





Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



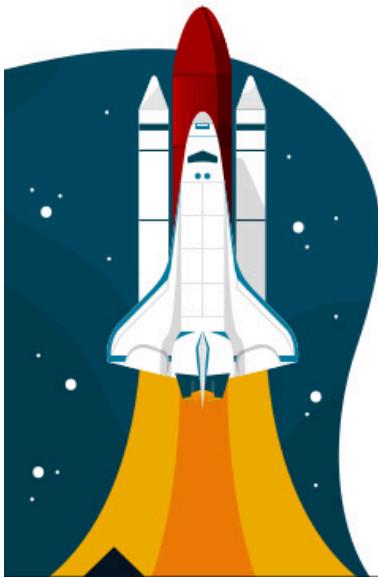
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Red Hat Cloud-native Microservices Development with Quarkus



Red Hat Build of Quarkus 1.11 DO378
Red Hat Cloud-native Microservices Development with
Quarkus
Edition 3 20220412
Publication date 20220412

Authors: Eduardo Ramirez Ronco, Guy Bianco IV, Jordi Sola Alaball,
Manuel Aude Morales, Randy Thomas, Ravishankar Srinivasan
Editor: Sam Ffrench

Copyright © 2021 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2021 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, OpenShift, Fedora, Hibernate, Ansible, CloudForms,
RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries
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 American, Inc. and/or its affiliates.

XFS® is a registered trademark of Hewlett Packard Enterprise Development LP or its subsidiaries in the United
States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is a trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open
source or commercial project.

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: David Saco, Heider Souza, Joel Birchler, Richard Allred, Sajith Eyamkuzhy

Document Conventions	ix
	ix
Introduction	xi
Red Hat Cloud-Native Microservices Development with Quarkus	xi
Orientation to the Classroom Environment	xii
1. Describing a Microservice Architecture	1
Describing Microservices	2
Quiz: Describing Microservices	8
Describing Microservice Architecture Patterns	12
Quiz: Describing Microservice Architecture Patterns	19
Guided Exercise: Verifying OpenShift Credentials	21
Summary	24
2. Implementing a Microservice with Quarkus	25
Describing Quarkus as a Microservice Runtime	26
Quiz: Describing Quarkus as a Microservice Runtime	29
Implementing a JSON Microservice with CDI and JAX-RS	31
Guided Exercise: Implementing a RESTful Microservice	42
Persisting Data with Panache	48
Guided Exercise: Persisting Data with Panache	53
Lab: Implementing a Microservice with Quarkus	59
Summary	71
3. Deploying Microservice-based Applications	73
Describing a Microservice Application	74
Guided Exercise: Describing a Microservice Application	79
Deploying Microservices in Red Hat OpenShift Container Platform	84
Guided Exercise: Deploying Microservices in Red Hat OpenShift Container Platform	95
Deploying Quarkus Applications in Red Hat OpenShift Container Platform	100
Guided Exercise: Deploying Quarkus Applications in Red Hat OpenShift Container Platform	105
Lab: Deploying Microservice-based Applications	109
Summary	116
4. Building Microservice Applications with Quarkus	117
Injecting Configuration Data	118
Guided Exercise: Injecting Configuration Data	126
Implementing Feature Toggles	134
Guided Exercise: Implementing Feature Toggles	137
Connecting Multiple Services	140
Guided Exercise: Connecting Multiple Services	144
Lab: Building Microservice Applications with Quarkus	158
Summary	166
5. Implementing Fault Tolerance	167
Applying Fault Tolerance Policies to a Microservice	168
Guided Exercise: Applying Fault Tolerance Policies	176
Implementing a Health Check Monitored by OpenShift	185
Guided Exercise: Implementing a Health Check	189
Lab: Implementing Fault Tolerance	195
Summary	210
6. Building and Deploying Native Quarkus Applications	211
Building Native Applications with Quarkus	212
Guided Exercise: Building Native Applications with Quarkus	217
Containerizing Quarkus Native Applications	220

Guided Exercise: Containerizing Quarkus Native Applications	223
Summary	228
7. Testing Microservices	229
Testing Quarkus Microservices	230
Guided Exercise: Testing Quarkus Microservices	237
Testing Microservices with Mock Frameworks	243
Guided Exercise: Testing Microservices with Mock Frameworks	249
Lab: Testing Microservices	254
Summary	262
8. Securing Microservices Communication	263
Implementing Communication Security in Quarkus Microservices	264
Guided Exercise: Implementing Communication Security in Quarkus Microservices	271
Authenticating and Authorizing Requests	280
Guided Exercise: Authenticating and Authorizing Requests	288
Lab: Securing Microservices	298
Summary	311
9. Monitoring Quarkus Microservices	313
Adding Metrics to a Microservice	314
Guided Exercise: Adding Metrics to a Microservice	319
Tracing Requests Across Microservices	326
Guided Exercise: Tracing Requests Across Microservices	333
Lab: Monitoring Microservices	339
Summary	350
10. Comprehensive Review of Red Hat Cloud-native Microservices Development with Quarkus	351
Comprehensive Review	352
Lab: Develop and Deploy Microservices on OpenShift	354

Document Conventions

This section describes various conventions and practices used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:



References

These describe where to find external documentation relevant to a subject.



Note

These 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 something that makes your life easier.



Important

These provide details of information that is easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring these admonitions will not cause data loss, but may cause irritation and frustration.



Warning

These should not be ignored. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Red Hat Cloud-Native Microservices Development with Quarkus

Red Hat Cloud-Native Microservices Development with Quarkus (DO378) emphasizes learning architectural principles and implementing microservices based on Quarkus and OpenShift. This course is based on OpenShift 4.6 and Quarkus 1.11. You will build on Java EE application development fundamentals and focus on how to develop, monitor, test, and deploy modern microservices applications. Many enterprises are looking for a way to take advantage of cloud-native architectures, but many do not know the best approach. Quarkus is an exciting new technology that brings the reliability, familiarity, and maturity of Java Enterprise with a container-ready lightning fast deployment time.

Course Objectives

- Build Quarkus-based microservice applications deployed on Red Hat OpenShift Container Platform.
- This course prepares the student to take the *Red Hat Certified Enterprise Microservices Developer* exam (EX378).

Audience

- This course is designed for application developers who already have a basic understanding of web application development.

Prerequisites

- The course *Red Hat Application Development I: Programming in Java EE* (AD183) or equivalent experience with Java EE development is required.
- Proficiency in using an IDE such as Red Hat® Developer Studio or VSCode
- Experience with Maven and version control is recommended, but not required.
- The course *Introduction to OpenShift Applications* (DO101) is recommended, but not required.

Orientation to the Classroom Environment

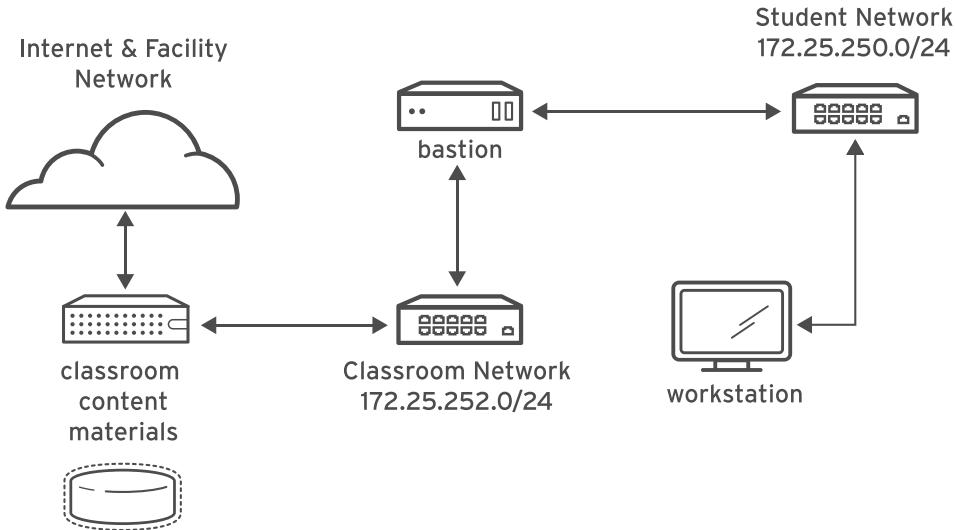


Figure 0.1: Classroom environment

In this course, the main computer system used for hands-on learning activities is **workstation**. No other machines are used by students for these activities.

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
content.example.com, materials.example.com, classroom.example.com	172.25.252.254, 172.25.253.254, 172.25.254.254	Classroom utility server
workstation.lab.example.com	172.25.250.9	Graphical workstation used for system administration
bastion.lab.example.com	172.25.250.254	Router linking student's VMs to classroom servers

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

Several systems in the classroom 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 is provided in the instructions for those activities.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning classroom. They are accessed through a web application hosted at rol.redhat.com [<http://rol.redhat.com>]. You should log in to this site using your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through a web page. The state of each virtual machine in the classroom is displayed on the page under the **Online Lab** tab.

Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots into the same state as when it was shut down (the disk will have been preserved).
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

Depending on the state of a machine, a selection of the following actions is available.

Classroom/Machine Actions

Button or Action	Description
PROVISION LAB	Create the ROL classroom. Creates all of the virtual machines needed for the classroom and starts them. Can take several minutes to complete.
DELETE LAB	Delete the ROL classroom. Destroys all virtual machines in the classroom. Caution: Any work generated on the disks is lost.
START LAB	Start all virtual machines in the classroom.
SHUTDOWN LAB	Stop all virtual machines in the classroom.
OPEN CONSOLE	Open a new tab in the browser and connect to the console of the virtual machine. You can log in directly to the virtual machine and run commands. In most cases, you should log in to the workstation virtual machine and use ssh to connect to the other virtual machines.
ACTION > Start	Start (power on) the virtual machine.

Button or Action	Description
ACTION > Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION > Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. Caution: Any work generated on the disk is lost.

At the start of an exercise, if instructed to reset a single virtual machine node, click ACTION > Reset for only the specific virtual machine.

At the start of an exercise, if instructed to reset all virtual machines, click ACTION > Reset

If you want to return the classroom environment to its original state at the start of the course, you can click DELETE LAB to remove the entire classroom environment. After the lab has been deleted, you can click PROVISION LAB to provision a new set of classroom systems.



Warning

The DELETE LAB operation cannot be undone. Any work you have completed in the classroom environment up to that point will be lost.

The Autostop Timer

The Red Hat Online Learning enrollment entitles you to a certain amount of computer time. To help conserve allotted computer time, the ROL classroom has an associated countdown timer, which shuts down the classroom environment when the timer expires.

To adjust the timer, click MODIFY to display the New Autostop Time dialog box. Set the number of hours until the classroom should automatically stop. Note that there is a maximum time of ten hours. Click ADJUST TIME to apply this change to the timer settings.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning classroom. They are accessed through a web application hosted at rol.redhat.com [<http://rol.redhat.com>]. You should log in to this site using your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through a web page. The state of each virtual machine in the classroom is displayed on the page under the **Online Lab** tab.

Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.

Virtual Machine State	Description
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots into the same state as when it was shut down (the disk will have been preserved).
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

Depending on the state of a machine, a selection of the following actions is available.

Classroom/Machine Actions

Button or Action	Description
PROVISION LAB	Create the ROL classroom. Creates all of the virtual machines needed for the classroom and starts them. Can take several minutes to complete.
DELETE LAB	Delete the ROL classroom. Destroys all virtual machines in the classroom. Caution: Any work generated on the disks is lost.
START LAB	Start all virtual machines in the classroom.
SHUTDOWN LAB	Stop all virtual machines in the classroom.
OPEN CONSOLE	Open a new tab in the browser and connect to the console of the virtual machine. You can log in directly to the virtual machine and run commands. In most cases, you should log in to the workstation virtual machine and use ssh to connect to the other virtual machines.
ACTION > Start	Start (power on) the virtual machine.
ACTION > Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION > Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. Caution: Any work generated on the disk is lost.

At the start of an exercise, if instructed to reset a single virtual machine node, click ACTION > Reset for only the specific virtual machine.

At the start of an exercise, if instructed to reset all virtual machines, click ACTION > Reset

If you want to return the classroom environment to its original state at the start of the course, you can click **DELETE LAB** to remove the entire classroom environment. After the lab has been deleted, you can click **PROVISION LAB** to provision a new set of classroom systems.



Warning

The **DELETE LAB** operation cannot be undone. Any work you have completed in the classroom environment up to that point will be lost.

The Autostop Timer

The Red Hat Online Learning enrollment entitles you to a certain amount of computer time. To help conserve allotted computer time, the ROL classroom has an associated countdown timer, which shuts down the classroom environment when the timer expires.

To adjust the timer, click **MODIFY** to display the **New Autostop Time** dialog box. Set the number of hours until the classroom should automatically stop. Note that there is a maximum time of ten hours. Click **ADJUST TIME** to apply this change to the timer settings.

Chapter 1

Describing a Microservice Architecture

Goal

Describe components and patterns of microservice-based application architectures.

Objectives

- Define what microservices are and the guiding principles for their creation.
- Describe the major patterns implemented in microservice architectures.

Sections

- Describing Microservices (and Quiz)
- Describing Microservice Architecture Patterns (and Quiz)
- Verifying OpenShift Credentials (Guided Exercise)

Describing Microservices

Objectives

After completing this section, you should be able to define what microservices are and the guiding principles for their creation.

Describing the Microservice Architecture

Microservice architecture is a method of dividing domains of business logic into self-contained processes, which communicate with other processes through the network. Each service revolves around a specific business domain, such as customer information or inventory pricing. You can create, manage, and deploy each microservice with different tools and programming languages. For example, a developer can build a microservice for a specific business domain and then develop a web application which calls that microservice.

Microservices can be independently managed and deployed using automation. They run in unique processes and communicate using a lightweight mechanism, usually HTTP calls to a REST API. This modularity means that microservices are suitable for running within a cloud environment, and fit naturally into containerized deployment platforms such as Kubernetes and the OpenShift Container Platform.

Additionally, a microservice generally has the following characteristics.

- Modeled around a single business problem or domain.
- Implements its own business logic, persistent storage, and external collaboration.
- Has an individually published contract, also known as an API.
- Capable of running in isolation.
- Independent and loosely-coupled from other services.
- Easily replaced or upgraded.
- Scaled and deployed independently of other services.

Analyzing Technology Trends and Reasons for Adoption

Many organizations want to adopt microservice architecture in their new developments and re-architect their existing applications using this approach.

The most popular reasons for adopting microservices include.

Faster deployment

Microservices are much smaller than traditional monolithic applications. Smaller services improve the time required to fix bugs because these services are released independently, meaning new features can be added, tested, and released quickly.

Rapid development

Microservices are developed and maintained by small teams. Small teams are typically focused on one or two microservices at most.

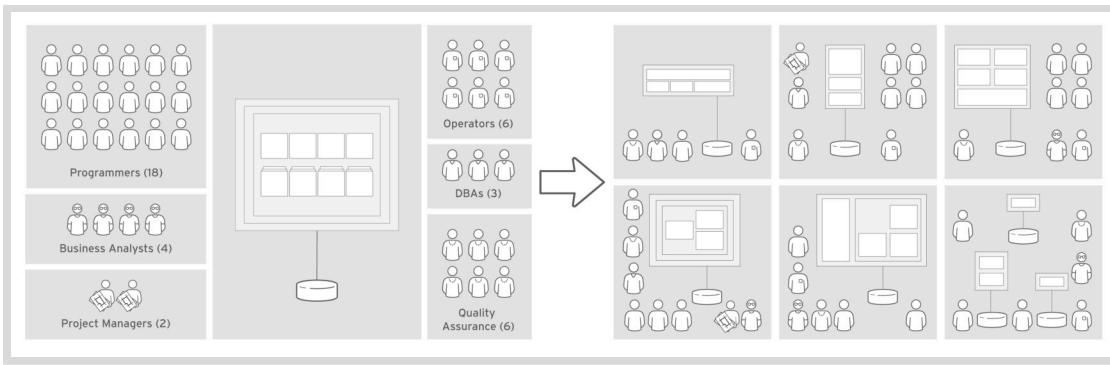


Figure 1.1: Rapid development

Collaboration becomes a bottleneck in large teams where team size is ten or more members. To encourage small teams in the development, management, and operation of microservices, some organizations use the "two-pizza team" concept. The two-pizza idea stipulates that if you cannot feed all the team members with two pizzas, then the team is too large. Generally, a reasonable team size is 5–7 members.

Less complex code

Microservices are much smaller than monolithic applications due to their design and architecture. Each microservice acts on a specific business operation and functions as a separate entity. This results in less complex code than larger applications. Microservices can also be tested individually and have its own release schedule.

Easier to scale

DevOps teams usually deploy microservices independently. Individual services can scale out horizontally depending on the amount of load the service receives, meaning that services receiving higher traffic get resources allocated.

Comparing Microservices and Monolithic Applications

Monolithic applications are large enterprise applications developed and deployed as a single project. A single team of developers write, test, deploy and release these applications as a unique artifact using only one programming language. They are complex to build, test, and migrate into production. Even though developers usually divide these applications into tiers and layers, they almost always package them into a single WAR or EAR file.

Hardware limits the amount of memory and other resources available to monolithic applications deployed as a single app. Monolithic applications must scale by replicating the entire application on multiple servers. Additionally, these applications tend to be more complex and tightly coupled, making them more challenging to maintain and update.

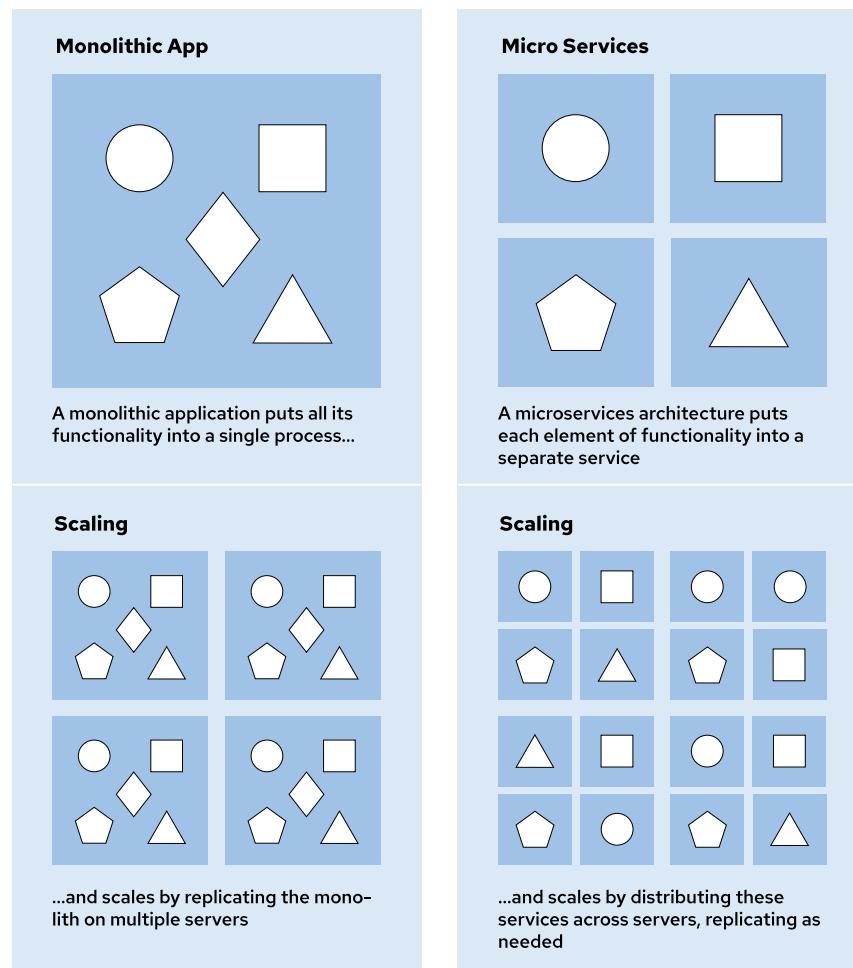


Figure 1.2: Comparing monolithic applications and microservices

Compared to monolithic applications, microservices are better organized, smaller, more loosely-coupled, and developed, tested, and deployed independently. Because microservices can be released individually, the time required to fix bugs or add new features is much shorter and changes can be deployed to production more efficiently. Additionally, because microservices are small and stateless, they can scale much easier.

Reviewing the Principles for Successful Adoption

As more companies attempt to move towards using a microservice architecture, a few essential principles have become critical to the success of microservice-based development.

These principles include.

Reorganizing to DevOps

Microservices teams are responsible for the entire lifecycle of their microservice, from development to operation. DevOps teams can work on their product at their own pace. The principle of "you build it, you own it" provides autonomy and speed for delivery teams.

Packaging the service as a container

A container is a set of isolated processes, which provides all of the necessary dependencies to run an application. By using tools such as Docker, the deployment platform for the microservice can be packaged and deployed as a container. Using containers allows developers to have control and confidence over the exact runtime environment the service uses.

Using an elastic infrastructure

Provides an elastic infrastructure to meet the on-demand requirements of the microservices. By scaling up only the microservices that require it, the system needs fewer resources and can react quicker because each service is very fast to start. OpenShift Container Platform automates the provisioning, management, and scaling of containerized applications and provides on-demand scaling by providing automatic horizontal pod scaling. As load increases on particular microservices, OpenShift automatically increases available resources for those microservices.

Self-serving Deployments

Use scripting or other tools to automate tasks related to the service infrastructure provision and deployment of the service. This way development and operation teams can spin up new environments as needed with no dependency of other teams. Using the same scripts or tools in both development and production environments also produce fewer inconsistencies between environments and ease the maintainability and error fixing.

Infrastructure As Code (IAC) is an approach that uses descriptive language to code versatile and adaptive provisioning and deployment processes using tools such as Ansible, Puppet, or Jenkins. Within IAC, operation teams manage the automated provisioning or deployment scripts, or any other resources related to this, in the version control system just like any other application code. This approach provides the benefits of versioning to the provisioning tasks and reduces the gap between development and operation teams.

Continuous integration and delivery pipeline

Continuous integration and delivery (CI/CD) is the practice of building, testing, and delivering software as the code is developed. A key to continuous delivery is to automate the entire pipeline, including code check-in, builds, tests, and deployments across multiple environments. When the build fails, make sure that fixing the broken code takes priority over other development tasks. This way code issues are detected and fixed before any new development task starts, avoiding them to base on broken code. Also, using continuous delivery and quick fixing broken code keeps software always in a deployable status, easing and speeding up the release process.

You can find dozens of tools for including CI/CD into your workflows. Some of the most common tools are Jenkins, Tekton, CircleCI, TeamCity, Travis XI or Bamboo. Some version control systems, such as GitHub or GitLab, also include CI/CD features.

Understanding the Complexities of Microservice Architecture

Designing and building a large scale distributed application with many microservices is occasionally challenging and complex. These challenges tend to arise when planning and managing a large number of small components interacting with each other. Testing an individual microservice is simple, but when multiple services depend upon each other, you have to test them together. Furthermore, introducing dependencies between microservices can cause the system to communicate in unpredictable ways if one of the services does not behave as expected.

Additionally, maintaining data consistency might be difficult in a large distributed system. Every microservice can have its own persistent storage, therefore changing shared entity data formats

Chapter 1 | Describing a Microservice Architecture

cannot happen without changing all the microservices. For example, if two microservices persist instances of the entity `Employee` and the data model for `Employee` changes, then both services must update the entity.

Tracing and monitoring are also more complex for microservice-based applications. Instead of monitoring a single application server, microservices are typically run on multiple servers and writing to several log files. Collecting all the log information and storing it in one place to analyze and visualize it efficiently is complex.

Finally, securing microservice applications can be a challenging and complicated task. Users authenticate with the UI layer just like in a monolithic application, but individual services must be protected by authentication and authorization as well. The added communication between services in a microservice-based application creates more opportunities for security risks and requires more points of authentication.

Describing the Principles of a Resilient Microservice Architecture

Designing a successful microservice architecture require forgetting some of the principles architects followed for a long time and applying new ones. Microservice architecture patterns relate to the size, lifecycle and behavior of microservices and their relations.

Domain-driven Design and Bounded Contexts

Microservices are designed based on Domain-Driven Design, an approach to software development which requires a close understanding of the business domain. Domains can have multiple bounded contexts, which encapsulate the details of a single business domain and defines integration points with other bounded contexts.

For example, in an e-commerce application: order, delivery, and billing are all examples of different bounded contexts. Each bounded context maintains a data model derived from the bounded domain model. Each bounded context translates into one or more microservices.

Advantages of the Bounded Domain Model

- Changes in the domain model only affect a limited number of services.
- Services are autonomous.

Disadvantages of the Bounded Domain Model

- Excellent domain knowledge is required to define a bounded context.
- Complexity increases to keep the system consistent between contexts.

Deploy Independently on a Lightweight Runtime

Deploying services independently is one of the most critical principles in microservices architecture. This means a microservice must be a self-contained deployment unit that can be deployed and used with independence from other services.

Microservices are stateless and maintain state outside the application in databases or data grids. Thanks to this, a microservice can scale easily and deploy independently of other services.

Microservices are good candidates for lightweight, embedded runtimes. Unlike full-fledged application servers, microservices are small, compact and single-purposed components.

Some of the desired runtimes for microservices are Quarkus, Eclipse Vert.x, and Sprint Boot. Additionally, JBoss Enterprise Application Platform (EAP) is considered lightweight as most of EAP's application server components start on-demand.

Microservices are packaged and delivered as containers, which include the runtime and its dependencies. Fully-automated build and deployment pipelines are a requirement to support continuous integration and delivery of microservices.

Design for Failure

Distributed applications can fail due to code errors, or hardware or network failures. Developers must design applications that anticipate and can recover from such failures.

Developers can incorporate various design patterns to make microservices applications fault-tolerant.

- Provide high availability microservices for failure management.
- Use the circuit breaker design pattern to prevent service or network failure from cascading to other services. The fault tolerance chapter of this course covers this pattern in more detail.
- Use the bulkhead design pattern to prevent overload of an individual service, and to isolate failures from rippling throughout the system by limiting concurrent access to dependent services and minimizing the impact of a dependent service that is down. The fault tolerance chapter of this course covers this pattern in more detail.

Applications require extensive testing to assess the effects of a variety of different types of system failures on the end-user experience. Real-time monitoring is necessary to detect these failures quickly and alert developers. Additionally, use a self-healing infrastructure, such as Kubernetes and OpenShift Container Platform to leverage readiness and liveness checks, which monitor the state of running services and allow the deployment platform to act on failures automatically. These approaches are discussed in detail later in the course.



References

What are Microservices?

<http://microservices.io/>

► Quiz

Describing Microservices

Choose the correct answers to the following questions.

- ▶ 1. **Which of the following statements about monolithic applications is true.**
 - a. A monolithic application is always packaged into a .tar.gz file.
 - b. A monolithic application is a lightweight application, which is divided into a multiple services and a client tier.
 - c. Monolithic applications are often complex to build and can be difficult to maintain.
 - d. Individual pieces of a monolithic application are typically developed by individual small autonomous teams.

- ▶ 2. **Which two of the following statements about microservices are true? (Choose two..)**
 - a. Microservices are managed centrally, and typically share one common database.
 - b. Microservices are typically deployed on full-fledged application servers, which manage their lifecycle.
 - c. Microservices are self-contained, independent services.
 - d. Microservices within a single organization can be built using different programming languages.

- ▶ 3. **A company named "At Your Doorstep Inc." is considering using microservice architecture for their new grocery delivery application. They have two small teams to manage IT development and operations in seven different locations. Which two of the following statements support their decision in favor of microservices? (Choose two.)**
 - a. All order operations must be centralized in a single data center.
 - b. Microservices can be built for each business operation.
 - c. Microservices must be deployed manually to ensure there are no errors.
 - d. Individual services in a microservice-based application can go live quickly by using continuous integration and delivery.

- ▶ 4. **Which of the following statements regarding quicker development of microservices is true.**
 - a. Microservices are developed and managed by large teams to support quicker development.
 - b. A separate development and operations team is key to quicker development.
 - c. Individual microservices are often developed, maintained, and operated by small teams.
 - d. All microservices in an application are tested collectively, which reduces the testing time.

► 5. Which of the two following statements about microservices architecture are correct?

(Choose two.)

- a. Microservice applications are required to be deployed on the same physical host.
- b. Microservices architecture supports high availability of individual microservices.
- c. Microservices can not be used if a company is embracing DevOps.
- d. Microservices are designed using a bounded context, which can communicate with other bounded contexts.

► Solution

Describing Microservices

Choose the correct answers to the following questions.

- ▶ 1. **Which of the following statements about monolithic applications is true.**
 - a. A monolithic application is always packaged into a .tar.gz file.
 - b. A monolithic application is a lightweight application, which is divided into a multiple services and a client tier.
 - c. Monolithic applications are often complex to build and can be difficult to maintain.
 - d. Individual pieces of a monolithic application are typically developed by individual small autonomous teams.

- ▶ 2. **Which two of the following statements about microservices are true? (Choose two..)**
 - a. Microservices are managed centrally, and typically share one common database.
 - b. Microservices are typically deployed on full-fledged application servers, which manage their lifecycle.
 - c. Microservices are self-contained, independent services.
 - d. Microservices within a single organization can be built using different programming languages.

- ▶ 3. **A company named "At Your Doorstep Inc." is considering using microservice architecture for their new grocery delivery application. They have two small teams to manage IT development and operations in seven different locations. Which two of the following statements support their decision in favor of microservices? (Choose two.)**
 - a. All order operations must be centralized in a single data center.
 - b. Microservices can be built for each business operation.
 - c. Microservices must be deployed manually to ensure there are no errors.
 - d. Individual services in a microservice-based application can go live quickly by using continuous integration and delivery.

- ▶ 4. **Which of the following statements regarding quicker development of microservices is true.**
 - a. Microservices are developed and managed by large teams to support quicker development.
 - b. A separate development and operations team is key to quicker development.
 - c. Individual microservices are often developed, maintained, and operated by small teams.
 - d. All microservices in an application are tested collectively, which reduces the testing time.

► 5. Which of the two following statements about microservices architecture are correct?

(Choose two.)

- a. Microservice applications are required to be deployed on the same physical host.
- b. Microservices architecture supports high availability of individual microservices.
- c. Microservices can not be used if a company is embracing DevOps.
- d. Microservices are designed using a bounded context, which can communicate with other bounded contexts.

Describing Microservice Architecture Patterns

Objectives

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

Comparing Synchronous and Asynchronous Interprocess Communication

Microservices are typically deployed individually, but most enterprise-level microservice architectures require that the services interact with each other as well as other external services. This communication is achieved using an interprocess communication (IPC) mechanism. Depending on the application's requirements, communication between microservices can be synchronous or asynchronous.

Synchronous Communication

Synchronous communication is based on a request and response model. In this model, the client waits for a timely response from a service. A basic example is communicating with a REST service over HTTP.

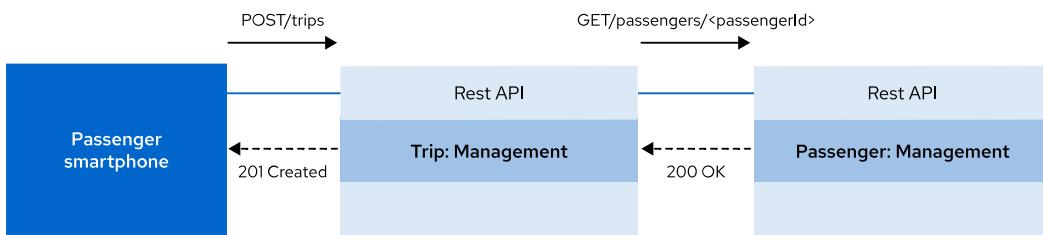


Figure 1.3: Synchronous communication between services

In this diagram, a passenger is using a smart phone client to buy a new train ticket. The phone client sends a POST request to the trip management service. The trip management service sends a GET request to the passenger management service. The passenger management service sends a response back with the status 200 OK to the trip management, which returns a success status 201 CREATED.

In this example, both clients wait for a response.

Advantages

- Easy to program and test.
- Provides a better real-time response.
- Firewall friendly, because it uses standard ports.
- No need for an intermediate broker or other integration software.

Disadvantages

- Supports only request-and-response style interaction.
- Requires both client and service to be available for the entire duration of the exchange.
- Forces the client to know the URL (location) of the service or use a service discovery mechanism to locate the service instances.

Asynchronous Message-based Communication

Microservices can communicate using an asynchronous message-based communication such as the AMQP or MQTT protocols. Microservices can use other message-based patterns such as point-to-point, publish-and-subscribe, request-and-reply, or request-and-notification. Asynchronous communication is non-blocking, therefore the client can continue making requests without the need to wait to receive a response.

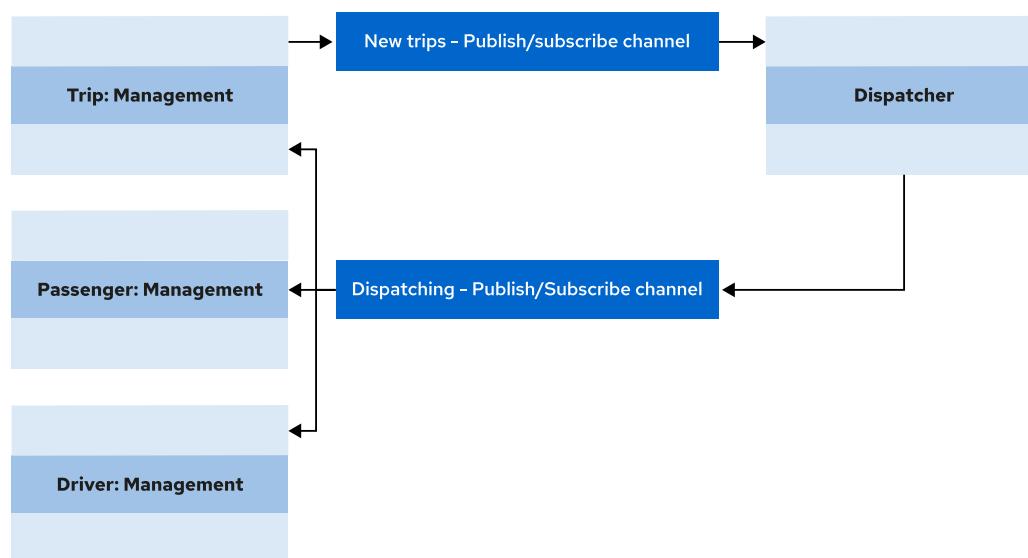


Figure 1.4: Asynchronous communication between services

In the diagram, the three services, trip management, passenger management and driver management, receive messages from a dispatcher using a single publish-subscribe channel. A trip management service uses another publish-subscribe channel to send messages to the dispatcher.

In this example, the dispatcher service does not reply directly to the trip management service when a new trip is submitted. Instead, the dispatcher service uses a different channel to reply to the trip management service and to notify the passenger and driver management services. This asynchronous approach allows the trip management service to continue processing user requests for more new trips without waiting on the dispatcher's processing and subsequent response.

Advantages

- *Decouples the client from the service:* The client is unaware of service instances. No discovery mechanism required.
- *Implements message buffering:* The message broker queues messages in a message buffer when the consumer is slow or unavailable.

- *Enables flexible client-service interaction:* The communication between client and service is flexible. The clients need not be available to receive the messages. Messaging supports various styles to ensure message delivery.

Disadvantages

- *Increases operational complexity:* There is additional configuration for message components. The message broker component must be highly available to ensure system reliability.
- *Difficult implementing request-and-response based interactions:* Each request message must contain a reply channel and a correlation identifier. The service writes the response and the correlation identifier to the reply channel. The client identifies the message using that correlation identifier.

Describing Fault Tolerance

Microservices-based applications can still fail like any other distributed application. The failure can happen in an individual service or in a chain of services. If one service in the dependency chain fails, then all of the upstream clients are affected. This causes cascaded failure.

Fault tolerance means that services can handle failures, and the end user experience is not impacted by a single service failure. Fault tolerance is imperative in a microservice-based application, because there are so many points of failure.

Fault tolerance implementations focuses on five topics.

Time-out

When a service requests another service, the later might take an excessive amount of time to respond, making the first service unresponsive. Time-out defines a time limit for the client service when awaiting for a response.

Retry

A service might be suffering transient circumstances forbidding it to properly function. A client hitting a temporarily failing service can repeat the same request some time later, expecting the failing service to recover.

Fallback

A failing service does not imply a direct failure in the client. In some cases, clients can revert the request to another service, or even provide a previously built response.

Circuit Breaker

If a client or a gateway is aware of a failing service, then it can avoid directing request to that service. The client can respond by either redirecting requests to other services, using a fallback response or by promptly returning a failure response without even requesting the failing service.

Bulkhead (or Connection Pools)

A service can limit the number of concurrent requests and the number of requests awaiting response. This limit saves the service from becoming overwhelmed or overusing underlying resources.

Describing the Circuit Breaker Pattern

The circuit breaker pattern helps ensure a microservice can gracefully handle downstream failures of services it depends on. A circuit breaker object wraps the function calls to the dependent services and monitors the success of the calls.

When everything is normal and the calls are succeeding, the circuit breaker is in the closed state. When the number of failures (an exception or timeout during the call) reaches a preconfigured threshold, the circuit breaker trips open. When the circuit breaker is open, no calls are made to the dependent service, but a fallback response is returned. After a configurable amount of time, the circuit breaker moves to a half-open state. In the half-open state, the circuit breaker executes the service calls periodically to check the health of the dependent service. If the service is healthy again, and the test calls are successful, the circuit state switches back to closed.

The circuit breaker lifecycle is shown in the following diagram.

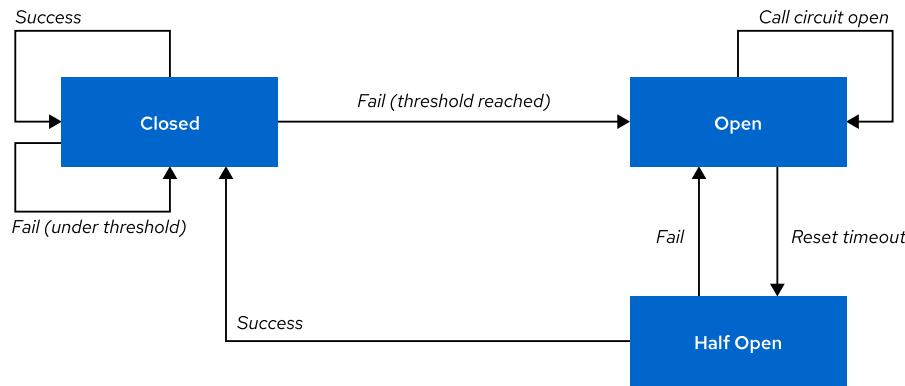


Figure 1.5: The circuit breaker lifecycle

Describing the Bulkhead Pattern

Use the bulkhead pattern to isolate dependencies from each other and to limit the number of concurrent threads attempting to access each of them. This isolation means that this call is restricted from using more than these threads, which would impact performance of other parts of the service, should the calls become unsaturated, or the dependent service is not performing well.

When an application makes a request for connection to a component behind the bulkhead, the bulkhead checks for the availability of a connection to the requested component. If the number of concurrent connections to the component is below the threshold, then the bulkhead allocates the connection. When no more concurrent connections are allowed, the bulkhead awaits for a predefined interval of time. If no connections become available within this duration, then the bulkhead rejects the call.

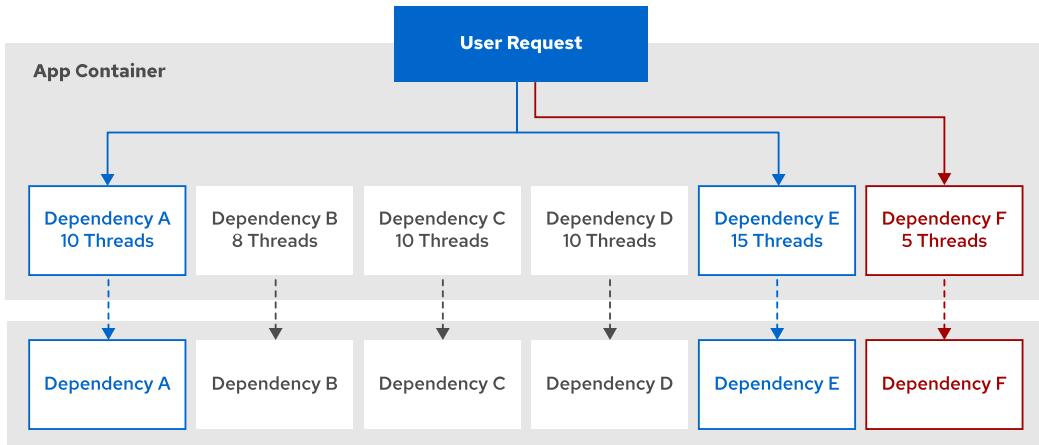


Figure 1.6: Dependent service failure limited by the bulkhead pattern

In the preceding figure, one or more clients perform 10 concurrent requests to the application components A, E and F. The bulkheads for each of those components are restricted to 10, 15 and 5 concurrent requests respectively. Components A and E can handle all concurrent requests but component F can only handle 5 concurrent requests, so the bulkhead refuses some of them.

Reviewing Fault Tolerance Implementations

Quarkus relies on the SmallRye implementation of the Eclipse Microprofile Fault Tolerance specifications. Originally based on Hystrix, Eclipse Microprofile Fault Tolerance defines annotations enabling fault tolerance features in Java microservices.

Other implementations of fault tolerance features are available in.

- Hystrix, the fault tolerance implementation from Netflix.
- Red Hat OpenShift Service Mesh and its upstream project Istio, which provides fault tolerance at network communications level.
- The circuit breaker component at Vert.x.
- Camel Fault Tolerance EIP.
- Apache Commons Circuit Breaker interface and implementations.

Describing Distributed Tracing

In monolithic applications, tracing a single user's interaction with the system could be accomplished by isolating a single instance of an application and reproducing a problem. Microservices-based applications are complex; a single microservice cannot provide the behavior, performance, or correctness of the entire application.

Distributed tracing is a tool that provides complete information of the application's behavior as the requests pass through multiple services. Distributed tracing tools can profile running services for reporting purposes. These tools collect data in the central aggregator for storage, reporting, and visualization.

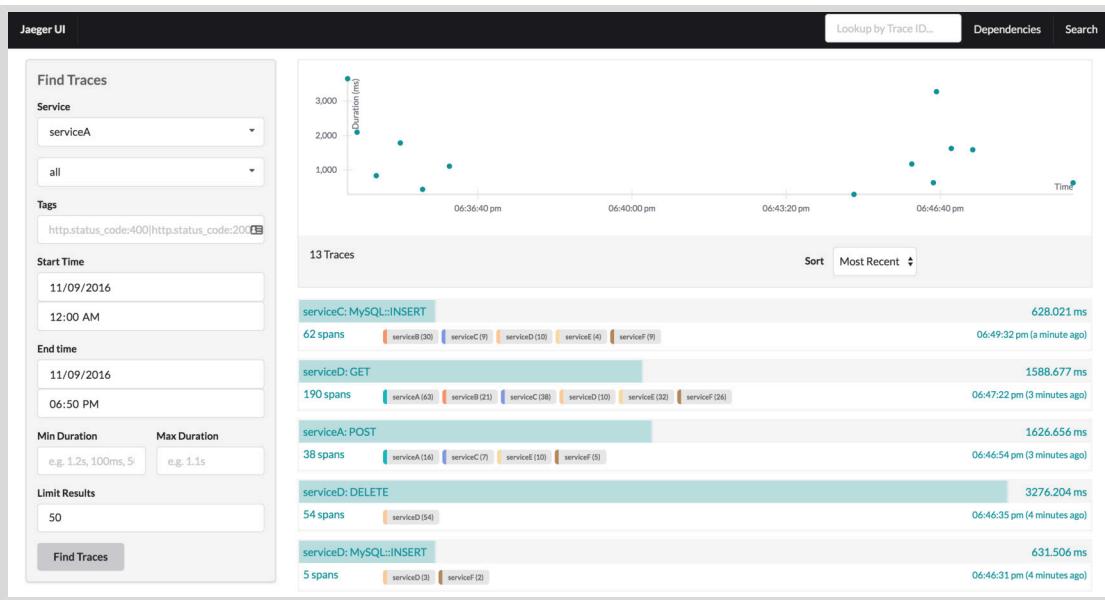


Figure 1.7: Distributed tracing in Jaeger - visualizing traces

Distributed tracing assigns each request a unique request ID or trace ID. The trace ID is passed to all services that are involved in handling the request and the trace ID is included in all log messages. Every service adds a new span ID to the trace. Span data are collected by or sent to a central aggregator for storage and visualization.

Services involved in the trace can add metadata, such as start and stop timestamps, and business-relevant data, to the span.

Presenting the OpenTracing API

OpenTracing API is a vendor-neutral open standard for tracing. OpenTracing provides distributed tracing of applications with minimal effort. It is supported across many languages, such as Java, JavaScript, and Go. It is implemented in Zipkin from Twitter, Jaeger from Uber, and Hawkular APM from Red Hat.

Adding Metrics to a Microservice

Metrics are numeric information bits about the usage and the performance of the application. Examples of common metrics are the number of requests received by an endpoint, the time needed to respond, or amount of memory used by the service.

Day two operations can use metrics information to understand the behavior of the microservices and act accordingly. Metrics allow operators to detect failing services or nodes, leverage and redistribute application workload, or find bottlenecks or points of improvement in the application.

There is no difference between generating metrics in microservice applications or in monolithic applications. But gathering metrics in microservice-based applications requires an external process centralizing the metrics. Tools such as Prometheus and Grafana gather metrics and provide visualization tools to get the best use of the metrics.

Maintaining Security in Microservices

Maintaining identity and access management through a series of independent services can be a real challenge in microservice-based applications. Requiring every service call to include an authentication step is not ideal. Fortunately, there are a number of possible solutions, including:

Single Sign-On

A common approach for authentication and authorization that permits the client to use a single set of login credentials to access multiple services.

Distributed sessions

A method of distributing identity between microservices and the entire system.

Client-side token

The client requests a token and uses this token to access a microservice. The token is signed by an authentication service. A microservice validates the token without calling the authentication service. JSON Web Token (JWT) is an example of token-based authentication.

Client-side token with API Gateway

API gateways cache client-side tokens. The validation of tokens is handled by the API gateway.



References

Microservices patterns by Chris Richardson Manning Publications

<https://www.manning.com/books/microservices-patterns>

Eclipse Microprofile Fault Tolerance Specifications

<https://download.eclipse.org/microprofile/microprofile-fault-tolerance-2.1.1/microprofile-fault-tolerance-spec.html>

SmallRye Fault Tolerance

<https://github.com/smallrye/smallrye-fault-tolerance>

Jaeger site

<https://www.jaegertracing.io/>

Prometheus documentation

<https://prometheus.io/docs/>

Grafana

<https://grafana.com/>

OpenTracing project

<https://opentracing.io>

OpenTelemetry project

<https://opentelemetry.io/>

► Quiz

Describing Microservice Architecture Patterns

Choose the correct answers to the following questions.

- ▶ 1. **Which three of the following statements about an asynchronous message-based communication in microservices are true? (Choose three.)**
 - a. Microservices can use the AMQP or MQTT protocols for asynchronous message-based communication.
 - b. An advantage of asynchronous communication is that messages can be buffered using message queues.
 - c. In asynchronous communication, clients must be available to receive messages to avoid losing the messages.
 - d. Using an asynchronous communication mechanism such as AMQP requires the use of a message broker or other similar middleware integration technology.
- ▶ 2. **Which three of the following are microservice architecture patterns? (Choose three.)**
 - a. Fault tolerance.
 - b. Service security.
 - c. Request tracing.
 - d. Self-serving deployments.
- ▶ 3. **Which two of the following statements regarding the fault tolerance patterns are true? (Choose two.)**
 - a. The circuit breaker pattern has three distinct states: open, half-open, and broken.
 - b. While in the half-open state, a circuit breaker periodically allows service calls to execute to check the health of the dependent service.
 - c. The bulkhead pattern allows you to allocate a dedicated thread pool or a semaphore for each dependent service call, limiting the number of concurrent calls to the service.
 - d. The bulkhead pattern is used as a layer of abstraction between a client and services.
- ▶ 4. **Which two of the following statements about distributed tracing and aggregated logging are correct? (Choose two.)**
 - a. OpenTracing API is a popular tracing implementation designed for monolithic applications.
 - b. Distributed tracing tools collect the trace data for storage and visualization.
 - c. Distributed tracing is only useful during development.
 - d. OpenTracing API is an open standard for tracing with implementations in multiple languages.

► Solution

Describing Microservice Architecture Patterns

Choose the correct answers to the following questions.

- ▶ 1. **Which three of the following statements about an asynchronous message-based communication in microservices are true? (Choose three.)**
 - a. Microservices can use the AMQP or MQTT protocols for asynchronous message-based communication.
 - b. An advantage of asynchronous communication is that messages can be buffered using message queues.
 - c. In asynchronous communication, clients must be available to receive messages to avoid losing the messages.
 - d. Using an asynchronous communication mechanism such as AMQP requires the use of a message broker or other similar middleware integration technology.
- ▶ 2. **Which three of the following are microservice architecture patterns? (Choose three.)**
 - a. Fault tolerance.
 - b. Service security.
 - c. Request tracing.
 - d. Self-serving deployments.
- ▶ 3. **Which two of the following statements regarding the fault tolerance patterns are true? (Choose two.)**
 - a. The circuit breaker pattern has three distinct states: open, half-open, and broken.
 - b. While in the half-open state, a circuit breaker periodically allows service calls to execute to check the health of the dependent service.
 - c. The bulkhead pattern allows you to allocate a dedicated thread pool or a semaphore for each dependent service call, limiting the number of concurrent calls to the service.
 - d. The bulkhead pattern is used as a layer of abstraction between a client and services.
- ▶ 4. **Which two of the following statements about distributed tracing and aggregated logging are correct? (Choose two.)**
 - a. OpenTracing API is a popular tracing implementation designed for monolithic applications.
 - b. Distributed tracing tools collect the trace data for storage and visualization.
 - c. Distributed tracing is only useful during development.
 - d. OpenTracing API is an open standard for tracing with implementations in multiple languages.

► Guided Exercise

Verifying OpenShift Credentials

In this exercise, you will configure the workstation to access an OpenShift Container Platform cluster, used by this course.

Outcomes

You should be able to configure your workstation to access an OpenShift cluster.

Before You Begin

To perform this exercise, ensure you have:

- Access to the DO378 course in the Red Hat Training Online Learning Environment.
- The connection parameters and a developer user account to access an OpenShift cluster managed by Red Hat Training.

Instructions

- 1. Before starting any exercise, you must configure your **workstation VM**.

For the following steps, use the values the Red Hat Training Online Learning environment provides to you when you provision your online lab environment.

OpenShift Details		
Username	RHT_OCP4_DEV_USER	youruser
Password	RHT_OCP4_DEV_PASSWORD	yourpassword
API Endpoint	RHT_OCP4_MASTER_API	https://api.na46-stage2.dev.nextcl.com:6443
Console Web Application	https://console-openshift-console.apps.na46-stage2.dev.nextcl.com	
Cluster Id	your-cluster-id	

Open a terminal on your **workstation VM** and execute the following command. Answer its interactive prompts to configure your workstation before starting any other exercise in this course.

If you make a mistake, you can interrupt the command at any time using **Ctrl+C** and start over.

```
[student@workstation ~]$ lab-configure
```

- 1.1. The **lab-configure** command starts by displaying a series of interactive prompts, and will try to find some sensible defaults for some of them.

This script configures the connection parameters to access the OpenShift cluster for your lab scripts

- Enter the API Endpoint: **api.cluster.domain.example.com:6443** ①
- Enter the Username: **youruser** ②
- Enter the Password: **yourpassword** ③

...output omitted...

- 1.2. The **lab-configure** command prints all the information that you entered and tries to connect to your OpenShift cluster.
- 1.3. If **lab-configure** finds any issues, then it displays an error message and exits. You will need to verify your information and run the **lab-configure** command again. The following listing shows an example of a verification error.

```
...output omitted...
Generated config:
· API Endpoint: https://api.cluster.domain.example.com:6443
· Wildcard Domain: apps.cluster.domain.example.com
· Nexus Server Host: nexus-common.cluster.domain.example.com
· Username: youruser
· Password: yourpassword
```

Verifying your API Endpoint...

Verifying your Nexus host...

Saving your Maven settings file...

Verifying your OpenShift developer user credentials...

...output omitted...

- 1.3. If **lab-configure** finds any issues, then it displays an error message and exits. You will need to verify your information and run the **lab-configure** command again. The following listing shows an example of a verification error.

```
...output omitted...
Verifying your API Endpoint...

ERROR:
Cannot connect to an OpenShift 4.x API using your URL.
Please verify your network connectivity and that the URL does not point to an
OpenShift 3.x nor to a non-OpenShift Kubernetes API.
No changes made to your lab configuration.
```

- 1.4. The `lab-configure` command automatically obtains the cluster and wildcard domains from the API Endpoint you provided.

```
...output omitted...
Verifying your cluster configuration...
...output omitted...
```

- 1.5. If all checks pass, then the `lab-configure` command saves your configuration.

```
...output omitted...
Saving your lab configuration file...

All fine, lab config saved. You can now proceed with your exercises.
```

If there were no errors saving your configuration, then you are ready to start any of the exercises in this course. If there were any errors, then do not try to start any exercise until you can execute the `lab-configure` command successfully.

Finish

This concludes the guided exercise.

Summary

In this chapter, you have learned that:

- Microservices and microservices architectures improve the application development, deployment, and maintenance processes.
- Adopting microservices architectures and creating resilient microservice applications requires specific principles, different from monolithic applications.
- Microservices architecture uses dedicated patterns such as service discovery, API gateway, distributed tracing, log aggregation, and various fault tolerance patterns.

Chapter 2

Implementing a Microservice with Quarkus

Goal

Develop Quarkus microservice applications.

Objectives

- Describe the features of Quarkus.
- Implement a microservice using the CDI, JAX-RS, and JSON-P specifications of MicroProfile.
- Persist data with simplified Panache Entities.

Sections

- Describing Quarkus as a Microservice Runtime (and Quiz)
- Implementing a JSON Microservice with CDI and JAX-RS (and Guided Exercise)
- Persisting Data with Panache (and Guided Exercise)

Lab

Implementing a Microservice with Quarkus

Describing Quarkus as a Microservice Runtime

Objectives

After completing this section, you should be able to describe the features of Quarkus.

History of the MicroProfile Specification

As the Java ecosystem has matured, the number of vendors and differing server technologies in the market have grown significantly. Developers have an enormous amount of Java EE experience, and the tools built to develop Java applications have become quite advanced. The industry is beginning to pivot towards a microservices-based architecture for new application developments, which are also mostly cloud-native applications.

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

MicroProfile is not meant to be a full standard like a Java Standards Request (JSR). These complete standards require years to finalize and are cumbersome to update. Because microservices applications are constantly evolving, no standard is ever truly final.

By focusing only on the high-level areas of commonality found in Java microservices applications, both vendors and the community can innovate collaboratively without the need to use a rigid standard. The community can always opt to standardize functionality later when there is more stability in a future version of MicroProfile. This approach, although different from the rigid JSR process of the Java world, allows community members and vendors to continue to innovate independently whilst using and contributing to MicroProfile wherever there is commonality across different microservices.

The initial 1.0 version of the MicroProfile specification included only the JAX-RS, CDI, and JSON-P specifications from Java EE, which is the absolute bare minimum required to build a microservice in Java. MicroProfile uses these traditional Java EE specifications because after over a decade of investment and optimization, these Java EE implementations are quite efficient. Additionally, there is the added benefit of developer familiarity, as these specifications are ubiquitous in Java development.

One of the stated goals of the MicroProfile initiative is to utilize existing API specifications where possible and combine them with new ones to create a baseline platform optimized for developing microservices in the cloud. The MicroProfile community continues to play an active role in defining the future versions of MicroProfile as Java EE technologies continue to evolve.

Quarkus for Microservices Development

One of the first implementations of the MicroProfile specification was the Wildfly Swarm / Thorntail application server, which is based on the Wildfly server code. It failed to provide the startup performance required in the microservices environment due to the dynamic nature of the majority of Java EE specifications.

To achieve the desired performance, these implementations needed a rewrite that would reduce startup time. This was done by relying on a more complete build process, which performs activities normally done during startup or at runtime, including configuration, bootstrapping, and reflection preparation. Quarkus emerged as the collective effort from the different technologies that encompass the MicroProfile implementation.

Quarkus focuses on these key factors to provide the best development experience for microservices.

Container Native

First and foremost, Quarkus is a cloud-native framework, which means that the main deployment platform is the container. To facilitate container-first development, Quarkus focuses on the following properties:

Fast startup

To rapidly scale out the services and to support *Function as a Service* (FaaS), Quarkus aims for startup times under one second.

Small memory footprint

In containerized environments, memory is a valuable resource because memory usage is multiplied when scaling out. By removing unnecessary classes at runtime, Quarkus achieves a lower memory footprint.

Smaller disk usage

By only copying the required classes and resources to the JAR, Quarkus reduces the size of the container image. When deploying to containers, images have to travel through the network and a big image can increase download and startup times.

Unification of Imperative and Reactive Programming Models

Java developers can use the familiar imperative programming model they already know. At the same time, they can take advantage of new cloud-native paradigms, such as event-driven, asynchronous, and reactive models. Because these different programming models coexist in the same framework, developers can choose the best approach.

Developer Centric

Quarkus brings together several technologies under a unified framework to work seamlessly under the same platform, which provides the following advantages:

- Configuration of all technologies in the same configuration file.
- Instant live reloading of changes in developer mode.
- An opinionated framework that sets usable defaults.
- Integrated native build for all included libraries.

Standard Libraries

The use of MicroProfile eases the transition from traditional enterprise Java EE development to microservices development. Developers will find all of the traditional standards, but now with a focus on development of lean services. This reuse of previous knowledge enables developers to quickly start developing cloud-native applications.

One Framework, Two Modes

One of the main drives in Quarkus is to provide the best support for Serverless, FaaS, and other cloud-based workloads. In an environment of this size, memory usage and startup time become critical.

Despite the performance improvements over traditional cloud frameworks, a one second startup time might still be too long. For this, Quarkus offers full support of the GraalVM's *Ahead-of-Time compilation* (AoT).

GraalVM is a special JVM from Oracle, which offers AoT compilation to executable Java applications, among other features. The *Close World Assumption* means that the JVM needs to know at compilation time which classes the application is going to use. To support native executable generation, this assumption must be met. Quarkus achieves this by providing its own library implementations and assessing the required classes at build time instead of at runtime through reflection.

During AoT compilation, GraalVM removes all unneeded classes from the application, dependent libraries, and JVM.

This native support of AoT and GraalVM makes Quarkus well-suited for cloud deployments.

Advantages of Quarkus

Compared to other Java-based cloud-native frameworks, Quarkus offers the following benefits:

- Faster response times, in general.
- Lower memory usage.
- Higher throughout in reactive routes.
- Full container image support.
- Compliance to MicroProfile conformance tests.



References

Product Documentation for Red Hat build of Quarkus

https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11

Microprofile

<https://micropattern.io/>

Quarkus (Upstream open source project)

<https://quarkus.io/>

► Quiz

Describing Quarkus as a Microservice Runtime

Choose the correct answers to the following questions.

- ▶ 1. **Which three of the following standards were the base of the MicroProfile 1.0 release? (Choose three.)**
 - a. JPA.
 - b. CDI.
 - c. JAX-RS.
 - d. JSON-P.

- ▶ 2. **Which two of the following statements about the development process in Quarkus are true. (Choose two.)**
 - a. Each library is configured in its own configuration file.
 - b. Provides live code changes in developer mode.
 - c. Integrates native build for included libraries.

- ▶ 3. **Why is memory usage so important in a cloud-native environment?**
 - a. As an application scales out, the number of developers grows.
 - b. As an application scales out, the memory usage is multiplied.
 - c. As an application scales out, the likelihood of a memory leak increases.
 - d. Memory usage is not that important in a cloud-native environment.

- ▶ 4. **Which two of the following statements are benefits of a shorter startup time in a cloud-native environment? (Choose two.)**
 - a. A shorter startup time can decrease application down time.
 - b. A shorter startup time can increase application down time.
 - c. A shorter startup time reduces overall deployment time.
 - d. A shorter startup time can increase developer productivity.

- ▶ 5. **How does Ahead of Time compilation (AoT) reduce memory usage?**
 - a. AoT does not reduce memory usage.
 - b. AoT removes classes until they are needed.
 - c. AoT uses a special JVM.
 - d. AoT removes unnecessary classes.

► Solution

Describing Quarkus as a Microservice Runtime

Choose the correct answers to the following questions.

- ▶ 1. **Which three of the following standards were the base of the MicroProfile 1.0 release? (Choose three.)**
 - a. JPA.
 - b. CDI.
 - c. JAX-RS.
 - d. JSON-P.

- ▶ 2. **Which two of the following statements about the development process in Quarkus are true. (Choose two.)**
 - a. Each library is configured in its own configuration file.
 - b. Provides live code changes in developer mode.
 - c. Integrates native build for included libraries.

- ▶ 3. **Why is memory usage so important in a cloud-native environment?**
 - a. As an application scales out, the number of developers grows.
 - b. As an application scales out, the memory usage is multiplied.
 - c. As an application scales out, the likelihood of a memory leak increases.
 - d. Memory usage is not that important in a cloud-native environment.

- ▶ 4. **Which two of the following statements are benefits of a shorter startup time in a cloud-native environment? (Choose two.)**
 - a. A shorter startup time can decrease application down time.
 - b. A shorter startup time can increase application down time.
 - c. A shorter startup time reduces overall deployment time.
 - d. A shorter startup time can increase developer productivity.

- ▶ 5. **How does Ahead of Time compilation (AoT) reduce memory usage?**
 - a. AoT does not reduce memory usage.
 - b. AoT removes classes until they are needed.
 - c. AoT uses a special JVM.
 - d. AoT removes unnecessary classes.

Implementing a JSON Microservice with CDI and JAX-RS

Objectives

After completing this section, you should be able to implement a microservice using the CDI, JAX-RS, and JSON-P specifications of MicroProfile.

MicroProfile and Microservice Basics

Quarkus takes advantage of the MicroProfile specification. Using Quarkus, developers can build RESTful web services, which are the foundation of microservice architectures.

The following components are fundamental for developers creating microservices:

- Quarkus DI, which is comparable to the *Contexts and Dependency Injection for Java* (CDI) specification
- Jakarta RESTful Web Services (JAX-RS)
- JSON processing

Using all three components, you can build basic RESTful web services in a straightforward and declarative manner.

Reviewing Dependency Injection

The Java EE 6 specification introduced the CDI specification in 2009. More generally, developers use the *Dependency Injection* (DI) design pattern to create loosely-coupled classes that are easier to unit test. When using Dependency Injection, instead of relying on a global state, dependencies of classes and methods are declared. The DI framework injects those dependencies into the respective constructors, properties, and methods.

CDI does so by binding the lifecycle of stateful components to domain-specific lifecycle contexts. CDI promotes loose decoupling by providing the following features:

- Uses an event-notification model
- Enables the usage of decorators for injected components
- Enables the association of interceptors with components by using type-safe bindings

Declaring Beans

A *bean* is any object managed by CDI. A bean specifies only the type and semantics of other beans it depends on. CDI manages the lifecycle of beans and references. It can then share both with other beans managed by CDI.

There is no need for beans to be aware of the lifecycle, implementation, threading, or other clients. This independence eases code maintenance.

You can use a variety of annotations to define beans. Some of the annotations available are the following:

- `@ApplicationScoped`

- `@SessionScoped`
- `@ConversationScoped`
- `@RequestScoped`
- `@Interceptor`
- `@Decorator`
- `@Dependent`

Additionally, this includes any annotation that is itself annotated with `@Stereotype`.

Bean Scopes

Application

When you annotate a bean with `@ApplicationScoped`, the object will live throughout the life of the application. It will be shared to all method calls to the application, meaning that it will be created the moment the application starts and remain there until its end.

Session

If you annotate a bean with `@SessionScoped`, then the object is created when a session starts and will be tied to the session as its context. This means that whenever a request is completed under a session, the bean will be made available. Likewise, if the session is destroyed or it times out, then the created bean will be lost as well.

Request

Beans under the `@RequestScoped` annotation have their lifetimes tied to each individual request. They are created when any service invocation starts, and their context is destroyed when the request completes.

Using DI in Quarkus

Quarkus' DI implementation, although based on CDI, only implements a subset of the CDI specification. This is mainly due to architectural goals, many of which are related to native compilation. The currently-documented limitations of the CDI implementation in Quarkus include the following:

- No support for `@ConversationScoped` decorators, portable extensions, specialization, passivation and passivizing scopes, and interceptor methods on superclasses.
- BeanManager includes only the methods `getBeans()`, `createCreationalContext()`, `getReference()`, `getInjectableReference()`, `resolve()`, `getContext()`, `fireEvent()`, `getEvent()` and `createInstance()`.
- DI ignores the content of the descriptor in the `beans.xml` file. By ignoring the content, Quarkus supports only plain beans from the `beans.xml` file, and provides no support to configure interceptors, decorators, or alternatives by using the file.

**Note**

You will find many differences between Quarkus and Java EE, including coding style and the implementation of the specifications. The result is that Java code you find using Quarkus will often have differences to traditional Java.

In most cases, the differences are motivated by Quarkus' goals, such as enabling fast startup, low memory usage, and native compilation. Even though the scope of the course does not cover Quarkus Native, coding style and practices shown will follow the recommendations within the Quarkus community.

Quarkus discovers beans using a simplified bean discovery, having only a single archive that has the bean discovery mode annotated.

Quarkus DI constructs the single bean archive by reading the following classes:

1. Application classes
2. Dependencies with a beans.xml descriptor (where the contents of the descriptor are ignored)
3. Any that have a Jandex index
4. Any referenced by quarkus.index-dependency in the application.properties
5. Any included in Quarkus integration code

Typically, private members are preferred in Java. However, with Quarkus DI, package-private modifiers are preferred because of the possibility of using Quarkus Native.

To achieve compatibility with Quarkus Native, you should not include an explicit access modifier in the following:

- Fields
- Constructors
- Initializers
- Observer methods
- Producer methods
- Injection fields
- Disposers
- Interceptors

Annotation usage is an important difference in how CDI defines bean discovery. Bean classes without a defining annotation are ignored. However, producer methods, fields, and observer methods are discovered, regardless of the class' lack of annotations. Another exception is when a business method is annotated with @Scheduled.

The following is an example of a bean declaration, which will be managed by Quarkus DI.

```
import javax.enterprise.context.ApplicationScoped;
@ApplicationScoped
```

```
public class ExampleBean {
    private String property;

    public void method() {
        System.out.println( "Saying Hi!" );
    }
}
```

Now that the class has been annotated, Quarkus DI will discover the bean. You can then use the bean by injecting it into other classes. An example of a property injection looks as follows:

```
import javax.inject.Inject;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class SomeResource {
    @Inject
    ExampleBean someProperty;

    public void useBean() {
        someProperty.method();
    }
}
```



Note

Quarkus DI is implemented by Quarkus Arc. Quarkus Arc configuration is controlled by entries that use the `quarkus.arc` prefix in the `application.properties` file.

Reviewing the JAX-RS Specification

JAX-RS, formerly known as Java API for RESTful Web Services, is a specification that provides support for creating web services following the *Representational State Transfer* (REST) architectural pattern. The JAX-RS API is primarily annotation-based and facilitates the development of web service endpoints and clients in Java by simplifying and standardizing much of the boilerplate code required to build a web service. This approach allows developers to focus their efforts entirely on the business logic of their application. Starting with version 1.1, the Java EE 6 specification started including JAX-RS, making it the industry-standard API for building Java-based REST services.

The following table includes descriptions of the core annotations that the JAX-RS specification defines for building REST service classes. These annotations are all used at either a class or method level. They define how the service interacts with clients, and allows the application server to map the incoming HTTP requests to the appropriate method. You can find all of these annotations in the `javax.ws.rs` package and its subpackages.

JAX-RS Class and Method-Level Annotations Summary

Annotation	Description
<code>@Path</code>	The relative path for a class or method.

Annotation	Description
@GET, @PUT, @POST, @DELETE, @HEAD	The HTTP request type of a method.
@Produces, @Consumes	The Internet media types for the content consumed as the request parameters or produced as the response. The <code>javax.ws.rs.core.MediaType</code> class specifies the allowed values.

Additionally, JAX-RS provides a set of annotations that you apply at the method parameter level and use to retrieve information directly from the HTTP request. These annotations, which are also found in the `javax.ws.rs` package and its subpackages, are summarized in the following table.

JAX-RS Method Parameter Level Annotations Summary

Annotation	Description
@PathParam	Binds to a segment of the URI. It is possible to include a placeholder with a matching name in the @Path value. An example URI that includes a path parameter is: <code>http://www.example.com/currencies/USD</code>
@QueryParam	Binds to an HTTP query parameter using its name. An example URI that includes a query parameter is: <code>http://www.example.com/rest?name=Test</code>
@HeaderParam	Binds to a header in the HTTP request using its name.
@FormParam	Binds to a form field using its name.
@Context	Returns the entire context of the <code>HttpServletRequest</code> object or the <code>SecurityContext</code> object for the incoming HTTP request. This annotation can also be used to inject class-level variables instead of method parameters. This can also retrieve the <code>UriInfo</code> for incoming requests.

Using JAX-RS in Quarkus

You can use JAX-RS in Quarkus by simply creating resources with the `@Path` annotation. The default Quarkus project already includes the most basic implementation, from which you can build upon. The following is a simplified example of a JAX-RS service.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.GET;

@Path("/example")❶
@Consumes(MediaType.APPLICATION_JSON)❷
@Produces(MediaType.APPLICATION_JSON)❸
public class ExampleResource {
    @GET❹
    public String greet() {
        return "Hello friend!";
    }
}
```

```

    }

    @GET
    @Path("/{name}")5
    public String greet(@PathParam("name") String name) {
        return "Hello " + name + "!";
    }
}

```

- ❶ Specifies the base path "/example" in the URL, which will be used to route the requests to this service.
- ❷ Indicates that the services will read the request's body by deserializing JSON. This will make the service accept requests with the header Content-Type and value application/json.
- ❸ Automates JSON serialization for you. Your services can return objects, which will be analyzed and automatically generate the JSON output. Sets the value of the Content-Type header on the response to application/json.
- ❹ Routes the HTTP GET requests to the annotated method.
- ❺ Another HTTP GET method that includes the name parameter in its path.

Understanding JSON Serialization and Deserialization in Quarkus

Traditional Java EE was dependent on JSON-P for many types of JSON manipulation. Following the JAX-RS 2.1 specification, Quarkus uses JSON-B. The JSON-P specification defines an API for processing JSON streams, transforming them to an object model and attaching events to it. The JSON-B specification declares a binding layer from the JSON structure to Java POJOs, mapping JSON values to attributes of Java object instances.



Note

Quarkus also supports using Jackson for JSON serialization, deserialization and binding. Refer to the `quarkus-resteasy-jackson` or `quarkus-jackson` extensions for usage details.

Binding a class for serialization and deserialization requires two steps:

1. Annotate your endpoints methods with JAX-RS annotations, such as `@GET`, `@POST`, `@PUT`, or `@DELETE`.
2. Annotate the service class or the endpoint method with `@Consumes(MediaType.APPLICATION_JSON)` for deserializing input parameters, or with `@Produces(MediaType.APPLICATION_JSON)` to serialize the results.



Note

JAX-RS also supports methods returning a `Response` object, instead of a POJO. When running in native mode, Quarkus is unable to identify the type of objects to serialize. In this case, use the `@RegisterForReflection` annotation to indicate GraalVM to index the POJO and enable serialization. This is only relevant when using Quarkus Native.

Quarkus JSON binding capabilities are based on code introspection. This means Quarkus must be able to instantiate the class on runtime and assign the values for its fields. To do so, Quarkus requires the POJO classes to follow two rules:

- Class constructors must be identifiable, by one of the following methods:
 - The POJO class must include a no-parameters constructor.
 - Annotate the POJO constructor with `@JsonbCreator`. In this case, parameter names must match JSON property names, or you must use `@JsonProperty` to indicate the matching property.
- Class attributes must be publicly available for reading and writing by either:
 - Declare class attributes with `public` access modifiers.
 - Include appropriate getter and setter methods for each attribute.

There are situations when you do not want to serialize a property. For example, if your POJO represents a user but you do not want to return private information, such as their password's hash. In this case, use the `@JsonbTransient` annotation on the properties that you want to avoid serializing.

If you only want to ignore the property for serialization, then use the `@JsonbTransient` annotation on the getter method. Similarly, if you only want to ignore the property for deserialization, then use the annotation on the setter.

You can achieve more advanced custom serialization and deserialization by defining a CDI bean of type `io.quarkus.jsonb.JsonbConfigCustomizer`, or a bean of type `javax.json.bind.JsonbConfig`.



Note

The annotations mentioned above are JSON-B specific. Jackson uses the `@JsonIgnore` or `@JsonIgnoreProperties` annotations to define ignored properties. Jackson can also fine-control serialization by defining a CDI bean of type `io.quarkus.jackson.ObjectMapperCustomizer`.

Using the Basics Together

Start with declaring the `ExampleData` class for the data you want to use in your RESTful service.

```
public class ExampleData {  
    public String name;  
    public int age;  
}
```

Create an `ExampleService` class, which you can define as a bean by using `@ApplicationScoped`. Having this annotation means that you can inject it into other classes that are managed by Quarkus DI.

```
@ApplicationScoped  
public class ExampleService {  
    private List<ExampleData> list = new ArrayList<ExampleData>();
```

```
public List<ExampleData> create(ExampleData data) {  
    list.add(data);  
    return list;  
}  
}
```

The final piece for this implementation is the `ExampleResource` class, which is an endpoint that receives and returns JSON data using the JAX-RS specification. The `ExampleService` object is injected into the class, providing the implementation of the service.

```
@Path("/example")  
@Consumes(MediaType.APPLICATION_JSON)  
@Produces(MediaType.APPLICATION_JSON)  
public class ExampleResource {  
    @Inject  
    ExampleService service;  
  
    @POST  
    public List<ExampleData> create(ExampleData data) {  
        return service.create(data);  
    }  
}
```

Using VSCodium for Quarkus Development

You can develop Quarkus services using your preferred IDE, such as Eclipse, IntelliJ, Apache NetBeans, or Visual Studio Code. Many IDEs have plug-ins and extensions that can improve your development experience.

In this course, coding instructions and screenshots use VSCodium, a free and open source IDE based on Visual Studio Code. The main difference between Visual Studio Code and VSCodium is that the former includes proprietary extensions. Additionally, VSCodium prevents functionality that tracks usage data from Visual Studio Code by getting rid of telemetry.

VSCodium is installed and configured to work with Quarkus in the `workstation` virtual machine in the lab environment.

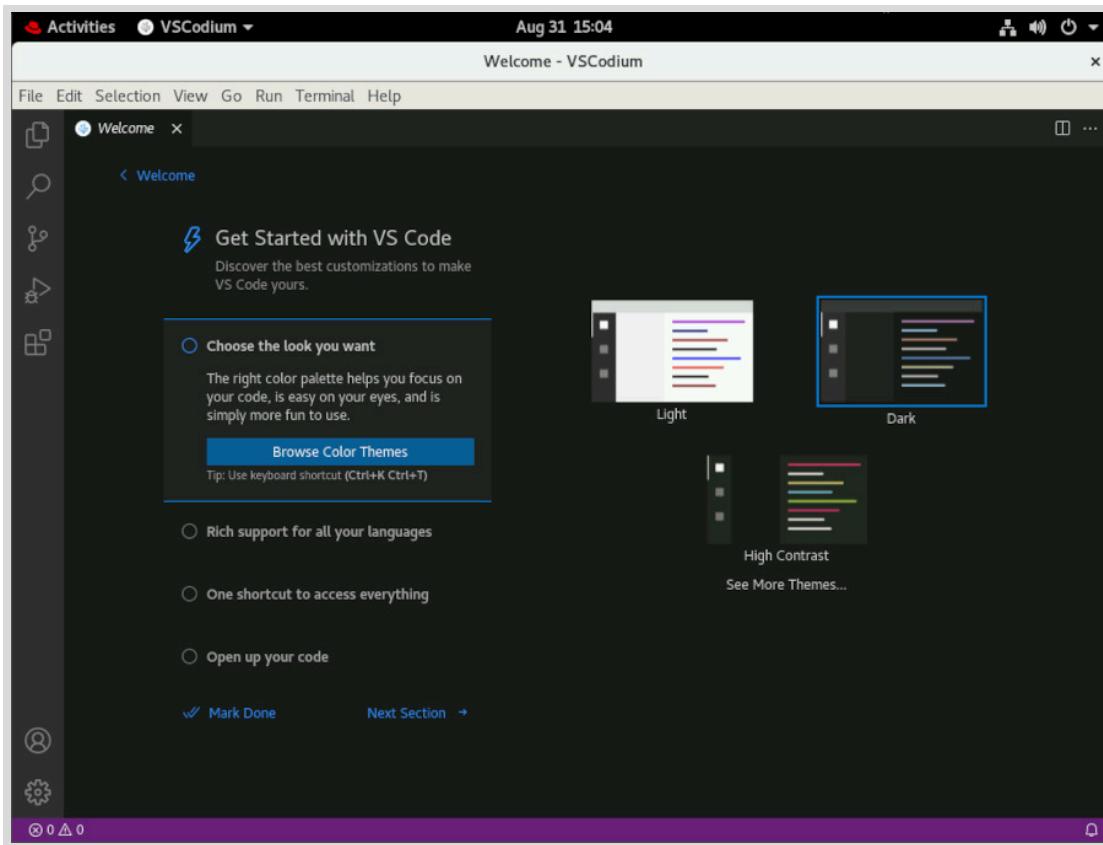


Figure 2.1: VSCodium on first open

To see the installed extensions, click the **Extensions** button on the left menu, fifth from the top. You will find many extensions that improve Java development, including the Quarkus extension, which expands the IDE to work better with Quarkus.

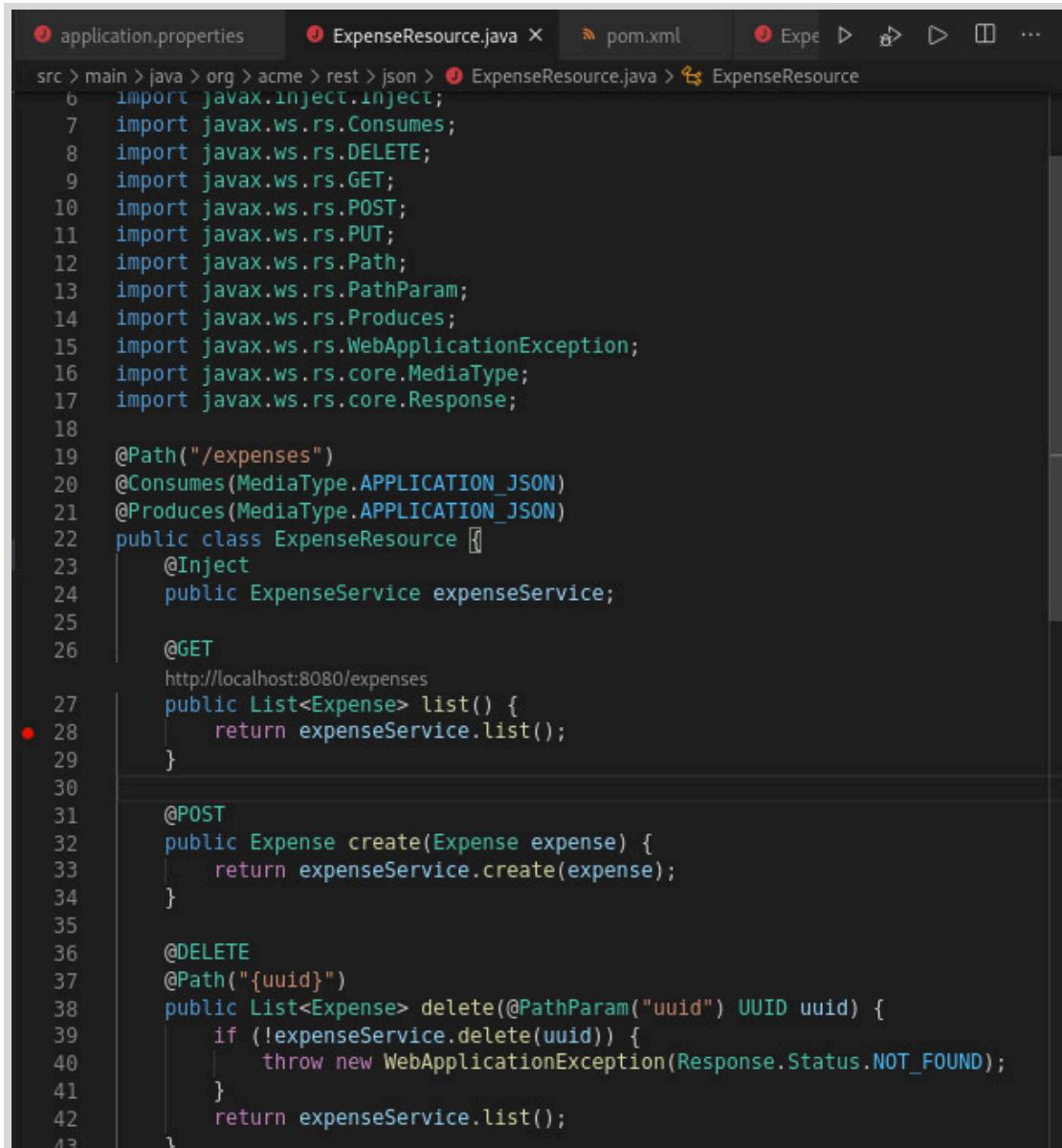
Running and Debugging Your Application

You can run and debug your Quarkus application from within VSCodium because the Quarkus extension integrates with Maven. You can open a project by clicking **File > Open Folder...** and selecting a folder with a `pom.xml` file. Wait for the extension to load the project dependencies.

You can now run and debug the application by opening the **Command Palette**. Press **Ctrl+Shift+P** or click **View > Command Palette**.

Type `quarkus` to see related commands available in your project. Select **Quarkus: Debug current Quarkus project** to start the Quarkus application in development mode and connect the IDE to a debugging session, which enables common debugging features.

You can add breakpoints by clicking on the red dot that appears when you hover to the left of the line number where you want the code to stop. This will add the red dot as a breakpoint indicator.



The screenshot shows the VSCode interface with the ExpenseResource.java file open. The code defines a REST resource for managing expenses. A red dot, indicating a breakpoint, is placed on line 28, which contains the method signature for listing expenses. The code uses annotations like @Path, @GET, @POST, and @DELETE to map HTTP methods to specific endpoints. It also uses Jersey annotations like @Consumes and @Produces to handle JSON data. The expenseService dependency is injected via @Inject.

```

1 application.properties
2 ExpenseResource.java X pom.xml
3 src > main > java > org > acme > rest > json > ExpenseResource.java > ExpenseResource
4   import javax.inject.Inject;
5   import javax.ws.rs.Consumes;
6   import javax.ws.rs.DELETE;
7   import javax.ws.rs.GET;
8   import javax.ws.rs.POST;
9   import javax.ws.rs.PUT;
10  import javax.ws.rs.Path;
11  import javax.ws.rs.PathParam;
12  import javax.ws.rs.Produces;
13  import javax.ws.rs.WebApplicationException;
14  import javax.ws.rs.core.MediaType;
15  import javax.ws.rs.core.Response;
16
17  @Path("/expenses")
18  @Consumes(MediaType.APPLICATION_JSON)
19  @Produces(MediaType.APPLICATION_JSON)
20  public class ExpenseResource {
21      @Inject
22      public ExpenseService expenseService;
23
24      @GET
25      @Path("expenses")
26      @Produces(MediaType.APPLICATION_JSON)
27      public List<Expense> list() {
28          return expenseService.list();
29      }
30
31      @POST
32      public Expense create(Expense expense) {
33          return expenseService.create(expense);
34      }
35
36      @DELETE
37      @Path("{uuid}")
38      public List<Expense> delete(@PathParam("uuid") UUID uuid) {
39          if (!expenseService.delete(uuid)) {
40              throw new WebApplicationException(Response.Status.NOT_FOUND);
41          }
42          return expenseService.list();
43      }
}

```

Figure 2.2: VSCodium with a breakpoint added to line 28

While running your application, if you send a request to your endpoint, you can see that the code will stop executing at the breakpoint, allowing you to verify the state of the variables. VSCodium will provide you with common debugging actions, such as step into, step over, and continue. The current step of code execution, in this case the breakpoint line, will be marked with a yellow background.

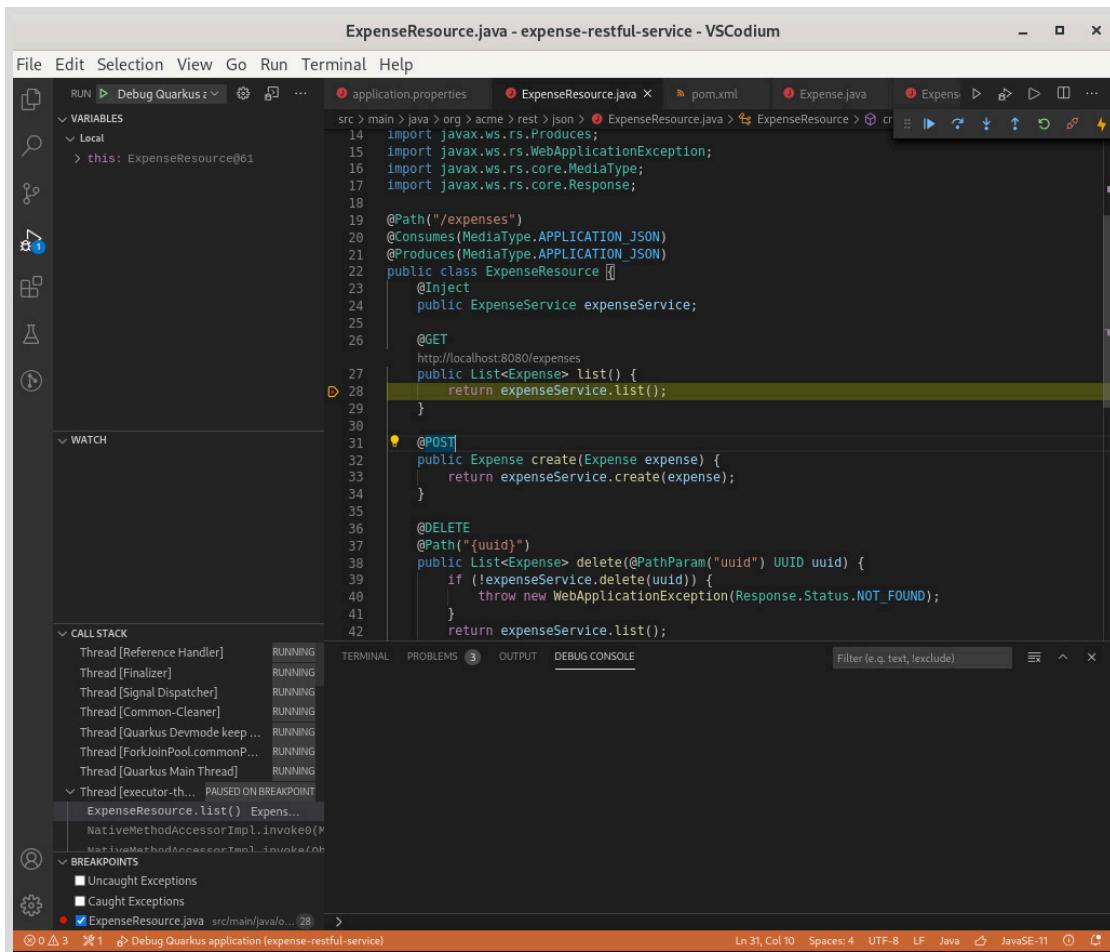


Figure 2.3: VSCode while running a debugging session



References

CDI 2.0 specification

<https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html>

JAX-RS 2.1 specification

<https://jcp.org/en/jsr/detail?id=370>

JSON Binding 1.0 Users Guide

<https://javaee.github.io/jsonb-spec/users-guide.html>

VSCode

<https://vscode.com/>

► Guided Exercise

Implementing a RESTful Microservice

In this exercise, you will develop a RESTful microservice using JAX-RS, Quarkus DI, and JSON serialization.

You will work with the `expense-restful-service` project to implement a Quarkus service with basic functionality. To achieve this, you will perform the following tasks:

1. Modify the `Expense` class to add a method that creates `Expense` objects from JSON data.
2. Prepare the `ExpenseService` class to make it a bean, thus enabling Quarkus DI to discover it and use it for injection.
3. Implement the endpoints in the `ExpenseResource` class, injecting the `ExpenseService` and adding GET, POST, PUT and DELETE methods using JAX-RS annotations.

Outcomes

You should be able to interact with the web service endpoints using cURL or the Swagger UI.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command will fetch the directory for the maven project with the code inside, which you will use during the activity.

```
[student@workstation ~]$ lab implement-rest start
```

Instructions

- 1. Open the `expense-restful-service` project.
 - 1.1. Open the VSCode application on your workstation.
 - 1.2. Click **File > Open Folder** and browse to the `/home/student/D0378/labs/implement-rest/expense-restful-service/` directory.
 - 1.3. Click the **OK** button.



Note

If prompted about trusting the authors of the selected folder, select **Yes, I trust the authors**.

- 2. Add to the `Expense` class a method to allow creating `Expense` objects from JSON data.

- 2.1. Navigate to the `src/main/java/org/acme/rest/json/Expense.java` file and open it.

- 2.2. Add the following import.

```
import javax.json.bind.annotation.JsonbCreator;
```

- 2.3. Add the `of()` method with the `JsonbCreator` annotation, which will tell JSON-B to use it when deserializing an `Expense` object that has `name`, `paymentMethod`, and `amount` properties. The constructor used by the method is already implemented.

```
...output omitted...
```

```
@JsonbCreator  
public static Expense of(String name, PaymentMethod paymentMethod, String  
amount) {  
    return new Expense(name, paymentMethod, amount);  
}  
...output omitted...
```

► 3. Make the `ExpenseService` class a CDI bean.

- 3.1. Navigate to the `src/main/java/org/acme/rest/json/ExpenseService.java` file and open it.

- 3.2. Add the following import.

```
import javax.enterprise.context.ApplicationScoped;
```

- 3.3. Use the `@ApplicationScoped` annotation on the `ExpenseService` class.

```
...output omitted...
```

```
@ApplicationScoped  
public class ExpenseService {  
...output omitted...
```

► 4. Implement the RESTful endpoints.

- 4.1. Navigate to the `src/main/java/org/acme/rest/json/ExpenseResource.java` file and open it. Start editing the file by adding the following imports.

```
import javax.ws.rs.Path;  
import javax.ws.rs.Consumes;  
import javax.ws.rs.Produces;  
import javax.ws.rs.core.MediaType;  
...output omitted...
```

- 4.2. Add the imported annotations to the `ExpenseResource` class, which will setup the endpoint to work with JSON. The `@Path` annotation will allow Quarkus to route requests that start the path with `/expenses`.

```
...output omitted...
@Path("/expenses")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class ExpenseResource {
...output omitted...
```

- 4.3. Add the following imports. The `Inject` annotation allows you to inject the `ExpenseService` already setup for CDI use above. You will use the rest of the imports for setting up your endpoints, as well as for handling an error.

```
import javax.inject.Inject;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.PathParam;
...output omitted...
```

- 4.4. Write the following changes to the `ExpenseResource` class to inject the `ExpenseService` created above and make it serve RESTful requests.

```
...output omitted...
public class ExpenseResource {
    @Inject①
    public ExpenseService expenseService;

    @GET②
    public Set<Expense> list() {
        return expenseService.list();
    }

    @POST③
    public Expense create(Expense expense) {
        return expenseService.create(expense);
    }

    @DELETE④
    @Path("{uuid}")⑤
    public Set<Expense> delete(@PathParam("uuid") UUID uuid) {⑥
        if (!expenseService.delete(uuid)) {
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        return expenseService.list();
    }

    @PUT⑦
    public void update(Expense expense) {
        expenseService.update(expense);
    }
...output omitted...
```

- ➊ The `@Inject` annotation tells Quarkus DI to find an available `ExpenseService` and inject it as the property.
 - ➋ Annotates the method to respond to requests with the GET HTTP method to retrieve expenses.
 - ➌ Annotates the method to respond to POST and PUT HTTP methods to create and update expenses.
 - ➍ Annotates the method to respond to the DELETE HTTP method to delete expenses.
 - ➎ Annotates the method to respond to a path that contains the UUID as a path parameter.
 - ➏ Annotates the method parameter to map to the value from the URL path.
- ▶ 5. Explore the microservice using Swagger UI.
- 5.1. Make sure to save the files.
 - 5.2. Click **View > Command Palette**, write Quarkus, select **Quarkus: Debug current Quarkus project** and make sure you do not get errors reported by VSCode. If you get errors, then abort the task, inspect the **Problems** tab in the bottom panel and fix them.
 - 5.3. Open up a browser and go to the Swagger UI endpoint of the application URL: <http://localhost:8080/swagger-ui>.

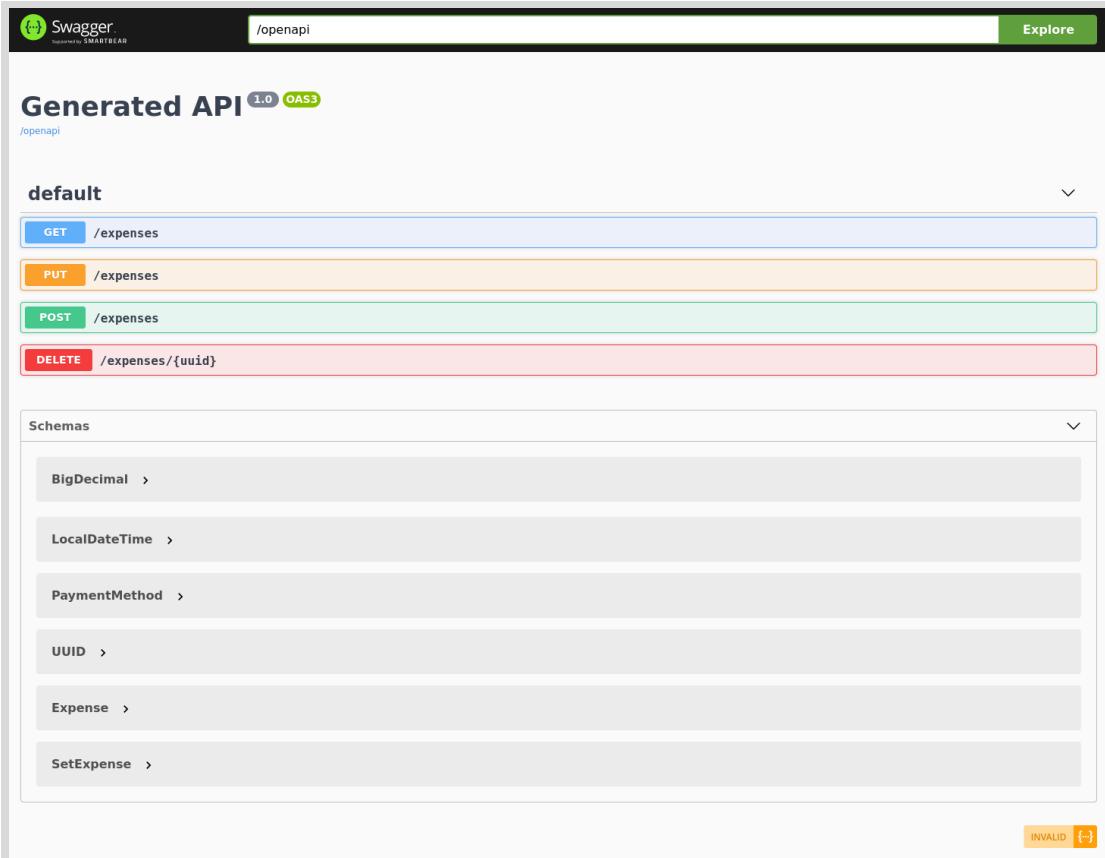


Figure 2.4: First view in the Swagger UI

- 5.4. Click **GET /expenses**. You will see the information for using the microservice, including example values.
- 5.5. Click **Try it out**, and then **Execute**. You will see the response body, which is an empty JSON list, because there is no data yet.
- ▶ 6. Use CDI's `@PostConstruct` annotation on the `ExpenseService` class to set some sample data when creating the service. This will populate the `expenses` set with some default values.

- 6.1. Go back to VSCode and open the `ExpenseService.java` file and add the following imports.

```
import javax.annotation.PostConstruct;
import org.acme.rest.json.Expense.PaymentMethod;
```

- 6.2. Write the following code to put some default data in the `expenses` object. Please note the `initData()` method could be named differently.

```
@PostConstruct
void initData() {
    expenses.add(new Expense("Groceries", PaymentMethod.CASH, "150.50"));
    expenses.add(new Expense("Civilization VI", PaymentMethod.DEBIT_CARD,
    "25.00"));
}
```

- 6.3. Save the file. Go back to the Swagger UI in your browser and click **Execute** again. You will see it populated the data, without the need to recompile your project manually.
- 7. Create a new Expense using the microservice.
- 7.1. Click **POST /expenses**.
 - 7.2. Click **Try it out**. Note that, because we added the `of()` method to `Expense`, we can skip the parameters `uuid` and `creationData` mentioned in the **Request Body** given to us by Swagger.
 - 7.3. Change the **Request Body** to have the following JSON code.
- ```
{
 "amount": 10,
 "name": "Foo",
 "paymentMethod": "CASH"
}
```
- 7.4. Click **Execute**. Inspect the **Response body**, **Code**, and **Response headers**.
  - 7.5. Click again **GET /expenses**.
  - 7.6. Click **Execute**.
  - 7.7. Inspect the **Response body** to make sure it contains the new expense.
- 8. Go back to VSCode and click **Run > Stop Debugging**.
- 9. A prompt asking you **Would you like to terminate the quarkus:dev task?** might appear. Click **Terminate task** and, if preferred, click **Remember my choice** after.
- 10. Close VSCode.

## Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab implement-rest finish
```

This concludes the guided exercise.

# Persisting Data with Panache

## Objectives

After completing this section, you should be able to persist data with simplified Panache Entities.

## Persistence in Quarkus

A common requirement when developing any kind of application is persisting data to a database. Quarkus uses the Jakarta Persistence API (JPA) as the standard API for data persistence, the default implementation is Hibernate ORM (Object-Relational Mapping). Quarkus brings a new complementary JPA layer, called Panache, which eases writing and using persistent entities.

## Configuring Persistence to an Application

To add persistence to your application you must add either the default Hibernate ORM extension, for enabling the standard JPA implementation, or you can enable the Panache extension, which provides a different approach to working with databases.

```
<dependencies>
 <!-- Hibernate ORM dependency -->
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-hibernate-orm</artifactId>
 </dependency>
</dependencies>
```

Or.

```
<dependencies>
 <!-- Panache dependency -->
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-hibernate-orm-panache</artifactId>
 </dependency>
</dependencies>
```

Additionally, you must add the driver dependency for your database. For example, if you wanted to enable connections to a PostgreSQL database.

```
<dependencies>
 <!-- JDBC driver dependency -->
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-jdbc-postgresql</artifactId>
 </dependency>
</dependencies>
```

For simplicity, Quarkus tries to concentrate all of its configuration in the same file: `application.properties`, therefore the file `persistence.xml` is completely optional. To configure a database connection to a PostgreSQL server, add the following to your `application.properties`.

```
datasource configuration
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = myusername
quarkus.datasource.password = mypassword
quarkus.datasource.jdbc.url = jdbc:postgresql://localhost:5432/mydatabase
```

The value of property `quarkus.datasource.jdbc.url` is dependent on the database provider. As an example of differences, if you wanted to connect to an SQLite database, then the value would look similar to `jdbc:sqlite:mydatabase.db`, where `mydatabase.db` is the SQLite database file.

## Implementing Persistence with Hibernate ORM

You can use the standard JPA implementation, the Hibernate ORM persistence layer, after adding the corresponding extension. Just inject the `EntityManager` and use it to persist and search for entities.

```
@ApplicationScoped
public class EmployeeService {
 @Inject
 EntityManager em;

 @Transactional
 public void createEmployee(String name) {
 Employee employee = new Employee();
 employee.setName(name);
 em.persist(employee);
 }
}
```

The drawback of the `EntityManager` approach is that it is not type safe, thus allowing mistakes to happen in the persistence code. For example, you could pass any kind of entity to the manager to persist it, and it would save it, so if you made the mistake of saving the wrong entity, then you would not find out during compilation.

## Keeping Persistence Simple with Panache

As we saw earlier, Quarkus offers a complementary layer to Hibernate ORM called Panache. Panache is designed to ease and speed up the development of persistent services. To achieve this, Panache provides the most common methods to access and filter data.

You can use Panache in the two most common data access patterns.

### Repository Pattern

The Repository Pattern is the way traditional Java applications have accessed data persistence layers. The purpose of this pattern is to separate the logic of data retrieval and its mapping to entities. To use it simply create a repository that extends from `PanacheRepository` for your entity and inject it where needed.

```
@ApplicationScoped
public class ExpenseRepository implements PanacheRepository<Expense> {
}
```

```
@Inject
ExpenseRepository expenseRepository;
```

Use the injected repository to persist an entity.

```
@Entity
public class Expense {
 private String name;
 private BigDecimal amount;
 private String description;
 private LocalDateTime creationDate;

 // Getters and Setters
}

Expense expense = new Expense();
expense.setName("Hotel stay");
expense.setAmount(100);
expense.setDescription("Conference travel");
expense.setCreationDate(LocalDateTime.now());

expenseRepository.persist(expense);
```



### Note

The current recommendation is to use Panache with the Active Record pattern. This does not make the repository pattern void of useful cases, especially when you want your business logic to be agnostic of its data sources.

### Active Record Pattern

The Active Record pattern defines the methods that access the entity in the entity itself, thus giving the entity an active role in its persistence. This means there is no longer a need to use external objects to manage the persistent objects, which avoids the requirement of repositories or entity managers and retains flexibility. One of the benefits of this approach is that objects can be used in the same way you would use any other Java object, making development more intuitive, faster, and easier. To use this pattern simply extend your entities from `PanacheEntity` and use the methods directly on the entity instance.

```
@Entity
public class Expense extends PanacheEntity {
 public String name;
 public BigDecimal amount;
 public String description;
 public LocalDateTime creationDate;
}
```

**Note**

Notice that the properties of the entity are public instead of private and accessed with getters and setters, which helps Quarkus to avoid using the Java Reflection API for compiling. During compilation, Quarkus will change public fields and create their setters and getters.

To operate on entities, just use the methods now present in the `Expense` class, inherited from `PanacheEntity`.

```
Expense expense = new Expense();
expense.name = "Hotel stay";
expense.amount = 100;
expense.description "Conference travel";
expense.creationDate = LocalDateTime.now();

expense.persist();
```

When using the Active Record pattern, you can add custom methods to the entity class to make them available for all instances. For example, you can create a method that only returns the expenses created today, with a method like this.

```
@Entity
public class Expense extends PanacheEntity {
 public String name;
 public BigDecimal amount;
 public String description;
 public LocalDateTime creationDate;

 public static List<Expense> findCurrent(){
 return list("creationDate", LocalDatetime.now());
 }
}
```

Both patterns share the same methods provided by Panache. Here is a non-comprehensive list of the methods provided by Panache and an example of how to use them.

```
// Persist the entity
instance.persist();

// check if the entity is persistent
instance.isPersistent()

// get a list of all entities
List<Entity> allEntitys = Entity.listAll();

// find a specific entity by ID
instance = Entity.findById(entityId);

// find a specific instance by ID via an Optional
Optional<Entity> optional = Entity.findByIdOptional(entityId);
```

```
// find all alive entities
List<Entity> aliveInstances = Entity.list("alive", true);

// count all entities
long countAll = Entity.count();

// count all alive entities
long countAlive = Entity.count("alive", true);

// delete all alive entities
Entity.delete("alive", true);

// delete all entities
Entity.deleteAll();

// delete by id
boolean deleted = Entity.deleteById(entityId);

// set all alive entities as not alive
Entity.update("alive = false where alive = ?1", true);
```



## References

### Jakarta Persistence

[https://en.wikipedia.org/wiki/Jakarta\\_Persistence](https://en.wikipedia.org/wiki/Jakarta_Persistence)

## ► Guided Exercise

# Persisting Data with Panache

In this exercise, you will learn how to persist data with Quarkus Panache using the Active Record pattern.

## Outcomes

You should be able to persist entities using Panache in a Quarkus microservice.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the starting application where we are going to add persistence is present. The code of the application is located in the `/home/student/D0378/labs/implement-persist/expense-service` folder.

```
[student@workstation ~]$ lab implement-persist start
```

## Instructions

- 1. First use the VSCode IDE to edit the Expense Service application.

- 1.1. Open the IDE, passing the application directory as parameter to load the folder.

```
[student@workstation ~]$ codium ~/D0378/labs/implement-persist/expense-service
```

- 2. Add Panache to the dependencies and configuration.

- 2.1. To add the dependencies, bring up the Command Palette in the editor using `Ctrl+Shift+P` or click `View > Command Palette` and type Quarkus. Select the option `Quarkus: add extension to current project`.



### Note

Currently, the VSCode Quarkus extension contains a bug that makes adding extensions to the project available only once the IDE has detected and solved the project dependencies. Two spinning arrows on the bottom right of your screen mean the IDE is still solving dependencies. If the `Quarkus: add extension to current project` option does not show in your command palette then wait for the spinning arrow to become a thumbs-up icon and try again. If the issue persists then restart your VSCode instance and wait until dependencies are solved.

In the modal window type `panache`, select the extension `Hibernate ORM with Panache` and press `Enter`. The `TERMINAL` will show the build for the extension installation. After installing the first extension, bring up the Command Palette again,

select the option Quarkus: add extension to current project, type h2, and select the extension JDBC Driver - H2.

To start the installation, press Enter after the selection. Again, the TERMINAL will show the build for the extension installation.

- 2.2. Open the `src/main/resources/application.properties` file using `Ctrl+P`. Add the following line to the file and save it.

```
quarkus.datasource.db-kind=h2 ①
quarkus.datasource.jdbc.url = jdbc:h2:mem:default ②
quarkus.datasource.username = admin ③
quarkus.hibernate-orm.database.generation = drop-and-create ④
```

- ① Datasource type
- ② JDBC URL of the in memory H2 database.
- ③ User to log on to the database.
- ④ Delete and create the database schema each time the application is loaded, including Development Mode live reload.

- 3. Add Panache persistence to the `Expense` entity in the `/src/main/java/org/acme/rest/json/Expense.java` file.

- 3.1. Add the `javax.persistence.Entity` annotation to the class. The class signature will look like this.

```
...output omitted...
import javax.persistence.Entity;
...output omitted...
@Entity
public class Expense {
...output omitted...
```

- 3.2. Make the `Expense` class extend the `io.quarkus.hibernate.orm.panache.PanacheEntity` to add the Panache extended ORM support. The class signature should look like.

```
...output omitted...
import io.quarkus.hibernate.orm.panache.PanacheEntity;
...output omitted...
@Entity
public class Expense extends PanacheEntity {
...output omitted...
```

- 3.3. Add a default constructor to the entity. This is a requirement of all JPA entities. The default constructor is defined like this.

```
...output omitted...
@Entity
public class Expense extends PanacheEntity {
 public Expense() {}
...output omitted...
```

- 3.4. Add an `update()` static method that will take care of updating an entity in the database. An example implementation looks like.

```
import java.util.Optional;
...output omitted...
public static void update(final Expense expense) { ❶
 Optional<Expense> previous = Expense.findByIdOptional(expense.id); ❷

 previous.ifPresentOrElse((update) -> { ❸
 update.uuid = expense.uuid;
 update.name = expense.name;
 update.amount = expense.amount;
 update.paymentMethod = expense.paymentMethod;
 update.persist();
 }, () -> { ❹
 throw new WebApplicationException(Response.Status.NOT_FOUND);
 });
}
...output omitted...
```

- ❶ The method is defined as `static` because its implementation is independent of an actual `Expense` instance.
- ❷ Look for an `Expense` with the given id.
- ❸ Code to execute if the `Expense` is found in the database.
- ❹ Code to execute if no `Expense` is found with the given id.



### Note

You might find some errors in your code stating that some element cannot be resolved to a Type. In most cases this is caused by a missing `import` statement and can be easily fixed by pressing `Ctrl+.` onto the offending word and selection the option to import the appropriate element.

- 4. Modify the implementation of the class `ExpenseResource` in `src/main/java/org/acme/rest/json/ExpenseResource.java` to use the new persistent capabilities of the `Expense` entity.
- 4.1. Remove the now obsolete `ExpenseService` injection from the `Expense` resource.  
Delete the following lines.

```
@Inject
ExpenseService service;
```

- 4.2. Review the `initData` method at the `src/main/java/org/acme/rest/json/ExpenseService.java` file. This method creates mock data to initialize the service. This approach is hard to modify and maintain. Quarkus recommends creating a `import.sql` file in the `resources` folder with the SQL statements needed to initialize the database.

Create the `src/main/resources/import.sql` file by selecting in the `src/main/resources` folder and selecting the **New File** option. Provide the name `import.sql` and add the following content to it.

```
insert into Expense (id, name, paymentmethod, amount)
 values (nextval('hibernate_sequence'), 'Groceries', '0','150.50');
insert into Expense (id, name, paymentmethod, amount)
 values (nextval('hibernate_sequence'), 'Civilization VI', '1','25.00');
```

The file `src/main/java/org/acme/rest/json/ExpenseService.java` is no longer needed. Delete it by right-clicking it in the left explorer window and selecting the **Delete** option.



#### Note

If you are not able to use the right-click menu, open a **TERMINAL** window and enter  
`rm src/main/java/org/acme/rest/json/ExpenseService.java`

- 4.3. Open the file `src/main/java/org/acme/rest/json/ExpenseResource.java` and replace the implementation of the method `list()` with a call to the `Expense.listAll()`. The new implementation must be.

```
@GET
public List<Expense> list() {
 return Expense.listAll();
}
```

- 4.4. Add the `javax.transaction.Transactional` annotation to the methods that modify entities: `create`, `delete`, and `update`. The method signature should look like the following.

```
import javax.transaction.Transactional;
...output omitted...
@POST
@Transactional
public Expense create(final Expense expense) {
...output omitted...
@DELETE
@Transactional
@Path(" {uuid} ")
public List<Expense> delete(@PathParam("uuid") final UUID uuid) {
...output omitted...
@PUT
@Transactional
public void update(final Expense expense) {
...output omitted...
```

- 4.5. Replace the call to the method `ExpenseService.create()` in the `create()` method with the static method `persist()` in the `Expense` entity.
- In the `create()` method, the line.

```
service.create(newExpense);
```

Must be replaced with the following line.

```
newExpense.persist();
```

- 4.6. Replace the call to the method `ExpenseService.delete()` with the `Expense.delete()` method. Use the version of the method that accepts a property name and a value to delete the entities with matching value in the indicated property.
- Replace the implementation of the `ExpenseResource.delete()` with the following code.

```
public List<Expense> delete(@PathParam("uuid") final UUID uuid) {
 long numExpensesDeleted = Expense.delete("uuid", uuid); ①

 if(numExpensesDeleted == 0) { ②
 throw new WebApplicationException(Response.Status.NOT_FOUND);
 }

 return Expense.listAll();
}
```

- ① Delete all `Expense` entities that match the given value in the `uuid` property.
- ② Raise an error if no entities existed with the given `uuid`.

- 4.7. Replace the implementation of the `ExpenseResource.update()` method with a call to the `Expense.update()` method we created in previous steps.

The implementation must match this.

```
@PUT
@Transactional
public void update(final Expense expense) {
 Expense.update(expense);
}
```

- ▶ 5. Launch and test the Expense Service application in development mode.
- 5.1. Make sure all previous changes are saved. Use the `File > Save All` menu entry to save changes in open files.
  - 5.2. Start the application in development mode by bringing up the Command Palette in the editor using `Ctrl+Shift+P` and type `Quarkus`. Select the option `Debug current Quarkus project`.
  - 5.3. Open up a browser and go to the Swagger UI endpoint of the application URL: `http://localhost:8080/swagger-ui`.
  - 5.4. Click the `POST` line and then click the `Try it out` button.

- 5.5. Replace the example Request body with the following JSON code and click Execute.

```
{
 "id": 3,
 "amount": 100,
 "name": "Scrum and XP from the Trenches",
 "paymentMethod": "CASH",
 "uuid": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
}
```

Check that the response from the server matches the 200 http response code.

- 5.6. Hide the POST block and click the GET line to expand its corresponding block.
- 5.7. Click the Try it out button and then click the Execute button.
- 5.8. Verify that the response from the service includes the just stored entity.
- 5.9. Finish the debugging session by pressing Ctrl+C in the Terminal window in the IDE.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab implement-persist finish
```

This concludes the guided exercise.

## ► Lab

# Implementing a Microservice with Quarkus

In this lab, you will develop an implementation of a RESTful microservice named `microservice-speaker`, using JAX-RS, CDI and Panache.

## Outcomes

You should be able to.

- Create service endpoints to different HTTP methods, receive parameters and consume and produce JSON data, using JAX-RS.
- Inject dependencies into classes to improve testability, using CDI.
- Add persistence and create methods using the Active Record pattern, using Panache.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this lab.

This command ensures that the directories and required files for starting and grading the lab are setup.

```
[student@workstation ~]$ lab implement-review start
```

## Instructions

1. Prepare the project from the terminal.
  - 1.1. The grading scripts for this activity work under the assumption that your code will be located in the `~/D0378/labs/implementation-review` directory. Change into that directory.

```
[student@workstation ~]$ cd ~/D0378/labs/implementation-review
```

- 1.2. Create the Red Hat Build of Quarkus project using maven.

```
[student@workstation implementation]$ mvn \
 io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create \
 -DprojectGroupId=org.acme \
 -DprojectArtifactId=microservice-speaker \
 -DplatformGroupId=com.redhat.quarkus \
 -DplatformVersion=1.11.7.Final-redhat-00009 \
 -DclassName="org.acme.conference.speaker.SpeakerResource" \
 -Dpath="/speaker"
```

**Note**

We are using the command line to create a project that uses the Red Hat Build of Quarkus in its version 1.11.7, instead of an upstream version.

- 1.3. Open the project in VSCode.
2. Remove the `SpeakerResourceTest.java` and `NativeSpeakerResourceIT.java` files from the `src/test/java/org/acme/conference/speaker/` directory, because the changes you are going to make will break the tests.
3. Change the `SpeakerResource` class, so that both the requests and responses are deserialized and serialized using `MediaType.APPLICATION_JSON`.
4. Remove the `hello()` method found in the `SpeakerResource` class, you will not be using it for this activity.
5. Add the Quarkus extension for `quarkus-resteasy-jsonb`.
6. Add the Quarkus extensions to the project needed for persistence. Use `Hibernate ORM with Panache`, and `JDBC Driver - H2`. Configure the datasource to use a H2 in-memory database with the username `admin`. Configure `hibernate-orm` to use a database generation strategy with value `drop-and-create`.
7. Copy the file `~/D0378/labs/implement-review/prepared/Speaker.java` to the `~/D0378/labs/implement-review/microservice-speaker/src/main/java/org/acme/conference/speaker/` directory. This file contains a prebuilt class that has no persistence implemented, as well as no logic to query its data.  
Change the `Speaker` class to be persisted into a database using Panache. Make sure the `biography` property is not serialized.
8. Implement a `findByIdUuid()` method for the `Speaker` class. It should try finding a `Speaker` instance using the `uuid` parameter.

**Note**

Hint: You can use the `firstResultOptional()` method inherited from `PanacheEntity`.

9. Implement a `deleteByUuid()` method for the `Speaker` class that receives a `UUID` object as input and returns nothing. It should try deleting a `Speaker` instance using the `uuid` parameter. The method should not raise any error if no instance has the `uuid` parameter as its own `uuid` value.
10. Implement a `search()` method that receives a `String` and returns a `List` of `Speaker`. Use the following HQL as the query: `"upper(nameFirst) like ?1 or upper(nameLast) like ?1"`. This will allow a case insensitive search of the received string for the fields `nameFirst` and `nameLast`.
11. Allow the JSON deserialization of the `Speaker` class using the public constructor with no `uuid` parameter.
12. Create a new file for the `org.acme.conference.speaker.SpeakerService` class.
13. Expose the `SpeakerService` class to Quarkus DI at the application scope.
14. Using the methods created for the `Speaker` class or the ones made available by `PanacheEntity`, implement the methods `listAll()`, `findByIdUuid()`, `search()`, `create()`, and `deleteByUuid()` in the `SpeakerService` class. This is also a good place

to use the `@Transactional` annotation to define transaction boundaries on all methods that create, delete, or update entities.

15. Reopen the `SpeakerResource.java` file. Inject the `SpeakerService` class into a service object.
16. Implement a `listAll()` method that responds to a GET request.
17. Implement a `search()` method that responds to GET requests in a path with value `/search`. The method must receive a query parameter with value `q` that is used for the search. It must return a list of speakers.
18. Implement a `findById()` method that responds to GET requests, receives a UUID from the path, and uses it to find a `Speaker` and return it. If it does not find a matching object, then it must return a 404 status code.
19. Implement a `create()` method that responds to POST requests, receives a `Speaker` object and persists it.
20. Implement a `delete()` method that responds to DELETE requests, which will receive a UUID from the path, and use it to find and delete the `Speaker` object.
21. Start the application in development mode by bringing up the Command Palette in the editor and typing `Quarkus`. Select the option `Debug current Quarkus project`. If you find any errors, then proceed to debug them.
22. Finish the debugging session by pressing `Ctrl+C` in the Terminal window in the IDE.

## Evaluation

Grade your work by running the `lab implement-review grade` command from your workstation machine.

```
[student@workstation ~]$ lab implement-review grade
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab implement-review finish
```

This concludes the lab.

## ► Solution

# Implementing a Microservice with Quarkus

In this lab, you will develop an implementation of a RESTful microservice named `microservice-speaker`, using JAX-RS, CDI and Panache.

## Outcomes

You should be able to.

- Create service endpoints to different HTTP methods, receive parameters and consume and produce JSON data, using JAX-RS.
- Inject dependencies into classes to improve testability, using CDI.
- Add persistence and create methods using the Active Record pattern, using Panache.

## Before You Begin

On the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command ensures that the directories and required files for starting and grading the lab are setup.

```
[student@workstation ~]$ lab implement-review start
```

## Instructions

1. Prepare the project from the terminal.

- 1.1. The grading scripts for this activity work under the assumption that your code will be located in the `~/D0378/labs/implementation-review` directory. Change into that directory.

```
[student@workstation ~]$ cd ~/D0378/labs/implementation-review
```

- 1.2. Create the Red Hat Build of Quarkus project using maven.

```
[student@workstation implementation-review]$ mvn \
 io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create \
 -DprojectGroupId=org.acme \
 -DprojectArtifactId=microservice-speaker \
 -DplatformGroupId=com.redhat.quarkus \
 -DplatformVersion=1.11.7.Final-redhat-00009 \
 -DclassName="org.acme.conference.speaker.SpeakerResource" \
 -Dpath="/speaker"
```

**Note**

We are using the command line to create a project that uses the Red Hat Build of Quarkus in its version 1.11.7, instead of an upstream version.

- 1.3. Open the project in VSCode.
2. Remove the `SpeakerResourceTest.java` and `NativeSpeakerResourceIT.java` files from the `src/test/java/org/acme/conference/speaker/` directory, because the changes you are going to make will break the tests.
3. Change the `SpeakerResource` class, so that both the requests and responses are deserialized and serialized using `MediaType.APPLICATION_JSON`.
  - 3.1. Annotate the `SpeakerResource` class with `Produces` and `Consumes`, both with `MediaType.APPLICATION_JSON` as parameter.

```
...output omitted...
@Path("/speaker")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class SpeakerResource {
 ...output omitted...
```

4. Remove the `hello()` method found in the `SpeakerResource` class, you will not be using it for this activity.
5. Add the Quarkus extension for `quarkus-resteasy-jsonb`.
  - 5.1. Bring up the Command Palette in the editor clicking `View > Command Palette` and type Quarkus. Select the option `Quarkus: add extension to current project`.  
In the modal window type `resteasy json-b`, select the extension `RESTEasy JSON-B` and press Enter.  
Alternatively, you can manually modify the `pom.xml` file located in the root of the project directly, adding the following inside of the `dependencies` element.

```
<dependencies>
 ...output omitted...
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-resteasy-jsonb</artifactId>
 </dependency>
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-jsonb</artifactId>
 </dependency>
</dependencies>
```

6. Add the Quarkus extensions to the project needed for persistence. Use `Hibernate ORM with Panache`, and `JDBC Driver - H2`. Configure the datasource to use a H2 in-memory database with the username `admin`. Configure `hibernate-orm` to use a database generation strategy with value `drop-and-create`.

- 6.1. Bring up the Command Palette in the editor clicking **View > Command Palette** and type Quarkus. Select the option **Quarkus: add extension to current project**.

In the modal window type panache, select the extension **Hibernate ORM with Panache**. After selecting the first extension, type h2 and select the extension **JDBC Driver - H2**.

Once both extensions are selected, press **Enter** to add both extensions to your project.

Alternatively, you can modify the **pom.xml** file located in the root of the project directly, adding the following inside of the **dependencies** element.

```
<dependencies>
 ...
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-jdbc-h2</artifactId>
 </dependency>
 <dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-hibernate-orm-panache</artifactId>
 </dependency>
</dependencies>
```

- 6.2. Open the **src/main/resources/application.properties** file. Add the following content to the file and save it.

```
quarkus.datasource.db-kind=h2
quarkus.datasource.jdbc.url = jdbc:h2:mem:default
quarkus.datasource.username = admin
quarkus.hibernate-orm.database.generation = drop-and-create
```

7. Copy the file **~/D0378/labs/implement-review/prepared/Speaker.java** to the **~/D0378/labs/implement-review/microservice-speaker/src/main/java/org/acme/conference/speaker/** directory. This file contains a prebuilt class that has no persistence implemented, as well as no logic to query its data.

Change the Speaker class to be persisted into a database using Panache. Make sure the **biography** property is not serialized.

- 7.1. In the terminal, run the following command, inside the **~/D0378/labs/implement-review** directory.

```
[student@workstation implement-review]$ cp ./prepared/Speaker.java \
./microservice-speaker/src/main/java/org/acme/conference/speaker/
```

- 7.2. Annotate the Speaker class with **@Entity**.

```
...output omitted...
import javax.persistence.Entity;

@Entity
public class Speaker {
...output omitted...
```

7.3. Change the Speaker class to extend from the PanacheEntity class.

```
...output omitted...
import javax.persistence.Entity;
import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
public class Speaker extends PanacheEntity {
...output omitted...
```

7.4. Annotate the biography property with @JsonbTransient.

```
...output omitted...
import javax.persistence.Entity;
import io.quarkus.hibernate.orm.panache.PanacheEntity;
import javax.json.bind.annotation.JsonbTransient;

public class Speaker extends PanacheEntity {
...output omitted...
@JsonbTransient
public String biography;
...output omitted...
```

8. Implement a `findById()` method for the Speaker class. It should try finding a Speaker instance using the `uuid` parameter.



### Note

Hint: You can use the `firstResultOptional()` method inherited from `PanacheEntity`.

8.1. Add the following code to the content of the Speaker class.

```
...output omitted...
import java.util.Optional;

public class Speaker extends PanacheEntity {
...output omitted...
public static Optional<Speaker> findById(UUID uuid) {
 return find("uuid", uuid).firstResultOptional();
}
...output omitted...
```

9. Implement a `deleteByUuid()` method for the `Speaker` class that receives a `UUID` object as input and returns nothing. It should try deleting a `Speaker` instance using the `uuid` parameter. The method should not raise any error if no instance has the `uuid` parameter as its own `uuid` value.

9.1. Add the following code to the content of the `Speaker` class.

```
...output omitted...

public class Speaker extends PanacheEntity {
...output omitted...
 public static void deleteByUuid(UUID uuid) {
 delete("uuid", uuid);
 }
...output omitted...
```

10. Implement a `search()` method that receives a `String` and returns a `List` of `Speaker`.

Use the following HQL as the query: `"upper(nameFirst) like ?1 or upper(nameLast) like ?1"`. This will allow a case insensitive search of the received string for the fields `nameFirst` and `nameLast`.

10.1. The method inside of the `Speaker` class should look like this.

```
...output omitted...
import java.util.List;
import java.util.Objects;

public class Speaker extends PanacheEntity {
...output omitted...
 public static List<Speaker> search(String query) {
 String queryNonNull = Objects.requireNonNullElse(query, "UNKNOWN");
 String queryUpper = queryNonNull.toUpperCase() + "%";
 return list("upper(nameFirst) like ?1 or upper(nameLast) like ?1",
 queryUpper);
 }
...output omitted...
```

11. Allow the JSON deserialization of the `Speaker` class using the public constructor with no `uuid` parameter.

11.1. Add the `@JsonbCreator` annotation to the constructor of `Speaker` with no `uuid` parameter.

```
...output omitted...
import javax.json.bind.annotation.JsonbCreator;
...output omitted...

public class Speaker extends PanacheEntity {
...output omitted...
 @JsonbCreator
 public Speaker(String nameFirst, String nameLast, String organization,
 String picture, String twitterHandle) {
 this(nameFirst, nameLast, organization, "Empty biography", picture,
```

```
 twitterHandle, UUID.randomUUID());
}
```

```
...output omitted...
```



### Note

This is a good time to check the PROBLEMS tab in VSCode and clear up any reported errors.

12. Create a new file for the `org.acme.conference.speaker.SpeakerService` class.
  - 12.1. On the Explorer found on the left, right-click the `speaker` part of the `java/org/acme/conference/speaker` sequence of folders.
  - 12.2. Click on **New File** and write `SpeakerService.java`. Press enter.
13. Expose the `SpeakerService` class to Quarkus DI at the application scope.
  - 13.1. Use the `@ApplicationScoped` annotation on the `SpeakerService` class.

```
...output omitted...
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class SpeakerService {
...output omitted...
```

14. Using the methods created for the `Speaker` class or the ones made available by `PanacheEntity`, implement the methods `listAll()`, `findById()`, `search()`, `create()`, and `deleteById()` in the `SpeakerService` class. This is also a good place to use the `@Transactional` annotation to define transaction boundaries on all methods that create, delete, or update entities.

- 14.1. Write the following code in the `SpeakerService` class.

```
...output omitted...
import java.util.List;
import java.util.Optional;
import java.util.UUID;
import javax.transaction.Transactional;
...output omitted...

public class SpeakerService {
 public List<Speaker> listAll() {
 return Speaker.listAll();
 }

 public Optional<Speaker> findById(UUID uuid) {
 return Speaker.findById(uuid);
 }

 public List<Speaker> search(String query) {
 return Speaker.search(query);
 }
}
```

```

@Transactional
public Speaker create(Speaker speaker) {
 speaker.persist();
 return speaker;
}

@Transactional
public void deleteByUuid(UUID uuid) {
 Speaker.deleteByUuid(uuid);
}
}

```

15. Reopen the `SpeakerResource.java` file. Inject the `SpeakerService` class into a service object.

- 15.1. Create a property called `service` of class `SpeakerService` inside of the `SpeakerResource` class, annotated with `@Inject`.

```

...output omitted...
import javax.inject.Inject;
...output omitted...
public class SpeakerResource {

 @Inject SpeakerService service;
}

```

16. Implement a `listAll()` method that responds to a GET request.

- 16.1. Write the following code.

```

...output omitted...

public class SpeakerResource {
...output omitted...
 @GET
 public List<Speaker> listAll () {
 return service.listAll();
 }
}

```

17. Implement a `search()` method that responds to GET requests in a path with value / search. The method must receive a query parameter with value `q` that is used for the search. It must return a list of speakers.

- 17.1. Write the following code.

```

...output omitted...
import java.util.List;
import javax.ws.rs.QueryParam;
...output omitted...

public class SpeakerResource {
...output omitted...

```

```

@GET
@Path("/search")
public List<Speaker> search (@QueryParam("q") String query) {
 return service.search(query);
}
...output omitted...

```

18. Implement a `findById()` method that responds to GET requests, receives a UUID from the path, and uses it to find a Speaker and return it. If it does not find a matching object, then it must return a 404 status code.

18.1. Write the following code.

```

...output omitted...
import javax.ws.rs.PathParam;
import javax.ws.rs.NotFoundException;
import java.util.UUID;
...output omitted...

public class SpeakerResource {
...output omitted...
 @GET
 @Path("/{uuid}")
 public Speaker findById(@PathParam("uuid") UUID uuid) {
 return service.findById(uuid).orElseThrow(NotFoundException::new);
 }
...output omitted...

```

19. Implement a `create()` method that responds to POST requests, receives a Speaker object and persists it.

19.1. Write the following code.

```

...output omitted...
import javax.ws.rs.POST;
...output omitted...

public class SpeakerResource {
...output omitted...
 @POST
 public Speaker create(Speaker newSpeaker) {
 service.create(newSpeaker);
 return newSpeaker;
 }
...output omitted...

```

20. Implement a `delete()` method that responds to DELETE requests, which will receive a UUID from the path, and use it to find and delete the Speaker object.

20.1. Write the following code.

```

...output omitted...
import javax.ws.rs.DELETE;
...output omitted...

```

```
public class SpeakerResource {
 ...output omitted...
 @DELETE
 @Path("/{uuid}")
 public void delete(@PathParam("uuid") UUID uuid) {
 service.deleteByUuid(uuid);
 }
 ...output omitted...
```

21. Start the application in development mode by bringing up the Command Palette in the editor and typing Quarkus. Select the option Debug current Quarkus project. If you find any errors, then proceed to debug them.
22. Finish the debugging session by pressing Ctrl+C in the Terminal window in the IDE.

## Evaluation

Grade your work by running the `lab implement-review grade` command from your workstation machine.

```
[student@workstation ~]$ lab implement-review grade
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab implement-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- The MicroProfile specification defines a baseline platform, which optimizes Java for a microservices-based architecture and provides portability of MicroProfile-based applications across multiple runtimes.
- Quarkus is the result of several efforts to address the startup time, performance, and memory footprint required in cloud-native applications, and focuses on several key aspects: container-native, the unification of imperative and reactive paradigms, developer-centric, and use of standard libraries.
- To further enhance its startup and memory footprint target, Quarkus offers the possibility to do AOT compilation using GraalVM.
- MicroProfile offers the building blocks for cloud-native applications borrowed from the Jakarta EE specification:
  - JAX-RS
  - CDI
  - JSON-P
  - JPA
- Quarkus offers Panache, a simplified API to persist entities in databases. Panache provides access to entities using the Active Record Pattern in addition to the Repository Pattern.



## Chapter 3

# Deploying Microservice-based Applications

### Goal

Deploy Quarkus microservice applications to an OpenShift cluster.

### Objectives

- Describe the architecture and features of a microservice application.
- Use Red Hat OpenShift Container Platform to deploy microservices.
- Create container images for Quarkus applications and deploy them in OpenShift Container Platform using Quarkus extensions.

### Sections

- Describing a Microservice Application (and Guided Exercise)
- Deploying Microservices in Red Hat OpenShift Container Platform (and Guided Exercise)
- Deploying Quarkus Applications in Red Hat OpenShift Container Platform (and Guided Exercise)

### Lab

Deploying Microservice-based Applications

# Describing a Microservice Application

---

## Objectives

After completing this section, you should be able to describe the architecture and features of a microservice application.

## Introducing the Conference Application

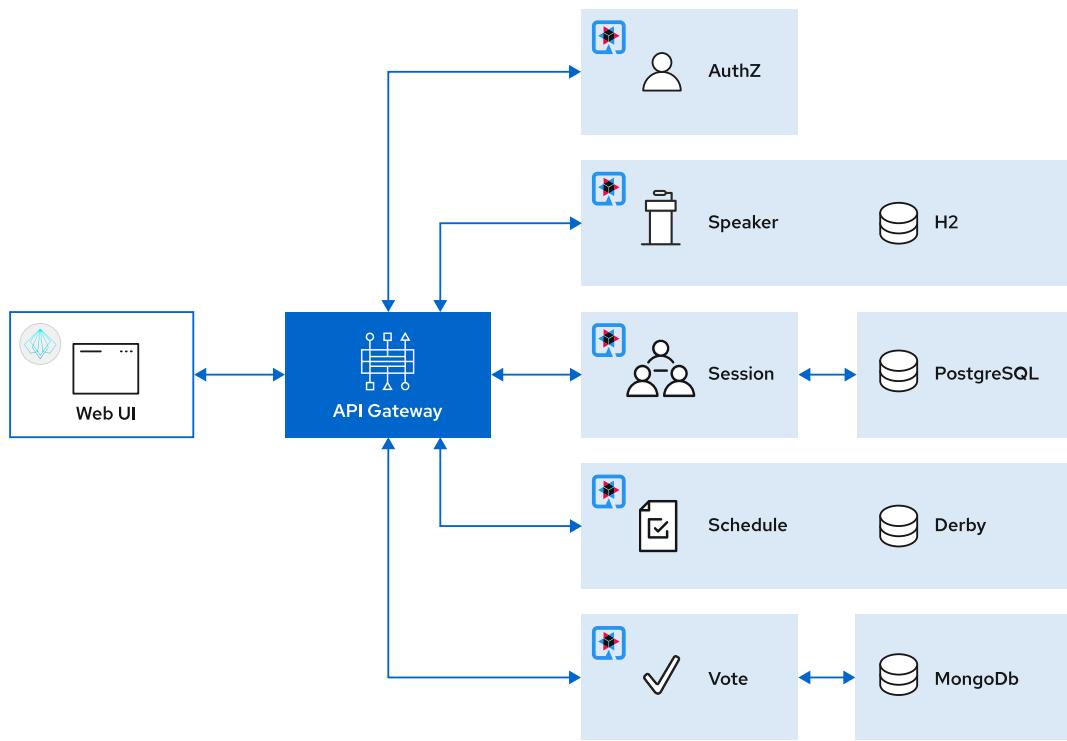
This section presents the Quarkus Conference application, which is the application used for the labs in this course. This application is a variation of the MicroProfile Conference application, provided by the Eclipse MicroProfile project to test and explain several related concepts.

The intent of this application is to handle the requirements for managing a conference. Conferences have the following characteristics.

- A conference is mainly a schedule, which includes a plan relating to the sessions within a conference, speakers that present the sessions, the attendees, and the posterior votes attendees give to the sessions.
- Every entity has a generated identifier. No fields are used as entity identifiers.
- Conferences (or schedules) consist of several sessions. Sessions are presented by many speakers, and attended by many attendees. The same speaker can give many sessions. The same attendee can rate the same or different sessions multiple times.

The Conference application encompasses four main microservices, each one related to each one of the entities described below. For the sake of showing different situations, each microservice takes slightly different design choices. Some microservices embed an in-memory database although some others rely on an external database. Some microservices implement the active record pattern, although others rely on the classic DAO or inject store beans, which manage persistence.

The following diagram shows everything you will be adding to the Conference application throughout this course. Note that you will be creating the application incrementally, so some components will not be initially available.

**Figure 3.1: Components of the conference application**

On the technical side, microservices composing the Conference application are REST services, which communicate through the HTTP protocol. Those Quarkus microservices run on top of a Vert.x server.

## **Listing Microservices That Compose the Conference Application**

As said, the Conference application comprises four microservices.

### **Speaker**

It stores and provides information about speakers, such as name, social network handles, or profile pictures. This microservice stores the speakers' information in a H2 in-memory database, so any service rollout resets the database to its initial state.

### **Schedule**

The Schedule microservice stores information about the timing and location of sessions. It offers some utility endpoints to retrieve active sessions on a given date or all sessions planned in a given venue. This microservice also stores its information in a Derby in-memory database.

### **Session**

This microservice saves information about sessions on all schedules. A session is no more than a link between a schedule and a collection of speakers. Session uses an external PostgreSQL database to store both session and speaker information.

### **Vote**

The Vote microservice includes several entities. On one side, it stores attendees for different sessions. On the other side, it stores the ratings those attendees give to attended sessions.

An external MongoDB database stores both attendees and ratings.

Those are the main microservices that compose the application, but there are three other components used during this course.

- The Web UI service is a PatternFly application used to visualize the whole application.
- The API Gateway component centralizes and enriches communications from clients such as the Web UI. It provides features like centralized authentication or API translation.
- The AuthZ is a mock microservice that generates JWT authentication tokens. It simulates a more complex identity management service.

The following table shows a non-comprehensive list of the endpoints available in different microservices.

#### End Points for the Conference Application Microservices

Service	Endpoint	HTTP Method	Description
Speaker	/	GET	Lists all speakers
	/	POST	Adds a new speaker
	/{uuid}	GET	Retrieves a speaker with a given UUID
	/{uuid}	PUT	Updates a speaker with a given UUID
	/{uuid}	DELETE	Removes a speaker with a given UUID
Session	/	GET	Retrieves all available sessions
	/	POST	Adds a new session
	/{sessionId}	GET	Retrieves a session given its id
	/{sessionId}	PUT	Updates a session given its id
	/{sessionId}	DELETE	Removes a session given its id
	/{sessionId}/speakers	GET	Gets all speakers for a given session
	/{sessionId}/speakers/{speakerId}	PUT	Adds a speaker to a session
	/{sessionId}/speakers/{speakerId}	DELETE	Removes a speaker from a session
Schedule	/all	GET	Retrieves all available schedules
	/{id}	GET	Retrieves a schedule given its id
	/{id}	POST	Adds a new schedule
	/{id}	PUT	Updates a schedule given its id

Service	Endpoint	HTTP Method	Description
	/{{id}}	DELETE	Removes a schedule given its id
	/active/{{dateTime}}	GET	Retrieves all active schedules at a given date and time
Vote	/attendee	GET	Retrieves all attendees for the conference
	/attendee	POST	Adds a new attendee
	/attendee/{{id}}	GET	Retrieves and attendee given its id
	/attendee/{{id}}	PUT	Updates and attendee given its id
	/attendee/{{id}}	DELETE	Removes and attendee given its id
	/rate	GET	Retrieves all ratings for all sessions
	/rate	POST	Adds a new rating for a given session
	/rate/{{id}}	PUT	Updates a rating given its id
	/rate/{{id}}	DELETE	Removes a rating given its id

## Describing Educational Simplifications in the Application

Despite being complete, this Conference application is far from being safe for production purposes. For the sake of simplicity and educational purposes, the application applied several workarounds and unsafe practices. Some examples are.

- Some microservices use an embedded database. Unless you design your application to be transient, you should persist all application data in an external storage. This storage can be an external database service, a shared file system, a distributed cache, or any system that can restore the data after an application shutdown.
- Storing passwords for databases in plain text in the microservices configuration. Never expose your production databases in your services, despite the fact that they are deployed in trusted environments. Use safer methods to store your passwords, such as Kubernetes secrets or, even better, safe storage solutions such as HashiCorp Vault.
- As previously stated, the AuthZ is a mock identity management service. For production applications, use complete identity management solutions such as LDAP, Keycloak, or Red Hat's single sign-on technology.
- Despite all microservices having Cross-origin resource sharing (CORS) filters enabled, those filters are configured to allow requests from any origin. It is a safer practice to limit the allowed origins to trusted ones.
- The exercises will instruct you to enable Swagger UI to expose API documentation and endpoints. Swagger UI is disabled by default when deploying Quarkus applications, because it is hazardous to expose your endpoints directly.
- There is no API gateway application. Front end web UI reaches microservices directly as needed.



## References

**The Quarkus Conference application is based on the MicroProfile Conference application**

<https://github.com/eclipse/microprofile-conference/>

## ► Guided Exercise

# Describing a Microservice Application

In this exercise, you will start all the services that compose the Quarkus Conference application and familiarize yourself with the front-end.

## Outcomes

You should be able to start and use the Quarkus Conference application.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the source code of the services, which are part of the Quarkus Conference application is ready to be deployed.

```
[student@workstation ~]$ lab deploy-microservicesapp start
```

## Instructions

- 1. Start the required databases for the microservices.

- 1.1. Start the PostgreSQL database required by the Session service.

```
[student@workstation ~]$ podman run -d --name postgresql-conference \
-p 5432:5432 \
-e POSTGRESQL_PASSWORD=confi_user \
-e POSTGRESQL_USER=conference_user \
-e POSTGRESQL_ADMIN_PASSWORD=conference \
-e POSTGRESQL_DATABASE=conference \
registry.access.redhat.com/rhscl/postgresql-10-rhel7:1
Trying to pull registry.access.redhat.com/rhscl/postgresql-10-rhel7:1...
...output omitted...
Storing signatures
8ff34d...14c442
```

- 1.2. Start the MongoDB database required for the Vote service.

```
[student@workstation ~]$ podman run -d --name vdb \
-p 27017:27017 quay.io/bitnami/mongodb:4.0
```

**Note**

You can safely ignore any error messages about failing to pull container images from `docker.io`, `registry.redhat.io`, `registry.access.redhat.com`, and `quay.io`. Podman searches for container images in multiple container registries, it shows this error when checking the defined registries one by one and fails to find the images. You can verify that the container image is downloaded and running by running the `podman ps` command.

## ▶ 2. Start the Speaker microservice.

## 2.1. Change to the Speaker microservice directory.

```
[student@workstation ~]$ cd \
~/D0378/labs/deploy-microservicesapp/quarkus-conference/microservice-speaker
```

## 2.2. Start the Speaker microservice using the Development Mode.

```
[student@workstation microservice-speaker]$ mvn quarkus:dev
...output omitted...
Listening for transport dt_socket at address: 5005
...output omitted...
... microservice-speaker 1.0.0-SNAPSHOT ... Listening on: http://0.0.0.0:8082
...output omitted...
```

**Note**

If you are not making use of the live code reload features, you can alternatively package the application and start the generated runner jar file.

```
[student@workstation microservice-speaker]$ mvn package -Dquarkus.profile=dev
...output omitted...
[student@workstation microservice-speaker]$ java -jar \
./target/microservice-speaker-1.0.0-SNAPSHOT-runner.jar
...output omitted...
```

Use the `-Dquarkus.profile=dev` option to ensure the packaged application uses the `dev` profile.

## ▶ 3. Start the Schedule microservice.

## 3.1. In a new terminal window, change to the Schedule microservice directory.

```
[student@workstation ~]$ cd \
~/D0378/labs/deploy-microservicesapp/quarkus-conference/microservice-schedule/
```

## 3.2. Start the Schedule microservice using the Development Mode.

```
[student@workstation microservice-schedule]$ mvn quarkus:dev
...output omitted...
[ERROR] Port 5005 in use, not starting in debug mode
...output omitted...
... microservice-schedule 1.0.0-SNAPSHOT ... Listening on: http://0.0.0.0:8083
...output omitted...
```



### Note

This error is just a notification that there is another Quarkus app using the debug port, and that debugging will not be possible in this run, but the application will run fine.

#### ► 4. Start the Session microservice.

- 4.1. In a new terminal window, change to the Session microservice directory.

```
[student@workstation ~]$ cd \
~/D0378/labs/deploy-microservicesapp/quarkus-conference/microservice-session
```

- 4.2. Start the Session microservice using the Development Mode. To avoid more port conflicts, use the `-Ddebug` option to set a different debug port.

```
[student@workstation microservice-session]$ mvn quarkus:dev -Ddebug=5007
...output omitted...
Listening for transport dt_socket at address: 5007
...output omitted...
... microservice-session 1.0.0-SNAPSHOT ... Listening on: http://0.0.0.0:8084
...output omitted...
```

#### ► 5. Start the Vote microservice.

- 5.1. In a new terminal window, change to the Vote microservice directory.

```
[student@workstation ~]$ cd \
~/D0378/labs/deploy-microservicesapp/quarkus-conference/microservice-vote
```

- 5.2. Start the Vote microservice using the Development Mode.

```
[student@workstation microservice-vote]$ mvn quarkus:dev -Ddebug=5008
...output omitted...
Listening for transport dt_socket at address: 5008
...output omitted...
... microservice-vote 1.0.0-SNAPSHOT ... Listening on: http://0.0.0.0:8081
...output omitted...
```

#### ► 6. Start the front end application using the pregenerated container image.

- 6.1. In a new terminal window, start the container of the front end application.

```
[student@workstation ~]$ podman run -p 8080:8080 \
-d quay.io/redhattraining/quarkus-conference-frontend:latest
Trying to pull quay.io/redhattraining/quarkus-conference-frontend:latest...
...output omitted...
INFO: Accepting connections at http://0.0.0.0:8080
```

► 7. Review the application.

- 7.1. Open a browser and navigate to the URL `http://localhost:8080`. This will open the Quarkus Conference web application.
- 7.2. Click > Schedules to see the list of Schedules. Then click **Schedule 1** to see the first Schedule data.



**Note**

If you do not see the Schedules list, then click the hamburger button on the top left corner of the page (the three horizontal lines button).

In the Schedule info you will find links to the sessions and the speakers that compose this Schedule.

- 7.3. Click > Speakers to show the list of Speakers in the system. Click any of the Speakers to find the data for the given speaker.
- 7.4. Click > Sessions to show the list of sessions available. Click any of the sessions available to find the session's information.
- 7.5. The Vote service does not have a separate UI to access its data. You can interact with it by clicking on one of the stars next to the Session's name, to vote how much you liked the session.



**Note**

The Vote service requires authentication and therefore will not work until authentication is added to the application.

► 8. Clean all the services running in terminals and in the background.

- 8.1. To stop the services running press `Ctrl+C` on each of the terminals you have opened.
- 8.2. Stop all the running containers used in this exercise.

```
[student@workstation ~]$ podman stop -a
```

- 8.3. Remove all containers used in this exercise.

```
[student@workstation ~]$ podman rm -a
```

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab deploy-microservicesapp finish
```

This concludes the guided exercise.

# Deploying Microservices in Red Hat OpenShift Container Platform

## Objectives

After completing this section, you should be able to use Red Hat OpenShift Container Platform to deploy microservices.

## OpenShift Container Platform Architecture

Red Hat OpenShift Container Platform is a self-service platform where development teams can deploy their applications. The platform integrates tools to build and run applications and manages the complete application lifecycle from initial development to production.

OpenShift Container Platform offers several deployment scenarios. One typical workflow starts when a developer provides the Git repository URL for an application to OpenShift Container Platform.

The platform automatically retrieves the source code from Git, and then builds and deploys the application. The developer can also configure OpenShift Container Platform to detect new Git commits, and then automatically rebuild and redeploy the application.

By automating the build and deployment processes, OpenShift Container Platform allows developers to focus on application design and development. By rebuilding your application with every change you commit, OpenShift Container Platform gives you immediate feedback. You can detect and fix errors early in the development process, before they become an issue in production.

OpenShift Container Platform provides the building mechanisms, libraries, and runtime environments for the most popular languages, such as Java, Ruby, Python, PHP, .NET, Node.js, and many more. It also comes with a collection of additional services, which you can directly use for your application, such as databases.

As traffic and load to your web application increases, OpenShift Container Platform can rapidly provision and deploy new instances of the application components. For the Operations team, it provides additional tools for logging and monitoring.

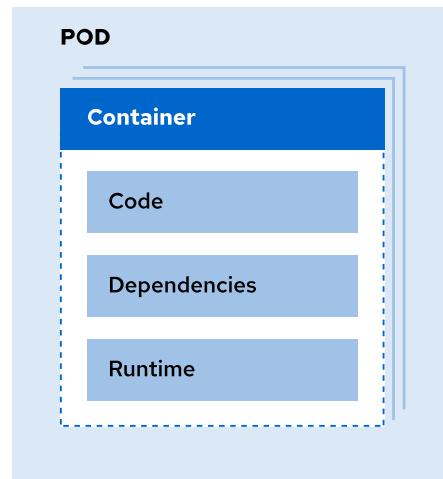
Red Hat OpenShift Online, at <https://www.openshift.com/>, is a public OpenShift Container Platform instance run by Red Hat. With that cloud platform, customers can directly deploy their applications online, without needing to install, manage, and update their own instance of the platform.

Red Hat also provides the Red Hat OpenShift Container Platform, which companies can deploy on their own infrastructure. By deploying your own instance of OpenShift Container Platform, you can fine tune the cluster performance specific to your needs. In this classroom, you will be provided access to a private OpenShift Container Platform cluster.

## Application Architecture

Several development teams or customers usually share the same OpenShift Container Platform instance. For security and isolation between projects and customers, OpenShift Container Platform builds and runs applications in isolated containers.

A container is a way to package an application with all its dependencies, such as runtime environments and libraries.

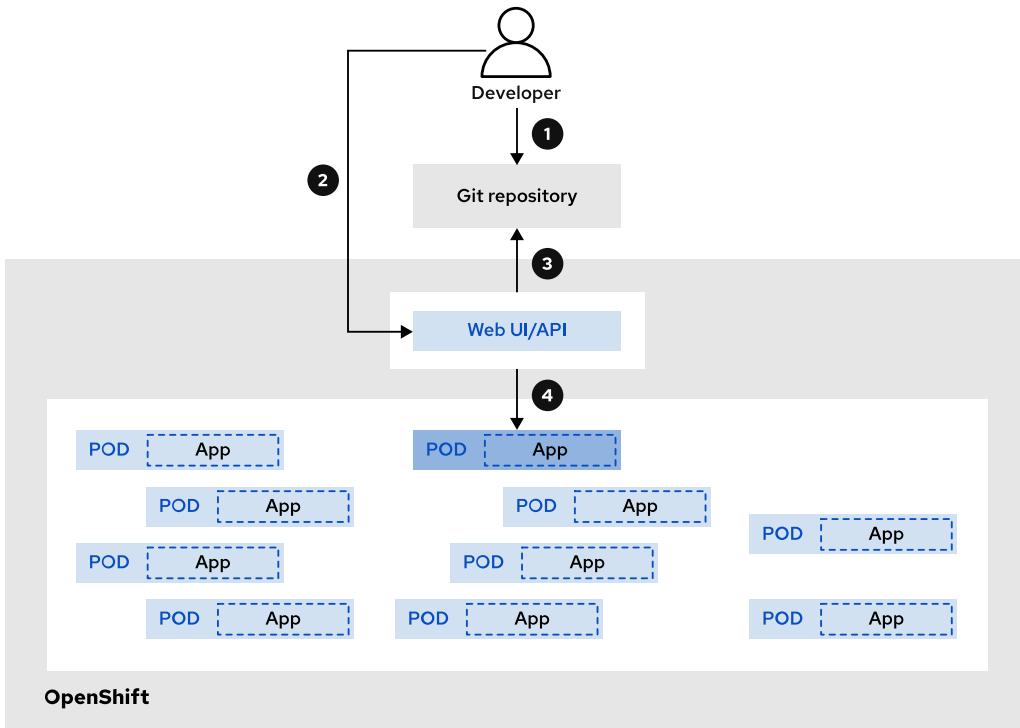


**Figure 3.2: A container in a pod**

The previous diagram shows a container for a generic application. The container groups the application runtime (for example, the Java Virtual Machine), the dependencies required by the application (such as those declared in the `pom.xml` file), and the application itself.

To manage the containerized applications, OpenShift Container Platform adds a layer of abstraction known as the pod.

Pods are the basic unit of work for OpenShift Container Platform. A pod encapsulates a container, and other parameters, such as a unique IP address or storage. A pod can also group several related containers that share resources.



**Figure 3.3: Pods running on OpenShift Container Platform**

The preceding diagram shows an OpenShift Container Platform instance hosting several applications and running them in pods.

## Deploying Applications in OpenShift Container Platform

OpenShift Container Platform offers a variety of ways to deploy applications, depending on how the application is available. If the application source code is available in a Git repository, then OpenShift Container Platform can use the Source-to-Image approach to create the container image. To deploy a new application from source, use the following workflow.

- The developers commit work to a Git repository.
- When ready to deploy their code, the developers use the OpenShift Container Platform web console to create the application. The URL to the Git repository is one of the required parameters.
- OpenShift Container Platform retrieves the code from the Git repository and builds the application.
- OpenShift Container Platform deploys the application in a pod.

If the application is already available as a container image, then OpenShift Container Platform can grab the image directly from the image repository and use it to create the application containers and pods.

Another approach to deploy applications is using Operators. Operators are first-class components of OpenShift Container Platform, which can deploy, update, and maintain applications inside the cluster.

## OpenShift Container Platform Resource Types

OpenShift Container Platform uses resources to describe the components of an application. When you deploy a new application, OpenShift Container Platform creates those resources for you, and you can view and edit them through the web console.

Some resource kinds available are:

### **Pod**

The representation of one or more containers running on the platform.

### **Route**

The association of a public URL to the application, so clients can reach it from outside OpenShift Container Platform.

### **ConfigMap**

A set of keys and values associated to an application, usually injected into containers as environment variables.

### **Deployment/DeploymentConfig**

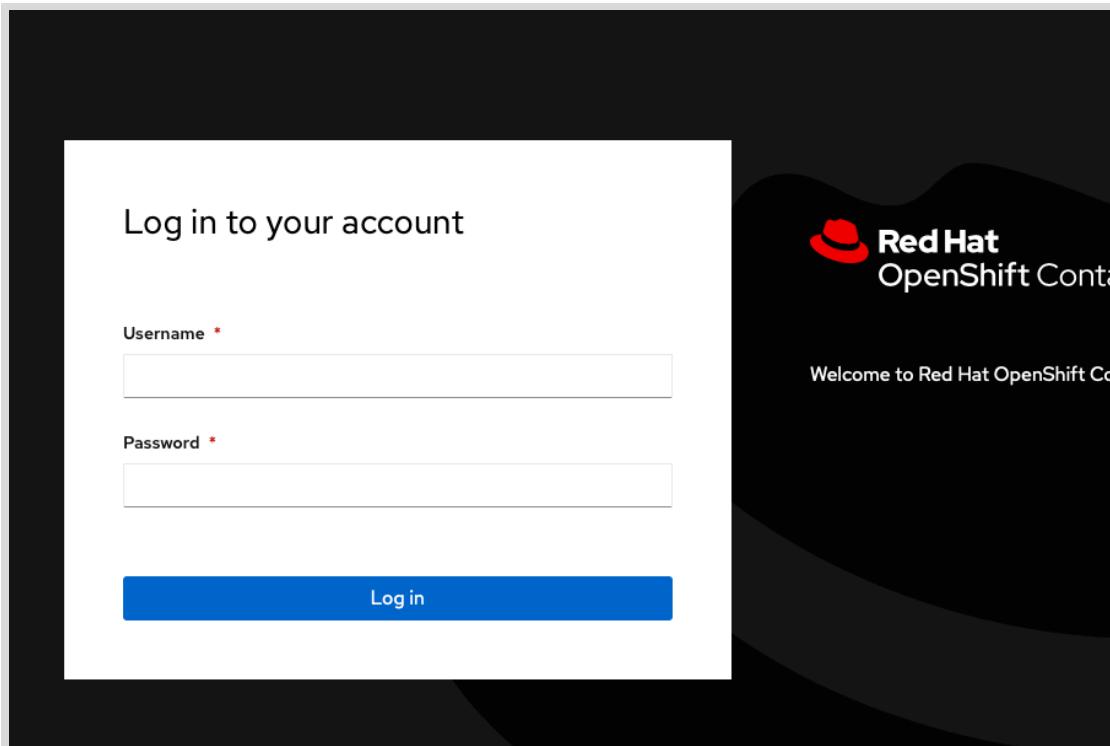
A declarative description on how to deploy the application. It usually contains the container images for the application, the number of containers to run, and the ConfigMap resources to inject.

## Introducing the Developer Web Console for OpenShift Container Platform

The OpenShift Container Platform web console is a browser-based user interface, which provides a graphical alternative to the command-line tools. With the web UI, developers can easily deploy and manage their applications.

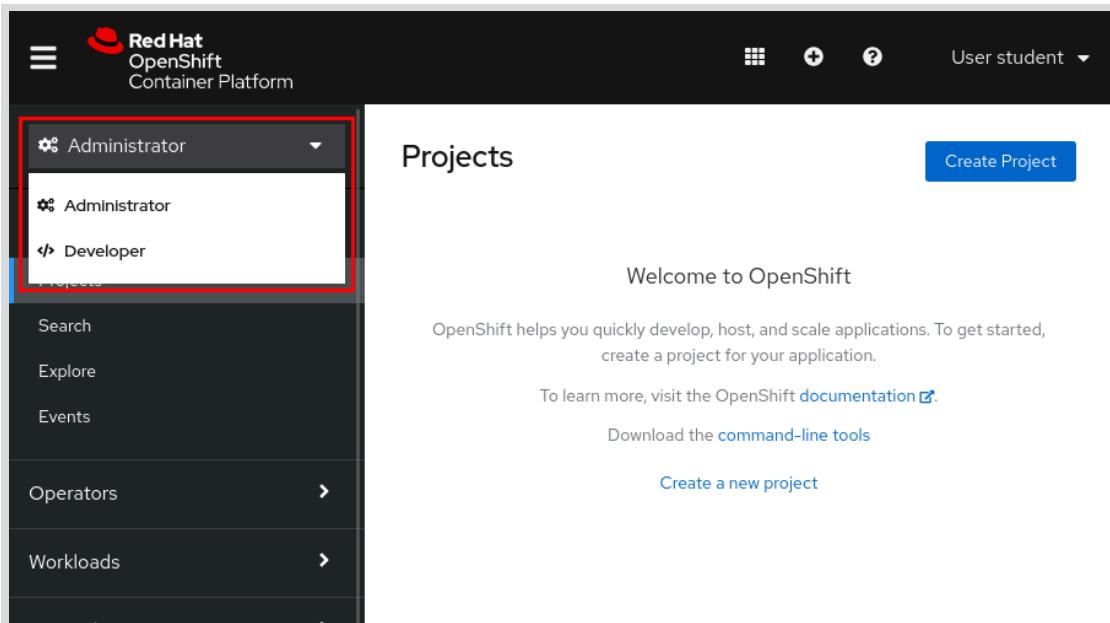
## Authenticating and Accessing the Developer Perspective

To access the web console, use the URL of your OpenShift Container Platform. To use OpenShift Container Platform, each developer must have an account. The following screen capture shows the login page.



**Figure 3.4: Logging in to OpenShift Container Platform**

The web console provides two perspectives, one for developers, and the other for administrators and operators. As shown in the following screen capture, the perspective is selected from the menu on the left. As a developer, you typically select the **Developer** perspective.



**Figure 3.5: The Administrator and Developer perspectives**

## Creating a Project

OpenShift Container Platform groups applications in **projects**. Using projects, developer teams can organize content in isolation from other teams. Projects allow for the grouping of an

application's individual components (front end, back end, and database), and can be used as lifecycle environments (development, QA, production).

To create a project, select **Create Project** from the Project list.

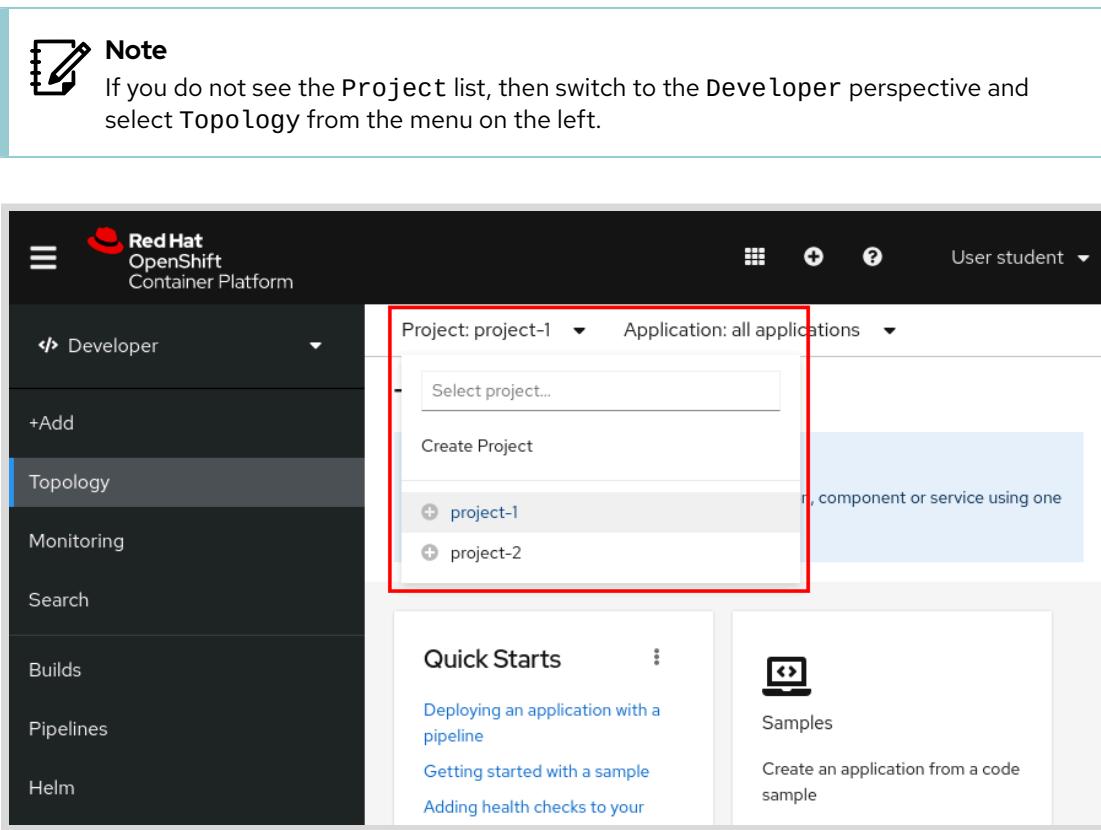
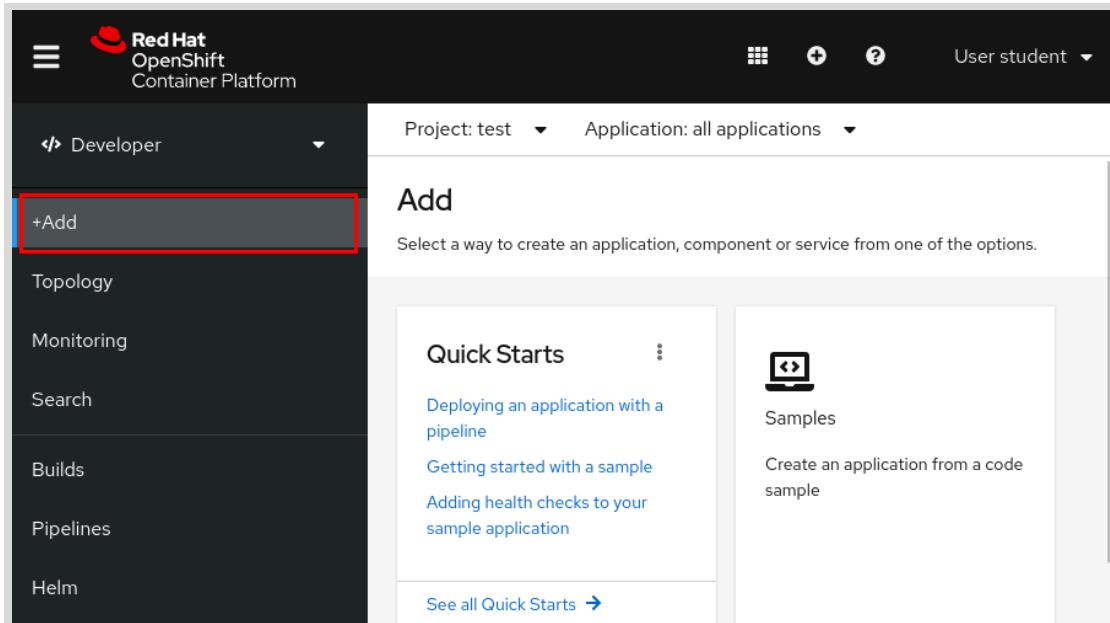


Figure 3.6: Creating a project

## Deploying a New Application

OpenShift Container Platform provides several methods to add a new application. The Add option is the entry point to an assistant, which allows you to choose between the available methods for deploying an application to the OpenShift Container Platform cluster as part of a specific project.



**Figure 3.7: Selecting a method to deploy a new application in OpenShift Container Platform**

With the **From Catalog** method, you can list and deploy the available ready to use applications, for example, a MariaDB database. You can also select the language of your application, and provide its source code from a Git repository.

The following screen capture shows some of the available languages.

**Note**  
You might need to select the **Template** check box under the **Type** filter to view the language templates.

The screenshot shows the 'Developer Catalog' screen. The left sidebar has '+Add' selected. The main content area is titled 'Developer Catalog' and contains a table with columns 'All Items' and 'Languages'. The 'Languages' column lists Java, JavaScript, .NET, Perl, and Ruby. There are filters for 'Filter by keyword...' and 'Group By: None'. A 'DHD' icon is at the bottom left, and a 'Template' link is at the bottom right. The total count is 32 items.

**Figure 3.8: Available languages in the catalog**

After selecting the application language from the catalog and clicking **Instantiate Template**, OpenShift Container Platform provides a form to collect the application details.

The screenshot shows the Red Hat OpenShift Container Platform web interface. On the left, there's a sidebar with options like Developer, +Add, Topology, Monitoring, Search, Builds, Pipelines, and Helm. The main area is titled 'Developer' and contains a form for deploying an application. The fields are:

- Java Version \***: latest
- Git Repository URL \***: https://github.com/jboss-openshift/openshift-quickstarts
- Git Reference**: master
- Context Directory**: undertow-servlet

Below the form, a note says: "Path within Git project to build: empty for root project directory."

**Figure 3.9: Deploying an application - Git details**

Use the fields in the first section of the form to provide Git repository details for the application.

If the source code to deploy is not in the Git **master** branch, then provide the branch name in the **Git Reference** field. Use the **Context Dir** field when the application to deploy is in a subdirectory, and not at the root of the Git repository.

The **Name** field is an internal name that OpenShift Container Platform uses to identify all the resources it creates during build and deployment. That name is used, for example, for the route resource that associates a public URL to the application.

## Reviewing an Application in the Web Console

The **Topology** option provides an overview of the applications in a project. The following screen capture shows two applications running in the **bookstore** project: the **frontend** PHP application and a database.

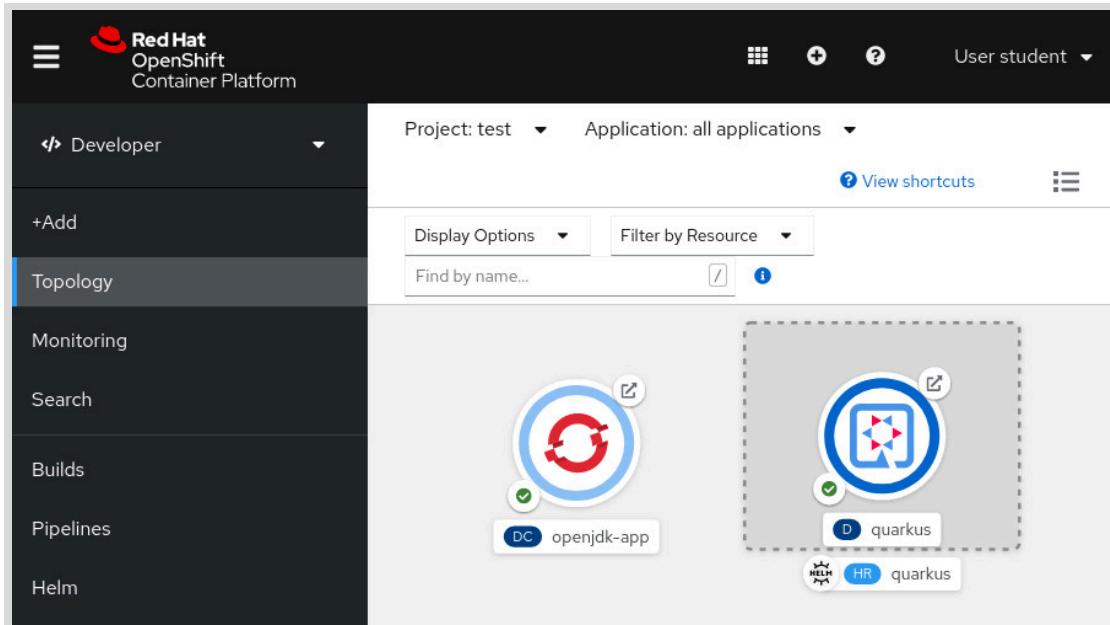


Figure 3.10: Overview of the applications in a project

Click the **application** icon to access application details. The following screen capture shows the resources that OpenShift Container Platform creates for the application. Notice that one pod is running, the build is complete, and a route provides external access to the application.

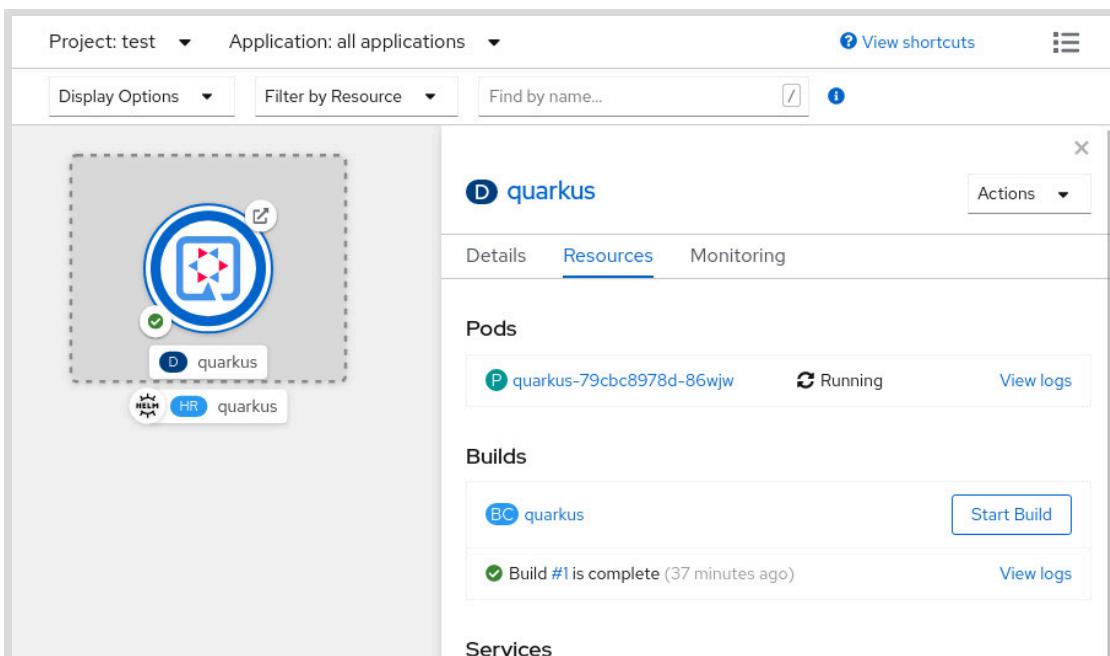


Figure 3.11: Resources of an application

Usually, when the web console displays a resource name, it is a link to the details page for that resource. The following screen capture displays the details of the Route resource.

The screenshot shows the Red Hat OpenShift Container Platform web console. At the top, the navigation bar includes the Red Hat logo, 'OpenShift Container Platform', and a user dropdown for 'User student'. Below the header, the project 'test' is selected. The main content area shows a route named 'quarkus' with status 'Accepted'. The 'Route Details' tab is active, displaying the following information:

- Name:** quarkus
- Namespace:** NS test
- Labels:** (empty)
- Location:** <https://quarkus-test.apps.na46-stage2.dev.nextcle.com>
- Status:** Accepted

At the bottom right of the details section, there are 'Edit' and 'Host' buttons. To the right of the 'Actions' button, a dropdown menu is visible.

Figure 3.12: Application's route details

## Editing OpenShift Container Platform Resources

Most resource detail pages from the OpenShift Container Platform web console provide an **Actions** button, which displays a menu.

This screenshot is similar to Figure 3.12, showing the route details for 'quarkus'. A red box highlights the 'Actions' dropdown menu, which contains the following options:

- Edit Labels
- Edit Annotations
- Edit Route
- Delete Route

Figure 3.13: Available actions for a route resource

This menu usually provides options to edit and delete the resource.



## References

For more information, refer to the Product Documentation for Red Hat OpenShift Container Platform at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6)

### Introduction to OpenShift Container Platform

<https://www.openshift.com/about/>

## ► Guided Exercise

# Deploying Microservices in Red Hat OpenShift Container Platform

In this exercise, you will deploy a simple application using the OpenShift Container Platform console. The application is available as a container image in `quay.io/redhattraining/quarkus-expense:latest`.

## Outcomes

You should be able to deploy applications in Red Hat OpenShift Container Platform using just the web console.

## Before You Begin

To use your provided OpenShift Container Platform console, you need the appropriate credentials supplied by your Red Hat Training Online Learning environment. Review the *Guided Exercise: Verifying OpenShift Credentials* section if you need guidance for finding your credentials.

## Instructions

- ▶ 1. Log in to the OpenShift Container Platform web console and create a project named `deploy-service-<your-username>`
  - 1.1. On your **workstation**, open a web browser and navigate to the **Console Web Application URL**. This URL starts by `https://console-openshift-console.apps`.
  - 1.2. Log in to your account using the user name and password provided by your Red Hat Training Online Learning environment.
  - 1.3. If not already there, then navigate to the **Home > Projects** page, and click the blue **Create Project** button on the top right side of the page.
  - 1.4. In the **Create Project** dialog, set the project name as `deploy-service-<your-username>`. You can leave the rest of the fields blank or add descriptive values at your will. Click **Create** when ready.
- ▶ 2. Deploy the `expenses` microservice using the **Deploy image** wizard. Use the container image at `quay.io/redhattraining/quarkus-expense:latest`.
  - 2.1. Use the top-left drop-down to switch from the **Administrator** perspective to the **Developer** perspective. If you do not land on the **Add** page, then select the **> +Add** entry on the left menu.
  - 2.2. From the various wizard cards available, click the **Container Image**.
  - 2.3. Select the **Image name from external registry** option and enter `quay.io/redhattraining/quarkus-expense:1.11` as the image name. Make sure that you write the right image tag.

Use `quarkus-expense-app` as the application name and `quarkus-expense` as the component name. OpenShift Container Platform names all generated resources after the component name.

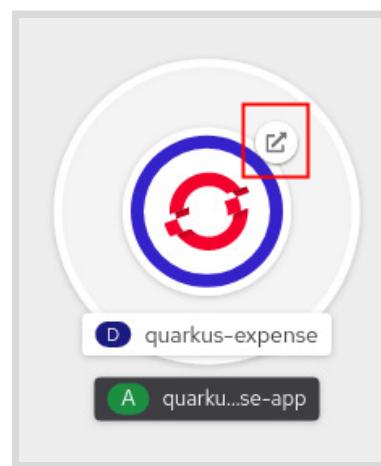
Select the **Deployment** option for the generated resource type. **Deployment Config** might also work, but later versions of OpenShift Container Platform encourage using **Deployment** resources.

From the **Advanced Options** section, check the **Create a route to the application** checkbox.

From the same section, click **Routing** to display the route options. Write `8080` in the **Target Port** field and click **Create "8080"**.

Finally, click **Create** to create the resources and deploy the application. The **Topology** page opens, and you can see your application deploying.

- ▶ 3. Use the topology view to see the deployment and application status and validate it is working correctly.
  - 3.1. Your application has deployed successfully if a dark blue ring wraps your application's icon in the topology view. You can see other useful information in the topology graph, such as the name of the deployment, the name of the containing application, the deployment's success status, and the relation between different deployments.
  - 3.2. You can see a black and white icon composed of a square with an outgoing arrow on the top-right of the deployment icon. Click this icon to open the URL assigned to your application. A new browser tab opens, showing a page titled `Your new Cloud-Native application is ready!` and containing some application details.



**Figure 3.14: The application in the topology view, highlighting the Open URL icon**

- 3.3. Add the path `/swagger-ui` to the URL just opened. That new URL loads the Swagger page to test your application endpoints.

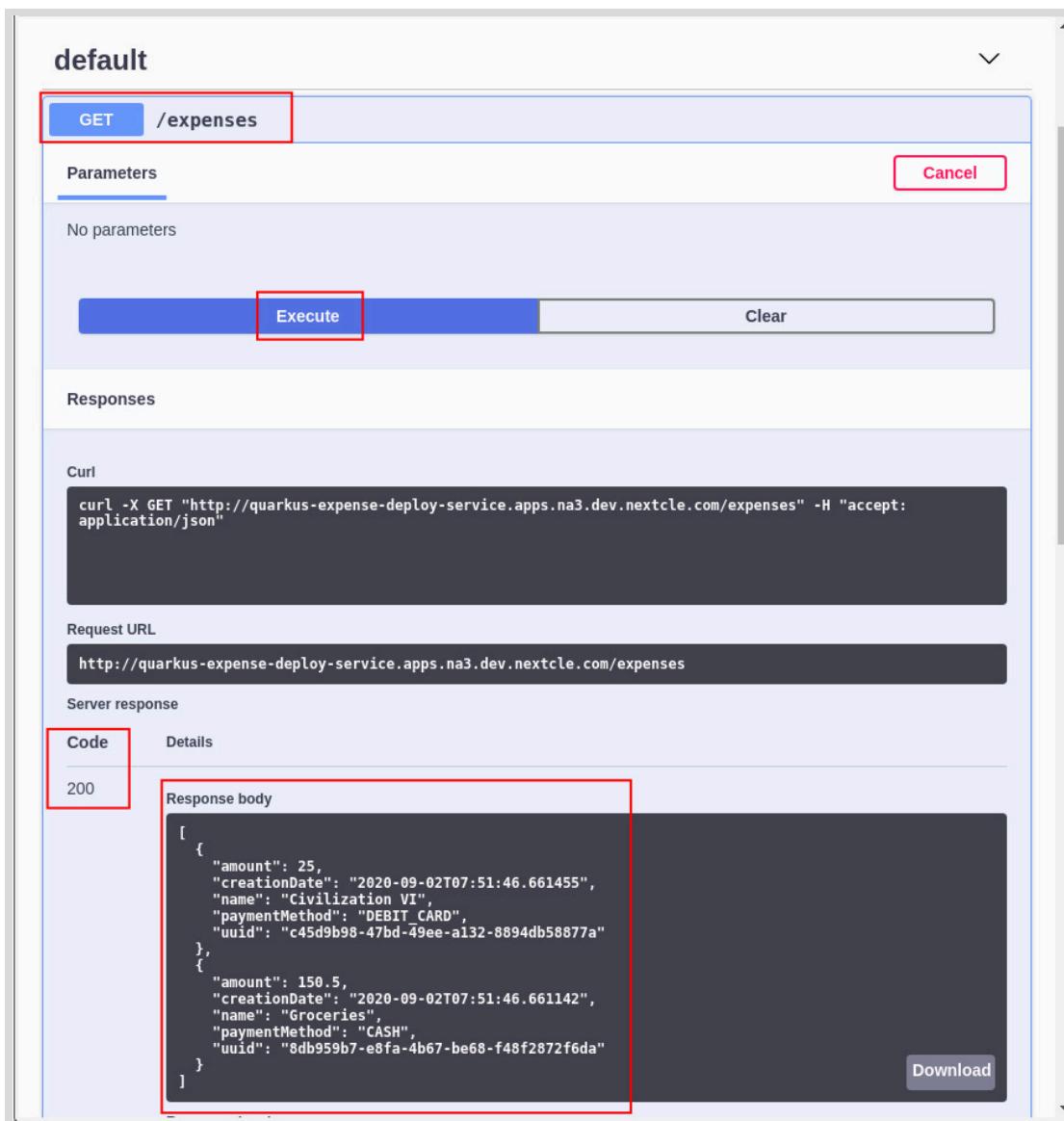


#### Note

Some browsers automatically change the protocol from `http` to `https` when editing the URL. For this exercise you must use `http` only.

Click `GET /expenses` to unfold the section. Then, click **Try it out** and **Execute** as they appear. The **Responses** section updates with the `curl` command used to invoke the endpoint, the request URL, the response code, and the response body. Confirm that

the response code is 200, and the response body contains a JSON array with two expense entries.



**Figure 3.15: The Swagger UI with highlighted referenced elements**

- 3.4. Close the Swagger page and go back to the topology view.
- ▶ 4. Review resources generated automatically by the deployment wizard: ImageStream, Deployment, Pod, Container, Service, and Route.
  - 4.1. Click the center of your application's icon (the OpenShift icon in this case). The side window showing the main details of your application opens on the right of the topology view.

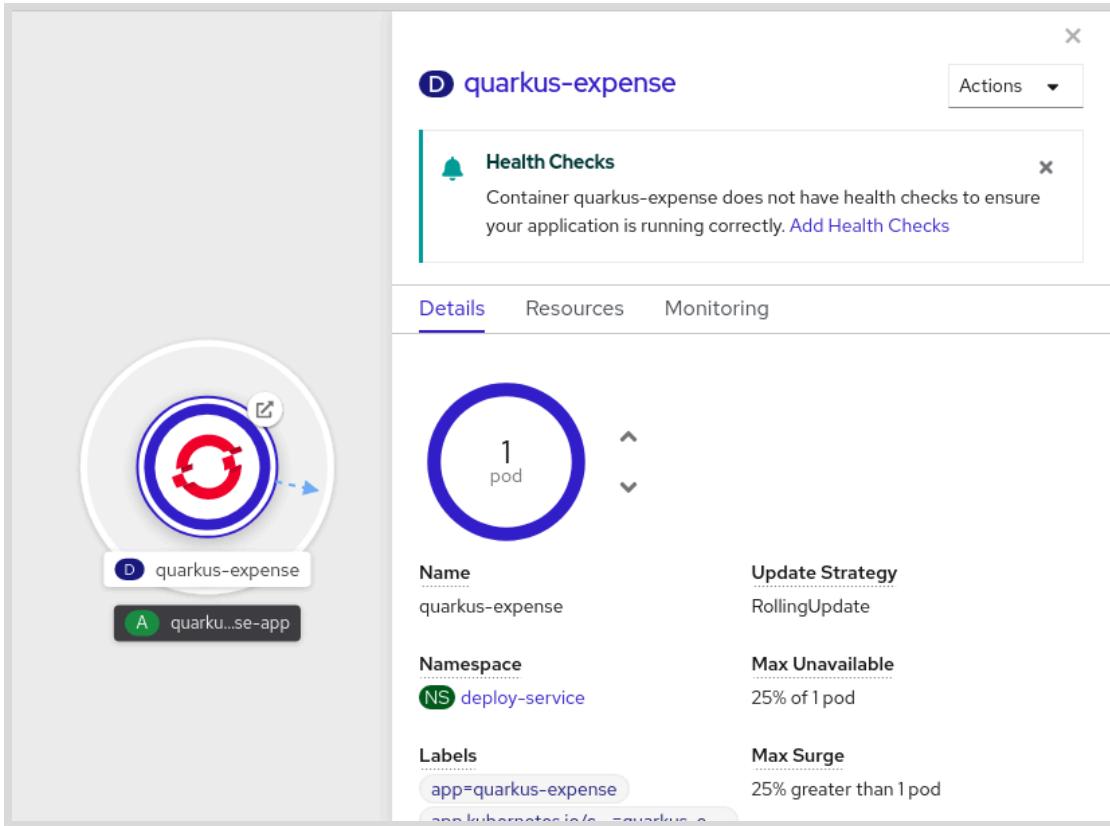


Figure 3.16: The side window showing the selected deployment details

**Note**

If the content of the side window does not look like the previous image, then you might have clicked outside the ring around the application's icon. This region contains all deployments belonging to the same application (just one in this case). Try clicking directly on the center of the application's icon.

- 4.2. In the side window, click the **Details** tab. This tab describes the **Deployment** resource created for the application. The blue ring with the text **1 pod** inside displays the status of the pods for this deployment. Use the up and down arrows beside the ring to increase or decrease the number of pod replicas.

This tab includes some interesting information about the deployment, such as the namespace, the labels applied to the deployment, or the update strategy used to rollout new versions of the application.

  - 4.3. In the side window, click the **Resources** tab. This tab shows some resources associated with the just deployed application.
  - 4.4. The **Pods** section shows the list of Pod resources created for the deployment. You can see each pod name (prefixed by the resource name **quarkus-expense**), the pod status (**Running** in this case), and a link to the pod logs.
- Open in a new browser tab any of the **View logs** links. The opened page includes the output of your application, which you can use to monitor its progress. Close the tab and return to the topology view.

- 4.5. The **Builds** section is empty for this deployment. The reason for this is that we deployed the application from a container image, so no build process is needed. Another kind of deployment (for example, when deploying directly from Git) generates **BuildConfiguration** and **Build** resources to drive the build process.
- 4.6. The **Services** section displays **service** resource entries associated with this deployment. Between many other properties, **service** resources determine what network ports expose the application and how these ports map to internal ports in the container.
- 4.7. The **Routes** section lists the URLs that expose the application to external networks. **Route** resources define external URLs, TLS security settings, and map to application **service** resources.
- 4.8. The deployment wizard automatically generates a few other resources. To find them, navigate to the **Search** entry in the left menu and select a resource from the **Resources** drop-down. For example, if you search for resources of type **ImageStreamTag**, then you will see a resource of that kind describing a local copy of the container image used to deploy the expenses application.

## Finish

Clean up your lab environment by deleting the `deploy-service-<your-username>` project.

Go back to the **Administrator** perspective and select the **Home > Projects** entry. In the project list, select the triple-dot icon on the right side of the `deploy-service` project and, in the folding menu, select **Delete Project**. Type `deploy-service` in the modal form that displays, and click **Delete**.

This concludes the guided exercise.

# Deploying Quarkus Applications in Red Hat OpenShift Container Platform

---

## Objectives

After completing this section, you should be able to create container images for Quarkus applications and deploy them in OpenShift Container Platform using Quarkus extensions.

## Quarkus OpenShift Extension

Cloud-native applications are usually deployed on container orchestration platforms, such as Kubernetes or OpenShift Container Platform. Quarkus is a developer-centric framework, which provides an extension that eases deployment into those container orchestration platforms for testing purposes.

To deploy to OpenShift Container Platform, Quarkus offers the Quarkus OpenShift extension with the Maven coordinates `io.quarkus:quarkus-openshift`. You can add this extension using the usual Quarkus method for adding extensions to a Maven project.

```
[user@host ~]$ mvn quarkus:add-extension -Dextensions="openshift"
```

Or by adding the dependency manually in the application's `pom.xml`, referencing its coordinates.

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-openshift</artifactId>
</dependency>
```

After adding the extension to the Quarkus project, you can request the extension to deploy the application to an OpenShift Container Platform cluster adding the `-Dquarkus.kubernetes.deploy=true` option to the package maven command. For example.

```
[user@host ~]$ mvn package \ ①
-Dquarkus.kubernetes.deploy=true ②
```

- ① The package command that usually creates the jar, when the Quarkus OpenShift extension is present, also generates the resources.
- ② This property triggers the actual deployment to OpenShift Container Platform.

This command packages the application, generates the Kubernetes/OpenShift resources and deploys the application into the cluster you are currently logged in to.

The Quarkus OpenShift extension is just a wrapper around the Quarkus Kubernetes extension, which is preconfigured for OpenShift Container Platform targets. This is the reason why the `deploy` property is named `quarkus.kubernetes`, because it is shared with its parent extension.

## Log in to OpenShift

For the Quarkus OpenShift extension to work, you must be logged in on an OpenShift Container Platform cluster. To do that, there is a tool to interact with OpenShift Container Platform clusters in the terminal called `oc`, for OpenShift Client. This tool allows you to run commands and apply changes to a remote cluster using its API without having to interact with it through HTTP.

Before running any other `oc` command, log into the cluster using the `oc login` command.

```
[user@host ~]$ oc login -u username -p password http://cluster-api-url
```

After successfully logging in to the cluster you must set the project to work on. If the project does not exist, then you can create it using the following command.

```
[user@host ~]$ oc new-project project-name
```

But if the project already exists, then you can switch to it by using the following command.

```
[user@host ~]$ oc project project-name
```

After logging in and selecting a project to work with, you can now use the Quarkus OpenShift extension to deploy a service into the cluster.

## Generating OpenShift Resources

The Quarkus OpenShift extension generates four resource files in the `target/kubernetes` directory: `kubernetes.yml` and `openshift.yml` and their JSON equivalents. Both the YAML and JSON files describe identical deployments, and you can use either, depending on your personal preference. The `openshift.yml` file defines all the resources necessary to deploy an application on an OpenShift Container Platform cluster.

This extension selects sensible defaults for the required resources, so typically no further configuration is needed just for testing purposes. You can also configure all the generated resources in the `application.properties` file, to fine tune the deployment of your application into the cluster.

The most common option for the Quarkus OpenShift extension is `quarkus.openshift.expose=true` which controls the creation of a `route` resource to expose the application. The default value for this option is `false`. To enable exposing the application route, simply add this line to the `application.properties` file or pass it as a parameter to the package command.

```
quarkus.openshift.expose=true
```

Another common customization is modifying the name of the application in the cluster. You can change this using the `quarkus.openshift.name` option, which defaults to the project name if not given explicitly. For example.

```
quarkus.openshift.name=my-cloudnative-app
```

A common practice in orchestrated container clusters is to add labels to the deployed containers to make the containers more easily identifiable. To add labels to your application's deployment, use `quarkus.openshift.labels.<label-name>=<label-value>`, for example.

```
quarkus.openshift.labels.release-status=production
```

In cloud-native applications, the primary way to configure an application is through environmental variables, which the application reads at startup time. The value of these environmental variables can be defined in the `application.properties` file or passed as parameters to the package command. Some commonly used variables are database configuration, service endpoints, logging configuration, locale, internationalization, encryption keys, and login data (although `Secrets` are preferred for confidential data).

To define a static value for an environmental variable, you can use the `quarkus.openshift.env.vars.<env-var-name>=<env-var-value>`, for example.

```
quarkus.openshift.env.vars.DB_USERNAME=dbuser
```

This will create an environmental variable with the name `DB_USERNAME` and the value `dbuser`.

If the application project contains `ConfigMap` or `Secret` resources then you can capture some or all of their values into environment variables by using the `quarkus.openshift.env.configmaps` and `quarkus.openshift.env.secrets` entries:

```
quarkus.openshift.env.configmaps=aConfigMap,anotherConfigMap ①
quarkus.openshift.env.secrets=aSecret,anotherSecret ②
quarkus.openshift.env.mapping.DB_USERNAME.from-configmap=datasource ③
quarkus.openshift.env.mapping.DB_USERNAME.with-key=username ④
quarkus.openshift.env.mapping.DB_PASSWORD.from-secret=datasource ⑤
quarkus.openshift.env.mapping.DB_PASSWORD.with-key=password ⑥
```

- ① Captures all entries in the `aConfigMap` and `anotherConfigMap` `ConfigMap` resources as environment variables.
- ② Captures all entries in the `aSecret` and `anotherSecret` `Secret` resources as environment variables.
- ③ ④ Captures the value of the entry with key `username` from the `datasource` `ConfigMap` as an environment variable named `DB_USERNAME`.
- ⑤ ⑥ Captures the value of the entry with key `password` from the `datasource` `Secret` as an environment variable named `DB_PASSWORD`.

## Configuring Container Images Builds

Apart from controlling the generated resources for the OpenShift Container Platform deployment, you can also control how the container image is generated.

To change the default value for the generated container name you can use `quarkus.container-image.group`, `quarkus.container-image.name` and `quarkus.container-image.tag`. The following example generates a container image with the name `com.myorganization/quarkus-app:1.0` using `application.properties`.

```
quarkus.container-image.group=com.myorganization
quarkus.container-image.name=quarkus-app
quarkus.container-image.tag=1.0
```

You can also modify the base image that was used to merge the built artifact. To do this select the image using `quarkus.s2i.base-jvm-image`. For example, to use the UBI image version 8 containing the OpenJDK 11, add `quarkus.s2i.base-jvm-image=registry.access.redhat.com/ubi8/openjdk-11` to the application properties or the package command.

This build can be considered an S2I binary build because the package created in your local machine gets copied to the base image to generate the final container image. A true S2I source build starts from the source code in a builder container located in the destination OpenShift Container Platform. This makes the build immutable and repeatable, which is the best approach for containerized workloads.



### Note

Because binary S2I builds are not repeatable and machine-dependent, Red Hat does not recommend the use of the Quarkus OpenShift extension to deploy to production clusters. Please refer to the Using S2I to deploy Quarkus applications on OpenShift section of the official Red Hat Build of Quarkus documentation for more information.

## Building Container Images

You can also build a container image by setting the `quarkus.container-image.build` configuration value to `true`. For building locally, add an extension that will take care of building the image, for example `container-image-jib`.

```
[user@host quarkus-project]$ mvn clean package \
-Dquarkus.container-image.build=true
```

The preceding command will create a container image. It will register the image with the local Docker daemon, unless the `quarkus.container-image.push` configuration value is set to a valid registry value. This means that, with the current implementation of the extension, you need Docker running, unless you are pushing to a registry.

You can also create container images with the `container-image-s2i` extension, which will require using the `openshift` extension described earlier in this section.

## Eclipse JKube

Eclipse JKube is an alternative approach to generate resources and deploy into OpenShift Container Platform.

Eclipse JKube is a generic maven plug-in, which you can use with Quarkus applications and any kind of cloud-native application that targets a Kubernetes like system. You can use JKube to deploy Quarkus applications to an OpenShift Container Platform, but you lose all the tight integration that the specialized `openshift` extension provides.



## References

### **Deploying your Quarkus applications to OpenShift**

[https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.11/html-single/deploying\\_your\\_quarkus\\_applications\\_to\\_openshift/index](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11/html-single/deploying_your_quarkus_applications_to_openshift/index)

### **Eclipse JKube**

<https://www.eclipse.org/jkube/>

## ► Guided Exercise

# Deploying Quarkus Applications in Red Hat OpenShift Container Platform

In this exercise, you will deploy a microservice application on an OpenShift Container Platform cluster.

## Outcomes

You should be able to deploy microservices applications on OpenShift Container Platform.

## Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the Expenses application is ready to be deployed into the cluster.

```
[student@workstation ~]$ lab deploy-openshift start
```

## Instructions

- 1. Log in to Red Hat OpenShift Container Platform with your developer user.

- 1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to the OpenShift Container Platform cluster.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} -p \
${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 1.3. Change to the exercise project.

```
[student@workstation ~]$ oc project deploy-openshift-${RHT_OCP4_DEV_USER}
```

- 2. Enter the Quarkus Expenses application directory.

```
[student@workstation ~]$ cd ~/D0378/labs/deploy-openshift/expense-service/
```

- 3. Add the Quarkus OpenShift extension to the application.

```
[student@workstation expense-service]$ mvn quarkus:add-extension \
-Dextensions="openshift"
...output omitted...
Extension io.quarkus:quarkus-openshift has been installed
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 4. Build the application and deploy it into the cluster. Expose the application using a Route resource.

```
[student@workstation expense-service]$ mvn clean package \
-Dquarkus.kubernetes.deploy=true \
-Dquarkus.openshift.expose=true \
-DskipTests
...output omitted...
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Service
expenses.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: ImageStream
expenses.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: ImageStream
openjdk-11.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: BuildConfig
expenses.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied:
DeploymentConfig expenses.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Route
expenses.
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 5. Verify that the service is deployed in the cluster.

```
[student@workstation expense-service]$ oc get service,pod
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/expenses ClusterIP 172.30.139.80 <none> 8080/TCP 26s

NAME READY STATUS RESTARTS AGE
pod/expenses-1-build 0/1 Completed 0 59s
pod/expenses-1-deploy 0/1 Completed 0 25s
pod/expenses-1-v7p6n 1/1 Running 0 23s
```

- 6. Verify that the application is working on the exposed service location.

```
[student@workstation expense-service]$ curl -s \
http://expenses-deploy-openshift-${RHT_OCP4_DEV_USER}.\${
RHT_OCP4_WILDCARD_DOMAIN}/expenses | jq
[
 {
 "id": 1,
 "amount": 150.5,
 "name": "Groceries",
 "paymentMethod": "CASH"
 },
 {
 "id": 2,
 "amount": 25,
 "name": "Civilization VI",
 "paymentMethod": "CREDIT_CARD"
 }
]
```

Check that the expenses initially loaded in the application appear in the response.

- 7. Review the generated resources that define the deployment of the application in the OpenShift Container Platform cluster.

- 7.1. Find out the name of the service and the destination port for the application in the service definition.

```
[student@workstation expense-service]$ grep '^kind: Service' -A 20 \
target/kubernetes/openshift.yml
kind: Service
metadata:
 annotations:
 app.quarkus.io/build-timestamp: "2020-09-07 - 15:34:15 +0000"
 app.openshift.io/vcs-url: "<unknown>"
 app.quarkus.io/commit-id: "<unknown>"
 app.quarkus.io/vcs-url: "<unknown>"
 labels:
 app.kubernetes.io/name: "expenses"
 app.kubernetes.io/version: "1.0-SNAPSHOT"
 app.openshift.io/runtime: "quarkus"
 name: "expenses"
spec:
 ports:
 - name: "http"
 port: 8080 targetPort: 8080
 selector:
 app.kubernetes.io/name: "expenses"
 app.kubernetes.io/version: "1.0-SNAPSHOT"
 type: "ClusterIP"
```

- 7.2. Find out the Route definition for the target Service.

```
[student@workstation expense-service]$ grep '^kind: Route' -A 20 \
target/kubernetes/openshift.yml
kind: Route
metadata:
 labels:
 app.kubernetes.io/name: "expenses"
 app.kubernetes.io/version: "1.0-SNAPSHOT"
 name: "expenses"
spec:
 host: ""
 path: "/"
 port:
 targetPort: 8080
 to:
 kind: "Service"
 name: "expenses"
```

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab deploy-openshift finish
```

This concludes the guided exercise.

## ► Lab

# Deploying Microservice-based Applications

In this lab, you will deploy the Conference application into OpenShift Container Platform using different techniques.

## Outcomes

You should be able to use different approaches to deploy microservice applications into OpenShift Container Platform.

## Before You Begin

To use your provided OpenShift Container Platform console, you need the appropriate credentials supplied by your Red Hat Training Online Learning environment. Review the *Guided Exercise: Verifying OpenShift Credentials* section if you need guidance for finding your credentials.

As the student user on the **workstation** machine, use the `lab` command to prepare your system for this lab.

This command prepares the source code for the different microservices creating the conference application in your workstation.

```
[student@workstation ~]$ lab deploy-review start
```

## Instructions

1. Log in to your OpenShift Container Platform instance and start using the project created for you. The project name is created by the `deploy-review-` prefix followed by your user name, which is provided by your Red Hat Training Online Learning environment.
2. Deploy the speaker microservice using the Quarkus `openshift` extension, and expose the application to the default route. The source code for the speaker microservice is available in `~/D0378/labs/deploy-review/microservice-speaker`.  
Use the `registry.access.redhat.com/ubi8/openjdk-11` image as the base image for the application container.  
Enable the Swagger UI in the production profile. To do so, add the `quarkus.swagger-ui.always-include=true` parameter to the build command.  
Review the deployment in the Topology view of your OpenShift Container Platform console. Use Swagger UI to validate that the service is running correctly. Remember the Swagger UI default path is `/swagger-ui`.
3. Deploy the schedule microservice using the container image deployment wizard from the OpenShift Container Platform console. The container images are already available at `quay.io/redhattraining/quarkus-conference-schedule`.

Review the deployment in the Topology view of your OpenShift Container Platform console. Use Swagger UI to validate the service is running correctly. The provided container image already has Swagger UI enabled for production deployments.

- 3.1. Go to the OpenShift Container Platform console and make sure you have the `deploy-review-your_username` project selected. Open the **Developer > Add** page and select **Container image** from the list of cards.  
Select the option **Image name from external registry** and use `quay.io/redhattraining/quarkus-conference-schedule` as the image name. Leave the default values for the rest of the fields. Verify the **Create a route to the application** checkbox is marked. Finally, click **Create**.
- 3.2. The **Developer > Topology** view opens automatically. Click the icon for the `quarkus-conference-schedule` microservice to see the details in the side window. Check that the service pod is running successfully and that there are no builds associated with this deployment.



#### Note

The deployment icon is surrounded by a grey boundary, which represents the application. To add Deployment and other resources to an application, add the label `app.kubernetes.io/part-of=application-name`. You can also add or remove components to an application by holding the **Shift** key whilst dragging the deployment icon in or out of the application boundary.

Click the **Open URL** button to open a new tab, which will bring you to the URL of the application's default route. Append `/swagger-ui` to the URL to enter the Swagger UI.



#### Note

Some browsers modify the URL protocol to `https` when editing the URL. Make sure you use the `http` protocol.

Expand the `GET /schedule/all` endpoint section and click the **Try it out** button. Click the **Execute** button and validate that the service responds a 200 HTTP code, and that the response body contains four schedule entries, each one with just one `id` and one `venueId` field.

## Evaluation

Grade your work by running the `lab deploy-review grade` command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab deploy-review grade
```

## Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab deploy-review finish
```

This concludes the lab.

## ► Solution

# Deploying Microservice-based Applications

In this lab, you will deploy the Conference application into OpenShift Container Platform using different techniques.

## Outcomes

You should be able to use different approaches to deploy microservice applications into OpenShift Container Platform.

## Before You Begin

To use your provided OpenShift Container Platform console, you need the appropriate credentials supplied by your Red Hat Training Online Learning environment. Review the *Guided Exercise: Verifying OpenShift Credentials* section if you need guidance for finding your credentials.

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command prepares the source code for the different microservices creating the conference application in your workstation.

```
[student@workstation ~]$ lab deploy-review start
```

## Instructions

1. Log in to your OpenShift Container Platform instance and start using the project created for you. The project name is created by the `deploy-review-` prefix followed by your user name, which is provided by your Red Hat Training Online Learning environment.
  - 1.1. Open a terminal window and log into the OpenShift Container Platform cluster using the `oc login` command.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ oc login ${RHT_OCP4_MASTER_API} \
-u ${RHT_OCP4_DEV_USER} -p ${RHT_OCP4_DEV_PASSWORD}
Login successful.
...output omitted...
```

- 1.2. Use the project created to hold your application by passing its name to the `oc project` command.

```
[student@workstation ~]$ oc project deploy-review-${RHT_OCP4_DEV_USER}
Now using project "deploy-review-your_username" on server ...
```

2. Deploy the speaker microservice using the Quarkus openshift extension, and expose the application to the default route. The source code for the speaker microservice is available in ~/D0378/labs/deploy-review/microservice-speaker.

Use the `registry.access.redhat.com/ubi8/openjdk-11` image as the base image for the application container.

Enable the Swagger UI in the production profile. To do so, add the `quarkus.swagger-ui.always-include=true` parameter to the build command.

Review the deployment in the Topology view of your OpenShift Container Platform console. Use Swagger UI to validate that the service is running correctly. Remember the Swagger UI default path is /swagger-ui.

- 2.1. In the terminal window, change the current directory to ~/D0378/labs/deploy-review/microservice-speaker.

```
[student@workstation ~]$ cd ~/D0378/labs/deploy-review/microservice-speaker
```

- 2.2. Add the `openshift` extension to the `maven` project in the `pom.xml` file.

```
[student@workstation microservice-speaker]$ mvn quarkus:add-extension \
-Dextension=openshift
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.acme.conference:microservice-speaker >-----
[INFO] Building microservice-speaker 1.0.0-SNAPSHOT
[INFO] -----[jar]-----
[INFO]
[INFO] --- quarkus-maven-plugin:1.7.5.Final-redhat-00007:add-extension (default-
cli) @ microservice-speaker ---
...output omitted...
Adding extension io.quarkus:quarkus-openshift
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

Optionally, instead of using the `mvn quarkus:add-extension` command, you can edit the `pom.xml` file directly and add the following dependency.

```
...output omitted...
<dependencies>
...output omitted...
<dependency>
<groupId>io.quarkus</groupId>
<artifactId>quarkus-openshift</artifactId>
</dependency>
...output omitted...
```

- 2.3. Execute the deployment process by adding the appropriate parameters to the `mvn package` command. Set `quarkus.openshift.expose=true` to expose the microservice in the default route. Set `registry.access.redhat.com/ubi8/openjdk-11` as the base image of the application with the `quarkus.s2i.base-jvm-image` argument. To enable Swagger in production, use the `quarkus.swagger-ui.always-include=true` argument.

```
[student@workstation microservice-speaker]$ mvn package \
-Dquarkus.kubernetes.deploy=true -Dquarkus.openshift.expose=true \
-Dquarkus.swagger-ui.always-include=true
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.acme.conference:microservice-speaker >-----
[INFO] Building microservice-speaker 1.0.0-SNAPSHOT
[INFO] -----[jar]-----
[INFO]
...output omitted...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ microservice-speaker ---
[INFO] Building jar: /home/student/D0378/labs/deploy-review/microservice-speaker/
target/microservice-speaker-1.0.0-SNAPSHOT.jar
[INFO]
[INFO] --- quarkus-maven-plugin:1.11.7.Final-redhat-00009:build (default) @
microservice-speaker ---
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 2.4. Open the OpenShift Container Platform console. The URL of the console and your credentials are available in your Red Hat Training Online Learning environment.  
Navigate to the **Developer > Topology** view and select your project from the **Project** drop-down at the top-left. Observe the application icon and its progress. Click the application icon to open the details side window. Wait until the application is surrounded by a dark blue ring indicating the application is completely deployed and available. If the ring becomes red, then review the build and pod logs and fix the issues found.  
Click **Open URL** to open a new tab to the URL of the application's default route. Append `/swagger-ui` to the URL to enter the Swagger UI.  
Expand the `GET /speaker` endpoint section and click the **Try it out** button. Click the **Execute** button and validate the service responds a 200 HTTP code, and the response body contains four speakers.
3. Deploy the `schedule` microservice using the container image deployment wizard from the OpenShift Container Platform console. The container images are already available at `quay.io/redhattraining/quarkus-conference-schedule`.  
Expose the microservice with the default route.  
Review the deployment in the Topology view of your OpenShift Container Platform console. Use Swagger UI to validate the service is running correctly. The provided container image already has Swagger UI enabled for production deployments.
  - 3.1. Go to the OpenShift Container Platform console and make sure you have the `deploy-review-your_username` project selected. Open the **Developer > Add** page and select **Container image** from the list of cards.  
Select the option **Image name from external registry** and use `quay.io/redhattraining/quarkus-conference-schedule` as the image name. Leave the default values for the rest of the fields. Verify the **Create a route to the application** checkbox is marked. Finally, click **Create**.

- 3.2. The **Developer > Topology** view opens automatically. Click the icon for the quarkus-conference-schedule microservice to see the details in the side window. Check that the service pod is running successfully and that there are no builds associated with this deployment.



### Note

The deployment icon is surrounded by a grey boundary, which represents the application. To add Deployment and other resources to an application, add the label `app.kubernetes.io/part-of=application-name`. You can also add or remove components to an application by holding the Shift key whilst dragging the deployment icon in or out of the application boundary.

Click the **Open URL** button to open a new tab, which will bring you to the URL of the application's default route. Append `/swagger-ui` to the URL to enter the Swagger UI.



### Note

Some browsers modify the URL protocol to https when editing the URL. Make sure you use the http protocol.

Expand the `GET /schedule/all` endpoint section and click the **Try it out** button. Click the **Execute** button and validate that the service responds a 200 HTTP code, and that the response body contains four `schedule` entries, each one with just one `id` and one `venueId` field.

## Evaluation

Grade your work by running the `lab deploy-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab deploy-review grade
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab deploy-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- The Quarkus Conference application comprises four back-end microservices (`session`, `speaker`, `schedule`, and `vote`) and a web frontend. Other components are to be added to the application throughout the course.
- This application's design applies several simplifications as it is intended for educational value rather than production.
- Red Hat OpenShift Container Platform can build and deploy microservice applications given their code, git references, container images, or operators.
- Use the `openshift` extension to deploy and expose Quarkus applications into the Red Hat OpenShift Container Platform directly from the command line or the IDE.
- Many Quarkus configuration entries control the behavior of the `openshift` extension, and the resources and container images it generates.

## Chapter 4

# Building Microservice Applications with Quarkus

### Goal

Build a persistent and configurable distributed Quarkus microservices application.

### Objectives

- Inject configuration data into a Quarkus microservice.
- Implement configurable feature toggles in microservices.
- Use service discovery to connect multiple services to each other.

### Sections

- Injecting Configuration Data (and Guided Exercise)
- Implementing Feature Toggles (and Guided Exercise)
- Connecting Multiple Services (and Guided Exercise)

### Lab

Building Microservice Applications with Quarkus

# Injecting Configuration Data

## Objectives

After completing this section, you should be able to inject configuration data into a Quarkus microservice.

## Reviewing MicroProfile Config Specification

In this section, you will learn how to create configuration files for your applications.

Quarkus uses the `MicroProfile Config` specification to help you avoid using hard-coded values. This profile enables dynamic configuration of applications, without making changes to the application itself. This facilitates running the application across multiple environments, which could include development, test, and production.

To facilitate making configuration files, MicroProfile uses the concept of a `ConfigSource`. A `ConfigSource` is a source for configured values, which can come from: System Properties, Environmental Variables, or a configuration file. Each application can have multiple configuration sources, which can coexist at the same time, by having different priorities.

After the configuration values have been set, the second part of using the configuration is to inject it into your code. This is done using the `@ConfigProperty` annotation. In Quarkus, you can also use the `@ConfigProperties` annotation as an alternative, which is less verbose.



### Note

If you look at the MicroProfile Config specification, then you will find differences to the Quarkus configuration. This is because Quarkus' implementation of the specification is not fully compliant.

For example, you can use the `@ConfigProperty` annotation without also annotating with `@Inject`, which is necessary outside of Quarkus.

## Configuring a Quarkus Application

What follows is an example of using the `application.properties` file as the configuration source resource when performing the injection with `@ConfigProperty`.

The `application.properties` file:

```
format=One format for %s unit.
unit=meter
debug-flag=true # We can omit this value.
```

An example Java code snippet used to read and assign the configuration:

```
...output omitted...
import org.eclipse.microprofile.config.inject.ConfigProperty;
```

```
@Path("/example")
public class ExampleResource {
 @ConfigProperty(name = "format")❶
 public String format;

 @ConfigProperty(name = "debug-flag", defaultValue="false")❷
 public boolean debugFlag;

 @ConfigProperty(name = "unit")❸
 public Optional<String> unit;

 @GET
 @Produces(MediaType.TEXT_PLAIN)
 public String getEndpoint() {
 if (debugFlag) {
 System.out.println("The format is: " + format);
 System.out.println("The unit is: " + unit.orElse("Not specified"));
 }
 return performComputation(format, unit);
 }
 ...output omitted...
}
```

- ❶ Injects a `format` string. If the value is not provided, then an exception will be thrown.
- ❷ Injects a `boolean` value called `debug-flag`, where the hyphen will be translated to set the following letter to uppercase. If no value is set in the configuration, then `false` will be used as the default value. Notice the value is set as a `String`, and not a `boolean`, but the string will be converted automatically to `boolean`.
- ❸ Uses an `Optional` class to declare the `unit` value as optional. If no value is set, then no exception will be thrown, but the `unit` will not have a value.

## Using ConfigProperties

The above example could be refactored to use the `@ConfigProperties` annotation.

First, you must add an `example` prefix to the names of your configuration values in `application.properties`, like the following:

```
example.format=One format for %s unit.
example.unit=meter
example.debug-flag=true
```

You can then create a configuration object, which loads the values from the `ConfigSource`, and then inject it into `ExampleResource`.

The `ExampleConfiguration.java` file loads the configuration. Because the properties are `private` members, getters and setters are required, but are omitted here for brevity. For `public` properties, no getters and setters are required for injection.

```
...output omitted...
import io.quarkus.arc.config.ConfigProperties;
```

```
@ConfigProperties(prefix = "example")
public class ExampleConfiguration {
 private String format;
 private boolean debugFlag = false; ①
 private Optional<String> unit;
...output omitted...
```

- ① In this case, in the configuration, the `debug-flag` variable will have its hyphen translated into setting the following letter into an uppercase, mapping it to `debugFlag`.

The behavior of the injection is the same as in the example using `ConfigProperty`. This means that the `format` property is still required and the `debugFlag` one has a default value of `false`. Additionally, because `unit` is optional, it does not have a value if not given.

An even shorter alternative style is to create an interface instead of a class.

```
...output omitted...
@ConfigProperties(prefix = "example")
public interface ExampleConfiguration {
 String getFormat();

 @ConfigProperty(defaultValue = "false")
 boolean getDebugFlag();

 Optional<String> getUnit();
}
```

You still need the `@ConfigProperty` annotation for `getDebugFlag()` to set a `defaultValue`, but the other two getters will be injected automatically. This interface will behave exactly the same as the class above.

The `ExampleResource` class can then use the configuration as follows.

```
...output omitted...
@Path("/example")
public class ExampleResource {
 @Inject
 ExampleConfiguration config;

 @GET
 @Produces(MediaType.TEXT_PLAIN)
 public String getEndpoint() {
 if (config.getDebugFlag()) {
 System.out.println("The format is: " + config.getFormat());
 System.out.println("The unit is: " + config.getUnit().orElse("Not
specified"));
 }
 return performComputation(config.getFormat(),
 config.getUnit().orElse(""));
 }
...output omitted...
```

`@ConfigProperties` supports nested objects.

```
@ConfigProperties(prefix = "example")
public interface ExampleConfiguration {
 String getFormat();
 @ConfigProperty(defaultValue = "false")
 boolean getDebugFlag();
 Optional<String> getUnit();
 ExampleUser getUser();
 public static interface ExampleUser {
 String getUsername();
 String getEmail();
 }
}
```

The above Java code would work with the following configuration, injecting an `ExampleUser` object into the `ExampleConfiguration` object, with the values found in the configuration.

```
example.format=One format for %s unit.
example.unit=meter
example.debug-flag=true
example.user.username=student
example.user.email= student@redhat.com
```

## Describing Configuration Profiles

Quarkus supports multiple configuration profiles. This allows multiple configurations for any object. For each, a different configuration will be chosen, depending on how the application is running.

By default, you have the following default configuration profiles available:

### **dev**

Activated when running the application in development mode.

### **test**

Activated when running the tests.

### **prod**

The default profile.

Properties with no preceding profile will be available to all profiles unless they are overridden with another value in the current profile configuration.

What follows is an example of an `application.properties` file, which would set the same configuration property for each of the three profiles.

```
example.user.username=student❶
example.user.email= student@redhat.com
%test.example.user.username=teststudent❷
%test.example.user.email= student@testing.com
%dev.example.user.username=devstudent❸
```

❶ The properties have no prefix, which means this is the default value for all profiles.

❷ The `%test` means that this value will override the default one in the `test` profile.

- ③ Notice that the `dev` profile does not set a value for `example.user.email`, which means that it will use the `prod` value for that variable.

## Custom Profiles

You can create your own custom profiles to further customize your configuration requirements. You just need to add a matching prefix in the configuration.

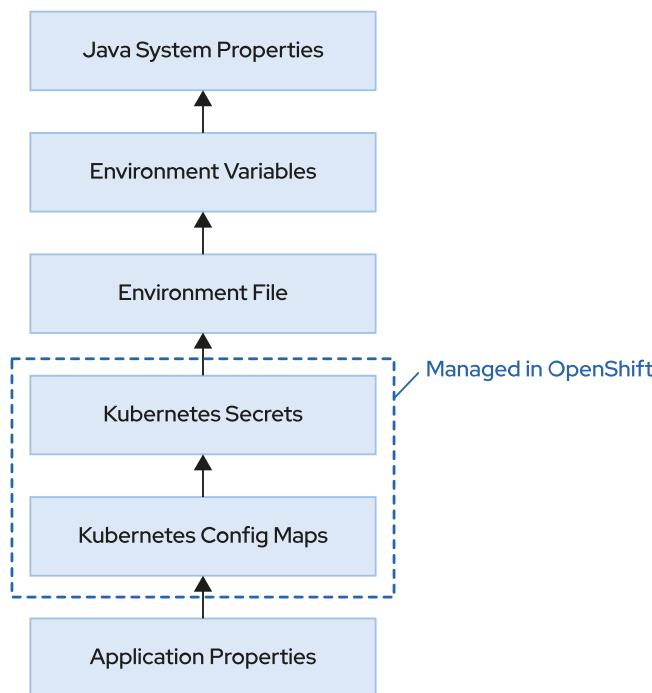
For example, for a profile with name `experiment`, the prefix is `%experiment`. When running the application, you must either pass the `quarkus.profile` system property or the `QUARKUS_PROFILE` environment variable and set it to `experiment`.

The following two examples are equivalent.

```
[user@host ~]$ mvn package -Dquarkus.profile=experiment
```

```
[user@host ~]$ QUARKUS_PROFILE=experiment mvn package
```

## Configuration Sources and Priorities



**Figure 4.1: Priority order in configuration sources**

The diagram shows some of the different configuration sources that can be used for setting variables. Priority increases from bottom to top.

If a property is set by multiple sources, then the value from the configuration source with the highest priority will override the other sources' values.

While Kubernetes ConfigMaps and Secrets usage is not covered in this course, they are both useful tools for production deployments.

## Other Configuration Sources

You have already learned about using `application.properties` as a configuration source. Here are some examples of using other sources available by default.

### Environment File

This option includes a file named `.env` in the root directory of the application. The main differences to `application.properties` are syntax and a higher priority.

Compared to `application.properties`, this option has the following syntax differences:

- All characters are expected to be in uppercase.
- Instead of using a dot (.) to indicate access, environment files use underscores (\_).
- Instead of a hyphen (-), an underscore replaces camel case in the Java code.
- Instead of a percentage sign (%), an underscore prefix indicates a profile.

The following is an example environment file, related to the examples above.

```
_DEV_EXAMPLE_USER_USERNAME=devstudentfromenv❶
EXAMPLE_DEBUG_FLAG=false❷
```

- ❶ The underscore at the beginning, followed by DEV indicates that the variable is set when running in the dev profile.
- ❷ This will override the `example.debug.flag` set in the `application.properties` file above. Notice that the hyphen changes to an underscore.

### Environment Variables

To get the same effect as the environment file, you have to prepend the environment variables to run the application. However, this also runs at a higher priority.

You also have the possibility of exporting the variables before running the command, such as in the following example:

```
[user@host ~]$ _DEV_EXAMPLE_USER_USERNAME=devstudentfromenv \
EXAMPLE_DEBUG_FLAG=false \
mvn compile quarkus:dev
```

Alternatively, you can export variables in a bash script and use the `source` command to set them. In that case, the name and location of the script is arbitrary.

### Java System Properties

*System Properties* have the highest priority of all default configuration sources. Their syntax is similar to the `application.properties` file, but are provided to the application as a parameter upon execution.

```
[user@host ~]$ mvn compile quarkus:dev -Dexample.debug.flag=true \
-Dexample.unit=yard
```

## Using Data Converters

Quarkus performs automatic conversions for values that are mapped to booleans, strings and integer values. However, you might want to automate conversions into objects of your own. For custom conversions, you can use MicroProfile's Converter interface.

Adapting the examples used earlier in this section, you could create your own converter for an `ExampleConfiguration.ExampleUser` class.

```
...output omitted...
@ConfigProperties(prefix = "example")
public interface ExampleConfiguration {
 ...output omitted...
 ExampleUser getUser();

 public static class ExampleUser {
 String username;
 String email;
 ...output omitted...
 }
}
```

If the desired value in the `application.properties` file has the two values contained in a single string with a colon as a separator, then it would require writing a custom converter implementation.

```
example.format=One format for %s unit.
example.unit=meter
example.debug-flag=false
example.user=student:student@redhat.com
```

The first step would be to create a new Java class that implements the `Converter` interface, using `ExampleConfiguration.ExampleUser` for the parameterization.

```
package org.acme.example;

import org.eclipse.microprofile.config.spi.Converter;

public class ExampleUserConverter implements
 Converter<ExampleConfiguration.ExampleUser> {
 @Override
 public ExampleConfiguration.ExampleUser convert(String value) {
 String[] values = value.split(":");
 var user = new ExampleConfiguration.ExampleUser();
 user.setUsername(values[0]);
 user.setEmail(values[1]);
 return user;
 }
}
```

This code converts a `String` by splitting it via the chosen separator, a colon, and creates an instance, sets its values, and returns the object.

After that, register the custom converter as a service. To do so, create the `src/main/resources/META-INF/services/` directory. Then, create a file named `org.eclipse.microprofile.config.spi.Converter`.

In the contents of the file, you can put the *Fully Qualified Name* (FQN) of your `Converter` implementation. Afterwards, the converter is registered and used for all `ExampleConfiguration.ExampleUser` objects.



## References

### Configuring your Quarkus applications

[https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.11/html-single/configuring\\_your\\_quarkus\\_applications/index](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11/html-single/configuring_your_quarkus_applications/index)

### Configuration For MicroProfile

<https://microprofile.io/project/eclipse/microprofile-config>

## ► Guided Exercise

# Injecting Configuration Data

In this exercise, you will modify an application that currently uses hard-coded values to configure using multiple configuration sources and injecting values into the application.

## Outcomes

You should be able to configure your application using multiple configuration sources, avoiding hard-coded values, as well as improving deployment of a microservice to different environments. You should also be able to identify the resulting value of the variable when setting it from multiple configuration sources.

## Before You Begin

This command ensures that the starting application is on your workstation. The code of the application is located in the /home/student/D0378/labs/apps-inject/expense-service folder.

```
[student@workstation ~]$ lab apps-inject start
```

## Instructions

- ▶ 1. Open VSCode and open the folder /home/student/D0378/labs/apps-inject/expense-service.
  - 1.1. Open the VSCode application on your workstation.
  - 1.2. Click **File > Open Folder** and browse to the /home/student/D0378/labs/apps-inject/expense-service/ directory.
  - 1.3. Click the **OK** button.
- ▶ 2. Analyze the static values of the `ExpenseResource` class. Run the application and hit the `create()` endpoint. Check its functionality using the Swagger UI.
  - 2.1. Open the `src/main/java/org/acme/rest/json/ExpenseResource.java` file. It contains two properties of type `BigDecimal`.

```
...output omitted...
public class ExpenseResource {

 private BigDecimal maxSingleExpenseAmount = new BigDecimal(5000);
 private BigDecimal maxMonthlyExpenseAmount = new BigDecimal(50000);

...output omitted...
 public Expense create(final Expense expense) {
 if (expense.amount.compareTo(maxSingleExpenseAmount) > 0) {
 throw new WebApplicationException(
 String.format("Expense amount is larger than the limit (%s)",
maxSingleExpenseAmount),
```

```

 Response.Status.NOT_ACCEPTABLE);
 }
...output omitted...
 if
(currentTotalAmount.add(expense.amount).compareTo(maxMonthlyExpenseAmount) > 0) {
 throw new WebApplicationException(
 String.format("If expense is accepted, it would exceed the monthly
expense limit (%s)", maxMonthlyExpenseAmount),
 Response.Status.NOT_ACCEPTABLE);
...output omitted...

```

- 2.2. Click View > Command Palette, type Quarkus and select Quarkus: Debug current Quarkus project.
- 2.3. Open up a browser and navigate to the Swagger UI: <http://localhost:8080/swagger-ui>.
- 2.4. Click POST /expenses.
- 2.5. Click Try it out.
- 2.6. Put the following data as the Request body.

```
{
 "amount": 3500,
 "name": "Medical Insurance",
 "paymentMethod": "DEBIT_CARD"
}
```

- 2.7. Click Execute. The response should have a Code of 200 and a Response body like the following:

```
{
 "id": ...output omitted...,
 "amount": 3500,
 "creationDate": ...output omitted...,
 "name": "Medical Insurance",
 "paymentMethod": "DEBIT_CARD",
 "uuid": ...output omitted...
}
```

- ▶ 3. Change all hard-coded properties from the ExpenseResource class and set them with ConfigProperty. Modify the application.properties file to set their values. Using Swagger UI, invoke the endpoint to see the changes.
- 3.1. Create the String properties singleLimitErrorMsg and monthlyLimitErrorMsg in the ExpenseResource class. Within the create method, use the new properties in the calls to String.format.

```

...output omitted...
public class ExpenseResource {

 private BigDecimal maxSingleExpenseAmount = new BigDecimal(5000);

```

```

private BigDecimal maxMonthlyExpenseAmount = new BigDecimal(50000);
private String singleLimitErrorMsg = "Expense amount is larger than the limit
(%s)";
private String monthlyLimitErrorMsg = "If expense is accepted, it would exceed
the monthly expense limit (%s)";

...output omitted...
public Expense create(final Expense expense) {
 if (expense.amount.compareTo(maxSingleExpenseAmount) > 0) {
 throw new WebApplicationException(
 String.format(singleLimitErrorMsg, maxSingleExpenseAmount),
 Response.Status.NOT_ACCEPTABLE);
 }
...output omitted...
 if
(currentTotalAmount.add(expense.amount).compareTo(maxMonthlyExpenseAmount) > 0) {
 throw new WebApplicationException(
 String.format(monthlyLimitErrorMsg, maxMonthlyExpenseAmount),
 Response.Status.NOT_ACCEPTABLE);
}
...output omitted...

```

- 3.2. Import `org.eclipse.microprofile.config.inject.ConfigProperty` and annotate the four properties. Name them by replacing uppercase letters with a dash (-), followed by the letter in lowercase. Keep the previously set values as the default values for the `singleLimitErrorMsg` and `monthlyLimitErrorMsg` properties, but remove the ones for the `maxSingleExpenseAmount` and `maxMonthlyExpenseAmount`.

```

...output omitted...
public class ExpenseResource {
 @ConfigProperty(name = "max-single-expense-amount")
 private BigDecimal maxSingleExpenseAmount;

 @ConfigProperty(name = "max-monthly-expense-amount")
 private BigDecimal maxMonthlyExpenseAmount;

 @ConfigProperty(name = "single-limit-error-msg", defaultValue = "Expense
amount is larger than the limit (%s)")
 private String singleLimitErrorMsg;

 @ConfigProperty(name = "monthly-limit-error-msg", defaultValue = "If expense
is accepted, it would exceed the monthly expense limit (%s)")
 private String monthlyLimitErrorMsg;
...output omitted...

```

- 3.3. Add values for `max-single-expense-amount` and `max-monthly-expense-amount` to the `src/main/resources/application.properties` file. Set the first to 4000 and the second to 7500.

```

...output omitted...
max-single-expense-amount=4000
max-monthly-expense-amount=7500

```

- 3.4. Go back to the Swagger UI in the web browser. Change the Request body so the content is equal to the following snippet, click **Execute**, and verify that you get an HTTP response with status code 406 saying Error : Not Acceptable.

```
{
 "amount": 4001,
 "name": "Medical Insurance",
 "paymentMethod": "DEBIT_CARD"
}
```

- 3.5. Change the Request body so that the amount value is 4000 and click **Execute**. Verify that you get an HTTP response with status code 200. Also verify that the Response body contains a new Expense object that looks like the following:

```
{
 "id": ...output omitted...
 "amount": 4000,
 "creationDate": ...output omitted...
 "name": "Medical Insurance",
 "paymentMethod": "DEBIT_CARD",
 "uuid": ...output omitted...
}
```

- 3.6. Without changing the Request body, click **Execute** again. Verify that you get a HTTP response with status code 406 saying Error : Not Acceptable.
- 3.7. Go back to VSCode and view the Terminal tab in the bottom pane. Find the stack trace from the last request received. At the top of the stack trace, you should see a message similar to the following:

```
ERROR [org.jbo.res.res.i18n] (executor-thread-1) RESTEASY002010: Failed to execute: javax.ws.rs.WebApplicationException: If expense is accepted, it would exceed the monthly expense limit (7500)
```

- 3.8. Go to the application.properties file and change the variable max-monthly-expense-amount to be 9001. Then, go back to Swagger UI and click **Execute** two more times.

You should get a HTTP response with status code 200 because the sum total of all transactions is less than 9001.

- 3.9. Click **Execute** once more. Because the total transaction amount now exceeds 9001, you will see the 406 and Error : Not Acceptable message.

Notice that you have not reloaded the application since starting it. Your code changes are still applied due to live reloading.

- 4. Create an ExpenseConfiguration interface with the four properties in ExpenseResource. Use the @ConfigurationProperties annotation to inject the configuration, using expense as the prefix. Modify the application.properties file to use the expense prefix. Adapt the ExpenseResource class to use the ExpenseConfiguration class by injecting an object and then using its values.
- 4.1. Create a new file src/main/java/org/acme/rest/json/ExpenseConfiguration.java. Do this in VSCode by opening the Explorer and

right-clicking json at the end of `java/org/acme/rest/json`. Click **New File** and enter the name `ExpenseConfiguration.java`.

- 4.2. Change `ExpenseConfiguration` to be an interface. Import the annotation `io.quarkus.arc.config.ConfigProperties` and annotate the interface with the prefix `expense`. Add getter method headers for all four properties.

When you are finished, the file should look like the following:

```
package org.acme.rest.json;

import io.quarkus.arc.config.ConfigProperties;
import java.math.BigDecimal;

@ConfigProperties(prefix = "expense")
public interface ExpenseConfiguration {
 BigDecimal getMaxSingleExpenseAmount();
 BigDecimal getMaxMonthlyExpenseAmount();
 String getsSingleLimitErrorMsg();
 String getMonthlyLimitErrorMsg();
}
```

- 4.3. Add the `org.eclipse.microprofile.config.inject.ConfigProperty` annotation to the getters for the String properties and provide default values.

```
package org.acme.rest.json;

import io.quarkus.arc.config.ConfigProperties;
import java.math.BigDecimal;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@ConfigProperties(prefix = "expense")
public interface ExpenseConfiguration {
 BigDecimal getMaxSingleExpenseAmount();
 BigDecimal getMaxMonthlyExpenseAmount();

 @ConfigProperty(defaultValue = "Expense amount is larger than the limit (%s)")
 String getsSingleLimitErrorMsg();

 @ConfigProperty(defaultValue = "If expense is accepted, it would exceed the
monthly expense limit (%s)")
 String getMonthlyLimitErrorMsg();
}
```



### Note

Be sure to import `@ConfigProperty` from the Eclipse MicroProfile package. Using one provided elsewhere can cause runtime errors that are difficult to troubleshoot.

- 4.4. Open the `application.properties` file and update the variables pertaining to the `ExpenseConfiguration` interface to have an `expense.` prefix.

```
quarkus.hibernate-orm.database.generation=drop-and-create
expense.max-single-expense-amount=4000
expense.max-monthly-expense-amount=9001
```

- 4.5. Open the `ExpenseResource.java` file and delete all four properties. Add an `ExpenseConfiguration configuration` property and annotate it with `javax.inject.Inject`.

```
...output omitted...
public class ExpenseResource {
 @Inject ExpenseConfiguration configuration;

 @GET
 public List<Expense> list() {
 return Expense.listAll();
 }
...output omitted...
```

- 4.6. Within the `ExpenseResource.create` method, fix each of the broken references by calling the relevant getter method provided by the `configuration` object. When you are finished, the method should look like the following:

```
public Expense create(final Expense expense) {
 if (expense.amount.compareTo(
 configuration.getMaxSingleExpenseAmount()) > 0) {
 throw new WebApplicationException(
 String.format(configuration.getSingleLimitErrorMsg(),
 configuration.getMaxSingleExpenseAmount()),
 Response.Status.NOT_ACCEPTABLE);
 }
...output omitted...
 if (currentTotalAmount.add(expense.amount).compareTo(
 configuration.getMaxMonthlyExpenseAmount()) > 0) {
 throw new WebApplicationException(
 String.format(configuration.getMonthlyLimitErrorMsg(),
 configuration.getMaxMonthlyExpenseAmount()),
 Response.Status.NOT_ACCEPTABLE);
 }
...output omitted...
```

Optionally, remove the `org.eclipse.microprofile.config.inject.ConfigProperty`, as it is no longer needed.

- 4.7. Restart the debugger within VSCode to reload the configuration.

**Note**

Sometimes, live reloading can misbehave when making drastic changes to the configuration structure. To avoid this, you will need to manually restart the application.

Additionally, because you are using an H2 in-memory database, data is lost whenever you stop the application. You will need to submit multiple transactions before the maximum limit error is triggered.

- 4.8. Within the Swagger UI, click **Execute**. Verify the HTTP code is 200. If the HTTP code is 500, verify that your changes are correct and try again.
- ▶ 5. Stop the running application in VSCode. Run it again using the built-in terminal, changing the error message by using environment variables.
  - 5.1. Within VSCode, open the **Terminal** tab in the bottom pane and press **Ctrl+C**. This stops the running application process.
  - 5.2. In the header of the **Terminal** tab, click + (plus) to open a new prompt. Verify that it is open to the `/home/student/D0378/labs/apps-inject/expense-service` directory by running the following command:

```
[student@workstation ~]$ cd ~/D0378/labs/apps-inject/expense-service
```

- 5.3. Start the application, but this time add the `EXPENSE_SINGLE_LIMIT_ERROR_MSG` environment variable.

```
[student@workstation expense-service]$ EXPENSE_SINGLE_LIMIT_ERROR_MSG="Too big (%s)" mvn compile quarkus:dev
```

- 5.4. Within the Swagger UI, modify the **Request** body to increase the expense amount to `10000`, so it looks like this.

```
{
 "amount": 10000,
 "name": "Medical Insurance",
 "paymentMethod": "DEBIT_CARD"
}
```

- 5.5. Click **Execute** and go to your terminal to read the error message. At the top of the stack trace, you should find a similar looking message. It is important that it now says `Too big`, which is the message set by the environment variable.

```
...output omitted...
...output omitted... ERROR [org.jbo.res.res.i18n] (executor-thread-1)
RESTEASY002010: Failed to execute: javax.ws.rs.WebApplicationException: Too big
(4000)
 at org.acme.rest.json.ExpenseResource.create(ExpenseResource.java:41)
...output omitted...
```

- 5.6. Kill the application by pressing **Ctrl+C** while in the terminal.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab apps-inject finish
```

This concludes the guided exercise.

# Implementing Feature Toggles

## Objectives

After completing this section, you should be able to implement configurable feature toggles in microservices.

## Introducing Feature Toggles

In an Agile development process one of the key factors to success is the ability to continuously push to production. Having long-living feature branches is not recommended because of the problems they generate when merging. A popular solution is to control the visibility of a feature in development or in the testing phase using a toggle, allowing developers to easily enable or disable features.

These Feature Toggles, also referred to as Feature Flags or Feature Switches, control the access to a feature in a released component. Some possible reasons to disable or hide a feature are because they are still in development, need testing, or are waiting for a coordinated launch. With a Feature Toggle you can control the presence of new functionality, the visibility of this functionality in the user interface (UI), or the use of a new algorithm to use in an existing procedure.

The Feature Toggle is especially useful to control the visibility of a feature in the UI. It controls when the feature is available to the users. You have to choose how the UI determines if a Feature Toggle is activated or not. The two most common options are: if a specific endpoint is not accessible, or creating an endpoint, which returns the state of all feature toggles of the application. Use caution with this last option, because it exposes the internal state of development to the outside world.

You can combine this technique with Canary Releases, in which you make the feature available to a subset of users as an initial test.



### Warning

Beware that although you can use Feature Toggles to solve problems in the Agile development process, there is a tradeoff: you are introducing Technical Debt into your code in the form of numerous conditional blocks, which will amount to an enormous amount of complexity if carried on over time.

The solution to this problem is to only maintain a Feature Toggle whilst you are testing its correctness and viability. The moment you finish testing the feature, there are only two outcomes: promote it as part of the application or retire it. Either way, you should remove the conditional blocks because the feature is now a permanent part of the application or is completely retired.

## Feature Toggles with Quarkus

To implement Feature Toggles in Quarkus, use the MicroProfile Config annotations seen earlier in this chapter. In its simplest form, you create a Boolean configuration parameter and control the flow of the code depending on the value.

```

@ConfigProperty
boolean newFeatureEnabled;

public void sampleFunction() {

 // Old code

 if(newFeatureEnabled) {
 // New feature code
 }

}

```

Or, in the case of replacing an old functionality with a new one.

```

@ConfigProperty
boolean alternativeAlgoEnabled;

public void sampleFunction() {

 if(alternativeAlgoEnabled) {
 // New algorithm code
 } else {
 // Old algorithm code
 }

}

```

A common practice when using Feature Toggles is to centralize their values in a single control class. This approach reduces duplicated code when the same feature toggle is used in several locations in the code. Take advantage of the `io.quarkus.arc.config.ConfigProperties` annotation to simplify this features control class.

```

features.new-user-interface-enabled=true
features.alternative-algorithm-enabled=true

```

```

@ConfigProperties(prefix = "features")
public class Features {
 public boolean newUserInterfaceEnabled;
 public boolean alternativeAlgorithmEnabled;
}

```

```

@Inject
Features appFeatures;

public void sampleFunction() {

 if(appFeatures.alternativeAlgorithmEnabled) {
 // New algorithm code
 } else {
 // Old algorithm code
 }

}

```

```
}
```

```
}
```

One additional recommendation is to use the same features control class to create methods that return if a feature is enabled or not. This way, if a feature changes the way in which it determines if it is enabled or not, then the code only needs to adapt in one centralized point.

```
@ConfigProperties(prefix = "features")
public class Features {
 public boolean newUserInterfaceEnabled;
 public boolean alternativeAlgorithmEnabled;

 public boolean isFeatureAEnabled() {
 return newUserInterfaceEnabled && alternativeAlgorithmEnabled;
 }
}
```

```
@Inject
Features appFeatures;

public void sampleFunction() {

 if(appFeatures.isFeatureAEnabled()) {
 // New algorithm code
 } else {
 // Old algorithm code
 }
}
```



## References

### Feature Toggle

[https://en.wikipedia.org/wiki/Feature\\_toggle](https://en.wikipedia.org/wiki/Feature_toggle)

### Feature Toggles, by Martin Fowler

<https://martinfowler.com/articles/feature-toggles.html>

### Canary Release

<https://martinfowler.com/bliki/CanaryRelease.html>

## ► Guided Exercise

# Implementing Feature Toggles

In this exercise, you will learn how to use MicroProfile Config to add the subtract capability to the Quarkus Calculator using Feature Toggles.

## Outcomes

You should be able to add a new feature to an application and toggle it on and off.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the application Quarkus Calculator is ready for modification. The location of the source code of this application is `/home/student/D0378/labs/apps-feature/quarkus-calculator-monolith`.

```
[student@workstation ~]$ lab apps-feature start
```

## Instructions

- 1. To begin adding the feature to the application enter its directory and start development mode.

- 1.1. Enter the Quarkus Calculator directory.

```
[student@workstation ~]$ cd ~/D0378/labs/apps-feature/quarkus-calculator-monolith
```

- 1.2. Open the application in the IDE.

```
[student@workstation quarkus-calculator-monolith]$ codium .
```

- 1.3. To start the application in development mode bring up the Command Palette in the editor using `Ctrl+Shift+P` and type Quarkus. Select the option `Debug current Quarkus project`.

- 2. First, we are going to test that the application is not capable of handling subtraction operations.

```
[student@workstation quarkus-calculator-monolith]$ curl -w "\n" \
http://localhost:8080/solver/3-5
Unable to parse: 3-5
```

- 3. Add the subtraction feature, which is disabled by default.

- 3.1. Open the `SolverResource.java` file.

## 3.2. Add the new Regular Expression pattern.

```
static final Pattern addPattern = Pattern.compile("(.)\\\\+(.)");
static final Pattern subtractPattern = Pattern.compile("(.)\\\\-(.)");
```

## 3.3. Add the subtract Feature Toggle configuration knob.

```
import org.eclipse.microprofile.config.inject.ConfigProperty;
...output omitted...
static final Pattern subtractPattern = Pattern.compile("(.)\\\\-(.)");

@ConfigProperty(name = "features.subtract-enabled")
private boolean subtractEnabled;
```

## 3.4. Add the new subtract function after the multiply one.

```
public Float multiply(String lhs, String rhs){
 log.info("Multiplying {} to {}", lhs, rhs);
 return solve(lhs)*solve(rhs);
}

public Float subtract(String lhs, String rhs) {
 log.info("Subtracting {} from {}", rhs, lhs);
 return solve(lhs)-solve(rhs);
}
```

## 3.5. Add the new subtract functionality after the add matching block.

```
Matcher addMatcher = addPattern.matcher(equation);
if (addMatcher.matches()) {
 return add(addMatcher.group(1),addMatcher.group(2));
}

if (subtractEnabled) {
 Matcher subtractMatcher = subtractPattern.matcher(equation);
 if (subtractMatcher.matches()) {
 return subtract(subtractMatcher.group(1),subtractMatcher.group(2));
 }
}
```

## 3.6. Add the configuration property in the application.properties file, disabled by default.

```
features.subtract-enabled=false
```

## ▶ 4. Confirm that the application continues to give an error when trying to use a subtraction.

```
[student@workstation quarkus-calculator-monolith]$ curl -w "\n" \
 http://localhost:8080/solver/3-5
Unable to parse: 3-5
```

- ▶ 5. Activate the subtraction feature by changing the configuration value in the `application.properties`.

```
features.subtract-enabled=true
```

- ▶ 6. Verify that the application properly handles a subtraction.

```
[student@workstation quarkus-calculator-monolith]$ curl -w "\n" \
http://localhost:8080/solver/3-5
-2.0
```

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab apps-feature finish
```

This concludes the guided exercise.

# Connecting Multiple Services

## Objectives

After completing this section, you should be able to use service discovery to connect multiple services to each other.

## Enabling a REST Client in Quarkus

A microservices application is a composition of multiple services. In a monolithic application, a service invokes another service by using a language-level method. In a microservice-based application, each microservice needs a way to locate other services to perform intra-service calls. It is especially problematic if your application runs in a cloud environment because the locations and number of instances of a service change frequently and are not predictable. To solve this problem you must use service discovery.

## Implementing the Client-side API

In previous chapters, we introduced how to create microservices that serve responses using the REST protocol. The `quarkus-rest-client` extension adds to the project the capabilities to create REST clients given just the service API definition and a few annotations. The `quarkus-rest-client` extension is based on RESTEasy, an open source implementation of the `MicroProfile Rest Client` specification. You can add this extension to the project using standard approaches such as the `mvn quarkus:add-extension` command.

Once the extension is available, you must obtain the service API definition. This API does not need to be the exact API for the service, but a sufficient subset of the endpoints offered by the service. Declare the API as an interface class, and annotate the interface with the `org.eclipse.microprofile.rest.client.inject.RegisterRestClient` annotation. The API class also needs to use the `javax.ws.rs.Path` to declare the relative path on the server.

```
@Path("/relative_path")
@RegisterRestClient
public interface MyServiceAPI {
 ...output omitted...
}
```

Each service endpoint available in the API maps to a method in the API class. Use the `javax.ws.rs.Path`, and `javax.ws.rs.Produces` annotations to declare the relative path of the endpoint and the response media type. Like you do when defining the server, use the `javax.ws.rs.GET`, `javax.ws.rs.PUT`, and friends to define the HTTP method for the API endpoint.

```
...output omitted...
@Path("/relative_path")
@RegisterRestClient
public interface MyServiceAPI {
 @GET
 ...output omitted...
}
```

```
@Path("{endpoint}")
@Produces(MediaType.TEXT_PLAIN)
String endpointMethod(@PathParam("paramName") String param);
}
```

Review more details about JAX-RS annotations and serialization in *Implementing a JSON Microservice with CDI and JAX-RS*.

## Using the REST Client

Once the service API is available, the next step is making use of it. Quarkus generates the REST client code at compilation, so you do not need to write a single line of code. The lifecycle of the clients is managed by Arc, a dependency injection framework, which simplifies usage even more.

To use a REST client with the API defined previously, you must create an attribute in your class and annotate it with both the `javax.inject.Inject` and `org.eclipse.microprofile.rest.client.inject.RestClient` annotations.

```
@ApplicationScoped // or @Path, or any other annotation that defines a CDI bean.
public class SomeBean {

 @Inject @RestClient MyServiceAPI myServiceClient
 ...output omitted...
}
```



### Note

Arc needs to manage the class where it injects the rest client. Because Quarkus treats all JAX-RS resources as CDI beans, annotating the class with `javax.ws.rs.Path` enables it as a CDI bean and allows client injection.

Use the REST client instance as any instance that implements the API interface. Quarkus transforms calls to local methods of this instance to REST HTTP calls.

```
@ApplicationScoped // or @Path, or any other annotation that defines a CDI bean.
public class SomeBean {

 @Inject
 @RestClient
 MyServiceAPI myServiceClient
 ...output omitted...
 myServiceClient.endpointMethod(param);
}
```

## Configuring the REST Client

Configuration for the generated REST clients is centralized in the `application.properties` file. The only mandatory entry for the clients to be usable is the URL of the destination service. To compose the name of the property, prepend the FQN (fully qualified name) of the API interface class to the `/mp-rest/url` keyword.

```
com.mycompany.MyServiceAPI/mp-rest/url=http://HOST:PORT
```

Just like any other property of Quarkus configuration, that property accepts profile prefixes.

```
%dev.com.mycompany.MyServiceAPI/mp-rest/url=http://HOST:PORT
```

The `_API_FQN_/mp-rest/scope` property defines the injection scope for the generated clients. If not provided, the default value is `javax.inject.Dependent`, meaning that Arc binds the lifecycle of the client to the lifecycle of the containing bean.

You can control other REST server and client features by properties using the `quarkus.resteasy` prefix. Refer to Quarkus documentation for an exhaustive list of the available options.

## Service Discovery in Red Hat OpenShift Container Platform

To handle service discovery, OpenShift Container Platform provides the `Service` resource. An OpenShift service holds a DNS name representing a set of pods (or external servers). The service identifies a set of replicated pods to proxy the connections it receives to the pods. Backing pods can be added to or removed from an OpenShift service arbitrarily. In contrast, the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address.

An OpenShift service is assigned an internal IP address, a port number, and a DNS name. The DNS name can be exposed externally by an OpenShift Route, but internal service discovery does not require this.

## Service Discovery by Environment Variable Injection

OpenShift Container Platform automatically injects environment variables for the host name and port number into other pods. For each OpenShift service inside an OpenShift Container Platform project, two environment variables are automatically defined and injected into the containers for all of the pods running inside the project. For an example service with the name of `calculator`, the environment variables would be:

- `CALCULATOR_SERVICE_HOST` is the service IP address.
- `CALCULATOR_SERVICE_PORT` is the service TCP port.



### Note

The service name is changed to comply with DNS naming restrictions: all letters are capitalized, and underscores (`_`) are replaced by dashes (`-`).

In Quarkus, you can use these environment variables directly in the `application.properties` file. From a client class named `com.example.RestClient`, accessing the `calculator` service must be configured as:

```
com.example.RestClient/mp-rest/url=http://${CALCULATOR_SERVICE_HOST}:
${CALCULATOR_SERVICE_PORT}
```

## DNS-based Service Discovery

Another way to discover a service from a pod is by using the RHOC P internal DNS server, which is visible only to pods. Each service is dynamically assigned an SRV record with a Fully Qualified Domain Name (FQDN) of the form.

```
SVC_NAME.PROJECT_NAME.svc
```

If the service belongs to the same project as the originating query, then you can skip the `.PROJECT_NAME.svc` part and just use the target service name as the URL host.

The service is available only for applications deployed in the same cluster. Use OpenShift routes if you need to access the service from outside the cluster.

When developing Quarkus applications, the usual approach is to use OpenShift Container Platform service discovery in the `%prod` configuration profile, always using `localhost` for the `%dev` profile. This approach typically requires using different ports for different microservices when locally testing.

```
%dev.quarkus.http.port=8085 ①
%prod.quarkus.http.port=8080 ②
%prod.com.mycompany.MyServiceAPI.AdderService/mp-rest/url=http://adder:8080③
%dev.com.mycompany.MyServiceAPI.AdderService/mp-rest/url=http://localhost:8084④
```

- ① Each microservice defines a different port when running locally or in the `%dev` profile.
- ② This line is not needed, as 8080 is the default port.
- ③ In the production profile, discover other services by just the service name.
- ④ In the development profile, other services run in `localhost` in dedicated ports.



### References

#### RESTEasy

<https://resteasy.github.io/>

#### Eclipse MicroProfileRest Client

<https://github.com/eclipse/microprofile-rest-client>

## ► Guided Exercise

# Connecting Multiple Services

In this exercise, you will break an existing monolithic application into several microservices, and configure communications for both local and remote deployment in OpenShift Container Platform.

## Outcomes

You should be able to establish REST communication between Quarkus microservices and configure them to discover other available microservices in OpenShift Container Platform.

## Before You Begin

To use your provided OpenShift Container Platform instance, you need the appropriate credentials supplied by your Red Hat Training Online Learning environment. Review the *Guided Exercise: Verifying OpenShift Credentials* section if you need guidance for finding your credentials.

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command prepares the source code for the Quarkus calculator application in your workstation and creates a project in your OpenShift Container Platform instance.

```
[student@workstation ~]$ lab apps-connect start
```

## Instructions

- 1. Review the Quarkus Calculator monolithic application. The source code is in the `~/D0378/labs/apps-connect/` folder. Use `VSCodium` to open the folder as a new Quarkus application.

```
[student@workstation ~]$ codium ~/D0378/labs/apps-connect/
```

Open the file `solver/src/main/java/com/redhat/training/SolverResource.java` and review the code. This class implements a simple equation resolver based on regular expressions. Allowed equations compose only of decimal numbers, sums, and multiplications. Compound operations are resolved recursively.

In this exercise, you will break this monolithic application into three microservices.

### **solver**

Evaluates that the equation is a decimal number, or a composition of sums and multiplications. Most of the functionality on the `solver` microservice will remain, except arithmetic operations.

### **adder**

Gets two equations and returns the sum of their results. Relies on the `solver` microservices to solve both sides of the sum.

**multiplier**

Gets two equations and returns the product of their results. Relies on the solver microservices to solve both sides of the multiplication.

- ▶ 2. When breaking a monolith into smaller services the first thing to do is define the API for the new services. Quarkus' approach for creating service APIs is to create `interface` classes and annotate them with the `@Path` annotation.

In this exercise you will be using those same interfaces as the client definition. To do this, you will be using the `@RegisterRestClient` annotation in the interface from the `rest-client` extension. Add this extension to the solver application by using the following command.

```
[student@workstation ~]$ cd ~/DO378/labs/apps-connect/solver
[student@workstation solver]$ mvn quarkus:add-extension -Dextension=rest-client
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.redhat.training:quarkus-solver >-----
[INFO] Building quarkus-solver 1.0-SNAPSHOT
[INFO] -----[jar]-----
[INFO]
[INFO] --- quarkus-maven-plugin:1.11.7.Final-redhat-00009:add-extension (default-cli) @ quarkus-solver ---
...output omitted...
Adding extension io.quarkus:quarkus-rest-client
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 19.172 s
[INFO] Finished at: 2020-09-21T16:18:36-04:00
[INFO] -----
```

Alternatively, you can use the Quarkus: Add extensions to the current project command from the View > Command Palette menu.

Add the `@RegisterRestClient` annotation to the `com.redhat.training.service.SolverService` interface.

```
...output omitted...
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
...output omitted...
@Path("/solver")
@RegisterRestClient
public interface SolverService {
 @GET
 @Path("{equation}")
 @Produces(MediaType.TEXT_PLAIN)
 Float solve(@PathParam("equation") String equation);
}
...output omitted...
```

- 3. Define interfaces for the two new microservices: AdderService and MultiplierService. Add those interfaces to the `com.redhat.training.service` package. Both interfaces have a single method, based on the already existing `add` and `multiply` functions.

Add the `@Path` annotation to the interface to define the root path for the service endpoints. Use `/adder` as the path for the adder service and `/multiplier` for the `multiplier`. Annotate the class with the `@RegisterRestClient` annotation. You will require this annotation to automatically create REST clients later in this exercise.

Use the `@GET` annotation to indicate that both methods must respond to GET HTTP requests. Both methods will respond to the same path where the first section is the first parameter, and the second section is the second parameter. Use the `@Path` annotation to define this path. For clarity, we will name `lhs` (left hand side) to the first parameter and `rhs` (right hand side) to the second parameter. Use the `@Produces` annotation with the `MediaType.TEXT_PLAIN` parameter to indicate that both methods will produce a response in plain text.

Resulting interfaces should look like the following.

```
...output omitted...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.MediaType;
import import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
...output omitted...
@Path("/adder")
@RegisterRestClient
public interface AdderService {
 @GET
 @Path("/{lhs}/{rhs}")
 @Produces(MediaType.TEXT_PLAIN)
 Float add(@PathParam("lhs") String lhs, @PathParam("rhs") String rhs);
}
```

```
...output omitted...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.MediaType;
import import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
...output omitted...
@Path("/multiplier")
@RegisterRestClient
public interface MultiplierService {
 @GET
 @Path("/{lhs}/{rhs}")
 @Produces(MediaType.TEXT_PLAIN)
 Float multiply(@PathParam("lhs") String lhs, @PathParam("rhs") String rhs);
}
```

You can review your results in the solution files `~/D0378/solutions/apps-connect/solver/src/main/java/com/redhat/training/service/AdderService.java` and `~/D0378/solutions/apps-connect/solver/src/main/java/com/redhat/training/service/MultiplierService.java`

- 4. Create two new Quarkus projects, which will contain the two new microservices: `adder` and `multiplier`. Use the `mvn io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create` command to generate the project skeleton automatically. Make sure you execute the following commands from the `~/D0378/solutions/apps-connect` folder and that you use the following information to create your projects.

```
[student@workstation solver]$ cd ~/D0378/labs/apps-connect
[student@workstation apps-connect]$ mvn \
 io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create \
 -DprojectGroupId=com.redhat.training \
 -DprojectArtifactId=adder \
 -DplatformGroupId=com.redhat.quarkus \
 -DplatformVersion=1.11.7.Final-redhat-00009 \
 -DclassName="com.redhat.training.AdderResource" \
 -Dpath="/adder" \
 -Dextensions="rest-client"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.apache.maven:standalone-pom >-----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----[pom]-----
[INFO]
[INFO] --- quarkus-maven-plugin:1.11.7.Final-redhat-00009:create (default-cli) @
 standalone-pom ---
Creating a new project in /home/student/D0378/labs/apps-connect/adder
Adding extension io.quarkus:quarkus-rest-client
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

```
[student@workstation apps-connect]$ mvn \
 io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create \
 -DprojectGroupId=com.redhat.training \
 -DprojectArtifactId=multiplier \
 -DplatformGroupId=com.redhat.quarkus \
 -DplatformVersion=1.11.7.Final-redhat-00009 \
 -DclassName="com.redhat.training.MultiplierResource" \
 -Dpath="/multiplier" \
 -Dextensions="rest-client"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.apache.maven:standalone-pom >-----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----[pom]-----
[INFO]
[INFO] --- quarkus-maven-plugin:1.11.7.Final-redhat-00009:create (default-cli) @
 standalone-pom ---
Creating a new project in /home/student/D0378/labs/apps-connect/multiplier
Adding extension io.quarkus:quarkus-rest-client
...output omitted...
```

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

Verify that value for the `-Dpath` parameter matches the value for the `Path` annotation you previously added to the API of the services.

**Note**

You can also use the Quarkus: Generate a Quarkus project entry from the command palette, but this wizard does not define the path parameter. If you decide to do so, then edit the `Path` annotation from the generated classes to match the provided values.

We are not covering tests here, so just remove the `src/test` folder from both projects.

To save yourself from accidental typos or copying errors, the `~/D0378/solutions/apps-connect/create-projects.sh` script creates the project and deletes the `src/test` folders for you.

- ▶ 5. Move the body for the `add` and `multiply` functions from the `Solver` service to the appropriate microservices, replacing the sample "hello" method. To be precise, move the `com.redhat.training.SolverResource.add(String lhs, String rhs)` method to the `adder/src/main/java/com/redhat/training/AdderResource.java` file, and the `com.redhat.training.SolverResource.multiply` method to the `multiplier/src/main/java/com/redhat/training/MultiplierResource.java` file.  
Replace the function signature with the one from the same method in the appropriate service APIs: `com.redhat.training.service.AdderService` and `com.redhat.training.service.MultiplierService`. Make sure you also copy the annotations attached to the method in the interface. You also must prepend the `public` keyword to the method signature.  
Add the appropriate `org.slf4j.Logger` log attribute to both classes. You can copy the logger definition from the `SolverResource` class, however you then must update the class provided to the `getLogger` method.  
After all updates, the `AdderResource` and `MultiplierResource` must look like the following.

```
...output omitted...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...output omitted...
@Path("/adder")
public class AdderResource {
 final Logger log = LoggerFactory.getLogger(AdderResource.class);
 @GET
 @Path("/{lhs}/{rhs}")
```

```
@Produces(MediaType.TEXT_PLAIN)
public Float add(@PathParam("lhs") String lhs, @PathParam("rhs") String rhs) {
 log.info("Adding {} to {}", lhs, rhs);
 return solve(lhs)+solve(rhs);
}

...
...output omitted...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...output omitted...
@Path("/multiplier")
public class MultiplierResource {
 final Logger log = LoggerFactory.getLogger(MultiplierResource.class);
 @GET
 @Path("/{lhs}/{rhs}")
 @Produces(MediaType.TEXT_PLAIN)
 public Float multiply(
 @PathParam("lhs") String lhs, @PathParam("rhs") String rhs) {
 log.info("Multiplying {} to {}", lhs, rhs);
 return solve(lhs)*solve(rhs);
 }
}
```

- 6. Share service definition between projects. Copy the services package, where you previously defined the API for the services, from the `solver` project to the `adder` and the `multiplier` projects. The package and class names must remain the same.

```
[student@workstation apps-connect]$ cp -r \
solver/src/main/java/com/redhat/training/service/ \
adder/src/main/java/com/redhat/training
[student@workstation apps-connect]$ cp -r \
solver/src/main/java/com/redhat/training/service/ \
multiplier/src/main/java/com/redhat/training
```



### Note

Using the same interface as the service definition and as the client definition and copying code between services is usually discouraged because it creates entanglement between microservices, which should be completely independent. Each service should contain its own API and define its view of the API for other services.

In some cases, when the API of the services is extremely stable, developers can create a shared artifact with the APIs of all services and use it for defining the clients. However, this creates a dependency between microservices that should be avoided.

In this exercise, we are reusing the service definition and APIs for the sake of simplicity.

- 7. Enforce the resource classes to follow the API defined for them by declaring the resource class to implement the API. This is not a mandatory step but ensures at compile time that the service follows the API as declared, avoiding misalignment between clients and servers.

```
import com.redhat.training.service.AdderService;

@Path("/adder")
public class AdderResource implements AdderService {
```

```
import com.redhat.training.service.MultiplierService;

@Path("/multiplier")
public class MultiplierResource implements MultiplierService
```

- 8. Inject into each service the rest client to connect the other services. Use the `@Inject` and the `@RestClient` annotations to inject the automatically generated client. Use the appropriate API interface for the service as the type of the attribute.

The `SolverResource` service needs a client for both the `AdderService` and the `MultiplierService` services. The `AdderService` and the `MultiplierService` only need a single client for the `SolverService`.

The following snippets show the client attributes for each service.

```
...output omitted...
import javax.inject.Inject;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import com.redhat.training.service.AdderService;
import com.redhat.training.service.MultiplierService;
import com.redhat.training.service.SolverService;
...output omitted...
public class SolverResource implements SolverService {
...output omitted...
@Inject
@RestClient
AdderService adderService;

@Inject
@RestClient
MultiplierService multiplierService;
...output omitted...
```

```
...output omitted...
import javax.inject.Inject;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import com.redhat.training.service.AdderService;
import com.redhat.training.service.SolverService;
...output omitted...
public class AdderResource implements AdderService {
...output omitted...
@Inject
@RestClient
SolverService solverService;
...output omitted...
```

```
...output omitted...
import javax.inject.Inject;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import com.redhat.training.service.MultiplierService;
import com.redhat.training.service.SolverService;
...output omitted...
public class MultiplierResource implements MultiplierService {
...output omitted...
```

```
@Inject
@RestClient
SolverService solverService;
...output omitted...
```

- ▶ 9. Services, in the current status, are still trying to use local function calls for the `add`, `multiply` and `solve` methods. Replace those local function calls by REST client calls. Simply prefix local calls with the name of the appropriate injected service.

In the `SolverResource` replace the call to the `add` function with a call to the `adderService.add` function, and the call to the `multiply` function with a call to the `multiplierService.multiply` function. In both the `AdderResource` and the `MultiplierResource`, replace the calls to the `solve` function with calls to the `solverService.solver` function.

```
public class SolverResource implements SolverService {
...output omitted...
 public Float solve(@PathParam("equation") String equation) {
...output omitted...
 return adderService.add(addMatcher.group(1), addMatcher.group(2));
...output omitted...
 return multiplierService.multiply(multiplyMatcher.group(1),
multiplyMatcher.group(2));
...output omitted...
```

```
public class MultiplierResource implements MultiplierService {
...output omitted...
 public Float multiply(@PathParam("lhs") String lhs, @PathParam("rhs") String
rhs) {
...output omitted...
 return solverService.solve(lhs)*solverService.solve(rhs);
...output omitted...
```

```
public class AdderResource implements AdderService {
...output omitted...
 public Float add(@PathParam("lhs") String lhs, @PathParam("rhs") String rhs) {
...output omitted...
 return solverService.solve(lhs)+solverService.solve(rhs);
...output omitted...
```

Those last changes resolve the compilation issues in all the services. If you are still seeing compilation problems, then review the `import` statements on your code and the previous steps.

- ▶ 10. When running microservices locally for development purposes, all services are exposed in the same host (`localhost`). To avoid conflicts between services, each one must bind to a different port. Use the `quarkus.http.port` entry in the `application.properties` file to assign the port 8080 to the `solver`, the port 8081 to the `added`, and the port 8082

to the **multiplier**. When running in a production environment, each microservice runs in an independent container or host, so no port conflicts arise.

Update the `src/main/resources/application.properties` file of each service to define the ports as explained.

```
quarkus.application.name=solver
%dev.quarkus.http.port=8080
```

```
quarkus.application.name=addler
%dev.quarkus.http.port=8081
```

```
quarkus.application.name=multiplier
%dev.quarkus.http.port=8082
```

► **11.** Configure the REST clients to reach other microservices.

In local development, the host for the URL of all microservices is `localhost`, while the port should be the one just defined in the previous step.

In this exercise, you will be using OpenShift Container Platform as the production environment. OpenShift Container Platform provides a service discovery method where the URL of all services is resolved by using the service name. That is, if the service is named `solver`, then the URL of the host for this service will be `http://solver`. In the previous step, we declared that all services will be using the port 8080 in production. Hence, the complete URL to find the `solver` service in production is `http://solver:8080`.

To configure which URL uses each REST client, MicroProfile defines a property named after the fully qualified name (FQN) of the API for the service, followed by the `mp-rest/url` suffix. Update the `src/main/resources/application.properties` file for each service to add the URL of the other services. For simplicity, you can add the following snippet to the configuration of all three microservices.

```
%prod.com.redhat.training.service.AdderService/mp-rest/url=http://adder:8080
%prod.com.redhat.training.service.SolverService/mp-rest/url=http://solver:8080
%prod.com.redhat.training.service.MultiplierService/mp-rest/url=http://
multiplier:8080

%dev.com.redhat.training.service.SolverService/mp-rest/url=http://localhost:8080
%dev.com.redhat.training.service.AdderService/mp-rest/url=http://localhost:8081
%dev.com.redhat.training.service.MultiplierService/mp-rest/url=http://
localhost:8082
```

In case you encounter problems related to the configuration of your microservices you can compare your solution with the files provided at `~/D0378/solutions/apps-connect/SERVICE_NAME/src/main/resources/application.properties`

► **12.** Now your monolithic solver service is broken into three microservices. Start up all three services locally, and use `curl` to test the correct functioning of the services.

12.1. Start the three services in development mode in separate terminals. If you are using VSCode terminal then it is useful to use the **Terminal > Split Terminal** feature to have the three terminal windows side by side.

```
[student@workstation apps-connect]$ cd solver; mvn quarkus:dev
[INFO] Scanning for projects...
...output omitted...
2020-09-22 05:34:11,753 INFO [io.quarkus] (main) Profile dev activated. Live
Coding activated.
2020-09-22 05:34:11,753 INFO [io.quarkus] (main) Installed features: [cdi, rest-
client, resteasy]
```

```
[student@workstation apps-connect]$ cd adder; mvn quarkus:dev
[INFO] Scanning for projects...
...output omitted...
2020-09-22 05:34:11,753 INFO [io.quarkus] (main) Profile dev activated. Live
Coding activated.
2020-09-22 05:34:11,753 INFO [io.quarkus] (main) Installed features: [cdi, rest-
client, resteasy]
```

```
[student@workstation apps-connect]$ cd multiplier; mvn quarkus:dev
[INFO] Scanning for projects...
...output omitted...
2020-09-22 05:34:11,753 INFO [io.quarkus] (main) Profile dev activated. Live
Coding activated.
2020-09-22 05:34:11,753 INFO [io.quarkus] (main) Installed features: [cdi, rest-
client, resteasy]
```

For ease of usage, we provided a `~/DO378/solutions/apps-connect/start-locally.sh` script, which starts all three microservices in background and terminate them after pressing Enter. Output for the three microservices might be garbled when using this script.



### Note

When starting several microservices locally you might see some errors of the form `[ERROR] Port 5005 in use, not starting in debug mode.` This is expected, because each microservice tries to bind to the debug port 5005, but only one service is allowed to bind at any time. This does not affect the services unless you try to establish a debug session to them, which is outside the scope of this exercise.

- 12.2. To test that the microservices are working properly open a new terminal window, and execute the following curl commands.

```
[student@workstation apps-connect]$ curl "http://localhost:8080/solver/5"
5.0
[student@workstation apps-connect]$ curl "http://localhost:8081/adder/5/3"
8.0
[student@workstation apps-connect]$ curl "http://localhost:8082/multiplier/5/4"
20.0
[student@workstation apps-connect]$ curl "http://localhost:8080/solver/5*4+3"
23.0
```

End the development session for each microservice by either pressing **Enter** if you used the `start-locally.sh` script or by pressing **Ctrl+C** in the terminal for each service started manually.

- **13.** Now that you have tested the microservices locally you can perform a production deployment in OpenShift Container Platform. We have already prepared a project named after your user name and suffixed by `-apps-connect`. Add the `openshift` extension to all microservices and use it to deploy them into that project. The only service that needs to be exposed with a route is the `solver` service.
- 13.1. To add the `openshift` extension to each service execute the following command in each of the services' folders.

```
[student@workstation apps-connect]$ cd solver
[student@workstation solver]$ mvn quarkus:add-extension -Dextension=openshift
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.redhat.training:solver >-----
[INFO] Building solver 1.0-SNAPSHOT
[INFO] -----[jar]-----
[INFO]
[INFO] --- quarkus-maven-plugin:1.11.7.Final-redhat-00009:add-extension (default-cli) @ adder ---
Adding extension io.quarkus:quarkus-openshift
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
[student@workstation solver]$ cd ../adder
[student@workstation adder]$ mvn quarkus:add-extension -Dextension=openshift
...output omitted...
[student@workstation solver]$ cd ../multiplier
[student@workstation multiplier]$ mvn quarkus:add-extension -Dextension=openshift
...output omitted...
[student@workstation multiplier]$ cd ..
```

- 13.2. To deploy the microservices in OpenShift first make sure you are logged into the cluster and using the appropriate project.

```
[student@workstation apps-connect]$ source /usr/local/etc/ocp4.config
[student@workstation apps-connect]$ oc login ${RHT_OCP4_MASTER_API} \
-u ${RHT_OCP4_DEV_USER} -p ${RHT_OCP4_DEV_PASSWORD}
Login successful.
...output omitted...
[student@workstation apps-connect]$ oc project ${RHT_OCP4_DEV_USER}-apps-connect
```

- 13.3. Use the `mvn package -Dquarkus.kubernetes.deploy=true` command to deploy the `solver` microservice. This is the only service that needs to be exposed, so you must add the `-Dquarkus.openshift.expose=true` option. Remember to also add the base image for building Quarkus applications in OpenShift Container Platform.

```
[student@workstation apps-connect]$ cd solver
[student@workstation solver]$ mvn package -Dquarkus.kubernetes.deploy=true \
-Dquarkus.s2i.base-jvm-image=registry.access.redhat.com/ubi8/openjdk-11 \
-Dquarkus.openshift.expose=true
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.redhat.training:quarkus-solver >-----
[INFO] Building quarkus-solver 1.0-SNAPSHOT
[INFO] -----[jar]-----
...output omitted...
[INFO] [io.quarkus.container.image.s2i.deployment.S2iProcessor] Performing s2i
binary build with jar on server...
...output omitted...
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Deploying to
openshift server:...
...output omitted...
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Route
solver.
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 13.4. Using the same `mvn package` command, deploy the adder and the multiplier microservices. Those services need not to be exposed with a route, so you can skip the `-Dquarkus.openshift.expose=true` option.

```
[student@workstation apps-connect]$ cd ../adder
[student@workstation adder]$ mvn package -Dquarkus.kubernetes.deploy=true \
-Dquarkus.s2i.base-jvm-image=registry.access.redhat.com/ubi8/openjdk-11
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.redhat.training:adder >-----
[INFO] Building adder 1.0-SNAPSHOT
[INFO] -----[jar]-----
...output omitted...
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Deploying to
openshift server: ...
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
[student@workstation adder]$ cd ../multiplier
[student@workstation multiplier]$ mvn package -Dquarkus.kubernetes.deploy=true \
-Dquarkus.s2i.base-jvm-image=registry.access.redhat.com/ubi8/openjdk-11
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.redhat.training:multiplier >-----
[INFO] Building multiplier 1.0-SNAPSHOT
[INFO] -----[jar]-----
...output omitted...
```

```
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Deploying to
openshift server: ...
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
[student@workstation multiplier]$ cd ..
```

- 13.5. Obtain the URL of the `solver` route and execute the following `curl` command to validate the application is deployed successfully.

```
[student@workstation apps-connect]$ SOLVER_ROUTE=$(oc get route solver \
-o template --template '{{ "http://" }}{{ .spec.host }}')
[student@workstation apps-connect]$ curl "${SOLVER_ROUTE}/solver/5*4+3"
23.0%
```

The `adder` and `multiplier` services are not exposed, so you can not use a route to test them. Nevertheless, because you own the project they are deployed in, you can directly execute commands on the containers to test the deployment.

```
[student@workstation apps-connect]$ ADDER_POD=$(oc get pods -o name \
-l app.kubernetes.io/name=adder)
[student@workstation apps-connect]$ oc exec ${ADDER_POD} -- \
curl -s localhost:8080/adder/5/6
11.0
[student@workstation apps-connect]$ MULTIPLIER_POD=$(oc get pods -o name \
-l app.kubernetes.io/name=multiplier)
[student@workstation apps-connect]$ oc exec ${MULTIPLIER_POD} -- \
curl -s localhost:8080/multiplier/5/6
30.0
```

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab apps-connect finish
```

This concludes the guided exercise.

## ► Lab

# Building Microservice Applications with Quarkus

In this lab, you will take the conference application and modify its microservices, using service discovery so one service calls the other for data, and adding the new call using a feature toggle, enabling and disabling the feature, as well as changing its behavior, from configuration.

## Outcomes

You should be able to.

- Create an HTTP Client to consume a microservice using service discovery.
- Avoid hard coding values within your code by using configuration.
- Add new features to your code, which can be enabled and disabled using feature toggles.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this lab.

This command ensures that the directories and required files for starting and grading the lab are set up.

```
[student@workstation ~]$ lab apps-review start
```

## Instructions

1. Open the projects found in `/home/student/D0378/labs/apps-review/quarkus-conference/microservice-speaker` and `/home/student/D0378/labs/apps-review/quarkus-conference/microservice-session` using Codium.
2. In the `microservice-speaker` service, analyze the `org.acme.conference.speaker.SpeakerResource.listAll()` method and the `org.acme.conference.speaker.Speaker` class, because you will be creating a request to the endpoint in this activity.
3. Within the `microservice-session` project, create a `SpeakerService` interface, annotated with `@RegisterRestClient`, which consumes the `org.acme.conference.speaker.SessionResource.listAll()` endpoint. Do not try to use the `Speaker` class already present in `microservice-session`, because the fields will not match. Instead, create a new `SpeakerFromService` class, which contains only the `uuid`, `nameFirst`, and `nameLast` fields, all of type `String`, for the `SpeakerService` interface. Remember to add a way for JSON-B to deserialize the data to the `SpeakerFromService` class.
4. Configure the rest client for `org.acme.conference.session.SpeakerService` in `microservice-session` by adding the `mp-rest` URL configuration entry in `application.properties`, making sure it points to `http://localhost:8082`.

5. Open the `SessionStore` class in `microservice-session`. Add a Boolean property, which enables or disables a feature toggle. When enabled, the session service uses local speaker data. When disabled, it uses the recently created client to `SpeakerResource.listAll()` in the `microservice-speaker` service. Name the toggle `features.session-integration`.
6. Modify the `findById()` method for the `SessionStore`. If the `session-integration` toggle is `true`, then the service must call the `listAll()` method from a `SpeakerService` object. Use the result from the service call to update the `Speaker` object's `name` property with the result of concatenating `nameFirst`, a space, and `nameLast`. Note that the property `uuid` in `Speaker` has the same value as `uuid` obtained from `SpeakerService`, so you can filter using it.
7. To check for yourself if your code is running properly, in a terminal go to the `/home/student/D0378/labs/apps-review/quarkus-conference` directory and execute the `start-locally.sh` script, which will start the four microservices and the web application. It takes some time for the script to ready the application, the script generates a significant amount of output. You will know it is ready when the script's output says `Press enter to Terminate`. Open the URL `http://localhost:8080/` in a browser. Click `Sessions`, and then click any of the available sessions. In this page, you will find a list of speakers, which should only include the first name.

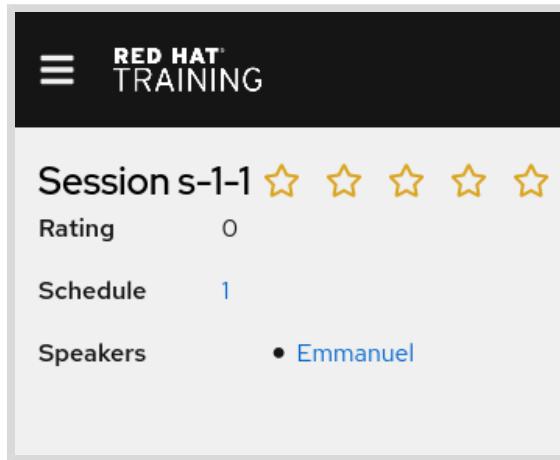


Figure 4.2: Displaying a session using the UI.

8. In the `application.properties` file from `microservice-session`, change the `features.session-integration` value to `true` and then, go back to your web browser and reload the page. If your code worked correctly, then you should find the names of the speakers to include last names.

## Evaluation

Grade your work by running the `lab apps-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab apps-review grade
```

## Finish

Terminate the `start-locally` script by pressing enter.

On the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab apps-review finish
```

This concludes the lab.

## ► Solution

# Building Microservice Applications with Quarkus

In this lab, you will take the conference application and modify its microservices, using service discovery so one service calls the other for data, and adding the new call using a feature toggle, enabling and disabling the feature, as well as changing its behavior, from configuration.

## Outcomes

You should be able to.

- Create an HTTP Client to consume a microservice using service discovery.
- Avoid hard coding values within your code by using configuration.
- Add new features to your code, which can be enabled and disabled using feature toggles.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this lab.

This command ensures that the directories and required files for starting and grading the lab are set up.

```
[student@workstation ~]$ lab apps-review start
```

## Instructions

1. Open the projects found in `/home/student/D0378/labs/apps-review/quarkus-conference/microservice-speaker` and `/home/student/D0378/labs/apps-review/quarkus-conference/microservice-session` using Codium.
  - 1.1. Open a terminal and type the following.

```
[student@workstation ~]$ codium \
~/D0378/labs/apps-review/quarkus-conference/microservice-speaker
[student@workstation ~]$ codium \
~/D0378/labs/apps-review/quarkus-conference/microservice-session
```

2. In the `microservice-speaker` service, analyze the `org.acme.conference.speaker.SpeakerResource.listAll()` method and the `org.acme.conference.speaker.Speaker` class, because you will be creating a request to the endpoint in this activity.  
Compare the returned data with the `Speaker` class found in `microservice-session`. Note the `uuid` fields match values in the different microservices when they refer to the same `Speaker` object. Use this field to identify the same `Speaker` entity in both services.

3. Within the `microservice-session` project, create a `SpeakerService` interface, annotated with `@RegisterRestClient`, which consumes the `org.acme.conference.speaker.SessionResource.listAll()` endpoint. Do not try to use the `Speaker` class already present in `microservice-session`, because the fields will not match. Instead, create a new `SpeakerFromService` class, which contains only the `uuid`, `nameFirst`, and `nameLast` fields, all of type `String`, for the `SpeakerService` interface. Remember to add a way for JSON-B to deserialize the data to the `SpeakerFromService` class.
- 3.1. Add the `quarkus-rest-client` extension by opening the `Command Palette`, writing `Quarkus: Add extensions to current project`, and then searching for `REST Client`. Select the extension, and press enter to continue.
  - 3.2. Create the `SpeakerFromService` POJO, using the following code.

```
package org.acme.conference.session;

import javax.json.bind.annotation.JsonbCreator;

public class SpeakerFromService {
 String uuid;
 String nameFirst;
 String nameLast;

 SpeakerFromService(String uuid, String nameFirst, String nameLast) {
 this.uuid = uuid;
 this.nameFirst = nameFirst;
 this.nameLast = nameLast;
 }

 @JsonbCreator
 public static SpeakerFromService of(String uuid, String nameFirst, String nameLast) {
 return new SpeakerFromService(uuid, nameFirst, nameLast);
 }
}
```

- 3.3. Add the following code in a `SpeakerService.java` file in the `microservice-session/src/main/java/org/acme/conference/session` directory.

```
package org.acme.conference.session;

import java.util.List;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

@Path("/speaker")
@RegisterRestClient
public interface SpeakerService {
 @GET
```

```
@Produces(MediaType.APPLICATION_JSON)
public List<SpeakerFromService> listAll();
}
```

4. Configure the rest client for org.acme.conference.session.SpeakerService in microservice-session by adding the mp-rest URL configuration entry in application.properties, making sure it points to http://localhost:8082.

- 4.1. Add the following line to the application.properties file found within microservice-session.

```
...output omitted...
org.acme.conference.session.SpeakerService/mp-rest/url=http://localhost:8082
```

5. Open the SessionStore class in microservice-session. Add a Boolean property, which enables or disables a feature toggle. When enabled, the session service uses local speaker data. When disabled, it uses the recently created client to SpeakerResource.listAll() in the microservice-speaker service. Name the toggle features.session-integration.

- 5.1. Add the following property to SessionStore.

```
...output omitted...
import org.eclipse.microprofile.config.inject.ConfigProperty;

public class SessionStore {
 @ConfigProperty(name = "features.session-integration")
 boolean sessionIntegration;
 ...output omitted...
}
```

And add the features.session-integration value in application.properties.

```
...output omitted...
features.session-integration=false
...output omitted...
```

6. Modify the findById() method for the SessionStore. If the session-integration toggle is true, then the service must call the listAll() method from a SpeakerService object. Use the result from the service call to update the Speaker object's name property with the result of concatenating nameFirst, a space, and nameLast. Note that the property uuid in Speaker has the same value as uuid obtained from SpeakerService, so you can filter using it.

- 6.1. Inject a SpeakerService object into the SessionStore class, using the @RestClient annotation. Modify the findById() method in the same class, to change the code into the following.

```
...output omitted...
import java.util.List;

import org.eclipse.microprofile.config.inject.ConfigProperty;
```

```

import org.eclipse.microprofile.rest.client.inject.RestClient;
...output omitted...

public class SessionStore {
 @ConfigProperty(name = "features.session-integration")
 boolean sessionIntegration;

 @Inject
 @RestClient
 SpeakerService speakerService;
 ...output omitted...

 public Optional<Session> findById (String sessionId) {
 Optional<Session> result = repository.find("id",
 sessionId).stream().findFirst();
 if (sessionIntegration && result.isPresent()) {
 List<SpeakerFromService> speakers = speakerService.listAll();
 Session session = result.get();
 for (var speaker : session.speakers) {
 var serviceSpeaker = speakers.stream().filter(s ->
s.uuid.equals(speaker.uuid)).findFirst();
 if (serviceSpeaker.isPresent()) {
 var sameSpeaker = serviceSpeaker.get();
 speaker.name = sameSpeaker.nameFirst + " " +
sameSpeaker.nameLast;
 }
 }
 return Optional.of(session);
 }
 return result;
 }
 ...output omitted...
}

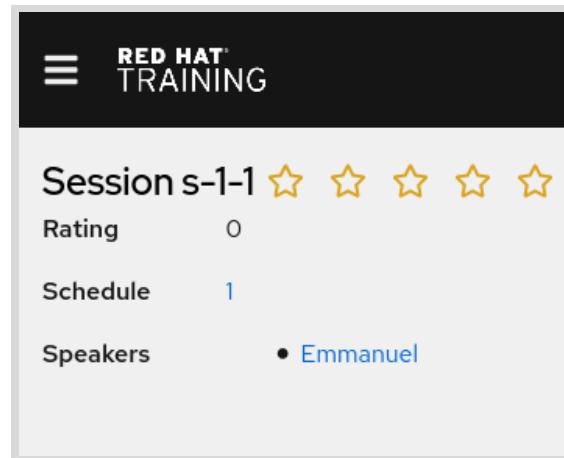
```

- To check for yourself if your code is running properly, in a terminal go to the /home/student/D0378/labs/apps-review/quarkus-conference directory and execute the start-locally.sh script, which will start the four microservices and the web application.

```
[student@workstation ~]$ cd ~/D0378/labs/apps-review/quarkus-conference
[student@workstation quarkus-conference]$./start-locally.sh
```

It takes some time for the script to ready the application, the script generates a significant amount of output. You will know it is ready when the script's output says Press enter to Terminate.

Open the URL <http://localhost:8080/> in a browser. Click Sessions, and then click any of the available sessions. In this page, you will find a list of speakers, which should only include the first name.



**Figure 4.2: Displaying a session using the UI.**

8. In the `application.properties` file from `microservice-session`, change the `features.session-integration` value to `true` and then, go back to your web browser and reload the page. If your code worked correctly, then you should find the names of the speakers to include last names.

## Evaluation

Grade your work by running the `lab apps-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab apps-review grade
```

## Finish

Terminate the `start-locally` script by pressing enter.

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab apps-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- Quarkus uses `MicroProfile Config` specification to define application configuration sources. That Configuration sources have a defined priority, starting from the `application.properties` file up to Java system properties. And that Quarkus provide three default configuration profiles: `dev`, `test` and `prod`, and that Developers can create custom configuration profiles as needed.
- You can use Quarkus Configuration to activate `Feature Toggles`, which are a method to early release features by dynamically controlling its availability.
- The `quarkus-rest-client` extension uses RESTEasy as the implementation for the `MicroProfile Rest Client` specification using the `RegisterRestClient` annotation on an interface. Use the generated REST client injecting with both `Inject` and `RestClient` annotations

## Chapter 5

# Implementing Fault Tolerance

### Goal

Implement fault tolerance in a microservice architecture.

### Objectives

- Apply fault tolerance policies to a Microservice.
- Implement a health check in a microservice and enable a probe in OpenShift to monitor it.

### Sections

- Applying Fault Tolerance Policies to a Microservice (and Guided Exercise)
- Implementing a Health Check Monitored by OpenShift (and Guided Exercise)

### Lab

Implementing Fault Tolerance

# Applying Fault Tolerance Policies to a Microservice

## Objectives

After completing this section, you should be able to apply fault tolerance policies to a Microservice.

## Introducing Fault Tolerance Policies

If a dependent service becomes unavailable, then *fault tolerance* ensures that a microservice fails gracefully. For example, hardware failures, network connectivity issues, routine maintenance, and failed deployments are all reasons a service could go offline at any moment.

When a service depends on another application, use a reliable fault tolerance framework to ensure that the microservice does not succumb to any downstream failures.

The fault tolerance policy implementation in Quarkus uses *SmallRye Fault Tolerance*. SmallRye FaultTolerance is an implementation of the *MicroProfile Fault Tolerance* specification. This specification uses multiple strategies to minimize the effects of dependency failures by implementing a set of recovery procedures for microservices.

The recovery procedures defined by the fault tolerance specification include the following:

### **Circuit breaker (@org.eclipse.microprofile.faulttolerance.CircuitBreaker)**

Supports a fail-fast approach if the system is suffering from an overload or is unavailable.

### **Bulkhead (@org.eclipse.microprofile.faulttolerance.Bulkhead)**

Limits the workload for a microservice to avoid failures caused by concurrency or service overload.

### **Fallback(@org.eclipse.microprofile.faulttolerance.Fallback)**

Executes an alternative method if the execution fails for the annotated method.

### **Retry policy(@org.eclipse.microprofile.faulttolerance.Retry)**

Defines the conditions for retrying a failing execution.

### **Timeout(@org.eclipse.microprofile.faulttolerance.Timeout)**

Defines the maximum execution time before interrupting a request.



#### Note

The Asynchronous policy, defined by the `org.eclipse.microprofile.faulttolerance.Asynchronous` annotation, has been intentionally left behind for two reasons. First, Asynchronous is not a fault tolerance policy by itself, but a technique to decouple requests from the response, avoiding unnecessary waits and optimizing resource usage. Second, Quarkus already integrates asynchronous and reactive programming techniques in its core, making the Asynchronous policy obsolete.

## Applying MicroProfile Fault Tolerance

Each policy has its use case and parameters. The following sections introduce common use cases for each policy and describe the meaning for each supported parameter.

### Avoiding Long Awaits with Timeout

When a service fails or depends on another unresponsive service, it can take a long time to generate the response. Depending on the situation or the underlying service, this time may be unacceptably long. The `org.eclipse.microprofile.faulttolerance.Timeout` annotation applies to an endpoint method and limits the amount of time the method takes to generate a response. If no response is available within the time frame, then the framework interrupts the invocation to the endpoint method, raising a `TimeoutException` exception.

The `Timeout` annotation accepts two parameters: the `value` indicates the amount of time and the `unit` defines the time unit used. For example, the following instructs the framework to interrupt the `getProduct` method, if it does not respond in less than 200 seconds.

```
@Timeout(value=200,unit = ChronoUnit.SECONDS)
public Product getProduct(int id) {
 ...output omitted...
```

The following table lists all of the `Timeout` annotation parameters.

#### Timeout Parameters

Annotation	Java type	Description
value	long	The maximum amount of time the method can take before it is interrupted.
unit	ChronoUnit	The time unit of the timeout.



#### Note

The FQN for `ChronoUnit` is `java.time.temporal.ChronoUnit`.

### Retrying Failing Requests

Sometimes, services have transient issues, which often only occur one or a few times. In those cases, it is often easier and cheaper to just retry the request. The `Retry` policy instructs the framework to repeat a request to a service if it throws an exception.

To activate the `Retry` policy on an endpoint method, annotate that method with the `org.eclipse.microprofile.faulttolerance.Retry`, and provide that annotation with appropriate options. Such options include the maximum number of retries or a list of situations that might cause the retry.

The following example retries the `getProduct` method up to 90 times, for at most 100 seconds, and only while the method raises `RuntimeException` exceptions or descendants.

```
@Retry(maxRetries=90, maxDuration=100, retryOn={RuntimeException.class})
public Product getProduct(int id) {
 ...output omitted...
}
```

The following table lists all of the `Retry` annotation parameters.

### Retry Parameters

Annotation	Java type	Description
maxRetries	long	The maximum number of retries.
delay	long	The amount of time for the delays between each retry.
delayUnit	ChronoUnit	The delay time unit.
maxDuration	long	The maximum amount of time that the retry attempts may take.
durationUnit	ChronoUnit	The duration time unit.
jitter	long	The range limit for a random amount of time added or subtracted to the delay.
jitterDelayUnit	ChronoUnit	The jitter time unit.
retryOn	Class[]	The exception families that cause execution of a retry.
abortOn	Class[]	The exceptions that cause the execution to abort.

## Providing Fallback Responses When Everything Else Fails

Services can still fail even when using a timeout or retrying requests. In those cases, if returning a meaningful response is mandatory, then the service must create a simpler or pre-built response.

The *Fallback* pattern allows the developer to define an alternative method the framework can use if the original endpoint method fails to respond. Quarkus uses the `org.eclipse.microprofile.faulttolerance.Fallback` annotation to define such an alternative method or class.

If the `Fallback` annotation defines a method, then that method must have exactly the same erasure as the original method and must be accessible from the original class.

```
@Fallback(fallbackMethod="getCachedProduct")
public Product getProduct(int id) {
 ...output omitted...
}
...output omitted...
public Product getCachedProduct(int id) {
 ...output omitted...
```

If the `Fallback` annotation defines a class, then that class must implement a method of the form `String handle(ExecutionContext context)`. That method must be accessible from the original class.

```
public class ProductResource {
 ...output omitted...
 @Fallback(ProductFallback.class)
 public Product getProduct(int id) {
 ...output omitted...
 }
 ...output omitted...
 public class ProductFallback {
 ...output omitted...
 public String handle(ExecutionContext context) {
 ...output omitted...
```

The following table lists all of the `Fallback` annotation parameters.

#### Fallback Parameters

Annotation	Java type	Description
<code>fallbackMethod</code>	<code>String</code>	The method to call if an exception is raised.
<code>value</code>	<code>Class</code>	The <code>StringFallbackHandler</code> implementation that manages the fallback.

## Avoiding Transient Errors with Circuit Breaker

The *Circuit Breaker* pattern prevents requests to a temporary failing or degraded service, exploiting the temporal affinity of some problems. It exploits the high probability that a request failure indicates that following requests will also fail.

Depending on the responsiveness status of the service, the *Circuit Breaker* pattern defines the following three circuit states:

### Closed

The circuit is closed and the service operates as expected. If a failure occurs, then the circuit breaker keeps the record. The circuit opens after reaching the specified number of failures.

### Open

The service does not work, and every call to the circuit breaker fails immediately. The circuit transitions to a half-open state after reaching the specified timeout.

## Half-open

After a grace period, Quarkus passes a single call to the service to evaluate whether it has stabilized. If it fails, then the circuit opens again. Otherwise, subsequent calls are allowed. After the specified number of successful executions, the circuit closes.

Quickly and safely detecting a failing service is the most complex topic of the **Circuit Breaker** policy. To detect a failing service, the **Circuit Breaker** pattern uses a request window, which is a set of consecutive requests. If the number of failing requests inside this window is below a threshold, then the framework considers the service malfunctioning and opens the circuit.

```
@org.eclipse.microprofile.faulttolerance.CircuitBreaker(requestVolumeThreshold =
 4, failureRatio = 0.5, delay = 1000)
public List<String> getProducts() {
 ...output omitted...
}
```

In the previous code, the circuit opens if half of the requests (`failureRatio = 0.5`) of four consecutive invocations (`requestVolumeThreshold=4`) fail. The circuit stays open for 1000 milliseconds and then it becomes **half-open**. After a successful invocation, the circuit is closed again.

The following table lists all the **CircuitBreaker** annotation parameters.

### CircuitBreaker Parameters

Annotation	Java type	Description
<code>successThreshold</code>	<code>long</code>	The number of consecutive successful invocations to close the circuit.
<code>requestVolumeThreshold</code>	<code>long</code>	The number of consecutive invocations used as the baseline to calculate the number of failures.
<code>failureRatio</code>	<code>double</code>	The minimum failure ratio required to open the circuit.
<code>delay</code>	<code>long</code>	The amount of time the circuit stays open to review that the service is back to normal.
<code>delayUnit</code>	<code>ChronoUnit</code>	The time units to define the delay
<code>failOn</code>	<code>Class[]</code>	List of exceptions counting as a service failure.
<code>skipOn</code>	<code>Class[]</code>	List of exceptions counting as a service success.

## Preventing Service Saturation with the Bulkhead Pattern

The *Bulkhead* pattern prevents a service from malfunctioning due to saturation. It is common to find services that become unreliable when receiving many simultaneous requests. The *Bulkhead* pattern limits the number of simultaneous requests and promptly fails requests beyond that number.

To enable the *Bulkhead* pattern, add the `org.eclipse.microprofile.faulttolerance.Bulkhead` annotation to the endpoint method. Then, indicate the maximum number of simultaneous requests allowed and the size of the waiting queue.

```
@Bulkhead(value = 5, waitingTaskQueue = 8)
public List getProducts() {
 ...output omitted...
```

In the previous example, the frameworks only allow five simultaneous invocations of the `getProducts()` method. If you invoke the method with more concurrent calls, then the bulkhead queues up to eight pending requests. Once the queue fills, additional requests trigger the fault tolerance implementation to throw a `BulkheadException` exception.

The following table lists all the *Bulkhead* annotation parameters.

### Bulkhead Parameters

Annotation	Java type	Description
<code>waitingTaskQueue</code>	<code>int</code>	The maximum number of requests in the queue.
<code>value</code>	<code>int</code>	The number of requests that can be processed by the method.

## Understanding Interactions Between Fault Tolerance Policies

You can apply multiple fault tolerance policies at the same time. For example, you can apply the *Circuit Breaker* to detect service failures and provide a *Fallback* method to generate short-cut responses.

Policies are applied in the following order:

1. If the circuit is open, then the *Circuit Breaker* policy throws an exception.
2. If the *Bulkhead* and its queue are full, then it throws an exception.
3. If none of the previous steps throw an exception, then the framework invokes the original endpoint method.
4. The *Timeout* policy interrupts the request if it takes too long.
5. The *Retry* policy retries the request.
6. The *Fallback* method returns the short-cut response if any of the previous throw an exception.

When mixing the `Retry` and `Circuit Breaker` policies, the circuit breaker records each retry and uses it to compute the circuit status in the threshold window. The same rule applies when mixing the `Bulkhead` policy and the `Circuit Breaker` policy: the circuit breaker uses bulkhead exceptions to determine the circuit status.

## Describing Bindings with Other MicroProfile Specifications

MicroProfile Fault Tolerance seamlessly integrates with MicroProfile Configuration and with MicroProfile Metrics specifications.

You can configure all annotations previously presented using the `application.properties` file and the following naming convention: `<classname>/<methodname>/<annotation>/<parameter>`. For example, the following line is equivalent to adding the `@Retry(maxRetries=90)` annotation to the `getProduct` method of the `com.acme.ProductResource` class.

```
com.acme.ProductResource/getProduct/Retry/maxRetries=90
```

You can skip the method name to apply the same annotation to all endpoint methods of the class.

```
com.acme.ProductResource/Retry/maxRetries=90
```

When used in conjunction with the `smallrye-metrics` extension, MicroProfile Metrics enable all the previous annotations to add appropriate metrics.

For example, the `Fallback` annotation generates a `ft.<name>.fallback.calls.total` metric, where `<name>` is the FQN of the original method. This metric counts the number of times the fallback method responds.

To obtain the complete list of metrics and their meanings, refer to the `MicroProfile Fault Tolerance` specification.

## Listing Other Fault Tolerance Implementations

Although Quarkus relies on the SmallRye implementation of the `Eclipse MicroProfile Fault Tolerance` specifications, there are many other implementations and techniques to ensure service reliability.

### Hystrix

Hystrix is a third-party library from Netflix OSS. It is an implementation of fault tolerance that supports MicroProfile requirements. Hystrix also supports latency, monitoring, and concurrency capabilities, as well as a web UI for monitoring metrics and identifying problems.

Like Quarkus, Hystrix approaches fault tolerance from code within the microservice, injecting the desired features through method annotations.

### Red Hat OpenShift Service Mesh

Based on the upstream project Istio, Red Hat OpenShift Service Mesh approaches fault tolerance from a platform level. OpenShift Service Mesh injects a `sidecar container` to the microservice container. This sidecar container intercepts all requests to the service and injects fault tolerance behavior, as needed.

This approach to fault tolerance has two main benefits:

- Configuration is outside of the application, so changes to policies do not require additional builds or deployments.
- The implementation resides outside of the application itself, which provides a language-agnostic implementation and relieves the development team from implementing it.



## References

### **MicroProfile Fault Tolerance Specifications**

<https://download.eclipse.org/microprofile/microprofile-fault-tolerance-2.1.1/microprofile-fault-tolerance-spec.html>

### **SmallRye Fault Tolerance repository**

<https://github.com/smallrye/smallrye-fault-tolerance>

### **Red Hat OpenShift Service Mesh product page**

<https://www.openshift.com/learn/topics/service-mesh>

## ► Guided Exercise

# Applying Fault Tolerance Policies

In this exercise, you will enable some fault tolerance features in one of the microservices for the `Istio-Tutorial` application.

## Outcomes

You should be able to apply Quarkus fault tolerance features to microservices.

## Before You Begin

On the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command places the code for the `Istio-Tutorial` application in your workstation.

```
[student@workstation ~]$ lab tolerance-policies start
```

## Instructions

- 1. In the terminal window, open a VSCode session in the `~/D0378/labs/tolerance-policies/istio-tutorial/recommendation`. Open the file `src/main/java/com/redhat/developer/demos/recommendation/rest/RecommendationResource.java` and review its contents.

```
[student@workstation ~]$ cd
~/D0378/labs/tolerance-policies/istio-tutorial/recommendation/
[student@workstation recommendation]$ codium .
```

The `recommendation` microservice returns the host name it is running on. This service provides the following endpoints that control its behavior:

- The `/misbehave` endpoint toggles whether it returns a 503 service unavailable HTTP response.
- The `/timeout` endpoint toggles a delay of 1 second on 50% of the responses.
- The `/crash` endpoint toggles 50% of the responses to crash by throwing an exception.
- The `/reset` endpoint disables all switches and resets the response counter to zero.

For testing purposes, use the provided `sequential.sh` and `parallel.sh` scripts in the `~/D0378/labs/tolerance-policies/istio-tutorial` directory.

- 2. Add the `quarkus-smallrye-fault-tolerance` extension to the project. Use the VSCode session command palette or execute the following command in the terminal.

```
[student@workstation recommendation]$ mvn quarkus:add-extension \
-Dextension=quarkus-smallrye-fault-tolerance
[INFO] Scanning for projects...
...output omitted...
Extension io.quarkus:quarkus-smallrye-fault-tolerance has been installed
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- ▶ 3. Apply the `Retry` annotation to the `getRecommendations` method so that Quarkus retries the endpoint 10 times.
  - 3.1. Start the `recommendation` microservice in a terminal window by using the IDE or the `mvn quarkus:dev` command. Keep this terminal open, we will be reviewing the logs.

```
[student@workstation recommendation]$ HOSTNAME=$HOSTNAME mvn quarkus:dev
...output omitted...
Listening for transport dt_socket at address: 5005
--_ _ _ _ _ / \ / / / _ | / _ \ / / / / / / /
-/ / / / / / / | / , _/ , < / / / \ \ \
--____/_ / | _/ | _/ | _| __/_ /
2021-09-14 12:10:45,290 INFO [io.quarkus] (Quarkus Main Thread) recommendation
1.0-SNAPSHOT on JVM (powered by Quarkus 1.11.7.Final-redhat-00009) started in
1.746s. Listening on: http://localhost:8080
2021-09-14 12:10:45,305 INFO [io.quarkus] (Quarkus Main Thread) Profile dev
activated. Live Coding activated.
2021-09-14 12:10:45,305 INFO [io.quarkus] (Quarkus Main Thread) Installed
features: [cdi, resteasy, smallrye-context-propagation, smallrye-fault-tolerance]
```



### Note

Specifying the `HOSTNAME` variable definition addresses a shortcoming in the way environmental variables are passed to Java applications.

- 3.2. Enable the service crashing half of the requests. Open a new terminal window and perform a request to the `/crash`.

```
[student@workstation ~]$ curl http://localhost:8080/crash
'crash' has been set to true
```

- 3.3. Use the `parallel.sh` script to perform 10 requests to the root path of the `recommendation` service. Observe both the service logs and the in `curl` output the stack traces of the service crashing.

```
[student@workstation recommendation]$/parallel.sh "curl localhost:8080" 10
...output omitted...
recommendation v1 from 'workstation.lab.example.com': 4
...output omitted...
Resulted in: org.jboss.resteasy.spi.UnhandledException:
 java.lang.RuntimeException: Resource failure.
...output omitted...
```

- 3.4. Edit the `RecommendationResource.java` file and add the `org.eclipse.microprofile.faulttolerance.Retry` annotation to the `getRecommendations` method. Add `maxRetries=10` to set the number of maximum retries to the endpoint.

```
import org.eclipse.microprofile.faulttolerance.Retry;
...output omitted...
@GET
@Retry(maxRetries = 10)
public Response getRecommendations() {
...output omitted...
```

- 3.5. Invoke the `/crash` endpoint to trigger code live reload and enable the service crash again. Then, repeat the same 10 parallel requests as before. You can now see the logs reporting the crash in the service, however, requests are repeated until succeeded so that the `curl` command receives a correct response.

```
[student@workstation recommendation]$ curl http://localhost:8080/crash
'crash' has been set to true
[student@workstation recommendation]$/parallel.sh "curl localhost:8080" 10
recommendation v1 from 'workstation.lab.example.com': 2
recommendation v1 from 'workstation.lab.example.com': 4
recommendation v1 from 'workstation.lab.example.com': 7
recommendation v1 from 'workstation.lab.example.com': 10
recommendation v1 from 'workstation.lab.example.com': 11
recommendation v1 from 'workstation.lab.example.com': 14
recommendation v1 from 'workstation.lab.example.com': 14
recommendation v1 from 'workstation.lab.example.com': 15
recommendation v1 from 'workstation.lab.example.com': 18
recommendation v1 from 'workstation.lab.example.com': 19
```



### Note

There is a small chance that all 10 retries fail and the service returns a failure. If this happens, then retry the parallel requests again.

- 3.6. Finalize this activity by clearing the status of the microservice. Invoke the `/reset` endpoint of the `recommendation` microservice.

```
[student@workstation recommendation]$ curl http://localhost:8080/reset
Service reset.
```

- 4. The recommendation service contains a potential race condition, which might cause the response counter to misbehave. Use the Bulkhead annotation to limit concurrent connections to the service to 1.
- 4.1. Perform 100 parallel requests to the root path of the recommendation service. As the race condition occurs, many responses will return duplicated values for the response count. Use the following command to display the duplicated responses.

```
[student@workstation recommendation]$./parallel.sh \
"curl -s localhost:8080" 100 | sort | uniq -c \
| grep "recommendation v1" | egrep -v "^[[:blank:]]+1[[:blank:]]"
10 recommendation v1 from 'workstation.lab.example.com': 84
3 recommendation v1 from 'workstation.lab.example.com': 86
```

Your output might differ. In this case, 10 requests returned 84 as the response counter, which is incorrect.

- 4.2. Add the `org.eclipse.microprofile.faulttolerance.Bulkhead` to the `getRecommendations` method to limit concurrent requests to 1. This avoids concurrency issues by aborting requests.



### Important

Remove the `Retries` annotation to avoid refused requests being retried.

```
import org.eclipse.microprofile.faulttolerance.Bulkhead;
...output omitted...
@GET
@Bulkhead(value=1)
public Response getRecommendations() {
...output omitted...
```

- 4.3. Reset the service to trigger live code reload and return the response counter to zero. Then, perform 100 parallel requests. See in both the logs and in the responses how the bulkhead rejects some requests.

```
[student@workstation recommendation]$ curl http://localhost:8080/reset
Service reset.
[student@workstation recommendation]$./parallel.sh "curl localhost:8080" 100
recommendation v1 from 'workstation.lab.example.com': 1
recommendation v1 from 'workstation.lab.example.com': 2
recommendation v1 from 'workstation.lab.example.com': 3
...output omitted...
<!doctype html>
<html lang="en">
<head>
<title>Internal Server Error - Error handling 13db50c9-
a9d5-43b0-97ed-1b6e6621a114-13, org.jboss.resteasy.spi.UnhandledException:
org.eclipse.microprofile.faulttolerance.exceptions.BulkheadException:
Bulkhead[com.redhat.developer.demos.recommendation.rest.RecommendationResource#ge
tRecommendations] rejected from bulkhead</title>
...output omitted...
```

Prove that there are no concurrency issues by executing the previous command again.

```
[student@workstation recommendation]$ curl http://localhost:8080/reset
Service reset.
[student@workstation recommendation]$../parallel.sh
"curl -s localhost:8080/" 100 | sort | uniq -c
| grep "recommendation v1" | egrep -v "^[[:blank:]]+1[[:blank:]]"
[student@workstation recommendation].
```

- ▶ 5. The bulkhead pattern avoided concurrency issues, but introduced service failures. Use the `Fallback` annotation to provide an alternative response when the service fails.
  - 5.1. Annotate the `getRecommendations` method with the `org.eclipse.microprofile.faulttolerance.Fallback` annotation. Indicate that the fallback policy must use the `getFallbackRecommendations` method as the fallback response generator method. This method is already present for the sake of simplicity.

```
import org.eclipse.microprofile.faulttolerance.Fallback;
...output omitted...
@GET
@Bulkhead(value=1)
@Fallback(fallbackMethod = "getFallbackRecommendations")
public Response getRecommendations() {
...output omitted...
 public Response getFallbackRecommendations() {
...output omitted...
```

- 5.2. Reset the service to trigger live code reload and return the response counter to zero. Again, perform 100 parallel requests. This time you can see no service crashes nor duplicated responses, but some responses show the fallback mechanism.

```
[student@workstation recommendation]$ curl http://localhost:8080/reset
Service reset.
[student@workstation recommendation]$../parallel.sh "curl localhost:8080" 100
recommendation v1 from 'workstation.lab.example.com': 1
recommendation v1 from 'workstation.lab.example.com': 3
...output omitted...
recommendation v1 from 'workstation.lab.example.com': 19
recommendation fallback from 'workstation.lab.example.com': 49
recommendation v1 from 'workstation.lab.example.com': 20
...output omitted...
```

Again, prove that there are no concurrency issues.

```
[student@workstation recommendation]$ curl http://localhost:8080/reset
Service reset.
[student@workstation recommendation]$../parallel.sh \
"curl -s localhost:8080/" 100 | sort | uniq -c \
| grep "recommendation v1" | egrep -v "^[[:blank:]]+1[[:blank:]]"
[student@workstation recommendation].
```

- ▶ 6. Simulate degraded service performance and address it using the `Timeout` annotation.
- 6.1. Use the `/timeout` endpoint to trigger the live code reload and to switch on service degraded performance.



### Important

Remove the `Bulkhead` annotation from the `getRecommendations` method.

```
[student@workstation recommendation]$ curl http://localhost:8080/reset
Service reset.
[student@workstation recommendation]$ curl http://localhost:8080/timeout
'timeout' has been set to true
```

The following command shows some of the responses took longer than one second to return.

```
[student@workstation recommendation]$../parallel.sh \
"time -f %E curl -s localhost:8080/" 10
recommendation v1 from 'workstation.lab.example.com': 1
0:00.03
recommendation v1 from 'workstation.lab.example.com': 5
0:00.02
...output omitted...
recommendation v1 from 'workstation.lab.example.com': 10
recommendation v1 from 'workstation.lab.example.com': 10
0:01.03 0:01.03 0:01.03
```

- 6.2. Use the `org.eclipse.microprofile.faulttolerance.Timeout` annotation to shortcut responses taking more than 500 milliseconds.

```
import org.eclipse.microprofile.faulttolerance.Timeout;
...output omitted...
@GET
@Fallback(fallbackMethod = "getFallbackRecommendations")
@Timeout(500)
public Response getRecommendations() {
```

- 6.3. Use the `/timeout` endpoint again to trigger live reload and enable service degraded performance. Repeat the previous `parallel.sh` command and observe timings and responses.

```
[student@workstation recommendation]$ curl http://localhost:8080/timeout
'timeout' has been set to true
[student@workstation recommendation]$../parallel.sh \
"time -f %E curl -s localhost:8080/" 10
recommendation v1 from 'workstation.lab.example.com': 65
0:00.02
recommendation v1 from 'workstation.lab.example.com': 68
0:00.03
recommendation v1 from 'workstation.lab.example.com': 71
0:00.04
```

```
recommendation v1 from 'workstation.lab.example.com': 70
0:00.03
recommendation v1 from 'workstation.lab.example.com': 73
0:00.03
recommendation fallback from 'workstation.lab.example.com': 75 0:00.52
recommendation fallback from 'workstation.lab.example.com': 76 0:00.53
recommendation fallback from 'workstation.lab.example.com': 77 0:00.52
recommendation fallback from 'workstation.lab.example.com': 78 0:00.54
recommendation fallback from 'workstation.lab.example.com': 79 0:00.53
```

Half of the responses suffer time degradation. The `Timeout` interrupts those responses after 500 milliseconds and directly returns the fallback response.

- ▶ 7. In the last section of this exercise you will enable the `circuit breaker` pattern and observe fallback time affinity.
  - 7.1. Enable both service performance degradation and service crash using both the `/timeout` and `/crash` endpoints. Then, start an infinite loop of requests to the service using the provided `sequential.sh` script.



### Important

Remove the `Timeout` annotation from the `getRecommendations` method.

```
[student@workstation recommendation]$ curl http://localhost:8080/crash
'crash' has been set to true
[student@workstation recommendation]$ curl http://localhost:8080/timeout
'timeout' has been set to true
[student@workstation recommendation]$../../sequential.sh \
"time -f %E curl -s localhost:8080/"
recommendation fallback from 'workstation.lab.example.com': 2
0:01.00
recommendation v1 from 'workstation.lab.example.com': 3
0:00.00
recommendation v1 from 'workstation.lab.example.com': 4
0:00.00
recommendation fallback from 'workstation.lab.example.com': 6 0:01.00
recommendation fallback from 'workstation.lab.example.com': 8 0:01.00
recommendation v1 from 'workstation.lab.example.com': 9
0:00.00
recommendation v1 from 'workstation.lab.example.com': 10
0:00.00
recommendation fallback from 'workstation.lab.example.com': 12 0:01.00
recommendation v1 from 'workstation.lab.example.com': 13
0:00.00
...output omitted...
```

Leave the script running. Observe how some requests take 1 second to respond and finally crash, hence the fallback response returns. You can also see this behavior in the service logs.

```

2020-10-05 04:06:55,388 INFO
[com.red.dev дем rec.res.RecommendationResource_Subclass] (executor-thread-39)
recommendation request from workstation.lab.example.com: 1
2020-10-05 04:06:55,594 INFO
[com.red.dev дем rec.res.RecommendationResource_Subclass] (executor-thread-39)
recommendation request from workstation.lab.example.com: 2
2020-10-05 04:06:55,600 INFO
[com.red.dev дем rec.res.RecommendationResource_Subclass] (executor-thread-39)
recommendation request from workstation.lab.example.com: 3
2020-10-05 04:06:55,600 INFO
[com.red.dev дем rec.res.RecommendationResource_Subclass] (executor-
thread-39) This request will be delayed 1000ms 2020-10-05 04:06:56,600
ERROR [com.red.dev дем rec.res.RecommendationResource_Subclass]
(executor-thread-39) Crashing! 2020-10-05 04:06:56,601 INFO
[com.red.dev дем rec.res.RecommendationResource_Subclass] (executor-thread-39)
recommendation request fallback from: workstation.lab.example.com: 4
2020-10-05 04:06:56,607 INFO
[com.red.dev дем rec.res.RecommendationResource_Subclass] (executor-thread-39)
recommendation request from workstation.lab.example.com: 5
...output omitted...

```

- 7.2. Stop the script by pressing **Ctrl+C**. Add the `org.eclipse.microprofile.faulttolerance.CircuitBreaker` annotation to the `getRecommendations` method. Add the required options to open the circuit when at least 2 requests fail from the last 5 (40% failures).

```

@GET
@Fallback(fallbackMethod = "getFallbackRecommendations")
@CircuitBreaker(requestVolumeThreshold = 5, failureRatio = 0.4)
public Response getRecommendations() {

```

- 7.3. Enable the timeout and crash switches using the endpoints, as they are disabled when the application reloads. Then, restart the `sequential.sh` script and observe the distribution and timings of the fallback responses.

```

[student@workstation recommendation]$ curl http://localhost:8080/crash
'crash' has been set to true
[student@workstation recommendation]$ curl http://localhost:8080/timeout
'timeout' has been set to true
[student@workstation recommendation]$../../sequential.sh \
 "time -f %E curl -s localhost:8080/"
...output omitted...
recommendation fallback from 'workstation.lab.example.com': 136
0:00.00
recommendation v1 from 'workstation.lab.example.com': 137
0:00.00
recommendation fallback from 'workstation.lab.example.com': 139 ①
0:01.00 recommendation fallback from 'workstation.lab.example.com': 141 0:01.00
recommendation v1 from 'workstation.lab.example.com': 142
0:00.00
recommendation v1 from 'workstation.lab.example.com': 143
0:00.00

```

```
recommendation v1 from 'workstation.lab.example.com': 144
0:00.00
recommendation fallback from 'workstation.lab.example.com': 145 2
0:00.00 recommendation fallback from 'workstation.lab.example.com': 146 0:00.00
recommendation fallback from 'workstation.lab.example.com': 147 0:00.00
...output omitted...
```

- ➊ Some requests fail due to the random nature of the service and the enabled flags. Because of the added delay, those responses take 1 second. Those failures are randomly spread throughout the responses.
- ➋ Once 2 out of 5 requests fail, the circuit opens and a streak of direct failures occur. Those failures are almost immediate.

In a third terminal window, reset service flags by using the `/reset` endpoint. Observe how the service returns to normal behavior after a few seconds and all responses succeed without delay.

Go to the terminal running the `sequential.sh` script and terminate it by pressing `Ctrl+C`. Use the same key combination in the terminal that you used to start your Quarkus application, to terminate the development session.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab tolerance-policies finish
```

This concludes the guided exercise.

# Implementing a Health Check Monitored by OpenShift

---

## Objectives

After completing this section, you should be able to implement a health check in a microservice and enable a probe in OpenShift to monitor it.

## MicroProfile Health

Cloud environments use several orchestration mechanisms to manage an application's lifecycle. One important aspect of this lifecycle management is checking if an instance of a service is still functioning properly. If a service fails, the orchestration system can redirect traffic away from it, and even restart the service in case of continued failure.

The *MicroProfile Health* specification allows services to notify the lifecycle orchestration mechanism of their current status, which allows the system to act accordingly. This is achieved by using two dedicated endpoints: `/q/health/ready` and `/q/health/live`.



### Note

Previous versions of the specification used the `/health` endpoint for both purposes. Now, this endpoint is optional and exposes the combination of the two health checks for backwards compatibility. Additionally, it is now at `/q/health`.

The liveness health check verifies that the services are in a healthy status: the application has not encountered an irrecoverable error. Examples of this could be a memory error or a deadlock in the code.

The readiness health check informs when services are ready to serve requests. This check must take into account any external service required for the proper functionality of the service. Examples of this check include waiting for a database or external services to become available.

## Adding Health Checks

Both the liveness and readiness checks implement the `HealthCheck` interface. This interface only contains one `call` method, which accepts no parameters and returns a `HealthCheckResponse`.

```
@FunctionalInterface
public interface HealthCheck {
 HealthCheckResponse call();
}
```

To create a liveness probe, you must create a class that implements the `HealthCheck` interface and annotate it with `@org.eclipse.microprofile.health.Liveness`. The following is an example class that implements the Liveness health check.

```
@Liveness
public class MyLivenessCheck implements HealthCheck {

 public HealthCheckResponse call() {
 // Construct the health response.
 }
}
```

The same applies to the readiness probe, using the `@org.eclipse.microprofile.health.Readiness` annotation.

```
@Readiness
public class MyReadinessCheck implements HealthCheck {

 @Override
 public HealthCheckResponse call() {
 // Construct the health response.
 }
}
```

## Constructing a HealthCheckResponse

To create an instance of the class `org.eclipse.microprofile.health.HealthCheckResponse` you can use the builder approach or the abbreviated approach.

The builder approach uses the `HealthCheckResponse.named()` method, which needs the name of the response. This method returns an instance of the `HealthCheckResponseBuilder` class, which you can use to further customize the response before building the `HealthCheckResponse`.

Once you have created the `HealthCheckResponseBuilder`, you must use one of its methods to set the state of the response: `up()`, `down()`, or `state(boolean up)`. Finally, use the method `build()` to construct the desired `HealthCheckResponse` instance.

The following is an example of a health check method.

```
@Override
public HealthCheckResponse call() {
 return HealthCheckResponse.named("my-health-check").up().build();
}
```

The abbreviated form uses the `up()` and `down()` static methods of the `HealthCheckResponse` class to create a named response directly. Following is the same example above in the abbreviated form.

```
@Override
public HealthCheckResponse call() {
 return HealthCheckResponse.up("my-health-check");
}
```

## Custom Health Data

The `HealthCheckResponse` object supports supplying additional data that the consuming end can use to further analyze the response status. You can add this supplementary information by using the `withData()` methods of the `HealthCheckResponseBuilder` class.

An example of providing this additional information follows.

```
public class CheckLoad implements HealthCheck {

 @Override
 public HealthCheckResponse call() {
 return HealthCheckResponse.named("load-average")
 .withData("average", "0,19")
 .up()
 .build();
 }
}
```

This extra data can include anything of interest to the receiving endpoint, for example:

- Metrics about the service or the platform
- Cause of the error
- Time to recover in case of error

## Health Checks in OpenShift

Red Hat OpenShift Container Platform (RHOC) uses readiness probes to determine whether it can route traffic to a container pod. It uses liveness probes to assess if the pod needs to be terminated and recreated.

The `openshift` extension automatically accounts for the health check probes defined in the application and adds them to the generated resources files. These generated resources include default configurations. You might want to change them in the `application.properties` file to customize the health checks.

For example:

```
quarkus.openshift.liveness-probe.initial-delay=20s ①
quarkus.openshift.liveness-probe.period=45s ②
quarkus.openshift.readiness-probe.initial-delay=2s ③
quarkus.openshift.readiness-probe.period=15s ④
```

- ➊ Time to delay the first request to check the service liveness.
- ➋ Time between liveness probes.
- ➌ Time to delay the first request to check the service readiness.
- ➍ Time between readiness probes.

## Integrated Health Checks

Some Quarkus extensions add health checks automatically by adding them to your application. For example, the `agroal` extension, which handles the datasource connection for persistence extensions, automatically adds a health check that monitors the database connection and reports DOWN if it is not available.



### References

#### MicroProfile Health

<https://download.eclipse.org/microprofile/microprofile-health-2.2/microprofile-health-spec.html>

## ► Guided Exercise

# Implementing a Health Check

In this exercise, you will activate health check capabilities in a microservice and monitor it in Red Hat OpenShift Container Platform (RHOCP) using a probe.

## Outcomes

You should be able to activate health check capabilities in a microservice implemented with Quarkus and monitor its health in RHOCP.

## Before You Begin

To use your provided RHOCP instance, you need the appropriate credentials supplied by your Red Hat Training Online Learning environment. Review the *Guided Exercise: Verifying OpenShift Credentials* section if you need guidance for finding your credentials.

On the **workstation** machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the application to check the health status is ready to modify.

```
[student@workstation ~]$ lab tolerance-health start
```

## Instructions

- ▶ 1. First, you will test the changes in your local environment, you must open the project in VSCodium and start the application in development mode.
  - 1.1. Open the VSCodium application on your workstation.
  - 1.2. Click **File > Open Folder** and browse to the `/home/student/D0378/labs/tolerance-health/quarkus-calculator-monolith/` directory.
  - 1.3. Click the **OK** button.
  - 1.4. Click **View > Command Palette**, write **Quarkus**, select **Quarkus: Debug current Quarkus project**.
- ▶ 2. Add the MicroProfile Health checks to the application using the `@org.eclipse.microprofile.health.Liveness` and `@org.eclipse.microprofile.health.Readiness` annotations.
  - 2.1. To add the MicroProfile Health dependency, bring up the Command Palette in the editor clicking **View > Command Palette** and type **Quarkus**. Select the option **Quarkus: add extensions to current project**.  
In the modal window type **health**, select the extension **SmallRye Health**.  
To finish selecting extensions, select **1 extension selected**.
  - 2.2. To control the liveness of your application, you must create a **LivenessResource** class, which allows you to disable liveness. In this case, control the liveness of your

application manually. In a real-world scenario, create a function to assess if your application is alive.

Create a file called `LivenessResource.java` in the `com.redhat.training` package, with the following content.

```
package com.redhat.training;

import javax.enterprise.context.ApplicationScoped;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/crash")
@ApplicationScoped
public class LivenessResource {
 private boolean alive = true;

 @GET
 @Produces(MediaType.TEXT_PLAIN)
 public String setCrash() {
 alive = false;

 return "Service not alive";
 }

 public boolean isAlive() {
 return alive;
 }
}
```

- 2.3. Create a liveness health check endpoint, which external tools can use to determine if your application is alive.

Create a `LivenessHealthResource.java` in the same package. Within this class, inject the `LivenessResource` class to monitor the status of the app.

The `LivenessHealthResource` class should look like the following:

```
package com.redhat.training;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Liveness;

@Liveness
@ApplicationScoped
public class LivenessHealthResource implements HealthCheck {
 @Inject
 LivenessResource liveness;

 @Override
```

```
public HealthCheckResponse call() {
 return liveness.isAlive() ? HealthCheckResponse.up("Liveness") :
HealthCheckResponse.down("Liveness");
}
```

- 2.4. Add a readiness probe health check endpoint, which external systems can use to determine if the application is ready to serve requests. To ease the readiness function, the response will only indicate that the application is ready once the readiness has been verified ten times.

Create a `ReadinessHealthResource.java` file in the same package. Within this class, inject the `LivenessResource` class to monitor the status of the app.

The `ReadinessHealthResource` class should look like the following:

```
package com.redhat.training;

import javax.enterprise.context.ApplicationScoped;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Readiness;

@Readiness
@ApplicationScoped
public class ReadinessHealthResource implements HealthCheck {
 private int counter = 0;

 @Override
 public HealthCheckResponse call() {
 return ++counter >= 10 ? HealthCheckResponse.up("Readiness") :
HealthCheckResponse.down("Readiness");
 }
}
```

- 3. Verify that the health checks are responding as expected when calling the `/q/health` endpoint.
- 3.1. Within a terminal, use the `curl` command to verify that the `/q/health` endpoint returns `DOWN` for the current status of the application.

```
[student@workstation ~]$ watch -d -n 2 curl -s http://localhost:8080/q/health
{
 "status": "DOWN",
 "checks": [
 {
 "name": "Liveness",
 "status": "UP"
 },
 {
 "name": "Readiness",
 "status": "DOWN"
 }
]
}
```

```

 }
]
}

```

Wait until the readiness count reaches the limit and the readiness health check reports UP.

```

{
 "status": "UP",
 "checks": [
 {
 "name": "Liveness",
 "status": "UP"
 },
 {
 "name": "Readiness",
 "status": "UP"
 }
]
}

```

- 3.2. In a new terminal use the `curl` command to call the `/crash` endpoint to make the application report the liveness health check as DOWN.

```
[student@workstation ~]$ curl http://localhost:8080/crash
Service not alive
```

- 3.3. Check in the previous terminal that the `/q/health` endpoint now reports DOWN, this time because of the liveness probe.

```

{
 "status": "DOWN",
 "checks": [
 {
 "name": "Liveness",
 "status": "DOWN"
 },
 {
 "name": "Readiness",
 "status": "UP"
 }
]
}

```

- 3.4. Stop the `curl` command in the first terminal pressing `Ctrl+C`.

► 4. Deploy the application in RHOCP to see how it incorporates health checks.

- 4.1. Add the Quarkus OpenShift dependency by bringing up the Command Palette in the editor clicking **View > Command Palette** and typing Quarkus. Select the option **Quarkus: add extensions to current project**.

In the modal window type `openshift`, select the extension **OpenShift**.

To finish selecting extensions, select 1 extension selected.

## 4.2. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

## 4.3. Log into the RHOCP cluster.

```
[student@workstation ~]$ oc login ${RHT_OCP4_MASTER_API} \
-u ${RHT_OCP4_DEV_USER} -p ${RHT_OCP4_DEV_PASSWORD}
Login successful
...output omitted...
```

## 4.4. Change to the exercise project.

```
[student@workstation ~]$ oc project tolerance-health-${RHT_OCP4_DEV_USER}
Now using project "tolerance-health-..."
```

## 4.5. Go to the project's directory.

```
[student@workstation ~]$ cd \
~/D0378/labs/tolerance-health/quarkus-calculator-monolith
```

## 4.6. Deploy the application into your RHOCP using the following command in the terminal.

```
[student@workstation quarkus-calculator-monolith]$ mvn clean package \
-Dquarkus.kubernetes.deploy=true \
-Dquarkus.openshift.expose=true \
-DskipTests
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

## 4.7. Verify that RHOCP does not finish deploying the application. Note that the suffix for the name of your pod will likely differ.

```
[student@workstation quarkus-calculator-monolith]$ oc get pods
NAME READY STATUS RESTARTS AGE
...output omitted...
quarkus-solver-1-hvps 0/1 Running 0 35s
...output omitted...
```

The ready column 0/1 means that the application pod is not ready yet, because its health checks are replying DOWN. Also notice that the status Running means that deployment is not finished until the application reports a healthy state.

After roughly one minute, list the pods again to view that the application pod is ready.

```
[student@workstation quarkus-calculator-monolith]$ oc get pods
NAME READY STATUS RESTARTS AGE
...output omitted...
quarkus-solver-1-hvps 1/1 Running 0 59s
...output omitted...
```

4.8. Close the VSCode and terminal windows.

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab tolerance-health finish
```

This concludes the guided exercise.

## ▶ Lab

# Implementing Fault Tolerance

In this lab, you will apply fault tolerance policies to the Quarkus Conference application.

## Outcomes

You should be able to apply different fault tolerance policies to Quarkus applications given identified potential faults.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this lab.

This command ensures that the necessary application source files are present the `~/DO378/labs/tolerance-review/` folder.

```
[student@workstation ~]$ lab tolerance-review start
```

## Instructions

1. Add liveness and readiness check for the `session` and `schedule` services. Both checks must always return a positive response. The name of the checks must be `Service is alive` for the liveness probe and `Service is ready` for the readiness probe.  
You can optionally add the default checks to the rest of the services.
2. The `Session` service reaches the `Speaker` service to enrich the speaker information. Apply fault tolerance policies to the `Session` service so that it reacts to `Speaker` service failures.  
Prepare the `Session` service to use fault tolerance features.
3. The `allSessions` method (`/sessions` endpoint) relies on the `Speaker` service to enrich speaker data. It is important that this service can respond with non-enriched data, if the `Speaker` service is not available.  
Create a fallback for the `allSessions` method and enable the fault tolerance policy to use that method if the original method fails to respond. Use the `SessionStore.findAllWithoutEnrichment` to obtain the sessions while skipping the `Speaker` service.
4. The `retrieveSession` method (`/sessions/{sessionId}` endpoint) depends on the `Speaker` service to enrich speaker information. Speaker information is important here, but response time is more significant.  
Create a fallback method for returning non-enriched speaker information when the `Speaker` service is unavailable. Utilize the `SessionStore.findByIdWithoutEnrichment` method to retrieve session information avoiding the `Speaker` service.  
To speed up further requests, when two consecutive requests to the `retrieveSession` method fail, return fallback responses for 30 seconds.
5. The `sessionSpeakers` endpoint method is used by other external clients. It is mandatory that this endpoint responds in less than one second or provides a failure.

For this lab we added a fake delay to the `SessionStore.findByIdWithEnrichedSpeakers` method, which will delay the enrichment of the speakers for one second.

6. When the session integration feature toggle is on, the `addSessionSpeaker` and `removeSessionSpeaker` endpoint methods try to validate the speaker against the `Speaker` service. This validation is important for eventual consistency. Your stakeholders have decided that if the `Speaker` service is unavailable, then those methods must fail.

The front-end application contains no page for updating speakers. Those endpoints are only used by third-party clients. Response speed is less important than eventual consistency.

Implement a retry policy on both methods. If they raise an exception, then continue retrying at a rate of one retry per second. Return a failure response if the method returns no successful response after 60 seconds.

7. The `updateSession` method is sensitive to race conditions. Protect that method from concurrent requests.

## Evaluation

Keep the `service-toggle.sh` script running during the evaluation with the `Speaker` service down. Restart the `Session` service before proceeding to revert all data updated during the exercise.

```

1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
4) Speaker 8) All

Choose an option: 4
Stopping session (10587)
2020-10-20 13:03:50,441 INFO [io.quarkus] (Shutdown thread) microservice-speaker
stopped in 0.017s
Service speaker stopped
Choose an option: 2
Stopping session (10581)
2020-10-20 13:03:51,043 INFO [io.quarkus] (Shutdown thread) microservice-session
stopped in 0.019s
Service session stopped
Choose an option: 2
Starting session
...output omitted...
Service session started

```

Grade your work by running the `lab tolerance-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab tolerance-review grade
```

## Finish

Finalize the first three services using the `service-toggle.sh` script.

```
1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
4) Speaker 8) All

Choose an option: 1
Stopping postgresql
f5cd....9a32
Service postgresql stopped
...output omitted...
Choose an option: 2
Stopping session (29462)
2020-10-19 11:44:59,096 INFO [io.quarkus] (Shutdown thread) microservice-session
stopped in 0.013s
Service session stopped
...output omitted...
Choose an option: 3
Stopping schedule (29809)
2020-10-19 11:45:02,103 INFO [io.quarkus] (Shutdown thread) microservice-schedule
stopped in 0.025s
Service schedule stopped
Choose an option: 10
```

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab tolerance-review finish
```

This concludes the lab.

## ► Solution

# Implementing Fault Tolerance

In this lab, you will apply fault tolerance policies to the Quarkus Conference application.

## Outcomes

You should be able to apply different fault tolerance policies to Quarkus applications given identified potential faults.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this lab.

This command ensures that the necessary application source files are present the `~/D0378/labs/tolerance-review/` folder.

```
[student@workstation ~]$ lab tolerance-review start
```

## Instructions

- Add liveness and readiness check for the `session` and `schedule` services. Both checks must always return a positive response. The name of the checks must be `Service is alive` for the liveness probe and `Service is ready` for the readiness probe.  
You can optionally add the default checks to the rest of the services.
  - Add the `smallrye-health` extension to all projects. You can do it by executing the `mvn quarkus:add-extension -Dextension=smallrye-health` in each microservice's folder, or you can directly use the parent pom provided.

```
[student@workstation ~]$ cd ~/D0378/labs/tolerance-review
[student@workstation tolerance-review]$ mvn quarkus:add-extension \
-Dextension="smallrye-health" -pl 'microservice-schedule,microservice-session'
[INFO] Scanning for projects...
...output omitted...
[INFO] Reactor Summary for microservice-session 1.0.0-SNAPSHOT:
[INFO]
[INFO] microservice-session SUCCESS [6.342 s]
[INFO] microservice-schedule SUCCESS [0.637 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- Open the parent project in VSCodium.

```
[student@workstation tolerance-review]$ codium .
```

- Create liveness and readiness check classes for both services.

In the `microservice-schedule` and `microservice-session` projects, create a file named `LivenessCheck.java` in the `org.acme.conference.schedule` and `org.acme.conference.session` packages, respectively. Except for the package name, the contents of the file will be the same for both microservices.

```
package org.acme.conference.schedule;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Liveness;

import javax.enterprise.context.ApplicationScoped;

@Liveness
@ApplicationScoped
public class LivenessCheck implements HealthCheck {
 @Override
 public HealthCheckResponse call() {
 return HealthCheckResponse.up("Service is alive");
 }
}
```

- 1.4. Create a file named `ReadinessCheck.java` in the `org.acme.conference.schedule` package.

```
package org.acme.conference.schedule;

import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Readiness;

import javax.enterprise.context.ApplicationScoped;

@Readiness
@ApplicationScoped
public class ReadinessCheck implements HealthCheck {
 @Override
 public HealthCheckResponse call() {
 return HealthCheckResponse.up("Service is ready");
 }
}
```

Copy those new classes to the `microservice-session` project.  
You must update the package declaration to match the right package:  
`org.acme.conference.session`.

- 1.5. Start the `Session`, `Schedule`, and `Speaker` services. Before starting the `Session` service, you must start the `Session's PostgreSQL`. Provided is a convenience `service-toggle.sh` script to start and stop services easily.

```
[student@workstation tolerance-review]$./service-toggle.sh
1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
```

```

4) Speaker 8) All
Choose an option: 1
Starting postgresql
...output omitted...
Service postgresql started
Choose an option: 2
Starting session
...output omitted...
Service session started
Choose an option: 3
Starting schedule
...output omitted...
Service schedule started
Choose an option: 4
Starting speaker
...output omitted...
Service speaker started

```

**Note**

You can see several errors while pulling container images from different image catalogs. Those errors are safe to ignore. The continuation prompt might be hidden between Quarkus output. Use the **Status** option to review the status of the services. Correctly started services include the process id or the container id next to the service name.

Leave this script running, you will use it throughout the exercise.

- 1.6. Verify that both health checks work properly. Open a new terminal, and execute the following `curl` command to retrieve the checks for the `session` service.

```
[student@workstation ~]$ curl localhost:8081/q/health
{
 "status": "UP",
 "checks": [
 {
 "name": "Service is alive",
 "status": "UP"
 },
 {
 "name": "Database connections health check",
 "status": "UP"
 },
 {
 "name": "Service is ready",
 "status": "UP"
 }
]
}
```

Note that both readiness and liveness checks are shown in the response. You must obtain the same results when checking the `schedule` service at port 8083.

Alternatively, you can use the dedicated liveness and readiness endpoints to validate the checks.

```
[student@workstation ~]$ curl localhost:8083/q/health/live
{
 "status": "UP",
 "checks": [
 {
 "name": "Service is alive",
 "status": "UP"
 }
]
}
[student@workstation ~]$ curl localhost:8083/q/health/ready
{
 "status": "UP",
 "checks": [
 {
 "name": "Service is ready",
 "status": "UP"
 },
 {
 "name": "Database connections health check",
 "status": "UP"
 }
]
}
```

2. The Session service reaches the Speaker service to enrich the speaker information. Apply fault tolerance policies to the Session service so that it reacts to Speaker service failures. Prepare the Session service to use fault tolerance features.

#### 2.1. Add the smallrye-fault-tolerance extension to the Session service.

```
[student@workstation ~]$ cd ~/DO378/labs/tolerance-review
[student@workstation tolerance-review]$ mvn quarkus:add-extension \
-Dextension="smallrye-fault-tolerance" -pl 'microservice-session'
[INFO] Scanning for projects...
...output omitted...
Extension io.quarkus:quarkus-smallrye-fault-tolerance has been installed
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

3. The allSessions method (/sessions endpoint) relies on the Speaker service to enrich speaker data. It is important that this service can respond with non-enriched data, if the Speaker service is not available.

Create a fallback for the allSessions method and enable the fault tolerance policy to use that method if the original method fails to respond. Use the `SessionStore.findAllWithoutEnrichment` to obtain the sessions while skipping the Speaker service.

- 3.1. Open the `~/D0378/labs/tolerance-review/microservice-session/src/main/java/org/acme/conference/session/SessionResource.java` file. Create a `allSessionsFallback` method in the `SessionResource` class. This method must have the same erasure as the `allSessions` method, but no annotations. Return the sessions provided by the `findAllWithoutEnrichment` method.

```
public Collection<Session> allSessionsFallback() throws Exception {
 logger.warn("Fallback sessions");
 return sessionStore.findAllWithoutEnrichment();
}
```

- 3.2. Annotate the `allSessions` method with the `org.eclipse.microprofile.faulttolerance.Fallback` annotation. Provide the name of the previous method in the `fallbackMethod` attribute of the extension.

```
import org.eclipse.microprofile.faulttolerance.Fallback;
...output omitted...
@GET
@Produces(MediaType.APPLICATION_JSON)
@Fallback(fallbackMethod="allSessionsFallback")
public Collection<Session> allSessions() throws Exception {
 return sessionStore.findAll();
}
```

- 3.3. Validate that the `Session` service returns the service list despite the `Speaker` service being unavailable.

Shut down the `Speaker` service in the `service-toggle.sh` script.

```
1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
4) Speaker 8) All
Choose an option: 4
Stopping speaker (21945)
2020-10-14 22:38:27,823 INFO [io.quarkus] (Shutdown thread) microservice-speaker
stopped in 0.012s
Service speaker stopped
```

Use the `curl` command to get the `http://localhost:8081/sessions` URL and observe that the sessions appear, despite the missing `Speaker` service. Also note that the terminal where the `Session` service is running shows the `Fallback sessions` message.

```
[student@workstation tolerance-review]$ curl -s localhost:8081/sessions | jq
[
 {
 "id": "s-1-1",
 "schedule": 1,
 "speakers": [
 {
 "id": 1,
 "name": "Emmanuel",
```

```

 "uuid": "s-1-1"
 }
]
},
...output omitted...
]
```

Re-enable the Speaker service.

```

Choose an option:
1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
4) Speaker 8) All
Choose an option: 4
Starting speaker
[INFO] Scanning for projects...
...output omitted...
Service speaker started

```

- The `retrieveSession` method (`/sessions/{sessionId}` endpoint) depends on the Speaker service to enrich speaker information. Speaker information is important here, but response time is more significant.

Create a fallback method for returning non-enriched speaker information when the Speaker service is unavailable. Utilize the `SessionStore.findByIdWithoutEnrichment` method to retrieve session information avoiding the Speaker service.

To speed up further requests, when two consecutive requests to the `retrieveSession` method fail, return fallback responses for 30 seconds.

1. Create a fallback method named `retrieveSessionFallback` with the same signature as the `retrieveSession` method.

```

public Response retrieveSessionFallback(final String sessionId) {
 logger.warn("Fallback session");
 return sessionStore.findByIdWithoutEnrichment(sessionId)
 .map(s -> Response.ok(s).build())
 .orElseThrow(NotFoundException::new);
}

```

2. Annotate the `retrieveSession` method with a `Fallback` annotation, providing the `retrieveSessionFallback` method name as the only parameter.

```

@GET
@Path("/{sessionId}")
@Produces(MediaType.APPLICATION_JSON)
@Fallback(fallbackMethod="retrieveSessionFallback")
public Response retrieveSession(@PathParam("sessionId") final String sessionId)
{

```

3. Annotate the same method with a `CircuitBreaker` annotation. Use the following values to define requests window size of 2 consecutive invocations and a circuit opening duration of 30 seconds.

```

import org.eclipse.microprofile.faulttolerance.CircuitBreaker;
...output omitted...
@GET
@Path("/{sessionId}")
@Produces(MediaType.APPLICATION_JSON)
@Fallback(fallbackMethod="retrieveSessionFallback")
@CircuitBreaker(requestVolumeThreshold = 2, failureRatio = 1, delay = 30_000)
public Response retrieveSession(@PathParam("sessionId") final String sessionId) {

```

#### 4.4. Validate the fallback and circuit breaker features.

With the Speaker service up, load the URL `http://localhost:8081/sessions/s-1-1` with a `curl` command. Observe that the Session service enriches the speaker name (contains the family name).

```

[student@workstation tolerance-review]$ curl -s \
 http://localhost:8081/sessions/s-1-1 | jq
{
 "id": "s-1-1",
 "schedule": 1,
 "speakers": [
 {
 "id": 1,
 "name": "Emmanuel Bernard",
 "uuid": "s-1-1"
 }
]
}

```

Use the `service-toggle.sh` script to shut down the Speaker service. Repeat the same `curl` command and observe that the speaker name does not contain the family name. Repeat the command again to trigger the circuit to open.

```

[student@workstation tolerance-review]$ curl -s \
 http://localhost:8081/sessions/s-1-1 | jq
{
 "id": "s-1-1",
 "schedule": 1,
 "speakers": [
 {
 "id": 1,
 "name": "Emmanuel",
 "uuid": "s-1-1"
 }
]
}

```

Start up the Speaker service again. Repeat the command and observe the speaker information is still not enriched. This is because the circuit is still open for 30 seconds.

**Note**

You have a 30 seconds window to start the service and execute the request again. If you obtain an enriched response, then shut down the Speaker service again and repeat the previous query twice to trigger the circuit open. Then, start the Speaker service again and repeat the request. If you still get an enriched request, then you may consider increasing the delay in the `CircuitBreaker` annotation to a higher value.

Wait 30 seconds and run the command again. The speaker family name shows again, proving the circuit is closed and the service is enriching the speaker information.

5. The `sessionSpeakers` endpoint method is used by other external clients. It is mandatory that this endpoint responds in less than one second or provides a failure.

For this lab we added a fake delay to the `SessionStore.findByIdWithEnrichedSpeakers` method, which will delay the enrichment of the speakers for one second.

- 5.1. Use the `Timeout` annotation on the `sessionSpeakers` method. Use a parameter value of 1000 milliseconds.

```
import org.eclipse.microprofile.faulttolerance.Timeout;
...output omitted...
@GET
@Path("/{sessionId}/speakers")
@Produces(MediaType.APPLICATION_JSON)
@Timeout(1000)
public Response sessionSpeakers(@PathParam("sessionId") final String sessionId) {
```

- 5.2. Validate the timeout by directly using the endpoint with the `curl` command. Because the `features.session-integration` is enabled in the `application.properties` file, all requests to the `/{sessionId}/speakers` endpoint will try to enrich speaker data and will suffer from the synthetic delay. Because of that delay, the timeout interrupts the request and returns a failure.

```
[student@workstation tolerance-review]$ (time curl \
-s localhost:8081/sessions/s-1-1/speakers) | grep "title"
<title>Internal Server Error ...
org.eclipse.microprofile.faulttolerance.exceptions.TimeoutException:
Timeout[org.acme.conference.session.SessionResource#sessionSpeakers] timed out</
title>
real 0m1.029s
user 0m0.003s
sys 0m0.003s
```

Open the file `src/main/resources/application.properties` of the Session service and disable the `features.session-integration` feature toggle by setting it to `false`. Now that the service response is not enriched, the delay does not apply, and the timeout is not raised. The first request after the feature toggle change takes more time to respond due to live code reload.

```
[student@workstation tolerance-review]$ (time curl \
-s localhost:8081/sessions/s-1-1/speakers)
[{"id":1,"name":"Emmanuel","uuid":"s-1-1"}]
real 0m0.019s
user 0m0.004s
sys 0m0.003s
```

Re-enable the `features.session-integration` toggle before continuing the exercise.

6. When the session integration feature toggle is on, the `addSessionSpeaker` and `removeSessionSpeaker` endpoint methods try to validate the speaker against the Speaker service. This validation is important for eventual consistency. Your stakeholders have decided that if the Speaker service is unavailable, then those methods must fail.

The front-end application contains no page for updating speakers. Those endpoints are only used by third-party clients. Response speed is less important than eventual consistency.

Implement a retry policy on both methods. If they raise an exception, then continue retrying at a rate of one retry per second. Return a failure response if the method returns no successful response after 60 seconds.

- 6.1. Add the `Retry` annotation to both endpoint methods. Use the `maxRetries` and `delay` options to declare the retries limits.

```
import org.eclipse.microprofile.faulttolerance.Retry;
...output omitted...
@PUT
@Path("/{sessionId}/speakers/{speakerId}")
@Produces(MediaType.APPLICATION_JSON)
@Retry(maxRetries=60, delay=1_000)
public Response addSessionSpeaker(@PathParam("sessionId") final String sessionId,
 @PathParam("speakerName") final String speakerName) {
 ...output omitted...
}
@DELETE
@Path("/{sessionId}/speakers/{speakerId}")
@Produces(MediaType.APPLICATION_JSON)
@Retry(maxRetries=60, delay=1_000)
public Response removeSessionSpeaker(@PathParam("sessionId") final String
 sessionId, @PathParam("speakerName") final String speakerName) {
 ...output omitted...
```

- 6.2. To validate the retry policy, shut down the Speaker service using the `service-toggle.sh` script.

```
Choose an option:
1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
4) Speaker 8) All
Choose an option: 4
Stopping speaker (16244)
...output omitted...
Service speaker stopped
```

In the terminal, execute the following `curl` command to trigger the association of a new speaker to a session. Because the `Speaker` service is down, the retry policy kicks in and the request takes more than a minute to complete.

```
[student@workstation tolerance-review]$ time curl \
-X PUT "http://localhost:8081/sessions/s-1-1/speakers/Alex"
...output omitted...
Resulted in: org.jboss.resteasy.spi.UnhandledException:
javax.ws.rs.ProcessingException: RESTEASY004655: Unable to invoke request:
org.apache.http.conn.HttpHostConnectException: Connect to localhost:8082
[localhost/127.0.0.1, localhost/0:0:0:0:0:0:1] failed: Connection refused
(Connection refused)
...output omitted...

real 1m2,274s
user 0m0,003s
sys 0m0,009s
```

Repeat the same `curl` command. This time, while waiting for the response, go to the terminal running the `service-toggle.sh` script and start up the `Speaker` service. Observe that the `curl` command obtains a successful enriched response the moment the `Speaker` service starts.

```
[student@workstation tolerance-review]$ time curl \
-X PUT "http://localhost:8081/sessions/s-1-1/speakers/Alex"
{"id":"s-1-1","schedule":1,"speakers":[{"id":1,"name":"Emmanuel","uuid":"s-1-1"}, {"id":3,"name":"Alex Soto","uuid":"s-1-3"}]}

real 0m16.345s
user 0m0.006s
sys 0m0.004s
```

7. The `updateSession` method is sensitive to race conditions. Protect that method from concurrent requests.

- 7.1. Limit the `updateSession` method to a single connection by using the `Bulkhead` annotation.

```
import org.eclipse.microprofile.faulttolerance.Bulkhead;
...output omitted...
@PUT
@Path("/{sessionId}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Bulkhead(1)
public Response updateSession(@PathParam("sessionId") final String sessionId,
 final Session session) {
...output omitted...
```

- 7.2. Validate the concurrency protection. Use the provided `parallel.sh` script to run 10 parallel updates to the protected endpoint. Verify that a single request succeeds while the rest are aborted by the bulkhead.

```
[student@workstation tolerance-review]$./parallel.sh \
'curl -s -o /dev/null -w "%{http_code}\n" -X PUT localhost:8081/sessions/s-1-1 \
-H "Content-Type: application/json" -d "{\"id\":\"s-1-1\", \"schedule\":1}''' 10
500
500
500
500
500
500
500
500
500
200
```

## Evaluation

Keep the `service-toggle.sh` script running during the evaluation with the Speaker service down. Restart the Session service before proceeding to revert all data updated during the exercise.

```
1) Session's PostgreSQL 5) Votes's MongoDB 9) Status
2) Session 6) Vote 10) Exit
3) Schedule 7) Frontend
4) Speaker 8) All
Choose an option: 4
Stopping session (10587)
2020-10-20 13:03:50,441 INFO [io.quarkus] (Shutdown thread) microservice-speaker
stopped in 0.017s
Service speaker stopped
Choose an option: 2
Stopping session (10581)
2020-10-20 13:03:51,043 INFO [io.quarkus] (Shutdown thread) microservice-session
stopped in 0.019s
Service session stopped
Choose an option: 2
Starting session
...output omitted...
Service session started
```

Grade your work by running the `lab tolerance-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab tolerance-review grade
```

## Finish

Finalize the first three services using the `service-toggle.sh` script.

1) Session's PostgreSQL	5) Votes's MongoDB	9) Status
2) Session	6) Vote	10) Exit
3) Schedule	7) Frontend	

```
4) Speaker 8) All
Choose an option: 1
Stopping postgresql
f5cd....9a32
Service postgresql stopped
...output omitted...
Choose an option: 2
Stopping session (29462)
2020-10-19 11:44:59,096 INFO [io.quarkus] (Shutdown thread) microservice-session
stopped in 0.013s
Service session stopped
...output omitted...
Choose an option: 3
Stopping schedule (29809)
2020-10-19 11:45:02,103 INFO [io.quarkus] (Shutdown thread) microservice-schedule
stopped in 0.025s
Service schedule stopped
Choose an option: 10
```

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab tolerance-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- Fault tolerance policies allow the creation of resilient microservices. Quarkus offers five policies out-of-the-box: **Circuit Breaker**, **Bulkhead**, **Fallback**, **Retry**, and **Timeout**.
- Other implementations such as **Red Hat OpenShift Service Mesh**, offer a higher level implementation of fault tolerance policies, relieving the services that implement them.
- Quarkus, through **SmallRye Health**, offers a **MicroProfile Health** specification implementation inline with Quarkus principles. Health checks are easily configurable and extendable by service developers.
- Health checks are automatically used when deploying Quarkus services to OpenShift Container Platform, as **liveness** and **readiness** probes.
- Many extensions automatically detect health extensions and add health check for their related services, such as databases or topics.

## Chapter 6

# Building and Deploying Native Quarkus Applications

### Goal

Describe Quarkus Native and its deployment on OpenShift Container Platform.

### Objectives

- Build native applications using Quarkus.
- Create container images for Quarkus native applications and deploy them in OpenShift Container Platform.

### Sections

- Building Native Applications with Quarkus (and Guided Exercise)
- Containerizing Quarkus Native Applications (and Guided Exercise)

# Building Native Applications with Quarkus

## Objectives

After completing this section, you should be able to build native applications using Quarkus.

## GraalVM and Native Compilation

Java applications traditionally run in a Java virtual machine (JVM), which interprets Java code that has been precompiled into bytecode. The JVM will read the bytecode and translate it into machine code during execution. This process has been optimized for decades, but still has an overhead, because it requires a virtual machine, creating a need for additional memory resources, as well as having an initial CPU cost. Ahead-of-time compilation implies compiling from the Java code directly into the CPU's machine code during build time, instead of during execution.

The process of compiling into machine code is called native compilation. The result is that the JVM is no longer required, saving memory resources and the initial CPU cost, thus reducing the footprint of the application.

Quarkus uses GraalVM for ahead-of-time compilation of Java applications. GraalVM is a Java VM and an SDK, which is owned by Oracle and provides a vast set of tools. From these tools, Quarkus only uses GraalVM Native Image, which is the technology that produces the executable binary files. The goal of native compilation is to vastly improve startup time, the memory footprint, and the initial performance of the application.

During compilation time, GraalVM performs extensive static analysis of the code, and packages classes, dependencies, runtime libraries and statically linked native code from its JDK.

In the static analysis, the compilation strips down all code considered unreachable, including classes, methods, and fields. The process makes the application independent of the Java VM. The result is lower runtime memory use and a startup time that can be measured in milliseconds.

Quarkus improves the usability of GraalVM by tracking down the uses of the reflection API, resources, and other special cases. Normally, you would have to manually track these down and create configuration files that enable a proper native build.

## Choosing Mandrel

Red Hat and the GraalVM community created **Mandrel**, a downstream distribution of GraalVM. The goal of creating Mandrel is to provide an open source and Red Hat supported distribution of GraalVM, maintaining stability and reliability.

Quarkus only uses GraalVM for native compilation. This means that other components of GraalVM, such as GraalVM Compiler, the Truffle Language Implementation framework, and its different runtimes are not used by the Quarkus project. Taking advantage of this, Mandrel uses OpenJDK, instead of GraalVM's JDK, and packages it with GraalVM Native Image, improving its maintainability and its focus.

## Code Caveats for Native Applications

When developing Quarkus applications with the goal of deploying them as native applications, it is important to know that some common Java features work differently or are not supported. The following are some special cases for native applications.

### Injecting Within Native Tests

Using Quarkus' dependency injection to inject into the tests does not work properly during native tests. This happens because the native executable that is tested is not running within a JVM, unlike the test code which runs in a JVM process.

### Registering for Reflection

Quarkus attempts, during native build time, to register all classes that are used in the code. A class, method, or field can be used via reflection, and not become part of the call tree, prompting GraalVM to eliminate them from the native executable. A common case of this is when working with serialization libraries, such as JsonB.

For this situation, Quarkus provides the `io.quarkus.runtime.annotations.RegisterForReflection` annotation, which helps during build time to identify what cannot be ignored or left out.

```
@Path("/person")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ExampleResource {
 @RegisterForReflection❶
 public static class Data {
 public int number;
 public String name;

 public Data(int number, String name) {
 this.number = number;
 this.name = name;
 }
 }

 private List<ExampleResource.Data> list = new ArrayList<>();

 public ExampleResource() {
 list.add(new Data(5, "a"));
 list.add(new Data(3, "b"));
 }

 @GET
 public Response list() {
 return Response.ok(list).build();❷
 }
}
```

- ❶ The annotation tells Mandrel that the `ExampleResource.Data` class will be used by reflection. Without it, GraalVM will not know it was required to be registered for reflection, and the class will produce either an error or an empty value when reflection uses it.

- ❷ The `Response` class uses reflection to serialize the object into JSON. In this scenario, without the previous annotation, the result would be an empty list.

## Avoiding Private Access Modifiers

Encapsulation has been a part of Java since its conception, and the use of private modifiers in fields is common place. However, Quarkus' dependency injection implementation has limitations that take place when building a native executable.

If a field is private, then Quarkus DI needs reflection during runtime to access it. If the member is not registered for reflection, then the reflective operation results in an unsupported operation. For this reason, one recommendation is to use package-private modifiers. This applies to injection fields, constructors, initializers, observer methods, producer methods and fields, disposers and interceptor methods.

An alternative is to keep using private modifiers, but rely on constructor injection.

## Using Third-party Libraries with Reflection

When using a third-party jar that uses reflection, and without a Quarkus extension, you cannot use the `@RegisterForReflection` annotation.

In this scenario, the solution is to create a `src/main/resources/reflection-config.json` file, which includes the classes that will require reflection.

```
[
 {
 "name" : "com.redhat.library.SpecialClass",
 "allDeclaredConstructors" : true,
 "allPublicConstructors" : true,
 "allDeclaredMethods" : true,
 "allPublicMethods" : true,
 "allDeclaredFields" : true,
 "allPublicFields" : true
 }
]
```

You must also configure the build arguments to use the reflection configuration file. This means adding the following line in the `application.properties` file.

```
quarkus.native.additional-build-args =-H:ReflectionConfigurationFiles=reflection-
config.json
```

## Use Cases for Native Applications

Although native builds bring advantages, its complexity can outweigh the benefits in certain circumstances. Use cases for native applications include production environments where low memory footprint, fast start-up time and initial CPU performance are key.

### Serverless architectures

In Function as a Service, as well as other serverless architectures, the startup time of an application becomes critical because during cold starts (when there are no running containers) and scaling up, the startup time can become a bottleneck.

## Applications with a high density memory requirement

In production environments that require running multiple instances of an application, often within the same host, where the bottleneck is not the CPU or IO.

## Deployments in orchestration platforms

When using technologies such as Red Hat OpenShift or Kubernetes, it is important to efficiently use memory and to enable quick startups for scaling properly.

## Command-line applications

Similarly to serverless applications, the life expectancy of common command-line applications is short, and startup time can be considered important.

# Disadvantages of Native Applications

There are great reasons to use native applications, but it is also important to know when native applications are not the best solution. To understand which scenarios are not ideal for native applications, it is useful for you to know its disadvantages.

## No JIT (Just In Time) compilation

For long running applications, the Java VM can further optimize critical code by performing dynamic analysis of the application during runtime. In this situation, performance can be improved by using the JVM over native builds.

## Code limitations

The lack of support for some Java constructs, including private access modifiers, reflection, and others.

## Slow build time

Building a native application is a resource intensive and slow process. This makes native compilation better suited for production use, and not desirable during the development process.

## Profiling and debugging does not work with Java tools

This means that you have to use gdb for debugging, which is not standard for Java developers, making it undesirable to use native builds during development. It can also make it complex to debug an application which behaves correctly using the JVM, but has bugs when running as native.

# Building a Native Application

The following is the fundamental command for building a native application.

```
[user@host ~]$ mvn package -Pnative
```

The command is dependent on Mandrel.

The easiest method for using Mandrel is to use its containerized image. You can tell Quarkus to use a Mandrel container to build the native executable by passing the `quarkus.native.container-build` parameter set to `true`. If you want to specify the container image used for the build process, you can specify it by setting the image name using the `quarkus.native.builder-image` parameter.

Another useful parameter is `quarkus.native.container-runtime`, which is used to specify the container runtime. The most common values are `podman` and `docker`.

These values can be passed as system variables, using the `application.properties` file, or via any other configuration source.

An example using system variables.

```
[user@host ~]$ mvn package -Pnative \
-Dquarkus.native.container-build=true \
-Dquarkus.native.container-runtime=podman \
-Dquarkus.native.builder-image=registry.access.redhat.com/quarkus/mandrel-20-rhel8
```

The build process generates multiple files and places them within a `target` / subdirectory found in the root folder of the project. The native executable file can be found inside, with a suffix of `-RUNNER`.

## Containerizing the Native Application

A common way of creating a containerized native application is to use the container image extension. The extension currently supports `Jib`, `Docker` and `s2i`. This implies running the build command discussed previously, but adding the parameter for `quarkus.container-image.build` set to true.

```
[user@host ~]$ mvn package -Pnative -Dquarkus.container-image.build=true
```

An alternative to build a container image using `podman` is, after building the native executable, to use the `Dockerfile.native` file found in the `src/main/docker` directory. Inside of the root directory of the project, you can run the following `podman` command to build the image.

```
[user@host ~]$ podman build -f src/main/docker/Dockerfile.native \
-t quarkus-course/native-example .
```

You can also build the native application within the image building process, by using the `Dockerfile.multistage` file instead.



### References

For more information on native executables, refer to the Product Documentation for Red Hat Build of Quarkus at  
[https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.7/html-single/compiling\\_your\\_quarkus\\_applications\\_to\\_native\\_executables/index](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.7/html-single/compiling_your_quarkus_applications_to_native_executables/index)

## ► Guided Exercise

# Building Native Applications with Quarkus

In this exercise, you will configure native compilation for a Quarkus microservice, build the application using a containerized Mandrel executable, and run the native application to see its functionality.

## Outcomes

You should be able to build native Quarkus applications.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that you have a PostgreSQL database container running and downloads the files required for the exercise.

```
[student@workstation ~]$ lab native-build start
```



### Note

Notice that the startup script starts a container that is running a PostgreSQL database. You can use the following command to verify that it is available:

```
ss -tulpn | grep 5432
```

If you exit the developer environment and return later, you may find that the database is no longer running. Execute the startup script again to resolve this issue.

## Instructions

- 1. Open the projects found in `/home/student/D0378/labs/native-build/expense-service` using Codium.

- 1.1. Open a terminal and type the following.

```
[student@workstation ~]$ codium ~/D0378/labs/native-build/expense-service
```

- 2. Configure the native compilation of the application, using Mandrel.

- 2.1. Open the `src/main/resources/application.properties` file.

- 2.2. Modify the `application.properties` file to include the following contents.

```
...output omitted...
native build configuration
quarkus.native.container-build=true①
quarkus.native.builder-image=registry.access.redhat.com/quarkus/mandrel-20-rhel8②
quarkus.native.container-runtime=podman③
```

- ① The environment lacks an installation of Mandrel, therefore we configure Quarkus to use a container for building the application.
- ② There are a diverse group of available images. This is a free Mandrel base image provided by Red Hat.
- ③ The workstation has podman installed, which means you must specify the use of Podman as the container runtime.

► 3. Build the native application.

- 3.1. Open a terminal window. Change the directory to /home/student/D0378/labs/native-build/expense-service.

```
[student@workstation ~]$ cd /home/student/D0378/labs/native-build/expense-service
```

- 3.2. Start the application using the development profile, with the intention of verifying the startup time.

```
[student@workstation expense-service]$ mvn quarkus:dev
...output omitted...
2020-10-13 15:27:02,439 INFO [io.quarkus] (Quarkus Main Thread) expense-restful-service 1.0-SNAPSHOT on JVM (powered by Quarkus 1.7.5.Final-redhat-00007) started
in 3.604s. Listening on: http://0.0.0.0:8081
2020-10-13 15:27:02,442 INFO [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2020-10-13 15:27:02,442 INFO [io.quarkus] (Quarkus Main Thread) Installed
features: [agroal, cdi, hibernate-orm, hibernate-orm-panache, jdbc-postgresql,
mutiny, narayana-jta, resteasy, resteasy-jsonb, smallrye-context-propagation,
smallrye-openapi, swagger-ui]
...output omitted...
```



### Warning

The native compilation process uses a significant amount of RAM. You can close Codium to free the resources used before starting build the process.

- 3.3. Stop the application by pressing **Ctrl+C**. Proceed to start the build process. This will take several minutes.

```
[student@workstation expense-service]$ mvn package -Pnative
...output omitted...
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
260433ms
[INFO] -----
[INFO] BUILD SUCCESS
```

```
[INFO] -----
[INFO] Total time: 04:26 min
[INFO] Finished at: 2020-10-13T01:19:59-05:00
[INFO] -----
```

- 4. Run the new native executable and verify it works properly.

- 4.1. Execute the application. The startup time will be two orders of magnitude faster than it was before.

```
[student@workstation expense-
service]$ target/expense-restful-service-1.0-SNAPSHOT-runner

--/ _ \ / / / _ | / _ \ ///_ / / / _/
-/ / / / / / _ | / , _/ , < / /_ \ \ \
--_\ __/_ / |/_/|/_/|_|\ __/_ /
...output omitted...
2020-10-13 01:11:34,224 INFO [io.quarkus] (main) expense-restful-service 1.0-
SNAPSHOT native (powered by Quarkus 1.7.5.Final-redhat-00001) started in 0.078s.
Listening on: http://0.0.0.0:8080
2020-10-13 01:11:34,224 INFO [io.quarkus] (main) Profile prod activated.
2020-10-13 01:11:34,224 INFO [io.quarkus] (main) Installed features: [agroal,
cdi, hibernate-orm, hibernate-orm-panache, jdbc-postgresql, mutiny, narayana-
jta, resteasy, resteasy-jsonb, smallrye-context-propagation, smallrye-openapi,
swagger-ui]
```

- 4.2. In another terminal, make a request to the service and verify it returns a valid response.

```
[student@workstation ~]$ curl -s localhost:8080/expenses/ | jq
[{"id":1,"amount":150.50,"name":"Groceries","paymentMethod":"CASH"}, {"id":2,"amount":25.00,"name":"Civilization VI","paymentMethod":"CREDIT_CARD"}]
```

- 4.3. Stop the application by pressing **Ctrl+C**.

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab native-build finish
```

This concludes the guided exercise.

# Containerizing Quarkus Native Applications

## Objectives

After completing this section, you should be able to create container images for Quarkus native applications and deploy them in OpenShift Container Platform.

## Containerizing Native Quarkus Applications

You can create container images for native Quarkus applications, which is the easiest way to take advantage of the low startup time of Quarkus microservices. This low startup time makes it easier to scale out capacity of your applications.

Container images only include the binary executable, saving all the extra space occupied by the JVM and the unused libraries, making the container image thinner and quicker for pods to scale in a container platform.

## Supported Container Building Methods

There are three extensions that allow you to create container images for your Quarkus applications.

### Docker

To build container images using Docker you must add the `container-image-docker` extension. This extension by default uses the Dockerfiles auto generated by Quarkus initializers in the `src/main/docker` folder. One of the Dockerfiles is for the JVM mode and the other is for native mode.

Add this extension to your project by using the `add-extension` command of the Quarkus plug-in.

```
[user@host ~]$ mvn quarkus:add-extension -Dextensions="container-image-docker"
```

Alternatively, add the dependency manually to your `pom.xml` file.

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-container-image-docker</artifactId>
</dependency>
```

### Jib

The `container-image-jib` extension uses the Jib tool to build the container image. Jib is a library and Maven/Gradle plug-in, which creates reproducible builds of container images. Jib also layers all application dependencies so rebuilds and publish are extremely fast.

Add this extension to your project by using the `add-extension` command of the Quarkus plug-in.

```
[user@host ~]$ mvn quarkus:add-extension -Dextensions="container-image-jib"
```

Alternatively, add the dependency manually to your pom.xml file.

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-container-image-jib</artifactId>
</dependency>
```

## Source2Image

The container-image-s2i extension uses the Source2Image tool to build the container image. This extension copies the existing artifacts into a builder image, which outputs a container image. This extension depends on the BuildConfig and ImageStream OpenShift Container Platform resources, but they can be autogenerated by the Quarkus Kubernetes extension.

Add this extension to your project by using the add-extension command of the Quarkus plugin.

```
[user@host ~]$ mvn quarkus:add-extension -Dextensions="container-image-s2i"
```

Alternatively, add the dependency manually to your pom.xml file.

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-container-image-s2i</artifactId>
</dependency>
```

## Building Container Images

Regardless of the method you choose to create container images, the following is the command to create a container image.

```
[user@host ~]$ mvn package -Pnative -Dquarkus.container-image.build=true
```

This command creates a local container image that you can run or push to a container registry. To push your container image at the same time it is built, you must change the quarkus.container-image.build property with the quarkus.container-image.push.

```
[user@host ~]$ mvn package -Pnative -Dquarkus.container-image.push=true
```

This command uses docker.io as the default container registry, but you can change the target registry with the property quarkus.container-image.registry.

This command also defaults to running Docker. You can instead use Podman by specifying the quarkus.native.container-runtime property, like the following example:

```
[user@host ~]$ mvn package -Pnative -Dquarkus.container-image.push=true \
-Dquarkus.native.container-runtime=podman
```

## Deploying Native Applications to OpenShift

In previous chapters, you learned how to deploy Quarkus applications to Red Hat OpenShift Container Platform (RHOCP), using the `quarkus.kubernetes.deploy` property. You can use the same approach with binary container image deployments by adding the `-Pnative` property.

The following is an example command to deploy a microservice to RHOCP in native mode.

```
[user@host ~]$ mvn package -Pnative -Dquarkus.kubernetes.deploy=true
```

A `binary container image deployment` is one that takes the native image generated locally and adds it to a base image to run it.

The alternative option to binary container image deployments is a `source container image deployment`, which is a deployment where RHOCP uses the source code of the application to build the native version in the cluster. To achieve this, you must reference the location of the source code and a builder image that has native building capabilities.

The following is an example of creating a new application using the `oc` command.

```
[user@host ~]$ oc new-app https://url-to-source-code
-i quay.io/quarkus/ubi-quarkus-native-s2i:20.2.0-jav11
```



### References

For more information on deploying native executables to OpenShift, refer to the Product Documentation for Red Hat Build of Quarkus at [https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.11/html-single/compiling\\_your\\_quarkus\\_applications\\_to\\_native\\_executables/index#proc-creating-container-openshift-build\\_quarkus-building-native-executable](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11/html-single/compiling_your_quarkus_applications_to_native_executables/index#proc-creating-container-openshift-build_quarkus-building-native-executable)

#### Source2Image

<https://github.com/openshift/source-to-image>

#### Jib

<https://github.com/GoogleContainerTools/jib>

## ► Guided Exercise

# Containerizing Quarkus Native Applications

In this exercise, you will learn how to deploy a native Quarkus application into Red Hat OpenShift Container Platform (RHOCP).

## Outcomes

You should be able to deploy a native Quarkus application into an OpenShift Container Platform.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the Expenses application is available in your workstation.

```
[student@workstation ~]$ lab native-container start
```

## Instructions

- 1. To compare the native Quarkus application with the same application running in JVM mode, you are going to deploy both in RHOCP. The Quarkus Expenses application you must use is located in the 1.11 branch of the <https://github.com/RedHatTraining/D0378-apps/> repository.
  - 1.1. To start the build process for the application in OpenShift, make sure you are logged into the cluster and using the appropriate project.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ oc login ${RHT_OCP4_MASTER_API} \
-u ${RHT_OCP4_DEV_USER} -p ${RHT_OCP4_DEV_PASSWORD}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project native-container-${RHT_OCP4_DEV_USER}
Now using project "native-container-youruser" ...
...output omitted...
```

- 1.2. Use the `oc` command to start the JVM build of the application.

```
[student@workstation ~]$ BUILD_REPO=\
https://github.com/RedHatTraining/D0378-apps#1.11
[student@workstation ~]$ oc new-app --name=jvm-expenses \
--context-dir=expense-restful-service \
"registry.access.redhat.com/ubi8/openjdk-11-${BUILD_REPO}"
...output omitted...
--> Creating resources ...
```

```
imagestream.image.openshift.io "openjdk-11" created
imagestream.image.openshift.io "jvm-expenses" created
buildconfig.build.openshift.io "jvm-expenses" created
deployment.apps "jvm-expenses" created
service "jvm-expenses" created
--> Success
Build scheduled, use 'oc logs -f bc/jvm-expenses' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/jvm-expenses'
Run 'oc status' to view your app.
```

13. Use the `oc logs` command to monitor the build of the application in the cluster.

```
[student@workstation ~]$ oc logs -f bc/jvm-expenses
Cloning "https://github.com/RedHatTraining/D0378-apps#1.11" ...
...output omitted...
Push successful
```

14. Review the startup time of the application in the logs. Press Tab twice after `jvm-expenses-` for the shell to autocomplete the name of the pod.

```
[student@workstation ~]$ oc logs jvm-expenses-6dfc9f4f87-6v6sd
Starting the Java application using /opt/jboss/container/java/run/run-java.sh ...
INFO exec java -Dquarkus.http.host=0.0.0.0 -Xms256m -Xmx1024m -XX:
+UseParallelOldGC -XX:MinHeapFreeRatio=10 -XX:MaxHeapFreeRatio=20 -
XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90 -XX:ParallelGCThreads=2 -
Djava.util.concurrent.ForkJoinPool.common.parallelism=2 -XX:CICompilerCount=2 -
XX:+ExitOnOutOfMemoryError -cp "." -jar /deployments/expense-restful-service-1.0-
runner.jar
...output omitted...
2021-09-22 17:27:21,990 INFO [io.quarkus] (main) expense-restful-service 1.0 on
JVM (powered by Quarkus 1.11.7.Final-redhat-00009) started in 1.186s. Listening
on: http://0.0.0.0:8080
2021-09-22 17:27:22,009 INFO [io.quarkus] (main) Profile prod activated.
2021-09-22 17:27:22,009 INFO [io.quarkus] (main) Installed features: [cdi,
resteasy, resteasy-jsonb, smallrye-openapi, swagger-ui]
```

Take note of the startup time for the application running in JVM mode in the RHOC.

► 2. Test that the application is running correctly in RHOC.

- 2.1. Expose the route of the service so that you can access the application.

```
[student@workstation ~]$ oc expose service jvm-expenses
route.route.openshift.io/jvm-expenses exposed
```

- 2.2. Use the `curl` command to call the `/expenses` endpoint.

```
[student@workstation ~]$ JVM_HOST=jvm-expenses-native-container-\${
RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN}
[student@workstation ~]$ curl -s ${JVM_HOST}/expenses | jq
[
```

```
{
 "amount": 25,
 "creationDate": "2020-10-22T12:05:28.697519",
 "name": "Civilization VI",
 "paymentMethod": "DEBIT_CARD",
 "uuid": "00299173-280d-4ca2-8866-b1125ba8e4f3"
},
{
 "amount": 150.5,
 "creationDate": "2020-10-22T12:05:28.697212",
 "name": "Groceries",
 "paymentMethod": "CASH",
 "uuid": "18fb6807-df8b-444c-9a47-cc37580f9fd2"
}
]
```

- ▶ 3. Build the native version of the Quarkus Expenses application in RHOCP to compare startup time.

- 3.1. Use the oc command to start the remote native build.

```
[student@workstation ~]$ oc new-app --name=native-expenses \
--context-dir=expense-restful-service \
"quay.io/quarkus/ubi-quarkus-native-s2i:20.3.3-java11-${BUILD_REPO}"
...output omitted...
--> Creating resources ...
buildconfig.build.openshift.io "native-expenses" created
deployment.apps "native-expenses" created
service "native-expenses" created
--> Success
Build scheduled, use 'oc logs -f buildconfig/native-expenses' to track its
progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
 'oc expose service/native-expenses'
Run 'oc status' to view your app.
```

- 3.2. GraalVM uses a lot of memory to build the native executable. Because of default memory settings, this compilation will fail. To solve this problem, cancel the current build, update the memory settings of the build pod, and relaunch the build process.

```
[student@workstation ~]$ oc cancel-build bc/native-expenses
build.build.openshift.io/native-expenses-1 marked for cancellation, waiting to be
cancelled
build.build.openshift.io/native-expenses-1 cancelled
[student@workstation ~]$ oc patch bc/native-expenses \
-p '{"spec":{"resources":{"limits":{"cpu":"4", "memory":"3.5Gi"}}}}'
buildconfig.build.openshift.io/native-expenses patched
[student@workstation ~]$ oc start-build native-expenses
build.build.openshift.io/native-expenses-2 started
```

- 3.3. Use the oc command to monitor the build of the native application in the cluster.

```
[student@workstation ~]$ oc logs -f bc/native-expenses
Cloning "https://github.com/RedHatTraining/D0378-apps#1.11" ...
...output omitted...
Push successful
```

- 3.4. Review the startup time in the logs and its improvement. Press Tab twice after jvm-expenses- for the shell to autocomplete the name of the pod.

```
[student@workstation ~]$ oc logs native-expenses-6fb6ffd998-vzncv
...output omitted...
2021-09-22 17:51:12,427 INFO [io.quarkus] (main) expense-restful-service
1.0 native (powered by Quarkus 1.11.7.Final-redhat-00009) started in 0.022s.
Listening on: http://0.0.0.0:8080
2021-09-22 17:51:12,427 INFO [io.quarkus] (main) Profile prod activated.
2021-09-22 17:51:12,427 INFO [io.quarkus] (main) Installed features: [cdi,
resteasy, resteasy-jsonb, smallrye-openapi, swagger-ui]
```

#### ► 4. Test that the native application is running correctly in RHOCP.

- 4.1. Expose the route of the service so you can access the application.

```
[student@workstation ~]$ oc expose service native-expenses
route.route.openshift.io/native-expenses exposed
```

- 4.2. Use the curl command to call the /expenses endpoint.

```
[student@workstation ~]$ NATIVE_HOST=native-expenses-native-container-\$${RHT_OCP4_DEV_USER}.\$${RHT_OCP4_WILDCARD_DOMAIN}
[student@workstation ~]$ curl -s ${NATIVE_HOST}/expenses | jq
[
 {
 "amount": 25,
 "creationDate": "2020-10-22T12:05:28.697519",
 "name": "Civilization VI",
 "paymentMethod": "DEBIT_CARD",
 "uuid": "00299173-280d-4ca2-8866-b1125ba8e4f3"
 },
 {
 "amount": 150.5,
 "creationDate": "2020-10-22T12:05:28.697212",
 "name": "Groceries",
 "paymentMethod": "CASH",
 "uuid": "18fb6807-df8b-444c-9a47-cc37580f9fd2"
 }
]
```

## Finish

On the workstation machine, use the lab command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab native-container finish
```

This concludes the guided exercise.

# Summary

---

In this chapter, you have learned that:

- By Using **GraalVM** and **Mandrel**, Quarkus can generate a native executable application package. Native executable files are more efficient in terms of memory and startup performance.
- **GraalVM** native compilation follows the Closed World principle. Developers must register code not statically available and reachable at compile time.
- The **kubernetes** and **openshift** extensions create the required resources and deploy Quarkus applications in the corresponding platform. Alternatively, the **S2I** extension packages and deploys the application directly from source code inside the container platform.

## Chapter 7

# Testing Microservices

### Goal

Implement unit and integration tests for microservices.

### Objectives

- Implement a quarkus microservice test case.
- Implement a microservice test using mock frameworks.

### Sections

- Testing Quarkus Microservices (and Guided Exercise)
- Testing Microservices with Mock Frameworks (and Guided Exercise)

### Lab

Testing Microservices

# Testing Quarkus Microservices

---

## Objectives

After completing this section, you should be able to implement a quarkus microservice test case.

## Implementing a Basic Test with Quarkus

Testing applications is an important part of the application development lifecycle, both locally and remotely in continuous integration servers. Historically, JUnit has been the *de facto* standard for running unit tests in Java. The `quarkus-junit5` extension integrates JUnit 5 into the Quarkus framework, enabling seamless integration and execution of tests.

To create a test in Quarkus, you must.

1. Add the `quarkus-junit5` extension to your project.
2. Override the `Maven Surefire Plugin` version, log manager and Maven home folder.
3. Create a test class and add the `io.quarkus.test.junit.QuarkusTest` annotation to the class. Following the Maven folder structure convention, test classes should be placed in the `/src/test/java` folder.
4. Annotate each test method with `org.junit.jupiter.api.Test` to indicate which methods the framework must execute.

Steps 1 and 2 are automatically done by Quarkus code initiators.

When tests are ready to run, use the standard Maven tooling to execute the tests using the `Surefire` plugin: `mvn test` or `mvn verify`.

Most IDEs include handy tools to execute the unit test and review the results.

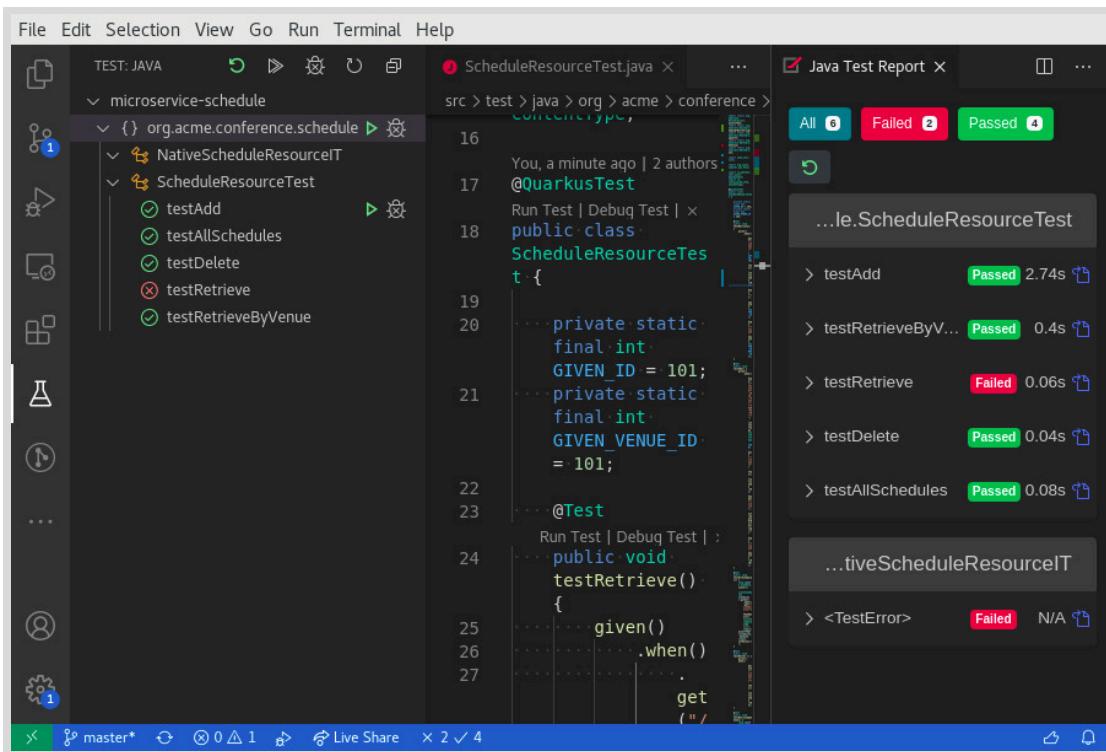


Figure 7.1: VSCodium JUnit Test extension

## Enabling the Test Profile and Test Configuration

Tests usually run in a non-production environment, so they need their own configuration. Also, the components to be tested need their own configuration in the test environment.

In Quarkus, these dedicated configurations are packaged in a configuration profile named `test`. This profile is enabled automatically when Quarkus runs the application in test mode.

As shown in *Injecting Configuration Data*, to define configuration entries for a specific profile prepend the configuration key with the `%` sign and the profile name, `test` in this case.

```
quarkus.datasource.db-kind=postgresql ①
%test.quarkus.datasource.db-kind=h2 ②
```

- ① When the `prod`, `dev` or any other profile than `test` is active, the application will use a PostgreSQL database.
- ② When the `test` profile is active, the application will use a H2 database.

Note that many configuration entries have default values, which depend on the active profile. For example, Hibernate's initialization SQL script has no default value for the `prod` profile, but uses `import-dev.sql` and `import-test.sql` for `dev` and `test` profiles respectively. The following snippet describes example values for the `quarkus.hibernate-orm.sql-load-script` configuration entry of each profile.

```
%dev.quarkus.hibernate-orm.sql-load-script = import-dev.sql
%test.quarkus.hibernate-orm.sql-load-script = import-test.sql
%prod.quarkus.hibernate-orm.sql-load-script = no-file
```

Quarkus includes a few configuration entries that control the behavior of the tests themselves. Those configuration entries should not be prefixed with %test. The following list shows some of the most important test configuration entries.

### Quarkus Test Configuration Entries

Entry	Description	Default
quarkus.http.test-port	The HTTP port the application will use when running in test mode.	8081
quarkus.http.test-ssl-port	The HTTPS port the application will use when running in test mode.	8444
quarkus.test.native-image-wait-time	Wait time for the native image to build during testing	5 min
quarkus.test.native-image-profile	The profile to use when testing the native image	prod
quarkus.test.profile	The profile to use when testing using @QuarkusTest	test
quarkus.swagger-ui.always-include	Enable Swagger UI when the prod profile is not enabled.	false
quarkus.http.test-timeout	When using REST-assured, timeout for obtaining a connection and a response.	30 sec

## Adding Test Resources

Tests usually need specific resources to perform executions or validations. Because Quarkus test classes are managed beans, Arc can inject any bean into the test class, facilitating the test class to interact with the application.

```
package org.acme.test;
import javax.inject.Inject;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
public class MyTestClass {

 @Inject MyService myService;

 @Test
 public void testMyService() {
 Assertions.assertTrue(myService.isWorking());
 }
}
```

## Injecting Test Endpoints

Some resources are not directly available as beans to inject. One example is the URL to a resource in the application or to an endpoint. In those cases, Quarkus provides the `TestHTTPResource` annotation to obtain resource URLs, and the `TestHTTPEndpoint` to inject the endpoint associated to a class.

```
@QuarkusTest
public class MyTestClass {

 @TestHTTPEndpoint(MyService.class) @TestHTTPResource URL endpoint;

 @Test
 public void testMyServiceEndPoint() {
 try (InputStream in = endpoint.openStream()) {
 String contents = new BufferedReader(new
InputStreamReader(in)).lines().collect(Collectors.joining("\n"));
 Assertions.assertTrue(contents.equals("hello"));
 }
 }

 @TestHTTPResource("index.html") URL resource;

 @Test
 public void testResource() {
 try (InputStream in = resource.openStream()) {
 String contents = new BufferedReader(new
InputStreamReader(in)).lines().collect(Collectors.joining("\n"));
 Assertions.assertTrue(contents.equals("Static resource"));
 }
 }
}
```

## Leveraging REST-assured

Although previous examples are functional, reading raw content from complex services is tedious and error-prone. REST-assured is a library, which facilitates testing of REST microservices, and it is fully integrated into Quarkus.

REST-assured uses the Given-When-Then notation to declare the test, and Hamcrest matchers to validate the results. The following example is equivalent to the previous one, but using REST-assured features.

```
import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class MyTestClass {

 @TestHTTPEndpoint(MyService.class)
 @TestHTTPResource
 URL endpoint;

 @Test
}
```

```

public void testMyServiceEndPoint() {
 given().when().get(endpoint).then().body(is("hello"));
}

@TestHTTPResource("index.html")
URL resource;

@Test
public void testResource() {
 given().when().get(resource).then().body(is("Static resource"));
}
}

```

Unlike the previous examples, test suites usually validate the endpoints for a single resource class. You can use the `TestHTTPEndpoint` annotation on the test class to define the default resource class to test.

```

@QuarkusTest
@TestHTTPEndpoint(MyService.class)
public class MyTestClass {

 @Test
 public void testMyServiceEndPoint() {
 given()
 .when().get() // no endpoint here!
 .then().body(is("hello"));
 }
}

```

## Providing Test Transient Dependencies

Integration testing usually requires the test suite to provide the tested application with its required dependencies. For example, if you are testing an application that relies on the existence of a remote database, then the test suite is responsible for making this database available during the tests.

Developing the required code to manage the lifecycle of those dependencies is complex, and a repetitive task for common dependencies such as databases or data busses.

Quarkus gives a standard solution for providing the most common dependencies via the `io.quarkus.test.common.QuarkusTestResource` annotation. When using this annotation on a test suite, and providing a resource class, Quarkus will handle the resource lifecycle for you.

The following example creates a Derby in-memory database available for the application during the execution of the test suite.

```

...output omitted...
@QuarkusTest
@QuarkusTestResource(DerbyDatabaseTestResource.class)
public class MyServiceTest {
...output omitted...

```

Other available resource classes are:

**H2DatabaseTestResource**

Starts an in-memory H2 database.

**LdapServerTestResource**

Starts a local LDAP server.

**KubernetesMockServerTestResource**

Starts a Kubernetes API mock server.

The list of available resources is constantly growing based on community needs. If you need another resource not currently available, then create your own resources by implementing the `io.quarkus.test.common.QuarkusTestResourceLifecycleManager` interface.

## Testing Quarkus Native Applications

Testing a native application generated with Quarkus is different from testing a JVM application. The main difference is that the application is no longer controlled by Java, rather it functions as a completely independent application. This means that Quarkus can only test the application API and exposed features, not application internals.

Before testing the application in native mode you must be able to create a native executable for the application. As shown in *Building Native Applications with Quarkus*, you must enable the native profile with the `-Pnative` parameter while packaging the application.

```
[user@host ~]$ mvn package -Pnative
```

Then, you must create a dedicated test class for native tests. Add the `io.quarkus.test.junit.NativeImageTest` annotation to the class to indicate executing the tests while in the native profile. In most cases, the native tests are exactly the same JVM tests, so the native test class can just extend the JVM test class to include the same test suite.

```
import io.quarkus.test.junit.NativeImageTest;

@NativeImageTest
public class MyServiceTestNativeIT extends MyServiceTest {
 // Execute the same tests but in native mode.
}
```

To execute the native tests, use the `mvn test -Pnative` command. The `test` option indicates Maven to execute the test without trying to compile the application again, as we already packaged the application in the previous step.



## References

For more information on testing with Quarkus, refer to the Product Documentation for Red Hat Build of Quarkus at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.11/html-single/testing\\_your\\_quarkus\\_applications/index](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11/html-single/testing_your_quarkus_applications/index)

### JUnit 5 official page

<https://junit.org/junit5/>

### REST-assured site

<https://github.com/rest-assured/rest-assured>

## ► Guided Exercise

# Testing Quarkus Microservices

In this exercise, you will create and execute unit and integration tests for the Expenses application.

### Outcomes

You should be able to create unit and integration tests for your Quarkus application.

### Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the code for the Expenses application is available on your workstation.

```
[student@workstation ~]$ lab test-junit start
```

### Instructions

In this exercise you will create tests to verify the correctness of the application in JVM and Native mode.

- 1. Open with VSCode the Expenses application located at the `~/D0378/labs/test-junit` folder.

```
[student@workstation ~]$ cd ~/D0378/labs/test-junit
[student@workstation test-junit]$ codium .
```

- 2. Open and examine the files `ExpensesCreationTest.java` and `EndPointTest.java` located in the `src/test/java/org/acme/rest/json` folder. A test skeleton is available for you with two test methods.

- The `EndPointTest.testEndPointPort` method checks the application endpoint is using port number 8888. Note the dependence on the `expensesURI` attribute. This attribute has no value, so the test is currently failing.
- The `ExpensesCreationTest.testCreateExpense` method is empty. You will fill this method to test that a newly created expense will appear when listing all expenses. Because the test assumes the database is clean, the new expense created must be the only expense listed.

- 3. When you try to execute the tests, then nothing happens. You must annotate the classes as a test class.

Add the `@QuarkusTest` annotation to both classes so that the test can run using JUnit.

```
import io.quarkus.test.junit.QuarkusTest;
...output omitted...
```

```
@QuarkusTest
public class ExpensesCreationTest {
...output omitted...
```

```
import io.quarkus.test.junit.QuarkusTest;
...output omitted...
```

```
@QuarkusTest
public class EndPointTest {
...output omitted...
```

- ▶ 4. Instruct Quarkus to inject the value of the `EndPointTest.expensesURI` attribute. This attribute must contain the URI where the application runs while in test mode.

Add the `io.quarkus.test.common.http.TestHTTPResource` annotation to tell Quarkus to inject the needed URI into the test. Add the `io.quarkus.test.common.http.TestHTTPEndpoint` annotation to indicate Quarkus to use `ExpenseResource.class` endpoints.

```
import io.quarkus.test.common.http.TestHTTPEndpoint;
import io.quarkus.test.common.http.TestHTTPResource;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
public class EndPointTest {

 @TestHTTPResource
 @TestHTTPEndpoint(ExpenseResource.class)
 URI expensesURI;
 ...output omitted...
```

- ▶ 5. Update the `application.properties` file to avoid conflicts with running applications. Configure a test port of 8888 using the `quarkus.http.test-port` entry.

```
Configuration file
key = value
quarkus.http.test-port=8888
...output omitted...
```

- 6. Fill the `testCreateExpense` method to check that, after using the POST endpoint to add an expense, the GET endpoint returns the added expense.

There are multiple ways of implementing this test. The following is a sample implementation, which uses the RESTAssured framework. Make sure you add the needed dependency to your `pom.xml` file.

```
<dependency>
 <groupId>io.rest-assured</groupId>
 <artifactId>rest-assured</artifactId>
 <scope>test</scope>
</dependency>
```

Then use RESTAssured to implement the test as required.

```
import static io.restassured.RestAssured.given;
import static io.restassured.RestAssured.when;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.is;
import org.acme.rest.json.Expense.PaymentMethod;
import io.restassured.http.ContentType;

...output omitted...

@Test
public void testCreateExpense() {
 given()
 .body(Expense.of("Test Expense", PaymentMethod.CASH, "1234"))
 .contentType(ContentType.JSON)
 .post("/expenses");
 when()
 .get("/expenses")
 .then()
 .statusCode(200)
 .assertThat().body("size()", is(1))
 .body(
 containsString("\"name\":\"Test Expense\""),
 containsString("\"paymentMethod\":\"" + PaymentMethod.CASH + "\""),
 containsString("\"amount\":1234.0")
);
}
```

- 7. Use the `mvn test` command to execute the tests and verify they are failing due to the microservice being unable to connect the database.

```
[student@workstation test-junit]$ mvn test
[INFO] Scanning for projects...
...output omitted...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] org.acme.rest.json.ExpensesCreationTest
...output omitted...
[ERROR] Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 10.541 s
<<< FAILURE! - in org.acme.rest.json.ExpensesCreationTest
```

```
[ERROR] testCreateExpense Time elapsed: 0.008 s <<< ERROR!
java.lang.RuntimeException: java.lang.RuntimeException: Failed to start quarkus
Caused by: java.lang.RuntimeException: Failed to start quarkus
Caused by: javax.persistence.PersistenceException: [PersistenceUnit: <default>]
 Unable to build Hibernate SessionFactory
Caused by: org.hibernate.exception.GenericJDBCException: Unable to open JDBC
 Connection for DDL execution
Caused by: org.h2.jdbc.JdbcSQLException: Connection is broken:
 "java.net.ConnectException: Connection refused (Connection refused):
 localhost" [90067-197]
Caused by: java.net.ConnectException: Connection refused (Connection refused)
...output omitted...
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
...output omitted...
```

The root cause is that the expenses microservice is trying to connect to the H2 In-memory Database, which the tests are not configured to run. To fix this, you must perform two changes.

The first change is instructing Quarkus to spin a new test database each time the test suite runs. The `io.quarkus.test.common.QuarkusTestResource` instructs the test framework to create a resource with a lifecycle tied to the test. Add the `io.quarkus.test.h2.H2DatabaseTestResource.class` parameter to the annotation to indicate Quarkus to spin up a H2 database.

```
import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.h2.H2DatabaseTestResource;
...output omitted...

@QuarkusTest
@QuarkusTestResource(H2DatabaseTestResource.class)
public class ExpensesCreationTest {
```

The class `H2DatabaseTestResource` does not belong to Quarkus core, but to the `io.quarkus:quarkus-test-h2` extension. Add this extension directly to the `pom.xml` file as a test dependency.

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-test-h2</artifactId>
</dependency>
```



### Note

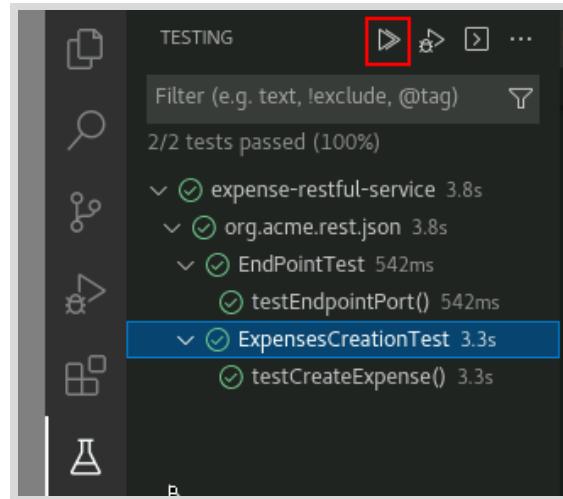
You can also use the following command to add the extension:

```
mvn quarkus:add-extension \
-Dextension=io.quarkus:quarkus-test-h2
```

- 8. Execute all the tests in JVM mode. You can use the `mvn test` command in a terminal. Both tests must pass successfully.

**Note**

You can also use the Test view from the left menu bar in VSCode. Use the double arrow on top of the tests tree to run all tests, or use single arrows beside each test to run tests individually



- 9. Prepare a test suite to execute the expenses creating tests in native mode.

Create a file named `/src/test/java/org/acme/rest/json/NativeExpensesCreationIT.java`. Make the `NativeExpensesCreationIT` class extend the `ExpensesCreationTest` class, so the same tests execute in native mode. Add the `@NativeImageTest` annotation to the class to mark it as a native test suite. The resulting class should look as follows.

```
package org.acme.rest.json;

import io.quarkus.test.junit.NativeImageTest;

@NoArgsConstructor
public class NativeExpensesCreationIT extends ExpensesCreationTest { }
```

During the execution of this test, Quarkus creates the native executable file. Due to high memory requirements, we recommend closing all unneeded applications, including VSCode, before running this test. Use the following command to create it.

```
[student@workstation test-junit]$ mvn verify -Pnative
[INFO] Scanning for projects...
...output omitted...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running org.acme.rest.json.NativeExpensesCreationIT
```

```
[INFO] H2 database started in TCP server mode; server status: TCP server running
at ... (only local connections)
...output omitted...
[INFO] H2 database was shut down; server status: Not started
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab test-junit finish
```

This concludes the guided exercise.

# Testing Microservices with Mock Frameworks

## Objectives

After completing this section, you should be able to implement a microservice test using mock frameworks.

## Testing Using Mocks

One of the core concepts of unit testing is to test individual units or components. The purpose of this is to be able to validate each unit independent of other units as much as possible. Mocks are a tool that aids unit testing by facilitating the isolation of a component from its dependencies.

Mocks are fake objects of classes, of which the component to be tested is dependent. They should contain only basic logic that exists with the sole purpose of enabling the tests for which they are created.

## Testing Using the Mock Annotation

Common use cases for mocking include tests using classes, which interact with external dependencies such as databases and web services. The MicroProfile CDI specification provides mechanisms that are useful for using mock classes. Among the useful annotations in the specification, some of the most commonly used are `@Alternative`, `@Priority`, and `@Dependent`.

To simplify straight forward mocks, Quarkus uses the MicroProfile specification and puts together the annotations mentioned previously, creating the `@io.quarkus.test.Mock` annotation. This annotation creates a curated bean, which during test time allows Quarkus DI to replace other beans with the same interface. Using this annotation with a class that extends the class intended to be mocked, allows you to use Quarkus DI to handle injecting the bean as part of the dependencies, overriding the original implementation with the mocked one. This helps to isolate the unit tests to the unit that you want to test.



### Note

To be more precise, the `@Mock` annotation is a shortcut for adding the `@Alternative`, `@Dependent` and `@Priority(1)` annotations. This gives the bean the highest priority, replacing any other beans of the same kind.

To mock objects via this method, you must create a class in your test subdirectory, found in `src/test/java`. The class must be annotated with `@Mock`, and the interface shared with the class that you want to mock. Then, create the required methods in the mock class, which can have a custom implementation that serves the intentions of the tests. The mock class can be annotated with intended scopes, which overrides the `@Dependent` annotation included with `@Mock`.

## Code Example

In the preceding example, the class `DataStore` is mocked and overrides its method with signature `public Data retrieve(UUID)`.

```
...output omitted...
@Mock
@ApplicationScoped
public class DataStoreMock extends DataStore {
 @Override
 public Data retrieve(UUID uuid) {
 return new Data(uuid, "name", 123);
 }
}
```

This class can now be injected into your tests.

```
...output omitted...
@QuarkusTest
public class SomeServiceTest {
 @Inject❶
 DataStore dataStore;

 public void testExample() {
 UUID uuid = ...output omitted...
 Data data = dataStore.retrieve(uuid);❷
 ...output omitted...
 }
}
```

- ❶ Quarkus DI will find the mock and use it for the injection.
- ❷ The method called here will be the retrieve method from the mock, which has minimal logic and predictable behavior for the test.



### Warning

Due to native executables running outside of a JVM process, injecting into tests is not supported when testing native executables.

## Using Mockito

The `@Mock` annotation is simple to use, but if many tests require different mocks for the same bean, then the approach can become difficult to maintain. A solution to this is to create the mocks during runtime, using **Mockito**.

**Mockito** is a mocking framework used for developing unit tests. The main goals of Mockito include enabling tests that are readable, have clean verification errors when they fail and provide a minimal API for building stubs and verification's. To use it in Quarkus, you add the `quarkus-junit5-mockito` extension in your `pom.xml` file.

The entry class is the `org.mockito.Mockito` class, which contains several static methods that simplify mocking classes.

## Learning Mockito's Fundamental API

For creating a mock, you use the `Mockito.mock()` method, passing the corresponding class as a parameter. This method dynamically creates a class on the fly, and creates an instance of the class, which is aware of the methods of the class being mocked.

Mockito uses a `when` and `then` structure for creating method stubs. First, the `Mockito.when()` method is used, which receives as its parameter an interaction with the mock object. This interaction can be calling a method of the instance of the mock object, including the parameters which will be passed to the method call. After the `when()` interaction has been defined, Mockito enables multiple consequences to the interaction, using the `thenReturn()`, `thenThrow()`, `thenAnswer()` and `thenCallRealMethod()` methods.

Here is an equivalent of the previous mocking example, but it is now using Mockito for mocking the `DataStore` class.

```
...output omitted...
@QuarkusTest
public class SomeServiceTest {
 public void testExample() {
 UUID uuid = ...output omitted...
 DataStore dataStore = Mockito.mock(DataStore.class); ①
 Mockito.when(dataStore.retrieve(uuid))
 .thenReturn(new Data(uuid, "name", 123)); ②
 Mockito.when(dataStore.retrieve(null))
 .thenThrow(new NullPointerException()); ③
 Data data = dataStore.retrieve(uuid); ④
 Data data = dataStore.retrieve(null); ⑤
 }
 ...output omitted...
```

- ① A single line creates a mock class and instantiates an object.
- ② The interaction with the `retrieve()` method is used to create a stub, which works only with the parameter specified. The parameter of `thenReturn()` must be of the same type as the type returned by the original `retrieve()` method.
- ③ Instruct the mock to throw a `NullPointerException` if the `uuid` parameter is `null`.
- ④ `data` will be a `Data` object as specified previously.
- ⑤ This line will actually throw an uncaught `NullPointerException`.

For any method call for which there is not a stub definition, Mockito will return a default value. This could be `false` for boolean type, `null` for general objects, an empty `String`, `0` for integers, and empty collections.

There are also cases when you might prefer to specify the stub behavior for broader parameter values. For this, Mockito provides the concept of `argument matchers`. Argument matchers implement the `ArgumentMatcher` interface and are meant to be reusable objects representing arguments, which can be passed to a method call of the mock object. Mockito provides a vast number of argument matchers for the most common classes, and for those that are not covered, it gives the developer the tool of creating custom argument matchers.

Among the long list of argument matchers provided by Mockito are `anyString`, `anyList`, `anyIterable`, `anyInt`, `anyCollectionOf`, and `any`. The `any` argument matcher is special, because it will match any parameter, regardless of value.

```
...output omitted...
public void testExample() {
 UUID uuid = ...output omitted...
 DataStore dataStore = Mockito.mock(DataStore.class);
 Mockito.when(dataStore.retrieve(Mockito.any()))
 .thenReturn(new Data(uuid, "name", 123)); ①
 Data data = dataStore.retrieve(uuid);
}
...output omitted...
```

- ① Every time there is a call to the `retrieve()` method, this stub will be called, regardless of the value of the parameter.



### Note

The mocking examples included in here are written with the purpose of showing how you can create mock objects. In real world tests, the mock objects are dependencies of the class that is intended to be tested.

You can also use the `@io.quarkus.test.junit.Mockito.InjectMock` annotation to use Quarkus DI to inject mocks created with Mockito. The `@InjectMock` annotation enables creating mocks which use Quarkus DI to inject them as dependencies, with control over configurations for the entire test class and for each test case.

```
...output omitted...
@QuarkusTest
public class SomeServiceTest {
 @InjectMock ①
 DataStore dataStore;

 @BeforeEach
 public void setup() {
 Mockito.when(dataStore.retrieve(null)).thenThrow(new
NullPointerException()); ②
 }

 public void testExample() {
 UUID uuid = ...output omitted...
 Mockito.when(dataStore.retrieve(Mockito.any()))
 .thenReturn(new Data(uuid, "name", 123)); ③
 Data data = dataStore.retrieve(uuid);
 ...output omitted...
 }
}
```

- ① The annotation indicates Quarkus DI to create a mock object using Mockito.  
② This stub will be available for all test cases inside of the `SomeServiceTest` class.

- ③ The `dataStore` object behaves the same as it did in the last example. This configuration is only available within the `testExample()` test.

## Verifying Behavior

The API of Mockito enables the developer to verify that there were invocations to the mock object. This is not used to verify direct interactions, where the testing code itself is calling the mock methods, it is used to test indirect interactions, where a second method performs the interaction.

The `Mockito.verify()` method is used at the end of a test. The method receives a mocked object and returns an object that shares the interface of the mock, the difference being that the logic it will execute is dependant on whether a method was called, and if specified, the number of times it was called. This allows you to verify that an interaction occurred and that your test is valid. If the verification fails, then it will throw an exception belonging to the `org.mockito.exceptionsverification` package.

Here are some examples of different verification's that can be performed on mock objects. Note that in this example, the static methods from the `Mockito` class have been imported.

```
DataStore dataStore = mock(DataStore.class);
when(dataStore.get(0)).thenReturn(new Data("name", 123));
System.out.println(dataStore.get(0).name);
verify(dataStore).get(0);
verify(dataStore, times(1)).get(0);
System.out.println(dataStore.get(0).age);
verify(dataStore, times(2)).get(0);
verify(dataStore, atMost(2)).get(0);
verify(dataStore, never()).get(1);
```



### Note

This is a common practice in tests using Mockito, because it arguably makes the code more readable.

All of the previous verification's would run without errors.

## Using PanacheMock for Mocking ActiveRecords with Panache

Mockito has recently added support to mocking static methods. However, using `io.quarkus.panache.mock.PanacheMock` remains the simplest way of mocking Panache classes and its static methods.

You must add the `quarkus-panache-mock` extension to your `pom.xml` file to enable using `PanacheMock`. After you have added the extension, using the class is straight forward. Instead of using `Mockito.mock()` as described previously, use `PanacheMock.mock()`. You can then create stubs for the mocked class using `Mockito` like you would normally do when creating with `Mockito.mock()`. Calls by other classes to the mocked entity's methods, used inside of the test will now use the stub created.

```
...output omitted...
public void testExample() {
 PanacheMock.mock(Data.class);
 Mockito.when(Data.find("name", "Joseph"))
 .thenReturn(new Data(UUID.randomUUID(), "Joseph", 123));
 Data data = Data.find("name", "Joseph");
 Assertions.assertEquals("Joseph", data.name);
}
...output omitted...
```



## References

For more information on Quarkus test mocks, refer to the Product Documentation for Red Hat Build of Quarkus at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.11/html-single/testing\\_your\\_quarkus\\_applications/index#proc-test-mocking-objects](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11/html-single/testing_your_quarkus_applications/index#proc-test-mocking-objects)

## Mockito Home Page

<https://site.mockito.org/>

## ► Guided Exercise

# Testing Microservices with Mock Frameworks

In this exercise, you will write unit tests for a Quarkus application's resource class by mocking its service class using both `io.quarkus.test.Mock` and `Mockito`. You will also write unit tests for the service class by mocking its persistence layer using `PanacheMock`.

## Outcomes

You should be able to write unit tests with mocks for a Quarkus microservice using `io.quarkus.test.Mock`, `Mockito` and `PanacheMock`.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that you have the files required for this activity in your workstation.

```
[student@workstation ~]$ lab test-mock start
```

## Instructions

- 1. Review the expense application for this activity. The source code is in the `~/D0378/labs/test-mock/expense-service/` folder. Use `VSCodium` to open the folder as a new Quarkus application.

```
[student@workstation ~]$ codium ~/D0378/labs/test-mock/expense-service/
```

Relevant preexisting files for this activity are in the `src/main/java/org/acme/rest/json/` directory. Inspect the `ExpenseResource`, `ExpenseService`, and `Expense` Java classes. You will write unit tests for `ExpenseResource` and `ExpenseService`.

Note that `ExpenseResource` depends on `ExpenseService`, and `ExpenseService` has `Expense` as a dependency.

- 2. Create a mock of `ExpenseService` using `io.quarkus.test.Mock` and use it for the `testUpdateExistingExpense()` method. Open the `src/test/java/org/acme/rest/json/ExpenseResourceTest.java` file and modify it to use a mock of `ExpenseService`. You will test the `ExpenseResource.update(Expense)` endpoint method. To isolate the `update()` method's logic from the `ExpenseService` implementation, you will create stubs for the `update(Expense)` and `exists(UUID)` methods.
- 2.1. Create a new java class named `ExpenseServiceMock`, which extends `ExpenseService` and is annotated with both `@io.quarkus.test.Mock` and `@ApplicationScoped`. Make sure the class is created in a new file named `src/test/java/org/acme/rest/json/ExpenseServiceMock.java`.

```
package org.acme.rest.json;

import javax.enterprise.context.ApplicationScoped;

import io.quarkus.test.Mock;

@Mock
@ApplicationScoped
public class ExpenseServiceMock extends ExpenseService {

}
```

- 2.2. Create an empty method with the signature `update(Expense)`, which overrides the `ExpenseService` method.

```
...output omitted...
public class ExpenseServiceMock extends ExpenseService {
 @Override
 public void update(Expense expense) {}
}
```

- 2.3. Create an `exists(UUID)` method that overrides the one found in `ExpenseService`. Add logic so that if it receives a `uuid` of value `"efffbfe08-0d71-46e7-87af-1132b1b94d87"`, then it returns `true`, otherwise it returns `false`.

Your code should look like this.

```
...output omitted...
public class ExpenseServiceMock extends ExpenseService {
 @Override
 public void update(Expense expense) {}

 @Override
 public boolean exists(UUID uuid) {
 return uuid.equals(
 UUID.fromString("efffbfe08-0d71-46e7-87af-1132b1b94d87"));
 }
}
```

- 2.4. At its current state, only the `testUpdateExistingExpense()` test in `ExpenseResourceTest` should pass. This is because you are injecting the `ExpenseResource`, and the previous code instructs Quarkus DI to use `ExpenseServiceMock` for the injection. To verify this, run the tests, to see the current outcome by using the `mvn clean test` command in the terminal.

```
[student@workstation ~]$ cd ~/DO378/labs/test-mock/expense-service/
[student@workstation expense-service]$ mvn clean test
[INFO] Scanning for projects...
...output omitted...
[INFO] -----
[INFO] T E S T S
```

```
[INFO] -----
[INFO] Running org.acme.rest.json.ExpenseResourceTest
...output omitted...
[INFO] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 7.443 s -
in org.acme.rest.json.ExpenseResourceTest
...output omitted...
[INFO] Running org.acme.rest.json.ExpenseServiceTest
[ERROR] Tests run: 2, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 0.296 s
<<< FAILURE! - in org.acme.rest.json.ExpenseServiceTest
...output omitted...
```



### Note

Notice that some test are failing, this is expected as only `testUpdateExistingExpense()` should work.

- ▶ 3. Within the `ExpenseResourceTest.testUpdateNotFoundExpense()` method, create a mock of `ExpenseService` using Mockito. Inject it to `expenseResourceNotFound` by setting its `expenseService` property to be the mock instance. The mock should override the `exists(UUID)` method (the mock found in the previous step), however now it must return `false` when called using the parameter `notFoundUuid`. There is no need to override the `update(Expense)` method, because void methods in mocks do nothing by default.

- 3.1. Add the extension for Mockito by modifying the `pom.xml` file.

```
...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-junit5-mockito</artifactId>
 <scope>test</scope>
</dependency>
...output omitted...
```

- 3.2. Create a mock of the `ExpenseService` class inside of `testUpdateNotFoundExpense()` by using `org.mockito.Mockito.mock()`.

```
...output omitted...
import org.mockito.Mockito;
...output omitted...
public void testUpdateNotFoundExpense() {
 UUID notFoundUuid = UUID.fromString(
 "896bca62-b865-4e04-bc45-d028c44b381a");
 Expense expense = new Expense();
 expense.uuid = notFoundUuid;

 ExpenseService expenseServiceMock = Mockito.mock(ExpenseService.class);
 Mockito.when(expenseServiceMock.exists(notFoundUuid)).thenReturn(false);

 ExpenseResource expenseResourceNotFound = new ExpenseResource();

 Assertions.assertThrows(
 WebApplicationException.class,
```

```

 () -> expenseResourceNotFound.update(expense));
}
...output omitted...

```

- 3.3. Set the `expenseService` property of `expenseResourceNotFound` to `expenseServiceMock`.

```

...output omitted...
ExpenseResource expenseResourceNotFound = new ExpenseResource();
expenseResourceNotFound.expenseService = expenseServiceMock;

Assertions.assertThrows(
 WebApplicationException.class,
 () -> expenseResourceNotFound.update(expense));
...output omitted...

```

- 3.4. Run the tests again and verify the output now shows two successful tests. The `ExpenseServiceTest` will still fail. We will fix that in the next step.

```

[student@workstation expense-service]$ mvn clean test
[INFO] Scanning for projects...
...output omitted...
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running org.acme.rest.json.ExpenseResourceTest
...output omitted...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 7.443 s -
in org.acme.rest.json.ExpenseResourceTest
...output omitted...

```

- 4. Modify the `src/test/java/org/acme/rest/json/ExpenseServiceTest.java` file to use mocks of the `Expense` class using `io.quarkus.panache.mock.PanacheMock`.

- 4.1. Add the `quarkus-panache-mock` extension by modifying the `pom.xml` file.

```

...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-panache-mock</artifactId>
 <scope>test</scope>
</dependency>
...output omitted...

```

- 4.2. Modify the `testDeleteExistingExpense()` method to first mock the `Expense` class, and then mock the `delete()` method, making sure it returns `1L` when called with the parameters "`uuid`" and `existingUuid`.

```

...output omitted...
import org.mockito.Mockito;
import io.quarkus.panache.mock.PanacheMock;
...output omitted...

```

```
public void testDeleteExistingExpense() {
 UUID existingUuid = UUID.fromString(
 "efffbfe08-0d71-46e7-87af-1132b1b94d87");
 PanacheMock.mock(Expense.class);
 Mockito.when(Expense.delete("uuid", existingUuid)).thenReturn(1L);
 ExpenseService expenseService = new ExpenseService();
 Assertions.assertDoesNotThrow () -> expenseService.delete(existingUuid);
}
...output omitted...
```

This will make the test succeed.

- 4.3. Modify the `testDeleteNotFoundExpense()` method to first mock the `Expense` class, and then mock the `delete()` method, making sure it returns `0L` when called with the parameters "uuid" and `notFoundUuid`.

```
...output omitted...
public void testDeleteNotFoundExpense() {
 UUID notFoundUuid = UUID.fromString(
 "896bca62-b865-4e04-bc45-d028c44b381a");
 PanacheMock.mock(Expense.class);
 Mockito.when(Expense.delete("uuid", notFoundUuid)).thenReturn(0L);
 ExpenseService expenseService = new ExpenseService();
 Assertions.assertThrows(
 WebApplicationException.class,
 () -> expenseService.delete(notFoundUuid));
}
...output omitted...
```

- 4.4. Run the tests. All unit tests should now be successful.

```
[student@workstation expense-service]$ mvn clean test
...output omitted...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.232 s -
in org.acme.rest.json.ExpenseResourceTest
[INFO] Running org.acme.rest.json.ExpenseServiceTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.126 s -
in org.acme.rest.json.ExpenseServiceTest
2020-11-05 08:06:52,187 INFO [io.quarkus] (main) Quarkus stopped in 0.027s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
...output omitted...
```

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab test-mock finish
```

This concludes the guided exercise.

## ▶ Lab

# Testing Microservices

In this lab, you will create tests for the Quarkus Conference application, using different techniques provided by the Red Hat Build of Quarkus.

## Outcomes

You should be able to use the features provided by Red Hat Build of Quarkus to develop different kinds of tests.

## Before You Begin

On the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command ensures that the Quarkus Conference application is available on your workstation.

```
[student@workstation ~]$ lab test-review start
```

After running the previous command, the `~/D0378/labs/test-review` folder contains the Quarkus Conference application source code.

This section uses a modified version of the `Schedule` microservice, which uses an external Derby database.



### Warning

Do not use static imports in this exercise. The evaluation script checks for the usage of non-static imports, and fails if not found.

## Instructions

1. The `Schedule` service contains a test suite in the `microservice-schedule/src/test/java/org/acme/conference/schedule/ScheduleResourceTest.java` file. All tests in the suite fail.

The test suite uses `JUnit`, which does not start the Quarkus microservice. Update the microservice to start the `Schedule` microservice in test mode.

The paths associated with each endpoint are absolute, but must be relative to the path defined in the `ScheduleResource` resource. Update the test class to use the base path of the `ScheduleResource` resource automatically.

2. After updating the test, suite tests still fail. The `Schedule` microservice cannot start in test mode because it relies on a Derby database.

Update the test to start a Derby in-memory database when the suite starts. The test database must be controlled by the test, not by the service, so do not use an embedded database. In other words, do not use the `org.apache.derby.jdbc.EmbeddedDriver` driver but the already provided `org.apache.derby.jdbc.ClientDriver` driver.

3. Move to the Speaker service. Open the `src/test/java/org/acme/conference/speaker/SpeakerResourceTest.java` file. The `testNewSpeaker` method validates that generating a new speaker returns a speaker entity with the correct values. This test fails because the Speaker service generates a random UUID using the `RandomIdGenerator` bean.

To solve this issue, override the `RandomIdGenerator` bean by using the provided `DeterministicIdGenerator` class. Make this class a unique bean. During the test, invoke the `DeterministicIdGenerator.setNextUUID` method to set the next UUID provided by the generator to the UUID created during the test.

4. The Speaker service uses an in-memory H2 database, which cannot be controlled by the test suite. Use the `panache-mock` extension to make the `testListAll` test method succeed. Mock the Speaker entity for the `listAll` method to return an empty collection, making the test succeed.
5. In the Vote service, tests in the `RatingStatsResourceTest` class check the correct behavior of the `RatingStatsResource`. Those tests have an external dependency on the `SessionRatingDAO` bean to obtain session data.  
Use Mockito to mock the `SessionRatingDAO` bean. The method `SessionRatingDAO.getRatingsBySession` must return a controlled set of `SessionRating` objects as defined in the `setUp` method.

## Evaluation

Grade your work by running the `lab test-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab test-review grade
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab test-review finish
```

This concludes the lab.

## ► Solution

# Testing Microservices

In this lab, you will create tests for the Quarkus Conference application, using different techniques provided by the Red Hat Build of Quarkus.

## Outcomes

You should be able to use the features provided by Red Hat Build of Quarkus to develop different kinds of tests.

## Before You Begin

On the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command ensures that the Quarkus Conference application is available on your workstation.

```
[student@workstation ~]$ lab test-review start
```

After running the previous command, the `~/D0378/labs/test-review` folder contains the Quarkus Conference application source code.

This section uses a modified version of the `Schedule` microservice, which uses an external Derby database.



### Warning

Do not use static imports in this exercise. The evaluation script checks for the usage of non-static imports, and fails if not found.

## Instructions

1. The `Schedule` service contains a test suite in the `microservice-schedule/src/test/java/org/acme/conference/schedule/ScheduleResourceTest.java` file. All tests in the suite fail.

The test suite uses `JUnit`, which does not start the Quarkus microservice. Update the microservice to start the `Schedule` microservice in test mode.

The paths associated with each endpoint are absolute, but must be relative to the path defined in the `ScheduleResource` resource. Update the test class to use the base path of the `ScheduleResource` resource automatically.

- 1.1. Add the `io.quarkus.test.junit.QuarkusTest` annotation to the `ScheduleResourceTest` class.

```
...output omitted...
import io.quarkus.test.junit.QuarkusTest;
...output omitted...
@QuarkusTest
public class ScheduleResourceTest {
...output omitted...
```

- 1.2. Use the `io.quarkus.test.common.http.TestHTTPEndpoint` to set the base path for all requests to the path defined in the `ScheduleResource` class.

```
...output omitted...
import io.quarkus.test.common.http.TestHTTPEndpoint;
...output omitted...
@QuarkusTest
@TestHTTPEndpoint(ScheduleResource.class)
public class ScheduleResourceTest {
...output omitted...
```

2. After updating the test suite tests still fail. The `Schedule` microservice cannot start in test mode because it relies on a Derby database.

Update the test to start a Derby in-memory database when the suite starts. The test database must be controlled by the test, not by the service, so do not use an embedded database. In other words, do not use the `org.apache.derby.jdbc.EmbeddedDriver` driver but the already provided `org.apache.derby.jdbc.ClientDriver` driver.

- 2.1. Add the `io.quarkus:quarkus-test-derby:test` dependency to the project's `pom.xml` file.

```
...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-test-derby</artifactId>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-resteasy</artifactId>
</dependency>
...output omitted...
```

- 2.2. Add the `QuarkusTestResource` annotation to the test class. Use the `DerbyDatabaseTestResource` class as the annotation's only parameter.

```
...output omitted...
import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.derby.DerbyDatabaseTestResource;
...output omitted...
@QuarkusTest
@TestHTTPEndpoint(ScheduleResource.class)
```

```
@QuarkusTestResource(DerbyDatabaseTestResource.class)
public class ScheduleResourceTest {
...output omitted...
```

- 2.3. Update the `application.properties` file to configure the Schedule service to use the test database when executing tests. Because the database kind and drivers are the same, only specify the new JDBC URL pointing to the localhost database.

```
...output omitted...
quarkus.datasource.db-kind=derby
%test.quarkus.datasource.jdbc.url=jdbc:derby://localhost/scheds;create=true
quarkus.datasource.jdbc.url=jdbc:derby://derby/scheds
quarkus.datasource.jdbc.driver=org.apache.derby.jdbc.ClientDriver
...output omitted...
```

- 2.4. Validate that all tests in the suite pass.

```
[student@workstation ~]$ cd ~/D0378/labs/test-review/microservice-schedule
[student@workstation microservice-schedule]$ mvn test
...output omitted...
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
...output omitted...
```

3. Move to the Speaker service. Open the `src/test/java/org/acme/conference/speaker/SpeakerResourceTest.java` file. The `testNewSpeaker` method validates that generating a new speaker returns a speaker entity with the correct values. This test fails because the Speaker service generates a random UUID using the `RandomIdGenerator` bean.

To solve this issue, override the `RandomIdGenerator` bean by using the provided `DeterministicIdGenerator` class. Make this class a unique bean. During the test, invoke the `DeterministicIdGenerator.setNextUUID` method to set the next UUID provided by the generator to the UUID created during the test.

- 3.1. The `DeterministicIdGenerator` class is not a managed bean. Add the `javax.inject.Singleton` to make it a unique bean, and the `@Mock` annotation to override other beans implementing the same interfaces.

```
...output omitted...
import javax.inject.Singleton;
import io.quarkus.test.Mock;
...output omitted...
@Mock
@Singleton
public class DeterministicIdGenerator implements IdGenerator {
...output omitted...
```

In the `SpeakerResourceTest` test class, inject the `DeterministicIdGenerator` instance using the `javax.inject.Inject` annotation.

```
...output omitted...
import javax.inject.Inject;
...output omitted...
public class SpeakerResourceTest {
...output omitted...
 @Inject DeterministicIdGenerator idGenerator;
...output omitted...
```

Invoke the `DeterministicIdGenerator.setNextUUID` method using provided UUID as the only parameter. Make sure you invoke this method before the test invokes the service endpoint.

```
...output omitted...
@Test
public void testNewSpeaker() {

 UUID uuid = new UUID(1, 1);

 idGenerator.setNextUUID(uuid);

 given()
...output omitted...
```

- The Speaker service uses an in-memory H2 database, which cannot be controlled by the test suite. Use the `panache-mock` extension to make the `testListAll` test method succeed. Mock the Speaker entity for the `listAll` method to return an empty collection, making the test succeed.

#### 4.1. Add the `io.quarkus:quarkus-panache-mock` extension to the `pom.xml` file.

```
...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-panache-mock</artifactId>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-resteasy</artifactId>
</dependency>
...output omitted...
```

Use PanacheMock and Mockito to make the `Speaker.listAll` method return an empty list.

```
...output omitted...
import io.quarkus.panache.mock.PanacheMock;
import org.mockito.Mockito;
import java.util.Collections;
...output omitted...
@Test
public void testListAll() {
 PanacheMock.mock(Speaker.class);
```

```
Mockito.when(Speaker.listAll())
 .thenReturn(Collections.emptyList());

 given()
 ...output omitted...
```

4.2. Validate that all tests in the suite pass.

```
[student@workstation ~]$ cd ~/D0378/labs/test-review/microservice-speaker
[student@workstation microservice-schedule]$ mvn test
...output omitted...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
...output omitted...
```

5. In the Vote service, tests in the RatingStatsResourceTest class check the correct behavior of the RatingStatsResource. Those tests have an external dependency on the SessionRatingDAO bean to obtain session data.

Use Mockito to mock the SessionRatingDAO bean. The method SessionRatingDAO.getRatingsBySession must return a controlled set of SessionRating objects as defined in the setUp method.

5.1. Add the quarkus-junit5-mockito extension to the pom.xml file.

```
...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-junit5-mockito</artifactId>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-resteasy</artifactId>
</dependency>
...output omitted...
```

5.2. Use @InjectMock to inject the SessionRatingDAO mock bean into the RatingStatsResourceTest test class.

```
...output omitted...
import io.quarkus.test.junit.mockito.InjectMock;
...output omitted...
public class RatingStatsResourceTest {
 ...output omitted...
 @InjectMock SessionRatingDAO sessionRatingDAO;
 ...output omitted...
```

5.3. In the setUp method, use Mockito to instruct the mock to return the values in the threeConsecutiveRatings collection.

```
...output omitted...
import static org.mockito.ArgumentMatchers.eq;
import org.mockito.Mockito;
...output omitted...
```

```
@BeforeEach
public void setUp () {
 List<SessionRating> threeConsecutiveRatings = ...output omitted...

 Mockito.when(sessionRatingDAO.getRatingsBySession(eq(RATINGSTATS_SESSION)))
 .thenReturn(threeConsecutiveRatings);
}
...output omitted...
```

5.4. Validate that all tests in the suite pass.

```
[student@workstation ~]$ cd ~/DO378/labs/test-review/microservice-vote
[student@workstation microservice-schedule]$ mvn test
...output omitted...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
...output omitted...
```

## Evaluation

Grade your work by running the `lab test-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab test-review grade
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab test-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- Quarkus integrates unit test features, based on the Maven Surefire plugin and JUnit5. It extends them with integration test features, based on the `@QuarkusTest` annotation and REST-assured.
- Tests can benefit from application bean injection, endpoint injection, and test dependencies provisioning using `@Inject`, `@TestHTTPEndpoint`, and `@QuarkusTestResource` annotations, respectively.
- Quarkus approaches mocking in three ways:
  - The `@Mock` annotation for overriding generic beans
  - The `quarkus-junit-mockito` extension for mocking application classes
  - The `quarkus-panache-mock` extension to mock Panache active record entities

## Chapter 8

# Securing Microservices Communication

### Goal

Secure microservice communications by applying origin validation, requests authentication and authorization.

### Objectives

- Validate the origin of microservice communications by using TLS and CORS.
- Apply authentication and authorization techniques to Quarkus microservices.

### Sections

- Implementing Communication Security in Quarkus Microservices (and Guided Exercise)
- Authenticating and Authorizing Requests (and Guided Exercise)

### Lab

Securing Microservices

# Implementing Communication Security in Quarkus Microservices

## Objectives

After completing this section, you should be able to validate the origin of microservice communications by using TLS and CORS.

## Cross-origin Resource Sharing (CORS)

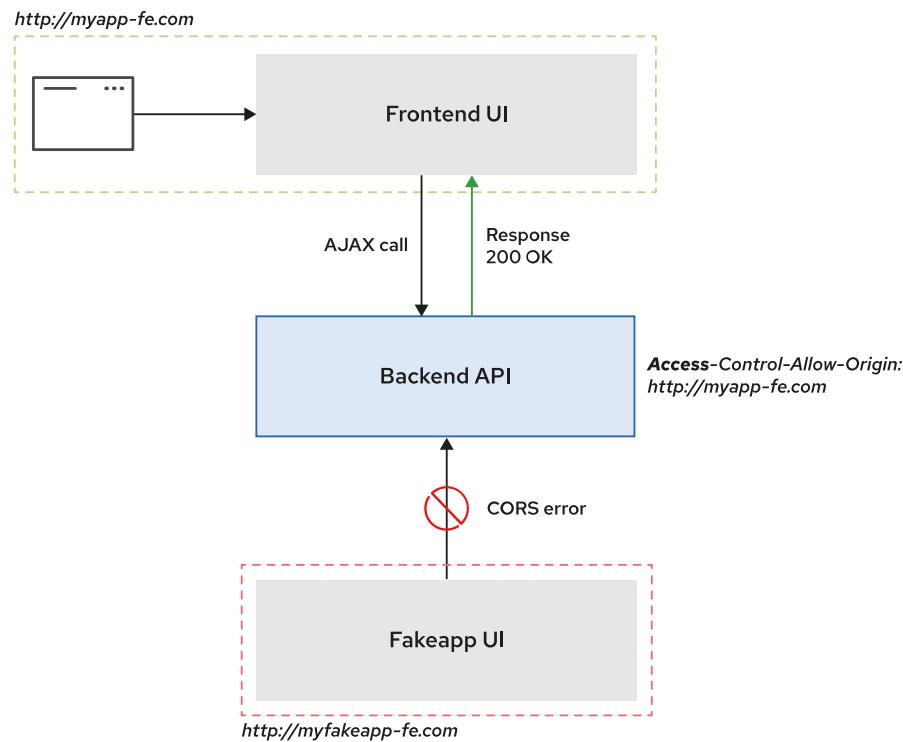
*Cross-origin resource sharing (CORS)* allows a web page to request resources from an external domain. Although images, style sheets, JavaScript files, iframes, and video files can be loaded from external sources, AJAX calls from a browser front-end to a back-end running on an external domain are forbidden in most modern web browsers.

In a modern microservices based architecture, a set of independent back-end microservices running on multiple servers provide a REST API, which are then invoked by using HTTP calls from a front-end typically running on a browser.

CORS allows a browser to interact with back end services in a secure manner and allows cross-origin requests. Most modern browsers enable CORS for cross-origin requests by default.

To allow cross-origin requests on the back end, you must configure your server to respond to cross-origin requests with a set of HTTP response headers. For example, assume a HTML page on servera makes a cross-origin AJAX request to serverb, the following happens.

- The browser sends a HTTP GET request with the `Origin: http://myapp-fe.com` HTTP header to `myapp-be.com`.
- The `myapp-be.com` server can be configured to allow only specific domains to make CORS requests, in which case it responds with a `Access-Control-Allow-Origin: http://myapp-fe.com` header.
- `myapp-be.com` can also be configured to allow wildcard domains, or any domain to make CORS requests, in which case it responds with an `Access-Control-Allow-Origin: *` header.



**Figure 8.1: Cross-origin resource sharing validation**

A wild card CORS policy is useful in cases where your API is serving public content that should be accessible by everyone, and where the requesting domain cannot be known beforehand. If the server is serving a set of well known domains, then you should configure your server to allow only specific domains in the `Access-Control-Allow-Origin` header.

CORS also allows the server to have finer grained control over what HTTP operations (POST, GET, PUT, DELETE and more) are allowed. Refer to the links in the references to see a full list of CORS related headers.

## Configuring CORS in Quarkus Applications

Quarkus comes with a CORS filter, which implements the `javax.servlet.Filter` interface and intercepts all incoming HTTP requests. CORS can be enabled in the Quarkus `application.properties` configuration file. You can enable and configure CORS related headers in this file.

A sample configuration for an application to enable CORS is as follows.

```
quarkus.http.cors=true ①
quarkus.http.cors.origins=http://example.com,http://www.example.io ②
quarkus.http.cors.methods=GET,PUT,POST ③
quarkus.http.cors.headers=X-My-Custom-Header,X-Another-Header ④
```

- ① Enable CORS. The default value is false.

- ② Comma separated list of domains, which are allowed to make CORS requests. All domains are allowed if this property is not set (defaults to '\*', allow all).
- ③ Comma separated list of HTTP methods allowed in CORS requests. All HTTP methods are allowed if this property is not set (defaults to '\*', allow all).
- ④ Comma separated list of HTTP headers allowed in CORS requests. All HTTP headers are allowed if this property is not set (defaults to '\*', allow all).

## Transport Layer Security (TLS)

Transport Layer Security (TLS) is a method for encrypting network communications. TLS is the successor to Secure Sockets Layer (SSL). TLS allows a client to verify the identity of the server and, optionally, allows the server to verify the identity of the client.

TLS is particularly important in the context of a microservices based architecture, where multiple separate services communicate with each other over HTTP based protocols. Enabling TLS prevents a malicious user from eavesdropping or tampering with the communication between services.

TLS is based around the concepts of certificates. A certificate has multiple parts, a public key, a private key, and a signature from a certificate authority. The private key is never made public. Any data encrypted with the private key can only be decrypted with the public key, and vice versa.

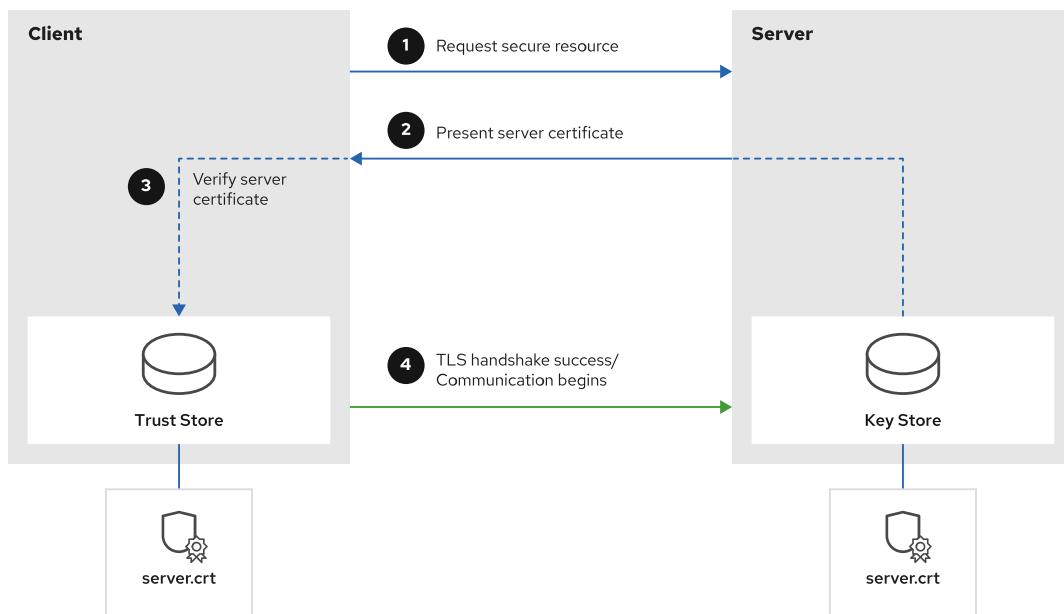


Figure 8.2: One-way Transport Layer Security

## Configuring TLS for Quarkus Applications

To enable TLS support in Quarkus applications, you must either provide a certificate and associated key file, or supply a keystore. All TLS configuration is done in the Quarkus `application.properties` file for the application.

**Note**

Although you can generate certificates by using the `openssl` utility, the preferred approach for TLS in Java applications is to use the Java `keytool` utility, which is available by default in all JVM installations. The `keytool` utility is cross-platform and the generated artifacts are compatible with `openssl`. You can use both tools to convert between the different key storage formats supported by the respective tools.

You will use the `keytool` utility in all the labs in this course.

The following are the steps to enable TLS in Quarkus applications.

- 1 Generate a key store for your application that contains a TLS key-pair. A key store contains a private key and an associated certificate for an application. You generate a key store by using the Java `keytool` utility.

```
keytool -noprompt -genkeypair -keyalg RSA -keysize 2048
 -validity 365 \ ①
 -dname "CN=myapp, OU=myorgunit, O=myorg, L=myloc, ST=state, C=country" \ ②
 -ext "SAN:c=DNS:myserver,IP:127.0.0.1" \ ③
 -alias myapp -storetype JKS \ ④
 -storepass mypass -keypass mypass \ ⑤
 -keystore myapp.keystore ⑥
```

- 1 Validity of the certificate in days.
  - 2 Unique name that identifies this application.
  - 3 Subject Alternative Name (SAN), which provides the DNS name of the server where the application is running and the IP address.
  - 4 Key store type. Keys can be stored in different formats. Common options are JKS, PKCS12, PKCS11. Default is PCKS12 since Java 9.
  - 5 Password to access key store and keys stored in them respectively.
  - 6 Path to generated key store.
- 2 To use the generated key store, add the following properties.

```
quarkus.http.ssl.certificate.key-store-file=/path/to/myapp.keystore
quarkus.http.ssl.certificate.key-store-password=mypass
quarkus.http.ssl-port=8443
```

Quarkus first tries to resolve the given path as a classpath resource, before attempting to read it from the file system. The type of key store can be provided as one of the following.

```
quarkus.http.ssl.certificate.key-store-file-type=[one of JKS, JCEKS, P12, PKCS12,
 PFX]
```

If the type is not provided, Quarkus attempts to deduce it from the file extensions, defaulting to type JKS.

The default port for TLS is 8443. The default port for unit tests is 8444, and it can be changed by setting the `quarkus.http.test-ssl-port` property.

For Quarkus native builds, to enable TLS you must set the `quarkus.ssl.native` property to `true`.

It is possible to disable the HTTP port and only support secure requests. This is done via the `quarkus.http.insecure-requests` property. There are three possible values.

#### **enabled**

The default option, both HTTP (8080) and HTTPS (8443) ports are open.

#### **redirect**

Plain HTTP requests are automatically redirected to the HTTPS port.

#### **disabled**

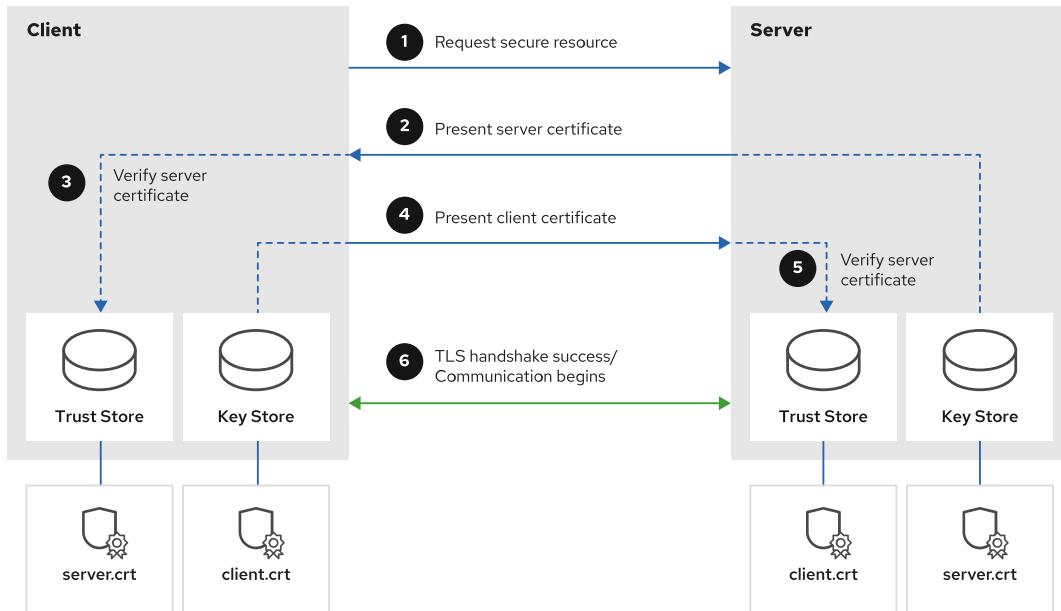
The HTTP port will not be opened. Use this value for applications that should always use the HTTPS protocol.

## Configuring Mutual TLS (mTLS) for Quarkus Applications

Although TLS proves the identity of the server to the client, the identity of the client is verified in the application layer. Mutual TLS authentication, or *two way authentication* is an extension of TLS, where traffic between the client and server is secure and trusted by both parties.

Mutual TLS is useful in a microservices based architecture in cases where you want to implement a "zero-trust" security model, where all communications happen between services only after they have verified the identity of each other. It is also recommended in scenarios where your application API is going to be invoked by external parties, outside your organizations network.

The client verifies the identity of the server, and the server verifies the identity of the client.



**Figure 8.3: Mutual Transport Layer Security**

Mutual TLS in Quarkus applications requires a special type of key store named a trust store. The trust store stores only public certificates of applications it trusts.

The following are the steps to enable mutual TLS in Quarkus applications.

1. Create a trust store. Import the public key certificates of all the applications that want to communicate with your application. Extract the certificate from the key store of a client application by using the following command.

```
keytool -exportcert -keystore myclient.keystore \ ①
-storepass mypass -alias myclient \ ②
-rfc -file myclient.crt ③
```

1. Path to key store of client application. The key store for the client application is also created using the `keytool` utility.
  2. Short alias name to identify the public key of this client application.
  3. Path to generated client application certificate.
2. Import the client certificate into the server trust store by using the following command.

```
keytool -noprompt -keystore myapp.truststore \ ①
-importcert -file myapp.crt \ ②
-alias myclient ③
-storepass mypass
```

1. Path to trust store file for your application (server, not the client).
  2. Path to certificate for the client application.
  3. Short alias name to identify the client application.
3. Use the same previous commands to export the certificate from the keystore of the server and import the certificate on the trust store of the client.
  4. Configure the client and server for mutual authentication. Start by adding the following properties to the server.

```
quarkus.http.ssl.client-auth=required
quarkus.http.ssl.certificate.trust-store-file=/path/to/myapp.truststore
quarkus.http.ssl.certificate.trust-store-password=mypass
```

5. On the client side, you must add extra \*/mp-rest properties in the `application.properties` to securely communicate with the server.

```
org.acme.client.mtls.GreetingService/mp-rest/url=https://myserver:8443 ①
org.acme.client.mtls.GreetingService/mp-rest/trustStore=/path/to/
myclient.truststore ②
org.acme.client.mtls.GreetingService/mp-rest/trustStorePassword=mypass
org.acme.client.mtls.GreetingService/mp-rest/keyStore=/path/myclient.keystore ③
org.acme.client.mtls.GreetingService/mp-rest/keyStorePassword=mypass
```

1. URL and HTTPS port where server application is listening for encrypted traffic.
2. Path to client application trust store.
3. Path to client application key store.

## Implementing Security at the Network Layer

Quarkus follows the approach of implementing security features in the code. This approach facilitates adding security to individual services but does not scale well for applications with dozens or hundred of microservices.

Red Hat OpenShift Service Mesh offers a different approach: it implements security at the network layer. This means that operators manage security (and other features) for all microservices in a single point of configuration named the **control plane**.

For more information about Red Hat OpenShift Service Mesh and the upstream project Istio see the related reference links at the end of the section or the DO328 course included in your Red Hat Online Learning subscription.



### References

#### **Cross-Origin Resource Sharing (CORS)**

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

#### **CORS filter in Quarkus**

<https://quarkus.io/version/1.11/guides/http-reference#cors-filter>

#### **Using TLS with Quarkus native executables**

<https://quarkus.io/version/1.11/guides/native-and-ssl>

#### **Learn how to do Mutual TLS in Quarkus apps**

<https://quarkus.io/blog/quarkus-mutual-tls/>

#### **Quarkus mTLS example**

<https://github.com/openlab-red/quarkus-mtls-quickstart>

#### **Difference Between a Java Keystore and a Truststore**

<https://www.baeldung.com/java-keystore-truststore-difference>

#### **Difference between openssl and keytool**

<https://security.stackexchange.com/questions/98282/difference-between-openssl-and-keytool>

#### **Red Hat OpenShift Service Mesh product page**

[https://access.redhat.com/documentation/en-us/redshift\\_container\\_platform/4.6/html-single/service\\_mesh/index](https://access.redhat.com/documentation/en-us/redshift_container_platform/4.6/html-single/service_mesh/index)

#### **Istio documentation**

<https://istio.io/latest/docs/>

## ► Guided Exercise

# Implementing Communication Security in Quarkus Microservices

In this exercise, you will secure communication between Quarkus microservices, and allow a browser to invoke services by enabling cross origin resource sharing (CORS).

## Outcomes

You should be able to.

- Enable cross origin resource sharing (CORS) in a Quarkus application.
- Enable Transport Layer Security (TLS) for Quarkus applications.
- Enable mutual TLS (mTLS) for Quarkus applications.

## Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command downloads all the lab files and script required for the exercise.

```
[student@workstation ~]$ lab secure-tls start
```

## Instructions

- 1. Start the microservices in the Quarkus calculator application. This application consists of three microservices. The services communicate with each other by using plain unencrypted HTTP.
- 1.1. Briefly review the source code of the Quarkus Calculator application. The source code is in the `~/D0378/labs/secure-tls/` folder. Use VSCode to open the folder as a new Quarkus application.

```
[student@workstation ~]$ codium ~/D0378/labs/secure-tls/
```

Each of the three microservices expose their own REST API. A single client call to solve an equation could result in multiple calls between these microservices.

- 1.2. Inspect the `~/D0378/labs/secure-tls/start.sh` script. This script starts all three microservices in dev mode.

```
...output omitted...
echo "Starting the 'solver' project "
cd solver
mvn clean quarkus:dev -Ddebug=5005 &
...output omitted...
```

13. Run the `~/D0378/labs/secure-tls/start.sh` script.

```
[student@workstation ~]$ sh ~/D0378/labs/secure-tls/start.sh
Starting the 'solver' project
...output omitted...
Starting the 'adder' project
...output omitted...
Starting the 'multiplier' project
...output omitted...
Press enter to Terminate
...output omitted...
```

- ▶ 2. Open the web user interface for the Quarkus application by using a browser. Configure the `solver` service to allow cross domain requests from the browser.

- 2.1. Inspect the `~/D0378/labs/secure-tls/index.html` file in VSCode. This web page captures a calculator expression and invokes the `solver` service to solve it.

```
...output omitted...
// call the solver service
var response = $.get('http://localhost:8080/solver/' + term);
...output omitted...
```

- 2.2. Open the `~/D0378/labs/secure-tls/index.html` file by using the Firefox web browser.

```
[student@workstation ~]$ firefox ~/D0378/labs/secure-tls/index.html
```

You will see a simple form with a text field for entering an expression and a button to submit the form.

## Quarkus Solver

Enter expression to solve

### NOTE

- Solver does not support subtraction or division. Do not enter expressions with the `"/"` or `"+"` symbol.
- Do not enter non-numeric expressions or special characters other than `"+"`, `"*"` and `".."`.
- Integer and decimal numbers are allowed.
- All results are converted to decimals (Java float).

- 2.3. Enter an expression in the text field. For example,  $5*4+3$ . Click **Solve**.

You will see an error.

**ERROR: failed to connect to server**

Open the Firefox Developer panel by clicking **Open Menu (Three vertical bars icon in the top right corner) > Web Developer > Web Console**.

You will see an error stating that the cross origin request is being blocked.

The screenshot shows the Firefox Developer Tools Web Console. At the top, there is an input field containing the equation  $5*4+3$  and a 'Solve' button. Below the input field, the text 'ERROR: failed to connect to server' is displayed in red. The 'Console' tab is selected in the toolbar. In the main pane, a red warning message is shown: 'Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at [http://localhost:8080/solver/5\\*4+3](http://localhost:8080/solver/5*4+3). (Reason: CORS header 'Access-Control-Allow-Origin' missing). [\[Learn More\]](#)'. Other tabs in the toolbar include Inspector, Debugger, Network, and Requests.

- 2.4. Allow the browser to invoke the solver service.

Edit the `~/D0378/labs/secure-tls/solver/src/main/resources/application.properties` file, and add the following property.

**quarkus.http.cors=true**

Save your changes to the `application.properties` file.

- 2.5. Refresh the browser and submit an equation. You should now see the result being printed below the text field.

The screenshot shows the Firefox Developer Tools Web Console after the 'quarkus.http.cors=true' property was added to the application properties. The input field now contains the result 'Result: 23.0'. The 'Console' tab is selected in the toolbar. The main pane is empty, indicating that the CORS issue has been resolved.

- 2.6. In the terminal where you ran the `start.sh` script to start the microservices, press `Enter` to stop all the microservices.
- 3. Secure the `solver` service by generating self-signed public key infrastructure (PKI) certificates, and enabling TLS for secure communication.
- 3.1. Inspect the `~/D0378/labs/secure-tls/1-genkeys.sh` script. This script generates a TLS key-pair for each of the three microservices using the Java `keytool` utility.

```
...output omitted...
echo "Generating keystore file for solver..."
keytool -noprompt -genkeypair -keyalg RSA -keysize 2048 -validity 365 \
-dname "CN=solver,OU=Training, O=RedHat, L=Raleigh, ST=NC, C=US" \
-ext "SAN:c=DNS:localhost,IP:127.0.0.1" \
-alias solver \
-storepass redhat -keypass redhat \
-keystore solver/solver.keystore
...output omitted...
```

**Note**

You only need the keystore for the `solver` service in this step. The keystores for the `adder` and `multiplier` service are used later in the exercise.

- 3.2. Execute the `1-genkeys.sh` script.

```
[student@workstation ~]$ sh ~/D0378/labs/secure-tls/1-genkeys.sh
Generating keystore file for solver...
Generating keystore file for adder...
Generating keystore file for multiplier...
```

- 3.3. Enable TLS for the `solver` service. Edit the `~/D0378/labs/secure-tls/solver/src/main/resources/application.properties` file, and add the following properties.

```
quarkus.http.ssl-port=8443 ①
quarkus.http.ssl.certificate.key-store-file=../solver.keystore ②
quarkus.http.ssl.certificate.key-store-password=redhat ③
```

- ① The TLS port to listen to for secure communication with clients.
- ② The relative path to the keystore file. Note that in Quarkus dev mode, the root path is the `target` directory for each service. The path to the keystore is relative to this directory.
- ③ The keystore password.

- 3.4. Execute the `~/D0378/labs/secure-tls/start.sh` script to restart the services. Wait until all three services have started.

In a new terminal window, run the `curl` command-line utility to test the `solver` service.

```
[student@workstation ~]$ curl https://localhost:8443/solver/5*4+3
curl: (60) SSL certificate problem: self signed certificate
...output omitted...
```

The client refuses to connect because you are using a self-signed certificate for this exercise.

Use the -k flag of curl to bypass the certificate check.

```
[student@workstation ~]$ curl -k https://localhost:8443/solver/5*4+3
23.0
```

- 3.5. You can pass the -ikv flag to the curl command to check the certificate details of solver and confirm that communication is secured.

```
[student@workstation ~]$ curl -ikv https://localhost:8443/solver/5*4+3
...output omitted...
* Connected to localhost (::1) port 8443 (#0)
...output omitted...
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
...output omitted...
* Server certificate:
* subject: C=US; ST=NC; L=Raleigh; O=RedHat; OU=Training; CN=solver
* start date: Dec 8 07:09:51 2020 GMT
* expire date: Dec 8 07:09:51 2021 GMT
* issuer: C=US; ST=NC; L=Raleigh; O=RedHat; OU=Training; CN=solver
* SSL certificate verify result: self signed certificate (18), continuing anyway.
...output omitted...
HTTP/2 200
...output omitted...
23.0
```

- 3.6. In the terminal where you ran the `start.sh` script to start the microservices, press `Enter` to stop all the microservices.
4. In the previous step, the communication between the client and the solver is secure. However, the communication between the three back-end microservices is still unencrypted.  
In this step, you configure mutual authentication between the services, so that all communication is encrypted and secure.  
Configure solver to allow unencrypted HTTP traffic on port 8080 from clients. Configure adder and multiplier to only allow secure encrypted traffic on port 8444 and 8445

respectively. Calls to `solver` from `adder` and `multiplier` should be allowed only on port 8443.

- 4.1. The `solver` service invokes the `adder` and `multiplier` service depending on the equation. You must import the public keys of the `adder` and `multiplier` service into the trust store of the `solver` service for mutual authentication to be successful.

Because `adder` and `multiplier` also invoke `solver` to reduce compound expressions, you must also import the public key of the `solver` service into the trust stores of both `adder` and `multiplier`.

You must first extract the public keys of the three services from their corresponding keystores. To do this, inspect the `~/D0378/labs/secure-tls/2-export-certs.sh` script, which extracts the certificates.

```
...output omitted...
echo "Exporting certificate for adder..."
keytool -exportcert -keystore adder/adder.keystore \
-storepass redhat -alias adder \
-rfc -file adder/adder.crt
...output omitted...
```

- 4.2. Execute the `2-export-certs.sh` script.

```
[student@workstation ~]$ sh ~/D0378/labs/secure-tls/2-export-certs.sh
Exporting certificate for adder...
Certificate stored in file <adder/adder.crt>
Exporting certificate for multiplier...
Certificate stored in file <multiplier/multiplier.crt>
Exporting certificate for solver...
Certificate stored in file <solver/solver.crt>
```

- 4.3. Inspect the `~/D0378/labs/secure-tls/3-gentrust.sh` script, which creates trust stores for the three services, and imports the required public keys of the services it communicates with.

Execute the `3-gentrust.sh` script.

```
[student@workstation ~]$ sh ~/D0378/labs/secure-tls/3-gentrust.sh
Importing certificate for adder into solver trust store...
Certificate was added to keystore
Importing certificate for multiplier into solver trust store...
Certificate was added to keystore
Importing certificate for solver into adder trust store...
Certificate was added to keystore
Importing certificate for solver into multiplier trust store...
Certificate was added to keystore
```

- 4.4. Configure the `solver` service for mutual authentication.

Start by configuring the location of the trust store for the `solver` service and enable mutual authentication. Add the following properties to the `~/D0378/labs/secure-tls/solver/src/main/resources/application.properties` file.

```
...output omitted...
quarkus.http.ssl.certificate.trust-store-file=../solver.truststore ①
quarkus.http.ssl.certificate.trust-store-password=redhat ②
quarkus.http.ssl.client-auth=required ③
```

- ① Relative location of the trust store for this service.
- ② Password for the trust store.
- ③ Enable mutual authentication.

The adder and multiplier service calls should happen over secure ports 8444 and 8445 respectively. Replace the value of the com.redhat.training.service.AdderService/mp-rest/url, and com.redhat.training.service.MultiplierService/mp-rest/url with the following values.

```
com.redhat.training.service.AdderService/mp-rest/url=https://localhost:8444
com.redhat.training.service.MultiplierService/mp-rest/url=https://localhost:8445
```

You also must explicitly provide the key store and trust store location for each service you invoke. Add the following properties below the corresponding https \*mp-rest/url property.

```
...output omitted...
com.redhat.training.service.AdderService/mp-rest/url=https://localhost:8444
com.redhat.training.service.AdderService/mp-rest/trustStore=../solver.truststore
com.redhat.training.service.AdderService/mp-rest/trustStorePassword=redhat
com.redhat.training.service.AdderService/mp-rest/keyStore=../solver.keystore
com.redhat.training.service.AdderService/mp-rest/keyStorePassword=redhat

com.redhat.training.service.MultiplierService/mp-rest/url=https://localhost:8445
com.redhat.training.service.MultiplierService/mp-rest/trustStore=../solver.truststore
com.redhat.training.service.MultiplierService/mp-rest/trustStorePassword=redhat
com.redhat.training.service.MultiplierService/mp-rest/keyStore=../solver.keystore
com.redhat.training.service.MultiplierService/mp-rest/keyStorePassword=redhat
```

Save your changes.

#### 4.5. Configure the adder service for mutual authentication.

Configure the location of the key store and trust store for the adder service and enable mutual authentication. Configure the service to not allow insecure HTTP requests. Add the following properties to the ~/D0378/labs/secure-tls/adder/src/main/resources/application.properties file.

```
...output omitted...
quarkus.http.ssl-port=8444
quarkus.http.ssl.certificate.key-store-file=../adder.keystore
quarkus.http.ssl.certificate.key-store-password=redhat
quarkus.http.ssl.certificate.trust-store-file=../adder.truststore
quarkus.http.ssl.certificate.trust-store-password=redhat
quarkus.http.ssl.client-auth=required
quarkus.http.insecure-requests=disabled
```

Change the URL of the solver service to the secure port (8443). Add the key store and trust store properties for securely calling solver.

```
...output omitted...
com.redhat.training.service.SolverService/mp-rest/url=https://localhost:8443
com.redhat.training.service.SolverService/mp-rest/trustStore=../adder.truststore
com.redhat.training.service.SolverService/mp-rest/trustStorePassword=redhat
com.redhat.training.service.SolverService/mp-rest/keyStore=../adder.keystore
com.redhat.training.service.SolverService/mp-rest/keyStorePassword=redhat
...output omitted...
```

Save your changes.

#### 4.6. Configure the multiplier service for mutual authentication.

Configure the location of the key store and trust store for the multiplier service and enable mutual authentication. Configure the service to not allow insecure HTTP requests. Add the following properties to the ~/D0378/labs/secure-tls/multiplier/src/main/resources/application.properties file.

```
...output omitted...
quarkus.http.ssl-port=8445
quarkus.http.ssl.certificate.key-store-file=../multiplier.keystore
quarkus.http.ssl.certificate.key-store-password=redhat
quarkus.http.ssl.certificate.trust-store-file=../multiplier.truststore
quarkus.http.ssl.certificate.trust-store-password=redhat
quarkus.http.ssl.client-auth=required
quarkus.http.insecure-requests=disabled
```

Change the URL of the solver service to the secure port (8443). Add the key store and trust store properties for securely calling solver.

```
...output omitted...
com.redhat.training.service.SolverService/mp-rest/url=https://localhost:8443
com.redhat.training.service.SolverService/mp-rest/trustStore=../
multiplier.truststore
com.redhat.training.service.SolverService/mp-rest/trustStorePassword=redhat
com.redhat.training.service.SolverService/mp-rest/keyStore=../multiplier.keystore
com.redhat.training.service.SolverService/mp-rest/keyStorePassword=redhat
...output omitted...
```

Save your changes.

- ▶ 5. Test the Quarkus calculator application and verify that it continues to work.

- 5.1. Restart the services by running the `~/D0378/labs/secure-tls/start.sh` script.
- 5.2. Test the application using `curl`. Invoke the secure endpoint of the `solver` service.

```
[student@workstation ~]$ curl -k https://localhost:8443/solver/5*4+3
curl: (35) error:1401E412:SSL routines:CONNECT_CR_FINISHED:sslv3 alert bad
certificate
```

Note the bad certificate error. The `solver` service is expecting the `curl` client to provide a valid certificate before allowing the connection.

You configured only `solver` to allow unencrypted HTTP requests. Communication between the three services happens over the secure encrypted HTTPS ports.

Invoke the HTTP endpoint of the `solver` service.

```
[student@workstation ~]$ curl http://localhost:8080/solver/5*4+3*3
29.0
```



### Note

If for some reason, you want to delete all the created certificates, key stores and trust stores created during the exercise, inspect and run the `~/D0378/labs/secure-tls/clean-stores.sh` script, and then re-run the helper scripts to create new TLS artifacts.

## Finish

Press `Enter` in the terminal where you ran the `start.sh` script to terminate the services. Close the terminal.

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab secure-tls finish
```

This concludes the guided exercise.

# Authenticating and Authorizing Requests

## Objectives

After completing this section, you should be able to apply authentication and authorization techniques to Quarkus microservices.

## Describing Authorization and Authentication in a Microservice-based Application

In a microservice-based application, *authentication* establishes a level of trust between microservices by verifying the identity of the origin of a request. *Authorization* checks whether the identity can perform the request. Consequently, authentication is a prerequisite for authorization.

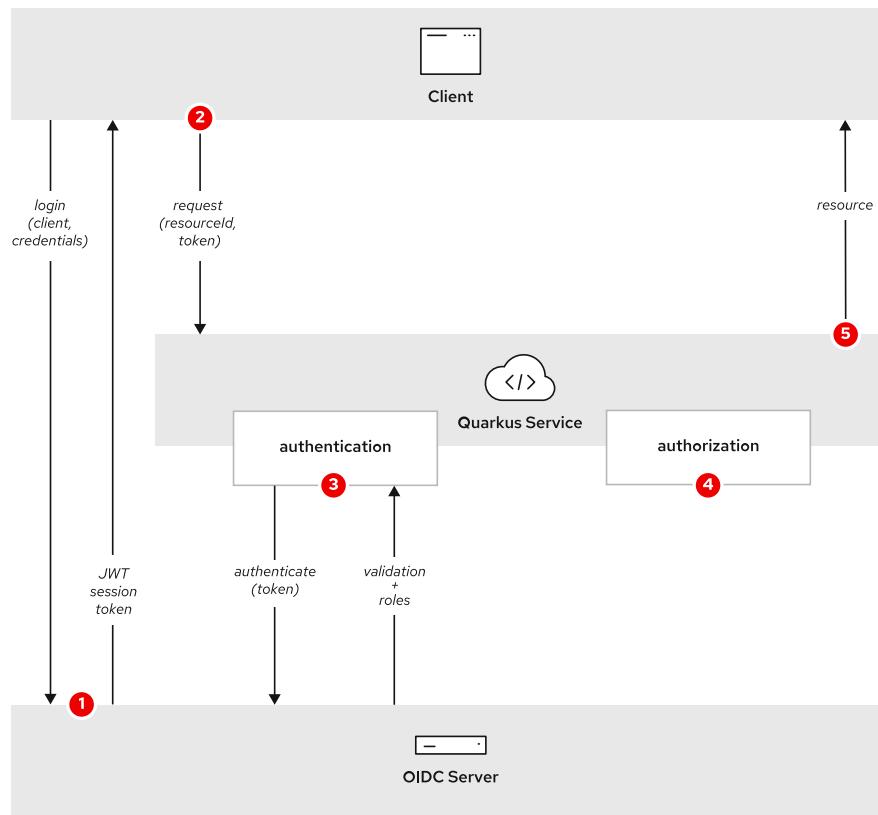
In a monolithic application, authorization and authentication concerns are often focused on securing user facing APIs. In a distributed application, such concerns affect every part of the system. Both authorization and authentication in a distributed application apply to:

- Requests from outside of the application, such as user requests.
- Requests from within the distributed application, such as a service requesting information from another service.

In an ideal production-ready distributed application, permissions for all identities should be as restricted as possible in order to minimize the attack surface of the application. Additionally, all internal requests are authenticated with a cryptographically verifiable identity. Unauthenticated requests are treated with caution because they can contain malicious payloads or can attempt to impersonate other services.

## Implementing Authentication and Authorization with OIDC

*OpenID Connect* (OIDC) is an authentication and authorization layer added to the OAuth 2.0 protocol. Those protocols delegate the creation of the session token, the validation, and optionally the authorization, to a OIDC-compliant server.



- ➊ The client provides the credential to the server and obtains the session token.
- ➋ The client requests the needed resource, providing the session token.
- ➌ The service forwards the token to the OIDC server that performs the authentication and, if positive, provides the roles associated to the identity.
- ➍ The service validates the roles associated to the identity against the roles required as defined in the service.
- ➎ If the token is authentic and the access authorized then the service returns the requested resource.

You can use the Quarkus OpenID Connect Extension to secure your JAX-RS applications. OpenID Connect requires an OIDC-compliant and OAuth-compliant authentication server, such as Red Hat Single Sign On (SSO). To use the OpenID Quarkus extension, first add it to your project:

```
./mvn quarkus:add-extension -Dextensions="oidc"
```

Alternatively, add the dependency manually into the `pom.xml` file:

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-oidc</artifactId>
</dependency>
```

Configure the OpenID Quarkus extension in the `application.properties` property file. The simplest OIDC configuration consists of the following properties:

```
quarkus.oidc.auth-server-url=http://oidc-server:8180/auth/realm/REALM_NAME①
quarkus.oidc.client-id=CLIENT_NAME②
quarkus.oidc.credentials.secret=CLIENT_SECRET③
```

- ① The URL of the OIDC server realm, which contains identities you want to authenticate and authorize.
- ② The client ID, used by the Quarkus application to request identity information.
- ③ The client credentials, if the client is protected by a password. Define other protection methods by using the `quarkus.oidc.credentials.client-secret.method` entry.

If you need access to the JWT claims, or if you need the JWT token for further requests, then you can inject the `org.eclipse.microprofile.jwt.JsonWebToken` class in your resource. Alternatively, you can inject the `javax.ws.rs.core.SecurityContext` class. For example:

```
public class ExampleResource {
 @Inject
 JsonWebToken jwt;

 public SecurityContext debugSecurityContext(@Context SecurityContext ctx) {
 return ctx;
 }
}
```

The Quarkus OpenID extension contains a number of other properties. To get a comprehensive list of the extension properties, search for properties starting with the `quarkus.oidc` prefix.

## Enabling Authentication in Quarkus

Quarkus provides a number of authentication extensions, which enable developers to implement authentication. There are two kinds of extensions, that combine to generate the authentication process:

- `Identity providers` validate the identity associated by the request. This identity is provided by an `authentication mechanism`. After the validation, Identity providers generate a `SecurityIdentity` instance available for the application.

Some identity provider extensions are:

- The `quarkus-elytron-security-jdbc`, which validates identities performing JDBC queries from a database.
- The `quarkus-security-jpa` extension, which uses JPA to validate identities from a data source.
- The `quarkus-elytron-security-ldap` extension, uses an LDAP server to validate the identity and retrieve user roles and groups.
- The `quarkus-elytron-security-properties-file`, embeds static identities in the application configuration file.
- A `HttpAuthenticationMechanism` is responsible for extracting the authentication credentials from the requests and ensuring their validity. Note that some times authentication mechanisms might delegate the identity validation to another extension.

Quarkus provides two built-in authentication mechanisms: the `basic authentication`, which asks the request initiator for credentials and validates them against an `Identity Provider`, and the `form based authentication`, where credentials must be provided as form parameters. Enable those mechanisms by the `quarkus.http.auth.basic` and `quarkus.http.auth.form.enabled` configuration entries respectively.

Mutual TLS is another embedded form of authentication. It takes the identity from the client certificate and validates it cryptographically.

Some extensions implementing authentication mechanisms are:

- The `quarkus-smallrye-jwt` extracts the identity from a bearer JWT token included in the request. This extension can validate the token cryptographically and provide token introspection capabilities.
- The `quarkus-oidc` extension enables the OpenID Connect authentication workflow. It supports Bearer Token for service applications and Authorization Token flow for web applications. This extension relies on the `quarkus-smallrye-jwt` extension to handle identity tokens.
- The `quarkus-elytron-security-oauth2` extension uses opaque bearer tokens to validate the identity. This extension is still in technology preview status, therefore it is using other token-based approaches such as `quarkus-smallrye-jwt` is recommended.

Quarkus uses the `HttpAuthenticationMechanism` interface for authenticating a request. Consequently, you can implement custom authorization workflows and protocols for which Quarkus does not provide any extension.

Quarkus extensions enable you to configure authentication settings by using extension properties. Extension properties often contain the location of the authentication server, cryptographic keys for identity verification, or endpoints that require authentication. For example, you can configure the `quarkus-smallrye-jwt` extension with the location and issuer of the public key used to validate the token:

```
mp.jwt.verify.publickey.location=META-INF/resources/publicKey.pem
mp.jwt.verify.issuer=https://example.com/issuer
```

## Authorizing Requests in Quarkus

Quarkus uses RBAC, or Role-Based Access Control, to implement authorization restrictions. With RBAC, each identity can be associated with one or more roles. Developers can specify which roles are authorized to use a specific resource.

When a resource contains authorization mapping, Quarkus checks that at least one of the roles associated with the request's identity is authorized to use the resource. If the identity does not contain at least one role specified in the authorization mapping, the call is unauthorized and Quarkus denies the request.

## Implementing Authorization Through Annotations

Developers can define authorization using the standard `javax.annotation.security` enterprise Java annotations. For example, the `@io.quarkus.security.Authenticated` annotation enforces authentication at a class or method level. Quarkus can also inject the `javax.ws.rs.core.SecurityContext` bean into the class or as method properties for the developer to access the underlying security information directly.

```

@Path("/")
public class SubjectExposingResource {

 @GET @Path("authenticated")
 @Authenticated ①
 public String getAuthenticated(@Context SecurityContext sec) {②
 ...output omitted...
 }

 @GET @Path("secured")
 @RolesAllowed("admin") ③
 public String getAdminsOnly() {
 ...output omitted...
 }

 @GET @Path("unsecured")
 @PermitAll ④
 public String getUnsecured() {
 ...output omitted...
 }
}

```

- ① The resource is available for all authenticated users regardless of their roles.
- ② Quarkus can inject the security context into methods or beans.
- ③ The resource is restricted to authenticated users with the `admin` role.
- ④ Both authenticated and unauthenticated users can access this resource.

## Implementing Authorization Through Configuration

Alternatively, developers can specify the authorization settings in the `application.properties` property file. This is useful when you need to change authorization settings at runtime, without recompiling the application.

First, create the policy entry, which defines a set of roles with granted permissions. Use entries of the format `quarkus.http.auth.policy.<POLICY_NAME>.roles-allowed` to create the policy, and assign a comma-separated list of roles.

```
quarkus.http.auth.policy.policy1.roles-allowed=user,admin
```

Quarkus provides two built-in policies: `permit` and `deny`. Those are equivalent to the `PermitAll` and `DenyAll` annotations.

Assign a permission set to the policy. This allows you to define the different paths and HTTP methods covered by the policy.

```
quarkus.http.auth.permission.permission1.policy=policy1
quarkus.http.auth.permission.permission1.paths=/secured/*
quarkus.http.auth.permission.permission1.methods=GET,POST

quarkus.http.auth.permission.permission2.policy=permit
quarkus.http.auth.permission.permission2.paths=/public/*
```

Defining authorization settings in the `application.properties` file enables you to make the following changes at runtime:

- Enable or disable a specific property.
- Switch the configuration profile, which can have different authorization settings.

Specifying authorization settings in a property file can make the application difficult to understand, but it provides you with more flexibility.



### Note

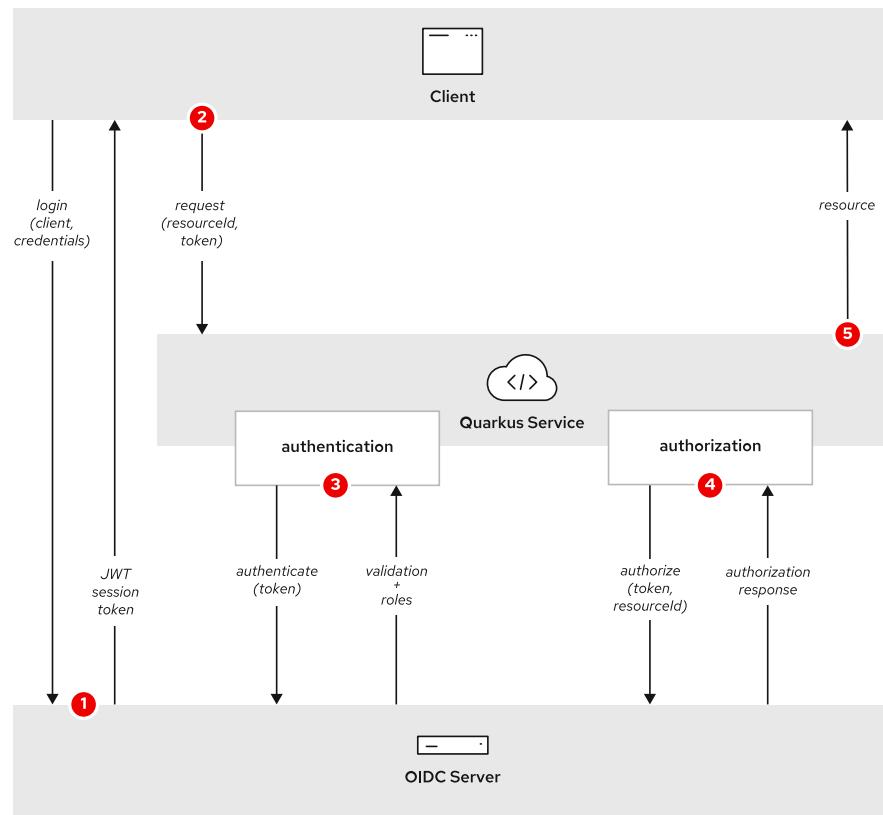
If you declare authentication in both extension properties and annotations, Quarkus verifies property-based checks before annotation-based checks. Both checks must pass for a request to succeed.

Other configuration properties might also influence the authorization behaviour of your application.

For example, set the `quarkus.security.jaxrs.deny-unannotated-endpoints` to `true` to forbid requests to endpoints that are not annotated with security constraints.

## Delegating Authorization to an Authorization Server

Developers can choose to delegate authorization to an external identity server instead of manually creating a mapping between roles and resources. For example, Red Hat Single Sign On (SSO), enables you to associate users and groups with resources. Authorization checks are performed by the Red Hat SSO server.



- ① The client provides the credential to the server and obtains the session token.
- ② The client requests the needed resource, providing the session token.
- ③ The service forwards the token to the OIDC server that performs the authentication and, if positive, provides the roles associated to the identity.
- ④ The service forwards the token and the resource ID to the OIDC server. The server authorizes the access and returns the appropriate response.
- ⑤ If the token is authentic and the access authorized then the service returns the requested resource.

Delegating authorization settings to an external server simplifies the application code, because the developers can focus on business logic. However, this creates a tight coupling of a microservice to a particular authorization server.

The `quarkus-keycloak-authorization` extension extends `quarkus-oidc` delegating resource permissions to a Keycloak server.



## References

### **Quarkus - Security Architecture and Guides**

<https://quarkus.io/version/1.11/guides/security>

### **Web authentication with HttpAuthenticationMechanism**

<https://developer.ibm.com/languages/java/tutorials/j-javaee8-security-api-2/>

### **javax.annotation.security (Java™ EE 8 Specification APIs)**

<https://javaee.github.io/javaee-spec/javadocs/javax/annotation/security/package-summary.html>

## ► Guided Exercise

# Authenticating and Authorizing Requests

In this exercise, you will secure REST endpoints of the `expense-service` application. You will:

- Enforce authentication for the `expense-service` application.
- Enforce role-based authorization for the `expense-service` application.
- Adopt deny-by-default settings for endpoints without authorization or authentication enforcement.

The endpoints implement the following authorization settings:

Endpoint name	Allowed groups
GET /expenses	read
PUT /expenses	modify
POST /expenses	modify
DELETE /expenses/{uuid}	delete

Any other endpoints will be inaccessible.

A containerized Keycloak instance with the following pre-configured as follows:

- A realm named `quarkus`.
- A client named `backend-service`, with a credential secret `secret`.

The `quarkus` realm includes the following credentials:

Name (UPN)	Password	Roles
jane	jane	read, modify
john	john	read, modify, delete
debug	debug	none

## Outcomes

You should be able to:

- Authenticate your API requests with a JWT token using the OIDC protocol.
- Secure your API endpoints by using the `@Authenticated` annotation.
- Enforce authorization of your API endpoints using the `@RolesAllowed` annotation.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab secure-jwt start
```

## Instructions

- 1. Start the Keycloak OIDC server.

Open a new terminal window, and execute the `start-keycloak.sh` helper script provided in the `/home/student/D0378/labs/secure-jwt` folder.

```
[student@workstation ~]$ cd /home/student/D0378/labs/secure-jwt
[student@workstation secure-jwt]$./start-keycloak.sh

Starting the 'KeyCloak' container
Press Ctrl+C to terminate

Error: no container with name or ID "kc" found: no such container
Added 'USERNAME' to '/opt/jboss/keycloak/standalone/configuration/keycloak-add-
user.json', restart server to load user
-b 0.0.0.0
...output omitted...
09:10:39,418 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin
console listening on http://127.0.0.1:9990
09:10:39,418 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: Keycloak
10.0.1 (WildFly Core 11.1.1.Final) started in 30667ms - Started 689 of 994
services (708 services are lazy, passive or on-demand)
```

Leave this terminal window open for the duration of the exercise.

- 2. Explore and start the application.

2.1. In a new terminal window, change into the `secure-jwt` lab directory:

```
[student@workstation ~]$ cd \
/home/student/D0378/labs/secure-jwt/expense-service
```

2.2. Open the application in VSCode:

```
[student@workstation expense-service]$ codium .
```

2.3. Explore the `src/main/java/org/acme/rest/pki` package.

The `PkiResource` is a helper class displaying security context for incoming requests. Note that the `/pki` endpoint does not require authentication nor user roles.

2.4. Explore the `src/main/java/org/acme/rest/json` package.

This package includes the sources for the expenses service.

2.5. Use the `mvn` command to start Quarkus in development mode.

```
[student@workstation expense-service]$ mvn clean quarkus:dev
...output omitted...
INFO [io.quarkus] (Quarkus Main Thread) expense-restful-service 1.0-SNAPSHOT on
JVM (powered by Quarkus 1.7.5.Final-redhat-00007) started in 1.618s. Listening
on: http://0.0.0.0:8080
INFO [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding
activated.
INFO [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, resteasy,
resteasy-jsonb, smallrye-openapi, swagger-ui]
```

- 3. Obtain the JWT tokens for all users and store them in local environment variables for later use:

Save the token values into `jane`, `john`, and `debug` shell variables.

- 3.1. In a new terminal window, change into `/home/student/D0378/labs/secure-jwt`:

```
[student@workstation secure-jwt]$ cd /home/student/D0378/labs/secure-jwt
```

- 3.2. Extract the token and review roles for the `jane` user.

Use the provided script `keycloak-token.sh` to simplify the extraction.

```
[student@workstation secure-jwt]$ jane=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service jane jane secret)
Using Keycloak: http://localhost:8081/auth/realms/quarkus/protocol/openid-
connect/token
realm: quarkus
client-id: backend-service
username: jane
Token successfully retrieved. Showing token roles:
{
 "roles": [
 "modify",
 "read",
 "offline_access",
 "uma_authorization"
]
}
```

- 3.3. Extract the token and review roles for the `john` user.

```
[student@workstation secure-jwt]$ john=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service john john secret)
Using Keycloak: http://localhost:8081/auth/realms/quarkus/protocol/openid-
connect/token
realm: quarkus
client-id: backend-service
username: john
Token successfully retrieved. Showing token roles:
{
 "roles": [
 "modify",
 "read",
 "offline_access",
 "uma_authorization"
]
}
```

```

 "modify",
 "read",
 "offline_access",
 "uma_authorization",
 "delete"
]
}

```

3.4. Extract the token and review roles for the debug user.

```

[student@workstation secure-jwt]$ debug=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service debug debug secret)
Using Keycloak: http://localhost:8081/auth/realms/quarkus/protocol/openid-
connect/token
realm: quarkus
client-id: backend-service
username: john
Token successfully retrieved. Showing token roles:
{
 "roles": [
 "offline_access",
 "uma_authorization"
]
}

```

► 4. Authenticate your user against the expense-service application.

4.1. Use the pki endpoint to review security context when no JWT token is present:

```

[student@workstation secure-jwt]$ curl -s localhost:8080/pki | jq
{
 "secure": false
}

```

4.2. Execute the method with a JWT token. Pass the token by using the Authorization: Bearer *token* header:

```

[student@workstation secure-jwt]$ curl -v localhost:8080/pki \
-H 'Content-Type: application/json' -H "Authorization: Bearer $jane" | jq
...output omitted...
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Length: 48
< Content-Type: application/json
<
* Connection #0 to host localhost left intact
{
 "authenticationScheme": "Bearer",
 "secure": false
}%

```

You sent a signed JWT token, but the Quarkus application cannot verify and validate the cryptographic origin of the token.

► 5. Connect the Quarkus application to the OIDC Keycloak server.

5.1. Add the quarkus-oidc extension to the Quarkus application.

```
[student@workstation secure-jwt]$ cd expense-service
[student@workstation expense-service]$ mvn quarkus:add-extension \
-Dextension=quarkus-oidc
...output omitted...
Extension io.quarkus:quarkus-oidc has been installed
...output omitted...
[student@workstation expense-service]$ cd ..
[student@workstation secure-jwt]$
```

5.2. In VSCode, open the `src/main/resources/application.properties` file.  
Set the following properties to the values provided at the beginning of the exercise:

```
...contents omitted...
quarkus.oidc.auth-server-url=http://localhost:8081/auth/realm/quarkus
quarkus.oidc.client-id=backend-service
quarkus.oidc.credentials.secret=secret
```

Save the `application.properties` file.

5.3. Re-execute your call:

```
[student@workstation secure-jwt]$ curl -s localhost:8080/pki \
-H 'Content-Type: application/json' -H "Authorization: Bearer $jane" | jq
{
 ...contents omitted...
 "userPrincipal": {
 ...contents omitted...
 "claims": {
 ... contents omitted...
 "issuer": "http://localhost:8081/auth/realm/quarkus",
 "jwtId": "464c4cc6-7db4-4643-a6b7-677c0c37d300",
 "rawJson": "{\"exp\":1629714223,\"iat\":1629713923,\"jti\":\"464c4cc6-7db4-4643-a6b7-677c0c37d300\",\"iss\":\"http://localhost:8081/auth/realm/quarkus\",\"aud\":\"account\",\"sub\":\"ff4bb675-5200-44d7-9250-6c90d22db3c5\",\"typ\":\"Bearer\",\"azp\":\"backend-service\",\"session_state\":\"6848c7f5-ffa6-4042-ad60-c1802ae33dae\",\"acr\":\"1\",\"realm_access\":{\"roles\":\"modify\",\"read\",\"offline_access\"},\"uma_authorization\"},\"resource_access\":{\"account\":{\"roles\":[\"manage-account\",\"manage-account-links\",\"view-profile\"]}},\"scope\":\"email profile\",\"email_verified\":false,\"preferred_username\":\"jane\"}",
 "subject": "ff4bb675-5200-44d7-9250-6c90d22db3c5"
 },
 "credential": {
 "token": "eyJh...00Bg",
 ...contents omitted...
 },
 "name": "jane"
 }
}
```

**Important**

JWT token can expire before running the latest request. Use the `keycloak-token.sh` again to obtain a fresh token when needed.

- 6. Enforce that all the `/expenses/*` endpoints require authentication.
- 6.1. In VSCode, open the `src/main/java/org/acme/rest/json/ExpenseResource` class. Then, add the `@Authenticated` annotation at the class level:

```
@Path("/expenses")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Authenticated
public class ExpenseResource {
```

- 6.2. Save the `ExpenseResource` class.
- 6.3. Verify that unauthenticated calls fail. Execute the `GET /expenses` endpoint without a JWT token:

```
[student@workstation secure-jwt]$ curl -v localhost:8080/expenses
...output omitted...
< HTTP/1.1 401 Unauthorized
< Content-Length: 0
...output omitted...
```

- 6.4. Verify that authenticated calls succeed. Execute the `GET /expenses` with a JWT token:

```
[student@workstation secure-jwt]$ curl -s localhost:8080/expenses \
-H "Authorization: Bearer $jane" | jq
[
 {
 "amount": 150.5,
 "creationDate": "2020-11-19T10:00:11.147615",
 "name": "Groceries",
 "paymentMethod": "CASH",
 "uuid": "23ccdef9-4f1c-4af4-9174-1e40fbdb776f"
 },
 ...output omitted...
```

- 7. Enforce that only users with the `read` group can execute the `GET /expenses` endpoint.
- 7.1. In the `src/main/java/org/acme/rest/json/ExpenseResource` class, annotate the `list()` method with the `@RolesAllowed("read")` annotation:

```

@GET
@RolesAllowed("read")
public Set<Expense> list() {
 return expenseService.list();
}

```

7.2. Save the ExpenseResource class.

7.3. Verify that the user `jane` can execute the `GET /expenses` endpoint:

```

[student@workstation secure-jwt]$ curl -s localhost:8080/expenses \
-H "Authorization: Bearer $jane" | jq
[
 {
 "amount": 150.5,
 "creationDate": "2020-11-19T10:00:11.147615",
 "name": "Groceries",
 ...
 }
]

```

7.4. Verify that the `debug` user is not authorized to execute the `GET /expenses` endpoint:

```

[student@workstation secure-jwt]$ curl -v localhost:8080/expenses \
-H "Authorization: Bearer $debug"
...
< HTTP/1.1 403 Forbidden
< Content-Length: 0
...

```

► 8. Enforce that only users with the `modify` group can execute the `POST /expenses` and `PUT /expenses` endpoints.

8.1. In the `src/main/java/org/acme/rest/json/ExpenseResource` class, annotate the `create` and `update` methods with the `@RolesAllowed("modify")` annotation:

```

@POST
@RolesAllowed("modify")
public Expense create(Expense expense) {
 return expenseService.create(expense);
}
//...contents omitted...

@PUT
@RolesAllowed("modify")
public void update(Expense expense) {
 expenseService.update(expense);
}

```

8.2. Optionally, verify that the user `jane` can execute both endpoints, and the user `debug` is unauthorized to execute the endpoints. For example:

```
[student@workstation secure-jwt]$ curl -v -X POST -d @payload.json \
-v localhost:8080/expenses \
-H "Authorization: Bearer $debug" -H "Content-Type: application/json"
...output omitted...
< HTTP/1.1 403 Forbidden
< Content-Length: 0

[student@workstation secure-jwt]$ curl -s -X POST -d @payload.json \
-v localhost:8080/expenses \
-H "Authorization: Bearer $jane" -H "Content-Type: application/json" | jq
{
 "amount": 15,
 "creationDate": "2020-11-23T09:53:42.289889",
 "name": "Food",
 "paymentMethod": "CASH",
 "uuid": "d494efd8-fb2b-4d3f-8846-dc6767abab81"
}
```

- ▶ 9. Enforce that only users with the `delete` group can execute the `DELETE /expenses` endpoint.
- 9.1. In the `src/main/java/org/acme/rest/json/ExpenseResource` class, annotate the `delete` method with the `@RolesAllowed("delete")` annotation:

```
@DELETE
@Path("{uuid}")
@RolesAllowed("delete")
public Set<Expense> delete(@PathParam("uuid") UUID uuid) {
...method omitted...
```

- 9.2. Optionally, verify that only the user `jane` can execute the endpoint. List the expenses and use the `uuid` of the `Groceries` expense as the parameter for the `delete` endpoint, for example:

```
[student@workstation secure-jwt]$ curl -v localhost:8080/expenses \
-H "Authorization: Bearer $jane" | jq
...output omitted...
[
 {
 "amount": 25,
 "creationDate": "2022-03-14T13:11:12.143846",
 "name": "Civilization VI",
 "paymentMethod": "DEBIT_CARD",
 "uuid": "bfbc5e2a-d6f5-4d6c-aa66-23ecaea01cd2"
 },
 {
 "amount": 150.5,
 "creationDate": "2022-03-14T13:11:12.143741",
 "name": "Groceries",
 "paymentMethod": "CASH",
 "uuid": "ded3b4a8-8a60-4a8d-94da-ac6efd703f7f"
}
```

```
[student@workstation secure-jwt]$ curl -v -X DELETE \
localhost:8080/expenses/ded3b4a8-8a60-4a8d-94da-ac6efd703f7f \
-H "Authorization: Bearer $jane"
...output omitted...
< HTTP/1.1 403 Forbidden
< Content-Length: 0
...output omitted...

[student@workstation secure-jwt]$ curl -X DELETE \
localhost:8080/expenses/ded3b4a8-8a60-4a8d-94da-ac6efd703f7f \
-H "Authorization: Bearer $john" | jq
...output omitted...
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Length: 593
< Content-Type: application/json
<
* Connection #0 to host localhost left intact
[
{
 "amount": 25,
 "creationDate": "2021-08-23T12:40:24.13547",
 "name": "Civilization VI",
 "paymentMethod": "DEBIT_CARD",
...output omitted...
```

- ▶ 10. Enforce deny-by-default behavior on endpoints that do not specify authorization permissions.

- 10.1. In VSCode, open the `src/main/resources/application.properties` file. Then, set the `quarkus.security.jaxrs.deny-unannotated-endpoints` property to `true`.

```
...content omitted...
quarkus.security.jaxrs.deny-unannotated-endpoints=true
...content omitted...
```

- 10.2. Verify that you cannot use the `/pki` endpoint because it does not specify authorization permissions.

```
[student@workstation secure-jwt]$ curl -v localhost:8080/pki
...output omitted...
< HTTP/1.1 401 Unauthorized
< www-authenticate: Bearer {token}
[student@workstation secure-jwt]$ curl -v localhost:8080/pki \
-H "Authorization: Bearer $debug"
...output omitted...
* Mark bundle as not supporting multiuse
< HTTP/1.1 403 Forbidden
< Content-Length: 0
```

- ▶ **11.** Close all terminal windows to terminate the Quarkus application. Then, close the VSCode IDE.

You can see the finished project at `/home/student/D0378/solution/secure-jwt/`.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab secure-jwt finish
```

This concludes the guided exercise.

## ► Lab

# Securing Microservices

In this lab, you will enable mutual TLS communication between two microservices in the `quarkus-conference` application. You will also add OIDC authentication and RBAC authorization, to protect the access to the REST endpoints in the application.

## Outcomes

You should be able to:

- Enable TLS in a Quarkus microservice by using a keystore and configuring it in the `application.properties` file.
- Enable mutual TLS by using keystores and trust stores and configuring them in the `application.properties` file.
- Protect API endpoints in a microservice from unauthenticated requests.
- Authenticate requests using JWT tokens provided by a containerized OIDC server (Keycloak).
- Protect API endpoints by creating and enforcing Role Based Access Control (RBAC).

## Before You Begin

On the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command downloads the code of the `quarkus-conference` application into the `workstation` and starts containerized dependencies of the application. It also provides scripts to facilitate the creation of certificates, as well as testing and grading of the lab.

```
[student@workstation ~]$ lab secure-review start
```

## Instructions

1. From the `~/D0378/labs/secure-review/` directory, analyze and execute the `prepare-certificates.sh` script, which creates keystores, certificates, and trust stores for each of the microservices in `~/D0378/labs/secure-review/quarkus-conference/`.

The following tables show all information related to the objects generated by the `prepare-certificates.sh` script:

**Keys pairs (public and private)**

<b>key alias</b>	<b>keystore</b>	<b>keystore/key password</b>
microservice-session	microservice-session/microservice-session.keystore.p12	redhat/redhat
microservice-speaker	microservice-speaker/microservice-speaker.keystore.p12	redhat/redhat

**Certificates (public keys)**

<b>key alias</b>	<b>certificate file</b>
microservice-session	microservice-session/microservice-session.crt
microservice-speaker	microservice-speaker/microservice-speaker.crt

**Trust stores**

<b>Trust store file</b>	<b>Trusted certificates</b>
microservice-speaker/microservice-speaker.truststore.p12	microservice-session
microservice-session/microservice-session.truststore.p12	microservice-speaker

2. Open ~/D0378/labs/secure-review/quarkus-conference with VSCode. Modify the application.properties file of each of the different Quarkus microservices in the project to enable HTTPS, including keystores and, whenever relevant, trust stores for mutual TLS.  
To avoid port conflicts, use the port 8444 for the HTTPS configuration of the microservice-speaker service and port 8445 for the microservice-session service.
3. The microservice-speaker and microservice-session microservices communicate with each other. Configure their application.properties files in order to enable mTLS in their communication, by using the trust store files. Enforce HTTPS for requests performed from one microservice to another. Make sure the microservices can correctly call each other.

4. Change the `microservice-session` microservice so that it requires authenticated requests.

To generate and authenticate session JWT tokens you have available a containerized Keycloak instance with the following details:

Server and realm URL	Client ID	Client Secret
<code>http://localhost:8882/auth/realms/quarkus</code>	backend-service	secret

The pre-configured realm contains the following users and roles:

User	Password	Roles
user1	user1	read, modify, delete
user2	user2	read

5. Modify the endpoints within `microservice-session` to enforce that:

- GET requests require the `read` role
- POST and PUT requests require the `modify` role
- DELETE requests require the `delete` role

6. In a new terminal window, start the microservices by running the `~/D0378/labs/secure-review/start-services.sh` script. Keep this terminal open and services working until the end of the exercise.

Use the provided `keycloak-token.sh` script to obtain the JWT tokens from the Keycloak instance for `user1` and `user2`. The syntax for the `keycloak-token.sh` is as follows:

```
[student@workstation secure-review]$ user_token=$(./keycloak-token.sh \
keycloak-host realm client-name username password client-password)
```

Validate that you can call all the different `microservice-session` endpoints, with the token obtained for the `user1` user. Also, ensure you can only call the GET endpoints by using the token obtained for the `user2` user.

After the validations, stop the services by pressing `Enter` in the terminal where it is running.

## Evaluation

Reset the database contents by pressing `Enter` in the terminal running the Quarkus services to terminate them and then start the services again with the `start-services.sh` script.

Keep the Quarkus services up and running in `dev` mode.

Grade your work by running the `lab secure-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation secure-review]$ lab secure-review grade
```

## Finish

Press `Enter` in the terminal running the Quarkus services to terminate them. Close the terminal and the VSCode window.

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation secure-review]$ lab secure-review finish
```

This concludes the lab.

## ► Solution

# Securing Microservices

In this lab, you will enable mutual TLS communication between two microservices in the `quarkus-conference` application. You will also add OIDC authentication and RBAC authorization, to protect the access to the REST endpoints in the application.

## Outcomes

You should be able to:

- Enable TLS in a Quarkus microservice by using a keystore and configuring it in the `application.properties` file.
- Enable mutual TLS by using keystores and trust stores and configuring them in the `application.properties` file.
- Protect API endpoints in a microservice from unauthenticated requests.
- Authenticate requests using JWT tokens provided by a containerized OIDC server (Keycloak).
- Protect API endpoints by creating and enforcing Role Based Access Control (RBAC).

## Before You Begin

On the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command downloads the code of the `quarkus-conference` application into the `workstation` and starts containerized dependencies of the application. It also provides scripts to facilitate the creation of certificates, as well as testing and grading of the lab.

```
[student@workstation ~]$ lab secure-review start
```

## Instructions

1. From the `~/D0378/labs/secure-review/` directory, analyze and execute the `prepare-certificates.sh` script, which creates keystores, certificates, and trust stores for each of the microservices in `~/D0378/labs/secure-review/quarkus-conference/`.

The following tables show all information related to the objects generated by the `prepare-certificates.sh` script:

**Keys pairs (public and private)**

<b>key alias</b>	<b>keystore</b>	<b>keystore/key password</b>
microservice-session	microservice-session/microservice-session.keystore.p12	redhat/redhat
microservice-speaker	microservice-speaker/microservice-speaker.keystore.p12	redhat/redhat

**Certificates (public keys)**

<b>key alias</b>	<b>certificate file</b>
microservice-session	microservice-session/microservice-session.crt
microservice-speaker	microservice-speaker/microservice-speaker.crt

**Trust stores**

<b>Trust store file</b>	<b>Trusted certificates</b>
microservice-speaker/microservice-speaker.truststore.p12	microservice-session
microservice-session/microservice-session.truststore.p12	microservice-speaker

- 1.1. Enter the ~/D0378/labs/secure-review/ directory. Read the `prepare-certificates.sh` script to understand what files it is generating. Read the `prepare-certificates.sh` script to understand what files it is generating.

```
[student@workstation ~]$ cd ~/D0378/labs/secure-review/
[student@workstation secure-review]$ cat prepare-certificates.sh
```

- 1.2. Run the script to generate the files required to enable TLS and mTLS for your microservices.

```
[student@workstation secure-review]$./prepare-certificates.sh
~/D0378/labs/secure-review/quarkus-conference ~/D0378/labs/secure-review
Generating keystore file for microservice-session...
Generating keystore file for microservice-speaker...
Exporting certificate for microservice-session...
Certificate stored in file <microservice-session/microservice-session.crt>
Exporting certificate for microservice-speaker...
Certificate stored in file <microservice-speaker/microservice-speaker.crt>
Importing certificate for microservice-session into microservice-speaker trust
store...
Certificate was added to keystore
```

```
Importing certificate for microservice-speaker into microservice-session trust
store...
Certificate was added to keystore
~/D0378/labs/secure-review
```

2. Open ~/D0378/labs/secure-review/quarkus-conference with VSCode. Modify the application.properties file of each of the different Quarkus microservices in the project to enable HTTPS, including keystores and, whenever relevant, trust stores for mutual TLS.

To avoid port conflicts, use the port 8444 for the HTTPS configuration of the microservice-speaker service and port 8445 for the microservice-session service.

- 2.1. Open the quarkus-conference project.

```
[student@workstation secure-review]$ codium \
~/D0378/labs/secure-review/quarkus-conference
```

**Note**

If a window opens asking if you trust the authors of the files in this folder then click **Yes, I trust the authors.**

- 2.2. Open the microservice-speaker/src/main/resources/application.properties file. Edit the contents to configure HTTPS and add the keystore. Disable HTTP requests that do not use HTTPS.

```
...output omitted...
Configure HTTPS with keystore.
quarkus.http.ssl-port=8444
quarkus.http.ssl.certificate.key-store-file=../microservice-speaker.keystore.p12
quarkus.http.ssl.certificate.key-store-password=redhat
quarkus.http.insecure-requests=disabled
...output omitted...
```

- 2.3. Open the microservice-session/src/main/resources/application.properties file. Edit the contents to configure HTTPS and add the keystore.

```
...output omitted...
Configure HTTPS with keystore.
quarkus.http.ssl-port=8445
quarkus.http.ssl.certificate.key-store-file=../microservice-session.keystore.p12
quarkus.http.ssl.certificate.key-store-password=redhat
...output omitted...
```

3. The microservice-speaker and microservice-session microservices communicate with each other. Configure their application.properties files in order to enable mTLS in their communication, by using the trust store files. Enforce HTTPS for requests performed from one microservice to another. Make sure the microservices can correctly call each other.

- 3.1. Open the microservice-speaker/src/main/resources/application.properties file. Edit the contents to add the trust store and force authentication on the HTTP SSL Client.

```
...output omitted...
Configure HTTPS with keystore.
quarkus.http.ssl-port=8444
quarkus.http.ssl.certificate.key-store-file=../microservice-speaker.keystore.p12
quarkus.http.ssl.certificate.key-store-password=redhat
quarkus.http.insecure-requests=disabled
quarkus.http.ssl.certificate.trust-store-file=../microservice-
speaker.truststore.p12
quarkus.http.ssl.certificate.trust-store-password=redhat
quarkus.http.ssl.client-auth=required
...output omitted...
```

- 3.2. Open the `microservice-session/src/main/resources/application.properties` file. Edit the contents to add the trust store.

```
...output omitted...
Configure HTTPS with keystore.
quarkus.http.ssl-port=8445
quarkus.http.ssl.certificate.key-store-file=../microservice-session.keystore.p12
quarkus.http.ssl.certificate.key-store-password=redhat
quarkus.http.ssl.certificate.trust-store-file=../microservice-
session.truststore.p12
quarkus.http.ssl.certificate.trust-store-password=redhat
...output omitted...
```

- 3.3. Modify the SpeakerService HTTP Client configuration in `microservice-session` to use `https` instead of `http`. Modify the client to also use the trust store and key stores for consuming the SpeakerService via mTLS.

```
...output omitted...
Intraservice comms
org.acme.conference.session.SpeakerService/mp-rest/url=https://localhost:8444
org.acme.conference.session.SpeakerService/mp-rest/trustStore=../microservice-
session.truststore.p12
org.acme.conference.session.SpeakerService/mp-rest/trustStorePassword=redhat
org.acme.conference.session.SpeakerService/mp-rest/keyStore=../microservice-
session.keystore.p12
org.acme.conference.session.SpeakerService/mp-rest/keyStorePassword=redhat
...output omitted...
```

4. Change the `microservice-session` microservice so that it requires authenticated requests.

To generate and authenticate session JWT tokens you have available a containerized Keycloak instance with the following details:

Server and realm URL	Client ID	Client Secret
http://localhost:8882/auth/realms/quarkus	backend-service	secret

The pre-configured realm contains the following users and roles:

User	Password	Roles
user1	user1	read, modify, delete
user2	user2	read

- 4.1. Add the quarkus-oidc extension to the microservice-session project by modifying its pom.xml file.

```
...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-oidc</artifactId>
</dependency>
...output omitted...
```

- 4.2. Configure the OIDC extension to communicate with the Keycloak server. Edit the application.properties file to add the following properties:

```
...output omitted...
Configure Keycloak client
quarkus.oidc.auth-server-url=http://localhost:8882/auth/realm/quarkus
quarkus.oidc.client-id=backend-service
quarkus.oidc.credentials.secret=secret
...output omitted...
```

- 4.3. Add the @Authenticated annotation to the org.acme.conference.session.SessionResource class.

```
package org.acme.conference.session;

import io.quarkus.security.Authenticated;
...output omitted...
@Path("sessions")
@ApplicationScoped
@Authenticated
public class SessionResource {
 ...output omitted...
```

5. Modify the endpoints within microservice-session to enforce that:

- GET requests require the read role
- POST and PUT requests require the modify role
- DELETE requests require the delete role

- 5.1. Modify all the endpoints with the corresponding annotation in the SessionResource class.

```
...output omitted...
import javax.annotation.security.RolesAllowed;
...output omitted...
@GET
```

```

@RolesAllowed("read")
public Collection<Session> allSessions() throws Exception {
 ...output omitted...
 @PUT
 @RolesAllowed("modify")
 public Session createSession(final Session session) {
 ...output omitted...
 @GET
 @RolesAllowed("read")
 public Response retrieveSession(@PathParam("sessionId") final String sessionId)
 {
 ...output omitted...
 @PUT
 @RolesAllowed("modify")
 public Response updateSession(@PathParam("sessionId") final String sessionId,
 final Session session) {
 ...output omitted...
 @DELETE
 @RolesAllowed("delete")
 public Response deleteSession(@PathParam("sessionId") final String sessionId) {
 ...output omitted...
 @GET
 @RolesAllowed("read")
 public Response sessionSpeakers(@PathParam("sessionId") final String sessionId)
 {
 ...output omitted...
 @PUT
 @RolesAllowed("modify")
 public Response addSessionSpeaker(@PathParam("sessionId") final String sessionId
 ...output omitted...
 @DELETE
 @RolesAllowed("delete")
 public Response removeSessionSpeaker(@PathParam("sessionId") final String
 sessionId,
 ...output omitted...

```

6. In a new terminal window, start the microservices by running the ~/D0378/labs/secure-review/start-services.sh script. Keep this terminal open and services working until the end of the exercise.

Use the provided keycloak-token.sh script to obtain the JWT tokens from the Keycloak instance for user1 and user2. The syntax for the keycloak-token.sh is as follows:

```
[student@workstation secure-review]$ user_token=$(./keycloak-token.sh \
keycloak-host realm client-name username password client-password)
```

Validate that you can call all the different microservice-session endpoints, with the token obtained for the user1 user. Also, ensure you can only call the GET endpoints by using the token obtained for the user2 user.

After the validations, stop the services by pressing Enter in the terminal where it is running.

- 6.1. Start the services by executing the script.

```
[student@workstation ~]$ cd ~/D0378/labs/secure-review/
[student@workstation secure-review]$./start-services.sh
...output omitted...
```

Do not close this terminal nor terminate the services.

- 6.2. In a new terminal, obtain the JWT token for both user1 and user2.

```
[student@workstation ~]$ cd ~/D0378/labs/secure-review/
[student@workstation secure-review]$ user1_token=$(./keycloak-token.sh \
localhost:8882 quarkus backend-service user1 user1 secret)
Using Keycloak: http://localhost:8882/auth/realms/quarkus/protocol/openid-
connect/token
realm: quarkus
client-id: backend-service
username: user1
Token successfully retrieved. Showing token roles:
{
 "roles": [
 "modify",
 "read",
 "offline_access",
 "uma_authorization",
 "delete"
]
}
[student@workstation secure-review]$ user2_token=$(./keycloak-token.sh \
localhost:8882 quarkus backend-service user2 user2 secret)
Using Keycloak: http://localhost:8882/auth/realms/quarkus/protocol/openid-
connect/token
realm: quarkus
client-id: backend-service
username: user2
Token successfully retrieved. Showing token roles:
{
 "roles": [
 "read",
 "offline_access",
 "uma_authorization"
]
}
```

- 6.3. In the same terminal, call different endpoints of the services using the \$user1\_token variable as the JWT bearer token. Also, proceed to do the same but sending the \$user2\_token variable as the JWT token. Verify that the requests go through only when it corresponds to their permissions.

```
[student@workstation secure-review]$ curl -ks \
-H 'Content-Type: application/json' -H "Authorization: Bearer ${user1_token}" \
https://127.0.0.1:8445/sessions/s-1-1 | jq
{
 "id": "s-1-1",
 "schedule": 1,
```

```

"speakers": [
 {
 "id": 1,
 "name": "Emmanuel Bernard",
 "uuid": "s-1-1"
 }
]
}

[student@workstation secure-review]$ curl -ks \
-H 'Content-Type: application/json' -H "Authorization: Bearer ${user2_token}" \
https://127.0.0.1:8445/sessions/s-1-1 | jq
{
 "id": "s-1-1",
 "schedule": 1,
 "speakers": [
 {
 "id": 1,
 "name": "Emmanuel Bernard",
 "uuid": "s-1-1"
 }
]
}

[student@workstation secure-review]$ curl -ki -X DELETE \
-H "Authorization: Bearer ${user2_token}" \
https://127.0.0.1:8445/sessions/s-1-1
HTTP/2 403
content-length: 0

[student@workstation secure-review]$ curl -ki -X DELETE \
-H "Authorization: Bearer ${user1_token}" \
https://127.0.0.1:8445/sessions/s-1-1
HTTP/2 204

```

## Evaluation

Reset the database contents by pressing **Enter** in the terminal running the Quarkus services to terminate them and then start the services again with the `start-services.sh` script.

Keep the Quarkus services up and running in dev mode.

Grade your work by running the `lab secure-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation secure-review]$ lab secure-review grade
```

## Finish

Press **Enter** in the terminal running the Quarkus services to terminate them. Close the terminal and the VSCode window.

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation secure-review]$ lab secure-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- Quarkus core includes CORS, TLS, and Mutual TLS features, enabled by configuration entries and keystores.
- Quarkus addresses authentication by using different extensions, which provide `Identity Providers` and `HttpAuthenticationMechanisms`. Some common extensions are `quarkus-smallrye-jwt`, `quarkus-oidc`, or `quarkus-elytron-security-properties-file`.
- Quarkus uses RBAC to establish different authorization policies. To configure authorization, developers might use annotations, such as `@RolesAllowed`, configuration entries with the `quarkus.http.auth` prefix, or delegate to an authorization server.
- The `quarkus-oidc` extension implements the `OpenID Connect` specifications. This extension uses `JWT` tokens for authentication and `OAuth 2.0` for authorization.



## Chapter 9

# Monitoring Quarkus Microservices

### Goal

Monitor the operation of a microservice by using metrics and distributed tracing.

### Objectives

- Generate microservice metrics by using Micrometer.
- Generate request traces by using SmallRye MicroProfile OpenTracing.

### Sections

- Adding Metrics to a Microservice (and Guided Exercise)
- Tracing Requests Across Microservices (and Guided Exercise)

### Lab

Monitoring Microservices

# Adding Metrics to a Microservice

## Objectives

After completing this section, you should be able to generate microservice metrics by using Micrometer.

## Collecting Metrics Using Micrometer

Any applications that you deploy in production need a monitoring solution to track metrics about the application's performance and usage. The value of monitoring details about the performance of an application cannot be overstated when defects arise, or application performance is unexpectedly poor. It is important to standardize how your applications expose metrics data, and the content and format of that metrics data, to enable the most efficient and simple monitoring solutions.

The primary function of a health check is to provide a quick indication of the application's health. Platforms, such as OpenShift, that orchestrate the deployment of applications, use health information to restart the application if the health check fails. Metrics, on the other hand, determine the health of an application by pinpointing the underlying software issues, provide long-term trend data for capacity planning, and performs proactive discovery of platform-related problems (such as network latency). You can also use metrics for configuring OpenShift to decide when it can scale the application and run on more or less pods based on application usage.

To collect custom metrics from your application, Quarkus provides the `quarkus-micrometer` extension, which provides a generic API facade using code instrumentation. It provides support for many commonly used monitoring systems.

Quarkus integrates with the popular Prometheus monitoring system by using the `quarkus-micrometer-registry-prometheus` extension, which depends on the base `quarkus-micrometer` extension, to help you collect and analyze your application metrics.

Prometheus is an open source systems monitoring and alerting toolkit, which includes a time-series database for storing metrics. It provides a powerful web based user interface to query and analyze performance trends from the data it collects.

Metrics collected by Prometheus can be visualized using a tool like Grafana. Grafana is an open source graphical visualization tool used for creating operational dashboards for software systems.

The `quarkus-micrometer` extension enables you to collect the following types of metrics:

- Base runtime metrics of the application related to garbage collection, memory, threads and system load averages.
- Custom metrics from the application domain (that is, business logic related) using Prometheus metric types like counters, timers, gauges and summaries.

Custom metrics are specific to a particular application. An example of an application metric is how many times a specific method is invoked, or the current count of active users in the last fifteen minutes.

## Prometheus Metric Types

Prometheus supports four different types of metrics.

### Counter

A counter is a cumulative metric, which represents a single variable whose value can only increase, or be reset to zero on restart. For example, you can use a counter to represent the number of errors, requests served and tasks completed.

### Gauge

A gauge is a metric that represents a single numerical value, which can be incremented or decremented. For example, you can use a gauge to represent the number of processes, or the number of concurrent users.

### Histogram

A histogram samples values and counts them in configurable buckets. It also provides a sum of all values. Histograms track the number and the sum of the values , allowing you to calculate the average of the values.

For example, you can categorize response times in buckets (range of values) of 200 milliseconds with an upper limit of 1000 milliseconds. Prometheus collects the values and categorizes the values in each of the buckets.

### Summary

Similar to a histogram, a summary samples values. However a summary also calculates configurable values over a sliding time window.

Histograms buckets are categorized on the Prometheus server, as opposed to summaries, which are calculated on the client side (that is, the service exposing the metrics).

## Enabling Metrics Collection in Quarkus

The following are the steps to enable custom metrics for Quarkus applications.

- Include the `quarkus-micrometer-registry-prometheus` dependency in your application Maven `pom.xml` file for enabling metrics collection.

```
[user@host]$ mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-micrometer-registry-prometheus"
```

This command adds the following dependency to your `pom.xml`:

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
</dependency>
```

- Import the required metrics classes in your application from the `io.micrometer.core.instrument` package. This package has all the standard Prometheus metric types, which you can use to instrument your custom metrics.
- To register meters, you need an instance of the `io.micrometer.core.instrument.MeterRegistry` class, which is configured and maintained by the Micrometer extension. The `MeterRegistry` can be injected into your application as follows:

```
package com.mycompany.myapp;

import io.micrometer.core.instrument.MeterRegistry;

public class MyRESTResource {

 @Inject
 private MeterRegistry registry;
}

...output omitted...
```

4. Add relevant metrics for your application.

For example, Counters, which are used to keep track of the number of times a method is invoked, can be created by adding the @Counted annotation to methods as follows:

```
@Counted(value = "card_transactions", description = "count of credit card
transactions")
public void processCreditCardTransaction() {
...output omitted...
```

You can also create counters by using the injected MeterRegistry instance:

```
@Inject
private MeterRegistry registry;

// Create a counter
Counter hitCounter = registry.counter("no_of_hits","description","Number of Hits
on the website");

// increment the hitCounter in some method...
hitCounter.increment();
```

Timers are used for measuring time elapsed, for example, time taken to execute a method or operation. They can be created as follows using anonymous function blocks:

```
Timer mytimer = registry.timer("example.operation","description","Time taken to
perform an operation");
mytimer.record(() -> {
 // do some operation
})
```

You can also provide a Java Supplier as a lambda, or as a method reference for the timer to wrap as follows:

```
// register a timer that wraps a method call
private Supplier<String> myTimer = registry.timer("my.timer",
 "description", "Time elapsed to call a some function that returns
 a String")
 .wrap((Supplier<String>) someClass::someMethod);

// invoke the method
myTimer.get();
```

Gauges are used to track the count of some items in your application, for example, the number of messages in a queue, mails in a mailbox, objects in a collection, and more. They can be used as follows:

```
// initialize a gauge starting at zero
AtomicInteger gauge = registry.gauge("mySimpleGauge", new AtomicInteger(0));

// set the value for the gauge
gauge.set(20);
```

You can also monitor the size of lists and collections as follows:

```
List<Integer> messagesInQueue = new ArrayList<Integer>();
registry.gaugeCollectionSize("messages.in.queue", Tags.empty(), messagesInQueue);
```



### Note

Use gauges to monitor metrics with a fixed upper bound. Prefer a counter to monitor things like HTTP request counts, as they can grow unbounded for long running services.

You can add other types of metrics to your application. For the complete list of options, refer to the documentation link provided in the references.

By default, Quarkus makes the metrics available at the `/q/metrics` endpoint in plain text Prometheus data format. This endpoint combines your custom metrics along with other standard base metrics for the application, such as garbage collection information, heap size, and other system level metrics.

To enable the collection of metrics in JSON format, add the following line to the `src/main/resources/application.properties` file:

```
quarkus.micrometer.export.json.enabled=true
```



## References

### Collecting Metrics in Quarkus Applications

[https://access.redhat.com/documentation/en-us/red\\_hat\\_build\\_of\\_quarkus/1.11/html-single/collecting\\_metrics\\_in\\_your\\_quarkus\\_applications/index](https://access.redhat.com/documentation/en-us/red_hat_build_of_quarkus/1.11/html-single/collecting_metrics_in_your_quarkus_applications/index)

### Types of Metrics in Prometheus

[https://prometheus.io/docs/concepts/metric\\_types/](https://prometheus.io/docs/concepts/metric_types/)

### Grafana

<https://grafana.com/>

### Quarkus - Micrometer Metrics

<https://quarkus.io/version/1.11/guides/micrometer>

## ► Guided Exercise

# Adding Metrics to a Microservice

In this exercise, you will add metrics to a Quarkus microservice and its endpoints, by using the Micrometer extension, and afterwards observe them using Grafana.

## Outcomes

You should be able to use the Micrometer extension for enabling the metrics of a Quarkus microservice.

## Before You Begin

On the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that you have:

- a Prometheus server and a Grafana server configured and running for capturing and displaying metrics.
- a PostgreSQL database running and available in the port 5432.
- the required files for this activity in your workstation.

```
[student@workstation ~]$ lab monitor-metrics start
```

## Instructions

- 1. Observe the configuration used to start the Prometheus and Grafana servers when starting the exercise.
- 1.1. Open the file found in `/home/student/D0378/labs/monitor-metrics/prometheus.yml` and observe the `metrics_path` and the `targets`.

```
global:
 scrape_interval: 3s

scrape_configs:
 - job_name: 'quarkus_mp_metrics'
 metrics_path: '/q/metrics'
 scrape_interval: 3s
 static_configs:
 - targets: ['localhost:8080']
```

These values tell the Prometheus server to obtain metrics data from the `http://localhost:8080/q/metrics/application` HTTP endpoint. In this exercise, you will run the `expense-service` development server. You must first modify its contents to enable the metrics endpoints created by using the `micrometer` extension.

- 1.2. Open the file `/home/student/D0378/labs/monitor-metrics/datasource.yml`. This file defines the Prometheus server, running at `localhost` and port `9090` as a data source for the Grafana server.

```
datasources:
- name: Prometheus
 type: prometheus
 access: proxy
 orgId: 1
 url: http://localhost:9090
 version: 1
 editable: true
...output omitted...
```

- 1.3. You can also review the `start_services.sh` script. Do not run this script. This script is an example of how to start the containers for Prometheus and Grafana.
- ▶ 2. Use VSCode to open the `expense-service` project, found in `/home/student/D0378/labs/monitor-metrics/expense-service`.
  - ▶ 3. Modify the `pom.xml` file to add the `micrometer-registry-prometheus` extension to the project.

```
...output omitted...
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
</dependency>
</dependencies>
...output omitted...
```

- ▶ 4. Modify the `src/main/java/org/acme/rest/json/ExpenseResource.java` file to inject the metric registry. Update the `list()` and `create()` endpoint methods to add a counter measuring the number of invocations and a timer measuring the time to execute.

- 4.1. Inject the `MeterRegistry` bean into the `ExpenseResource` class.

```
import io.micrometer.core.instrument.MeterRegistry;
...output omitted...
public class ExpenseResource {
...output omitted...
 @Inject
 public MeterRegistry registry;
...output omitted...
```

- 4.2. Create a counter and a timer for listing expenses in a post-construct method.

```

import java.util.function.Supplier;
import io.micrometer.core.instrument.Counter;
...output omitted...
public class ExpenseResource {
 ...output omitted...
 private Counter listCounter;
 private Supplier<List<Expense>> listTimer;
...output omitted...

```

- 4.3. Initialize the counter and the timer in a `PostConstruct` annotated method.

```

import javax.annotation.PostConstruct;
...output omitted...
public class ExpenseResource {
 ...output omitted...
 @PostConstruct
 public void initMeters() {
 listCounter = registry.counter("callsToExpenseList",
 "description", "How many requests to list expenses have
happened.");
 listTimer = registry.timer("expenseListTimer",
 "description", "How long it takes to list expenses.")
 .wrap((Supplier<List<Expense>>) expenseService::list);
 }
...output omitted...

```

- 4.4. Update the `list` method body so that it increments the `listCounter` and uses the `listTimer`.

```

import java.util.function.Supplier;
import io.micrometer.core.instrument.Counter;
...output omitted...
public class ExpenseResource {
 ...output omitted...
 public List<Expense> list() {
 listCounter.increment();
 return listTimer.get();
 }
...output omitted...

```

- 4.5. Create the `callsToExpenseCreate` counter and `expenseCreateTimer` timer. In this case, instead of pre-creating the meters you can directly refer to them in the body of the `create` method:

```

...output omitted...
public Expense create(final Expense expense) {
 registry.counter("callsToExpenseCreate").increment();
 return registry.timer("expenseCreateTimer").wrap(
 (Supplier<Expense>) () -> expenseService.create(expense)).get();
}
...output omitted...

```

- 5. Add a gauge that provides the amount of time since the last invocation to the `list` method.
- 5.1. Create an object that stores the gauge value in the `ExpenseResource` class. For simplicity, you can use the `org.apache.commons.lang3.time.StopWatch` class.

```
import org.apache.commons.lang3.time.StopWatch;
...output omitted...
public class ExpenseResource {
 ...output omitted...
 private final Stopwatch stopWatch = Stopwatch.createStarted();
 ...output omitted...
```

- 5.2. Create the gauge meter in the initialization method of the `ExpenseResource` class. The gauge must link to the attribute just created:

```
import io.micrometer.core.instrument.Tags;
...output omitted...
public class ExpenseResource {
 ...output omitted...
 @PostConstruct
 public void initMeters() {
 registry.gauge("timeToLastExpenseList",
 Tags.of("description", "Time from the last time expenses list was
retrieved"),
 stopWatch, Stopwatch::getTime);
 ...output omitted...
```

- 5.3. Reset and restart the stopwatch on each call to the `list` method.

```
...output omitted...
public class ExpenseResource {
 ...output omitted...
 @GET
 public List<Expense> list() {
 listCounter.increment();
 stopWatch.reset();
 stopWatch.start();
 return listTimer.get();
 } ...output omitted...
```

- 6. Validate the current metrics work by starting the development server and making a request to the metrics application endpoint using curl.

- 6.1. In a terminal, inside of the `/home/student/D0378/labs/monitor-metrics/expense-service` directory, run the development server.

```
[student@workstation expense-service]$ mvn compile quarkus:dev
```

- 6.2. In a second terminal validate that the application is working as expected:

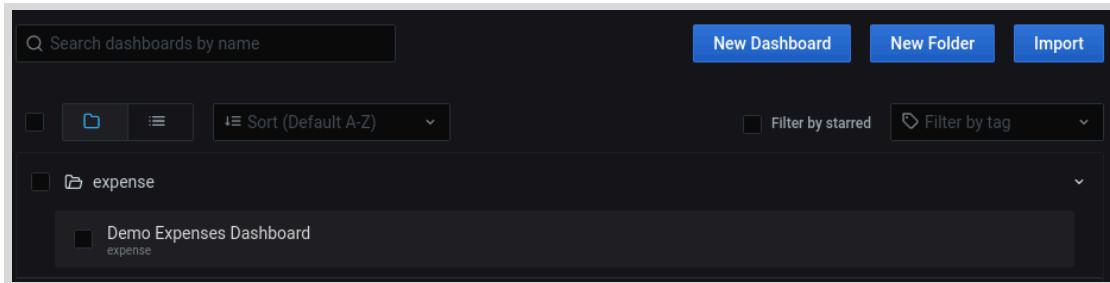
```
[student@workstation ~]$ curl -s localhost:8080/expenses | jq
[
 {
 "id": 1,
 "amount": 150.5,
 "name": "Groceries",
 "paymentMethod": "CASH"
 },
 {
 "id": 2,
 "amount": 25,
 "name": "Civilization VI",
 "paymentMethod": "CREDIT_CARD"
 }
]
```

- 6.3. Test the /q/metrics endpoint by using the curl tool. You will obtain metrics in Prometheus format, including the metrics you just created and summaries of the endpoint metrics.

```
[student@workstation ~]$ curl localhost:8080/q/metrics
...output omitted...
http_server_requests_seconds_count{method="GET",outcome="SUCCESS",status="200",uri="/expenses",} 2.0
http_server_requests_seconds_sum{method="GET",outcome="SUCCESS",status="200",uri="/expenses",} 0.319538126
http_server_requests_seconds_max{method="GET",outcome="SUCCESS",status="200",uri="/expenses",} 0.311554788
...output omitted...
HELP callsToExpenseList_total
TYPE callsToExpenseList_total counter
callsToExpenseList_total{description="How many requests to list expenses have happened.",} 2.0
...output omitted...
HELP timeToLastExpenseList
TYPE timeToLastExpenseList gauge
timeToLastExpenseList{description="Time from the last time expenses list was retrieved",} 137478.0
...output omitted...
HELP expenseListTimer_seconds_max
TYPE expenseListTimer_seconds_max gauge
expenseListTimer_seconds_max{description="How long it takes to list expenses.",} 0.203109386
...output omitted...
HELP expenseListTimer_seconds
TYPE expenseListTimer_seconds summary
expenseListTimer_seconds_count{description="How long it takes to list expenses.",} 2.0
expenseListTimer_seconds_sum{description="How long it takes to list expenses.",} 0.2067391
...output omitted...
```

- 7. Open Grafana in the web browser and observe the dashboard with the metrics created.

- 7.1. Enter the URL `http://localhost:3000/`. Use `admin` as both, login and password. Then, because this is the first time Grafana has been accessed, it will request a password change. Skip the password reset by clicking on the `Skip` link found below the `Submit` button.
- 7.2. After the login has been successful, change to the URL `http://localhost:3000/dashboards`. Click the `expense` folder, to find the `Demo Expenses Dashboard` link. Click the link to enter the dashboard view.



- 7.3. To generate requests, in a terminal, execute the script found in `/home/student/D0378/labs/monitor-metrics/generate_data.sh`.

```
[student@workstation ~]$ cd ~/D0378/labs/monitor-metrics/
[student@workstation monitor-metrics]$./generate_data.sh
Sending requests to GET http://localhost:8080/expenses/ and POST http://
localhost:8080/expenses/
```

- 8. Observe the dashboard being updated by looking at the already open dashboard in Grafana.



- ▶ 9. Close VSCode and, in the terminal window, press **Ctrl+C** to stop the development server.

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab monitor-metrics finish
```

This concludes the guided exercise.

# Tracing Requests Across Microservices

## Objectives

After completing this section, you should be able to generate request traces by using SmallRye MicroProfile OpenTracing.

## Describing Distributed Tracing

Distributed Tracing is the process of tracking the performance of individual services in an application by tracing the path of the service calls in the application. Each time a user takes action in an application, a request is executed, which might require many services to interact to produce a response.

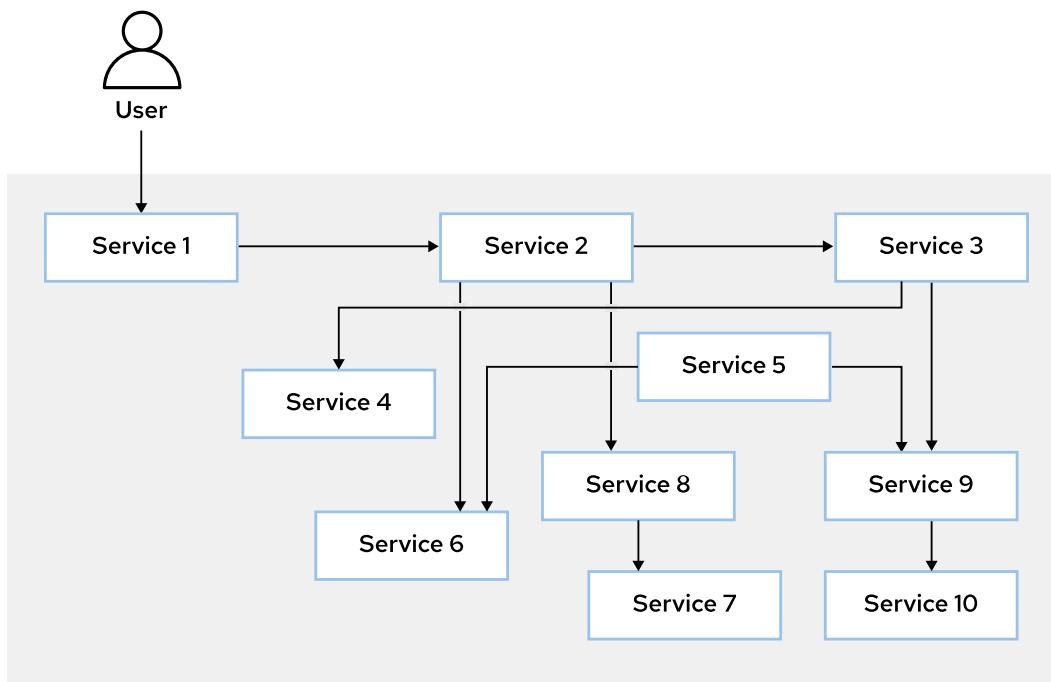
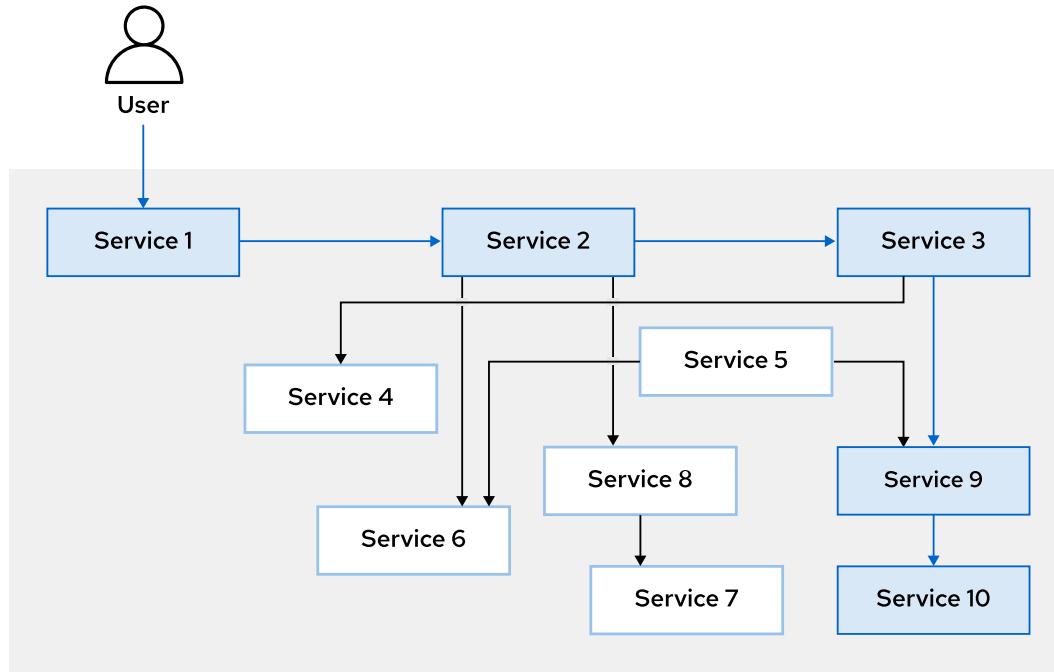


Figure 9.3: Without distributed tracing

**Figure 9.4: With distributed tracing**

In a microservices enabled architecture with a large number of individual microservices, a single client request that achieves a certain business requirement might involve calling multiple individual microservices in a particular sequence. An important aspect of maintaining and developing a distributed system is troubleshooting performance issues. Because a single client call can interact with multiple services, analyzing the debugging logs of an individual service might not help troubleshooting performance issues.

Distributed tracing allows developers to visualize call flows in a microservices application. Understanding the sequence of calls (how many calls occur in a serial fashion versus how many occur in parallel), and sources of latency is useful when maintaining a distributed system.

For example, if a request takes too long, causing performance issues, then identify the service or services causing the slowdown and examine the network latency between service calls.

Distributed tracing is useful for monitoring, network profiling, and troubleshooting the interaction between services in modern, cloud-native, microservices-based applications.

## Traces and Spans in Distributed Tracing

In the context of distributed tracing, it is important to understand two terms.

### **Span**

A Span represents a logical unit of work, which has a unique name, a start time, and the duration of execution. To model the service call flow in a service mesh, spans are nested and executed in a particular order.

### **Trace**

A Trace is an execution path of services in the service mesh, and is comprised of one or more spans.

Consider an application with the following microservices.

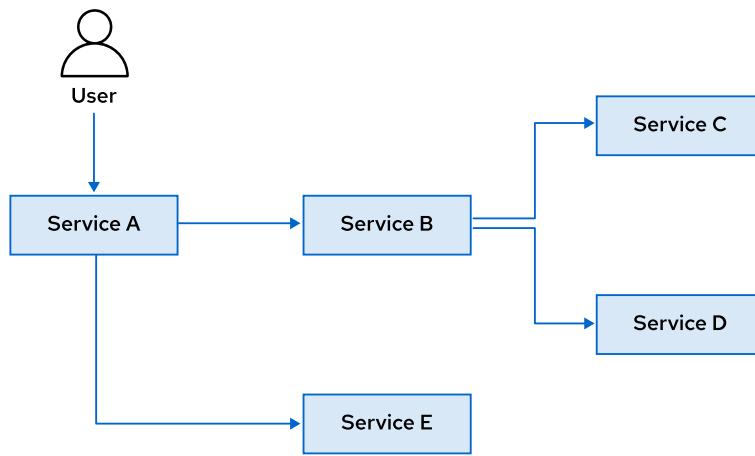
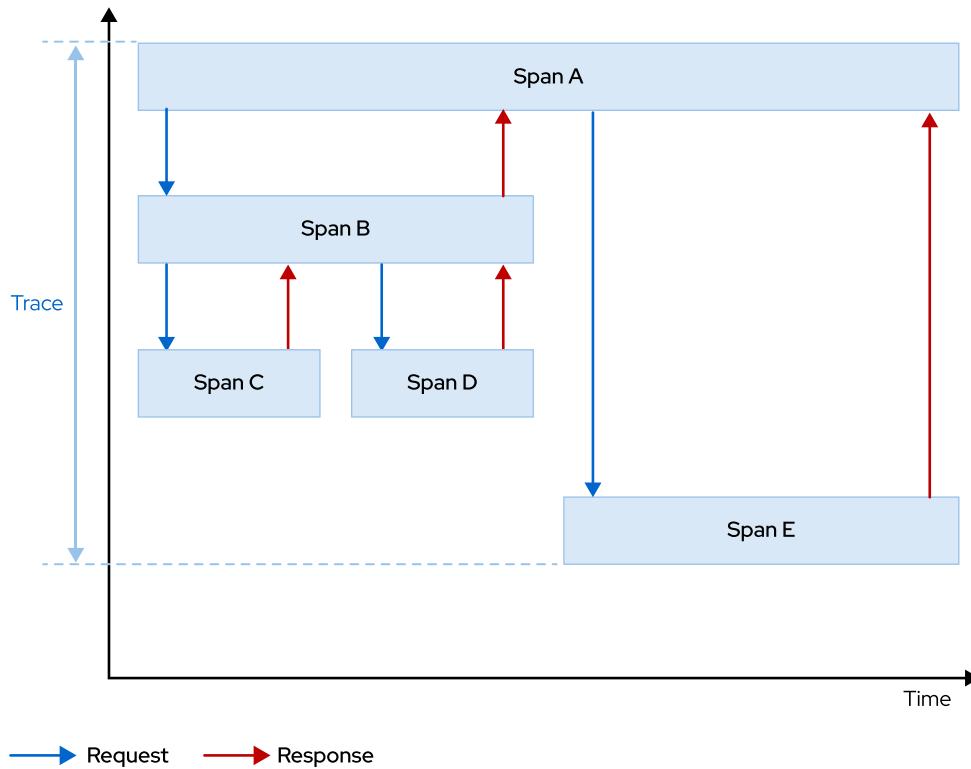


Figure 9.5: Request call path

In this example.

- Service A is the request entry point for the application.
- Because Service A is the entry point for the application, it is called the parent span. As shown in Figure 9.5, Service A makes two service calls: one to Service B and one to Service E. Thus, Service B and Service E are child spans of Service A.
- Service B in turn calls Service C and Service D before returning the response to Service A. Service B is a parent span. Service C and Service D are child spans.

The following is a line graph representation of a single trace and its constituent spans.

**Figure 9.6: Traces and spans**

## Introducing Jaeger

Jaeger is a distributed tracing platform. Jaeger allows developers to configure their services to enable gathering runtime statistics about their performance.

Jaeger is made up of several components, which work together to collect, store, and display tracing data.

### Jaeger Client

The OpenTracing API defines standard APIs for instrumentation and distributed tracing of microservices applications. Jaeger clients are language specific implementations of the OpenTracing API. They are used to configure applications for distributed tracing.

### Jaeger Agent

The Jaeger agent is a network daemon, which listens for span data sent over User Datagram Protocol (UDP), which it batches and sends to the collector. The agent is meant to be placed on the same host as the application being traced.

### Jaeger Collector

The Collector receives runtime statistics from the agent and places them in an internal queue for processing. This allows the collector to return a response immediately to the agent.

### Storage

Collectors require a persistent storage back end. Jaeger has a pluggable mechanism for storage.

### Query

Query is a service that retrieves runtime statistics from storage.

## Jaeger Console

Jaeger provides a web based console that lets you visualize your distributed tracing data. Using the console, you can trace the path of requests as they flow through the services in your application.

## Enabling Distributed Tracing in Quarkus Applications

You must explicitly enable tracing in your applications to generate traces and spans and to propagate context information.

Enabling tracing for Quarkus based applications is very simple. Quarkus supports tracing with minimum source code changes.

To enable tracing in Quarkus applications, the following steps are required.

- Include the `quarkus-smallrye-opentracing` dependency in the project Maven `pom.xml` file. This dependency implements the OpenTracing API, an open standard based on Jaeger for distributed tracing.

```
<dependency>
 <groupId>io.quarkus</groupId>
 <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

You can also use the Maven Quarkus plug-in to add the dependency.

```
[user@demo ~]$ mvn quarkus:add-extension \
-Dextension="quarkus-smallrye-opentracing"
```

- Enable tracing related properties in the Quarkus `application.properties` file. This file is used to externalize configuration for Quarkus applications.

```
quarkus.jaeger.service-name=myservice ①
quarkus.jaeger.sampler-type=const ②
quarkus.jaeger.sampler-param=1 ③
quarkus.jaeger.endpoint=http://localhost:14268/api/traces ④
quarkus.jaeger.propagation=b3 ⑤
quarkus.jaeger.reporter-log-spans=true ⑥
```

- ① A unique service name to identify traces and spans sent to Jaeger. You will be able to identify traces and spans by using this name in the Jaeger web console.
- ② Indicates the rate at which traces and spans should be collected from this application. Options include; `const` (collect all samples), `probabilistic` (random sampling), `ratelimiting` (collect samples at a configurable rate per second), and `remote` (control sampling from a central Jaeger back end). Refer to <https://www.jaegertracing.io/docs/1.20/sampling/> for more details.
- ③ Indicates the percentage of requests for which spans should be collected. It should value between 0 and 1. 0 indicates no sample collection, and a 1 indicates collecting all samples (100%). This value should not be set to 1 in production environments with a large number of requests. This is usually set to 1 in development and QA environments to debug and troubleshoot inter-service latency and performance issues.

- ④ The URL of the Jaeger collector. By default, the traces and spans are sent to a local collector using UDP. However, in cloud environments and secure enterprise deployments, UDP multi-casting is usually disabled. In these scenarios, configure the application to send trace information over TCP.
  - ⑤ Indicates Jaeger to propagate all x-b3-\* related headers. This is required to construct the parent-child relationships in the call graph. Failure to set this makes the parent-child relationships not show between spans in the Jaeger web console.
  - ⑥ Logs span information for all incoming and outgoing traffic from the application.
3. The first two steps are enough to enable tracing for all REST endpoints (that is, classes with JAX-RS HTTP method annotations such as @GET or @POST) in the application. However, for more complex applications, you can control which classes and methods must be enabled for tracing. For example, you can annotate select classes and methods in your service layer and data access layer code to troubleshoot and identify bottlenecks.
- Annotate classes with the @Traced annotation to enable tracing for the entire class. You can also add this annotation at a method level to enable tracing for specific methods and exclude the rest.



### Note

The previous steps are enough to enable distributed tracing for Quarkus applications. All incoming and outgoing traffic is traced and the context propagation of relevant headers is taken care of automatically.

To customize the tracing (for example, add more contextual information to traces and spans with tags and other metadata), you can use the OpenTracing API. Refer to the OpenTracing documentation at <https://opentracing.io/guides/java/> for more details.

## Viewing Traces and Spans Using the Jaeger Web Console

The Jaeger web console provides a graphical view of the traces and spans collected during tracing.

You can run a local instance of Jaeger by running the "all in one" container image provided by the Jaeger project. This container image contains all the Jaeger components bundled as a single, runnable image for development and test environments.

To start the all in one Jaeger container, run the following command.

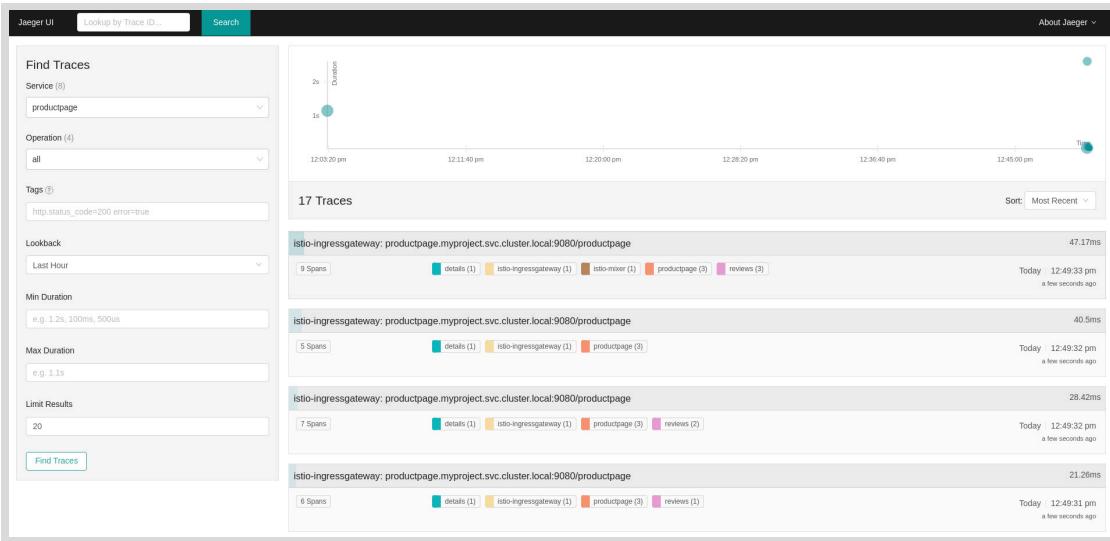
```
[user@demo ~]$ podman run --rm --name jaeger \
-p 5775:5775/udp \
-p 6831:6831/udp \
-p 6832:6832/udp \
-p 5778:5778 \
-p 16686:16686 \
-p 14268:14268 \
quay.io/jaegertracing/all-in-one:1.21.0
```

To view details about traces and spans in the Jaeger console, do the following.

1. Navigate to <http://localhost:16686> and open the Jaeger web console.

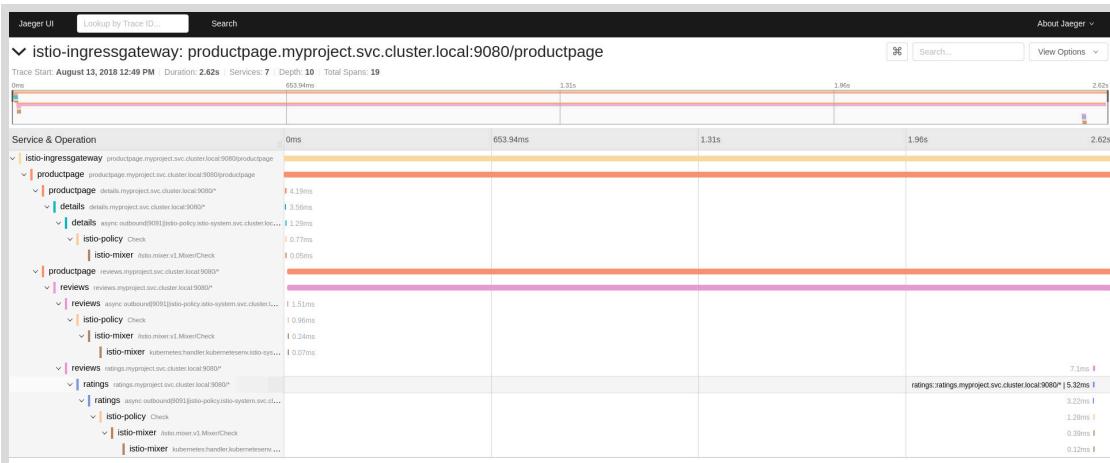
## Chapter 9 | Monitoring Quarkus Microservices

2. In the left pane of the Jaeger console, from the Service menu, select your application and click Find Traces at the bottom of the pane. A list of traces gathered for the application are displayed.



**Figure 9.7: List of traces**

3. Click one of the traces in the list to open a detailed view of that trace.



**Figure 9.8: Trace details**



### References

For more information, refer to the Jaeger documentation at <https://www.jaegertracing.io/>

### OpenTracing API

<https://opentracing.io/>

## ► Guided Exercise

# Tracing Requests Across Microservices

In this exercise, you will enable distributed tracing for a set of microservices, and visualize traces and spans using the Jaeger web console.

## Outcomes

You should be able to use the Quarkus OpenTracing extension to enable distributed tracing for microservices.

## Before You Begin

To use your provided OpenShift Container Platform instance, you need the appropriate credentials supplied by your Red Hat Training Online Learning environment. Review the *Guided Exercise: Verifying OpenShift Credentials* section, if you need guidance for finding your credentials.

As the **student** user on the **workstation** machine, use the `lab` command to prepare your system for this exercise.

This command prepares the source code for the Quarkus calculator application, which you used in a previous exercise on your workstation.

```
[student@workstation ~]$ lab monitor-trace start
```

## Instructions

- 1. Briefly review the source code of the Quarkus Calculator application. The source code is in the `~/D0378/labs/monitor-trace/` folder. Use **VSCodium** to open the folder as a new Quarkus application.

```
[student@workstation ~]$ codium ~/D0378/labs/monitor-trace/
```

The Quarkus calculator application contains three microservices.

### **solver**

Evaluates that the equation is a decimal number, or a composition of sums and multiplications.

### **adder**

Gets two equations and returns the sum of their results. Relies on the **solver** microservices to solve both sides of the sum.

### **multiplier**

Gets two equations and returns the product of their results. Relies on the **solver** microservices to solve both sides of the multiplication.

Each of the three microservices expose their own REST API. A single client call to solve an equation could result in multiple calls between these microservices.

Enabling distributed tracing will enable you to trace the path of these calls and help in troubleshooting. Tracing allows you to gather the timing data for each of the calls, and will help you analyze performance bottlenecks.

In the following steps, you will enable tracing for all the microservices, and use a Jaeger container running locally on the **workstation** VM to capture and visualize the traces.

- ▶ 2. Add the `quarkus-smallrye-opentracing` Quarkus extension to all the three services to enable tracing.
  - 2.1. Inspect the `~/D0378/labs/monitor-trace/add-tracing.sh` script. This script adds the `quarkus-smallrye-opentracing` Quarkus extension to each of the microservices.

```
...output omitted...
echo "Adding tracing extension to the 'solver' project "
cd solver
mvn quarkus:add-extension -Dextension="quarkus-smallrye-opentracing"
...output omitted...
```

- 2.2. Run the `~/D0378/labs/monitor-trace/add-tracing.sh` script.

```
[student@workstation ~]$ sh ~/D0378/labs/monitor-trace/add-tracing.sh
Adding tracing extension to the 'solver' project
...output omitted...
Extension io.quarkus:quarkus-smallrye-opentracing has been installed
...output omitted...
[INFO] BUILD SUCCESS
...output omitted...
```

- ▶ 3. Start a local instance of Jaeger by using podman.
  - 3.1. Inspect the `~/D0378/labs/monitor-trace/jaeger.sh` script, which starts the Jaeger all in one container.
  - 3.2. In a new terminal on the **workstation** VM, start the all in one Jaeger container by using the `jaeger.sh` script.

```
[student@workstation ~]$ sh ~/D0378/labs/monitor-trace/jaeger.sh
Starting the all-in-one Jaeger container
...output omitted...
"Starting jaeger-collector HTTP server","http host-port":"":14268"}
...output omitted...
"Query server started","port":16686,"addr":"":16686"}
"Health Check state change","status":"ready"
...output omitted...
```

- ▶ 4. Edit the Quarkus `application.properties` file in all of the three services to send tracing information to Jaeger.
  - 4.1. Edit the `~/D0378/labs/monitor-trace/adder/src/main/resources/application.properties` file, and add the following properties.

```
quarkus.jaeger.service-name=adder
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=
%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```

The `quarkus.jaeger.endpoint` property configures the URL of the Jaeger collector, which gathers tracing data from the microservices.

- 4.2. Edit the `~/D0378/labs/monitor-trace/multiplier/src/main/resources/application.properties` file, and add the following properties.

```
quarkus.jaeger.service-name=multiplier
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=
%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```

- 4.3. Edit the `~/D0378/labs/monitor-trace/solver/src/main/resources/application.properties` file, and add the following properties.

```
quarkus.jaeger.service-name=solver
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=
%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```



### Important

The value of the `quarkus.jaeger.service-name` property must be unique. This value is used to identify and label the traces in the Jaeger web console. You should set this value to the name of the corresponding service (adder, multiplier, and solver).

5. Start the three microservices.

Inspect and run the `~/D0378/labs/monitor-trace/start.sh` script in a new terminal on the workstation VM.

```
[student@workstation ~]$ sh ~/D0378/labs/monitor-trace/start.sh
Starting the 'solver' project
...output omitted...
Starting the 'adder' project
```

```
...output omitted...
Starting the 'multiplier' project
...output omitted...
Press enter to Terminate
...output omitted...
```

- 6. Capture traces by invoking the REST endpoints exposed by the microservices. Use the Jaeger web console to visualize the traces and execution timing.

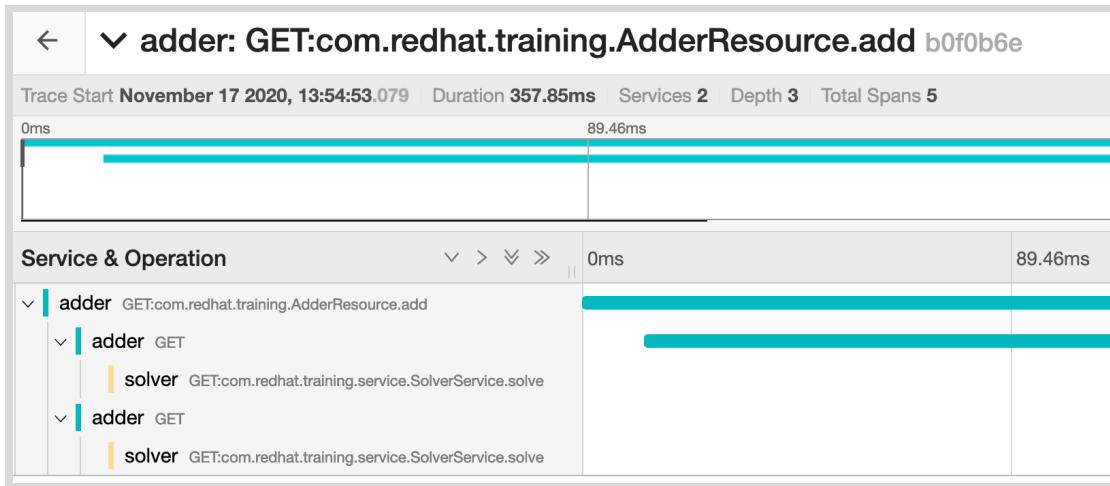
- 6.1. On the **workstation** VM, navigate to the Jaeger web console at `http://localhost:16686` in a web browser. You should not see any traces because you have not invoked any endpoint in the application.
- 6.2. Open a new terminal on the **workstation** VM, and invoke the endpoint for the **adder** microservice.

```
[student@workstation ~]$ curl "http://localhost:8081/adder/5/3"
8.0
```

Refresh the Jaeger web console. Select the **adder** service from the **Service** field in the **Search** panel on the left. Click **Find Traces** to view the trace.

The screenshot shows the Jaeger web interface. On the left, the **Search** panel is open, displaying search criteria for the **Service** (set to **adder**), **Operation** (set to **all**), and **Tags** (set to `http.status_code=200 error=true`). Below these are filters for **Lookback** (set to **Last Hour**), **Min Duration** (set to `e.g. 1.2s, 100ms, 500us`), and **Max Duration** (set to `e.g. 1.2s, 100ms, 500us`). The **Limit Results** dropdown is set to **20**. At the bottom of the search panel is a **Find Traces** button. To the right, the main pane displays a timeline chart titled **Duration** with a single trace point at 03:43:20 am. Below the chart, the text **1 Trace** is displayed. A callout box highlights the trace entry in the list, which includes the operation name **adder: GET:com.redhat.training.AdderResource.add**, a trace ID **b0f0b6e**, and a count of **5 Spans**. To the right of the spans are two colored boxes: a blue one labeled **adder (3)** and a yellow one labeled **solver (2)**.

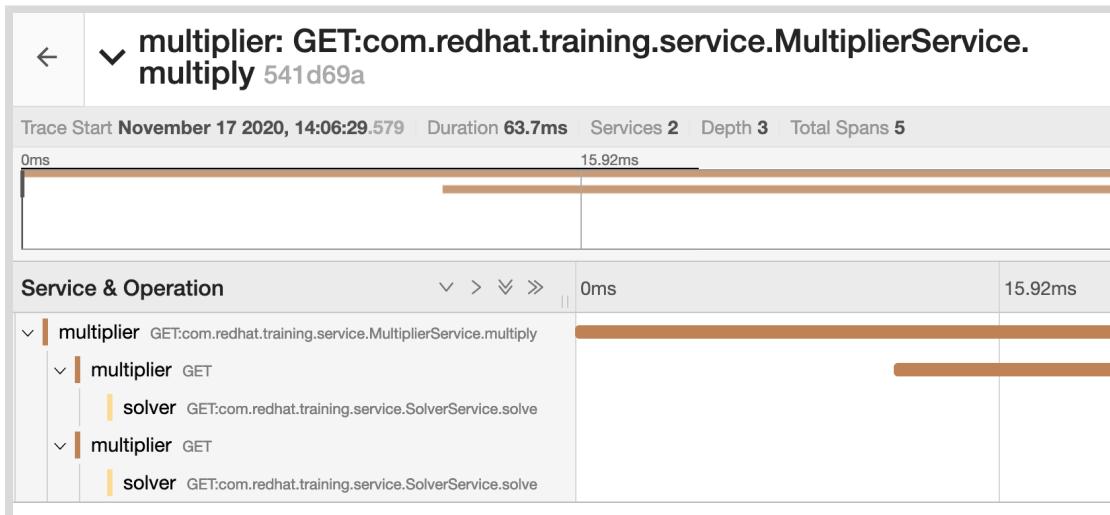
Click the **adder:GET:com.redhat.training.AdderResource.add** trace to view the details of the trace.



- 6.3. Switch to the terminal where you ran the `curl` command and invoke the endpoint for the `multiplier` microservice.

```
[student@workstation ~]$ curl "http://localhost:8082/multiplier/5/4"
20.0
```

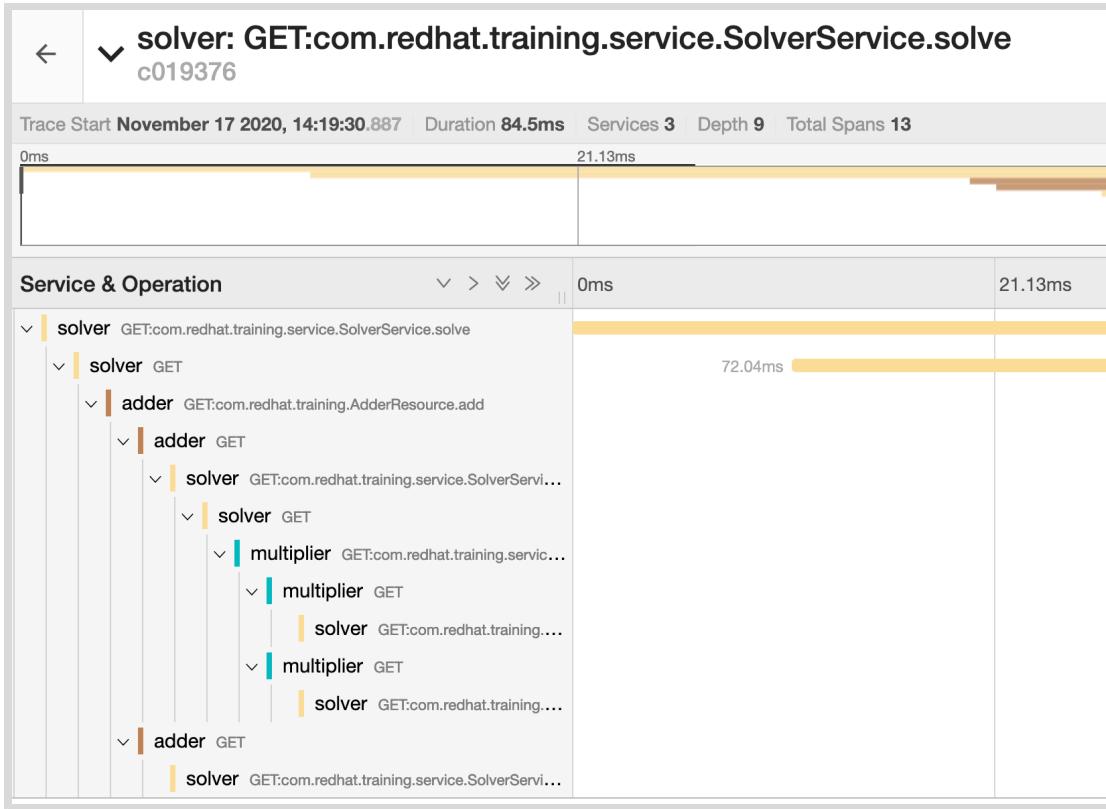
Find the trace for this invocation by selecting the `multiplier` service from the `Service` field in the Jaeger home page, and click `Find Traces`. Click the `multiplier:GET:com.redhat.training.MultiplierService.multiply` trace to view the details of the trace. The output should be as follows.



- 6.4. Invoke the endpoint for the `solver` microservice. This service can take compound equations with addition and multiplication terms as input.

```
[student@workstation ~]$ curl "http://localhost:8080/solver/5*4+3"
23.0
```

Find the trace for this invocation by selecting the `solver` service from the `Service` field in the Jaeger home page, and click `Find Traces`. Click the `solver:GET:com.redhat.training.SolverService.solve` trace to view the details of the trace. The output should be as follow.



#### ► 7. Clean up. Stop the microservices and the Jaeger container.

- 7.1. To stop the three microservices, press **Enter** in the terminal window where you ran the `start.sh` script.
- 7.2. To stop the Jaeger container, press **Ctrl+C** in the terminal window where you ran the `jaeger.sh` script.

## Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab monitor-trace finish
```

This concludes the guided exercise.

## ► Lab

# Monitoring Microservices

In this lab, you will enable tracing in the conference application to identify and fix a performance issue. You will also collect custom application metrics and display them on a dashboard.

## Outcomes

You should be able to:

- Enable distributed tracing to track the flow of requests between microservices in the application, and identify performance issues.
- Enable custom application metrics collection for specific microservice endpoints.
- Visualize the tracing and metrics data by using Jaeger and Grafana respectively.

## Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command ensures that the directories and required files for starting and grading the lab are set up.

```
[student@workstation ~]$ lab monitor-review start
```

## Instructions

1. Briefly inspect the source code of the `schedule`, `session`, and `speaker` microservices in the `~/D0378/labs/monitor-review` folder.
2. Inspect and run the `~/D0378/labs/monitor-review/containers.sh` script. This script starts the following containers on the workstation VM.
3. Inspect and run the `~/D0378/labs/monitor-review/services.sh` script. This script starts the `schedule`, `session`, and `speaker` back-end microservices on the workstation in `dev` mode by using the Maven Quarkus plug-in.
4. Navigate to the web application home page at `http://localhost:8080` by using a browser. Explore the features of the web application.



### Note

If you do not see the web application content, then click the hamburger button on the top left corner of the page (the three horizontal lines button) to hide the application menu. Optionally you can increase the screen resolution in your workstation and maximize your browser window.

5. The three microservices communicate with each other in different ways depending on the page being viewed. Enable distributed tracing in all three back-end microservices to identify the cause of the slow down.

Enable tracing with the following properties.

- Collect all (100%) trace samples, for all requests, from the front-end.
- Send all traces to the Jaeger collector instance running locally on port 14268.
- Propagate all b3-\* headers to ensure full traceability, and log all spans collected.

6. Restart the microservices by running the `services.sh` script. Invoke the session details page and visualize the collected traces and spans in the Jaeger UI.

The Jaeger UI is available at <http://localhost:16686>.

7. Identify the method causing the slowdown. Method calls taking more than 3 seconds are considered unacceptably slow.

8. The endpoint responsible for slow response times is invoking other classes in the service, which do not show up in the collected traces. Enable tracing on these dependent classes and methods to isolate the performance issue.

9. Fix the offending code to remove the bottleneck. Re-test the session detail page and confirm that it now renders faster than when you started the exercise.

10. Stop all the running back-end microservices by pressing `Enter` in the terminal window where you ran the `services.sh` script.

You have been asked by the operations team to also enable metrics collection for certain endpoints. Prometheus is configured to scrape application metrics from the `session` microservice. Grafana is configured to render a pre-created dashboard from data stored in Prometheus.

The Grafana UI is available at <http://localhost:3000/dashboards>.

The Prometheus UI is available at <http://localhost:9090>.

Enable the following metrics in the `session` microservice.

- Collect the total count of attendees who clicked the `I am interested` button in the session detail page. Clicking this button calls the `subscribe()` method in the `SessionResource` class. The metric must be named `subscriberCount`. This metric, despite being a never-decreasing metric, is displayed as a `Gauge` in the Grafana dashboard.
- Collect the response time of the endpoint method that is responsible for the data displayed on the session details page. The `retrieveSession()` method in the `SessionResource` class fetches the session details. The metric must be named `sessionDetailsFetch`. This metric is displayed as a `Graph` in the Grafana dashboard.

11. Run the `services.sh` script to restart all the back-end microservices.

Browse the session detail page multiple times and click the `I am interested` button to gather metrics.

Verify that the metrics are displayed in the Grafana dashboard.

## Evaluation

Grade your work by running the `lab monitor-review grade` command from your `workstation` machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab monitor-review grade
```

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab monitor-review finish
```

This concludes the lab.

## ► Solution

# Monitoring Microservices

In this lab, you will enable tracing in the conference application to identify and fix a performance issue. You will also collect custom application metrics and display them on a dashboard.

## Outcomes

You should be able to.

- Enable distributed tracing to track the flow of requests between microservices in the application, and identify performance issues.
- Enable custom application metrics collection for specific microservice endpoints.
- Visualize the tracing and metrics data by using Jaeger and Grafana respectively.

## Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this lab.

This command ensures that the directories and required files for starting and grading the lab are set up.

```
[student@workstation ~]$ lab monitor-review start
```

## Instructions

1. Briefly inspect the source code of the `schedule`, `session`, and `speaker` microservices in the `~/D0378/labs/monitor-review` folder.
2. Inspect and run the `~/D0378/labs/monitor-review/containers.sh` script. This script starts the following containers on the workstation VM.
  - Starts an instance of Jaeger, Prometheus and Grafana as containers.
  - Starts an instance of the front-end web application as a container.
  - Starts an instance of PostgreSQL and MongoDB databases used by the back-end microservices to store application data.
3. Inspect and run the `~/D0378/labs/monitor-review/services.sh` script. This script starts the `schedule`, `session`, and `speaker` back-end microservices on the workstation in `dev` mode by using the Maven Quarkus plug-in.
4. Navigate to the web application home page at `http://localhost:8080` by using a browser. Explore the features of the web application.

**Note**

If you do not see the web application content, then click the hamburger button on the top left corner of the page (the three horizontal lines button) to hide the application menu. Optionally you can increase the screen resolution in your workstation and maximize your browser window.

- 4.1. Click **Schedules** to view the list of scheduled conferences. Click any of the schedules to view the details of the conference.
- 4.2. Click **Speakers** to view the list of speakers. Click any of the speakers to view the details of the speaker.
- 4.3. Click **Sessions** to view the list of scheduled sessions. Click any of the sessions to view the details of the session.  
Note that there is a significant delay when rendering the session detail page in comparison to the other pages.
- 4.4. Stop all the running back-end microservices by pressing **Enter** in the terminal window where you ran the `services.sh` script.

**Note**

If you cancel the script process or close the terminal window without letting it terminate then some services may keep on running in the background. This will cause service ports to be unavailable and services unable to start again. Use the `killall java` command to terminate all services manually in case of need.

5. The three microservices communicate with each other in different ways depending on the page being viewed. Enable distributed tracing in all three back-end microservices to identify the cause of the slow down.  
Enable tracing with the following properties.
  - Collect all (100%) trace samples, for all requests, from the front-end.
  - Send all traces to the Jaeger collector instance running locally on port 14268.
  - Propagate all `b3-*` headers to ensure full traceability, and log all spans collected.
- 5.1. Add the `quarkus-smallrye-opentracing` extension to all three microservices. Use the `mvn quarkus:add-extension` command from the directory of the parent project to add the extensions.

```
[student@workstation ~]$ cd ~/D0378/labs/monitor-review/
[student@workstation monitor-review]$ mvn quarkus:add-extension \
-Dextension="quarkus-smallrye-opentracing" \
-pl="microservice-session,microservice-schedule,microservice-speaker"
```

- 5.2. Add the following snippet to the respective microservice `application.properties` file to configure tracing. Replace `service-name` with the name of the corresponding microservice, which are `schedule`, `session` and `speaker`.

```
quarkus.jaeger.service-name=<service-name>
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
quarkus.jaeger.endpoint=http://localhost:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```

6. Restart the microservices by running the `services.sh` script. Invoke the session details page and visualize the collected traces and spans in the Jaeger UI.

The Jaeger UI is available at <http://localhost:16686>.

- 6.1. Run the `services.sh` script to restart the back-end microservices after enabling tracing.

```
[student@workstation ~]$ sh ~/D0378/labs/monitor-review/services.sh
```

- 6.2. Navigate to the web application home page at <http://localhost:8080> in a browser.

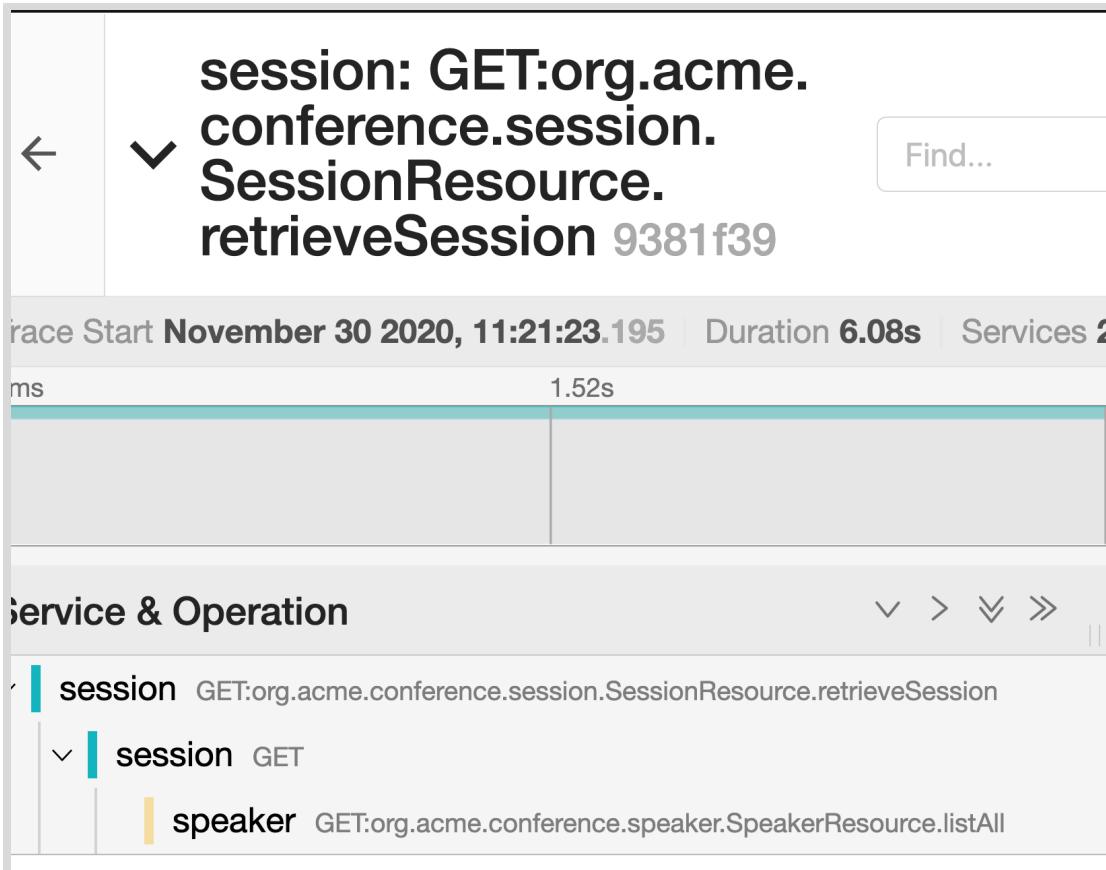
Click **Sessions** to view the list of scheduled sessions. Click any of the sessions to view the details of the session.

- 6.3. Navigate to the Jaeger web console at <http://localhost:16686> in a browser.

- 6.4. In the Jaeger web console, select the **session** service from the **Service** field in the Search panel on the left. Click **Find Traces** to view the trace.

Trace Details	Duration
session: GET:org.acme.conference.session.SessionResource.retrieveSession	6.09s
session: GET:org.acme.conference.session.SessionResource.allSessions	859.49ms

- 6.5. Click the trace for the `retrieveSession` method to view the details.



7. Identify the method causing the slowdown. Method calls taking more than 3 seconds are considered unacceptably slow.
  - 7.1. The `retrieveSession()` method of the `org.acme.conference.session.SessionResource` class in the `session` microservice takes around 6 seconds to respond.  
Inspect the source code of this method to identify why this method is slow.
8. The endpoint responsible for slow response times is invoking other classes in the service, which do not show up in the collected traces. Enable tracing on these dependent classes and methods to isolate the performance issue.
  - 8.1. The `retrieveSession()` method calls the `findById()` method in the `org.acme.conference.session.SessionStore` class to fetch the session details from the database.
 

```
final Optional<Session> result = sessionStore.findById(sessionId);
```

 Inspect the source code of the `SessionStore` class. Observe that tracing is not enabled at the class or method level.
  - 8.2. Enable tracing for the `findById()` method. Import the required class for enabling tracing.
 

```
import org.eclipse.microprofile.opentracing.Traced;
```

- 8.3. Add the `@Traced` annotation to the `findById()` method.

```
@Traced
public Optional<Session> findById(String sessionId) {
...output omitted...
```

- 8.4. Save your changes and navigate to the session details page from the front-end web application.

Inspect the trace from the Jaeger web console, and confirm that the `findById()` method is the cause of the slow response.



9. Fix the offending code to remove the bottleneck. Re-test the session detail page and confirm that it now renders faster than when you started the exercise.

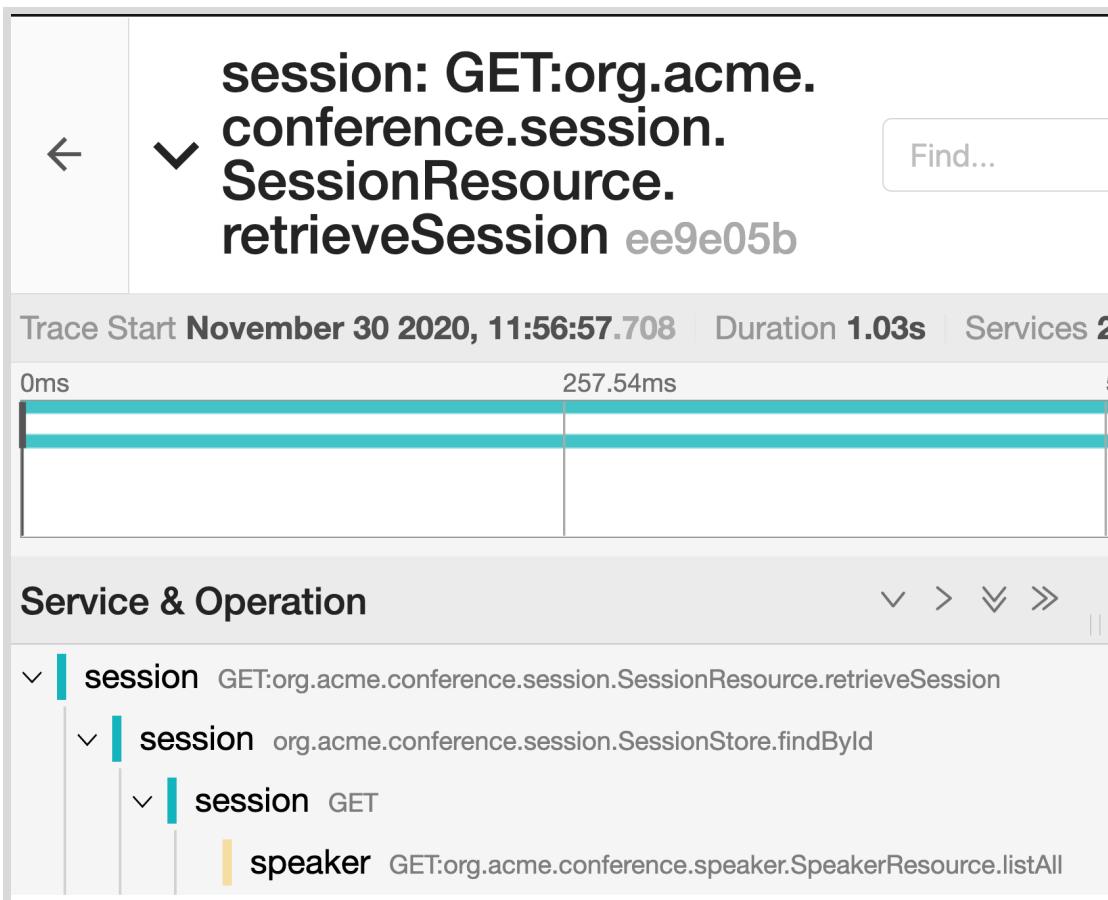
- 9.1. The `findById()` method simulates a slow SQL query call, which causes a 5 second pause during execution.

```
@SuppressWarnings("unchecked")
List<Tuple> result = em.createNativeQuery(
 "SELECT 1 " +
 "FROM pg_sleep(5) ", Tuple.class).getResultList();
```

Delete or comment out this line and the corresponding import for `Tuple`, to speed up the response time.

- 9.2. Save your changes and navigate to the session details page from the front-end web application.

Inspect the trace from the Jaeger web console, and confirm that the `findById()` method is no longer a bottleneck, and the session details page renders much quicker.



- Stop all the running back-end microservices by pressing `Enter` in the terminal window where you ran the `services.sh` script.

You have been asked by the operations team to also enable metrics collection for certain endpoints. Prometheus is configured to scrape application metrics from the `session` microservice. Grafana is configured to render a pre-created dashboard from data stored in Prometheus.

The Grafana UI is available at `http://localhost:3000/dashboards`.

The Prometheus UI is available at `http://localhost:9090`.

Enable the following metrics in the `session` microservice.

- Collect the total count of attendees who clicked the `I am interested` button in the session detail page. Clicking this button calls the `subscribe()` method in the `SessionResource` class. The metric must be named `subscriberCount`. This metric, despite being a never-decreasing metric, is displayed as a `Gauge` in the Grafana dashboard.
- Collect the response time of the endpoint method that is responsible for the data displayed on the session details page. The `retrieveSession()` method in the `SessionResource` class fetches the session details. The metric must be named `sessionDetailsFetch`. This metric is displayed as a `Graph` in the Grafana dashboard.

- 10.1. Add the quarkus-smallrye-metrics extension to the session microservice. Use the mvn quarkus:add-extension command from the root directory of the session microservice to add the extension.

```
[student@workstation ~]$ cd ~/DO378/labs/monitor-review/
[student@workstation monitor-review]$ cd microservice-session
[student@workstation microservice-session]$ mvn quarkus:add-extension \
-Dextension="quarkus-micrometer-registry-prometheus"
```

- 10.2. Import the required class in the SessionResource class for enabling metrics.

```
import io.micrometer.core.annotation.Counted;
import io.micrometer.core.annotation.Timed;
```

- 10.3. Add the @Counted annotation to the subscribe() method.

```
@Counted(value = "subscriberCount", description = "How many attendees are
interested in sessions.")
@Produces(MediaType.APPLICATION_JSON)
public Response subscribe(@PathParam("sessionId") final String sessionId) {
...output omitted...
```

- 10.4. Add the @Timed annotation to the retrieveSession() method.

```
@Timed(value = "sessionDetailsFetch", description = "How long it takes to fetch
details of a session.")
@Produces(MediaType.APPLICATION_JSON)
...output omitted...
public Response retrieveSession(@PathParam("sessionId") final String sessionId) {
...output omitted...
```

11. Run the services.sh script to restart all the back-end microservices.

Browse the session detail page multiple times and click the **I am interested** button to gather metrics.

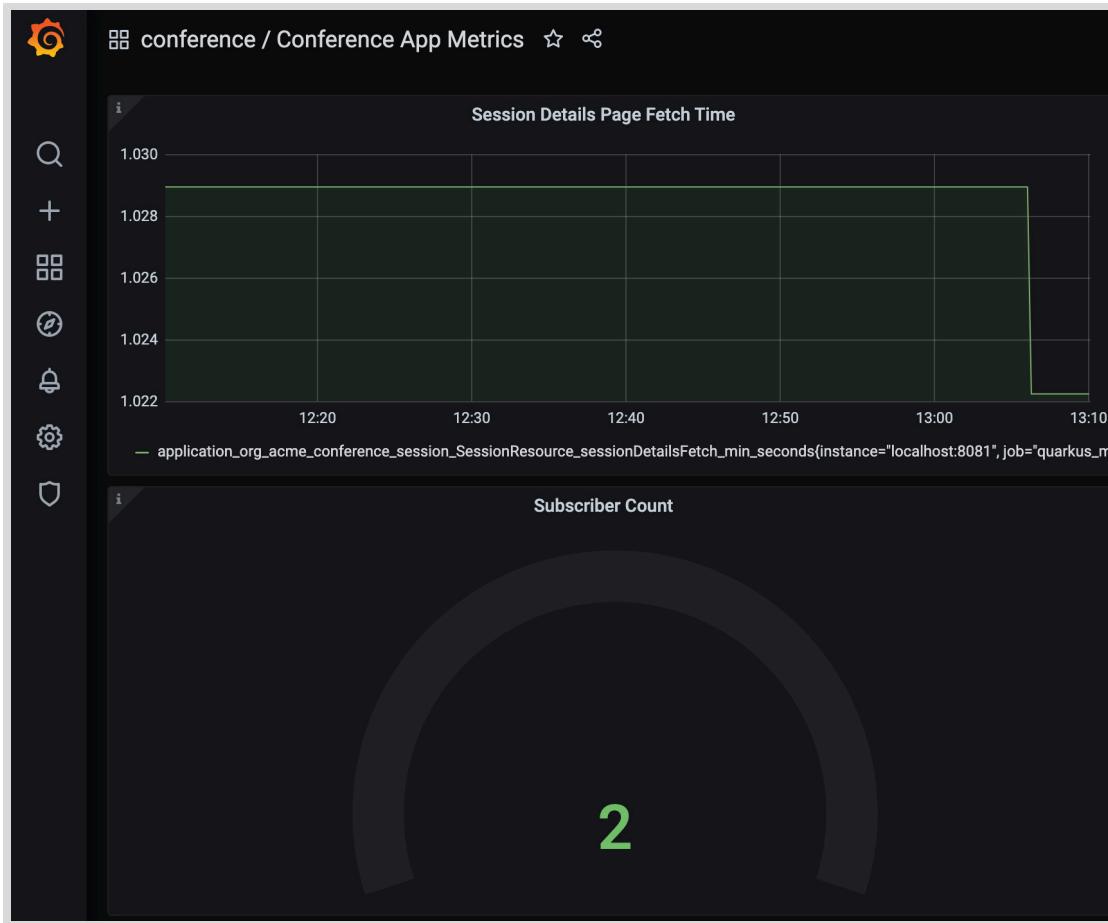
Verify that the metrics are displayed in the Grafana dashboard.

- 11.1. Navigate to the Grafana UI at <http://localhost:3000/dashboards> in a browser.

Log in with admin as the user name and admin as the password. You will be prompted to change your default password. Click **Skip** to skip changing the password.

- 11.2. Click the **conference** folder, and then click the **Conference App Metrics** dashboard.

The dashboard shows the metrics collected from the application. Click the **I am interested** button multiple times, and verify that the **Subscriber Count** gauge shows the number of times that the button was clicked.



## Evaluation

Grade your work by running the `lab monitor-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab monitor-review grade
```

## Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab monitor-review finish
```

This concludes the lab.

# Summary

---

In this chapter, you have learned that:

- Quarkus enables metrics generation via the `quarkus-smallrye-metrics` extension. This extension implements the MicroProfile Metrics standard.
- The `/metrics`, `/metrics/base`, `/metrics/vendor`, and `/metrics/application` endpoints expose metrics data in JSON and Prometheus format.
- Use the `@Counted`, `@Gauge`, and `@Metered` annotations to generate counter, gauge, and summary metrics respectively.
- Quarkus uses the `quarkus-smallrye-opentracing` extension to inject tracing information in Jaeger format in all service requests.
- Configuring that extension is enough for tracing all endpoints in the application. Use the `@Traced` annotation to start spans at method level.

## Chapter 10

# Comprehensive Review of Red Hat Cloud-native Microservices Development with Quarkus

### Goal

Review tasks from *Red Hat Cloud-native Microservices Development with Quarkus*

### Objectives

- Review tasks from *Red Hat Cloud-native Microservices Development with Quarkus*

### Sections

- Reviewing Microservice Development Concepts (and Lab)

### Lab

Develop and Deploy Microservices on OpenShift

# Comprehensive Review

---

## Objectives

After completing this section, you should be able to demonstrate knowledge and skills learned in *Red Hat Cloud-native Microservices Development with Quarkus*.

## Reviewing Red Hat Cloud-native Microservices Development with Quarkus

Before beginning the comprehensive review for this course, you should be comfortable with the topics covered in each chapter.

You can refer to earlier sections in the textbook for extra study.

### **Chapter 1, Describing a Microservice Architecture**

Describe components and patterns of microservice-based application architectures.

- Define what microservices are and the guiding principles for their creation.
- Describe the major patterns implemented in microservice architectures.

### **Chapter 2, Implementing a Microservice with Quarkus**

Develop Quarkus microservice applications.

- Describe the features of Quarkus.
- Implement a microservice using the CDI, JAX-RS, and JSON-P specifications of MicroProfile.
- Persist data with simplified Panache Entities.

### **Chapter 3, Deploying Microservice-based Applications**

Deploy Quarkus microservice applications to an OpenShift cluster.

- Describe the architecture and features of a microservice application.
- Use Red Hat OpenShift Container Platform to deploy microservices.
- Create container images for Quarkus applications and deploy them in OpenShift Container Platform using Quarkus extensions.

### **Chapter 4, Building Microservice Applications with Quarkus**

Build a persistent and configurable distributed Quarkus microservices application.

- Inject configuration data into a Quarkus microservice.
- Implement configurable feature toggles in microservices.
- Use service discovery to connect multiple services to each other.

## **Chapter 5, Implementing Fault Tolerance**

Implement fault tolerance in a microservice architecture.

- Apply fault tolerance policies to a Microservice.
- Implement a health check in a microservice and enable a probe in OpenShift to monitor it.

## **Chapter 6, Building and Deploying Native Quarkus Applications**

Describe Quarkus Native and its deployment on OpenShift Container Platform.

- Build native applications using Quarkus.
- Create container images for Quarkus native applications and deploy them in OpenShift Container Platform.

## **Chapter 7, Testing Microservices**

Implement unit and integration tests for microservices.

- Implement a quarkus microservice test case.
- Implement a microservice test using mock frameworks.

## **Chapter 8, Securing Microservices Communication**

Secure microservice communications by applying origin validation, requests authentication and authorization.

- Validate the origin of microservice communications by using TLS and CORS.
- Apply authentication and authorization techniques to Quarkus microservices.

## **Chapter 9, Monitoring Quarkus Microservices**

Monitor the operation of a microservice by using metrics and distributed tracing.

- Generate microservice metrics by using Micrometer.
- Generate request traces by using SmallRye MicroProfile OpenTracing.

## ► Lab

# Develop and Deploy Microservices on OpenShift

In this review, you will update an existing microservices application, create a new microservice, and add fault tolerance, security, and tracing capabilities.

## Outcomes

You should be able to:

- Create a new Quarkus microservice, validate it by using unit tests, and deploy it to OpenShift.
- Connect two microservices by using the REST protocol, applying fault tolerance techniques to enhance the application's resiliency.
- Protect microservice endpoints from unauthorized access by using RBAC and the OIDC flow.
- Generate and gather usage metrics for specific methods in a microservice.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this lab.

```
[student@workstation ~]$ lab comprehensive-review start
```

This command places the source code of the Exchange application in the `~/D0378/labs/comprehensive-review` folder of your workstation. It prepares the `frontend`, `history` and `news` microservices and a Keycloak application container acting as an OIDC server with a pre-configured realm.

## Specifications

The Exchange application consists of four services:

- The `Frontend` service web application, which serves as a visual entry point for the application. This service uses AJAX to perform REST requests to the `Exchange` service.
- The `Exchange` service acts as an API gateway for the application. It centralizes requests to the `History` and `News` services.
- The `History` service provides historical currency exchange values.
- The `News` service offers a random selection of news about several topics, financial news being the topic of interest for this exercise.

Pay special attention to port numbers for each service. Some services use command parameters to provide port numbers but some others use configuration entries.

## Services of the Exchange Application

Service	Technology	Port	Endpoints
Frontend	React.js	3000	• /frontend
Exchange	Quarkus	8080	• /exchangeRate/historicalData (POST) • /exchangeRate/singleCurrency (POST) • /currencies • /news
History	Node.js	8082	• / (POST)
News	Python	6000	• /news/{topic}

The Exchange gateway uses three hard-coded currencies. You have been instructed to add a new microservice to provide and manage more currencies.

The new service, named **currency**, must be a Quarkus REST microservice with the following requirements:

- The **currency** service must use the following port numbers:
  - 5000 for HTTP requests in production and development mode.
  - 5001 for HTTP requests in test mode.
  - 5100 for secure HTTPS requests in production and development mode.
  - 5101 for secure HTTPS requests in test mode.
- The **currency** service must expose a GET REST endpoint at the /{currency} path. This endpoint returns the currency in JSON format including the currency name and its sign. For example, a GET request to the /USD path must return `{"name": "USD", "sign": "$"}`.
- The **currency** service must expose a GET REST endpoint at the root path (/) providing a list with the 3-letter code for all currencies available. The following currencies must be available: AUD, CAD, CHF, EUR, GBP, JPY, and USD.
- The **currency** service must expose a DELETE REST endpoint at the /{currency} path. This endpoint deletes the currency from the service, so it does not return the currency anymore. Endpoints for creating or updating currencies are out of scope for this exercise.
- The **currency** service must use a H2 database to store currencies. For the purpose of this exercise an in-memory database can be used. Use the following URL to connect to the test database: `jdbc:h2:mem:currency;DB_CLOSE_DELAY=-1`. You can use the `~/DO378/labs/comprehensive-review/currencies.sql` file for initializing the database contents.
- The code for the **currency** service must include at least one test suite, named `org.redhat.currency.CurrencyResourceTest`, with at least one test. All tests must succeed. Test contents are not relevant but you can use the following line to create a test, which checks the service provides the required currency names:

```
Stream.of("AUD", "CAD", "CHF", "EUR", "GBP", "JPY", "USD")
 .forEach(ccy->RestAssured.when().get("/"+ccy)
 .then().statusCode(200).body("name", is(ccy))
```

- The test suite must provide a test H2 database and not use the database embedded in the service. In test mode, the currency service must use the database provided by the test. Use the following URL to connect to the test database: `jdbc:h2:tcp://localhost/~currency`.
- You must provide liveness and readiness probes for the currency service. As the service is only relying on the database, probes will return positive results when the database is alive and ready.
- The `DELETE` endpoint in the currency service must be protected from unauthorized access. Only authenticated users with the role `confidential` can delete currencies. You must use an OIDC authentication flow to authenticate users.

You are provided with a script that starts a Keycloak container image with a configured realm. Use the following data to connect to the Keycloak server:

Server and realm URL	<code>http://localhost:8081/auth/realms/quarkus</code>
Client ID	<code>backend-service</code>
Client Secret	<code>secret</code>

The pre-configured realm contains the following users and roles:

User name	Password	Roles
jdoe	jdoe	user, confidential
alice	alice	user
admin	admin	user, admin

You can use the provided `~DO378/labs/comprehensive-review/keycloak-token.sh` script to obtain the token from the Keycloak server. The following snippet is an example set of commands to retrieve a token for the user `jdoe` and use it in a `curl` command:

```
[student@workstation comprehensive-review]$ TOKEN=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service jdoe jdoe secret)
[student@workstation comprehensive-review]$ echo $TOKEN
eyJH...FcTA
[student@workstation comprehensive-review]$ curl -D - -X DELETE -k \
https://mysecureservice:446 -H "Authorization: Bearer $TOKEN"
HTTP/1.1 204 No Content
```

- The currency service must respond only to secure HTTPS requests. You can use insecure ports (5000 and 5001) during development but the final version must only respond to secure ports (5100 and 5101).

The lab start script created the needed keystore, certificate and truststore for the currency service. Those are placed in the ~/D0378/labs/comprehensive-review folder with the names `currency.keystore`, `currency.crt`, and `currency.truststore` respectively. The password for both the `keystore` and `truststore` files is `redhat`. Copy those files into the appropriate services project folders to include them into the service package.



### Note

MicroProfile REST client configuration uses a dedicated notation to load files from the application classpath: `com.redhat.exchange.service.CurrencyService/mp-rest/trustStore=classpath:/currency.truststore`. Also, remember that Quarkus adds files in the `src/main/resources` folder to the application package and makes them available in the classpath.

The exchange service is still not ready to enable mutual TLS, so the currency service must accept unauthenticated clients.

- The currency service must be containerized. Create a container image named `localhost/student/currency:1.0.0-SNAPSHOT` and start it locally. The container name must be `currency`. Remember that the application inside the container can not use `localhost` to reach other services running on the same workstation. A solution for this is to provide the application with the primary IP for the host. The following command sets environment variables to the container with the appropriate values:

```
[student@workstation ~]$ podman run --name currency \
-e OIDC_HOST=$(hostname -i) -e OIDC_PORT=8081 \
-p 5000:5000 localhost/student/currency:1.0.0-SNAPSHOT
```

When the currency service is ready, update the exchange gateway service to use it:

- Setup a REST client interface to connect to the currency service. The interface class must be `com.redhat.exchange.service.CurrencyService`. The client does not need to include the `DELETE` endpoint, but it does need to expose the method for retrieving the name and sign for a single currency and the method for retrieving all currency names. Remember to add the needed configuration for the REST client to find the service. Creating a DTO class to hold data returned by the currency service is not mandatory but is recommended.
- The `CurrencyResource` class in the exchange service is currently providing default values for both the currency names and signs. Update this class to use the REST client for the currency service instead. Do not delete the existing methods but rename them.
- Create a feature flag named `feature.external_currencies`. When this flag is set to true, use the REST client to retrieve the currencies from the newly created service. When the flag is false or unset, use the default values returned by the renamed methods.
- The currency service is still on probation. The exchange service must use the old default values if the currency service fails, is unavailable, or takes more than one second to respond. Note that this behavior only applies to the `/currencies` endpoint. Other methods, such as `/singleCurrency` can keep its current behavior.
- The exchange service must generate count and time metrics for all four endpoints. Generate another metric, which counts the number of times the application obtains the

sign of a currency. In other words, how many times the `getCurrencySign` method is used. The name of the metric must be `currencysign_count`.

All metrics must be served in the Prometheus format.

## Starting Available Services

Start the dependent services so you can perform an end-to-end validation of your application. Keep it running for the duration of this lab.

```
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/start-services.sh
```

If you need to run any service independently then execute the following appropriate command in a separate terminal

```
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/frontend/start.sh
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/news/start.sh
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/history/start.sh
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/start-keycloak.sh
```

## Instructions

1. Create the new `currency` service.
2. Create a meaningful test for the `currency` service.
3. Add health checks to the `currency` service.
4. Use an OIDC flow to restrict access to the `DELETE` endpoint.
5. Enable SSL communications for the `currency` service.
6. Create and start a container for the `currency` service.
7. Update the `exchange` service to communicate with the `currency` service. Configure the `feature.external_currencies` feature flag to alternate between default and external currencies.
8. Protect the `getCurrencyNames` method from a failing `currency` service.
9. Enable metrics generation on the `exchange` service. Expose metrics in Prometheus format. Add a metric that counts usage of the method `CurrencyResource.getCurrencyNames`. The metric must be named `currencysign.count`.

## Evaluation

Make sure all applications and containers are running: `frontend`, `news`, `history`, and `Keycloak` from the provided scripts, `currency` as the generated container, and `exchange` in Quarkus development mode.

Grade your work by running the `lab comprehensive-review grade` command from your `workstation` machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab comprehensive-review grade
```

## Finish

Stop all running applications and containers. In each terminal, press **Ctrl+C** to stop the application, and then close the terminal. Close all VSCode windows already open.

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab comprehensive-review finish
```

This concludes the lab.

## ► Solution

# Develop and Deploy Microservices on OpenShift

In this review, you will update an existing microservices application, create a new microservice, and add fault tolerance, security, and tracing capabilities.

## Outcomes

You should be able to:

- Create a new Quarkus microservice, validate it by using unit tests, and deploy it to OpenShift.
- Connect two microservices by using the REST protocol, applying fault tolerance techniques to enhance the application's resiliency.
- Protect microservice endpoints from unauthorized access by using RBAC and the OIDC flow.
- Generate and gather usage metrics for specific methods in a microservice.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this lab.

```
[student@workstation ~]$ lab comprehensive-review start
```

This command places the source code of the Exchange application in the `~/D0378/labs/comprehensive-review` folder of your workstation. It prepares the `frontend`, `history` and `news` microservices and a Keycloak application container acting as an OIDC server with a pre-configured realm.

## Specifications

The Exchange application consists of four services:

- The `Frontend` service web application, which serves as a visual entry point for the application. This service uses AJAX to perform REST requests to the `Exchange` service.
- The `Exchange` service acts as an API gateway for the application. It centralizes requests to the `History` and `News` services.
- The `History` service provides historical currency exchange values.
- The `News` service offers a random selection of news about several topics, financial news being the topic of interest for this exercise.

Pay special attention to port numbers for each service. Some services use command parameters to provide port numbers but some others use configuration entries.

## Services of the Exchange Application

Service	Technology	Port	Endpoints
Frontend	React.js	3000	• /frontend
Exchange	Quarkus	8080	• /exchangeRate/historicalData (POST) • /exchangeRate/singleCurrency (POST) • /currencies • /news
History	Node.js	8082	• / (POST)
News	Python	6000	• /news/{topic}

The Exchange gateway uses three hard-coded currencies. You have been instructed to add a new microservice to provide and manage more currencies.

The new service, named **currency**, must be a Quarkus REST microservice with the following requirements:

- The **currency** service must use the following port numbers:
  - 5000 for HTTP requests in production and development mode.
  - 5001 for HTTP requests in test mode.
  - 5100 for secure HTTPS requests in production and development mode.
  - 5101 for secure HTTPS requests in test mode.
- The **currency** service must expose a GET REST endpoint at the /{currency} path. This endpoint returns the currency in JSON format including the currency name and its sign. For example, a GET request to the /USD path must return `{"name": "USD", "sign": "$"}`.
- The **currency** service must expose a GET REST endpoint at the root path (/) providing a list with the 3-letter code for all currencies available. The following currencies must be available: AUD, CAD, CHF, EUR, GBP, JPY, and USD.
- The **currency** service must expose a DELETE REST endpoint at the /{currency} path. This endpoint deletes the currency from the service, so it does not return the currency anymore. Endpoints for creating or updating currencies are out of scope for this exercise.
- The **currency** service must use a H2 database to store currencies. For the purpose of this exercise an in-memory database can be used. Use the following URL to connect to the test database: `jdbc:h2:mem:currency;DB_CLOSE_DELAY=-1`. You can use the `~/D0378/labs/comprehensive-review/currencies.sql` file for initializing the database contents.
- The code for the **currency** service must include at least one test suite, named `org.redhat.currency.CurrencyResourceTest`, with at least one test. All tests must succeed. Test contents are not relevant but you can use the following line to create a test, which checks the service provides the required currency names:

```
Stream.of("AUD", "CAD", "CHF", "EUR", "GBP", "JPY", "USD")
 .forEach(ccy->RestAssured.when().get("/"+ccy)
 .then().statusCode(200).body("name", is(ccy))
```

- The test suite must provide a test H2 database and not use the database embedded in the service. In test mode, the currency service must use the database provided by the test. Use the following URL to connect to the test database: `jdbc:h2:tcp://localhost/~currency`.
- You must provide liveness and readiness probes for the currency service. As the service is only relying on the database, probes will return positive results when the database is alive and ready.
- The `DELETE` endpoint in the currency service must be protected from unauthorized access. Only authenticated users with the role `confidential` can delete currencies. You must use an OIDC authentication flow to authenticate users.

You are provided with a script that starts a Keycloak container image with a configured realm. Use the following data to connect to the Keycloak server:

Server and realm URL	<code>http://localhost:8081/auth/realms/quarkus</code>
Client ID	<code>backend-service</code>
Client Secret	<code>secret</code>

The pre-configured realm contains the following users and roles:

User name	Password	Roles
jdoe	jdoe	user, confidential
alice	alice	user
admin	admin	user, admin

You can use the provided `~DO378/labs/comprehensive-review/keycloak-token.sh` script to obtain the token from the Keycloak server. The following snippet is an example set of commands to retrieve a token for the user `jdoe` and use it in a `curl` command:

```
[student@workstation comprehensive-review]$ TOKEN=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service jdoe jdoe secret)
[student@workstation comprehensive-review]$ echo $TOKEN
eyJH...FcTA
[student@workstation comprehensive-review]$ curl -D - -X DELETE -k \
https://mysecureservice:446 -H "Authorization: Bearer $TOKEN"
HTTP/1.1 204 No Content
```

- The currency service must respond only to secure HTTPS requests. You can use insecure ports (5000 and 5001) during development but the final version must only respond to secure ports (5100 and 5101).

The lab start script created the needed keystore, certificate and truststore for the currency service. Those are placed in the ~/D0378/labs/comprehensive-review folder with the names `currency.keystore`, `currency.crt`, and `currency.truststore` respectively. The password for both the `keystore` and `truststore` files is `redhat`. Copy those files into the appropriate services project folders to include them into the service package.



### Note

MicroProfile REST client configuration uses a dedicated notation to load files from the application classpath: `com.redhat.exchange.service.CurrencyService/mp-rest/trustStore=classpath:/currency.truststore`. Also, remember that Quarkus adds files in the `src/main/resources` folder to the application package and makes them available in the classpath.

The exchange service is still not ready to enable mutual TLS, so the currency service must accept unauthenticated clients.

- The currency service must be containerized. Create a container image named `localhost/student/currency:1.0.0-SNAPSHOT` and start it locally. The container name must be `currency`. Remember that the application inside the container can not use `localhost` to reach other services running on the same workstation. A solution for this is to provide the application with the primary IP for the host. The following command sets environment variables to the container with the appropriate values:

```
[student@workstation ~]$ podman run --name currency \
-e OIDC_HOST=$(hostname -i) -e OIDC_PORT=8081 \
-p 5000:5000 localhost/student/currency:1.0.0-SNAPSHOT
```

When the currency service is ready, update the exchange gateway service to use it:

- Setup a REST client interface to connect to the currency service. The interface class must be `com.redhat.exchange.service.CurrencyService`. The client does not need to include the `DELETE` endpoint, but it does need to expose the method for retrieving the name and sign for a single currency and the method for retrieving all currency names. Remember to add the needed configuration for the REST client to find the service. Creating a DTO class to hold data returned by the currency service is not mandatory but is recommended.
- The `CurrencyResource` class in the exchange service is currently providing default values for both the currency names and signs. Update this class to use the REST client for the currency service instead. Do not delete the existing methods but rename them.
- Create a feature flag named `feature.external_currencies`. When this flag is set to true, use the REST client to retrieve the currencies from the newly created service. When the flag is false or unset, use the default values returned by the renamed methods.
- The currency service is still on probation. The exchange service must use the old default values if the currency service fails, is unavailable, or takes more than one second to respond. Note that this behavior only applies to the `/currencies` endpoint. Other methods, such as `/singleCurrency` can keep its current behavior.
- The exchange service must generate count and time metrics for all four endpoints. Generate another metric, which counts the number of times the application obtains the

sign of a currency. In other words, how many times the `getCurrencySign` method is used. The name of the metric must be `currencysign_count`.

All metrics must be served in the Prometheus format.

## Starting Available Services

Start the dependent services so you can perform an end-to-end validation of your application. Keep it running for the duration of this lab.

```
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/start-services.sh
```

If you need to run any service independently then execute the following appropriate command in a separate terminal

```
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/frontend/start.sh
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/news/start.sh
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/history/start.sh
[student@workstation ~]$ ~/D0378/labs/comprehensive-review/start-keycloak.sh
```

## Instructions

1. Create the new currency service.

- 1.1. In a new terminal, move to the lab folder and create a new Quarkus project named currency. Open the project with VSCode.

```
[student@workstation ~]$ cd ~/D0378/labs/comprehensive-review
[student@workstation comprehensive-review]$ mvn \
io.quarkus:quarkus-maven-plugin:1.11.7.Final-redhat-00009:create \
-DprojectGroupId=org.redhat \
-DprojectArtifactId=currency \
-DplatformGroupId=com.redhat.quarkus \
-DplatformVersion=1.11.7.Final-redhat-00009 \
-DclassName="org.redhat.currency.CurrencyResource" \
-Dextensions=quarkus-hibernate-orm-panache,quarkus-jdbc-h2,quarkus-resteasy-jsonb
...output omitted...
[student@workstation comprehensive-review]$ cd currency
[student@workstation currency]$ codium .
```

Update the `application.properties` file to set the application ports as defined in the requirements:

```
quarkus.http.port=5000
quarkus.http.test-port=5001
```

Create the `Currency` entity. You can use the same class for defining the persistence entity and the DTO. Doing so is not a best practice. However, we are using this approach here for the sake of simplicity. Use the example provided so, when the entity is serialized, it matches the expected format: `{"name": "USD", "sign": "$"}`

```
@Entity
public class Currency extends PanacheEntityBase {
 @Id
 public String name;
 public String sign;
}
```

Configure the data source for the service. As required, the kind of database must be an in-memory h2. You can copy the `~/D0378/labs/comprehensive-review/currencies.sql` file to the `src/main/resources` folder and use it to initialize the database, or you can create the initialization script manually, as long as it contains the required currencies.

```
...output omitted...
quarkus.datasource.db-kind=h2
quarkus.datasource.jdbc.url=jdbc:h2:mem:currencies;DB_CLOSE_DELAY=-1
quarkus.hibernate-orm.database.generation=create
quarkus.hibernate-orm.sql-load-script=currencies.sql
```

Replace the sample contents in the `CurrencyResource` with a resource class, which implements, at least, the requested endpoints.

```
@Path("/")
@Produces(MediaType.APPLICATION_JSON)
public class CurrencyResource {

 @DELETE @Path("/{id}")
 @Transactional
 public void delete(@PathParam("id") String id) {
 if (!Currency.deleteById(id)) throw new NotFoundException();
 }

 @GET @Path("/{id}")
 @Transactional
 public Currency getCurrency(@PathParam("id") String id) {
 return Currency.<Currency>findByIdOptional(id)
 .orElseThrow(NotFoundException::new);
 }

 @GET
 @Transactional
 public List<String> getCurrencyNames() {
 return Currency.<Currency>streamAll()
 .map(c -> c.name).collect(Collectors.toList());
 }
}
```

Quarkus serves the static resource file `src/main/resources/META-INF/resources/index.html` in the root path before delegating to the endpoints in the

CurrencyResource class. Delete this file and the whole META-INF folder to obtain the expected results when performing requests to the root path.

This is enough for the service to start. You can start it with the `./mvnw quarkus:dev` command, and test it with a `curl localhost:5000` command. Keep the application running to ease next steps.

2. Create a meaningful test for the currency service.
  - 2.1. Use the `quarkus-test-h2` extension to create a test database. Add the extension by using the VSCode Quarkus extension or by using the following command:

```
[student@workstation currency]$./mvnw \
 quarkus:add-extension -Dextension=io.quarkus:quarkus-test-h2
...output omitted...
Extension io.quarkus:quarkus-test-h2 has been installed
...output omitted...
```

- 2.2. Configure the currency service so it uses the test database when running in test mode. Add the following entries to the `application.properties` file:

```
%test.quarkus.hibernate-orm.database.generation=drop-and-create
%test.quarkus.datasource.jdbc.url=jdbc:h2:tcp://localhost/~currency
```

- 2.3. The generated code already includes a Quarkus test, but it is meaningless for the currency service. Replace it with a meaningful test, which validates the required currencies are available:

```
@Test
public void defaults_are_present() {
 Stream.of("AUD", "CAD", "CHF", "EUR", "GBP", "JPY", "USD").forEach(ccy->
 RestAssured.when().get("/"+ccy)
 .then().statusCode(200)
 .body("name", is(ccy))
);
}
```

Add the `@QuarkusTestResource(H2DatabaseTestResource.class)` annotation to the test so it spins up a new database for testing purposes.

```
...output omitted...
@QuarkusTest
@QuarkusTestResource(H2DatabaseTestResource.class)
public class CurrencyResourceTest {
...output omitted...
```

- 2.4. Run the tests in JVM mode and acknowledge that the test suite passes. Use the `mvnw verify` to validate the tests.

```
[student@workstation currency]$./mvnw clean verify
...output omitted...
[INFO]
[INFO] Results:
[INFO]
```

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```



### Note

Other approaches to run the test suite, such as the VSCode Test Runner for Java extension, might also try to run the `NativeCurrencyResourceIT` test. This test is not intended to succeed at this point, so test failures for this test can be safely ignored.

### 3. Add health checks to the currency service.

- 3.1. Use the `smallrye-health` extension to add health checks to the service. Add the extension by using the VSCode Quarkus extension or by using the following command:

```
[student@workstation currency]$./mvnw \
quarkus:add-extension -Dextension=smallrye-health
...output omitted...
Extension io.quarkus:quarkus-smallrye-health has been installed
...output omitted...
```

Verify the health endpoint is now available:

```
[student@workstation currency]$ curl localhost:5000/q/health
{
 "status": "UP",
 "checks": [
 {
 "name": "Database connections health check",
 "status": "UP"
 }
]
}
```

### 4. Use an OIDC flow to restrict access to the DELETE endpoint.

- 4.1. To protect the `DELETE` endpoint from unauthorized access you can use the `quarkus-oidc` extension. Add the extension by using the VSCode Quarkus extension or by using the following command:

```
[student@workstation currency]$./mvnw \
quarkus:add-extension -Dextensions="quarkus-oidc"
...output omitted...
Extension io.quarkus:quarkus-oidc has been installed
...output omitted...
```

Configure the OIDC extension to connect to the Keycloak server. Add the following entries to the `application.properties` file.

```
quarkus.oidc.auth-server-url=http://:${OIDC_HOST}:localhost}:${OIDC_PORT}:8081}/auth/
realms/quarkus
quarkus.oidc.client-id=backend-service
quarkus.oidc.credentials.secret=secret
```

Using environment variables in the configuration eases running the application in a container.

- 4.2. Add the `RolesAllowed` annotation to the `delete` method. Provide the only role allowed `confidential` to the annotation.

```
...output omitted...
public class CurrencyResource {
...output omitted...
 @RolesAllowed("confidential")
 @DELETE @Path("/{id}")
 @Transactional
 public void delete(@PathParam("id") String id) {
...output omitted...
```

- 4.3. Validate that only the user `jdoe` can delete currencies.

```
[student@workstation currency]$ curl -D - localhost:5000/USD; echo
HTTP/1.1 200 OK
Content-Length: 25
Content-Type: application/json

{"name":"USD","sign":""}
[student@workstation currency]$ curl -D - -X DELETE localhost:5000/USD
HTTP/1.1 401 Unauthorized
Content-Length: 0

[student@workstation currency]$ TOKEN=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service alice alice secret)
realm: quarkus
client-id: backend-service
username: alice
Token succesfully retrieved.
[student@workstation currency]$ curl -D - -X DELETE \
-H "Authorization: Bearer $TOKEN" \
localhost:5000/USD
HTTP/1.1 403 Forbidden
Content-Length: 0

[student@workstation currency]$ TOKEN=$(./keycloak-token.sh \
localhost:8081 quarkus backend-service jdoe jdoe secret)
realm: quarkus
client-id: backend-service
username: jdoe
Token succesfully retrieved.
[student@workstation currency]$ curl -D - -X DELETE \
```

```
-H "Authorization: Bearer $TOKEN" \
localhost:5000/USD
HTTP/1.1 204 No Content
```

5. Enable SSL communications for the currency service.

- 5.1. Configure the currency service to respond secure requests in the required ports. Disable HTTP responses from the service and configure the required certificate properties to enable HTTPS communication. Add the following entries in the `application.properties` file:

```
quarkus.http.ssl-port=5100
quarkus.http.test-ssl-port=5101
quarkus.http.insecure-requests=disabled
quarkus.http.ssl.certificate.key-store-file=currency.keystore
quarkus.http.ssl.certificate.key-store-password=redhat
```

To include the keystore into the service copy the `curreny.keystore` file from the `~/D0378/labs/comprehensive-review` folder to the `src/main/resources` folder.

```
[student@workstation currency]$ cp/currency.keystore src/main/resources/
```

To allow unauthenticated clients, such as the exchange service, to use the service, add the `quarkus.http.ssl.client-auth` entry to the same configuration file.

```
quarkus.http.ssl.client-auth=none
```

Those changes disable HTTP responses, but break the tests already in place. The easiest approach to fix them is to allow insecure HTTP connections in test mode. Enable the `%test.quarkus.http.insecure-requests` entry in the `application.properties` configuration file.

```
%test.quarkus.http.insecure-requests=enabled
```



### Note

The previous changes impact the core components of the Quarkus framework, so you will need to restart your currency service to pick the changes.

Validate that the HTTP responses are not allowed anymore but HTTPS responses work as expected.

```
[student@workstation currency]$ curl http://localhost:5100/USD
curl: (52) Empty reply from server
[student@workstation currency]$ curl -v -k https://localhost:5100/USD
...output omitted...
* Server certificate:
* subject: CN=currency,OU=Training,O=RedHat,L=Raleigh,ST=NC,C=US
...output omitted...
> GET /USD HTTP/1.1
...output omitted...
```

```
< HTTP/1.1 200 OK
...
{"name": "USD", "sign": "$"}%
```

Validate the test created in the previous steps execute successfully.

## 6. Create and start a container for the currency service.

- 6.1. Use the `container-image-jib` extension to create the container requested for the service. Add the extension using the VSCode Quarkus extension or by using the following command:

```
[student@workstation currency]$./mvnw \
quarkus:add-extension -Dextension=container-image-jib
...
Extension io.quarkus:quarkus-container-image-jib has been installed
...
```

Create the container image by using the package Maven phase and providing the `Dquarkus.container-image.build=true` option.



### Note

This step requires the `podman-docker` package available in your workstation.

```
[student@workstation currency]$./mvnw package \
-Dquarkus.container-image.build=true
[INFO] Scanning for projects...
...
[INFO] [io.quarkus.container.image.jib.deployment.JibProcessor] Created container
image student/currency:1.0.0-SNAPSHOT (sha256:48a7...8b72)
...
[INFO] BUILD SUCCESS
...
```

Start the container image you just created in a new terminal. You must pass the `OIDC_HOST` and `OIDC_PORT` environment variables used in the service configuration. You also must expose the port served by the application. Remember to stop the Quarkus application in development mode, if it is still running, to avoid port collisions.

```
[student@workstation ~]$ podman run \
-e OIDC_HOST=$(hostname -i) -e OIDC_PORT=8081 \
-p 5100:5100 localhost/student/currency:1.0.0-SNAPSHOT
```

Validate the containerized application is serving requests. Execute the following command in a new terminal window:

```
[student@workstation ~]$ curl -k https://localhost:5100/USD ; echo
{"name": "USD", "sign": "$"}
```

Keep the container running.

7. Update the exchange service to communicate with the currency service. Configure the `feature.external_currencies` feature flag to alternate between default and external currencies.

- 7.1. Open the `~/D0378/labs/comprehensive-review/exchange` folder with VSCode. Create the `Currency` DTO class. It must be coherent with the currency entity used in the currency service.

```
package com.redhat.exchange.dto;

public class Currency {
 public String sign;
 public String name;
}
```

- 7.2. Create the `CurrencyService` interface, which defines the REST client to the currency service. You only need to add the two GET methods declared in the service.

```
@Path("/")
@RegisterRestClient
public interface CurrencyService {

 @GET
 @Produces(MediaType.APPLICATION_JSON)
 List<String> getCurrencyNames();

 @GET
 @Path("/{name}")
 @Produces(MediaType.APPLICATION_JSON)
 Currency getCurrency(@PathParam("name") String name);
}
```

- 7.3. Add the configuration for the REST client to connect to the service. Add the following line to the `application.properties` file:

```
com.redhat.exchange.service.CurrencyService/mp-rest/url=https://
${CURRENCY_ENDPOINT:localhost}:${CURRENCY_PORT:5100}
```

Copy the trust store generated by the lab start script into the `src/main/resources` folder.

```
[student@workstation currency]$ cd ../exchange
[student@workstation exchange]$ cp ../currency.truststore src/main/resources/
```

- 7.4. Configure the rest client to use the trust store and provide the password `redhat`.

```
com.redhat.exchange.service.CurrencyService/mp-rest/trustStore=classpath:/
currency.truststore
com.redhat.exchange.service.CurrencyService/mp-rest/trustStorePassword=redhat
```

- 7.5. Create in the configuration file a feature toggle entry which controls if the application uses the currency REST client or it relies on the default values.

```
feature.external_currencies=false
```

- 7.6. Inject both the REST client and the feature flag into the CurrencyResource.

```
...output omitted...
public class CurrencyResource {

 @ConfigProperty(name = "feature.external_currencies", defaultValue = "false")
 boolean externalCurrencies;

 @Inject
 @RestClient
 CurrencyService currencies;
 ...output omitted...
```

- 7.7. Update the getCurrencyNames and getCurrencySign methods to make use of the feature flag and the REST client. One way to do this is by creating two new methods getDefaultCurrencyNames and getDefaultCurrencySign returning the default values. Then rewrite the old methods to delegate to the new methods or to the REST client based on the feature flag.

```
...output omitted...
public class CurrencyResource {
 ...output omitted...
 @GET
 public List<String> getCurrencyNames() {
 if (externalCurrencies)
 return currencies.getCurrencyNames();
 else
 return getDefaultCurrencyNames();
 }

 public String getCurrencySign(final String currencyName){
 if (externalCurrencies)
 return currencies.getCurrency(currencyName).sign;
 else
 return getDefaultCurrencySign(currencyName);
 }

 public List<String> getDefaultCurrencyNames() {
 return List.of("EUR", "USD", "JPY");
 }

 public String getDefaultCurrencySign(String currencyName) {
 switch (currencyName){
 case "EUR": return "€";
 case "USD": return "$";
 case "JPY": return "¥";
 }
 return null;
 }
}
```

- 7.8. Validate the changes are working as expected. Start the application in a new terminal by using the `mvn quarkus:dev` command. Then, perform a request to the currencies endpoint, and confirm you are having only the default currencies.

```
[student@workstation exchange]$ curl -s localhost:8080/currencies | jq
[
 "EUR",
 "USD",
 "YEN"
]
```

- 7.9. Enable the feature flag by setting the `feature.external_currencies` entry to `true` in the `application.properties` file. Repeat the previous request. Now the exchange service returns all currencies provided by the currency service.

```
[student@workstation exchange]$ curl -s localhost:8080/currencies | jq
[
 "AED",
 "AFN",
 "ALL",
 ...output omitted...
 "YER",
 "ZAR",
 "ZMW"
]
```

8. Protect the `getCurrencyNames` method from a failing currency service.

- 8.1. Add the `quarkus-smallrye-fault-tolerance` extension to the service to enable fault tolerance features. Add the extension by using the VSCode Quarkus extension or by using the following command:

```
[student@workstation exchange]$./mvnw \
 quarkus:add-extension -Dextension=smallrye-fault-tolerance
 ...output omitted...
Extension io.quarkus:quarkus-smallrye-fault-tolerance has been installed
 ...output omitted...
```

Use the just created `getDefauleCurrencyNames` method as a fallback method when the currency service is not available or failing. Add the `Fallback` annotation to the `getCurrencyName` method to define the fallback. Use the `Timeout` annotation to protect the method from a degraded currency service. Default values for the `Timeout` annotation match the requirements.

```
...output omitted...
public class CurrencyResource {
 ...output omitted...
 @GET
 @Fallback(fallbackMethod = "getDefauleCurrencyNames")
 @Timeout
 public List<String> getCurrencyNames() {
```

```
...output omitted...
public List<String> getDefaultCurrencyNames() {
...output omitted...
```

- 8.2. Validate that the exchange service is resilient to failures of the currency service.

Shut down the currency service by pressing **Ctrl+C** in the terminal running the service container. Make sure the exchange service has the `external_currencies` feature enabled.

Repeat the last request you performed to the `/currencies` endpoint for the exchange service. Because the dependency is not available Quarkus uses the fallback method to respond.

```
[student@workstation ~]$ curl -s localhost:8080/currencies | jq
[
 "EUR",
 "USD",
 "YEN"
]
```

Start the currency container back.

```
[student@workstation ~]$ podman run \
-e OIDC_HOST=$(hostname -i) -e OIDC_PORT=8081 \
-p 5100:5100 localhost/student/currency:1.0.0-SNAPSHOT
```

Repeat the previous request to see the whole set of currencies:

```
[student@workstation ~]$ curl -s localhost:8080/currencies | jq
[
 "AED",
 "AFN",
 "ALL",
 ...output omitted...
 "YER",
 "ZAR",
 "ZMW"
]
```

9. Enable metrics generation on the exchange service. Expose metrics in Prometheus format. Add a metric that counts usage of the method `CurrencyResource.getCurrencyNames`. The metric must be named `currencysign.count`.

- 9.1. Add the `quarkus-micrometer-registry-prometheus` extension to the project. Add the extension by using the VSCode Quarkus extension or by using the following command:

```
[student@workstation exchange]$./mvnw \
quarkus:add-extension -Dextension=micrometer-registry-prometheus
...output omitted...
Extension io.quarkus:quarkus-micrometer-registry-prometheus has been installed
...output omitted...
```

This extension automatically generates the default metrics for all endpoints but not for the `getCurrencySign` method. To do so, inject the Micrometer `MeterRegistry` into the `CurrencyResource`. In the `getCurrencySign` method use the resource to fetch and increment a counter named `currencysign.count`.

```
...output omitted...
public class CurrencyResource {
 @Inject
 io.micrometer.core.instrument.MeterRegistry registry;
 ...output omitted...

 public String getCurrencySign(final String currencyName){
 registry.counter("currencysign.count").increment();
 }
 ...output omitted...
```

Validate the generation of metrics. Open the frontend application at `localhost:3000` and navigate to the **Exchange** section. Run some random requests to generate the metrics. Then execute the following command in your terminal to see the generated metrics:

```
[student@workstation exchange]$ curl localhost:8080/q/metrics
...output omitted...
ft_com_redhat_exchange_CurrencyResource_getCurrencyNames_invocations_total_none_total{scope="application",} 4.0
...output omitted...
currencysign_count_total 4.0
...output omitted...
http_server_requests_seconds_count{method="GET",outcome="SUCCESS",status="200",uri="/currencies",} 4.0
http_server_requests_seconds_sum{method="GET",outcome="SUCCESS",status="200",uri="/currencies",} 0.995493069
...output omitted...
```

Your actual output might differ, but you will see references to the `getCurrencyNames` method and the `currencysign.count` counter.



### Note

Sometimes live code reloading fails to generate the new application endpoints. This causes the `/q/metrics` endpoint to show as not found. If you get no response for the previous `curl` command try to stop the live coding session of the `exchange` service and start it again.

## Evaluation

Make sure all applications and containers are running: `frontend`, `news`, `history`, and `Keycloak` from the provided scripts, `currency` as the generated container, and `exchange` in Quarkus development mode.

Grade your work by running the `lab comprehensive-review grade` command from your `workstation` machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab comprehensive-review grade
```

## Finish

Stop all running applications and containers. In each terminal, press **Ctrl+C** to stop the application, and then close the terminal. Close all VSCode windows already open.

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab comprehensive-review finish
```

This concludes the lab.