICATION_JSON)
            .body("id", equalTo("456789"))
            .body("cartItemTotal", equalTo(new Float(0.0)))
            .body("shoppingCartItemList", hasSize(0));
    }
...
}
```

5.5. Run the **CartEndpointTest** class as a JUnit test.

The **JUnit** view shows that all five tests passed. If you get a different outcome, review the changes you made during this step.

► 6. Implement a health endpoint.



Note

If you want to skip Step 6, stash your changes, check out the branch named **do292-cart-lab-health**, and continue from Step 8.

6.1. Add the Spring Boot Actuator dependency to the project.

Open the **pom.xml** file from the **cart-service** project. Switch to the **pom.xml** tab.

Right before the **org.springframework.boot:spring-boot-starter-test** dependency, add the **org.springframework.boot:spring-boot-actuator** dependency. Use the following listing as a reference:

```
...
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
```

```
<scope>test</scope>
</dependency>
...
```

6.2. Inspect the incomplete **HealthCheckEndpoint** class.

Open the **HealthCheckEndpoint** class from the **com.redhat.coolstore.cart.rest** package.

Notice that the **HealthCheckEndpoint** class already contains JAX-RS annotations to answer requests made to the **/health** entry point using the HTTP GET method.

6.3. Inject a **HealthEndpoint** object from Spring Actuator.

Use the following listing as a reference:

```
...
import org.springframework.boot.actuate.endpoint.HealthEndpoint;
...
@Path("/")
@Component
public class HealthCheckEndpoint {

    @Autowired
    private HealthEndpoint healthEndpoint;
...
}
```

6.4. Change the **getHealth** method return type to a **Health** object.

Use the following listing as a reference:

```
...
import org.springframework.boot.actuate.health.Health;
...
@Path("/")
@Component
public class HealthCheckEndpoint {

    @Autowired
    private HealthEndpoint healthEndpoint;

    public Health getHealth() {
        return null;
    }
...
}
```

6.5. Return the health status from Spring Actuator.

Call the **invoke** method from the **HealthEndpoint** object and use its return value as the response.

Use the following listing as a reference:

```
...
@Path("/")
@Component
public class HealthCheckEndpoint {
```

```

@Autowired
private HealthEndpoint healthEndpoint;

public Health getHealth() {
    return healthEndpoint.invoke();
}
...

```

- 6.6. Inspect the unit tests for the **HealthCheckEndpoint** class.

Open the **HealthCheckEndpointTest** class from the **com.redhat.coolstore.cart.rest** package.

The **HealthCheckEndpoint** class is very similar to the **CartEndpointTest** class tests and contains a single test method. Do not make any changes to the **HealthCheckEndpoint** class.

- 6.7. Run the **HealthCheckEndpointTest** class as a JUnit test.

The **JUnit** view shows that the only test passed. If you get a different outcome, review the changes you made during this step.

► 7. (Optional) Perform a manual test of the health endpoint.

- 7.1. Package the microservice as a Fat JAR:

Open a terminal window and use Maven to package the microservice:

```

[student@workstation ~]$ cd ~/coolstore/cart-service
[student@workstation cart-service]$ mvn clean package -DskipTests
...
[INFO] BUILD SUCCESS
...

```

- 7.2. Run the microservice.

Use the **java -jar** command to run the microservice from the fat JAR. You need to provide a value for the **catalog.service.url** system property, which can be empty:

```

[student@workstation cart-service]$ java -Dcatalog.service.url= \
    -jar target/cart-service-1.0.0-SNAPSHOT.jar
...
2018-05-30 11:25:32.515  INFO 11746 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2018-05-30 11:25:32.519  INFO 11746 --- [           main]
c.r.c.cart.CartServiceApplication      : Started CartServiceApplication in
3.006 seconds (JVM running for 3.343)

```

- 7.3. Invoke the health endpoint.

Open another terminal window and use the **curl** command:

```

[student@workstation ~]$ curl localhost:8080/health
{"status":"UP","diskSpace":{"status":"UP","total":107361468416,
"free":100067471360,"threshold":10485760}

```

Go back to the first terminal and press **Ctrl+C** to stop the **java** command.

► 8. Integrate Spring Configuration with OpenShift configuration maps.



Note

If you want to skip Step 8, stash your changes, check out the branch named **do292-cart-lab-solution**, and continue from Step 9.

8.1. Define an application name to be used to search for a configuration map.

Open the **application.properties** file in the **/src/main/resources** source folder. Switch to the **Source** tab.

Right after the **jaxrs.classes-scan-packages** entry, add the **spring.application.name** with a value of **cart-service**. Use the following listing as a reference:

```
cxf.path=/
cxf.jaxrs.component-scan=true
cxf.jaxrs.classes-scan-packages=com.redhat.coolstore.cart.rest
spring.application.name=cart-service
```

8.2. Inspect the configuration map resource fragment file.

Open the **configmap.yaml** file in the **/src/main/fabric8** source folder. Note the value of the **catalog.service.url** key and the name of the configuration map.

```
data:
  catalog.service.url: http://catalog-service.apps.lab.example.com
metadata:
  name: cart-service
```

8.3. Add the Spring Cloud Kubernetes Configuration Starter dependency to the project.

Open the **pom.xml** file from the **cart-service** project. Switch to the **pom.xml** tab.

Right before the **org.springframework.boot:spring-boot-actuator** dependency, add the **org.springframework.cloud:spring-cloud-starter-kubernetes-config** dependency. Use the following listing as a reference:

```
...
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
  <version>${version.spring-cloud.kubernetes}</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-actuator</artifactId>
</dependency>
...
```

8.4. Activate and configure the Fabric8 Maven Plug-in (FMP) to use the Spring Boot generator.

Inside the `<executions>` element of the FMP plug-in, uncomment the `<execution>` element that adds the FMP to the standard Maven life cycle.

Inside the `<configuration>` element of the FMP plug-in, add a `<generator>` element with a `<config>` element that references the `<sprint-boot>` generator.

Use the listing that follows as a reference:

```
...
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
  <executions>
    <execution>
      <id>fmp</id>
      <phase>package</phase>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <generator>
      <config>
        <spring-boot>
        </spring-boot>
      </config>
    </generator>
  </configuration>
</plugin>
```

▶ 9. Deploy the Catalog microservice to OpenShift and test the microservice API.

9.1. Log in to OpenShift and create a new project.

From a terminal window, log in to OpenShift as the **developer** user and create the **cart-service** project:

```
[student@workstation cart-service]$ oc login -u developer -p redhat \
  https://master.lab.example.com
Login successful.
...
[student@workstation cart-service]$ oc new-project cart-service
Now using project "cart-service" on server "https://master.lab.example.com:443".
...
```

9.2. Authorize pods to access OpenShift resources.

Authorize access from pods to the OpenShift Master API by assigning the **view** role to the **default** service account:

```
[student@workstation cart-service]$ oc policy add-role-to-user \
  view -z default
role "view" added: "default"
```

Verify that the **default** service account has the **view** role:

```
[student@workstation cart-service]$ oc get rolebinding view
NAME      ROLE      USERS      GROUPS      SERVICE ACCOUNTS      SUBJECTS
view      /view                default
```

9.3. Deploy the Cart microservice.

Use the **fabric8:deploy** Maven goal. Skip tests to have a faster deployment. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation cart-service]$ mvn fabric8:deploy -DskipTests
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ cart-service ---
...
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ cart-
service ---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ cart-service ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ cart-service ---
...
[INFO] BUILD SUCCESS
...
```

9.4. Verify that the FMP created a configuration map that contains the expected microservice URL parameter:

```
[student@workstation cart-service]$ oc describe cm cart-service
...
Data
=====
catalog.service.url:
-----
http://catalog-service.apps.lab.example.com
...
```

9.5. Verify that the Cart microservice is operational.

Wait until the Cart microservice pod is ready and running:

```
[student@workstation cart-service]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
cart-service-1-4w6lh   1/1     Running   0          6m
cart-service-s2i-1-build   0/1     Completed  0          6m
```

Verify the microservice logs:

```
[student@workstation cart-service]$ oc logs cart-service-1-4w6lh
...
2018-05-31 01:31:18.229 INFO 1 --- [           main]
c.r.c.cart.CartServiceApplication      : Started CartServiceApplication in
7.439 seconds (JVM running for 9.332)...
```

- 9.6. Invoke the HTTP API to create an empty shopping cart.

Get the host name of the route that was created by FMP:

```
[student@workstation cart-service]$ oc get route
NAME            HOST/PORT
catalog-service  cart-service.apps.lab.example.com ...
```

Use the **curl** command to invoke the **/cart/123456** resource URI:

```
[student@workstation cart-service]$ curl -si \
  http://cart-service.apps.lab.example.com/cart/123456
HTTP/1.1 200
...
>{"id":"123456","cartItemTotal":0.0,"shippingTotal":0.0,"cartTotal":0.0,
 "shoppingCartItemList":[]}
```

- 9.7. Invoke the HTTP API to add a product to the shopping cart.

Use the **curl** command to invoke the **/cart/123456** resource URI:

```
[student@workstation cart-service]$ curl -X POST -si \
  http://cart-service.apps.lab.example.com/cart/123456/444435/3
HTTP/1.1 200
...
>{"id":"123456","cartItemTotal":318.0,"shippingTotal":0.0,"cartTotal":318.0,
 "shoppingCartItemList":[{"price":106.0,"quantity":3,"product":
 {"itemId":"444435","name":"Oculus Rift","desc":"The world of gaming has also
 undergone some very unique and compelling tech advances in recent years. Virtual
 reality, the concept of complete immersion into a digital universe through a
 special headset, has been the white whale of gaming and digital technology ever
 since Nintendo marketed its Virtual Boy gaming system in 1995.","price":106.0}]}]
```

► 10. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation aloha]$ lab cart-deploy grade
```

► 11. Commit your changes to your local Git repository:

```
[student@workstation coolstore]$ git commit -a -m \
"Finished the exercise."
```

► 12. Remove the **cart-service** project from the IDE workspace.

► 13. **Optional:** Clean up to redo the exercise from scratch.



Important

Later during this course, you will need the Cart microservice to be fully operational.
Do not delete the **cart-service** project in OpenShift unless you really want to start over this exercise.

13.1. Perform this and the following steps only if you wish to start over this exercise.

Reset your local repository to the remote branch:

```
[student@workstation coolstore]$ git reset --hard \
origin/do292-cart-lab-begin
```

Delete the **cart-service** project in OpenShift.

```
[student@workstation ~]$ oc delete project cart-service
project "cart-service" deleted
```

This concludes the guided exercise.

Summary

In this chapter, you learned:

- Spring Boot makes it easy to create standalone applications, based on the Spring and the Spring Cloud frameworks, for an OpenShift cluster.
- Spring Boot applications are composed of managed beans that are connected using the **@Component** and **@Autowired** annotations.
- The **@Value** annotation injects values from system properties, and if the Spring Cloud Config Kubernetes libraries are in the class path, these system properties are initialized from OpenShift configuration maps.
- Spring Boot applications rely on auto-configuration for infrastructure components such as web containers. If you need explicit configuration, provide system properties and **@Configuration** classes.
- Spring Boot starters provide recommended dependency graphs for Spring Boot features such as web services, data access, and testing.
- Spring Boot applications based on the Red Hat OpenShift Application Runtimes use the bill-of-materials provided by the Spring Boot community.
- Red Hat certifies a subset of the libraries from a Spring Boot releases, and supports selected libraries such as JAX-RS with Apache CXF and JPA with Hibernate.
- The Sprint Boot Actuator provides health probes, and allows an application to define additional health providers as managed beans.
- The Spring Boot Maven Plug-in packages a Spring Boot application and its dependencies into a Fat JAR.

Chapter 5

Developing an API Gateway

Goal

Develop and deploy an API Gateway using the WildFly Swarm runtime.

Objectives

- Develop an API Gateway for multiple microservices.

Sections

- Developing an API Gateway for Microservices (and Guided Exercise)

Lab

Developing an API Gateway

Developing an API Gateway for Microservices

Objectives

After completing this section, students should be able to develop an API gateway for multiple microservices.

Introducing API Gateways

In a microservices-based architecture, you deploy a number of different microservices, each handling a specific business functionality or concern. A client or front end service that consumes these different microservices needs to keep track of the fine-grained APIs provided by each of the microservices.

Consider the scenario of an e-commerce shopping application like the one below. The client renders a user interface based on consuming different microservices for pricing, catalog, rating, shopping cart functionality, and more.

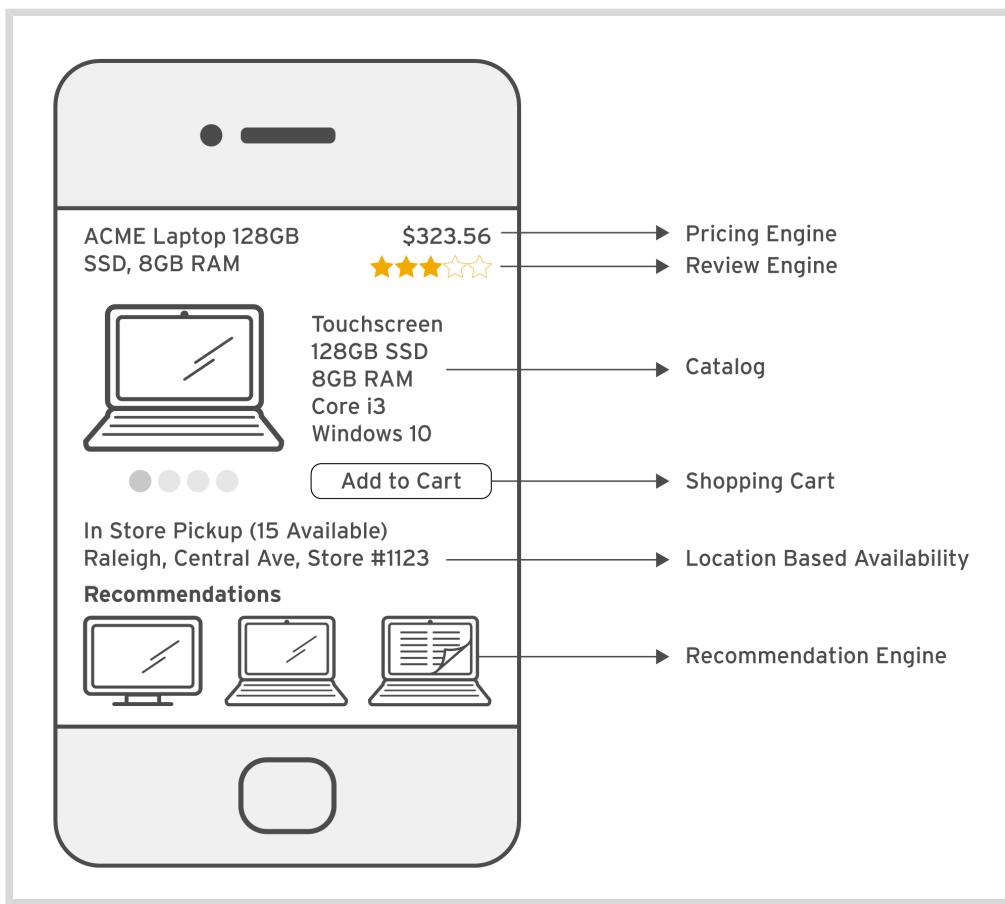


Figure 5.1: Client consuming different microservices

As the number of microservices that the client needs to keep track of increases, it becomes very difficult to develop and scale the application. There are several challenges with the approach of exposing microservices endpoints directly to clients:

- The client must track multiple endpoints, and it creates tight coupling between the microservice interfaces and the client.
- The client needs to be aware of implementation details of each microservice due to their fine-grained nature. Refactoring both client and microservices code becomes difficult and complex.
- Rendering a single page or view requires multiple round trips between the client and the microservices, increasing latency and reducing performance and scalability.
- Code related to common concerns like security, compliance auditing, logging, and request throttling are duplicated in all microservices.
- Not all microservices may expose endpoints compatible with the protocol that the client expects.
- It makes scaling the microservices in a transparent manner difficult because the client needs to keep track of numerous instances of the microservice. The client needs to embed intelligent routing logic to keep track of dynamically changing endpoints, resulting in increased code complexity.

You can use an *API gateway* to mitigate these challenges. An API gateway is a component in a microservices architecture that acts as a proxy between clients and the back-end microservices. It reduces the tight coupling between the client and the microservices and can also add functionality not implemented by the target microservices.

An API gateway can act as a single point of entry for all client requests, and provides a simpler, more generic API specific to the needs of a client. For example, you could have API gateways targeting web browser clients, and another API gateway for mobile clients.

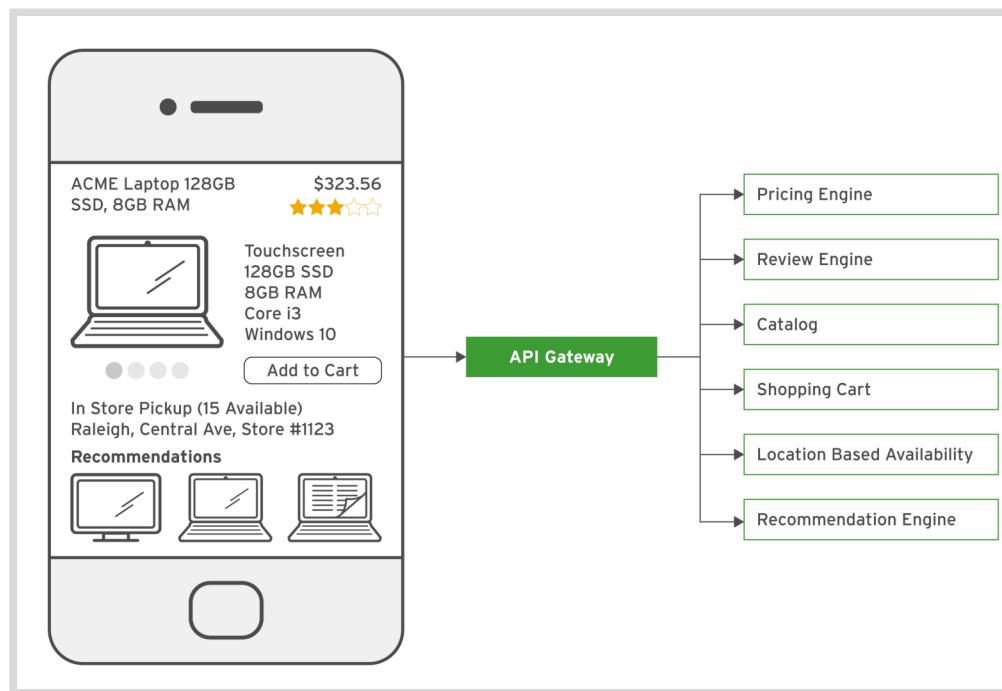


Figure 5.2: API gateway between clients and microservices

Advantages of API Gateways

The advantages of using an API gateway are:

- Hides the implementation details of microservices and exposes simpler and more generic APIs to clients
- Intelligent routing and protocol translation between client and microservices
- Centralized request logging
- Security: SSL termination, authentication, and authorization
- Prevention of Denial of Service (DoS) attacks and blacklisting malicious clients
- Request-rate limiting and throttling
- Automatic fallback response in scenarios where microservices performance is below a given threshold

Disadvantages of API Gateways

The disadvantages of using an API gateway are:

- The API gateway could become a single point of failure if it is not properly designed for redundancy and scalability
- Increased network latency because all client requests have to go through the API gateway to access the microservices
- Building, deploying, maintaining, monitoring, and managing an API gateway incurs extra cost overhead
- There is a risk of tight coupling between the API gateway code and the microservices. Changes to the microservices endpoint could cause cascading updates to the API gateway layer code

Implementing API Gateways

You can implement API gateways using a number of technologies, depending on the needs of your application. Some common implementations are:

General-Purpose Microservice Frameworks

You can develop HTTP-based and messaging-based API gateways using general-purpose frameworks and runtimes such as WildFly Swarm, Spring Boot, and Vert.x. These frameworks provide APIs for exposing HTTP APIs, consuming HTTP APIs, sending and receiving messages, processing data using multiple formats, and so on.

Red Hat Fuse

The Red Hat Fuse product offers a rich ecosystem of components to handle service orchestration, data transformation, protocol translation, content based routing, request limit throttling, and offers a flexible *domain specific language* (DSL) for composing REST APIs. Developing code to translate requests from one HTTP API to another HTTP API, and also from or to messaging APIs, is usually much easier using Apache Camel features, which are part of Red Hat Fuse, than using general-purpose frameworks and runtimes.

Red Hat 3scale

The Red Hat 3scale product is an enterprise-class, highly scalable API management solution providing advanced features for deploying API gateways. It provides features like rate limiting, billing, analytics, security and more.

Developing API Gateways with WildFly Swarm and RESTeasy

Consider a scenario where you are developing an API gateway for an e-commerce shopping application. Assume that the application is composed of two microservices:

1. TaxService: Calculates taxes for an item purchased by a customer.

This microservice exposes a single HTTP GET endpoint at: **http://server1.domain1.com/api/tax/{itemId}**

2. ShippingService: Calculates shipping costs for an item purchased by a customer.

This microservice also exposes a single HTTP GET endpoint at: **http://server2.domain2.com/api/shipping/{itemId}**

Both endpoints take an item ID as an input parameter, and return the tax amount and shipping costs, respectively. Note that both microservices are running on different servers on different domains.

Assume that you have a web user interface that uses the API gateway as the single point of entry for all back end services. The web user interface expects the API gateway to provide an HTTP GET endpoint at the location: **http://gateway.example.com/api/buy/{itemId}**

The API gateway endpoint also expects an item ID as input. It invokes the TaxService and the ShippingService microservices to calculate tax and shipping costs respectively, and provides a JSON response as follows:

```
{
  transactionId: 123456
  itemId: 56789
  taxAmount: 12.35
  shippingCost: 10.15
  totalCost: 22.50
}
```

Note that the code listing in the following sections are pseudo code meant to illustrate concepts and are not meant to be executed as is. Imports, data validation, and other best practices are omitted for brevity.

Coding RESTeasy Proxies

The JBoss community created the *RESTeasy* framework to provide an implementation of the standard JAX-RS APIs, and also to experiment with easier ways to develop HTTP API consumers and producers.

RESTeasy is able to generate HTTP API requests from a *proxy interface*, that uses standard JAX-RS to define an HTTP API consumer, while standard JAX-RS uses the same annotations to define an HTTP API producer. Coding HTTP API calls using RESTeasy proxy interfaces is easier than using the JAX-RS Client API **WebTarget** class alone.

The first step is to create proxy interface stubs for the TaxService and ShippingService microservices and method declarations with JAX-RS REST annotations for each of the endpoints that should be invoked from the API gateway.

For example, the proxy interface for the TaxService looks like the following:

```
public interface TaxServiceProxy {

    @GET
    @Path("/api/tax/{itemId}")
    public Double calculateTax(@PathParam("itemId")Integer itemId);

}
```

Similarly, the proxy interface for the ShippingService looks like the following:

```
public interface ShippingServiceProxy {

    @GET
    @Path("/api/shipping/{itemId}")
    public Double calculateShipping(@PathParam("itemId")Integer itemId);

}
```

The response from the API gateway can be encapsulated in the following simple Java class:

```
public class Invoice {
    private Integer transactionId;
    private Integer itemId;
    private Double taxAmount;
    private Double shippingCost;
    private Double totalCost;
    ...
    // setters and getters ommitted for brevity
}
```

Now that you have defined the proxy stubs for the microservices you want to invoke from the API gateway, use the RESTeasy framework **ResteasyWebTarget** class to create concrete instances of the proxy interfaces at run time. This approach provides a convenient way to work with plain Java objects and invoke the remote microservice endpoints as if they were running locally. The RESTeasy framework takes care of all the conversion to and from JSON.

Coding JAX-RS End Points

You can define the REST endpoint implementation for the API gateway as follows:

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
...

public class APIGatewayResource {

    private String taxServiceURL = "http://server1.domain1.com/"; ①
    private String shippingServiceURL = "http://server2.domain2.com/"; ②

    private TaxServiceProxy buildTaxClient() { ③
        Client client = ClientBuilder.newClient();
    }
}
```

```

        WebTarget target = client.target(taxServiceURL);
        ResteasyWebTarget restEasyTarget = (ResteasyWebTarget) target;
        return restEasyTarget.proxy(TaxServiceProxy.class);
    }

    private ShippingServiceProxy buildShippingClient() { ④
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(shippingServiceURL);
        ResteasyWebTarget restEasyTarget = (ResteasyWebTarget) target;
        return restEasyTarget.proxy(ShippingServiceProxy.class);
    }

    @GET
    @Path("/api/buy{itemId}")
    public Invoice buyItem(@PathParam("itemId") Integer itemId) {
        TaxServiceProxy taxClient = buildTaxClient();
        ShippingServiceProxy shipClient = buildShippingClient();

        Invoice invoice = new Invoice();
        Double taxAmount = taxClient.calculateTax(itemId); ⑤
        Double shippingCost = shipClient.calculateShipping(itemId); ⑥
        invoice.setTaxAmount(taxAmount);
        invoice.setShippingCost(shippingCost);
        invoice.setTotalCost(taxAmount + shippingCost);
        ...
        // set other attributes on invoice object

        return invoice; ⑦
    }
}

```

- ① The URL of the TaxService microservice
- ② The URL of the ShippingService microservice
- ③ Build a proxy client instance of the TaxServiceProxy interface
- ④ Build a proxy client instance of the ShippingServiceProxy interface
- ⑤ Invoke the TaxService microservice
- ⑥ Invoke the ShippingService microservice
- ⑦ Return the invoice object (renders an object in JSON format to clients)

Handling Cross Origin Resource Sharing (CORS)

Web-based user interfaces are usually developed as a set of static HTML, CSS, and JavaScript files that are downloaded, processed, and executed by a web browser. To avoid security concerns related to downloading static files from multiple sources, modern web browsers enforce the *same-origin* policy.

By the same-origin policy, JavaScript code executed by a web browser can only send HTTP requests to the same DNS domain that provided the JavaScript code. This avoids user data, such as a web form, to embed JavaScript calls to malware in a different domain.

If you need to make your HTTP API available to browser-based JavaScript applications, your HTTP API needs to enable cross origin resource sharing (CORS) to allow the browser clients to request resources from your HTTP API.

It is a common scenario to use an API gateway to add CORS headers because server-side applications, such as most microservices, do not implement the same-origin policy.

You enable CORS by setting some HTTP extra headers in the response from the API gateway. For JAX-RS based REST API implementations, create a filter class by extending the **javax.ws.rs.container.ContainerResponseFilter** class and add the required CORS headers to the response.

A sample filter class looks like the following:

```
...
@Provider
public class CORSFilter implements ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext requestContext,
    ContainerResponseContext responseContext) throws IOException {
        ...
        responseContext.getHeaders().add("Access-Control-Allow-Origin", "*"); ①
        responseContext.getHeaders().add("Access-Control-Allow-Methods", "GET,
    POST, PUT, DELETE, OPTIONS, HEAD"); ②
        ...
    }
}
```

- ① Allow access to resources from all origins. You can also specify the full domain name to restrict access to clients from the specified domain.
- ② Allow clients to request the methods provided in this list.

The full list of CORS headers that can be added to the response is provided in the references section.

Testing the API Gateway

Writing unit tests for API gateways is challenging in the sense that, unlike testing REST APIs that are standalone and can be tested in isolation, API gateways depend on other microservices, which may not be running on the same server as the gateway.

Good unit test practice recommends writing tests in a manner independent of external services. Many external services have strict control over the amount of requests you can make in a certain period (rate limiting) and you cannot guarantee that the external services will be available every time you run the unit tests. Degraded performance of these external services could cause cascading failures in the API gateway and provide misleading test results.

One approach to solving this problem is to *mock* the external microservice using a framework like *WireMock*.

WireMock is a framework to simulate HTTP based APIs. You can mock an endpoint as part of your test set up and run the unit tests against the mock service instead of the real external service.

You can mock an HTTP endpoint as follows:

```
...
@Rule
public WireMockRule taxServiceMockRule = new
WireMockRule(options().port(8888)); ①
```

```

@Rule
public WireMockRule shipServiceMockRule = new
WireMockRule(options().port(9999)); ②

@Test
@RunAsClient
public void testBuyItem() {

    taxServiceMockRule.stubFor(get(urlMatching("/api/tax/[0-9]+"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "application/json")
            .withBodyFile("tax-response.json")
        )); ④

    shipServiceMockRule.stubFor(get(urlMatching("/api/shipping/[0-9]+"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "application/json")
            .withBodyFile("shipping-response.json")
        )); ⑥

    WebTarget target = client
        .target("http://localhost:8080")
        .path("/api")
        .path("/buy")
        .path("/123456"); ⑦

    Response response = target.request(MediaType.APPLICATION_JSON).get();
    assertThat(response.getStatus(), equalTo(new Integer(200)));
    // assert other attributes from the response
    ...
}

```

- ① Define a new WireMock rule. Mock TaxService runs on port 8888 during test.
- ② Define a new WireMock rule. Mock ShippingService runs on port 9999 during test.
- ③ Wildcard mapping of resource URL for TaxService.
- ④ Wildcard mapping of resource URL for ShippingService.
- ⑤ Mock service returns a sample response in JSON format from the file **tax-response.json**. This file is stored in the **src/test/resources** folder in the Maven project for the API gateway.
- ⑥ Mock service returns a sample response in JSON format from the file **shipping-response.json**. This file is stored in the **src/test/resources** folder in the Maven project for the API gateway.
- ⑦ Invoke the API gateway endpoint and assert attributes in the resulting JSON response.

WireMock offers many more options to mock any type of REST API. Refer to the documentation which is provided in the references section.



References

The API Gateway Pattern

<http://microservices.io/patterns/apigateway.html>

Cross-Origin Resource Sharing (CORS)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

RESTeasy framework

<https://resteasy.github.io/>

WireMock framework

<http://wiremock.org/>

Further information about Red Hat Fuse is available at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/

Further information about Red Hat 3scale is available at

https://access.redhat.com/documentation/en-us/red_hat_3scale/

► Guided Exercise

Developing an API Gateway for Microservices

In this exercise, you will develop an API gateway for two separate microservices deployed on OpenShift.

Outcomes

You should be able to:

- Develop an API gateway using the WildFly Swarm runtime and deploy it on OpenShift.
- Map resource URIs to two different microservices deployed on OpenShift.
- Add cross origin resource-sharing headers to allow web browser clients to invoke the API gateway.

To perform this exercise, you need access to:

- A running OpenShift cluster
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-hello-gw-*** branches in the classroom Git repository, containing the source code for the API gateway, Bonjour, and Hola microservices
- The **org.wildfly.swarm:bom**, **io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts on the classroom Nexus proxy server

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab hello-gateway setup
```

You can compare your source code changes while performing this exercise with the solution branch **do292-hello-gw-solution** of the **hello-microservices** repository.

- 1. Switch to the **do292-hello-gw-begin** branch in the **hello-microservices** Git repository and import the **hello-api-gateway** project into JBoss Developer Studio.
 - 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-hello-gw-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
  do292-hello-gw-begin
...
Switched to a new branch 'do292-hello-gw-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/hello-microservices/api-gateway** folder as a Maven project.
 - 1.5. If you imported the project before, refresh the project to update its configuration. Make sure the project now displays the correct branch name, which is **do292-hello-gw-begin**.
 - ▶ 2. Complete the proxy interface for the **Bonjour** microservice.
- The API gateway handles all client requests, and routes requests to either the **Bonjour** or **Hola** microservice depending on the language code in the URI. The API gateway maps requests from clients to microservices in the following manner:
- HTTP GET requests to the path **/api/es** are sent to the **Hola** microservice at **/api/hola**
 - HTTP GET requests to the path **/api/fr** are sent to the **Bonjour** microservice at **/api/bonjour**
- The complete code for the two microservices is provided to you as part of the **bonjour** and **hola** Maven projects. Do not import these projects into the IDE, or make any changes to them. They are ready for deployment on OpenShift.
- 2.1. Inspect the proxy interface for the **Hola** microservice.
- Open the **HolaProxy** interface from the **com.redhat.training.msa.gateway.proxy** package in the IDE. This interface is complete. Do not make any changes to it.
- The interface has a single method annotated with the URI of the external **Hola** microservice. Note that the **@Path** annotation does not contain the complete URL, and uses a relative path to avoid hard coding the host name of the microservice.
- 2.2. Using the code in the **HolaProxy** interface as a reference, complete the **bonjour()** method in the **BonjourProxy** interface from the **com.redhat.training.msa.gateway.proxy** package.
- The **bonjour()** method proxies HTTP GET requests to the **/bonjour** path.

Add the following code to the **bonjour()** method:

```
...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
...
@Path("bonjour")
@Produces("text/plain")
@GET
public String bonjour();
```

- 3. Complete the API gateway code that proxies requests for the **Bonjour** microservice.

The API gateway uses the **RESTEasy** framework for invoking the REST API provided by the **Bonjour** and **Hola** microservices. The REST API exposed by the API gateway is implemented in the **APIGatewayResource** class.

- 3.1. Inspect the configuration map that configures the gateway.

The configuration map is provided as fabric8 resource fragments. Open the **/src/main/fabric8/cm.yml** file and verify that the URLs for the **Hola** and **Bonjour** microservices are provided:

```
...
kind: ConfigMap
data:
  HOLASERVICE_URL: http://hola-hello-gateway.apps.lab.example.com/api
  BONJOURSERVICE_URL: http://bonjour-hello-gateway.apps.lab.example.com/api
...
```

- 3.2. Inspect the deployment configuration resource fragment for the API gateway.

Open the **/src/main/fabric8/deployment.yml** file and verify that the configuration map values are mapped to environment variables in the deployment configuration:

```
...
containers:
- env:
  - name: HOLASERVICE_URL
    valueFrom:
      configMapKeyRef:
        key: HOLASERVICE_URL
        name: app-config
  - name: BONJOURSERVICE_URL
    valueFrom:
      configMapKeyRef:
        key: BONJOURSERVICE_URL
        name: app-config
...

```

- 3.3. Inspect the REST API implementation for the API gateway.

Open the **APIGatewayResource** class from the **com.redhat.training.msa.gateway.api** package.

Inspect the **buildHolaProxy()** and **buildBonjourProxy()** methods. These methods use the **RESTEasy** framework API to create proxy instances for the **Hola** and **Bonjour** microservices respectively.

Observe how the URLs for the microservices are injected into this class using the MicroProfile configuration API. The **HOLA_SERVICE_URL** and **BONJOUR_SERVICE_URL** properties are configured and managed using the OpenShift configuration map defined in the previous step to avoid hard coding the locations where the microservices are running.

- 3.4. Inspect the **hola()** method. The **hola()** method handles HTTP GET requests to the **/api/es** path and proxies the call to the **Hola** microservice using the **HolaProxy** class.
- 3.5. Using the code in the **hola()** method as a reference, complete the implementation of the **bonjour()** method.

The **bonjour()** method handles HTTP GET requests to the **/api/fr** path.

The **bonjour()** method should be as follows:

```
@GET
@Path("/fr")
@Produces("text/plain")
public String bonjour() {
    BonjourProxy proxy = buildBonjourProxy();
    String response = proxy.bonjour();

    return response;
}
```

- 4. Complete the code in the **CORSFilter** class to allow requests from all clients.

- 4.1. Open the **CORSFilter** class from the **com.redhat.training.msa.gateway.api** package.

Add headers to allow clients from all locations in the **filter()** method.

Add the following line in the **filter()** method:

```
...
public void filter(ContainerRequestContext requestContext,
    ContainerResponseContext responseContext) throws IOException {
    responseContext.getHeaders().add("Access-Control-Allow-Origin", "*");
    responseContext.getHeaders().add("Access-Control-Allow-Headers", "origin,
        content-type, accept, authorization");
...
}
```

- 5. Run the unit tests for the API gateway.

- 5.1. Inspect the unit tests for the **APIGatewayResource** class.

Open the **APIGatewayResourceTest** class from the **com.redhat.training.msa.gateway.api** package.

There is no need to make changes to this class. The **APIGatewayResourceTest** class uses the Mockito framework to mock implementations of the **Hola** and the **Bonjour** microservices.

Inspect the implementation of the tests in the **testHolaProxy()** and **testBonjourProxy** methods. Observe how the REST API for the **Hola** and **Bonjour** microservices are mocked to return valid HTTP status code, content types, and responses. The body of the response from the mocked methods are retrieved from the plain text files stored in the **/src/test/resources/_files** folder.

- 5.2. Run the **APIGatewayResourceTest** class as a JUnit test.

Ignore the **ArtifactNotFoundException** errors in the **Console** view. These are generated by Arquillian and do not affect the test outcomes.

The **JUnit** view shows that both tests passed.

► 6. Deploy the API gateway, Hola and Bonjour microservices to OpenShift.

- 6.1. Log in to OpenShift and create a new project.

From a terminal window, log in to OpenShift as the **developer** user and create the **hello-gateway** project:

```
[student@workstation hello-microservices]$ oc login -u developer -p redhat \
https://master.lab.example.com
Login successful.

...
[student@workstation hello-microservices]$ oc new-project hello-gateway
Now using project "hello-gateway" on server "https://master.lab.example.com:443".
...
```

- 6.2. Deploy the Bonjour microservice.

Use the **fabric8:deploy** Maven goal. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation hello-microservices]$ cd ~/hello-microservices/bonjour
[student@workstation bonjour]$ mvn clean fabric8:deploy -DskipTests
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ bonjour ---
...
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ bonjour
---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ bonjour ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ bonjour ---
...
[INFO] BUILD SUCCESS
...
```

- 6.3. Deploy the Hola microservice.

Use the **fabric8:deploy** Maven goal. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation hello-microservices]$ cd ~/hello-microservices/hola
[student@workstation hola]$ mvn clean fabric8:deploy -DskipTests
...
[INFO] --- maven-war-plugin:2.5:war (default-war) @ hola ---
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ hola ---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:build (default) @ hola ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ hola ---
...
[INFO] BUILD SUCCESS
...
```

6.4. Deploy the API gateway microservice.

Use the **fabric8:deploy** Maven goal. Skip tests to have a faster deployment. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation hello-microservices]$ cd ~/hello-microservices/api-gateway
[student@workstation api-gateway]$ mvn clean fabric8:deploy -DskipTests
...
[INFO] --- maven-war-plugin:2.5:war (default-war) @ hello-api-gateway ---
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ hello-api-
gateway ---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.38:build (default) @ hello-api-gateway ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ hello-api-gateway
---
...
[INFO] BUILD SUCCESS
...
```

6.5. It may take some time for all three microservices to be built and deployed on OpenShift. Verify that all the three microservice pods are ready and running:

```
[student@workstation api-gateway]$ oc get pod
NAME          READY   STATUS    ...
...
bonjour-1-6szrq      1/1     Running   ...
...
hello-api-gateway-1-vltcq  1/1     Running   ...
...
hola-1-x4dg2        1/1     Running   ...
...
```

► 7. Test the API gateway.

7.1. Invoke the API gateway REST API to test the Hola microservice.

Use the **curl** command to invoke the **/api/es** resource URL:

```
[student@workstation cart-service]$ curl -si \
  http://gw.apps.lab.example.com/api/es
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: *
...
Hola de hola-hello-gateway.apps.lab.example.com
Value in property 'a' => Tres
Value in property 'b' => Cuatro
```

- 7.2. Invoke the API gateway REST API to test the Bonjour microservice.

Use the **curl** command to invoke the **/api/fr** resource URL:

```
[student@workstation cart-service]$ curl -si \
  http://gw.apps.lab.example.com/api/fr
HTTP/1.1 200 OK
...
Access-Control-Allow-Origin: *
...
Bonjour de bonjour-hello-gateway.apps.lab.example.com
```

► 8. Clean up.

- 8.1. Delete the **hello-gateway** OpenShift project.

```
[student@workstation api-gateway]$ oc delete project hello-gateway
project "api-gateway" deleted
```

- 8.2. Commit your changes to your local Git repository:

```
[student@workstation bonjour]$ git commit -a -m \
  "Finished the API Gateway exercise."
```

- 8.3. Remove the **api-gateway** project from the IDE workspace.

- 8.4. If you wish to start this exercise over, reset your local repository to the remote branch:

```
[student@workstation api-gateway]$ git reset --hard \
  origin/do292-hello-gw-begin
```

This concludes the guided exercise.

► Lab

Developing an API Gateway

Performance Checklist

In this lab, you will develop an API gateway for the Coolstore application.

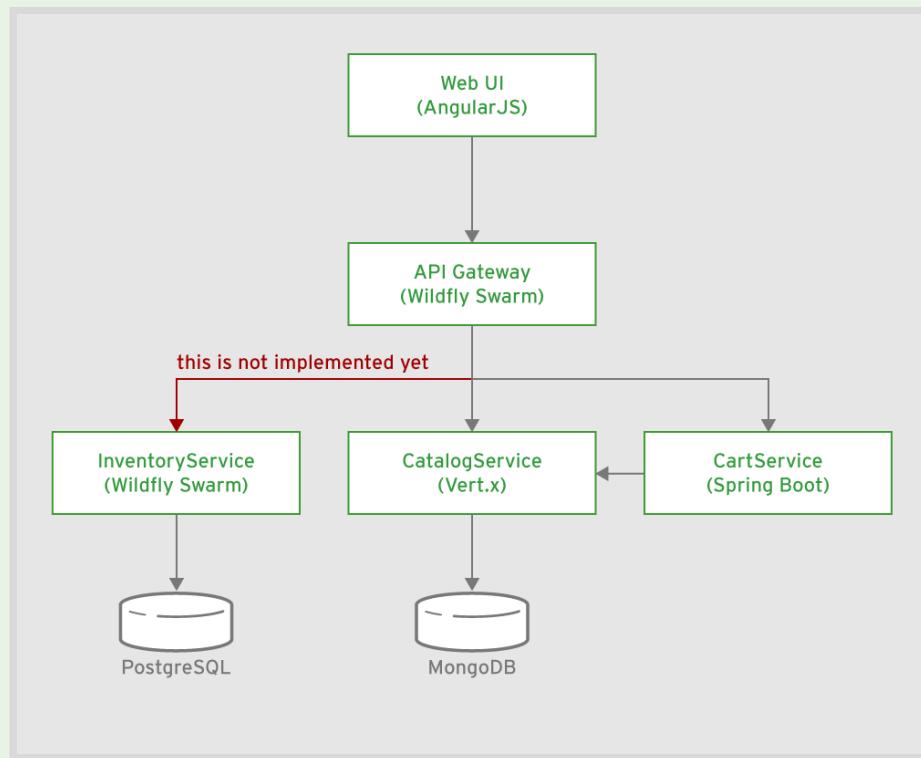
Outcomes

You should be able to:

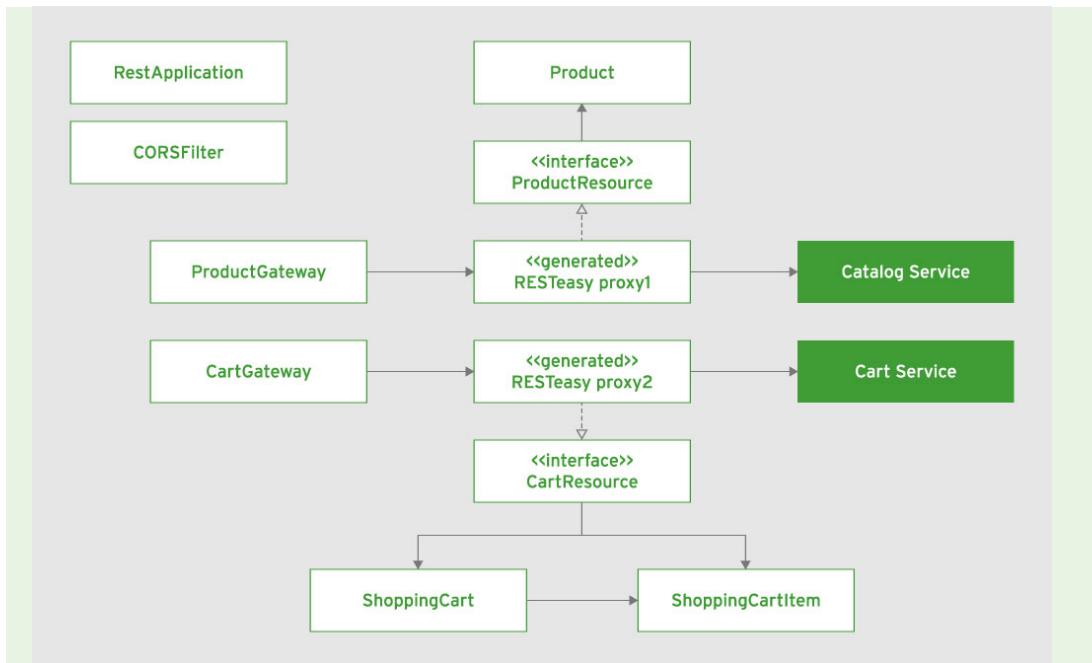
- Develop an API gateway microservice that proxies the HTTP API calls to the Cart and Catalog microservices.
- Implement and test an HTTP API that allows cross origin resource sharing (CORS), and can be consumed by the Coolstore UI microservice.
- Deploy the API gateway microservice to OpenShift.
- Deploy the Coolstore UI microservice to OpenShift.

The API Gateway Microservice

The following diagram shows a high level overview of the Coolstore application and how the API gateway acts as a single point of entry for the Coolstore web user interface:



The following UML class diagram illustrates the relationships between all classes in the API gateway microservice.



The API gateway microservice is actually composed of two proxy interfaces (**ProductResource** and **CartResource**), two API implementation classes (**ProductGateway** and **CartGateway**), and a number of support classes. The RESTEasy framework generates runtime instances from the proxy interfaces, and automatically takes care of converting JAVA objects to and from JSON.

The API gateway microservice runs on the WildFly Swarm runtime. It provides an HTTP API to the Coolstore UI application, which runs on the user's web browser as an AngularJS application. The Coolstore UI application also a web server that uses Node.js. Its sole purpose is to serve static HTML, CSS, and JavaScript files to the web browser.

You must deploy the application according to the following requirements:

- The API gateway microservice should be deployed in an OpenShift project called **api-gateway-service**
- The Coolstore UI microservice should be deployed in an OpenShift project called **coolstore-ui**
- The HTTP API for the API gateway should be accessible at the URL:
`http://coolstore-gateway.apps.lab.example.com/api`
- The Coolstore UI should be accessible at the URL:
`http://coolstore.apps.lab.example.com`
- The Git repository that contains the source code for the microservice is available at:
`http://services.lab.example.com/coolstore`

To perform this lab, you need access to:

- A running OpenShift cluster

- The Cart and Catalog microservices deployed and running on OpenShift. You need to perform the following exercises before attempting this lab:
 - The guided exercise *Developing Microservices with the Vert.x Runtime* starting from Step 10
 - The guided exercise *Developing Microservices with the Spring Boot Runtime* starting from Step 9
 - For each of these guided exercises, you may also need to perform the *Before you begin* section and review the first step to make sure you have their prerequisites set up and are starting from the correct solution branch
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-gateway-lab-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **coolstore/coolstore-gateway-jaxrs** Maven project
- The **org.wildfly.swarm:bom**, **io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts in the classroom Nexus proxy server

If you need help using Git and JBoss Developer Studio, refer to *Appendix A: Managing Git Branches*, and *Appendix B: Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the prerequisites:

```
[student@workstation ~]$ lab gateway-deploy setup
```

You can compare your source code changes while performing this lab with the solution branch **do292-gateway-lab-solution** of the **coolstore** repository.

1. Log in to the OpenShift cluster as the **developer** user, with a password of **redhat**. Create the project for the API gateway microservice.
2. Switch to the **do292-gateway-lab-begin** branch in the **coolstore** Git repository. Import the **~/coolstore/coolstore-gateway-jaxrs** Maven project into JBoss Developer Studio.
If you already have a clone of the **coolstore** project, and you have uncommitted local changes, either commit or stash your local changes.



Note

If you want to skip Step 3 to Step 8 and want to deploy the fully completed API gateway to OpenShift, check out the branch named **do292-gateway-lab-solution**, and continue from Step 9.

3. Complete the implementation of the **ProductResource** proxy interface in the **com.redhat.coolstore.gateway.proxy** package.
Complete the **ProductResource** interface according to the following specification:
 - The **ProductResource** proxy instances should consume and produce JSON responses.
 - The **getProducts()** method should respond to HTTP GET requests at the **/products** path.

- The **getProduct()** method should respond to HTTP GET requests at the **/product/{itemId}** path. This method should accept a single string argument called **itemId**.
- The **addProduct()** method should accept HTTP POST requests at the **/product** path.

Use the fully implemented **CartResource** interface in the **com.redhat.coolstore.gateway.model** package as reference to complete the **ProductResource** interface.

4. Complete the implementation of the **ProductGateway** class in the **com.redhat.coolstore.gateway.api** package.

Using the RESTEasy framework, complete the **ProductGateway** class according to the following specification:

- Complete the **buildClient()** method to return an instance of **ProductResource**. Use the RESTEasy **ResteasyWebTarget** class to generate and return a concrete instance of the **ProductResource** proxy interface.
- The **getProducts()** method should respond to HTTP GET requests at the **/products** path.
- The **getProduct()** method should respond to HTTP GET requests at the **/product/{itemId}** path. This method should accept a single string argument called **itemId**.
- The **addProduct()** method should accept HTTP POST requests at the **/product** path.

Use the fully implemented **CartGateway** class in the **com.redhat.coolstore.gateway.api** package as reference to complete the **ProductGateway** class.

5. Complete the implementation of the **CORSFilter** class in the **com.redhat.coolstore.gateway.api** package.

Complete the **CORSFilter** class according to the following specification:

- Add the appropriate header that allows all clients to invoke the API gateway.
- Add the appropriate header that allows all clients to request the following HTTP request types: **GET**, **POST**, **PUT**, **DELETE**, **OPTIONS**, and **HEAD**.

6. Complete the implementation of the **ProductGatewayTest** unit test class in the **com.redhat.coolstore.gateway.api.tests** package.

Complete the **getProducts()** test method according to the following specification:

- Using the WireMock framework, mock the response to HTTP GET requests to the **/products** path. You must return a JSON response with an HTTP status code of **200**.

The body of the response should be a JSON array of products. The complete JSON array of products that is to be sent as a response is provided in the **/src/test/resources/_files/products.json** file.

- Instantiate a JAX-RS client call to the mock method at the path **/api/products**.
- Parse the JSON response and assert the following:
 - The HTTP status code of the response must be **200**.
 - The length of the products array must be 8.
 - The **itemId** attribute of the first product in the array must be **329299**.
 - The **name** attribute of the first product in the array must be **Red Fedora**.

Use the fully implemented **CartGatewayTest** class in the **com.redhat.coolstore.gateway.api.tests** package as reference to complete the **ProductGatewayTest** class.

You can also refer to the **getProduct()** method in the **ProductGatewayTest** class.

7. Run the unit tests for the API gateway implementation in the **CartGatewayTest** and **ProductGatewayTest** classes. Verify that all the tests pass.

If you are running these tests for the very first time, the tests may take a long time to run. You can safely ignore any errors in the IDE **Console** view about failures related to the downloading of artifacts in the **org.glassfish** Maven group.

8. Complete the resource fragment files for the configuration map and the deployment configuration to initialize the **CATALOG_SERVICE_URL** environment variable from the configuration map.

Add a new key to the configuration map called **CATALOG_SERVICE_URL**. Set the value for this key as **http://catalog-service.apps.lab.example.com**.

Add to the deployment configuration the **CATALOG_SERVICE_URL** environment variable that references the **CATALOG_SERVICE_URL** key from the **gw-config** configuration map.

9. Deploy the API gateway microservice to OpenShift using the Fabric8 Maven plug-in.

Test the API gateway by sending an HTTP GET request to the following URL:

http://coolstore-gateway.apps.lab.example.com/api/products

10. Deploy the Coolstore UI microservice to OpenShift to a new project, using the template in the **~/coolstore/coolstore-ui/ocp/coolstore-ui-template.yaml** file.

The template provides all the parameters required to deploy the Coolstore UI microservice. You do not need to pass any parameters to this template. Name the OpenShift application for the Coolstore UI as **coolstore-ui**.

11. Test the Coolstore UI using a web browser.

- 11.1. Navigate to **http://coolstore.apps.lab.example.com** using a web browser to open the Coolstore UI home page.

The screenshot shows a shopping cart page from the Red Hat Cool Store. At the top, there's a navigation bar with the Red Hat logo and links for "Red Hat Cool Store" and "Your Shopping Cart". Below the navigation, there are two product items listed:

- Red Fedora**: Official Red Hat Fedora. Price: \$34.99. Add To Cart button.
- Forge Laptop Sticker**: JBoss Community Forge Project Sticker. Price: \$8.50. Add To Cart button.

At the bottom of the page, a note says: "Firefox automatically sends some data to Mozilla so that we can improve your experience."

- 11.2. Add a few products to the shopping cart. Click **Shopping Cart** in the top right corner and verify that the products you added are seen in the cart, and that the calculated amounts are correct.

The screenshot shows the "Shopping Summary" page. It displays the following information:

- Cart Total: \$69.98
- Promotional Item Savings: \$0.00
- Subtotal: \$69.98
- Shipping: \$6.99
- Promotional Shipping Savings: \$0.00
- Total Order Amount: \$76.97**

At the bottom, there are two buttons: "Sign in unavailable" and "Keep Shopping".

12. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were successful:

```
[student@workstation coolstore-gateway-jaxrs]$ lab gateway-deploy grade
```

13. Commit your changes to your local Git repository:

```
[student@workstation coolstore-gateway-jaxrs]$ git commit -a -m \  
"Finished the API gateway lab."
```

14. Remove the **coolstore-gateway** project from the IDE workspace.
15. **Optional:** Clean up to redo the exercise from scratch.
 - 15.1. Perform this and the following steps only if you wish to start over this exercise.
Reset your local repository to the remote branch:

```
[student@workstation coolstore-gateway-jaxrs]$ git reset --hard \  
origin/do292-gateway-lab-begin
```

Delete the **api-gateway-service** and **coolstore-ui** projects in OpenShift.

```
[student@workstation coolstore-gateway-jaxrs]$ oc delete project coolstore-ui  
[student@workstation coolstore-gateway-jaxrs]$ oc delete project \  
api-gateway-service
```

This concludes the lab.

► Solution

Developing an API Gateway

Performance Checklist

In this lab, you will develop an API gateway for the Coolstore application.

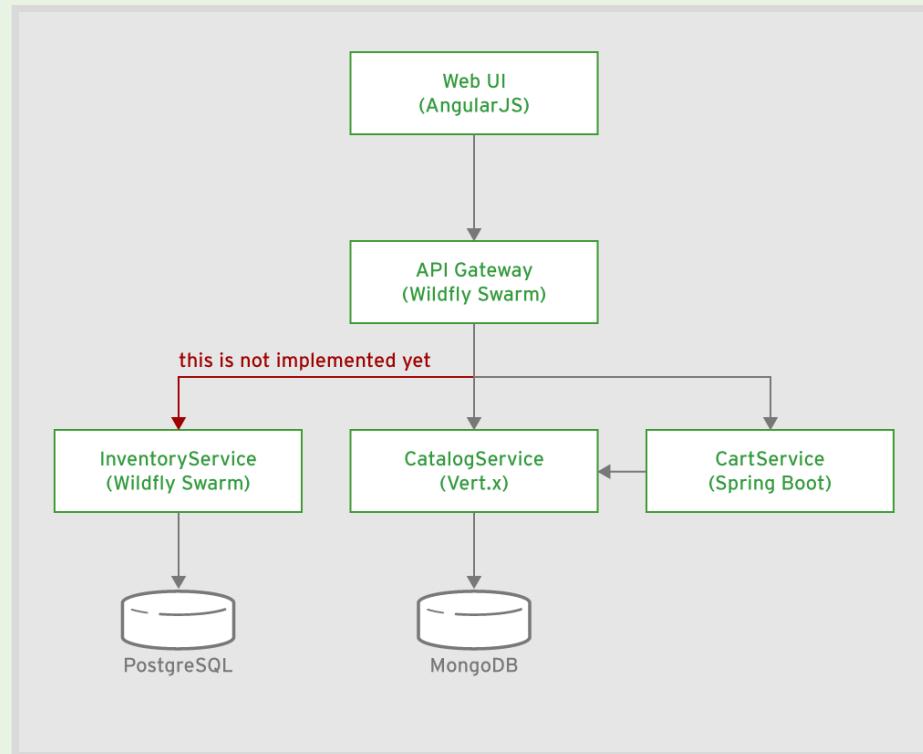
Outcomes

You should be able to:

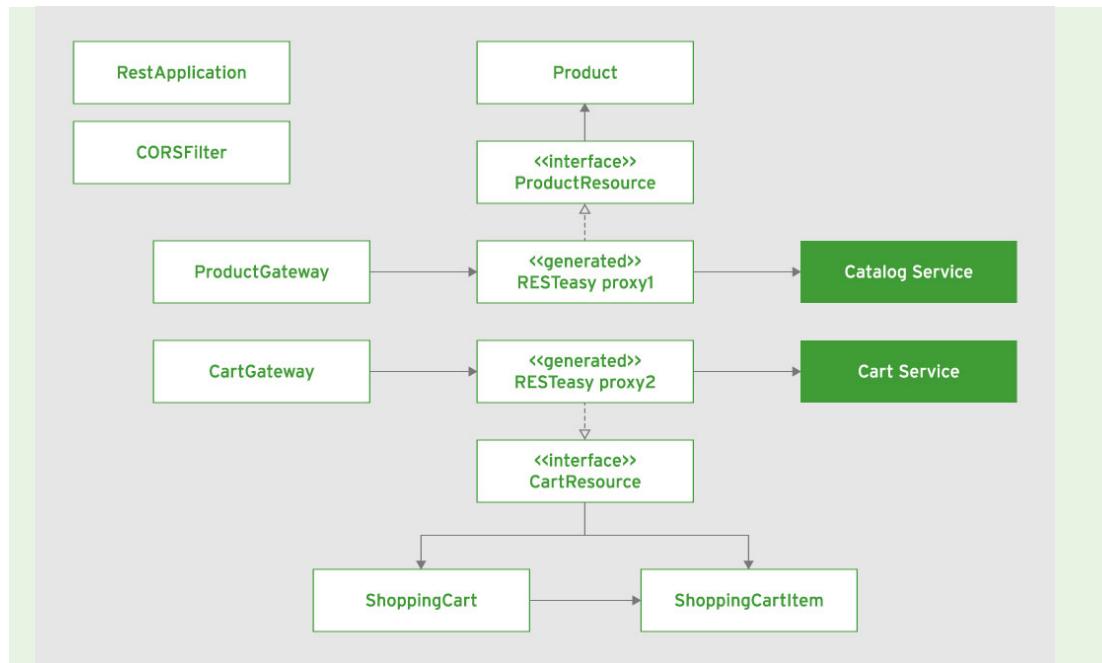
- Develop an API gateway microservice that proxies the HTTP API calls to the Cart and Catalog microservices.
- Implement and test an HTTP API that allows cross origin resource sharing (CORS), and can be consumed by the Coolstore UI microservice.
- Deploy the API gateway microservice to OpenShift.
- Deploy the Coolstore UI microservice to OpenShift.

The API Gateway Microservice

The following diagram shows a high level overview of the Coolstore application and how the API gateway acts as a single point of entry for the Coolstore web user interface:



The following UML class diagram illustrates the relationships between all classes in the API gateway microservice.



The API gateway microservice is actually composed of two proxy interfaces (**ProductResource** and **CartResource**), two API implementation classes (**ProductGateway** and **CartGateway**), and a number of support classes. The RESTEasy framework generates runtime instances from the proxy interfaces, and automatically takes care of converting JAVA objects to and from JSON.

The API gateway microservice runs on the WildFly Swarm runtime. It provides an HTTP API to the Coolstore UI application, which runs on the user's web browser as an AngularJS application. The Coolstore UI application also a web server that uses Node.js. Its sole purpose is to serve static HTML, CSS, and JavaScript files to the web browser.

You must deploy the application according to the following requirements:

- The API gateway microservice should be deployed in an OpenShift project called **api-gateway-service**
- The Coolstore UI microservice should be deployed in an OpenShift project called **coolstore-ui**
- The HTTP API for the API gateway should be accessible at the URL:
`http://coolstore-gateway.apps.lab.example.com/api`
- The Coolstore UI should be accessible at the URL:
`http://coolstore.apps.lab.example.com`
- The Git repository that contains the source code for the microservice is available at:
`http://services.lab.example.com/coolstore`

To perform this lab, you need access to:

- A running OpenShift cluster

- The Cart and Catalog microservices deployed and running on OpenShift. You need to perform the following exercises before attempting this lab:
 - The guided exercise *Developing Microservices with the Vert.x Runtime* starting from Step 10
 - The guided exercise *Developing Microservices with the Spring Boot Runtime* starting from Step 9
 - For each of these guided exercises, you may also need to perform the *Before you begin* section and review the first step to make sure you have their prerequisites set up and are starting from the correct solution branch
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**)
- The **redhat-openjdk18-openshift** image stream
- The **do292-gateway-lab-*** branches in the classroom Git repository, containing the source code for the microservice as part of the **coolstore/coolstore-gateway-jaxrs** Maven project
- The **org.wildfly.swarm:bom, io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts in the classroom Nexus proxy server

If you need help using Git and JBoss Developer Studio, refer to *Appendix A: Managing Git Branches*, and *Appendix B: Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the prerequisites:

```
[student@workstation ~]$ lab gateway-deploy setup
```

You can compare your source code changes while performing this lab with the solution branch **do292-gateway-lab-solution** of the **coolstore** repository.

1. Log in to the OpenShift cluster as the **developer** user, with a password of **redhat**. Create the project for the API gateway microservice.
 - 1.1. Log in to OpenShift as the **developer** user:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
```

- 1.2. Create a new project for the application:

```
[student@workstation ~]$ oc new-project api-gateway-service
```

2. Switch to the **do292-gateway-lab-begin** branch in the **coolstore** Git repository. Import the **~/coolstore/coolstore-gateway-jaxrs** Maven project into JBoss Developer Studio.

If you already have a clone of the **coolstore** project, and you have uncommitted local changes, either commit or stash your local changes.

- 2.1. If you have not done so yet, clone the **coolstore** project from the classroom Git repository.

From the home directory of the student user on the workstation VM, clone the **coolstore** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/coolstore
Cloning into 'coolstore'...
...
```

2.2. Switch to the **do292-gateway-lab-begin** branch:

```
[student@workstation ~]$ cd ~/coolstore
[student@workstation coolstore]$ git checkout do292-gateway-lab-begin
...
Switched to a new branch 'do292-gateway-lab-begin'
```

- 2.3. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/coolstore/coolstore-gateway-jaxrs** folder as a Maven project.
- 2.4. If you imported the project before, refresh the project and update its configuration. Make sure that the project now displays the correct branch name, which is **do292-gateway-lab-begin**.



Note

If you want to skip ??? to ??? and want to deploy the fully completed API gateway to OpenShift, check out the branch named **do292-gateway-lab-solution**, and continue from ???.

3. Complete the implementation of the **ProductResource** proxy interface in the **com.redhat.coolstore.gateway.proxy** package.
Complete the **ProductResource** interface according to the following specification:
 - The **ProductResource** proxy instances should consume and produce JSON responses.
 - The **getProducts()** method should respond to HTTP GET requests at the **/products** path.
 - The **getProduct()** method should respond to HTTP GET requests at the **/product/{itemId}** path. This method should accept a single string argument called **itemId**.
 - The **addProduct()** method should accept HTTP POST requests at the **/product** path.

Use the fully implemented **CartResource** interface in the **com.redhat.coolstore.gateway.model** package as reference to complete the **ProductResource** interface.

- 3.1. Edit the **ProductResource** interface in the **com.redhat.coolstore.gateway.model** package.
Add the following JAX-RS annotations to the **ProductResource** interface to indicate that all the methods in the class consume and produce JSON:

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public interface ProductResource {
    ...
}
```

3.2. Add the following JAX-RS annotations to the **getProducts()** method:

```
@GET
@Path("/products")
public List<Product> getProducts();
...
```

3.3. Add the following JAX-RS annotations to the **getProduct()** method:

```
@GET
@Path("/product/{itemId}")
public Product getProduct(@PathParam("itemId") String itemId);
...
```

3.4. Add the following JAX-RS annotations to the **addProduct()** method:

```
@POST
@Path("/product")
public Product addProduct(Product product);
...
```

4. Complete the implementation of the **ProductGateway** class in the **com.redhat.coolstore.gateway.api** package.

Using the RESTEasy framework, complete the **ProductGateway** class according to the following specification:

- Complete the **buildClient()** method to return an instance of **ProductResource**. Use the RESTEasy **ResteasyWebTarget** class to generate and return a concrete instance of the **ProductResource** proxy interface.
- The **getProducts()** method should respond to HTTP GET requests at the **/products** path.
- The **getProduct()** method should respond to HTTP GET requests at the **/product/{itemId}** path. This method should accept a single string argument called **itemId**.
- The **addProduct()** method should accept HTTP POST requests at the **/product** path.

Use the fully implemented **CartGateway** class in the **com.redhat.coolstore.gateway.api** package as reference to complete the **ProductGateway** class.

4.1. Edit the **ProductGateway** class in the **com.redhat.coolstore.gateway.api** package.

Complete the **buildClient()** method as follows:

```

private ProductResource buildClient() {
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target(catalogServiceUrl);
    ResteasyWebTarget restEasyTarget = (ResteasyWebTarget) target;
    return restEasyTarget.proxy(ProductResource.class);
}
...

```

4.2. Add the following JAX-RS annotations to the **getProducts()** method:

```

@GET
@Path("/products")
public List<Product> getProducts() {
    ProductResource proxy = buildClient();
    return proxy.getProducts();
}
...

```

4.3. Add the following JAX-RS annotations to the **getProduct()** method:

```

@GET
@Path("/product/{itemId}")
public Product getProduct(@PathParam("itemId") String itemId) {
    ProductResource proxy = buildClient();
    return proxy.getProduct(itemId);
}
...

```

4.4. Add the following JAX-RS annotations to the **addProduct()** method:

```

@POST
@Path("/product")
public Product addProduct(Product product) {
    ProductResource proxy = buildClient();
    proxy.addProduct(product);
}
...

```

5. Complete the implementation of the **CORSFilter** class in the **com.redhat.coolstore.gateway.api** package.

Complete the **CORSFilter** class according to the following specification:

- Add the appropriate header that allows all clients to invoke the API gateway.
- Add the appropriate header that allows all clients to request the following HTTP request types: **GET, POST, PUT, DELETE, OPTIONS, and HEAD**.

- 5.1. Edit the **CORSFilter** class in the **com.redhat.coolstore.gateway.api** package.

Add the following CORS headers to the **filter()** method:

```

...
public void filter(ContainerRequestContext requestContext,
ContainerResponseContext responseContext) throws IOException {

    responseContext.getHeaders()
        .add("Access-Control-Allow-Origin", "*");

    responseContext.getHeaders()
        .add("Access-Control-Allow-Methods",
            "GET, POST, PUT, DELETE, OPTIONS, HEAD");
    ...
}
...

```

6. Complete the implementation of the **ProductGatewayTest** unit test class in the **com.redhat.coolstore.gateway.api.tests** package.

Complete the **getProducts()** test method according to the following specification:

- Using the WireMock framework, mock the response to HTTP GET requests to the **/products** path. You must return a JSON response with an HTTP status code of **200**.

The body of the response should be a JSON array of products. The complete JSON array of products that is to be sent as a response is provided in the **/src/test/resources/_files/products.json** file.

- Instantiate a JAX-RS client call to the mock method at the path **/api/products**.
- Parse the JSON response and assert the following:
 - The HTTP status code of the response must be **200**.
 - The length of the products array must be 8.
 - The **itemId** attribute of the first product in the array must be **329299**.
 - The **name** attribute of the first product in the array must be **Red Fedora**.

Use the fully implemented **CartGatewayTest** class in the **com.redhat.coolstore.gateway.api.tests** package as reference to complete the **ProductGatewayTest** class.

You can also refer to the **getProduct()** method in the **ProductGatewayTest** class.

- 6.1. Edit the **ProductGatewayTest** class in the **com.redhat.coolstore.gateway.api.tests** package.

Remove the call to **fail()** at the end of the **getProducts()** test method and add the following code to mock the **/api/products** resource URL of the Catalog service:

```

...
@Test
@RunAsClient
public void getProducts() throws Exception {
    catalogMockRule.stubFor(get(urlMatching("/products"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "application/json")
            .withBodyFile("products.json")));
}
...

```

- 6.2. Add the following code to invoke the **/api/products** resource URL in the API Gateway service:

```

...
import com.eclipsesource.json.Json;
import com.eclipsesource.json.JsonArray;
...
public void getProducts() throws Exception {
    catalogMockRule.stubFor(get(urlMatching("/products"))
        .willReturn(aResponse()
            .withStatus(200)
            .withHeader("Content-Type", "application/json")
            .withBodyFile("products.json")));

    WebTarget target = client.target("http://localhost:" + port)
        .path("/api")
        .path("/products");
    Response response = target.request(MediaType.APPLICATION_JSON).get();
    JsonArray items = Json.parse(response.readEntity(String.class)).asArray();
}
...

```

- 6.3. Add the following code to assert attributes from the API gateway response:

```

...
public void getProducts() throws Exception {
    ...
    Response response = target.request(MediaType.APPLICATION_JSON).get();
    JsonArray items = Json.parse(response.readEntity(String.class)).asArray();

    assertThat(response.getStatus(), equalTo(new Integer(200)));
    assertThat(items.size(), equalTo(new Integer(8)));
    assertThat(items.get(0).asObject()
        .getString("itemId", ""), equalTo("329299"));
    assertThat(items.get(0).asObject()
        .getString("name", ""), equalTo("Red Fedora"));
}
...

```

7. Run the unit tests for the API gateway implementation in the **CartGatewayTest** and **ProductGatewayTest** classes. Verify that all the tests pass.

If you are running these tests for the very first time, the tests may take a long time to run. You can safely ignore any errors in the IDE **Console** view about failures related to the downloading of artifacts in the **org.glassfish** Maven group.

- 7.1. Run the **ProductGatewayTest** test case from the **com.redhat.coolstore.gateway.api.tests** package as a JUnit test.
Verify that all tests pass.
- 7.2. Run the **CartGatewayTest** test case from the **com.redhat.coolstore.gateway.api.tests** package as a JUnit test.
Verify that all tests pass.
8. Complete the resource fragment files for the configuration map and the deployment configuration to initialize the **CATALOG_SERVICE_URL** environment variable from the configuration map.
Add a new key to the configuration map called **CATALOG_SERVICE_URL**. Set the value for this key as **http://catalog-service.apps.lab.example.com**.
Add to the deployment configuration the **CATALOG_SERVICE_URL** environment variable that references the **CATALOG_SERVICE_URL** key from the **gw-config** configuration map.

- 8.1. Edit the **configmap.yml** file in the **/src/main/fabric8** folder, and add the following:

```
...
data:
  CATALOG_SERVICE_URL: http://catalog-service.apps.lab.example.com
  CART_SERVICE_URL: http://cart-service.apps.lab.example.com
```

- 8.2. Edit the **deployment.yml** file in the **/src/main/fabric8** folder and add the following:

```
...
env:
  - name: CATALOG_SERVICE_URL
    valueFrom:
      configMapKeyRef:
        key: CATALOG_SERVICE_URL
        name: gw-config
  - name: CART_SERVICE_URL
    valueFrom:
      configMapKeyRef:
        key: CART_SERVICE_URL
        name: gw-config
```

9. Deploy the API gateway microservice to OpenShift using the Fabric8 Maven plug-in.

Test the API gateway by sending an HTTP GET request to the following URL:

http://coolstore-gateway.apps.lab.example.com/api/products

- 9.1. Deploy the API gateway microservice to the OpenShift cluster.

Use the **fabric8:deploy** Maven goal. Skip tests to have a faster deployment. If you see a **RejectedExecutionException**, you can safely ignore it.

```
[student@workstation coolstore]$ cd coolstore-gateway-jaxrs
[student@workstation coolstore-gateway-jaxrs]$ mvn fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:resource (default) @ coolstore-gateway ---
...
[INFO] --- maven-war-plugin:2.5:war (default-war) @ coolstore-gateway ---
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ coolstore-gateway ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (default) @ coolstore-gateway ---
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ coolstore-gateway
---
...
[INFO] BUILD SUCCESS
...
```

- 9.2. Verify that FMP created a configuration map that contains the URL of the Cart and Catalog microservices:

```
[student@workstation coolstore-gateway-jaxrs]$ oc describe configmap gw-config
...
Data
=====
CART_SERVICE_URL:
-----
http://cart-service.apps.lab.example.com
CATALOG_SERVICE_URL:
-----
http://catalog-service.apps.lab.example.com
...
```

- 9.3. Wait until the API gateway microservice pod is ready and running:

```
[student@workstation coolstore-gateway-jaxrs]$ oc get pod
NAME           READY   STATUS    RESTARTS   AGE
coolstore-gateway-1-2k9tz   1/1     Running   0          33s
coolstore-gateway-s2i-1-build 0/1     Completed  0          1m
```

Verify that the API gateway microservice logs resemble the following:

```
[student@workstation coolstore-gateway-jaxrs]$ oc logs coolstore-gateway-1-c9sk4
...
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: WildFly Swarm
7.1.0.redhat-77 (WildFly Core 3.0.12.Final-redhat-1) started in 8589ms...
...
INFO [org.wildfly.swarm.runtime.deployer] (main) deploying coolstore-
gateway-1.0.0.war
```

```
...
INFO [org.jboss.as.server] (main) WFLYSRV0010: Deployed "coolstore-gateway-1.0.0.war"
...
INFO [org.wildfly.swarm] (main) WFSWARM99999: WildFly Swarm is Ready
...
```

9.4. Verify that the API gateway can fetch the list of products using the catalog service:

```
[student@workstation coolstore-gateway-jaxrs]$ curl -si \
  http://coolstore-gateway.apps.lab.example.com/api/products
HTTP/1.1 200 OK
Content-type: application/json
...
[ {
  "itemId" : "329299",
  "name" : "Red Fedora",
  "desc" : "Official Red Hat Fedora",
  "price" : 34.99
}, {
  "itemId" : "329299",
  "name" : "Forge Laptop Sticker",
  "desc" : "JBoss Community Forge Project Sticker",
  "price" : 8.5
} ...
```

10. Deploy the Coolstore UI microservice to OpenShift to a new project, using the template in the **~/coolstore/coolstore-ui/ocp/coolstore-ui-template.yaml** file.

The template provides all the parameters required to deploy the Coolstore UI microservice. You do not need to pass any parameters to this template. Name the OpenShift application for the Coolstore UI as **coolstore-ui**.

10.1. Create a new project for the Coolstore UI microservice:

```
[student@workstation coolstore-gateway-jaxrs]$ oc new-project coolstore-ui
```

10.2. Deploy the Coolstore UI microservice using the provided template:

```
[student@workstation coolstore-gateway-jaxrs]$ oc new-app \
  --name coolstore-ui \
  -f ~/coolstore/coolstore-ui/ocp/coolstore-ui-template.yaml
--> Deploying template "coolstore-ui/coolstore-ui" ...
...
* With parameters:
...
* COOLSTORE_GW_URL=http://coolstore-gateway.apps.lab.example.com/api
* COOLSTORE_UI_HOSTNAME=coolstore.apps.lab.example.com

--> Creating resources ...
...
--> Success
...
```

10.3. Check the build logs for the Coolstore UI microservice and wait until the build finishes.

```
[student@workstation coolstore-gateway-jaxrs]$ oc logs -f bc/coolstore-ui
Cloning "http://services.lab.example.com/coolstore" ...
...
---> Installing application source ...
---> Building your Node application from source
...
Pushing image docker-registry.default.svc:5000/coolstore-ui/coolstore-
ui:latest ...
Pushed 5/6 layers, 93% complete
Pushed 6/6 layers, 100% complete
Push successful
```

10.4. Wait until the Coolstore UI microservice pod is ready and running:

```
[student@workstation coolstore-gateway-jaxrs]$ oc get pod
NAME           READY   STATUS    RESTARTS   AGE
coolstore-ui-1-build   0/1     Completed   0          1m
coolstore-ui-1-z2m8l   1/1     Running    0          8s
```

11. Test the Coolstore UI using a web browser.

- 11.1. Navigate to <http://coolstore.apps.lab.example.com> using a web browser to open the Coolstore UI home page.

The screenshot shows a web browser displaying the Red Hat Cool Store. The top navigation bar includes the Red Hat logo and links for 'Red Hat Cool Store' and 'Your Shopping Cart'. Below the navigation, there are four product cards arranged in a grid:

- Red Fedora**: Official Red Hat Fedora. Price: \$34.99. Add To Cart button.
- Forge Laptop Sticker**: JBoss Community Forge Project Sticker. Price: \$8.50. Add To Cart button.
- Ogio Caliber Polo**: Moisture-wicking 100% polyester. Price: Not explicitly shown. Add To Cart button.
- 16 oz. Vortex Tumbler**: Double-wall insulated, BPA-free, acrylic cup. Price: Not explicitly shown. Add To Cart button.

A note at the bottom of the page states: "Firefox automatically sends some data to Mozilla so that we can improve your experience."

- 11.2. Add a few products to the shopping cart. Click **Shopping Cart** in the top right corner and verify that the products you added are seen in the cart, and that the calculated amounts are correct.

The screenshot shows a "Shopping Summary" page with the following details:

- Cart Total: \$69.98
- Promotional Item Savings: \$0.00
- Subtotal: \$69.98
- Shipping: \$6.99
- Promotional Shipping Savings: \$0.00

Total Order Amount: \$76.97

At the bottom, there are two buttons: "Sign in unavailable" and "Keep Shopping".

12. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were successful:

```
[student@workstation coolstore-gateway-jaxrs]$ lab gateway-deploy grade
```

13. Commit your changes to your local Git repository:

```
[student@workstation coolstore-gateway-jaxrs]$ git commit -a -m \
"Finished the API gateway lab."
```

14. Remove the **coolstore-gateway** project from the IDE workspace.

15. Optional: Clean up to redo the exercise from scratch.

15.1. Perform this and the following steps only if you wish to start over this exercise.

Reset your local repository to the remote branch:

```
[student@workstation coolstore-gateway-jaxrs]$ git reset --hard \
origin/do292-gateway-lab-begin
```

Delete the **api-gateway-service** and **coolstore-ui** projects in OpenShift.

```
[student@workstation coolstore-gateway-jaxrs]$ oc delete project coolstore-ui
[student@workstation coolstore-gateway-jaxrs]$ oc delete project \
api-gateway-service
```

This concludes the lab.

Summary

In this chapter, you learned:

- Use an API gateway in a microservices architecture to avoid exposing the fine-grained API of each microservice directly to clients. An API gateway acts as a single point of access for client requests and offers a simpler, and more generic API to clients.
- The API gateway intelligently routes requests to microservices, and performs protocol translation between clients and microservices.
- The API gateway can help with request throttling and limiting requests below a certain threshold.
- Using an API gateway in a microservices architecture should be weighed against the cost of developing and maintaining it since it incurs extra cost overhead in terms of both effort and performance.
- You can implement API gateways using general purpose components like the RHOAR runtimes, integration frameworks like Red Hat Fuse, and enterprise API management products like Red Hat 3scale.
- To allow clients not belonging to the same domain as the API gateway to invoke the resources offered by the gateway, you need to enable cross origin resource sharing (CORS) in the API gateway. Enable CORS by setting extra headers in the response from the API gateway.
- To enable easier and standalone testing of API gateways, you can use a mocking framework like WireMock.

Chapter 6

Implementing Fault Tolerance with Hystrix

Goal

Implement fault tolerance in a series of microservices using the Hystrix libraries.

Objectives

- Make a microservice fault tolerant with Hystrix.
- Deploy the Hystrix Dashboard and Turbine to OpenShift.

Sections

- Creating a Fault Tolerant Microservice (and Guided Exercise)
- Deploying Hystrix Dashboard and Turbine (and Guided Exercise)

Lab

Implementing Fault Tolerance with Hystrix

Creating a Fault Tolerant Microservice

Objective

After completing this section, students should be able to make a microservice fault tolerant with Hystrix.

Developing Robust Applications with Fault Tolerance

Fault tolerance ensures that a microservice handles failures gracefully, and optionally provides a fallback if a dependent service becomes unavailable. Hardware failures, network connectivity issues, routine maintenance, and failed deployments are all reasons a service could go offline at any moment.

Fault tolerance is not applicable only when a microservice calls another microservice. Every time a microservice calls an external service, such as a database, it can implement fault tolerance.

Two common patterns in fault tolerance APIs are the *circuit breaker* and *bulkhead*. The circuit breaker pattern isolates a consumer from failures in a dependent service, so the dependent service is not called until it recovers. The bulkhead pattern limits concurrent invocations to a dependent service, so the dependent service is not overloaded by a single consumer. The remainder of this section focus on the circuit breaker pattern.

Describing the Circuit Breaker Pattern

A circuit breaker acts as a gate that decides whether a block of code should be executed or not. When the circuit is *closed*, the code executes. When the circuit is *open*, the code does not execute. If the circuit is closed, and the code execution fails, the circuit is opened. After a configurable time delay, the circuit is closed again.

A typical circuit breaker implementation may be configured to consider that a failure happens either when the code throws an exception, or when the code takes too long to run. It may also be configured to wait for a number of failures to occur before opening the circuit.

A circuit breaker does not make the consumer of a service immune to failures of the service. It makes the consumer fail fast without actually trying to consume the service. This alleviates the load on the service and on the network that connects the consumer to the service, and allows the service and the network to recover gracefully from congestion, high load, and other transient issues.

A circuit breaker alleviates the consumer from the time, memory, and CPU spent waiting (or *pooling*) for the service's response, which is potentially an unusable error response. It also prevents the service failure from cascading; that is, from creating another failure at the consumer.

The consumer is expected to be able to handle circuit breaker failures, and also service failures, in a way that it can resume working without requiring either a restart or a redeployment.

Describing Hystrix

Hystrix is a Java library from Netflix open source software (Netflix OSS). Netflix OSS provides fault tolerance, latency, monitoring, and concurrency capabilities, as well as monitoring tools that help identifying problems in microservices.

The Hystrix library is framework and runtime-agnostic. Developers using any of the runtimes supported by Red Hat OpenShift Application Runtimes can also use the Hystrix libraries, but some frameworks provide alternatives that adhere more closely than others to the frameworks overall design. For example:

- WildFly Swarm provides the MicroProfile Fault Tolerance API, which is based on CDI interceptors. WildFly Swarm uses Hystrix internally to implement the MicroProfile Fault Tolerance API.
- Vert.x provides the Circuit Breaker module, which follows the Vert.x asynchronous programming model. It is not based on Hystrix, but is compatible with the Hystrix metrics API.
- Spring Boot includes the Spring Cloud Hystrix library, which provides an annotation-based approach to wrap a method with a Hystrix circuit breaker.

To enable the Hystrix fault tolerance library, add the following dependency to the project's POM file:

```
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
</dependency>
```

Implementing Hystrix Fault Tolerance in Microservices

Hystrix implements fault tolerance using the *command* design pattern. It provides the **HystrixCommand** abstract class, which defines a **run** abstract method.

To wrap a chunk of code with a circuit breaker using Hystrix, a developer must perform three tasks:

- Define and configure a command class that extends the **HystrixCommand** class.
- Override the **run** method and move the block of code to be executed inside it.
- Replace the original block of code with a call to the **execute** method from the command class.

The **HystrixCommand** class also provides configuration parameters to define the thresholds that opens and closes the circuit. To set these parameters, a developer either calls **HystrixCommand** methods directly or defines system properties.

For example, consider the following listing, which invokes the Aloha microservice by calling the **aloha** method from the **AlohaService** class:

```
public class HolaResource {
    ...
    @Inject
    private AlohaService alohaService;
    ...
    public List<String> holaChaining(){
        ArrayList<String> list = new ArrayList<String>();
        list.add(alohaService.aloha());
        return list;
    }
}
```

To wrap the **aloha** method call with a circuit breaker, first define a Hystrix command class:

```

public class HolaChainingCommand extends HystrixCommand<String> {

    ...

    public HolaChainingCommand(...) {
        super(Setter
            .withGroupKey(HystrixCommandGroupKey.Factory.asKey("group")) ❶
            .andCommandKey(HystrixCommandKey.Factory.asKey(HOLA_CHAINING_COMMAND_KEY)) ❷
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter()
                    .withCircuitBreakerRequestVolumeThreshold(2) ❸
                    .withCircuitBreakerSleepWindowInMilliseconds(5000)) ❹
            );
        ...
    }

    ...
}

```

- ❶ Groups multiple **HystrixCommand** instances. Hystrix uses this value to fetch aggregated metrics and common configuration parameters for a set of Hystrix circuit breakers.
- ❷ Identifies this **HystrixCommand** instance's metrics.
- ❸ Defines the number of failed requests that Hystrix must process before making the service unavailable by opening the circuit.
- ❹ Defines the amount of time in milliseconds that the circuit is kept opened. After that amount of time, the circuit is closed again.

Move the code that invokes the Aloha microservice to inside the **run** method of the command class:

```

public class HolaChainingCommand extends HystrixCommand<String> {

    private AlohaService alohaService;

    public HolaChainingCommand(AlohaService alohaService) {
        ...
        this.alohaService = alohaService;
    }

    ...

    @Override
    protected String run() throws Exception {
        return alohaService.aloha();
    }
}

```

Finally, replace the original call to the **aloha** method with a call to the command class:

```

public class HolaResource{
    ...
    @Inject
    private AlohaService alohaService;
    ...
    public List<String> holaChaining(){
        ArrayList<String> list = new ArrayList<String>();
        list.add(new HolaChainingCommand(alohaService).execute());
        return list;
    }
}

```

The Hystrix library also provides an HTTP API that outputs circuit breaker metrics and tools such as the Hystrix Dashboard and Hystrix Turbine that displays and collects these metrics.

Configuring Hystrix for the WildFly Swarm Runtime

Hystrix also implements the bulkhead microservice architecture pattern, which limits the number of simultaneous calls to a service. To use this feature, an application provides Hystrix with a thread pool. Otherwise, Hystrix can only have a single command object running at a time.

The following listing shows how to configure Hystrix to use a thread pool managed by the WildFly Swarm runtime:

```

package com.redhat.training.msa.hola.rest;

...
@ApplicationScoped ①
public class HystrixConfig {

    // The default factory is used
    @Resource(lookup = "java:comp/DefaultManagedThreadFactory")
    private ManagedThreadFactory threadFactory; ②

    // Initialize eagerly
    void init(@Observes @Initialized(ApplicationScoped.class) Object event) { ③
    }

    @PostConstruct ④
    public void onStartup() {
        HystrixPlugins.
        getInstance().
        registerConcurrencyStrategy(new HystrixConcurrencyStrategy() { ⑤
            @Override
            public ThreadPoolExecutor getThreadPool(HystrixThreadPoolKey
threadPoolKey,
                HystrixProperty<Integer> corePoolSize, ⑥
                HystrixProperty<Integer> maximumPoolSize,
                HystrixProperty<Integer> keepAliveTime,
                TimeUnit unit,
                BlockingQueue<Runnable> workQueue) {
                    return new ThreadPoolExecutor(corePoolSize.get(),
                        maximumPoolSize.get(),
                        keepAliveTime.get(),

```

```

        unit,
        workQueue,
        threadFactory);
    }
});

@PreDestroy ⑦
public void onShutdown() {
    Hystrix.reset(1, TimeUnit.SECONDS);
}

}

```

- ① Defines an application-scoped managed bean responsible for initializing Hystrix.
- ② Injects the **ManagedThreadFactory** instance from the WildFly Swarm runtime.
- ③ Initializes the application-scoped managed bean during startup.
- ④ Defines a method that must be executed during the creation of the managed bean.
- ⑤ Registers a **ConcurrencyStrategy** instance in Hystrix to use the thread pool from WildFly Swarm.
- ⑥ Overrides the **getThreadPool** method to configure the thread pool used by Hystrix.
- ⑦ Resets Hystrix when WildFly Swarm is shut down, releasing resources created during startup.

Wildfly Swarm also provides the **org.wildfly.swarm:hystrix** fraction that enables the Hystrix metrics stream at the resource URL **/hystrix.stream** by default. To enable the metrics endpoint, add the following dependency to the Maven POM definition file:

```

<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>hystrix</artifactId>
    <version>${version.hystrix}</version>
</dependency>

```

The **version.hystrix** property must use the Red Hat OpenShift Application Runtimes versions supported by the product's release notes.

Note that the metrics endpoint provides a continuous stream of metrics. A monitoring application is supposed to just keep the HTTP connection open, instead of making multiple HTTP requests to the metrics endpoint.

Initializing Hystrix in a Spring Boot Microservice

Spring Boot provides a set of annotations that simplifies the initialization of Hystrix. In order to use them, you must add the following dependency to the project POM:

```

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>${version.spring-cloud}</version>
</dependency>

```

The **version.spring-cloud** property must use the Red Hat OpenShift Application Runtime versions supported by the product's release notes.

To enable the circuit breaker support in a Spring Boot microservice, add the `org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker`, and the `org.springframework.cloud.netflix.hystrix.EnableHystrix` annotations to the class.

```
...
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;

...
@EnableCircuitBreaker
@EnableHystrix
public class CartServiceApplication {
    ...
}
```

To enable the Hystrix metrics endpoint, configure a servlet provided by Spring Boot to be accessible using the `/hystrix.stream` URL as follows:

```
package com.redhat.coolstore.cart;

import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import
com.netflix.hystrix.contrib.metrics.eventstream.HystrixMetricsStreamServlet;

@Configuration①
public class HystrixMetricsServletRegistrar {

    @Bean
    public ServletRegistrationBean hystrixMetricsServletRegistrationBean() {
        ServletRegistrationBean srb = new ServletRegistrationBean();
        srb.setServlet(new HystrixMetricsStreamServlet());②
        srb.addUrlMappings("/hystrix.stream");③
        srb.setEnabled(true);
        return srb;
    }

}
```

- ① Adds the class as a bean definition provider to Spring framework.
- ② Adds the `HystrixMetricsStreamServlet` instance among the Servlets available for external access.
- ③ Binds the servlet to the `/hystrix.stream` URI.

Initializing The Vert.x Circuit Breaker Module

Vert.x provides its own implementation of the circuit breaker pattern because Hystrix does not follow its asynchronous programming model. The Vert.x Circuit Breaker module provides Hystrix-compatible configuration properties and metrics so that a management application monitoring Hystrix circuit breakers can manage a Vert.x microservice using the Circuit Breaker Vert.x module.

To add the Circuit Breaker module to a Vert.x project, include the following dependency in the Maven POM file:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-circuit-breaker</artifactId>
  <version>${version.vertx}</version>
</dependency>
```

The **version.vertx** property defines the Vert.x compliant version of the Red Hat OpenShift Application Runtimes.

To enable the metrics endpoint, you must add a route for the **/hystrix.stream** resource URI that delegates to the **HystrixMetricsHandler** class:

```
@Override
public void start(Future<Void> startFuture) throws Exception {
    Router router = Router.router(vertx);
    ...
    router.get("/hystrix.stream").handler(HystrixMetricHandler.create(vertx));
    ...
}
```

To wrap a service call with a circuit breaker, first create and configure a **CircuitBreaker** object:

```
public class ApiVerticle extends AbstractVerticle {

    private CircuitBreaker circuitBreaker;

    @Override
    public void start(Future<Void> startFuture) throws Exception {
        ...
        circuitBreaker = CircuitBreaker.create("coolstore-product-circuit-breaker",
            vertx, new CircuitBreakerOptions()
                .setMaxFailures(5) ①
                .setTimeout(2000) ②
                .setResetTimeout(10000) ③
        );
        ...
    }
}
```

- ① Defines the number of failures before opening the circuit.
- ② Defines the amount of time the circuit breaker must wait until it increments the failure counter.
- ③ Defines the time spent in the open state before allowing other requests to access the service again.

Then wrap the call with a call to the **execute** method from the **CircuitBreaker** object:

```
public class ApiVerticle extends AbstractVerticle {

    private void getProducts(RoutingContext rc) {

        circuitBreaker.execute(future -> {
            // query MongoDB
        });
    }
}
```

```
}).setHandler(ar -> {
    // process MongoDB results
});
}
```



References

Netflix OSS Hystrix

<https://github.com/Netflix/Hystrix>

WildFly Swarm Netflix Fraction

<https://reference.wildfly-swarm.io/fractions/hystrix.html>

Vert.x circuit breaker

<https://vertx.io/docs/vertx-circuit-breaker/java/>

Spring Cloud Netflix

<https://cloud.spring.io/spring-cloud-netflix/>

► Guided Exercise

Making a Microservice Fault Tolerant

In this exercise, you will add a circuit breaker to a microservice that invokes another microservice.

Outcomes

You should be able to:

- Implement a circuit breaker using Hystrix in a microservice based on WildFly Swarm.
- Enable the Hystrix metrics stream endpoint in the microservice.

To perform this exercise, you need access to:

- The **do292-fault-tolerance-*** branches in the classroom Git repository, containing the source code for the Hola and Aloha microservices.
- The **org.wildfly.swarm:hystrix** and **com.netflix.hystrix:hystrix-core** Maven artifacts in the classroom Nexus proxy server.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab fault-tolerance setup
```

You can compare your source code changes while performing this exercise with the branch **do292-fault-tolerance-solution** of the **hello-microservices** Git repository.

► 1. Switch to the **do292-fault-tolerance-begin** branch in the **hello-microservices** Git repository and import the project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-fault-tolerance-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
do292-fault-tolerance-begin
...
Switched to a new branch 'do292-fault-tolerance-begin'
```

- 1.4. Open Red Hat JBoss Developer Studio and, if you have not already done so, import the **~/hello-microservices/hola** folder as a Maven project.
 - 1.5. If you imported the project previously, refresh the project and update its configuration. Make sure the project now displays the correct branch name, which is **do292-fault-tolerance-begin**.
 - 1.6. Ignore errors in the starter project.
JBoss Developer Studio builds the new **hola** project and reports syntax errors. Ignore them for now. You will fix these errors as you proceed with the exercise.
- ▶ 2. Add the Hystrix library to the Hola microservice.
This fixes compilation errors in the project and allows you to test the Hola microservice without circuit-breaker functionality.
- 2.1. Inspect the Hola project's Maven POM file.
Open the **pom.xml** file from the **hola** project.
Note the reference to a parent POM file, which provides values for system properties. These properties are used to define the version strings for Maven artifacts required to build the microservice.
 - 2.2. Add the **com.netflix.hystrix:hystrix-core** Maven artifact as a dependency. Use the **hystrix.version** property as the version attribute for the dependency.
Right after the opening **dependencies** element in your project POM, uncomment the **hystrix-core** dependency as follows:

```
...
<dependencies>
  <dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>${hystrix.version}</version>
  </dependency>
...
```

- The IDE rebuilds the project and removes all error markers.
- ▶ 3. Review the initial state of the **hola** project.
This version of the Hola microservice is also a consumer of the Aloha microservice. It fetches the output of the Aloha microservice and concatenates with its own output.
- 3.1. Inspect the code that implements the client of the Aloha microservice.
Open the **AlohaService** interface in the **com.redhat.training.msa.hola.client** package. It describes the endpoints of the Aloha microservices for a RESTeasy proxy.

Open the **ClientConfiguration** class in the **com.redhat.training.msa.hola.client** package. It provides a CDI producer method that creates a **ResteasyWebTarget** proxy object to the Aloha microservice.

Note that the **ClientConfiguration** class takes two configuration properties: **alohaPort** and **alohaHostname**. The JAX-RS **WebTarget** object uses them to locate and call the Aloha microservice.

Open the **microprofile-config.properties** file in the **/src/main/resources/META-INF** folder. This file provides default values for the Hola microservice configuration properties suitable for local testing of the microservice.

- 3.2. Inspect the code that invokes the Aloha microservice.

Open the **HolaResource** class in the **com.redhat.training.msa.hola.rest** package and inspect the **holaChaining** method. It adds the responses from Hola and the Aloha microservices into a Java **List** object, which is returned as JSON data to clients.

- 3.3. Start the Aloha microservice.

Open a terminal window, enter the **aloha** Maven project folder, and package the Aloha microservice as a Fat JAR:

```
[student@workstation ~]$ cd ~/hello-microservices/aloha
[student@workstation aloha]$ mvn clean package -DskipTests
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ aloha ---
...
[INFO] BUILD SUCCESS
...
```

Run the Aloha microservice, passing the WildFly Swarm system property that specifies an alternate HTTP port:

```
[student@workstation aloha]$ java -jar target/aloha-swarm.jar \
-Dswarm.http.port=7070
...
2018-07-11 13:36:04,139 INFO  [org.jboss.as.server] (main) WFLYSRV0010: Deployed
"aloha.war" (runtime-name : "aloha.war")
2018-07-11 13:36:04,156 INFO  [org.wildfly.swarm] (main) WFSWARM99999: WildFly
Swarm is Ready
```

Leave the Aloha microservice running.

- 3.4. Start the Hola microservice.

Open a second terminal window, enter the **hola** Maven project folder, and package the Aloha microservice as a Fat JAR. Do not clean the project so that you are not forced to refresh the project in the IDE:

```
[student@workstation ~]$ cd ~/hello-microservices/hola
[student@workstation hola]$ mvn package -DskipTests
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ hola ---
...
[INFO] BUILD SUCCESS
...
```

Run the Hola microservice:

```
[student@workstation hola]$ java -jar target/hola-swarm.jar
...
2018-07-11 13:36:59,892 INFO [org.jboss.as.server] (main) WFLYSRV0010: Deployed
"hola.war" (runtime-name : "hola.war")
2018-07-11 13:36:59,898 INFO [org.wildfly.swarm] (main) WFSWARM99999: WildFly
Swarm is Ready
```

Leave the Hola microservice running.

- 3.5. Test the Hola microservice and verify that it successfully invokes the Aloha microservice.

Open a third terminal window and use the **curl** command to invoke the **/api/hola-chaining** endpoint:

```
[student@workstation ~]$ curl -si http://localhost:8080/api/hola-chaining
HTTP/1.1 200 OK
...
["Hola de localhost\n","Aloha mai localhost\n"]
```

- 3.6. Leave the Aloha microservice running. Stop only the Hola microservice using **Ctrl+C**.

► 4. Implement a circuit-breaker class that invokes the Aloha microservice.

- 4.1. Complete the Hystrix command class.

Open the **HolaResource** class in the **com.redhat.training.msa.hola.rest** package. Scroll down until you find the **HolaChainingCommand** inner class.

Configure a Hystrix circuit breaker by invoking the **HystrixCommand** constructor from the parent class, using the following parameters:

- Command key: **HolaChainingCommand**
- Request Volume Threshold: **2**
- Circuit breaker sleep window: **5000** ms

As the command key is reused later, declare it as a constant named **HOLA_CHAINING_COMMAND_KEY** in the inner class.

Use the following listing as a reference for the changes to the inner class:

```
...
public class HolaResource {
```

```

public static class HolaChainingCommand extends HystrixCommand<String> {

    public final static String
        HOL_A_CHAINING_COMMAND_KEY="HolaChainingCommand";
    private AlohaService alohaService;

    public HolaChainingCommand(AlohaService proxy) {
        super(Setter
            .withGroupKey(HystrixCommandGroupKey.Factory.asKey("group"))
            .andCommandKey(HystrixCommandKey.Factory.asKey(
                HOL_A_CHAINING_COMMAND_KEY))
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter()
                    .withCircuitBreakerRequestVolumeThreshold(2)
                    .withCircuitBreakerSleepWindowInMilliseconds(5000))
        );
        this.alohaService=alohaService;
    }
    ...
}

```

- 4.2. Complete the **run** method in the inner class.

Replace the **return null** statement with a call to the Aloha microservice proxy:

```

public class HolaResource {
    ...
    public static class HolaChainingCommand extends HystrixCommand<String>{
        ...
        @Override
        protected String run() throws Exception {
            return alohaService.aloha();
        }
    }
    ...
}

```

- 4.3. Inspect the test case for the **HolaChainingCommand** class.

Inspect the **HolaCircuitBreakerTest** class from the **com.redhat.training.msa.hola** package.

The **HolaCircuitBreakerTest** class uses the Mockito framework to mock the Aloha microservice proxy. The test validates whether the circuit breaker opens after invoking the **holaChaining** method from the Hola microservice. To open the circuit breaker, the mock implementation throws a **RuntimeException** exception when the Aloha service is invoked.

- 4.4. Run the test methods for the **HolaCircuitBreakerTest** test case. The only test method pass.
- 5. Add the circuit breaker pattern to the **/api/hola-chaining** endpoint and complete the Hystrix configuration for the Hola microservice.
- 5.1. Complete the **holaChaining** method to use the **HystrixCommand** object.

Continue editing the **HolaResource** class in the **com.redhat.training.msa.hola.rest** package.

Create a **HolaChainingCommand** object and invokes its **execute** method:

```
public class HolaResource {
    ...
    @GET
    @Path("/hola-chaining")
    @Produces("application/json")
    public List<String> holachaining() {
        List<String> greetings = new ArrayList<>();
        greetings.add(hola());
        greetings.add(new HolaChainingCommand(alohaService).execute());
        return greetings;
    }
}
```

- 5.2. Inspect the **HystrixConfig** managed bean in the **com.redhat.training.msa.hola.rest** package. It uses WildFly Swarm's thread pool to allow the Hystrix circuit breaker implementation to accept multiple requests simultaneously.

- 5.3. Add the Hystrix Servlet dependency in the project's Maven POM file.

Open the **pom.xml** file from the **hola** project.

Add the **org.wildfly.swarm:hystrix** Maven artifact as a dependency just after the **org.wildfly.swarm:microprofile-config** Maven artifact declaration in your project POM, as follows:

```
...
<dependencies>
    ...
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>microprofile-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>hystrix</artifactId>
        <version>${version.wildfly.swarm.community}</version>
    </dependency>
    ...

```

▶ 6. Test the circuit breaker functionality of the Hola microservice.

Recall that you left the Aloha microservice running in Step 3.3. If you stopped it, you need to start the Aloha microservice again now.

- 6.1. Repackage and restart the Hola microservice.

```
[student@workstation ~]$ cd ~/hello-microservices/hola
[student@workstation hola]$ mvn clean package -DskipTests
...
[INFO] --- wildfly-swarm-plugin:7.1.0.redhat-77:package (default) @ hola ---
...
[INFO] BUILD SUCCESS
...
[student@workstation hola]$ java -jar target/hola-swarm.jar
```

```
...
2018-07-11 13:36:59,892 INFO [org.jboss.as.server] (main) WFLYSRV0010: Deployed
  "hola.war" (runtime-name : "hola.war")
2018-07-11 13:36:59,898 INFO [org.wildfly.swarm] (main) WFSWARM99999: WildFly
  Swarm is Ready
```

6.2. Monitor the circuit breaker stream.

Open new a terminal window and use the **curl** command to invoke the Hystrix Servlet.

```
[student@workstation ~]$ curl http://localhost:8080/hystrix.stream
ping:
ping:
...
```

Leave the window opened and the **curl** command running so you can see when the circuit breaker shows updated metrics. If the **curl** command stops, simply run it again.

6.3. Switch to another terminal window and use the **curl** command to invoke the **/api/hola-chaining** entrypoint.

Because the Aloha microservice is running, the circuit breaker should make no difference in the output:

```
[student@workstation ~]$ curl -si http://localhost:8080/api/hola-chaining
HTTP/1.1 200 OK
...
["Hola de localhost\n", "Aloha mai localhost\n"]
```

6.4. Switch to the terminal monitoring the circuit breaker stream to inspect the circuit breaker metrics.

The output changes from empty **ping:** entries to **data:** entries with lots of data. Note that the data entry **HolaChainingCommand** has the following attributes: **isCircuitBreakerOpen** is false, **errorCount** is 0, and **requestCount** is 1.

```
[student@workstation ~]$ curl http://localhost:8080/hystrix.stream
...
ping:
...
data: {"type":"HystrixCommand", "name":"HolaChainingCommand", "group":"group",
"currentTime":1531334090593, "isCircuitBreakerOpen":false, "errorPercentage":0,
"errorCount":0, "requestCount":1, ...}
```

After a few moments, Hystrix resets its in-memory metrics, and you see that the **requestCount** attribute is back to 0:

```
...
data: {"type": "HystrixCommand", "name": "HolaChainingCommand", "group": "group",
"currentTime": 1531334090593, "isCircuitBreakerOpen": false, "errorPercentage": 0,
"errorCount": 0, "requestCount": 0, ...
...
```

- 6.5. Stop the Aloha microservice and use the **curl** command to invoke the **/api/hola-chaining** endpoint multiple times.

The Hola microservice returns a **ConnectException** for the first attempt because the Aloha microservice is no longer running:

```
[student@workstation ~]$ curl -si http://localhost:8080/api/hola-chaining
HTTP/1.1 500 Internal Server Error
...
Caused by: java.net.ConnectException: Connection refused (Connection refused)
```

After a few tries, the Hola microservice returns a **RuntimeException** stating that the circuit is open, and no attempt is made to invoke the Aloha microservice:

```
[student@workstation ~]$ curl -si http://localhost:8080/api/hola-chaining
HTTP/1.1 500 Internal Server Error
...
Caused by: java.lang.RuntimeException: Hystrix circuit short-circuited and is OPEN
```

After some time, the circuit closes and the Hola microservice returns again a **ConnectException**.

If you continue invoking the **/api/hola-chaining** endpoint, you see the circuit open and close. If you restart the Aloha service, and wait until the circuit breaker closes the circuit, you see again the combined outputs of the Hola and Aloha microservices.

- 6.6. When you invoke the **/api/hola-chaining** endpoint with the Aloha service stopped, the metrics stream changes to show that the **isCircuitBreakerOpen** attribute is true and **errorCount** is higher than 0, for example:

```
...
data: {"type": "HystrixCommand", "name": "HolaChainingCommand", "group": "group",
"currentTime": 1531334851138, "isCircuitBreakerOpen": true, "errorPercentage": 100,
"errorCount": 1, "requestCount": 1, ...
...
```

If you take too long to see switch the terminal that is monitoring the metrics stream, Hystrix may have already reset its metrics and you see that the circuit is closed.

- 6.7. Stop the **curl** command, and also the Aloha and Hola microservices using **Ctrl+C**.

▶ 7. Clean up.

- 7.1. Commit your changes to your local Git repository:

```
[student@workstation hola]$ git commit -a -m \
  "Finished the exercise."
[do292-fault-tolerance-begin] Finished the exercise
```

7.2. Remove the **hola** project from the IDE workspace.

► **8.** Optional clean up.

If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation hola]$ git reset --hard \
  origin/do292-fault-tolerance-begin
```

This concludes the guided exercise.

Deploying Hystrix Dashboard and Turbine

Objectives

After completing this section, students should be able to deploy Hystrix Dashboard and Turbine server on an OpenShift cluster to monitor Hystrix-based applications.

Monitoring Circuit Breaker Status with Hystrix Dashboard

The Hystrix metrics endpoint provides a continuous stream of data about each circuit breaker, such as the time to execute a method call, the status of the circuit (opened or closed), and detailed statistics about the threshold parameters.

The information is provided as raw JSON data that can be consumed by any monitoring tool.

The *Hystrix Dashboard* is a tool that receives data from a Hystrix metrics endpoint and provides a graphical representation of the data feed in a web user interface. The Hystrix Dashboard is available as a container image from the *Fabric8 Kubeflix* project.

To deploy Hystrix Dashboard in OpenShift, you may use the following command:

```
$ oc new-app --name hystrix-dashboard \
  --docker-image registry/fabric8/hystrix-dashboard:1.0.28
```

To enable external access to Hystrix, expose the **hystrix-dashboard** service as an OpenShift route.

```
$ oc expose svc hystrix-dashboard \
  --hostname hystrix.apps.lab.example.com
```

Note

The Hystrix Dashboard needs Internet access to download JavaScript libraries from a content distribution network (CDN) to render the web user interface. The classroom for this course does not provide Internet access, so the dashboard does not load or function as expected.

The home page of the dashboard:



Figure 6.1: Hystrix dashboard Welcome page

To start displaying statistics and graphs, the Hystrix Dashboard requires information, such as:

- Metrics endpoint URL: The URL where an application publishes Hystrix circuit breaker data
- Delay: How frequently the dashboard is refreshed
- Authorization: The authentication token used to access the Hystrix data

The only mandatory field is the URL. After clicking **Monitor Stream**, Hystrix Dashboard connects to the Hystrix REST API and renders any information captured by Hystrix.

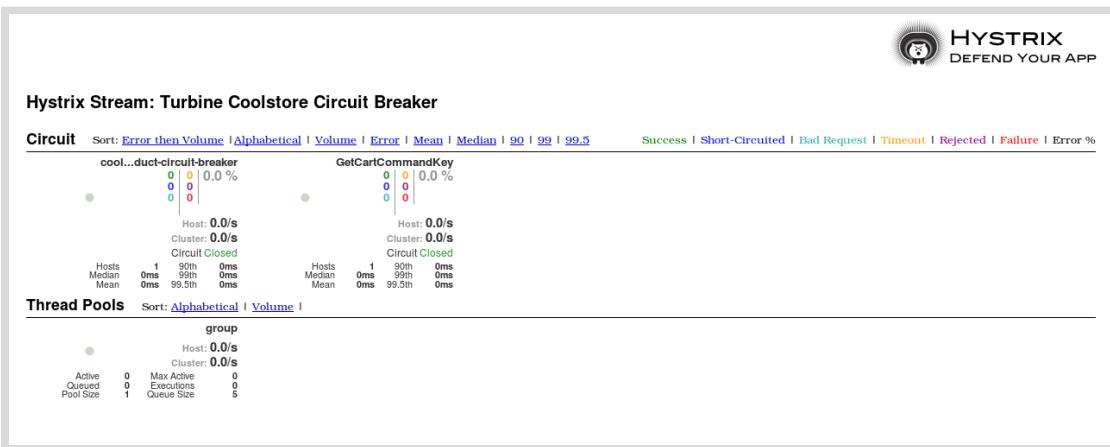


Figure 6.2: Hystrix Dashboard circuit breaker metrics

If the Hystrix Dashboard does not show any table or graphics, but only a **loading** message, this means that the target application has not invoked any circuit breaker. The Hystrix metrics endpoint only provides data about **HystrixCommand** object that were invoked by the application.

Aggregating Circuit Breaker Metrics with Turbine Server

In a large deployment, there are many microservices that use circuit breakers and each microservice provides its own Hystrix metrics stream. The Hystrix Dashboard is able to monitor only one metrics endpoint at a time.

The *Turbine Server* aggregates information from multiple metrics streams so that the Hystrix Dashboard can monitor only the aggregated metrics stream provided by Turbine to display information about multiple microservices.

Turbine is able to discover microservices using service registries such as Eureka from Netflix OSS, and the Fabric8 Kubeflix project adds to Turbine the ability to discover services and pods deployed on an OpenShift cluster.

To monitor Hystrix circuit breakers embedded in microservices deployed on OpenShift, Turbine requires:

- A list of the target projects to monitor, and of the services and pods inside each project that provide a Hystrix metrics stream.
- View permission in the target projects, so Turbine can list available pods and services, and get their IP addresses. Assign this permission to the service account that runs the Turbine pod.
- View permission in its own project, so Turbine can read its configuration from a configuration map object.

Turbine fetches the list of target projects, services, and pods from a YAML configuration file, which can be embedded into an OpenShift configuration map:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: turbine-server
data:
  application.yml: |
    turbine.instanceUrlSuffix: :8080/hystrix.stream
    turbine.aggregator.clusterConfig: clusterName
    turbine.aggregator.clusters.clusterName: prjName1.svcName1,prjName2.svcName2
```

Notice that the **turbine.aggregator.clusters** entry must be provided as a single text line.

The *clusterName* is an identifier used internally by Turbine to group statistics from multiple metrics streams. Both *prjName1* and *prjName2* are OpenShift project names. *svcName1* and *svcName2* are OpenShift service and pod names inside their respective projects. You can add many more pairs of *prjName.svcName* using commas.

For example, to monitor the **gateway** and the **ui** services from the **frontend** project, and group their statistics into the **web** cluster. Use the following configuration map:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: turbine-server
data:
  application.yml: |
```

```
turbine.instanceUrlsuffix: :8080/hystrix.stream
turbine.aggregator.clusterConfig: web
turbine.aggregator.clusters.web: frontend.gateway,frontend.ui
```

If Turbine is monitoring services and pods inside its own project, it requires that the **default** service account has the **view** role in the project:

```
$ oc policy add-role-to-group view -z default
```

If Turbine is monitoring services and pods from different projects, it requires access to all these projects. If Turbine runs as part of the *turbine-server-project* and needs to monitor applications from the *application-project*, the following command assigns the required **view** role:

```
$ oc policy add-role-to-group view \
  system:serviceaccounts:turbine-server-project \
  -n application-project
```

Identifying Hystrix Instances Monitored by a Turbine Server

Turbine also requires that the OpenShift service and pod resources include the following two labels:

- **hystrix.enabled**: the true value means the service or pod is discovered, and the false value means the service or pods is ignored by Turbine.
- **hystrix.cluster**: the name of the cluster that organizes and aggregates data from the service or pod.

Add these labels to the **metadata** of a service resource and to the pod **template** inside a deployment configuration resource.

The following listing shows an example service resource fragment with the Turbine labels:

```
apiVersion: "v1"
kind: "Service"
metadata:
  labels:
    hystrix.enabled: true
    hystrix.cluster: web
spec:
  ports:
    ...

```

The following listing shows an example deployment configuration resource fragment with Turbine labels:

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        hystrix.enabled: true
```

```

hystrix.cluster: web
spec:
  containers:
    ...

```

Deploying Turbine Server on OpenShift

After creating a configuration map with the Turbine server configuration, and enabling access to the target projects, deploy the Turbine Server image on an OpenShift cluster:

```

$ oc new-app --name turbine-server \
--docker-image registry/fabric8/turbine-server:1.0.28

```

To enable external access to Turbine, expose the **turbine-service** service as an OpenShift route.

```

$ oc expose svc turbine-server \
--hostname turbine.apps.lab.example.com

```

Accessing Turbine Data from Hystrix Dashboard

In Hystrix Dashboard, you need to use the Hystrix metrics endpoint provided by Turbine instead of each microservice's Hystrix metrics endpoint. The URL for the Turbine metrics endpoint is:

`http://turbine-server-hostname/turbine.stream?cluster=clusterName`

For example, if the Turbine server is exposed as a route with the **turbine.apps.lab.example.com** host name, and the *clusterName* defined in the Turbine configuration is **web**, the URL is:

`http://turbine.apps.lab.example.com/turbine.stream?cluster=web`

To list which Hystrix instances are monitored by Turbine Server, access the `discovery` URI from Turbine Server, for example:

`http://turbine.apps.lab.example.com/discovery`



References

Hystrix Dashboard project page

<https://github.com/Netflix-Skunkworks/hystrix-dashboard>

Turbine Server project page

<https://github.com/Netflix/Turbine/wiki>

Fabric8 Kubeflix project page

<https://github.com/fabric8io/kubeflix>

► Guided Exercise

Deploying Hystrix Turbine

In this exercise, you will deploy Hystrix Turbine on OpenShift to monitor a microservice that implements a Hystrix-based circuit breaker.

Outcomes

You should be able to monitor the Hystrix circuit breaker from the Hola microservice.

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The Hystrix Turbine container image (**fabric8/turbine-server:1.0.28**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-turbine-*** branches in the classroom Git repository, containing the source code for the microservice Maven project.

You can compare your source code changes while performing this exercise with the branch **do292-turbine-solution** of the **hello-microservices** Git repository.

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab hystrix-turbine setup
```

You can compare your source code changes while performing this lab with the solution branch **do292-turbine-solution** of the **hello-microservices** repository.

- 1. Switch to the **do292-turbine-begin** branch in the **hello-microservices** Git repository.

- 1.1. If you have not done so yet, clone the **hello-microservices** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **hello-microservices** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/hello-microservices
Cloning into 'hello-microservices'...
...
```

- 1.2. If you already have a clone of the **hello-microservices** project, either commit or stash your local changes.
- 1.3. Switch to the **do292-turbine-begin** branch:

```
[student@workstation ~]$ cd ~/hello-microservices
[student@workstation hello-microservices]$ git checkout \
  do292-turbine-begin
...
Switched to a new branch 'do292-turbine-begin'
```

► 2. Deploy the Aloha microservice on OpenShift.

- 2.1. Log in to OpenShift and create the **hola-monitor** project:

```
[student@workstation hello-microservices]$ oc login -u developer -p redhat \
  https://master.lab.example.com
Login successful.
...
[student@workstation hello-microservices]$ oc new-project hola-monitor
Now using project "hola-monitor" on server "https://master.lab.example.com:443".
...
```

- 2.2. Deploy the Aloha microservice in the OpenShift.

Open a terminal window and use the Fabric8 Maven Plug-in (FMP) to deploy the Aloha microservice:

```
[student@workstation ~]$ cd ~/hello-microservices/aloha
[student@workstation aloha]$ mvn clean fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ aloha ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator wildfly-swarm
...
[INFO] F8: Creating BuildServiceConfig aloha-s2i for Source build
[INFO] F8: Creating ImageStream aloha
[INFO] F8: Starting Build aloha-s2i
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ aloha ---
...
[INFO] Creating a Service from openshift.yml namespace hola-monitor name aloha
...
[INFO] Creating a ConfigMap from openshift.yml namespace hola-monitor name aloha
...
[INFO] Creating a DeploymentConfig from openshift.yml namespace hola-monitor name
aloha
...
[INFO] Creating Route hola-hystrix:hola host: aloha.apps.lab.example.com
...
[INFO] BUILD SUCCESS
```

You can safely ignore any timeout warnings and also errors caused by a **RejectedExecutionException** exception.

- 2.3. Verify that the Aloha microservice is operational.

Wait for the Aloha microservice pod to be ready and running, and verify that it is accessible:

```
[student@workstation aloha]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
aloha-1-zj5fg 1/1     Running   0          1m
aloha-s2i-1-build 0/1     Completed  0          2m
[student@workstation aloha]$ curl -si \
  http://aloha.apps.lab.example.com/api/aloha
HTTP/1.1 200 OK
...
Aloha mai aloha.apps.lab.example.com
```

- ▶ 3. Complete and deploy the Hola microservice on OpenShift.

- 3.1. Add the labels required by Turbine to the service resource fragment file.

Open the `~/hello-microservices/hola/src/main/fabric8/service.yml` file in a text editor and add the `hystrix.enabled` and `hystrix.cluster` labels to the service:

Use the following listing as a reference for the changes to the `service.yml` file:

```
apiVersion: "v1"
kind: "Service"
metadata:
  labels:
    hystrix.enabled: true
    hystrix.cluster: hola
spec:
  ports:
    - name: 8080-tcp
      protocol: TCP
      port: 8080
      targetPort: 8080
```

- 3.2. Add the labels required by Turbine to the deployment configuration resource fragment file.

Open the `~/hello-microservices/hola/src/main/fabric8/deployment.yml` file with a text editor and add the `hystrix.enabled` and `hystrix.cluster` labels to the pod template embedded in the deployment configuration:

Use the following listing as a reference for the changes to the `deployment.yml` file:

```
spec:
  replicas: 1
  template:
    metadata:
      labels:
        hystrix.enabled: true
        hystrix.cluster: hola
    spec:
```

```
containers:
- env:
  ...
```

3.3. Deploy the Hola microservice in the OpenShift cluster.

Open a terminal window and use the Fabric8 Maven Plug-in (FMP) to deploy the Hola microservice:

```
[student@workstation aloha]$ cd ~/hello-microservices/hola
[student@workstation hola]$ mvn clean fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (fmp) @ hola ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator wildfly-swarm
...
[INFO] F8: Creating BuildServiceConfig hola-s2i for Source build
[INFO] F8: Creating ImageStream hola
[INFO] F8: Starting Build hola-s2i
...
[INFO] --- fabric8-maven-plugin:3.5.38:deploy (default-cli) @ hola ---
...
[INFO] Creating a Service from openshift.yml namespace hola-monitor name hola
...
[INFO] Creating a ConfigMap from openshift.yml namespace hola-monitor name app-
config
...
[INFO] Creating a DeploymentConfig from openshift.yml namespace hola-monitor name
hola
...
[INFO] Creating Route hola-hystrix:hola host: hola.apps.lab.example.com
...
[INFO] BUILD SUCCESS
```

You can safely ignore any timeout warnings and also errors caused by a **RejectedExecutionException** exception.

3.4. Wait until the Hola microservice pod is ready and running.

```
[student@workstation hola]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...
hola-1-9nppt   1/1     Running   0          12s
hola-s2i-1-build 0/1     Completed  0          47s
```

3.5. Verify that a configuration map is created, and that it points to the Aloha microservice through its OpenShift service:

```
[student@workstation hola]$ oc describe cm app-config
...
Data
=====
project-defaults.yaml:
...
```

```
alohaHostname: "aloha.apps.lab.example.com"
alohaPort: "80"
...
```

- 3.6. Verify that the service and the pod from the Hola microservice contain the labels required by Turbine:

```
[student@workstation hola]$ oc get svc hola --show-labels
NAME ... LABELS
hola ... app=hola,expose=true,group=com.redhat.training.msa,
hystrix.cluster=hola,hystrix.enabled=true,provider=fabric8,version=1.0

[student@workstation hola]$ oc get pod -l app=hola --show-labels
NAME      ... LABELS
hola-1-9nprt ... app=hola,deployment=hola-1,deploymentconfig=hola,
group=com.redhat.training.msa,hystrix.cluster=hola,hystrix.enabled=true,
provider=fabric8,version=1.0
```

- 3.7. Verify that the Hystrix Servlet is monitoring the circuit breakers in the Hola microservice.

From the terminal window, run the following command:

```
[student@workstation hola]$ curl http://hola.apps.lab.example.com/hystrix.stream
ping:
ping:
ping:
...
```

Leave the **curl** command running.

- 3.8. Invoke the /api/hola-chaining endpoint to start generating the circuit breaker information.

From another terminal window, run the following command

```
[student@workstation ~]$ curl -si \
  http://hola.apps.lab.example.com/api/hola-chaining
HTTP/1.1 200 OK
...
["Hola de hola.apps.lab.example.com\n", "Aloha mai aloha.apps.lab.example.com\n"]
```

- 3.9. Verify the Hystrix metrics stream.

In the window running the **curl** command, an output similar to the following is expected because you have invoked the circuit breaker.

```

...
ping:

data: {"type":"HystrixCommand", "name":"HolaChainingCommand", "group":"group", ...}
...
data: {"type":"HystrixThreadPool", "name":"group", "currentTime":1530125188523, ...}
...

```

- 3.10. Switch to the terminal window monitoring the Hystrix metrics stream and press **Ctrl+C** to stop the **curl** command.

▶ 4. Deploy Hystrix Turbine on OpenShift.

- 4.1. Complete the Turbine configuration map.

Open the **~/hello-microservices/hola/ocp/turbine-cm.yml** file with a text editor and complete it so that the cluster configuration named **hola** points to the service and pods named **hola** inside the **hola-monitor** project.

Use the following listing as a reference for the changes to the **turbine-cm.yml** file:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: turbine-server
data:
  application.yml: |
    turbine.instanceUrlSuffix: :8080/hystrix.stream
    turbine.aggregator.clusterConfig: hola
    turbine.aggregator.clusters.hola: hola-monitor.hola

```

Note that the value of the **turbine.aggregator.clusterConfig** attribute in the previous listing is equal to the value of the **hystrix.cluster** label in the Hola microservice pod.

- 4.2. Create the Turbine configuration map.

Open a terminal window, enter the **hola** Maven project, and use the **oc create** command:

```

[student@workstation ~]$ cd ~/hello-microservices/hola
[student@workstation hola]$ oc create -f ocp/turbine-cm.yml
configmap "turbine-server" created

```

- 4.3. Verify the values in the Turbine configuration match the values in the resource fragment file:

```

[student@workstation hola]$ oc describe cm turbine-server
...
Data
=====
application.yml:
-----
turbine.instanceUrlSuffix: :8080/hystrix.stream

```

```
turbine.aggregator.clusterConfig: hola
turbine.aggregator.clusters.hola: hola-monitor.hola
...
```

4.4. Assign permission to use the OpenShift Master API to the **default** service account:

```
[student@workstation hola]$ oc policy add-role-to-user view -z default
role "view" added: "default"
```

4.5. Verify that the **default** service account has the **view** role:

```
[student@workstation hola]$ oc get rolebindings view
NAME      ROLE      USERS      GROUPS      SERVICE ACCOUNTS      SUBJECTS
view      /view          default
```

4.6. Deploy Turbine from the container image in the private registry:

```
[student@workstation hola]$ oc new-app --name turbine \
--docker-image registry.lab.example.com/fabric8/turbine-server:1.0.28
--> Found Docker image 3d8f073 (17 months old) from registry.lab.example.com for
"registry.lab.example.com/fabric8/turbine-server:1.0.28"
...
--> Creating resources ...
  imagestream "turbine" created
  deploymentconfig "turbine" created
  service "turbine" created
--> Success
...
```

► 5. Verify that Turbine can fetch metrics from the Hola microservice.

5.1. Wait for the Turbine pod to be ready and running.

```
[student@workstation hola]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
...
turbine-1-k9f9k   1/1     Running   0          14s
```

5.2. Review the Turbine server logs to verify that it found the Hola microservice:

```
[student@workstation hola]$ oc logs turbine-1-k9f9k
...
2018-07-12 13:05:01.916 INFO [bootstrap,,,] 1 --- [           main]
  b.c.PropertySourceBootstrapConfiguration : Located property source:
    ConfigMapPropertySource [name='configmap.turbine-server.hola-monitor']
...
2018-07-12 13:05:10.590 INFO [bootstrap,,,] 1 --- [       Timer-0]
  c.n.t.discovery.InstanceObservable      : Hosts up:1, hosts down: 0
2018-07-12 13:05:10.596 INFO [bootstrap,,,] 1 --- [       Timer-0]
  c.n.t.monitor.instance.InstanceMonitor : Url for host: http://10.130.0.70:8080/
hystrix.stream hola
...
```

5.3. Expose the Turbine server for external access:

```
[student@workstation hola]$ oc expose svc turbine \
  --hostname turbine.apps.lab.example.com
route "turbine" exposed
```

5.4. Access the metrics stream from the Turbine server, using the **/turbine.stream?cluster=hola** endpoint:

```
[student@workstation hola]$ curl \
  http://turbine.apps.lab.example.com/turbine.stream?cluster=hola
...
: ping
data: {"rollingCountFallbackSuccess":0,"rollingCountFallbackFailure":0,...}
...
data: {"currentCorePoolSize":10,"currentLargestPoolSize":2,...}
...
data: {"reportingHostsLast10Seconds":1,"name":"meta", ...}
...
```

Press **Ctrl+C** to stop the **curl** command.

► 6. Clean up.

6.1. Delete the **hola-monitor** OpenShift project.

```
[student@workstation ~]$ oc delete project hola-monitor
project "hola-monitor" deleted
```

6.2. Commit your changes to your local Git repository:

```
[student@workstation aloha]$ git commit -a -m \
  "Finished the exercise."
```

► 7. Optional clean up.

If you wish to start over this exercise, reset your local repository to the remote branch:

```
[student@workstation aloha]$ git reset --hard \
origin/do292-turbine-begin
```

This concludes the guided exercise.

► Guided Exercise

Implementing Fault Tolerance with Hystrix

In this exercise, you will implement a circuit breaker in the API Gateway microservice using Hystrix, and monitor the status of the circuit breaker using Turbine server.

Outcomes

You should be able to:

- Configure and implement a circuit breaker in the API Gateway microservice.
- Monitor the circuit breaker using the Hystrix metrics stream endpoint to identify whether the circuit breaker is open or closed.
- Deploy Turbine server to capture metrics and aggregate information from multiple circuit breakers.
- Configure the microservices deployed on OpenShift to automatically provide metrics to Turbine server.

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The API Gateway, Cart and Catalog microservices deployed and running on OpenShift. You need to perform the following exercises before attempting this lab:
 - The guided exercise *Developing Microservices with the Vert.x Runtime* starting from Step 10
 - The guided exercise *Developing Microservices with the Spring Boot Runtime* starting from Step 9
 - The Lab *Developing an API Gateway*
 - For each of these labs, you may also need to perform the *Before you begin* section and review the first step to make sure you have their prerequisites set up and are starting from the correct solution branch
- The Red Hat OpenJDK 1.8 S2I builder image (**redhat-openjdk-18/openjdk18-openshift**).
- The kubeflix Turbine Server container image (**fabric8/turbine-server:1.0.28**).
- The **redhat-openjdk18-openshift** image stream.
- The **do292-hystrix-*** branches in the classroom Git repository, containing the source code for the **coolstore-gateway** microservice Maven project.
- The **org.wildfly.swarm:bom, io.fabric8:fabric8-maven-plugin**, and **org.wildfly.swarm:wildfly-swarm-plugin** Maven artifacts in the classroom Nexus proxy server

If you need help using Git and JBoss Developer Studio, refer to Appendix A, *Managing Git Branches* and Appendix B, *Working With Red Hat JBoss Developer Studio*.

Run the following command on the **workstation** VM to validate the exercise prerequisites:

```
[student@workstation ~]$ lab hystrix-deploy setup
```

You can compare your source code changes while performing this lab with the solution branch **do292-hystrix-lab-solution** of the **coolstore** repository.

- 1. Switch to the **do292-hystrix-lab-begin** branch in the **coolstore** Git repository and import the project into JBoss Developer Studio.

- 1.1. If you have not done so yet, clone the **coolstore** project from the classroom Git repository.

From the home directory of the **student** user on the **workstation** VM, clone the **coolstore** project from the classroom Git repository:

```
[student@workstation ~]$ git clone \
  http://services.lab.example.com/coolstore
Cloning into 'coolstore'...
...
```

- 1.2. If you already have a clone of the **coolstore** project, either commit or stash your local changes.

- 1.3. Switch to the **do292-hystrix-lab-begin** branch:

```
[student@workstation ~]$ cd ~/coolstore
[student@workstation coolstore]$ git checkout do292-hystrix-lab-begin
...
Switched to a new branch 'do292-hystrix-lab-begin'
```

- 1.4. Open the Red Hat JBoss Developer Studio IDE and, if you have not already done so, import the **~/coolstore/coolstore-gateway-jaxrs** folder as a Maven project.

- 1.5. If you imported the project before, refresh the project and update its configuration. Make sure that the project displays the correct branch name, which is **do292-hystrix-lab-begin**.

- 1.6. Ignore errors in the starter project.

The IDE builds the new **coolstore-gateway** project and reports syntax errors in the **com.redhat.coolstore.gateway.api.tests** package. Ignore them for now. You will fix these errors as you proceed with this exercise.

- 2. In the previous chapter, you implemented an API gateway for the coolstore application that proxied requests to the Cart and Catalog microservices. In this exercise, you will implement circuit breakers in the API gateway for methods that invoke the Cart microservice.

Inspect the initial state of the **coolstore-gateway** project in the IDE.

- 2.1. Review the WildFly Swarm thread pool configuration that allows Hystrix to accept multiple requests simultaneously.

Inspect the **HystrixConfig** class in the **com.redhat.coolstore.gateway.api** package.

Review the code in the **onStartup()** method. It registers a new **ThreadPoolExecutor** instance using the **HystrixConcurrencyStrategy** class.

- 2.2. Open the **CartGateway** class in the **com.redhat.coolstore.gateway.api** package in the IDE. The circuit breaker is fully implemented for the **getCart** method in this class. Do not make any changes to the **getCart** method.
- A **GetCartCommand** static inner class that extends the **HystrixCommand** class is declared as follows:

```
...
public class CartGateway {
...
    public static class GetCartCommand extends HystrixCommand<ShoppingCart> {
        public final static String GET_CART_COMMAND_KEY="GetCartCommandKey";
        private String cartId;
        private CartResource proxy;

        public GetCartCommand(CartResource proxy, String cartId) {
            super(Setter
                .withGroupKey(HystrixCommandGroupKey.Factory.asKey("group"))
                .andCommandKey(HystrixCommandKey.Factory.asKey(GET_CART_COMMAND_KEY))
                .andCommandPropertiesDefaults(
                    HystrixCommandProperties
                        .Setter()
                        .withCircuitBreakerRequestVolumeThreshold(2)
                        .withCircuitBreakerSleepWindowInMilliseconds(5000)));
        }
...
}
```

The **run** method in the inner class delegates the call to the Cart microservice.

```
...
@Override
protected ShoppingCart run() throws Exception {
    return proxy.getCart(cartId);
}
...
```

Finally, the **getCart** method from the **CartGateway** class delegates the execution to the inner class. It instantiates the **GetCartCommand** static inner class and invokes the **execute** method.

```
...
public ShoppingCart getCart(@PathParam("cartId") String cartId) {
    CartResource proxy = buildClient();
    return new GetCartCommand(proxy,cartId).execute();
}
...
```

- 2.3. Inspect the test case for the **GetCartCommand** class.

Inspect the **CartGatewayGetCartCircuitBreakerTest** class in the **com.redhat.coolstore.gateway.api.tests** package.

Do not make any changes to this class. The

CartGatewayGetCircuitBreakerTest class uses the Mockito framework to mock implementations of the **CartResource** interface, so it mimics the Cart microservice calls.

- 2.4. Run the **CartGatewayGetCartCircuitBreakerTest** test case from the **com.redhat.coolstore.gateway.api.tests** package as a JUnit test and ignore the warning from the IDE about the errors in the project. All the tests should pass.

- 3. Implement the circuit breaker pattern for the **addToCart** method. Use the code from the circuit breaker implementation for the **getCart** method as a reference to implement this step.

- 3.1. Change the **addToCart** method from the **CartGateway** class in the **com.redhat.coolstore.gateway.api** package to support the circuit breaker pattern with Hystrix.

Create a static inner class in the **CartGateway** class that extends the **HystrixCommand** class, and use the **ShoppingCart** as the generic type.

Immediately after the **addToCart** method, create the **AddToCartCommand** static inner class that extends the **HystrixCommand** class. Use the following listing as a reference:

```
...
public class CartGateway {
...
    public ShoppingCart addToCart(@PathParam("cartId") String cartId,
        @PathParam("itemId") String itemId,
        @PathParam("quantity") int quantity) {
...
    }
...
    public static class AddToCartCommand extends HystrixCommand<ShoppingCart>{
}
...
}
```

You can ignore the error about the undefined constructor. You will fix this in the next step.

- 3.2. Configure the circuit breaker by invoking the **HystrixCommand** parent class constructor, and using the following parameters:

- Group key: **group**
- Command key: **AddToCartCommandKey**
- Request Volume Threshold: **2**
- Circuit breaker sleep window: **5000 ms**

As the command key is reused later, declare it as a constant named **ADD_TO_CART_COMMAND_KEY** in the inner class.

Use the following listing as a reference:

```
...
public class CartGateway {
    ...
    ...
    public static class AddToCartCommand extends HystrixCommand<ShoppingCart>{
        public final static String ADD_TO_CART_COMMAND_KEY="AddToCartCommandKey";

        public AddToCartCommand() {
            super(Setter
                .withGroupKey(HystrixCommandGroupKey.Factory.asKey("group"))
                .andCommandKey(HystrixCommandKey.Factory.asKey(ADD_TO_CART_COMMAND_KEY))
                .andCommandPropertiesDefaults(
                    HystrixCommandProperties
                        .Setter()
                        .withCircuitBreakerRequestVolumeThreshold(2)
                        .withCircuitBreakerSleepWindowInMilliseconds(5000)));
        }

        ...
    }
}
```

- 3.3. Add the cart ID, the item ID, the item quantity, and the **CartResource** proxy instance as constructor arguments.

Store these arguments as inner class attributes as you will use them later to invoke the proxied Cart microservice HTTP API endpoints.

Use the following listing as a reference:

```
public class CartGateway {
    ...
    public static class AddToCartCommand extends HystrixCommand<ShoppingCart>{
        public final static String ADD_TO_CART_COMMAND_KEY="AddToCartCommandKey";

        private String cartId;
        private String itemId;
        private int quantity;
        private CartResource proxy;

        public AddToCartCommand(CartResource proxy, String cartId,
                               String itemId, int quantity) {
            super(...);

            this.cartId=cartId;
            this.itemId=itemId;
            this.quantity=quantity;
            this.proxy=proxy;
        }
    }
}
```

- 3.4. Override the **run** method in the inner class to delegate the call to the Cart microservice HTTP API endpoint. Use the following listing as a reference:

```
public class CartGateway {
    ...
    public static class AddToCartCommand extends HystrixCommand<ShoppingCart>{
        ...
        public AddToCartCommand(...);
        ...
        @Override
        protected ShoppingCart run() throws Exception {
            return proxy.addToCart(cartId,itemId,quantity);
        }
        ...
    }
}
```

**Note**

Save the changes to the **CartGateway** class. At this point, the errors shown in the IDE **Problems** view are cleared.

- 3.5. Update the **addToCart** method from the **CartGateway** class to delegate the execution to the inner class.

In the **addToCart** method, instantiate the **AddToCartCommand** static inner class and invoke the **execute** method. Use the following listing as a reference:

```
public class CartGateway {
    ...
    public ShoppingCart addToCart(@PathParam("cartId") String cartId,
        @PathParam("itemId") String itemId,
        @PathParam("quantity") int quantity) {
        CartResource proxy = buildClient();
        return new AddToCartCommand(proxy,cartId,itemId,quantity).execute();
    }
}
```

- ▶ 4. Validate the circuit breaker implementation by running the provided unit tests for the API gateway microservice.
 - 4.1. Inspect the test cases for the **AddToCartCommand** class.
Inspect the **CartGatewayAddToCartCircuitBreakerTest** unit test in the **com.redhat.coolstore.gateway.api.tests** package.
Do not make any changes to this class. The **CartGatewayAddToCartCircuitBreakerTest** class uses the Mockito framework to mock implementations of the **CartResource** interface, so it mimics the Cart microservice calls. After executing the class, the test validates whether the circuit breaker opens if the Cart microservice REST endpoint is not available.
 - 4.2. Run the **CartGatewayAddToCartCircuitBreakerTest** test case from the **com.redhat.coolstore.gateway.api.tests** package as a JUnit test. All the tests should pass.
- ▶ 5. Check that Hystrix provides metrics about the circuit breaker in your local environment.

- 5.1. Inspect the `~/coolstore/coolstore-gateway-jaxrs/run.sh` file. The script sets the `CART_SERVICE_URL` and `CATALOG_SERVICE_URL` environment variables and starts the microservice locally.
- 5.2. Execute the `run.sh` script to start the microservice.

Open a terminal window and run the following command to start WildFly Swarm:

```
[student@workstation ~]$ cd ~/coolstore/coolstore-gateway-jaxrs
[student@workstation coolstore-gateway-jaxrs]$ ./run.sh
...
2018-06-15 08:30:51,812 INFO  [org.wildfly.swarm] (main) WFSWARM99999: WildFly
Swarm is Ready
```

- 5.3. Monitor the circuit breaker metrics stream.

Use the following `curl` command in a new terminal window to inspect the metrics stream.

```
[student@workstation ~]$ curl http://localhost:8080/hystrix.stream
ping:
ping:
...
```

The endpoint does not generate any information because the circuit breakers were not invoked. Leave the window open with the `curl` command running, to get the information after invoking one of the circuit breakers.

- 5.4. Invoke the circuit breaker for the `/api/cart/<cartId>` resource URL to generate metrics.

In a new terminal window, access the `/api/cart/1111` resource URL using the `curl` command.

```
[student@workstation ~]$ curl http://localhost:8080/api/cart/1111
<html><head><title>ERROR</title><style>
...
Caused by: com.netflix.hystrix.exception.HystrixRuntimeException:
GetCartCommandKey failed and no fallback available.
...
Caused by: org.apache.http.client.ClientProtocolException: URI does not specify a
valid host name: localhost:7070/cart/1111
...
```

Since the Cart microservice is not running locally on the workstation VM, the command returns an error.

- 5.5. Review the terminal window running the `curl` command monitoring the Hystrix metrics stream. The output generates a different set of messages that provides information about the circuit breaker. The circuit breaker remains in the closed state since the request volume threshold is not reached.

```
...
data: {...name:"GetCartCommandKey",..., "isCircuitBreakerOpen":false...}
data: {"type":"HystrixThreadPool", "name":"group", "currentTime":1529065899601, ...}
```

- 5.6. Invoke the `/api/cart/1111` resource URL three or more times, and observe how the circuit breaker moves to open state due to exceeded request volume threshold:

```
...
data: {...name:"GetCartCommandKey",...,isCircuitBreakerOpen:true...}
```

- 5.7. Terminate the `curl` command monitoring the Hystrix metrics endpoint.

Press `Ctrl+C` to stop the `curl` command.

- 5.8. Stop WildFly Swarm.

Press `Ctrl+C` to stop the WildFly Swarm instance.

- ▶ 6. Add the required labels to the API gateway pod and deployment configuration to enable discovery by the Turbine server.

- 6.1. Edit the deployment configuration resource fragment for the API gateway in the `src/main/fabric8/deployment.yml` file.

Add the following labels:

```
spec:
  template:
    metadata:
      labels:
        hystrix.cluster: coolstore
        hystrix.enabled: true
  spec:
    containers:
...
...
```

- 6.2. Edit the service resource fragment for the API gateway in the `src/main/fabric8/svc.yml` file.

Add the following labels:

```
metadata:
  labels:
    hystrix.cluster: coolstore
    hystrix.enabled: true
```

- ▶ 7. Redeploy the modified API gateway with circuit breakers to OpenShift, and check that the metrics endpoint provides information about the circuit breakers in the application.

- 7.1. Log in to OpenShift as the `developer` user.

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
Login successful.
...
```

7.2. Access the **api-gateway-service** OpenShift project.

```
[student@workstation ~]$ oc project api-gateway-service
Now using project "api-gateway-service" on server "https://
master.lab.example.com:443".
```

7.3. Redeploy the API Gateway microservice using the Fabric8 Maven Plug-in (FMP) on OpenShift.

Skip the tests to minimize the amount of time required to deploy the microservice. From the terminal window, run the following commands:

```
[student@workstation ~]$ cd ~/coolstore/coolstore-gateway-jaxrs
[student@workstation coolstore-gateway-jaxrs]$ mvn clean \
fabric8:deploy -DskipTests
...
[INFO] --- fabric8-maven-plugin:3.5.38:build (default) @ coolstore-gateway ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator wildfly-swarm
[INFO] F8: wildfly-swarm: Using ImageStreamTag 'redhat-openjdk18-openshift:latest'
as builder image
...
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your pods start up
[INFO] -----
[INFO] BUILD SUCCESS
```

7.4. Wait until the API Gateway pod is running. Run the **oc get pods** command and wait until the following output is displayed.

```
[student@workstation coolstore-gateway-jaxrs]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
coolstore-gateway-2-bbc9j   1/1     Running   0          1m
coolstore-gateway-s2i-2-build   0/1     Completed  0          2m
```

7.5. Inspect the API gateway pod labels. Verify that the pods have the labels required by Turbine:

```
[student@workstation coolstore-gateway-jaxrs]$ oc describe po \
coolstore-gateway-2-bbc9j | grep -A5 Labels:
Labels:      app=coolstore-gateway
              deployment=coolstore-gateway-2
              deploymentconfig=coolstore-gateway
              group=com.redhat.coolstore
              hystrix.cluster=coolstore
              hystrix.enabled=true
```

7.6. Test that the Hystrix metrics endpoint streams data. From the command line execute:

```
[student@workstation coolstore-gateway-jaxrs]$ curl \
http://coolstore-gateway.apps.lab.example.com/hystrix.stream
```

You should see the following:

```
ping:
```

```
ping:
```

```
...
```

The endpoint shows no data because you have not invoked any of the circuit breakers.

- 7.7. Invoke the **/api/cart/<cartId>** HTTP GET endpoint, to generate information about the circuit breaker added to the **getCart** method of the **CartGateway** class. From a new terminal window, run the following command:

```
[student@workstation ~]$ curl \
http://coolstore-gateway.apps.lab.example.com/api/cart/1111
{"id":"1111","cartItemTotal":0.0...}
```

Switch to the terminal window where you are monitoring the hystrix metrics endpoint. You should see the following output:

```
data: {..."name":"GetCartCommandKey",..."isCircuitBreakerOpen":false...
data: {"type":"HystrixThreadPool","name":"group","currentTime":1532088098083...}
```

Since the Cart microservice is running and servicing requests at the endpoint, the circuit breaker remains in a closed state.

- 7.8. Invoke the **/api/cart/<cartId>/<itemId>/<quantity>/** HTTP POST endpoint, to generate information about the circuit breaker added to the **addToCart** method of the **CartGateway** class. From a new terminal window, run the following command:

```
[student@workstation ~]$ curl -X POST \
http://coolstore-gateway.apps.lab.example.com/api/cart/1111/444435/2
{"id":"1111","cartItemTotal":212.0...}
```

Switch to the terminal window where you are monitoring the hystrix metrics endpoint. You should see the following output:

```
data: {..."name":"AddToCartCommandKey",..."isCircuitBreakerOpen":false...
...
data: {..."name":"GetCartCommandKey",..."isCircuitBreakerOpen":false...}
```

Since the Cart microservice is running and servicing requests at the endpoint, the circuit breaker remains in closed state.

► 8. Scale down the Cart microservice and verify circuit breaker state.

- 8.1. Use the **oc scale** command to temporarily shut down the Cart microservice:

```
[student@workstation ~]$ oc scale --replicas=0 \
dc/cart-service -n cart-service
deploymentconfig "cart-service" scaled
```

Invoke the `/api/cart/<cartId>` HTTP GET endpoint once more. The request fails with an internal server error because the Cart microservice is not running:

```
[student@workstation ~]$ curl \
http://coolstore-gateway.apps.lab.example.com/api/cart/1111
...Internal Server Error...
```

Request the same URL three more times.

- 8.2. Switch to the terminal window monitoring the hystrix metrics endpoint. You should see the following output:

```
data: {"name": "AddToCartCommandKey", "isCircuitBreakerOpen": false}
...
data: {"name": "GetCartCommandKey", "isCircuitBreakerOpen": true}
```

The **GetCartCommandKey** circuit breaker is in the open state because Hystrix detected the failure and opened the circuit to prevent further requests to the Cart microservice.

- ▶ 9. Stop monitoring the metrics endpoint.
Press **Ctrl+C** to stop the `curl` command.

- ▶ 10. Deploy the Turbine server to aggregate metrics from the API gateway microservice.

- 10.1. Create a new OpenShift project called **monitor** to deploy the Turbine server.

```
[student@workstation ~]$ oc new-project monitor
```

- 10.2. Edit the configuration map that configures Turbine to capture information from Hystrix-enabled applications running on a different project.

Edit the `~/coolstore/ocp/config-map.yml` file and change the `turbine.aggregator.clusters.coolstore` attribute as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: turbine-server
data:
  application.yml: |
    turbine.aggregator.clusters.coolstore: api-gateway-service.coolstore-gateway
    turbine.aggregator.clusterConfig: coolstore
    turbine.instanceUrlSuffix: :8080/hystrix.stream
```

- 10.3. Create the configuration map resource for Turbine:

```
[student@workstation ~]$ oc create -f ~/coolstore/ocp/config-map.yml
configmap "turbine-server" created
```

10.4. Check that the configuration map resource has been created:

```
[student@workstation ~]$ oc describe cm turbine-server
Name:          turbine-server
Namespace:    monitor
Labels:        <none>
Annotations:   <none>

Data
=====
application.yml:
-----
turbine.aggregator.clusters.coolstore: api-gateway-service.coolstore-gateway
turbine.aggregator.clusterConfig: coolstore
turbine.instanceUrlSuffix: :8080/hystrix.stream

Events:  <none>
```

10.5. Authorize pods to access resources provided by the OpenShift Master API by assigning the **view** role to the **default** service account:

```
[student@workstation ~]$ oc policy add-role-to-user \
  view -z default
role "view" added: "default"
```

Verify that the **default** service account has the **view** role:

```
[student@workstation ~]$ oc get rolebinding view
NAME      ROLE      USERS      GROUPS      SERVICE ACCOUNTS      SUBJECTS
view      /view      default
```

10.6. The Turbine server needs read permission to access resources from the project where the API gateway is deployed.

Run the **oc policy** command as follows:

```
[student@workstation ~]$ oc policy add-role-to-group view \
  system:serviceaccounts:monitor -n api-gateway-service
role "view" added: "system:serviceaccounts:monitor"
```

Verify that the **system:serviceaccounts:monitor** group has the **view** role:

```
[student@workstation ~]$ oc get rolebinding view -n api-gateway-service
NAME      ROLE      USERS      GROUPS ...
view      /view      system:serviceaccounts:monitor
```

10.7. To simplify deployment of the Turbine server, an OpenShift template is provided in the **~/coolstore/ocp/turbine-template.yml** file. Briefly review the template

file using a text editor. This template does not require any parameters. Do not make any changes to it.

Open a new terminal window and deploy the Turbine container image using the `~/coolstore/ocp/turbine-template.yml` template file:

```
[student@workstation ~]$ oc new-app \
-f ~/coolstore/ocp/turbine-template.yml
--> Deploying template "monitor/turbine" ...

--> Creating resources ...
imagestream "turbine-server" created
deploymentconfig "turbine-server" created
route "turbine-server" created
service "turbine-server" created
--> Success
Access your application via route 'turbine.apps.lab.example.com'
Run 'oc status' to view your app.
```

10.8. Wait until the Turbine pod is ready and running. Run the `oc get pods` command and wait until the following output is displayed.

```
[student@workstation coolstore-gateway-jaxrs]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
turbine-server-1-voa5t   1/1     Running   0          1m
...
```

10.9. Discover the hystrix metrics endpoints using the `http://turbine.apps.lab.example.com/discovery` resource URI.

```
[student@workstation ~]$ curl http://turbine.apps.lab.example.com/discovery
<h1>Hystrix Endpoints:</h1>
<h3>http://10.130.0.80:8080/hystrix.stream coolstore:true</h3>
```

Verify that the IP address from the above command matches the IP address of the API gateway pod:

```
[student@workstation ~]$ oc get pods -o wide \
-n api-gateway-service
NAME           ... IP           ...
coolstore-gateway-2-bbc9j ... 10.130.0.80 ...
```

The Turbine server is able to identify and communicate with the API Gateway microservice pod to gather Hystrix metrics.

- ▶ 11. Access the Hystrix metrics stream from the Turbine server using the `/turbine.stream?cluster=coolstore` endpoint:

```
[student@workstation ~]$ curl \
http://turbine.apps.lab.example.com/turbine.stream?cluster=coolstore
...
: ping
```

```
data: {"rollingCountFallbackSuccess":0,"rollingCountFallbackFailure":0,...  
...  
data: {"currentCorePoolSize":10,"currentLargestPoolSize":2,...  
...  
data: {"reportingHostsLast10Seconds":1,"name":"meta", ...  
...
```

Turbine server is able to aggregate circuit breaker metrics from the coolstore application.

Press **Ctrl+C** to stop the **curl** command.

► **12.** Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation coolstore-gateway-jaxrs]$ lab hystrix-deploy grade
```

► **13.** Commit your changes to your local Git repository:

```
[student@workstation coolstore-gateway-jaxrs]$ git commit -a -m \  
"Finished the Hystrix lab."
```

► **14.** Remove the **coolstore-gateway** project from the IDE workspace.

► **15. Optional:** Clean up to redo the exercise from scratch.

15.1. Perform this and the following steps only if you wish to start over this exercise.

Reset your local repository to the remote branch:

```
[student@workstation coolstore]$ git reset --hard \  
origin/do292-hystrix-lab-begin
```

15.2. Delete the **monitor** OpenShift project to remove the Turbine server.

```
[student@workstation ~]$ oc delete project monitor  
project "monitor" deleted
```

This concludes the guided exercise.

Summary

In this chapter, you learned:

- Microservices can fail due to multiple sources, such as network outages or external service unavailability. Netflix Hystrix provides an API for fault tolerance, supporting mechanisms to identify and monitor these problems.
- Netflix Hystrix implements the circuit breaker pattern that identifies and tracks service problems, and provide a fail-fast mechanism to avoid large amount of requests to overload a microservice.
- To implement circuit breaker with Hystrix, you must create subclasses of the **HystrixCommand** class, and configure in the class constructor the circuit breaker thresholds to identify unavailability and failures. The subclass must be used instead of the existing code that may break to introduce Hystrix circuit breaker capabilities.
- Hystrix outputs the circuit breaker status through a REST API endpoint.
- Hystrix Dashboard provides a web interface to monitor circuit breakers from a single microservice.
- Turbine Server aggregates information on multiple microservices circuit breakers in a single place and can be used with Hystrix Dashboard.
- The Kubeflix provides both Turbine and Hystrix Dashboard as container images that can be deployed on OpenShift.

Appendix A

Managing Git Branches

Managing Git Branches

This appendix reviews how to work with Git branches for executing the exercises in this course, and how to use the Gitweb web application to browse the repositories and branches in the classroom Git server.

Introducing Git

The source code for applications in this course is stored in a Git repository on the **services** VM. Each exercise has a starting point, a solution, and sometimes intermediate states stored in particular named Git branches.

During this course, you work with a local copy of the repository on the **workstation** VM, which is a clone of the remote repository, and includes all branches. You can commit changes to the local repository, but are not allowed to push changes back to the remote repository on the **services** VM. The remote repository from which a local repository is cloned is called the **origin**.

You can make changes to a working directory, which contains a copy of all the files from the current active branch. If you need to switch to another branch, you can commit the changes in the working directory to the active branch, save (**git stash**) the changes in the working directory, or discard the uncommitted changes in the working directory.

Useful Commands for Working with Git Branches

The following table lists the most useful tasks, and the Git commands to work with Git branches in the classroom. They operate only on the local repository, unless stated otherwise:

Command	Task
git branch	List the current branch in a Git repository.
git branch -a	List all branches in a Git repository.
git checkout mybranch1	Switch to a branch named mybranch1 .
git checkout -b mybranch2	Create a new branch named newbranch2 and switch to it.
git branch -d mybranch1	Delete the mybranch1 branch.
git diff	View differences between the current state of the working directory and the currently active branch.
git diff mybranch2	View differences between the current state of the working directory and a branch named mybranch2 .
git checkout -- .	Revert the working directory to the current state of a branch, discarding uncommitted changes in the working directory.

Command	Task
<code>git checkout -- filename</code>	Revert a specific file to the current state in a branch. This command is useful when you want to revert changes only to a specific file without affecting other files in the working directory.
<code>git reset --hard origin/branch_name</code>	Reset the contents of a local branch, discarding all changes (commits) made to the local repository. This command overrides the local repository and the working directory with contents from a remote repository (the <code>origin</code>).

Managing Changes in the Working Directory

Many of the hands-on exercises ask you to switch to a specific branch in a Git repository when starting an exercise. As you follow instructions and work through the exercise, you will make changes in your local working directory. Before switching branches, for example to move to another exercise, you may want to save your local changes without committing the changes.

The following table lists commands to manage changes in local working directory that have not been committed to the active branch:

Command	Task
<code>git stash</code>	Store (in Git parlance, this is called "stashing") your local changes temporarily before switching the working directory to another branch.
<code>git stash list</code>	List all the stashes. You can stash multiple changes in your local repository. Each stash is stored in a stack like data structure following <i>Last In First Out</i> (LIFO) semantics.
<code>git stash pop</code>	Apply (in Git parlance, this is called "popping") the latest stashed change (the topmost stash) to the working directory.
<code>git stash apply stash_index</code> <small>(where <code>stash_index</code> is a unique string that identifies a particular stash)</small>	Apply a specific stash to the working directory.

Using the Gitweb User Interface

The Gitweb application provides a web-based user interface to browse and view details of Git repositories and branches in the classroom Git server. You can use Gitweb to browse the source code tree of a named branch (for example, solution files for a lab) when performing the hands-on labs.

Gitweb is available at the URL:

<http://services.lab.example.com>

**Note**

The Gitweb application provides a read-only interface that does not allow the user to commit changes through it.

The home page of Gitweb lists all the Git repositories available in the classroom.

The screenshot shows a web browser window with the URL <git://services.lab.example.com/>. The page displays a table of repositories:

Project	Description	Owner	Last Change	Actions
coolstore	Unnamed repository; edit this...	Apache	9 days ago	summary shortlog log tree
hello-microservices	Unnamed repository; edit this...	Apache	9 days ago	summary shortlog log tree
php-helloworld	Unnamed repository; edit this...	Apache	6 days ago	summary shortlog log tree

Figure A.1: List of Git repositories

Click any of the repositories listed on the home page to bring up the summary page for the repository. The bottom half of the summary page shows the list of branches in the repository under the heading **heads**.

The screenshot shows a web browser window with the URL [git://services.lab.example.com/do292-hello-deploy](#). The page displays a table of branches under the **heads** section:

Date	Branch	Actions
9 days ago	do292-aloha-deploy-begin	shortlog log tree
10 days ago	master	shortlog log tree
10 days ago	do292-aloha-deploy-solution	shortlog log tree
11 days ago	lab-microservices-arquillian-solution	shortlog log tree
11 days ago	lab-microservices-arquillian	shortlog log tree
11 days ago	lab-fault-tolerance-annotation-solution	shortlog log tree
11 days ago	lab-fault-tolerance-annotation	shortlog log tree
11 days ago	demo-fault-tolerance-annotation	shortlog log tree
11 days ago	demo-configmap	shortlog log tree
11 days ago	wf-swarm-2018.3.3	shortlog log tree
2 weeks ago	do292-hello-deploy	shortlog log tree
2 weeks ago	lab-configmap	shortlog log tree
2 weeks ago	lab-configmap-solution	shortlog log tree
2 weeks ago	do292-hola-deploy	shortlog log tree
3 weeks ago	do292-smoke-test	shortlog log tree
4 weeks ago	do292-spring-demo-solution	shortlog log tree

Figure A.2: List of Git branches in a repository

Click any of the branches to show the commit log for the branch. From this page, you can click on **tree** to view the source code tree for the branch.

The screenshot shows a web-based Git interface for the 'hello-microservices' repository. At the top, there's a header with a URL and a 'commit' button. Below the header, there's a section titled 'hello-microservices'. The main content area displays a list of commits from 2018-02-16 to 9 days ago. Each commit entry includes the author, date, subject, and a green link labeled 'do292-aloha-deploy-'. The commits are:

- 9 days ago Ravi Srinivasan minor changes to Lab 1.7 begin branch do292-aloha-deploy-begin
- 10 days ago Ravi Srinivasan Begin branch for Lab 1.7
- 10 days ago Ravi Srinivasan Aloha app for Lab 1.7 do292-aloha-deploy-solution
- 2018-02-18 Ravi Srinivasan Branch for Lab 1.7 do292-aloha-deploy
- 2018-02-18 Ravi Srinivasan Branch for GE 1.6
- 2018-02-17 Ravi Srinivasan Bumped up FMP plugin version to fix resource validation bug
- 2018-02-16 Ravi Srinivasan Simple Hello service for Demo 1.5

Figure A.3: Git branch log

You can browse the source code tree and view the source code of individual files from the **tree** page for a branch.

The screenshot shows a 'tree' page for the 'do292-aloha-deploy-solution' branch. The title is 'minor changes to Lab 1.7 begin branch'. The tree structure is as follows:

```

- rw-r--r--    68 .gitignore blob | history | raw
- rw-r--r--  11357 LICENSE blob | history | raw
drwxr-xr-x    - aloha tree | history
- rw-r--r--   2724 pom.xml blob | history | raw

```

Figure A.4: Git branch tree

You can browse the source code tree of a named branch using a shortcut URL, which is of the form:

`http://services.lab.example.com/repo_name/tree/branch_name`

For example, to browse the source code tree for a branch named **do292-aloha-deploy-solution** in the **hello-microservices** Git repository, navigate to:

`http://services.lab.example.com/hello-microservices/tree/do292-aloha-deploy-solution`



References

Git branch man page

<https://git-scm.com/docs/git-branch>

What is a Git branch?

<https://git-scm.com/book/en/v1/Git-Branching-What-a-Branch-Is>

Git Tools - Stashing

<https://git-scm.com/book/en/v1/Git-Tools-Stashing>

Gitweb

<https://git-scm.com/docs/gitweb>

Appendix B

Working With Red Hat JBoss Developer Studio

Working with Red Hat JBoss Developer Studio

This appendix reviews how to perform common tasks using the Red Hat JBoss Studio integrated development environment (IDE).

Introducing Red Hat JBoss Developer Studio

Red Hat JBoss Developer Studio, or simply *JBoss Developer Studio*, is an integrated development environment (IDE) supporting multiple programming languages, and it is targeted to deploy on platforms such as Red Hat JBoss Enterprise Application Platform (JBoss EAP), Red Hat Fuse, Node.js, and Red Hat OpenShift Container Platform.

You can install and run JBoss Developer Studio on Linux, Windows, and MacOS operating systems. It requires that a Java development kit (JDK) is installed.

Besides the Red Hat cloud and middleware product families, JBoss Developer Studio integrates with popular tools used by developers, such as Git, Maven, and Arquillian.

JBoss Developer Studio is based on the Eclipse platform and can be extended with plug-ins from the large Eclipse IDE community.

Red Hat provides no-cost access to JBoss Developer Studio and many other development tools such as the Red Hat Container Development Kit (CDK) through a free developer's subscription offered through the *Red Hat Developer* program and web site.

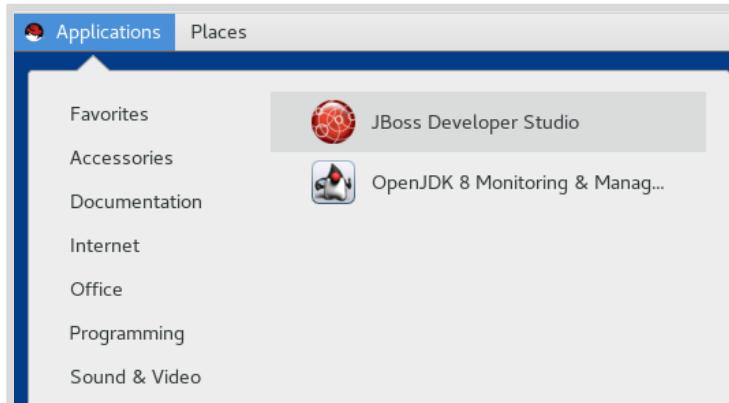
Performing Common Tasks Using JBoss Developer Studio

These are instructions for performing common tasks using JBoss Developer Studio.

Starting Red Hat JBoss Developer Studio

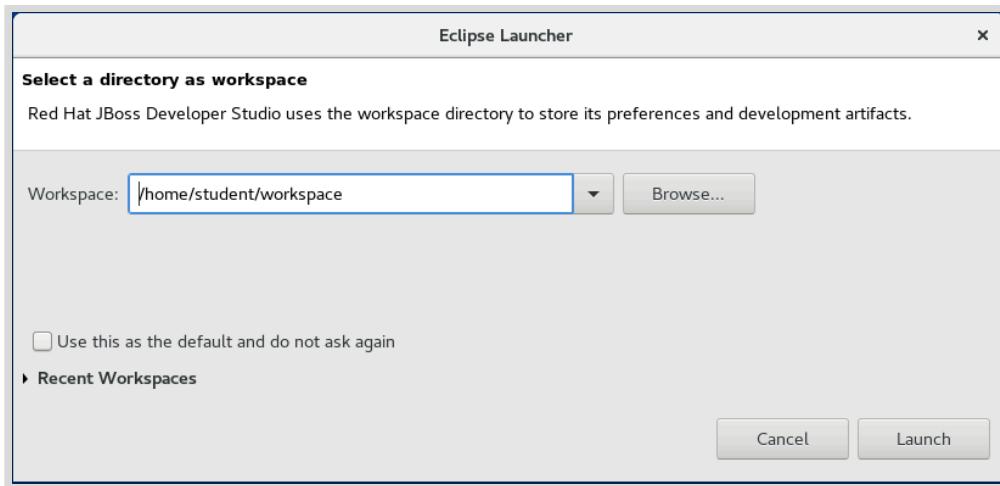
The following instructions apply to the JBoss Developer Studio installation on **workstation** virtual machine, using the default graphical desktop from Red Hat Enterprise Linux 7.

1. On the Gnome desktop, click **Applications** → **Programming** → **JBoss Developer Studio**.



2. If Gnome displays a warning that JBoss Developer Studio is not responding, click **Wait**.

3. Accept the default workspace folder **/home/student/workspace** and click **Launch**.



4. The JBoss Developer Studio IDE's workspace shows a number of views at the left, right, and bottom areas of the screen. The center area is reserved for source code editors.

Using Keyboard Shortcuts

JBoss Developer Studio provides a number of keyboard shortcuts for command development tasks. The following list presents some of the most used:

Ctrl+S

Saves the file opened in the current editor tab.

Ctrl+Space

Triggers context-sensitive code auto-completion.

Shift+Ctrl+C

Comments and uncomments selected lines.

Shift+Ctrl+F

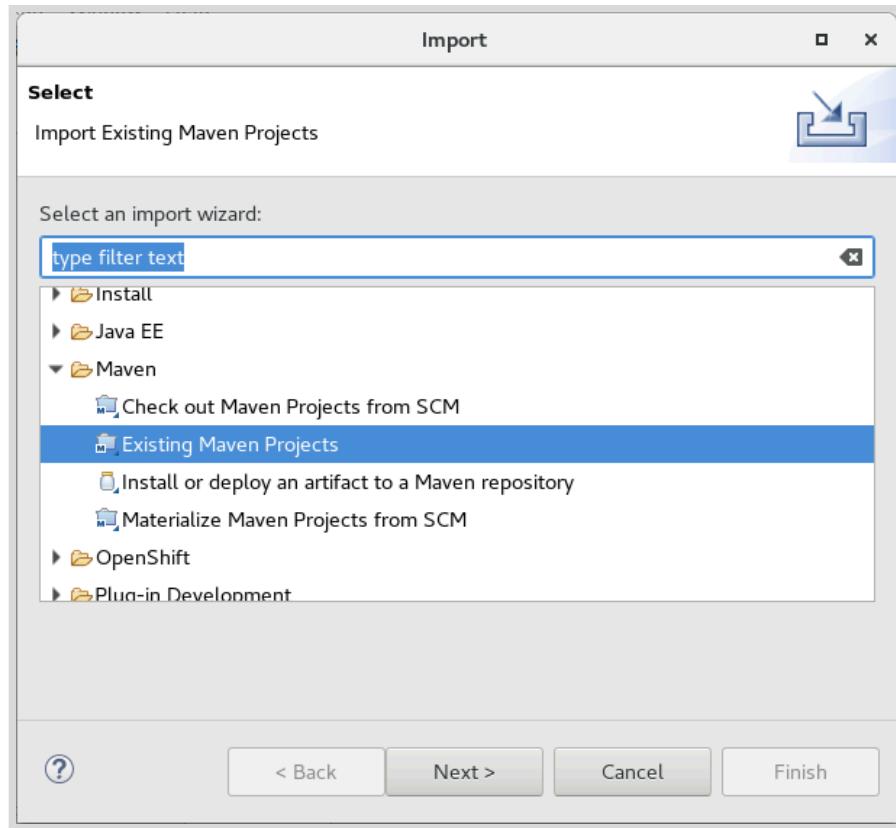
Formats and indents selected lines.

Importing a Maven Project

JBoss Developer Studio is able to import a Maven project, which is a folder containing a **pom.xml** file, and creates an Eclipse Java project.

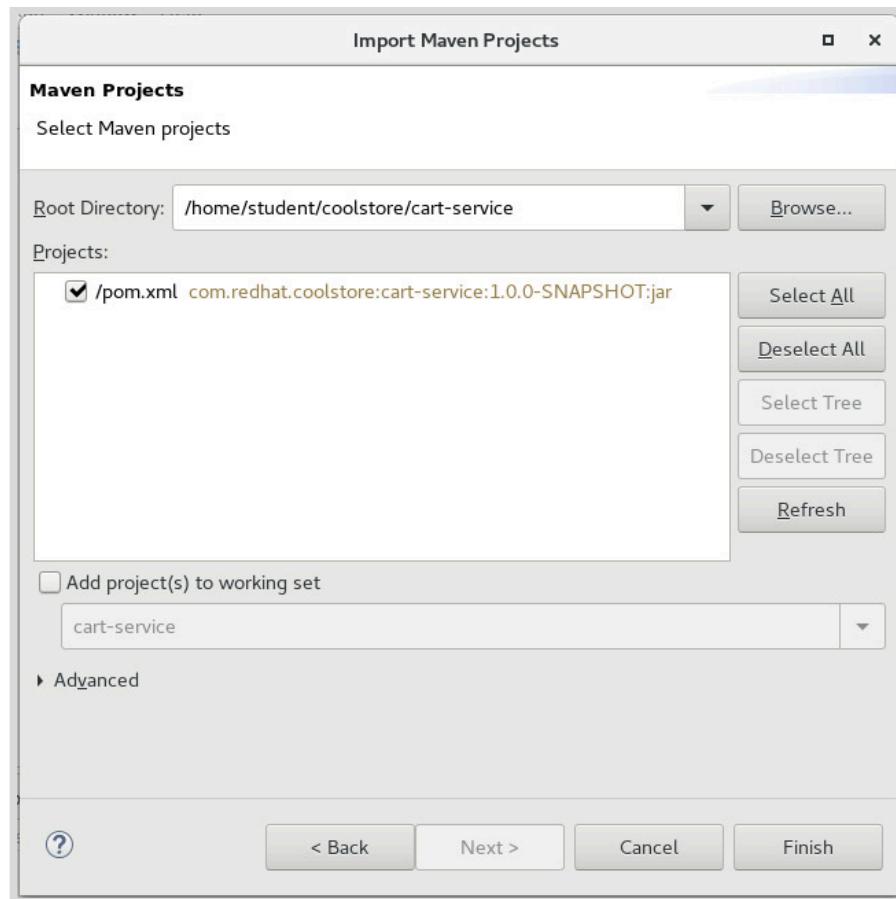
Because the IDE uses its internal compiler to offer source code auto-completion and report errors, it is able to perform tasks that Maven would not, such as running unit tests for a project with compilation errors.

1. In the JBoss Developer Studio top-level menu bar, click **File** → **Import**. Expand the **Maven** folder and select **Existing Maven Projects**. Click **Next**.



2. Click **Browse** and navigate to the Maven project folder. The IDE automatically selects the **pom.xml** file. Click **Finish**.

Be sure you entered a folder with the child Maven project (or module), not a folder with a parent POM file.



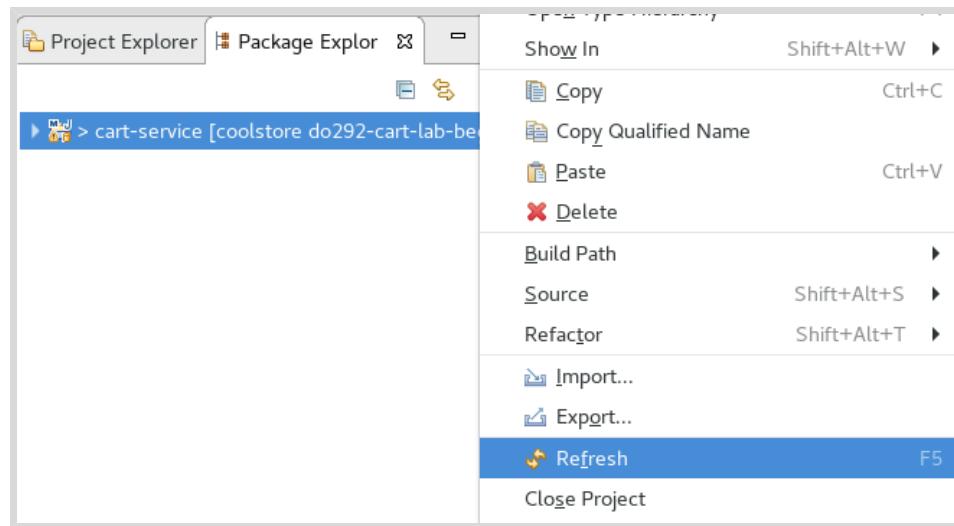
3. The IDE builds the project and, after a few moments, may display error and warning markers. Wait for the build to finish before opening project files.
4. In general, when working with JAVA based projects, prefer to use the **Package Explorer** view in the left sidebar for easy navigation between packages in the project.

Switching to a Git Branch

JBoss Developer Studio requires that you refresh a project after you make changes to any file inside the project folder outside of the IDE, for example by using **git** commands.

Because the IDE uses its internal compiler to offer source code auto-completion and to report errors, its data about the project logical structure may become stale and needs to be refreshed.

1. In the JBoss Developer Studio **Package Explorer** view, right-click the project name and click **Refresh**, or press **F5**.



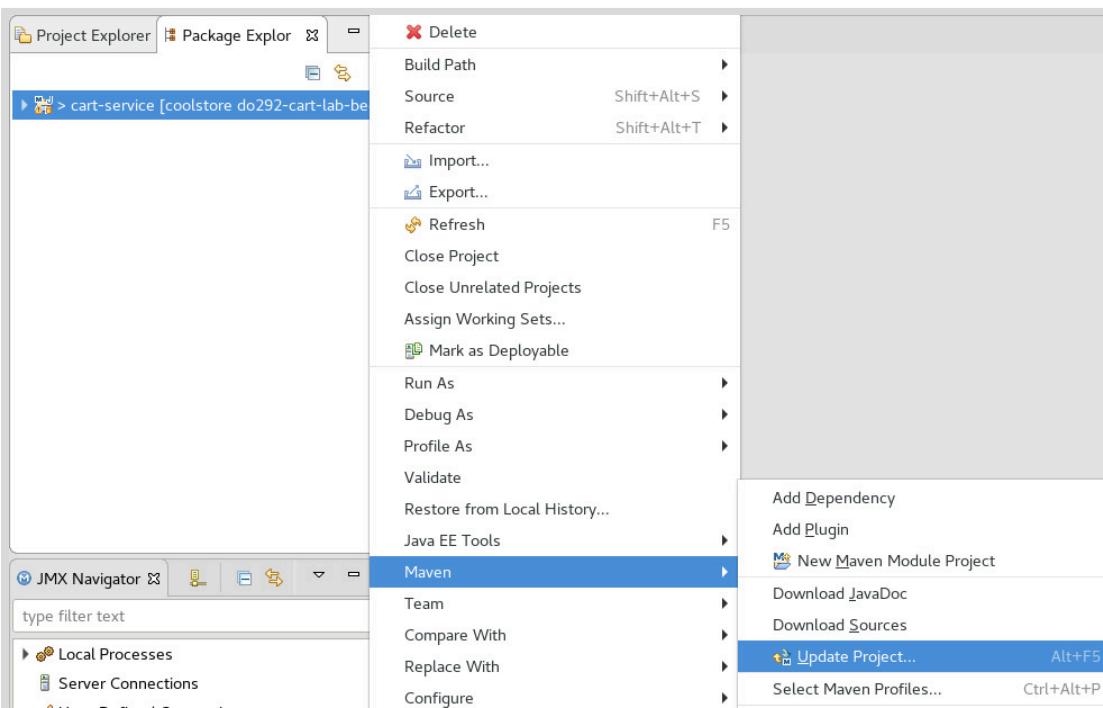
2. If the new branch has different Maven POM settings, you also need to update the project configuration.

Updating a Project Configuration

JBoss Developer Studio requires that you update the IDE project settings after you make changes to project's Maven POM file, for example: after switching to a new branch.

Because the IDE uses its internal compiler to offer source code auto-completion and to report errors, its project settings may become stale after you make changes to project's Maven POM file, including when you make the changes using the IDE's **Maven POM Editor**.

1. In the JBoss Developer Studio **Package Explorer** view, right-click the project name and select **Maven → Update Project**, or press **Ctrl+F5**.

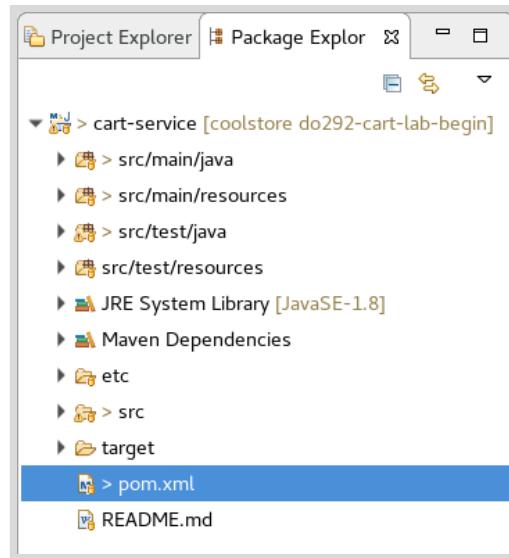


- The IDE rebuilds the project and, after a few moments, may display error and warning markers. Wait for the build to finish before opening project files.

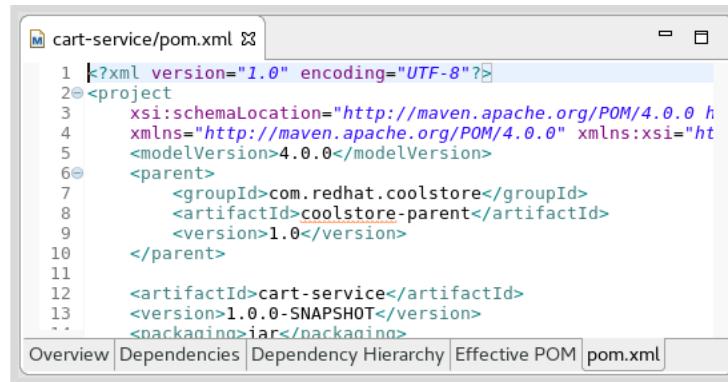
Opening a Maven Project POM file

JBoss Developer Studio offers a graphical editor for Maven POM files. In this course, the instructions assume you edit the raw XML source.

- In the JBoss Developer Studio **Package Explorer** view, expand the project, and open the **pom.xml** file.



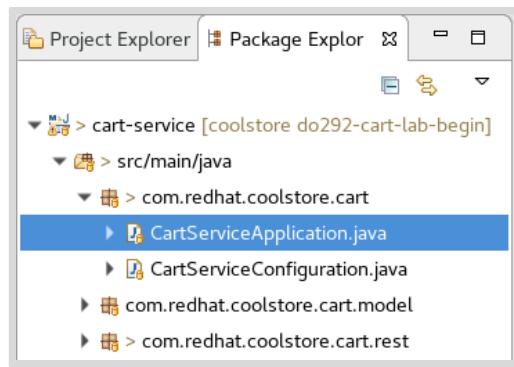
- The IDE opens a **Maven POM Editor**. Switch to the **pom.xml** tab to view or edit the POM file XML source code. When done, press **Ctrl+S** to save your changes.



Opening a Java Class or Interface

When you open or edit a Java class, you actually open the JAVA file that contains the class definition. The standard Maven project structure keeps application source files separate from test source code.

- In the JBoss Developer Studio **Package Explorer** view, expand the source folder, usually **/src/main/java**, and the class' package. Open the JAVA file with the name of the class.

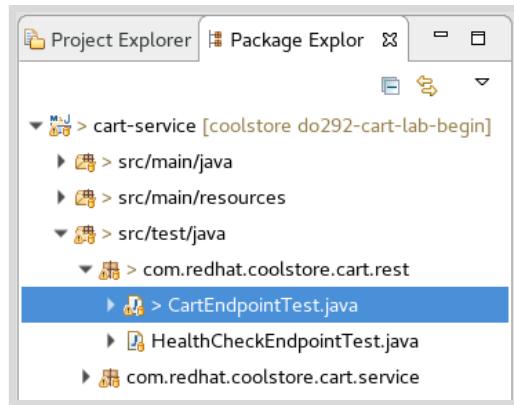


2. The IDE opens a **Java Editor**. When done, press **Ctrl+S** to save your changes.

Opening a JUnit Test Case

A JUnit test case is a Java class. The standard Maven project structure keeps application source files separate from test source code.

1. In the **Package Explorer** view, expand the test source folder, usually the **/src/test/java** folder, and the test case's package. Open the JAVA file with the name of the test case.

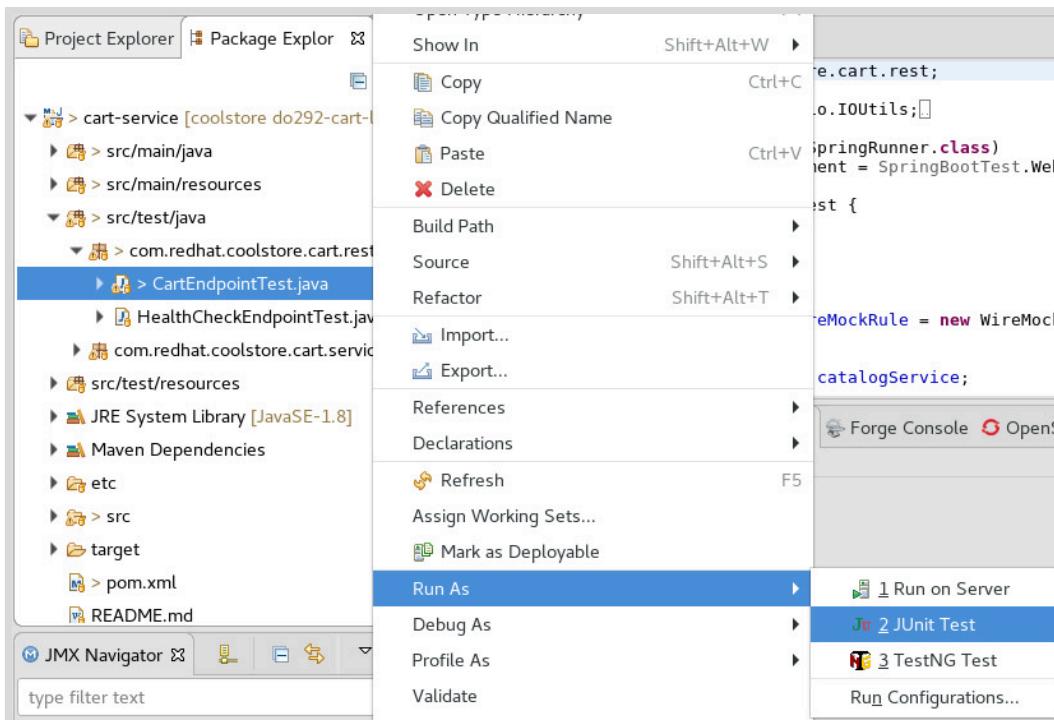


2. The IDE opens a **Java Editor**. When done, press **Ctrl+S** to save your changes.

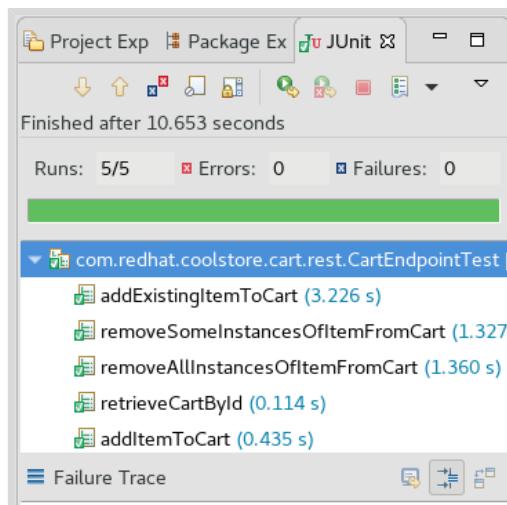
Running a JUnit Test Case

JBoss Developer Studio is able to run JUnit test cases, even when there are errors in the project, and provides a nice graphical display of the test results.

1. In the **Package Explorer** view, right-click the **BonjourResourceTest.java** file and select **Run As → JUnit Test**.



2. If you see the **Errors in Workspace** dialog, click **Proceed**. Some exercises ask you to run unit tests for projects that are still incomplete, and contains errors that do not affect the tests.
3. The IDE opens the **JUnit** view with the test case progress and its results.

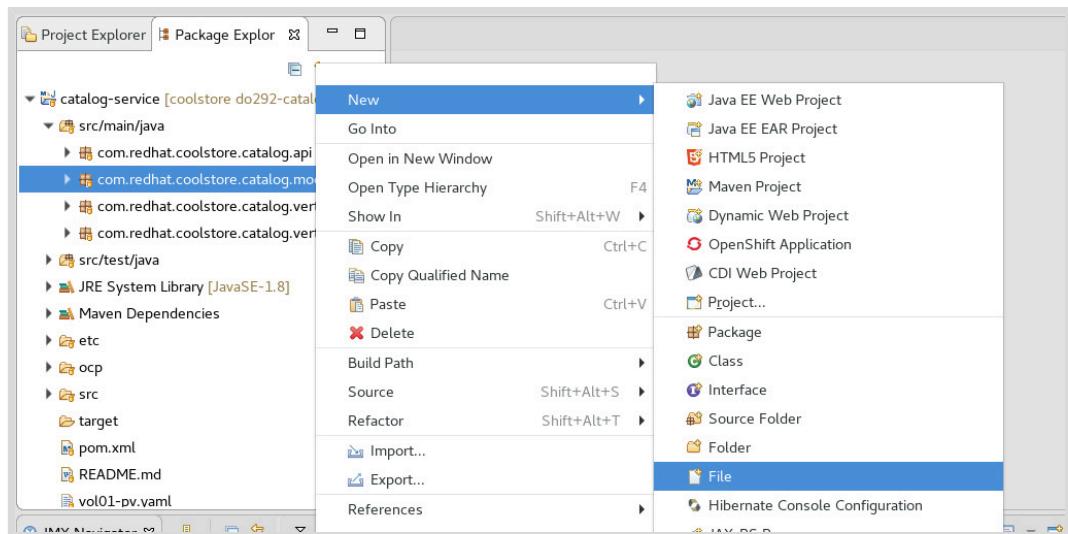


You can also right-click a package name to run all test cases inside that package, and right-click a source folder to run all test cases inside that folder.

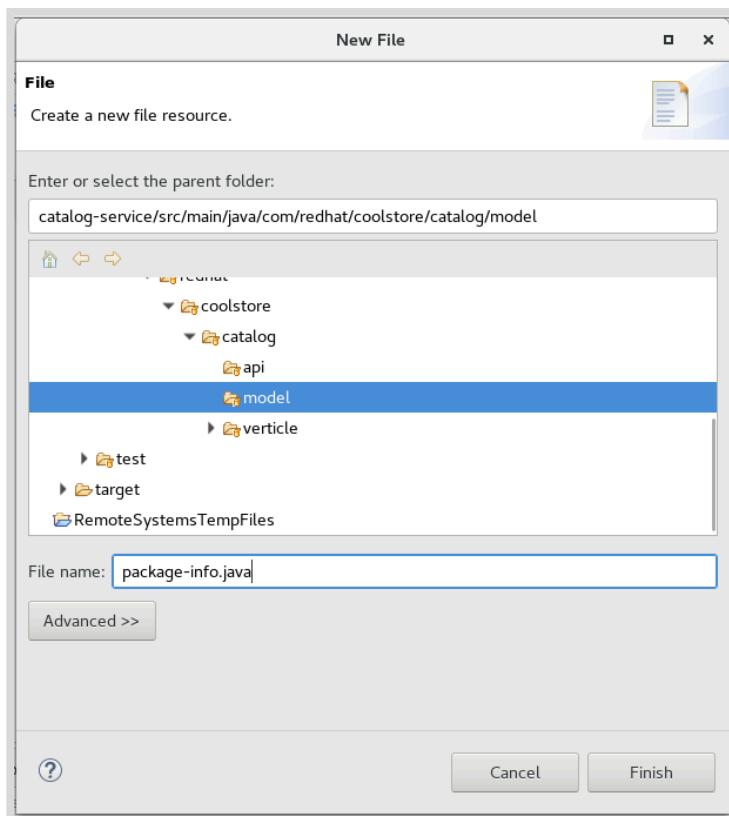
Creating a package-info.java File

The **package-info.java** file provides annotations and JavaDoc comments for a Java package.

1. In **Package Explorer** view, expand the **/src/main/java** source folder, right-click the package name, and click **New → File**.



2. Type **package-info.java** in the **File name** field. Click **Finish**.

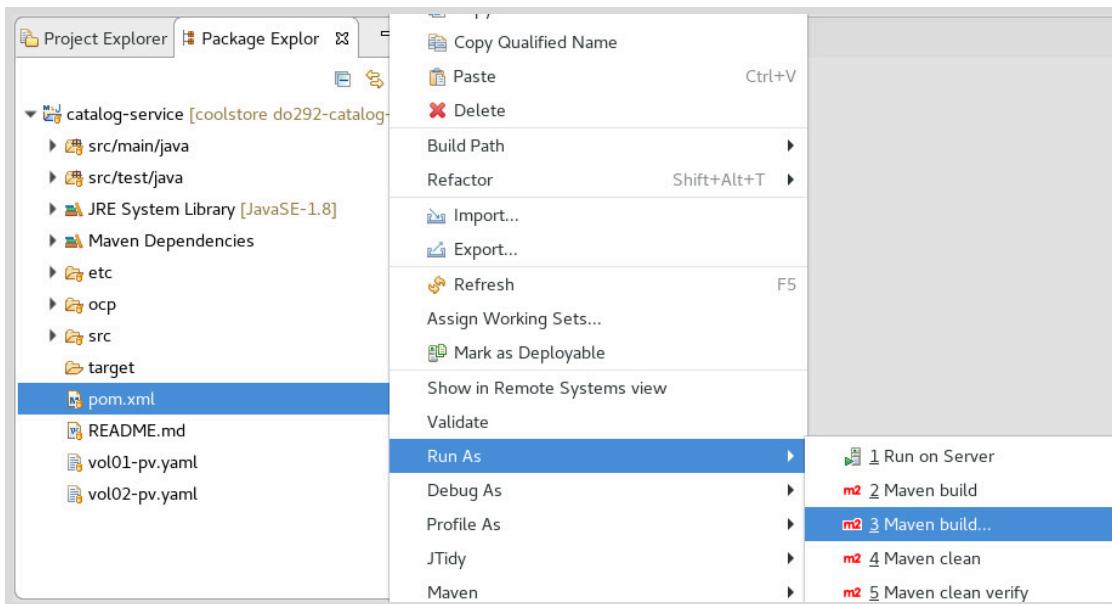


3. The IDE opens a **Java Editor**. When done, press **Ctrl+S** to save your changes.

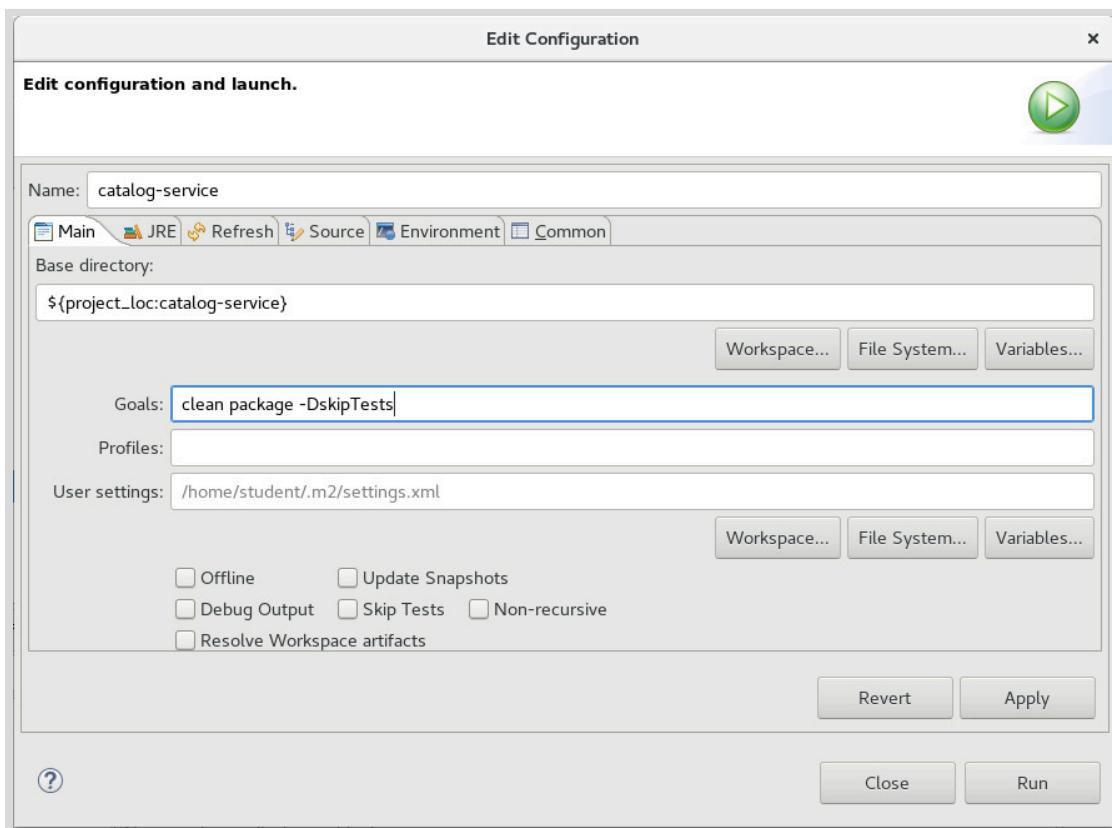
Running a Maven Goal

If you do not want to switch to a command shell and them back to the JBoss Developer Studio, you can run any Maven goal directly from the IDE.

- In the **Package Explorer** view, right-click the **pom.xml** file. Click the second **Run As → Maven build** entry (the entry that ends with ellipsis) to open the **Edit Configuration** dialog.



- Type the **mvn** command arguments in the **Goals** field and click **Run** to start the Maven build.



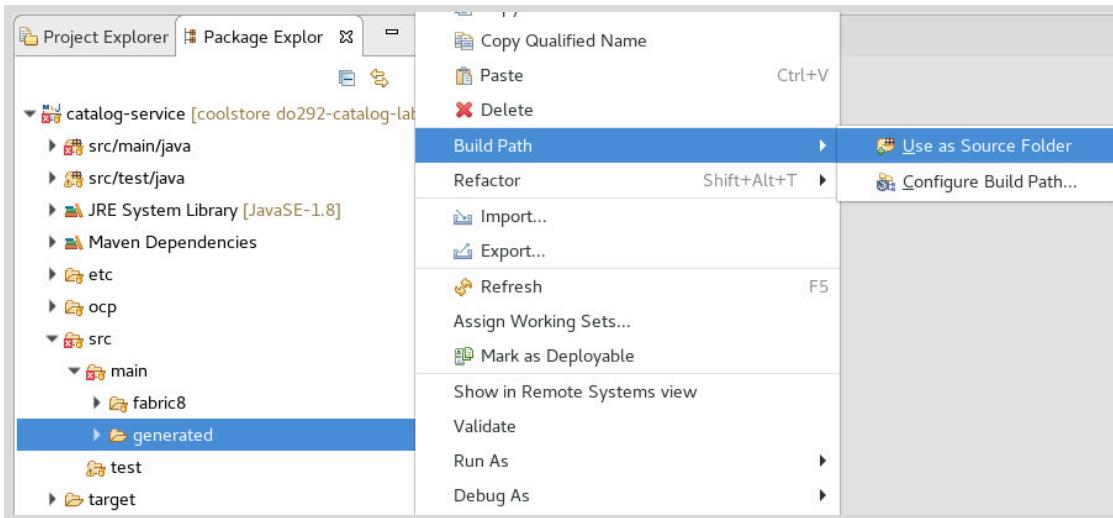
- The **Console** view shows the logs of the Maven build.

Adding a Source Folder to the Build Path

The standard Maven project structure can contain any number of source folders under **/src/main** or **/test/main**, but JBoss Developer Studio only recognizes **java** and **resources** for internal IDE builds, by default.

To add additional source folders, for example because your project requires a Maven plug-in that performs static source code generation follow these steps:

1. Scroll down the **Package Explorer** view until you find the **src** folder. Expand it and the child **main** folder. Right-click the new source folder and click **Build Path** → **Use as Source Folder**.

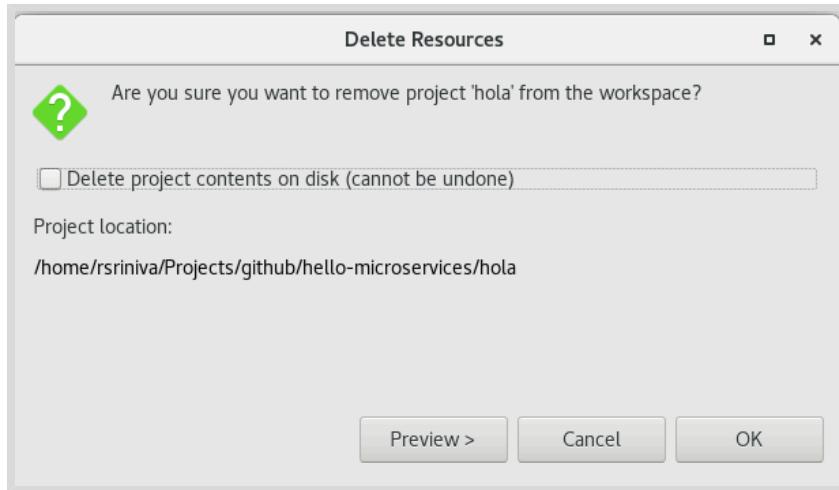


2. The IDE rebuilds the project to include classes and resources in the new folder.

Removing a Project from the IDE Workspace

To remove a project from the IDE, without deleting the actual files in the project directory, follow these steps:

1. Right-click the project in the **Package Explorer** view, and click **Delete**. You should see the following dialog:



2. Do **NOT** select the **Delete project contents on disk** option.

Click **OK** to remove the project from the IDE workspace.



References

Red Hat JBoss Developer Studio in the Red Hat Developer web site

<https://developers.redhat.com/products/devstudio/overview/>

Further information about the JBoss Developer Studio IDE is available in the *Red Hat JBoss Developer Studio 11.2 Product Documentation* at

https://access.redhat.com/documentation/en-us/red_hat_jboss_developer_studio/11.2/

