

NEBRASS LAMOUCHI

PLAYING WITH JAVA MICROSERVICES WITH QUARKUS AND KUBERNETES



Hands-on guide for the absolute beginner

PLAYING WITH JAVA MICROSERVICES WITH QUARKUS AND KUBERNETES

Nebrass Lamouchi

Table of Contents

Dedication	2
Acknowledgements	3
Preface	4
What this book covers	5
Reader feedback	6
Introduction	7
Getting started with Containerization	9
Introduction to containerization	10
Introducing Docker	11
Installation and first hands-on	12
Docker Architecture	16
Diving into Docker Containers	19
Meeting the Docker Services	31
Achieving more with Docker	34
Containerization is not Docker only	39
Conclusion	42
Introduction to the Monolithic architecture	43
Introduction to an actual situation	44
Presenting the context	45
How to solve these issues ?	45
Coding the Monolithic application	46
Presenting our domain	47
Coding the application	49
Conclusion	105
Upgrading the Monolithic application	106
Implementing QuarkuShop Tests	107
Building and Running QuarkuShop	129
Conclusion	144
Building & Deploying the Monolithic application	145
Introduction	146
Importing the Project in Azure DevOps	146
Creating the CI/CD pipelines	149
Conclusion	170
Adding the anti-disasters layers	171
Introduction	172
Implementing the Security Layer	172
Implementing the Monitoring Layer	215
Conclusion	225
Microservices Architecture Pattern	226
Introduction	227
Microservices Architecture	228

Making the Switch	229
Splitting the Monolith: Bombarding the domain	230
Introduction	231
What is Domain-Driven Design ?	231
Bombarding QuarkuShop	233
Dependencies and Commons	233
Entities	233
Refactoring Databases	235
Transactional Boundaries	235
Summary	237
Applying DDD to the code	238
Introduction	239
Applying Bounded Contexts to Java Packages	239
Locating & breaking the BC Relationships	241
Meeting the microservices concerns and patterns	248
Introduction	249
Cloud Patterns	249
What's next?	256
Getting started with Kubernetes	257
Introduction	258
What is Kubernetes ?	258
Run Kubernetes locally	268
Practical Summary & Conclusion	270
Additional reading	271
Implementing the Cloud Patterns	273
Introduction	274
Bringing the Monolithic Universe to Kubernetes	274
Conclusion	292
Building the Kuberntized Microservices	293
Introduction	294
Creating the Commons Library	294
Implementing the Product µservice	298
Implementing the Order µservice	305
Implementing the Customer µservice	314
Implementing the User µservice	316
Conclusion	317
Flying all over the Sky with Quarkus and Kubernetes	318
Introduction	319
Implementing the Circuit Breaker pattern	319
Implementing the Log Aggregation pattern	323
Implementing the Distributed Tracing pattern	333
Implementing the API Gateway pattern	342
Conclusion	345

Playing with Quarkus in Azure	346
Bringing Dapr into the game	348
Final words & thoughts	350
About the author	351

Playing with Java Microservices with Quarkus and Kubernetes

© 2020 Nebrass Lamouchi. All rights reserved. Version 1.0.0-SNAPSHOT.

Version Date: 2020-10-21

Published by Nebrass Lamouchi, nibrass@gmail.com

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recoding, scanning or otherwise except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

Dedication

To Mom, it's impossible to thank you adequately for everything you've done.

To the soul of Dad, I miss you every day...

To Firass, you are the best, may God bless you...

To my sweetheart, since you've come into my life, there are so many new emotions and feelings begging to come out...

Acknowledgements

I would like to express my gratitude to the many people who saw me through this book.

To all those who provided support, read, wrote, offered comments and assisted in the editing and design.

To my wife and my family thank you for all your great support and encouragements.

To Laurent Ellerbach, thank you for all your support and guidance !

Last and not least, I beg forgiveness of all those who have been with me over the course of the years and whose names I have failed to mention.

Preface

Playing with Java Microservices with Quarkus and Kubernetes will teach you how to build and design microservices using **Java** and the **Quarkus platform**.

This book covers topics related to developing **Java microservices** and deploying them to **Kubernetes**.

Traditionally, **Java** developers have been used to developing large, complex monolithic applications. The experience of developing and deploying monoliths has been always slow and painful. This book will help **Java** developers to quickly get started with the features and the concerns of the **microservices architecture**. It will introduce **Docker** and **Kubernetes** to help them for the **microservices** deployment in the cloud.

The book is made for **Java** developers who want to build **microservices** using the new stack and who wants to deploy them to **Kubernetes**.

You will be guided on how to install the appropriate tools to work properly. For those who are new to **Enterprise Development** using **Quarkus**, you will be introduced to its core principles and main features thru a deep step-by-step tutorial on many components. For experts, this book offers some recipes that illustrate how to **split monoliths** and **implement microservices** and **deploy them as containers to Kubernetes**.

The following are some of the key challenges that we will address in this book:

- Introducing containerization with **Docker**
- Introducing **Quarkus** for beginners
- Implementing the sample monolithic application
- **Splitting a monolith** using the **Domain Driven Design** approach
- Implementing the Cloud patterns
- Rethinking the deployment process
- Introducing the **Kubernetes** world
- Introducing the best practices and the recommended patterns while making **Cloud Native** applications

By the end of reading this book, you will have practical hands-on experience of building microservices using **Quarkus** and you will master deploying them as containers to **Kubernetes**.

What this book covers

- **Chapter 0: Getting started with Containerization**, presents the containerization with **Docker & Podman**.
- **Chapter 1: Introduction to the Monolithic architecture**, provides an introduction to the **Monolithic architecture** with a focus on its **advantages** and **drawbacks**.
- **Chapter 2: Coding the Monolithic application**, offers a step-by-step tutorial for modeling and creating the monolith using **Maven, Java 11 & Quarkus**.
- **Chapter 3: Upgrading the Monolithic application**, lists some upgrades for our **Monolithic application** such as adding tests and most-wanted features.
- **Chapter 4: Building & Deploying the Monolithic application**, covers how to build and package our example application before showing how to deploy it in the runtime environment.
- **Chapter 5: Adding the anti-disasters layers**, introduces how to implement the **Security** and **Monitoring** layers to help the application to avoid disasters.
- **Chapter 6: Microservices Architecture Pattern**, presents the drawbacks that can be faced in a typical **Monolithic architecture** and shows the motivations for seeking something new, like the **Microservices Architecture Pattern**.
- **Chapter 7: Splitting the Monolith: Bombarding the domain**, presents the **Domain Driven Design concepts** to successfully understand the **Business Domain** of the **Monolithic Application**, to be able to split it into sub-domains.
- **Chapter 8: Applying DDD to the code**, shows how to apply all the **DDD concepts** to the Monolithic Application source code and to successfully **split the monolith into Bounded Contexts**.
- **Chapter 9: Meeting the microservices concerns and patterns**, covers a deep presentation of the concerns and the cloud patterns related to the **Microservices Architecture Pattern**.
- **Chapter 10: Getting started with Kubernetes**, covers how to build our real standalone business microservices.
- **Chapter 11: Implementing the Cloud Patterns**, presents in deep the greatest container orchestrator: **Kubernetes**
- **Chapter 12: Building the Kuberntezed Microservices**, shows how to migrate the monolith code to build microservices while applying **Kubernetes** concepts and features.
- **Chapter 13: Flying all over the Sky with Quarkus and Kubernetes** will introduce you to how implement additional cloud patterns using **Quarkus** and **Kubernetes**.
- **Chapter 14: Playing with Quarkus in Azure** shows how **Microsoft Azure** can help you to modernize applications faster and how you can use **Serverless Architecture** to add additional great features.
- **Chapter 15: Bringing Dapr into the game** presents **Dapr** & covers how a **Distributed Application Runtime** can improve a microservices architecture.
- **and more 😊**



As this book is published online, I will be providing regular updates for chapters, adding new ones, covering more items, etc.

The source code of this book is available on my **GitHub**:

- <https://github.com/nebrass/playing-with-quarkus-monolith-book-code>
- <https://github.com/nebrass/playing-with-quarkus-microservices-book-code>

Reader feedback

I always welcome feedback from my great readers. 😊 Please let me know what you thought about this book — what you liked or disliked.

If you have any question or inquiry, feel free to get in touch with me directly on  [lnibrass@gmail.com].

Introduction

Quarkus is the latest Java framework made by **Red Hat**, initially started in 2018. **Quarkus** is the latest Java Framework project in the long list of frameworks created by **Red Hat**. **Quarkus** is the successor of the **Wildfly Swarm** and **Thorntail** frameworks. These two frameworks came to the market in the golden age of the **Spring Boot** framework. This is why they were defeated and they didn't win the battle. I personally wasn't motivated for the **Wildfly Swarm** since the first *POCs* I made and I didn't get the catching feeling to adopt it. I will come back later to these two frameworks and I will cover why they were failing with more details.

But why **Red Hat** has made another framework? What will bring this framework?

Red Hat is one of the biggest contributors to the **Java platform** from the design and specifications until providing implementations and tooling. While contributing to the **GraalVM** project, **Red Hat** spotted the opportunity to make a framework dedicated to this great project and to the Cloud.

Wait! What is **GraalVM**? and what is the added value of this project?

Based on the Oracle official documentation

GraalVM is a high-performance runtime that provides significant improvements in application performance and efficiency which is ideal for µservices. It is designed for applications written in **Java, JavaScript, LLVM-based languages such as C and C++**, and other dynamic languages. It removes the isolation between programming languages and enables interoperability in a shared runtime. It can run either standalone or in the context of **OpenJDK, Node.js or Oracle Database**.

GraalVM can run in the context of **OpenJDK** to make Java applications run faster with a new just-in-time compilation technology. **GraalVM** takes over the compilation of Java bytecode to machine code.

The compiler of **GraalVM** provides performance advantages for highly abstracted programs due to its ability to remove costly object allocations in many scenarios.

— Official GraalVM Documentation from Oracle, <https://www.graalvm.org/docs/why-graal/>

I'm sure that this abstract definition doesn't give any clear view about this **runtime**. But what's sure, you can notice how many times the word "**performance**" appeared in this definition.

In my words, **GraalVM** offers the ability to compile your **Java code** into a **native executable binary** which will be executed natively in the operation system without the need to the **Java runtime**. The **native executable binary** offers two main features:

- Faster startup
- Lower memory footprint

Here, **Red Hat** found a big opportunity to make a **Java Framework** dedicated for this *high-performance runtime*. This is what encouraged them to give **Quarkus** a slogan: **SUPersonic Subatomic Java**.

This would be not big idea to make a framework just based on a specific runtime. **Red Hat** had the same point of view, so they added more features (that we will cover later) to their checklist so that their framework can succeed in the competition against his opponent, the famous **Spring Boot** !

This was the point that attracted me to discover **Quarkus** ! I wanted to try how **Red Hat** is prepared to compete against **Spring Boot** ?

In this book, I will try to initiate you to **Quarkus** and **GraalVM** in a soft way. If you are a beginner, you will not face any problem dealing with it, maybe you will not see so much difference between it and **Spring Boot**, but I will try to make the picture clear for you ! 😊

We will start by building a small "Hello World" application using **Quarkus** and next we will be making our **Monolith** that we will migrate to the **Microservices architecture**.

You can jump to the **Chapter 2** if you are already used to the **Quarkus Platform**.

CHAPTER ZERO

Getting started with Containerization

Introduction to containerization

In the Java World, the applications are packaged in many formats before being deployed to the runtime. These environments can be physical machines (Bare Metal) or virtual machines.

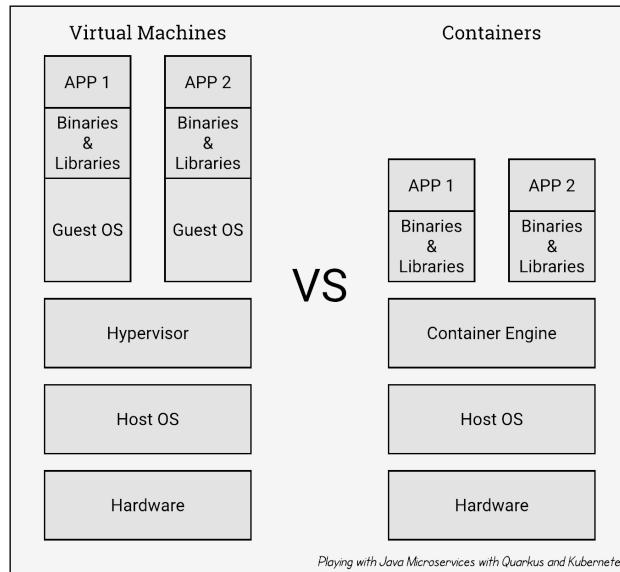
The major risk that encounters software application, is updates of the runtime that can break the application. For example, an operating system update might include many updates, including libraries that can affect the running application with incompatible updates.

Generally, there are other software that coexist with our application on the same host like databases or proxies. They are all sharing the same operating systems and libraries. So, even if an update did not cause any crash directly for the application, maybe there will be one for any of the other services.

Althought these upgrades are risky, we cannot deny them because of their interest in matter of security and stability, thru the provided bug fixes and enhances in updates. But, we can do some tests to our application and its context to see if updates will cause regressions or not. This task can be heavy especially if our application is huge.

Keep calm ! There is an excellent solution, we can use **containers**, which are a kind of isolated partition inside a single operating system. They provide many of the same benefits as virtual machines, such as security, storage, and network isolation, while they require far fewer hardware resources. **Containers** are great because they are quicker to launch and to terminate.

Containerization allows isolating and tracking resources utilization. This isolation protects our application from many risks linked with host OS update.



Containers have many benefits for both developers and system administrators.

- **Consistent Environment:** Containers give developers the ability to create predictable environments that are isolated from other applications. Containers can also include software dependencies needed by the application, such as specific versions of programming language runtimes and other software libraries. From the developer's perspective, all this is guaranteed to be consistent no matter where the application will be deployed. All this translates to productivity: developers and IT Ops teams spend less time debugging and diagnosing differences in environments, and more time shipping new functionality for users. And it means fewer bugs since developers can now make assumptions in dev and test

environments they can be sure will hold true in production.

- **Run Anywhere:** Containers are able to run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or bare metal; on a developer’s machine or in data centers on-premises; and of course, in the public cloud. Wherever you want to run your software, you can use containers.
- **Isolation:** Containers virtualize CPU, memory, storage, and network resources at the OS-level, providing developers with a sandboxed view of the OS logically isolated from other applications.

Finally, containers boost the microservices development approach because they provide a lightweight and reliable environment to create and run services that can be deployed to a production or development environment without the complexity of a multiple machine environment.

There are many container formats available.  **Docker** is a popular, open-source container format.

Introducing Docker

What is Docker ?

 **Docker** is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of  **Linux** containers to deploy applications is called *containerization*.



Containerization is increasingly popular because containers are:

- **Flexible:** Even the most complex applications can be containerized.
- **Lightweight:** Containers leverage and share the host kernel.
- **Interchangeable:** You can deploy updates and upgrades on-the-fly.
- **Portable:** You can build locally, deploy to the cloud, and run anywhere.
- **Scalable:** You can increase and automatically distribute container replicas.
- **Stackable:** You can stack services vertically and on-the-fly.

Images and containers

A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application—the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a runtime instance of an image—what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, `docker ps`, just as you would in  **Linux**.

Installation and first hands-on

Installation

To start playing with Docker, we need to install Docker. You can grab the version that is compatible with your platform from this URL: <https://docs.docker.com/install/>

But when you will come on that page, you will find two versions: **Docker CE** and **Docker EE**; So which one you need ?

- **Docker CE**: Docker Community Edition is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker CE has three types of update channels, **stable**, **test**, and **nightly**:
 - **Stable** gives you latest releases for general availability.
 - **Test** gives pre-releases that are ready for testing before general availability.
 - **Nightly** gives you latest builds of work in progress for the next major release.
- **Docker EE**: Docker Enterprise Edition is designed for enterprise development and IT teams who build, ship, and run business-critical applications in production and at scale. Docker EE is integrated, certified, and supported to provide enterprises with the most secure container platform in the industry.

The installation of Docker is very easy, and does not need a tutorial to be done ☺



For **Windows** user, ensure that **Docker** Desktop For Windows uses **Linux** containers.

After installation, to check if **Docker** is installed, we can for example start by checking the installed version by running the command `docker version`:

```
Client: Docker Engine - Community ②
  Version:           19.03.12
  API version:      1.40
  Go version:       go1.13.10
  Git commit:       48a66213fe
  Built:            Mon Jun 22 15:45:44 2020
  OS/Arch:          linux/amd64
  Experimental:    false

Server: Docker Engine - Community ①
  Engine:
    Version:           19.03.12
    API version:      1.40 (minimum version 1.12)
    Go version:       go1.13.10
    Git commit:       48a66213fe
    Built:            Mon Jun 22 15:44:15 2020
    OS/Arch:          linux/amd64
    Experimental:    false
  containerd:
    Version:           1.2.13
    GitCommit:         7ad184331fa3e55e52b890ea95e65ba581ae3429
  runc:
    Version:           1.0.0-rc10
    GitCommit:         dc9208a3303feef5b3839f4323d9beb36df0a9dd
  docker-init:
    Version:           0.18.0
    GitCommit:         fec3683
```

1. The **Docker daemon** listens for **Docker API** requests and manages Docker objects. A daemon can also communicate with other daemons to manage Docker services.
2. The **Docker client** is the primary way that many Docker users interact with Docker. When you run commands, the client sends these commands to the **Docker daemon**, which carries them out. The `docker` command uses the **Docker API**. The Docker client can communicate with more than one Docker daemon.

Docker uses a **client-server** architecture. The Docker **client** talks to the Docker **daemon** (`dockerd`), which does the heavy lifting of building, running, and distributing your Docker containers. The Docker **client** and **daemon** can run on the same system, or you can connect a Docker **client** to a remote Docker **daemon**. The Docker **client** and **daemon** communicate using a **REST API**, over **UNIX sockets** or a **network interface**.

Run your first container

We will start by running the **hello-world** image:

```
$ docker run hello-world

Unable to find image "hello-world:latest" locally
latest: Pulling from library/hello-world
9db2cabcacae0: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

What is Docker Hub?

Docker Hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, so you can deploy images to your hosts. provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

Docker Hub provides the following major features:

- **Image Repositories:** Find and pull images from community and official libraries, and manage, push to, and pull from private image libraries to which you have access.
- **Automated Builds:** Automatically create new images when you make changes to a source code repository.
- **Webhooks:** A feature of Automated Builds, Webhooks let you trigger actions after a successful push to a repository.
- **Organizations:** Create work groups to manage access to image repositories.
- **GitHub and Bitbucket Integration:** Add the Hub and your Docker Images to your current workflows.

To list the existing local Docker images:

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello-world	latest	2cb0d9787c4d	4 weeks ago	1.85kB



The **Image ID** is a random generated *HEX* value to identify an **Image**.

To list all the Docker containers:

```
$ docker container ls --all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	NAMES
54f4984ed6a8	hello-world	"/hello"	20 sec ago	Exited (0)	9 sec ago
suzyland					

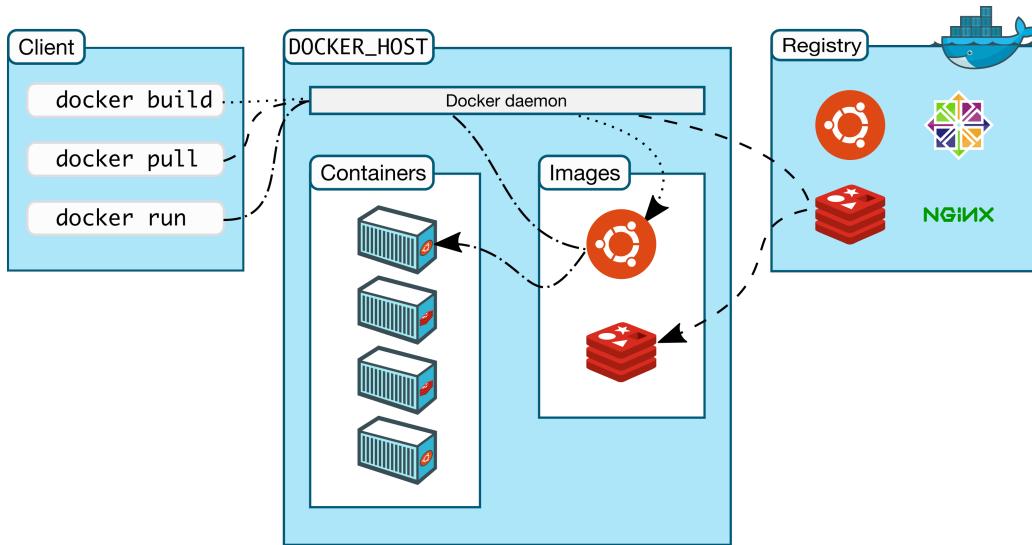
Some details listed here:

1. The **Container ID** is a random generated HEX value to identify an **Container**.
2. The **Container Name** that we got here is a randomly generated string name that the Docker Daemon created for us, as we didn't specify one.

Docker Architecture

Docker uses a client-server architecture:

- The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.



The Docker daemon

The Docker daemon (`dockerd`) listens for **Docker API** requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (`docker`) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker registries

A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. If you use **Docker Datacenter (DDC)**, it includes **Docker Trusted Registry (DTR)**.

When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the `docker push` command, your image is pushed to your configured registry.

Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other

objects. This section is a brief overview of some of those objects.

Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the `ubuntu` image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Containers

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Docker Machine

Docker Machine is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands. You can use Machine to create Docker hosts on your local Mac or Windows box, on your company network, in your data center, or on cloud providers like Azure, AWS, or Digital Ocean.

Using `docker-machine` commands, you can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to your host.

Point the Machine CLI at a running, managed host, and you can run `docker` commands directly on that host. For example, run `docker-machine env default` to point to a host called `default`, follow on-screen instructions to complete `env` setup, and run `docker ps`, `docker run hello-world`, and so forth.

Docker Machine was the only way to run Docker on Mac or Windows previous to Docker v1.12. Starting with the beta program and Docker v1.12, Docker for Mac and Docker for Windows are available as native apps and the better choice for this use case on newer desktops and laptops.



The Docker Team recommends to try out these new apps.

The installers for Docker for Mac and Docker for Windows include Docker Machine, along with Docker Compose (don't be afraid, we will use it in the next steps).

Why should I use it?

Docker Machine enables you to provision multiple remote Docker hosts on various flavors of Linux.

Additionally, Machine allows you to run Docker on older Mac or Windows systems, as described in the previous topic.

Docker Machine has these two broad use cases:

Using Docker on older machines

I have an older desktop system and want to run Docker on Mac or Windows:

If you work primarily on an older Mac or Windows laptop or desktop that doesn't meet the requirements for the new Docker for Mac and Docker for Windows apps, then you need Docker Machine to run Docker Engine locally. Installing Docker Machine on a Mac or Windows box with the Docker Toolbox installer provisions a local virtual machine with Docker Engine, gives you the ability to connect it, and run docker commands.

Provision remote Docker instances

Docker Engine runs natively on **A Linux** systems. If you have a **A Linux** box as your primary system, and want to run docker commands, all you need to do is download and install Docker Engine. However, if you want an efficient way to provision multiple Docker hosts on a network, in the cloud or even locally, you need Docker Machine.

Whether your primary system is Mac, Windows, or Linux, you can install Docker Machine on it and use docker-machine commands to provision and manage large numbers of Docker hosts. It automatically creates hosts, installs Docker Engine on them, then configures the docker clients. Each managed host ("machine") is the combination of a Docker host and a configured client.

What's the difference between Docker Engine and Docker Machine?

When people say "**Docker**" they typically mean **Docker Engine**, the client-server application made up of the Docker daemon, a REST API that specifies interfaces for interacting with the daemon, and a command line interface (CLI) client that talks to the daemon (through the REST API wrapper). Docker Engine accepts **docker** commands from the CLI, such as **docker run <image>**, **docker ps** to list running containers, **docker image ls** to list images, and so on.

Docker Machine is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them). Typically, you install Docker Machine on your local system.

Docker Machine has its own command line client **docker-machine** and the Docker Engine client, **docker**. You can use Machine to install Docker Engine on one or more virtual systems. These virtual systems can be local (as when you use Machine to install and run Docker Engine in VirtualBox on Mac or Windows) or remote (as when you use Machine to provision Dockerized hosts on cloud providers).

The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed "***machines***".

Diving into Docker Containers

Introduction

The best way to learn more about Docker, is to write an app in the Docker way :)

Your new development environment

In the past, if you were to start writing a Java app, your first order of business was to install a Java runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Java runtime as an image, no installation necessary. Then, your build can include the base Java image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a **Dockerfile**.

Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to "copy in" to that environment.

However, after doing that, you can expect that the build of your app defined in this **Dockerfile** behaves exactly the same wherever it runs.

Example of a Dockerfile

```
# Use an OpenJDK Runtime as a parent image
FROM openjdk:8-jre-alpine

# Add Maintainer Info
LABEL maintainer="lnibrass@gmail.com"

# Define environment variables
ENV JAVA_OPTS="-Xmx2048m"

# Set the working directory to /app
WORKDIR /app

# Copy the executable into the container at /app
ADD some-application-built-somewhere.jar app.jar

# Make port 8080 available to the world outside this container
EXPOSE 8080

# Run app.jar when the container launches
CMD ["java", "-Djava.security.egd=file:/dev/.urandom", "-jar", "/app/app.jar"]
```

This **Dockerfile** will:

- create a container based on **OpenJDK 8**
- define an environment variable of the maximum memory allocation pool for the JVM to 2GB
- define the working directory to **/app**
- copy **some-application-built-somewhere.jar** file from local path as **app.jar**
- open the port **8080**
- define the startup command which will be executed when the container will start

Create sample application

We will create [sample-app](#): a HelloWorld Quarkus application using [code.quarkus.io](#) we need just to define the **GroupId** and the **ArtifactId**:

The screenshot shows the Quarkus application generator interface. At the top, it displays "QUARKUS" and "Available with Enterprise Support". Below that, it says "Configure your application details" with fields for Group (com.targa.labs.dev), Artifact (sample-app), and Build Tool (Maven). A "CONFIGURE MORE OPTIONS" button is also present. On the right, a large blue button says "Generate your application (alt + ⌘)".

Pick your extensions

A search bar contains "RESTEasy, Hibernate ORM, Web...". Under "Selected Extensions", there is a list of checked extensions: RESTEasy JAX-RS (INCLUDED), RESTEasy JSON-B, RESTEasy Jackson, Hibernate Validator, and REST Client.

Web

Extension	Description	More Options
RESTEasy JAX-RS (INCLUDED)	REST endpoint framework implementing JAX-RS and more	⋮
RESTEasy JSON-B	JSON-B serialization support for RESTEasy	⋮
RESTEasy Jackson	Jackson serialization support for RESTEasy	⋮
Hibernate Validator	Validate object properties (field, getter) and method parameters fo...	⋮
REST Client	Call REST services	⋮

The generated application already comes with a sample Rest API:

```
@Path("/hello")
public class ExampleResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }
}
```

When we compile this application using `mvn clean install`, our JAR, will be available in the **target** folder.

In the **target** folder, run this command will show only the files in the current folder:

```
$ ls -p target/ | grep -v /
sample-app-1.0.0-SNAPSHOT.jar
sample-app-1.0.0-SNAPSHOT-runner.jar
```

Instead of writing our own **Dockerfile**, while generating a new project, Quarkus comes with a set of Dockerfiles available under the folder **src/main/docker**:

```
$ ls -p src/main/docker | grep -v /
```

Dockerfile.fast-jar	①
Dockerfile.jvm	②
Dockerfile.native	③

These **Dockerfile** files used to build a container that runs the **Quarkus** application:

- ① with the **fast-jar** packaging format, which was designed to provide faster startup times
- ② in **JVM mode**
- ③ in **Native mode** (no JVM) ↪ we will talk a lot about the **Native mode** in the next chapters ☺

Let's build the **JVM mode** based **Docker Container** and tag it as *nebrass/sample-jvm:1.0.0-SNAPSHOT*:

```
> mvn package && docker build -f src/main/docker/Dockerfile.jvm -t nebrass/sample-jvm:1.0.0-SNAPSHOT .
```

```
Sending build context to Docker daemon 51.09MB
Step 1/11 : FROM registry.access.redhat.com/ubi8/ubi-minimal:8.1
8.1: Pulling from ubi8/ubi-minimal
b26afdf22be4: Pull complete
218f593046ab: Pull complete
Digest: sha256:df6f9e5d689e4a0b295ff12abc6e2ae2932a1f3e479ae1124ab76cf40c3a8cdd
Status: Downloaded newer image for registry.access.redhat.com/ubi8/ubi-minimal:8.1
--> 91d23a64fdf2
①
①
①
①
①
②
Step 2/11 : ARG JAVA_PACKAGE=java-11-openjdk-headless
--> Running in 6f73b83ed808
Removing intermediate container 6f73b83ed808
--> 35ba9340154b
②
Step 3/11 : ARG RUN_JAVA_VERSION=1.3.8
--> Running in 695d7dcf4639
Removing intermediate container 695d7dcf4639
--> 04e28e22951e
②
Step 4/11 : ENV LANG="en_US.UTF-8" LANGUAGE="en_US:en"
--> Running in 71dc02dbe31
Removing intermediate container 71dc02dbe31
--> 7c7c69eead06
②
Step 5/11 : RUN microdnf install curl ca-certificates ${JAVA_PACKAGE} \
&& microdnf update \
&& microdnf clean all \
&& mkdir /deployments \
&& chown 1001 /deployments \
&& chmod "g+rwx" /deployments \
```

```

&& chown 1001:root /deployments \
&& curl https://repo1.maven.org/maven2/io/fabric8/run-java-
sh/${RUN_JAVA_VERSION}/run-java-sh-${RUN_JAVA_VERSION}-sh.sh -o /deployments/run-java.sh
\
&& chown 1001 /deployments/run-java.sh \
&& chmod 540 /deployments/run-java.sh \
&& echo "securerandom.source=file:/dev/urandom" >>
/etc/alternatives/jre/lib/security/java.security
---> Running in 2274fdc94d6f
Removing intermediate container 2274fdc94d6f
---> 9fd48c2d9482
Step 6/11 : ENV JAVA_OPTIONS="-Dquarkus.http.host=0.0.0.0
-Djava.util.logging.manager=org.jboss.logmanager.LogManager"
---> Running in c0e3ddc80993
Removing intermediate container c0e3ddc80993
---> 26f287fde6f6
Step 7/11 : COPY target/lib/* /deployments/lib/
---> 1c3aa9a683a6
Step 8/11 : COPY target/*-runner.jar /deployments/app.jar
---> d1bdd5e96e5e
Step 9/11 : EXPOSE 8080
---> Running in 728f82b270d2
Removing intermediate container 728f82b270d2
---> 704cd49fd439
Step 10/11 : USER 1001
---> Running in 5f7aef93c3d7
Removing intermediate container 5f7aef93c3d7
---> 5a773add2a6d
Step 11/11 : ENTRYPOINT [ "/deployments/run-java.sh" ]
---> Running in cb6d917592bc
Removing intermediate container cb6d917592bc
---> 9bc81f158728
Successfully built 9bc81f158728
Successfully tagged nebrass/sample-jvm:latest

```

- As Docker didn't find the **ubi8/ubi-minimal** image locally, it proceeds to downloading it from the registry.access.redhat.com/ubi8/ubi-minimal:8.1.
- Every instruction in the **Dockerfile** is built in a dedicated step; and it generates a separated **LAYER** in the **IMAGE**; the shown **HEX code** at the end of each **STEP** is the **ID** of the **LAYER**.
- The **IMAGE ID**.
- Our built image is tagged with **nebrass/sample-jvm:latest**; we specified only the name (**nebrass/sample-jvm:latest**) and Docker added for us automatically the latest version tag.

What is the UBI base image?

The provided **Dockerfiles** use UBI (*Universal Base Image*) as parent image. This base image has been tailored to work perfectly in containers. The **Dockerfiles** use the minimal version of the base image to reduce the size of the produced image.

Where is your built image? It's in your machine's local **Docker Image Registry**:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nebrass/sample-jvm	1.0.0-SNAPSHOT	9bc81f158728	5 minutes ago	501MB
ubi8/ubi-minimal	8-jre-alpine	ccfb0c83b2fe	4 weeks ago	107MB

We see that we have locally two images; the `ubi8/ubi-minimal` is the base image; and `nebrass/sample-jvm` which is our built image.

Run the app

Let's run the built container, mapping your machine's port **28080** to the container's published port **8080** using **-P**:

```
$ docker run --rm -p 28080:8080 nebrass/sample-jvm:1.0.0-SNAPSHOT
```

```
1 exec java -Dquarkus.http.host=0.0.0.0
-Djava.util.logging.manager=org.jboss.logmanager.LogManager -XX:+ExitOnOutOfMemoryError
-cp . -jar /deployments/app.jar
2 --
3 --/ _\ \_ \| / / / _ | / _\ \_ \| / / / _\ /
4 -/_ / / / / / _ \| , _\ , < / / / \ \ \
5 --\_\_\_\_\_/_\|/_/_|/_/_|\_\_\_/_\_/
6 2020-08-05 13:53:44,198 INFO [io.quarkus] (main) sample-app 1.0.0-SNAPSHOT on JVM
(powered by Quarkus 1.6.1.Final) started in 0.588s. Listening on: http://0.0.0.0:8080
7 2020-08-05 13:53:44,218 INFO [io.quarkus] (main) Profile prod activated.
8 2020-08-05 13:53:44,219 INFO [io.quarkus] (main) Installed features: [cdi, resteasy]
```

- The line 1 is showing the command that will start the Java application in the container
 - The lines 2-8 are listing the Logs of the Quarkus application. There you will see a message mentioning **Listening on: http://0.0.0.0:8080**. This message is coming from inside the container, which doesn't know you mapped port **8080** of that container to **28080**, making the correct URL **http://localhost:28080**.

If you open the URL <http://localhost:28080> in a web browser, you will land to the default Quarkus `index.html` file:

The screenshot shows a Mozilla Firefox window titled "sample-app - 1.0.0-SNAPSHOT - Mozilla Firefox". The address bar shows "localhost:28080". The main content area displays the following text:

Your new Cloud-Native application is ready!

Congratulations, you have created a new Quarkus application.

Why do you see this?

This page is served by Quarkus. The source is in `src/main/resources/META-INF/resources/index.html`.

What can I do from here?

If not already done, run the application in *dev mode* using: `mvn compile quarkus:dev`.

- Add REST resources, Servlets, functions and other services in `src/main/java`.
- Your static assets are located in `src/main/resources/META-INF/resources`.
- Configure your application in `src/main/resources/application.properties`.

Do you like Quarkus?

Go give it a star on [GitHub](#).

How do I get rid of this page?

Just delete the `src/main/resources/META-INF/resources/index.html` file.

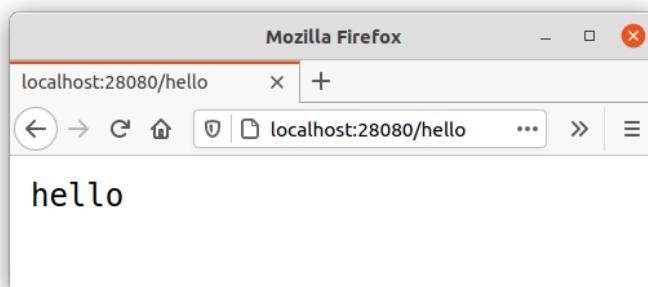
Application

- GroupId: com.targa.labs.dev
- ArtifactId: sample-app
- Version: 1.0.0-SNAPSHOT
- Quarkus Version: 1.6.1.Final

Next steps

- [Setup your IDE](#)
- [Getting started](#)
- [Quarkus Web Site](#)

Go to <http://localhost:28080/hello> in a web browser, you will see the response of our **ExampleResource** REST API.



If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of `localhost`.



For example, <http://192.168.99.100:28080/hello>. To find the IP address, use the command `docker-machine ip`.

You can also use the `curl` command in a shell to view the same content.

```
$ curl http://localhost:28080/hello
hello%
```

What is cURL ?

`cURL` is a computer software project providing a library and command-line tool for transferring data using various protocols. The `cURL` project produces two products, `libcurl` and `cURL`. It was first released in 1997. The name stands for "Client URL".

- **libcurl:** is a free client-side URL transfer library, supporting cookies, DICT, FTP, FTPS, Gopher, HTTP (with HTTP/2 support), HTTP POST, HTTP PUT, HTTP proxy tunneling, HTTPS, IMAP, Kerberos, LDAP, POP3, RTSP, SCP, and SMTP. The library supports the file URI scheme, SFTP, Telnet, TFTP, file transfer resume, FTP uploading, HTTP form-based upload, HTTPS certificates, LDAPS, proxies, and user-plus-password authentication.
- **cURL:** `cURL` is a command-line tool for getting or sending files using URL syntax.

Since `cURL` uses `libcurl`, it supports a range of common Internet protocols, currently including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, LDAP, DAP, DICT, TELNET, FILE, IMAP, POP3, SMTP and RTSP

This port mapping of `28080:8080` demonstrates the difference between `EXPOSE` within the `Dockerfile` and what the `publish` value is set to when running `docker run -p`. In later steps, mapping the port `28080` on the host to port `8080` in the container and use `http://localhost`.

In the console where you run the Docker container, just hit `CTRL+C` in your terminal to quit.



On Windows, explicitly stop the container

On Windows systems, `CTRL+C` does not stop the container. So, first type `CTRL+C` to get the prompt back (or open another shell), then type `docker container ls` to list the running containers, followed by `docker container stop <Container NAME or ID>` to stop the container. Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

But, if you want to run the Docker container in **detached mode**, just add `-d` to run command:

```
$ docker run -d -p 28080:8080 nebrass/sample-jvm:1.0.0-SNAPSHOT
fbf2fba8e9b14a43e3b25aea1cb94b751bbbb6b73af05b84aab3f686ba5019c8
```

Here we got the long container ID for our app and then are kicked back to our terminal `✉` then the container is running in the background.

We can also see the abbreviated container ID with `docker ps` (and both long or short form are working interchangeably while running any command on a container):

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	STATUS	PORTS
fbf2fba8e9b1	nebrass/sample-jvm..	"/deployments/run-ja..."	Up	28080->8080/tcp

Notice that **CONTAINER ID** matches what's on <http://localhost:28080>.

Now use `docker stop` to end the process, using the **CONTAINER ID**, like so:

```
docker stop fbf2fba8e9b1
```

The result of the command will show the **CONTAINER ID** that got stopped.

Share your image

To demonstrate the portability of what we just created, let's upload our built image and run it somewhere else. After all, you need to know how to push to registries when you want to deploy containers to production.

A **registry** is a collection of repositories, and a repository is a collection of images—sort of like a *GitHub repository*, except the code is already built. An account on a registry can create many repositories. The `docker` CLI is pointing on the **Docker's public registry** by default.



We use **Docker's public registry** here just because it's free and pre-configured, but there are many public ones to choose from, and you can even set up your own private registry using **Docker Trusted Registry**.

Log in with your Docker ID

If you don't have a Docker account, sign up for one at <https://hub.docker.com/>.

Log in to the Docker public registry on your local machine.

```
docker login -u <username> -p <password>
```

Tag the image

The notation for associating a local image with a repository on a registry is `username/repository:tag`. The tag is optional, but recommended, since it is the mechanism that registries use to give Docker images a version. Give the repository and tag meaningful names for the context, such as `get-started:part2`. This puts the image in the `get-started` repository and tag it as `part2`.

Now, put it all together to tag the image. Run docker tag image with your username, repository, and tag names so that the image uploads to your desired destination. The syntax of the command is:

```
docker tag image username/repository:tag
```

For example, if we want to promote our **SNAPSHOT** version to **Final**, we do:

```
docker tag nebrass/sample-jvm:1.0.0-SNAPSHOT nebrass/sample-jvm:1.0.0-Final
```

Run the command: **docker images** to see your newly tagged image:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nebrass/sample-jvm	1.0.0-Final	598712377440	43 minutes ago	501MB
nebrass/sample-jvm	1.0.0-SNAPSHOT	598712377440	43 minutes ago	501MB
ubi8/ubi-minimal	8.1	91d23a64fdf2	4 weeks ago	107MB

Publish the image

Upload your tagged image to the repository:

```
docker push username/repository:tag
```

Once complete, the results of this upload are publicly available. If you log in to Docker Hub, you see the new image there, with its pull command.



You need to be authenticated using **docker login** command before pushing images.

We will push our **nebrass/sample-jvm:1.0.0-Final** tagged image:

```
docker push nebrass/sample-jvm:1.0.0-Final
```

The push refers to repository [docker.io/nebrass/sample-jvm]

0f8895a56cf0: Pushed

9c443a7a1622: Pushed

fb6a9f86c4e7: Pushed

eddba477a8ae: Pushed

f80c95f61fff: Pushed

1.0.0-Final: digest:

sha256:30342f5f4a432a2818040438a24525c8ef9d046f29e3283ed2e84fdbdbe3af55 size: 1371

Pull and run the image from the remote repository

From now on, you can use `docker run` to run your app on any machine, if the image is not available locally, the Docker daemon will look to it in the Docker Hub:

Let's remove all the local containers related to `nebrass/sample-jvm`:

```
docker ps -a | awk '{ print $1,$2 }' | grep nebrass/sample-jvm | awk '{print $1 }' |  
xargs -I {} docker rm {}
```

Let's remove all the local images related to `nebrass/sample-jvm`:

```
docker images | awk '{ print $1":"$2 }' | grep nebrass/sample-jvm | xargs -I {} docker  
rmi {}
```

If the image isn't available locally on the machine, Docker will pull it from the repository.

```
$ docker run -p 28080:8080 nebrass/sample-jvm:1.0.0-Final  
  
Unable to find image 'nebrass/sample-jvm:1.0.0-Final' locally  
1.0.0-Final: Pulling from nebrass/sample-jvm  
b26afdf22be4: Already exists  
218f593046ab: Already exists  
284bc7c3a139: Pull complete  
775c3b820c36: Pull complete  
4d033ca6332d: Pull complete  
Digest: sha256:30342f5f4a432a2818040438a24525c8ef9d046f29e3283ed2e84fdbdbe3af55  
Status: Downloaded newer image for nebrass/sample-jvm:1.0.0-Final  
...
```

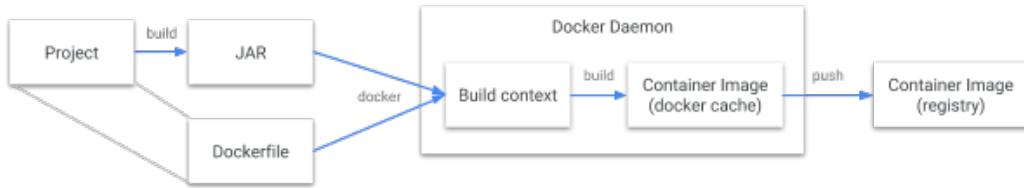
No matter where `docker run` executes, it pulls your image, along with the `ubi8/ubi-minimal` and all the dependencies, and runs your code. It all travels together in a neat little package, and you don't need to install anything on the host machine for Docker to run it.

Playing with Google Jib

Jib is a Maven plugin for building Docker and OCI images for Java applications from Google.

Jib is a fast and simple container image builder that handles all the steps of packaging your application into a container image. **It does not require you to write a Dockerfile or have docker installed**, and it is directly integrated into **Maven** by just adding the plugin to your build and you'll have your Java application containerized in no time.

Docker build flow:



Jib build flow:



Jib is available as plugins for **Maven** and **Gradle** and requires minimal configuration. Simply add the plugin to your build definition and configure the target image. Jib also provides additional rules for building an image to a Docker daemon if you need it.



If you are building to a private registry, make sure to configure Jib with credentials for your registry. Learn more about this in [the official Jib Documentation](#).

In the Quarkus ecosystem, we have already a **Jib** extension ready-to-use called [quarkus-container-image-jib](#). This extension is powered by **Jib** for performing container image builds. The major benefit of using **Jib** with Quarkus is that all the dependencies (everything found under [target/lib](#)) are cached in a different layer than the actual application making rebuilds really fast and small (when it comes to pushing). Another important benefit of using this extension is that it provides the ability to create a container image without having to have any dedicated client side tooling (like Docker) or running daemon processes (like the Docker daemon) when all that is needed is the ability to push to a container image registry.

To use this feature, add the following extension to your project:

```
./mvnw quarkus:add-extension -Dextensions="container-image-jib"
```



In situations where all that is needed to build a container image and no push to a registry is necessary (essentially by having set `quarkus.container-image.build=true` and left `quarkus.container-image.push` unset - it defaults to `false`), then this extension creates a container image and registers it with the Docker daemon. This means that although Docker isn't used to build the image, it is nevertheless necessary.

Also note that using this mode, the built container image will show up when executing `docker images`.

Building with Jib

Build your container image without pushing it to a container image registry with:

```
mvn clean package -Dquarkus.container-image.build=true
```

If you want to build an image and push it to the authenticated container image registry:

```
mvn clean package -Dquarkus.container-image.build=true -Dquarkus.container-image.push=true
```

There are many configuration options available to customize the image or to do specific actions:

Configuration property	Purpose
<code>quarkus.container-image.group</code>	The group the container image will be part of. If not set defaults to the logged in user
<code>quarkus.container-image.name</code>	The name of the container image. If not set defaults to the application name
<code>quarkus.container-image.tag</code>	The tag of the container image. If not set defaults to the application version
<code>quarkus.container-image.registry</code>	The container registry to use. If not set defaults to the authenticated registry
<code>quarkus.container-image.username</code>	The username to use to authenticate with the registry where the built image will be pushed
<code>quarkus.container-image.password</code>	The password to use to authenticate with the registry where the built image will be pushed

You can dig more into the great Jib extension on the [Quarkus Container Images Guide](#).

That's all tale !! 😊

Meeting the Docker Services

In a distributed application, different pieces of the app are called "**services**". For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just "**containers in production**". A **service** only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Fortunately it's very easy to define, run, and scale services with the Docker platform — just write a `docker-compose.yml` file.

Your first docker-compose.yml file

A `docker-compose.yml` file is a YAML file that defines how Docker containers should behave in production.

Save this file as `docker-compose.yml` wherever you want. Be sure you have pushed the image you created in Part 2 to a registry, and update this `.yml` by replacing `username/repo:tag` with your image details.

Example of a docker-compose.yml file

```

version: "3"
services:
  web:
    image: nebrass/sample-jvm:1.0.0-Final
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.5"
          memory: 250M
      restart_policy:
        condition: on-failure
    ports:
      - "8080:8080"
    networks:
      - webnetwork
networks:
  webnetwork:

```

This `docker-compose.yml` file tells Docker to do the following:

- Pull the image we uploaded in step 2 from the registry.
- Run 5 instances of that image as a service called `web`, limiting each one to use, at most, 10% of the CPU (across all cores), and 250MB of RAM.
- Immediately restart containers if one fails.
- Map port `28080` on the host to web's port `8080`.
- Instruct `web` containers to share port `8080` via a load-balanced network called `webnetwork`.
- Define the `webnetwork` network with the default settings (which is a load-balanced overlay network).

Run your new load-balanced app

Before we can use the docker stack deploy command we first run:

```
docker swarm init
```



If you don't run docker swarm init you get an error that "*this node is not a swarm manager*."

What is Docker Swarm?

A **Swarm** is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you're used to, but now they are executed on a cluster by a Swarm manager.

The machines in a Swarm can be physical or virtual. After joining a Swarm, they are referred to as nodes.

A Swarm is made up of multiple nodes, which can be either physical or virtual machines. The basic concept is simple enough: run `docker swarm init` to enable swarm mode and make your current machine a swarm manager, then run `docker swarm join` on other machines to have them join the swarm as workers.

Choose a tab below to see how this plays out in various contexts. We use VMs to quickly create a two-machine cluster and turn it into a swarm.

Now let's run it. You need to give your app a name, for example **first-cluster**:

```
docker stack deploy -c docker-compose.yml first-cluster
```

Our single service stack is running 5 containers instances of our deployed image on one host. Let's investigate by getting the service ID for the exclusive service in our **Swarm Cluster**:

```
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
vxytki7m5aby	first-cluster_web	replicated	5/5	nebrass/sample-jvm:1.0.0-Final
				:28080->8080/tcp

Look for output for the **web** service, prepended with your app name. If you named it the same as shown in this example, the name is **first-cluster_web**. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of **replicas** you defined in **docker-compose.yml**. List the tasks for your service:

```
docker service ps first-cluster_web
```

Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

```
docker container ls -q
```

Scale the app

You can scale the app by changing the `replicas` value in `docker-compose.yml`, saving the change, and re-running the `docker stack deploy` command:

```
docker stack deploy -c docker-compose.yml first-cluster
```

Docker performs an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run `docker container ls -q` to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

Remove the app and the swarm

Remove the app from the docker stack:

```
docker stack rm first-cluster
```

Remove the swarm:

```
docker swarm leave --force
```

It's as easy as that to stand up and scale your app with Docker. You've taken a huge step towards learning how to run bona fide containers in production.

Achieving more with Docker

Docker optimizes the developer experience and boosts the productivity in many ways. For example, we can use Docker containers to:

1. have the needed software and tools very quickly. For example using Docker to have a local Database or SonarQube instance, etc.
2. build, debug and run the source code, even if you don't have the needed environment. For example, if you don't have JDK installed locally, or even if you don't have the same required version as a project, you can although run it and test it inside containers.
3. resolve some technical requirements, for example, if you have an application that can be executed only on a specific port, that's already used on your machine, you can use the container ports mapping feature to use the same application containerized with a different exposed port.
4. and more ! 😊

Getting the needed tools fast & quickly

We can for example get some needed tools quickly. For example, we can have a **Database** instance in some seconds without any installations, without touching the local environment or even modifying any local file. You can avoid being afraid of some **Mysql** daemon that will remain using the port **3306** even after uninstall 😅

Getting a Dockerized PostgreSQL instance

For example, to have a local **PostgreSQL** instance:

```
docker run -d --name demo-postgres \
-e POSTGRES_USER=developer \
-e POSTGRES_PASSWORD=someCrazyPassword \
-e POSTGRES_DB=demo \
-p 5432:5432 postgres:13
```

① We will run the container that we will call **demo-postgres** in detached mode (as daemon in background)

② We will define many environment variables:

- postgres user: **developer**
- postgres password: **someCrazyPassword**
- postgres database: **demo**

③ We will forward the port **5432** → **5432** and we will use the **official PostgreSQL Image v13**

We can use these credentials in our Java application like any other standalone **PostgreSQL**.

Unfortunately, the Data will be stored exclusively inside the Container. If it's crashing or got deleted, all the data will be lost.

If we want to persist data available inside the container. We need to map a local mount point as a Docker Data Volume to a path inside the container.

I create a volumes folder (we can give the folder any name we like) in my home directory and then create subfolders for each of the applications I need to create data volume mount points for.

Let's start by creating a folder that will be used to store the data in my host machine:

```
mkdir -p $HOME/docker-volumes/postgres
```

This **\$HOME/docker-volumes/postgres** folder will be mapped to the **/var/lib/postgresql/data** folder of the **PostgreSQL Container**, where **PostgreSQL** is storing physical data files.

Now, the Docker command that runs a persistent PosgreSQL container will be:

```
docker run -d --name demo-postgres \
-e POSTGRES_USER=developer \
-e POSTGRES_PASSWORD=someCrazyPassword \
-e POSTGRES_DB=demo \
-p 5432:5432 \
-v $HOME/docker/volumes/postgres:/var/lib/postgresql/data \
postgres:13
```

Let's move to another usecase that I love to talk about always while talking about Docker benefits: having a local **SonarQube** server 😊

Getting a Dockerized SonarQube instance

We can easily have a **SonarQube** instance easily in just one Docker command:

```
docker run -d --name sonarqube \
-p 9000:9000 \
-p 9092:9092 \
sonarqube:8.4.1-community
```

This command will run a container base on the **Docker Image** `sonarqube:8.4.1-community` and exposes the two **SonarQube** ports `9000` and `9092`.

If you are used to **SonarQube** in your company, you can have your local instance, and you can import all the **Quality Profiles** (aka **Gates**) that your team is actually using.

Unleashing the requirements chains

We will not go far here while searching for a usecase. We can take as example the **Quarkus** requirements as usecase: one of the prerequisites is having **GraalVM**. In case of you dont have it installed locally, you can use a **GraalVM Docker Image** to have a container that lets you do **GraalVM** based builds.

Let's go back to the `sample-app` that we generated before. If we want to build a **GraalVM** based JAR file without having **GraalVM** installed locally, we can use this example of Dockerfile, that we will save as `src/main/docker/Dockerfile.multistage`:

`src/main/docker/Dockerfile.multistage`

```

1 ## Stage 1 : build with maven builder image with native capabilities
2 FROM quay.io/quarkus/centos-quarkus-maven:20.1.0-jav11 AS build
3 COPY pom.xml /usr/src/app/
4 RUN mvn -f /usr/src/app/pom.xml -B de.qaware.maven:go-offline-maven-
   plugin:1.2.5:resolve-dependencies
5 COPY src /usr/src/app/src
6 USER root
7 RUN chown -R quarkus /usr/src/app
8 USER quarkus
9 RUN mvn -f /usr/src/app/pom.xml -Pnative clean package
10
11 ## Stage 2 : create the docker final image
12 FROM registry.access.redhat.com/ubi8/ubi-minimal
13 WORKDIR /work/
14 COPY --from=build /usr/src/app/target/*-runner /work/application
15
16 # set up permissions for user '1001'
17 RUN chmod 775 /work /work/application \
18     && chown -R 1001 /work \
19     && chmod -R "g+rwx" /work \
20     && chown -R 1001:root /work
21
22 EXPOSE 8080
23 USER 1001
24
25 CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]

```

This Dockerfile contains **two embedded Dockerfiles**: you can notice there are already **two FROM instructions**. Each part of this Dockerfile is called a **Stage**, this is why we gave the Dockerfile the extension **multistage**.

In the first stage we will generate the native Quarkus executable using Maven, while in the second stage we are using the built JAR file to create the Docker runtime image. We did it just using an installed Maven and Docker runtimes.

Another benefit of the Multi-stage builds is that they enable you to create smaller container images with better caching and smaller security footprint. Multi-stage builds are a new feature requiring Docker 17.05 or higher on the daemon and client.

Multistage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain.

One of the most challenging things about building images is keeping the image size down. Each instruction in the Dockerfile adds a layer to the image, and you need to remember to clean up any artifacts you don't need before moving on to the next layer. To write a really efficient Dockerfile, you have traditionally needed to

employ shell tricks and other logic to keep the layers as small as possible and to ensure that each layer has the artifacts it needs from the previous layer and nothing else.

It was actually very common to have one Dockerfile to use for development (which contained everything needed to build your application), and a slimmed-down one to use for production, which only contained your application and exactly what was needed to run it. This has been referred to as the "builder pattern". Maintaining two Dockerfiles is not ideal.

With multi-stage builds, you use multiple **FROM** statements in your Dockerfile, the same that we have in our [Dockerfile.multistage](#). Each **FROM** instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final Docker image.

We build the multistaged Dockerfile, we just build it in the sameway as we do with the regular (one stage) Dockerfile:

```
docker build -f Dockerfile.multistage -t nebrass/sample-app-multistaged .
```

If you run it, it's the same docker image like we built in the previous steps, but with less size:

```
$ docker images | grep nebrass
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nebrass/sample-app-jvm	latest	5e1111eeae2b	13 minutes ago	501MB
nebrass/quarkus-multistaged	latest	50591fb707e7	58 minutes ago	199MB

Same application packaged in both of images, but why we have different sizes ?

Before the multi-stage builds era, we used just to write the instructions that will be executed, and every instruction in the Dockerfile add the layer to the image. So even if we had some cleaning instructions, they will have a dedicated weight. In some cases, like many RedHat Docker Images, we are used to create some scripts that we run in just one instruction in the Dockerfile.

After the multistage builds came to the world, while building crossing different stages, the final image won't have any unnecessary content except the required artifact only.

Containerization is not Docker only

Introduction

Docker is not the first publicly available containerization solution in the market. By the time, many other alternatives appeared. Many of them came to resolve some Docker limitations, such as **Podman** and **Buildah**.

What are the Docker limitations ?

After installing Docker, we have a system **docker.service** that will remain running. To check its status, just type the command:

```
$ sudo service docker status

● docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
  Active: active (running) since Thu 2020-08-06 09:23:19 CEST; 8h ago
    TriggeredBy: ● docker.socket
    Docs: https://docs.docker.com
   Main PID: 1659 (dockerd)
     Tasks: 30
    Memory: 3.5G
   CGroup: /system.slice/docker.service
           └─1659 /usr/bin/dockerd -H fd://
--containerd=/run/containerd/containerd.sock
...
...
```



You can also use operating system utilities, such as **sudo systemctl is-active docker** or **sudo status docker**, or checking the service status using Windows Task Manager for example.

This permanently running daemon is the only way to communicate with the Docker Engine. Unfortunately, this is a **Single Point of failure**. This same daemon will be the parent process for all the running containers processes. So, if this daemon is killed or got corrupted, we will lose the communication with the Engine and the running containers processes will remain orphans. Even more, if this process is so greedy with the resources, we cannot complain 😱😱.

This same daemon is requiring root access to be able to do his job. Which can be annoying in many cases where developers are not granted full root rights on their workstations.

These limitations created the opportunity for other tools to start gaining popularity like **Podman**.

Meeting Podman & Buildah

Podman is a **daemonless** Linux native tool designed to make it easy to find, run, build, share and deploy images and containers compliant with the [Open Containers Initiative](#).



podman

Podman is called **daemonless** because it doesn't have any daemon process. **Podman** directly interacts with the image registry, with the container and image storage, and with the Linux kernel through the **runc** container runtime process (which is not a daemon).

What is the runc ?

runc is the Docker's container format and runtime, which Docker donated to the Open Containers Initiative.

It is available now at <https://github.com/opencontainers/runc>.

What is the Open Containers Initiative ?

The **Open Container Initiative** (aka **OCI**) is a lightweight, open governance structure (project), formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtime. The **OCI** was launched on June 22nd 2015 by Docker, CoreOS and other leaders in the container industry

The **OCI** currently contains two specifications: the **Runtime Specification** (runtime-spec) and the **Image Specification** (image-spec). The **Runtime Specification** outlines how to run a "filesystem bundle" that is unpacked on disk. At a high-level an **OCI** implementation would download an **OCI Image** then unpack that image into an **OCI Runtime** filesystem bundle. At this point the **OCI Runtime** Bundle would be run by an **OCI Runtime**.

This entire workflow should support the UX that users have come to expect from container engines like **docker** and **rkt**: primarily, the ability to run an image with no additional arguments:

- `docker run example.com/org/app:v1.0.0`
- `rkt run example.com/org/app,version=v1.0.0`

To support this UX the **OCI Image** Format contains sufficient information to launch the application on the target platform (e.g. command, arguments, environment variables, etc). This specification defines how to create an **OCI Image**, which will generally be done by a build system, and output an image manifest, a filesystem (layer) serialization, and an image configuration. At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem serialization archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The combination of the image manifest, image configuration, and one or more filesystem serializations is called the **OCI Image**.

Podman allows you to do all of the Docker commands without the daemon dependency. We can play the same Docker commands with Podman, just by changing the word **docker** by **podman**. We can even create a **docker** alias for **podman** and everything will work like a charm ! 😊

Containers under the control of Podman can either be run by root or by a non-privileged user. Podman manages the entire container ecosystem which includes pods, containers, container images, and container volumes using the libpod library. Podman specializes in all of the commands and functions that help you to maintain and modify OCI container images, such as pulling and tagging. It allows you to create, run, and maintain those containers and container images in a production environment.

We can do builds with Podman like we are used to with Docker. Podman uses the same code as Buildah for the build process.



You can either:

- build using a **Dockerfile** using **podman build**
- run a container and make lots of changes and then commit those changes to a **new image tag**

Buildah can be described as a superset of commands related to creating and managing container images and, therefore, it has much finer-grained control over images.

The Buildah benefits:

- Powerful control for creating image layers. We can even commit many changes into a single layer, instead of an instruction per layer with the classic Docker World.
- **Buildah CLI** is used to write Image instructions the same way we are creating Linux scripts. If you check the **Buildah CLI** help, you will find that the **CLI commands**, contains the ones that we can have in any Dockerfile instruction:

<code>add</code>	Add content to the container
<code>commit</code>	Create an image from a working container
<code>copy</code>	Copy content into the container
<code>from</code>	Create a working container based on an image
<code>images</code>	List images in local storage
<code>info</code>	Display Buildah system information
<code>mount</code>	Mount a working container's root filesystem
<code>pull</code>	Pull an image from the specified location
<code>push</code>	Push an image to a specified destination
<code>rename</code>	Rename a container
<code>run</code>	Run a command inside of the container
<code>tag</code>	Add an additional name to a local image
<code>umount</code>	Unmount the root file system of the specified working containers
<code>unshare</code>	Run a command in a modified user namespace

Podman becomes a tool that solves two problems:

- It allows operators to examine containers and images with commands they are familiar with using.
- It also provides developers with the same tools they are used to.

So **Docker** users, developers, or operators, can move to **Podman**, do all the fun tasks that they are familiar with from using **Docker**, and do much more. Although, there are many cases where you cannot **Podman** as alternative of **Docker**. For example, in my case, I'm using **TestContainers** to get lightweight instances of **Databases** for my JUnit Tests. This great library has a strong dependency to **Docker** for provisioning **Databases** for Tests, and unfortunately, there is no active tasks for migrating it to **Podman** 😞. You can check this issue [testcontainers-java#2088](#).

Conclusion

The containerization is one of the most powerful technologies that came into the Developers world 🎉. Many other technologies were born and based on the containers, such as **Podman**, **Buildah**, and many others 😎. For sure, we will not use them all, but we will pick all the suitable ones that will help us do our job.

CHAPTER ONE

Introduction to the Monolithic
architecture

Introduction to an actual situation

Nowadays, we are using many methods to access to online applications. For example, we can use the browser or client apps to access to Facebook or Twitter. This ability is offered by their exposed Application programming interfaces (APIs). An API is a software boundary that allows two applications to communicate between each other using a specific protocol (HTTP for example). In the Facebook context, when using the mobile app, while sending a message, the app uses the Facebook APIs for making the request. Next, the application will use its business logic and will make all the necessary databases queries and will finalize the request by returning HTTP responses.

This is the case also of Amazon, Ebay and AliExpress: our main competitors 😊 Yes ! We will be using our Java skills to develop an ecommerce application that will beat them all ! 😊



Yes ! this book will cover (again) an Online Shop use case 😊

Our e-commerce will have many components managing customers, orders, products, reviews, carts, authentication, etc.

We are recruiting many developers for different teams. Every team is dedicated to a specific business domain and every business domain will have it's dedicated code. We have already an architect will be responsible for packaging and building the application.

Our Java application will be deployed as a huge **WAR** file that is wrapping all the code from the different teams and all the external dependencies of the project such as frameworks and libraries. To deliver our application to the Production Team, the project manager has to be sure that all the software teams have delivered all their needed code; if one of the teams had some delays for any reason, all the application will be delayed for sure, as it cannot be delivered with unfinished parts.

→ How can we solve this ? 🤔

We are hearing about agility, does our project can adopt agility ? Can we get benefit from it ?

→ How can we solve this ? 🤔

We know that there are many periods that our e-commerce will be highly loaded by customers, like Black Fridays, Christmas, Valentine's day, etc. So, we asked our Production Team to provide us with high-availability for our application. The solution were to have many running instances of our application to be sure to handle all the load. But does this solution is the best one ? Does our current architecture has benefits in matter of high availability ?

→ How can we solve this ? 🤔

Presenting the context

The actual architecture of our application is **Monolithic**: It consists of a single application that is :

- Simple to design - we can easily design and refactor the components as we have a full view of the full ecosystem
- Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
- Simple to deploy - we simply need to deploy the WAR file (or directory hierarchy) to the appropriate runtime
- Simple to scale - we can scale the application by running multiple copies of the application behind a load balancer

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant:

- The large monolithic code base intimidates developers, especially new ones. The source code can be difficult to read/understand and refactor. As a result, development typically will slow down. It will be difficult to understand how to correctly implement a change, which will decrease the quality of the code.
- Overloaded development machines - large application will cause heavy load on the development machines, which decreases productivity.
- Overloaded servers - large application are hard to monitor and need huge server resources, which impacts productivity.
- Continuous deployment is difficult - a large monolithic application is also hard to deploy. In order to update one component you have to redeploy the entire application. The risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterate rapidly and redeploy frequently.
- Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension: just by creating copies of the monolith. Each copy of application instance will access all the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might be memory intensive. With a monolithic architecture we cannot scale each component independently.
- Obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size it's useful to divide up the engineering organization into teams that focus on specific functional areas. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development and deployment efforts. It is much more difficult for a team to make a change and update production.

How to solve these issues ?

Here we will be talking about **Microservices**: the main subject of this book and one of the most trendy words of nowadays, at least until now, when I am writing this book.. This is what we will be talking about in the next few chapters. Keep tuned ! 😊

I wish you happy reading 😊 Good luck ! 😊

CHAPTER TWO

Coding the Monolithic application

Presenting our domain

In our online shop, the registered customers can purchase products that they found in our catalog. A customer can read and post reviews about articles that he already purchased. To simplify the use-case, for payments we will be using a *Stripe* or *Paypal* as payment gateway.

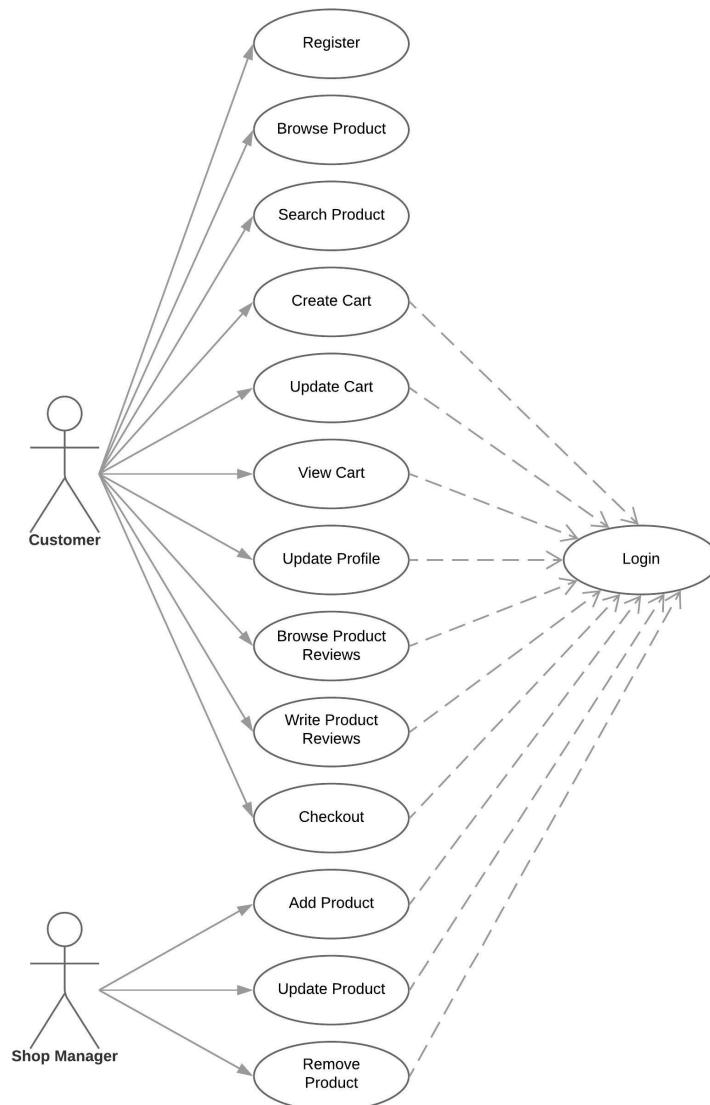


We will not be covering payment transactions in this book.

Use Case Diagram

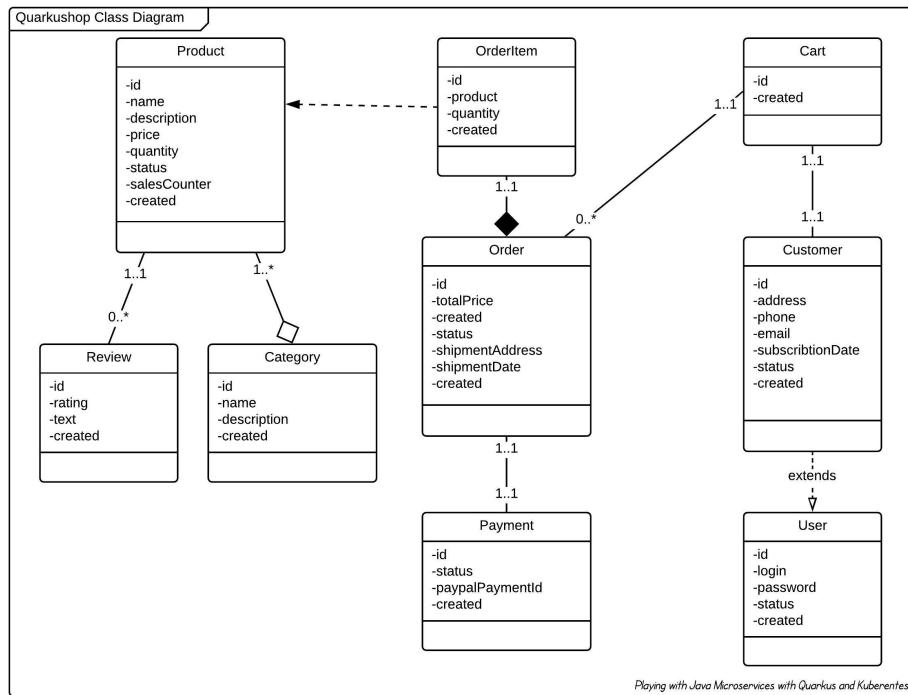
The functionalities offered by the application:

- For the **Customer**: **Search and Browse Product, Browse and Write Reviews, Create, View and Update Cart, Update Profile and Checkout and pay.**
- For the **Shop Manager**: **Add product, Update product and Remove product.**



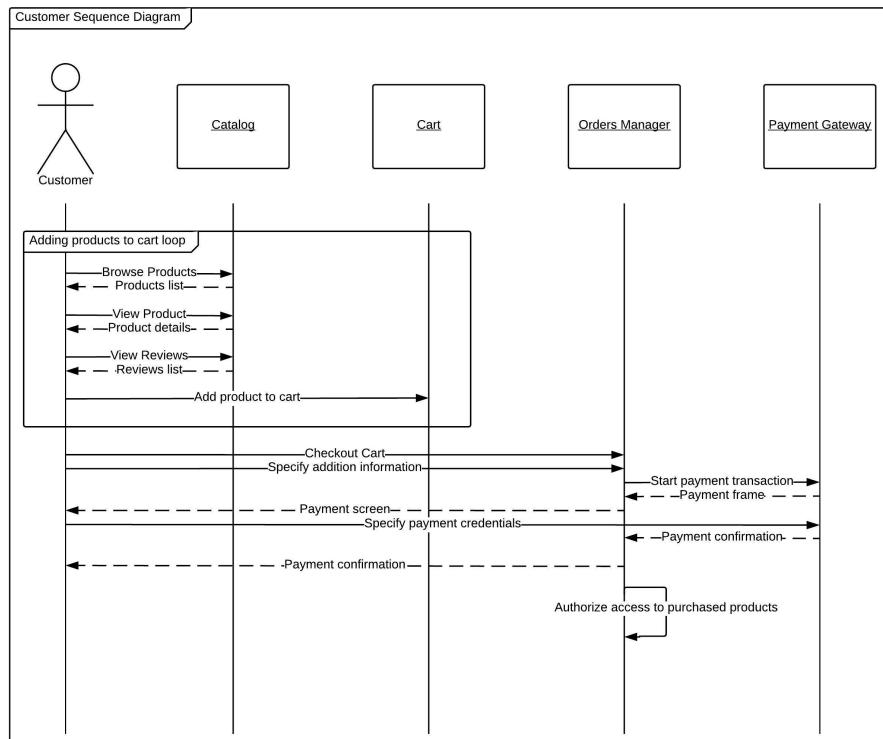
Class Diagram

Our class diagram will look like this:



Sequence Diagram

The sequence diagram for a classic shopping trip will look be:



Coding the application

After we elaborated some UML parts, we can start coding the application.

But before attacking the code, let's list the technical stack that we will be using.

Presenting the technology stack

In this book we are implementing only the backend of an ecommerce application, so our stack will be:

-  **PostgreSQL 13**
-  **Java 11 with GraalVM 20.1.0+**
- **Maven 3.6.2+**
- **Quarkus 1.6.1+**
- Latest  **Docker**

PostgreSQL database

As a requirement, we will need to have a  **PostgreSQL** instance. We will use our  **Docker** skills to have it quickly  just by typing:

```
docker run -d --name demo-postgres \
-e POSTGRES_USER=developer \
-e POSTGRES_PASSWORD=p4SSW0rd \
-e POSTGRES_DB=demo \
-p 5432:5432 postgres:13
```

① We will run the container that we will call **demo-postgres** in detached mode (as daemon in background)

② We will define many environment variables:

- postgres user: **developer**
- postgres password: **p4SSW0rd**
- postgres database: **demo**

③ We will forward the port **5432** → **5432**, and we will use the **official PostgreSQL Image v13**

Java 11

 **Java Standard Edition 11** is a major feature edition, released in *September 25th, 2018*.

 **Java 11** came with:

- JEP 309: Dynamic class-file constants
- JEP 318: Epsilon: a no-op garbage collector
- JEP 323: Local-variable syntax for lambda parameters
- JEP 331: Low-overhead heap profiling
- JEP 321: HTTP client (standard)

- JEP 332: Transport Layer Security (TLS) 1.3
- JEP 328: Flight recorder
- JEP 333: ZGC: a scalable low-latency garbage collector (Experimental)
- JEP 335: Deprecated the Nashorn JavaScript engine
- Unicode 10.0.0 support
- JavaFX, Java EE and CORBA modules have been removed from JDK

Yes, you can think why we used **Java 11** and not **15** (the latest while writing this book) ☺? The choice was made on **Java 11** because it is the maximum version available for GraalVM.

The Quarkus Team recommends using **Java 8** or **11**. I made the choice of **11** because I wanted to get the opportunity to cover newer  **Java SE** version.

Maven

We will be based on the official introduction of Maven from its [official website](#):

Maven, a Yiddish word meaning accumulator of knowledge, was originally started as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects each with their own Ant build files that were all slightly different and JARs were checked into CVS. We wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information and a way to share JARs across several projects.

The result is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features



Quarkus Framework

Quarkus is a full-stack, **Cloud-Native Java framework** made for **JVMs** and **native compilation**, optimizing Java specifically for containers and enabling it to become an effective platform for serverless, cloud, and **Kubernetes** environments.

Quarkus is designed to work with popular **Java standards**, frameworks, and libraries like **Eclipse MicroProfile** and **Spring**, as well as **Apache Kafka**, **RESTEasy (JAX-RS)**, **Hibernate ORM (JPA)**, **Spring Infinispan**, and many more..

The dependency injection mechanism in **Quarkus** is based on **CDI (contexts and dependency injection)** and includes an extension framework to expand functionality and to configure, boot, and integrate a framework into your application. Adding an extension is as easy as adding a maven dependency, or you can use **Quarkus Maven plugin**.

It also provides the correct information to **GraalVM** for **native compilation** of your application.

Quarkus was designed to be easy to use right from the start, with features that work well with little to no configuration.

Developers can choose the **Java frameworks** they want for their applications, which can be run in **JVM mode** or compiled and run in **Native mode**.

Built with an eye to enjoyability for developers, **Quarkus** also includes the following capabilities:

- Live coding so that developers can immediately check the effect of code changes and quickly troubleshoot them
- Unified imperative and reactive programming with an embedded managed event bus
- Unified configuration
- Easy native executable generation



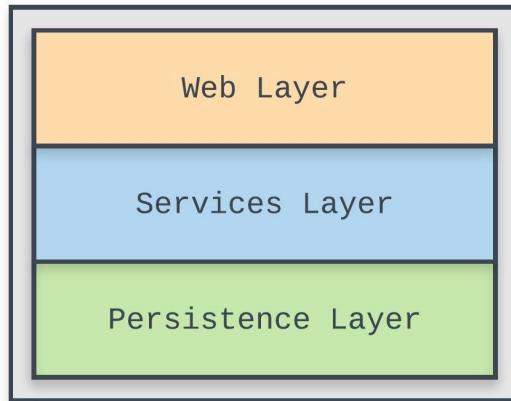
QUARKUS

JetBrains IntelliJ IDEA

Implementing the QuarkuShop

Now, we will start the implementation of our application. We will split the implementation using the layers composing a typical Java EE Application. For this splitting, I will be using an old architectural pattern called **Entity Control Boundary**, initially published by **Ivar Jacobson** in **1992**. This pattern aims to classify each Software Component based on its responsibility.

- **Entity → Persistence Layer** holds the Entities, JPA Repositories and related classes.
- **Control → Services Layer** holds the Services, Configuration, Batches, etc.
- **Boundary → Web Layer** holds the Webservices Endpoints.



Generating the project skull

Here we go ! We will start discovering and using the great features and options offered by the **Quarkus Framework**.

To avoid the hard parts when creating new project and getting it started, the **Quarkus Team** has created the **Code Quarkus** Project: an online tool for generating a **Quarkus** application structure easily. It offers the ability to choose the build tool (Maven or Gradle) and to pick the **Extensions** that you want to add to your project.

Quarkus Extension

Think of **Quarkus Extensions** as the project dependencies. **Extensions** configure, boot and integrate a framework or technology into your **Quarkus** application. They also do all of the heavy lifting of providing the right information to **GraalVM** for your application to compile natively.



If you are used to the **Spring Boot ecosystem**: The **Code Quarkus** is the equivalent of the **Spring Initializr** and the **Quarkus Extension** is the equivalent of the **Spring Boot Starters**.

We can generate a **Quarkus** based application :

- Using a web-based interface code.quarkus.io
 - Using the **Quakus Maven Archetype**, for example:

```
mvn io.quarkus:quarkus-maven-plugin:1.6.1.Final:create \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=getting-started \
  -DclassName="org.acme.getting.started.GreetingResource" \
  -Dpath="/hello"
```

To run the generated application, just run: `mvn quarkus:dev`:

What is the role of `mvn quarkus:dev` ?

We now you can run it using `mvn quarkus:dev`, which **enables hot deployment with background compilation**, which means that **when you modify your Java files or your resource files and refresh your browser these changes will automatically take effect**.

This works too for resource files like the configuration property file.

The act of refreshing the browser triggers a scan of the workspace, and if any changes are detected the Java files are compiled, and the application is redeployed, then your request is serviced by the redeployed application. If there are any issues with compilation or deployment an error page will let you know.

When you go to <http://localhost:8080/index.html>:

Your new Cloud-Native application is ready!

Congratulations, you have created a new Quarkus application.

Why do you see this?

This page is served by Quarkus. The source is in [src/main/resources/META-INF/resources/index.html](#).

What can I do from here?

If not already done, run the application in *dev mode* using: `mvn compile quarkus:dev`.

- Add REST resources, Servlets, functions and other services in [src/main/java](#).
- Your static assets are located in [src/main/resources/META-INF/resources](#).
- Configure your application in [src/main/resources/application.properties](#).

Do you like Quarkus?

Go give it a star on [GitHub](#).

How do I get rid of this page?

Just delete the [src/main/resources/META-INF/resources/index.html](#) file.

Application

GroupId: org.acme
ArtifactId: getting-started
Version: 1.0-SNAPSHOT
Quarkus Version: 1.6.1.Final

Next steps

[Setup your IDE](#)
[Getting started](#)
[Quarkus Web Site](#)

For our **Quarkus** project, we will use the **web interface to generate the project skull**:

The screenshot shows the Quarkus web interface for configuring an application. At the top, it displays the Quarkus logo and navigation links for 'Back to quarkus.io' and 'Available with Enterprise Support'. Below this, the 'Configure your application details' section shows the following configuration:

Group	com.targa.labs
Artifact	quarkushop
Build Tool	Maven
Version	1.0.0-SNAPSHOT
Package Name	com.targa.labs
Quarkus Version	1.6.1.Final

Below this, the 'Generate your application (alt + ⌘)' button is visible. The 'Selected Extensions' section lists several extensions:

- RESTEasy JSON-B
- Hibernate ORM
- Hibernate Validator
- JDBC Driver - PostgreSQL
- Quarkus Extension for Spring Data JPA API (PREVIEW)
- Flyway
- SmallRye OpenAPI

The 'Web' section contains a list of checked extensions:

- RESTEasy JAX-RS (INCLUDED)
- RESTEasy JSON-B
- Hibernate Validator
- REST Client
- REST Client JAXB
- REST Client JSON-B
- REST Client Jackson
- REST resources for Hibernate ORM with P... (EXPERIMENTAL)
- RESTEasy JAXB
- RESTEasy Mutiny (PREVIEW)

Each extension entry includes a brief description and a 'More' button.

We will pick some **Extensions**:

- **RESTEasy JAX-RS** ← automatically included in every generated project
- **RESTEasy JSON-B**
- **SmallRye OpenAPI**
- **Hibernate ORM**
- **Hibernate Validator**
- **JDBC Driver - PostgreSQL**
- **Quarkus Extension for Spring Data JPA API**
- **Flyway**

What is these extensions?

- **RESTEasy JSON-B** : Adds the **JSON-B** serialization Library support for **RESTEasy**
- **SmallRye OpenAPI** : Document your **REST APIs** based on the **OpenAPI Specification** – comes with **Swagger UI**
- **Hibernate ORM** : Adds all the requirements to define the persistent model with **Hibernate ORM** as **JPA Implementation**
- **Hibernate Validator** : Adds the mechanisms for validating the input/output of the **REST APIs** and/or the parameters and return values of the methods of your business services
- **JDBC Driver - PostgreSQL** : Adds all the requirement to help you connect to the **PostgreSQL** database via **JDBC**
- **Quarkus Extension for Spring Data JPA API** : Brings the **Spring Data JPA** to **Quarkus** to create your **Data Access Layer** as you are used to in **Spring Boot**
- **Flyway**: Handle the database schema migrations

We need to add **Lombok** to our **pom.xml**: **Project Lombok** is a Java library that automatically plugs into your editor and build tools, spicing up your Java. Lombok saves time and effort of writing boilerplate code like Getters/Setters/Constructors/etc.

The **Lombok** Maven dependency:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
</dependency>
```



As we walk into our example we will add more dependencies when we need them.

Once we created the blank project, we will start building our monolithic layers. We will begin by the **Persistence Layer**.

Creating the Persistence Layer

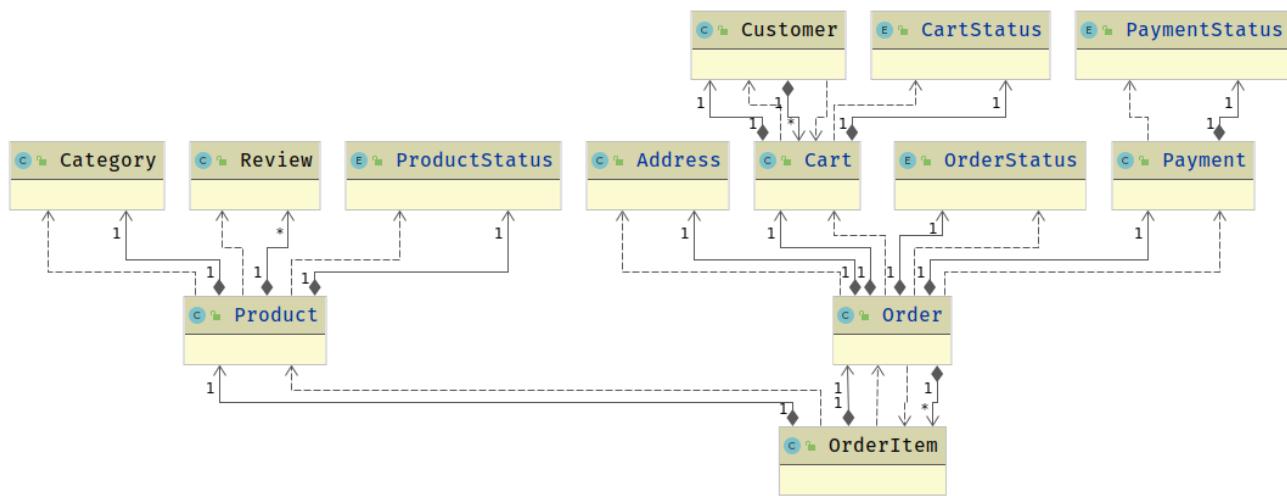
If we look back to our class diagram, we have this **Entity** classes:

- Address
- Cart
- Category
- Customer
- Order
- OrderItem
- Payment
- Product
- Review

With some **Enumerations**:

- CartStatus
- OrderStatus
- ProductStatus
- PaymentStatus

This diagram illustrates the relation between the classes:



For every **entity** of the list, we will create:

- A **JPA Entity**
- A **Spring Data JPA Repository**: we will enjoy our **Spring Boot** skills to create components in **Quarkus**

Our entities will share some attributes that are always commonly used, like **id**, **created date**, etc. These attributes will be located in the **AbstractEntity** class that will be **extended** by our entities.

The **AbstractEntity** will look like:

```
@Getter ①
@Setter ①
@MappedSuperclass ②
@EntityListeners(AuditingEntityListener.class) ③
public abstract class AbstractEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "created_date", nullable = false)
    private Instant createdDate;

    @Column(name = "last_modified_date")
    private Instant lastModifiedDate;
}
```

- ① We are using **Lombok** annotations to generate getters and setters for the **AbstractEntity** class.
- ② Declares this class as **JPA base class** which holds properties will be inherited by the subclasses entities.
- ③ Activates the **Auditing** the **Entity** using the **AuditingEntityListener** class:

```
public class AuditingEntityListener {
    @PrePersist ①
    void preCreate(AbstractEntity auditable) {
        Instant now = Instant.now();
        auditable.setCreatedDate(now);
        auditable.setLastModifiedDate(now);
    }

    @PreUpdate ②
    void preUpdate(AbstractEntity auditable) {
        Instant now = Instant.now();
        auditable.setLastModifiedDate(now);
    }
}
```

- ① Specifies that the annotated method will be invoked before **Persisting** the entity in the database.
- ② Specifies that the annotated method will be invoked before **Updating** the entity in the database.

Cart

The class **Cart** entity will look like:

```

@Getter ①
@Setter ①
@NoArgsConstructor ②
@ToString(callSuper = true) ③
@Entity ④
@Table(name = "carts") ④
public class Cart extends AbstractEntity {

    @ManyToOne
    private final Customer customer;

    @NotNull ⑤
    @Column(nullable = false) ⑥
    @Enumerated(EnumType.STRING)
    private final CartStatus status;

    public Cart(Customer customer, @NotNull CartStatus status) {
        this.customer = customer;
        this.status = status;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Cart cart = (Cart) o;
        return Objects.equals(customer, cart.customer) &&
               status == cart.status;
    }

    @Override
    public int hashCode() {
        return Objects.hash(customer, status);
    }
}

```

① This **Lombok** annotation generates getters/setters for all fields.

② This **Lombok** annotation generates a no-args constructor, which is **required by JPA**.

③ This **Lombok** annotation generates the **toString()** method based on the current class fields and including the superclass fields.

④ This is an **@Entity** and its corresponding **@Table** will be named **carts**.

⑤ **Validation annotation** used to control the integrity of the data.

⑥ Column definition:

name, length and a definition of a **Nullability Constraint**.



The difference between the **Validation annotations** and **Constraints** defined in the **@Column**, is that the **Validation annotations** is application-scoped and the **Constraints** are DB scoped.

The **CartRepository** will be:

```
@Repository ①
public interface CartRepository extends JpaRepository<Cart, Long> { ②

    List<Cart> findByStatus(CartStatus status); ③

    List<Cart> findByStatusAndCustomerId(CartStatus status, Long customerId); ③
}
```

- ① Indicates that an annotated class is a "**Repository**", originally defined by **Domain-Driven Design** (Eric Evans, 2003) as "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects".
- ② JPA specific extension of a **Repository** in **Spring Data JPA**. This will enable **Spring Data** to find this interface and automatically create an implementation for it.
- ③ These methods are implementing queries automatically using **Spring Data Query Methods Builder Mecanism**.

What is a Spring Data JpaRepository ?

JpaRepository extends **PagingAndSortingRepository** which in turn extends **CrudRepository**. Their main functions are:

- **CrudRepository** mainly provides CRUD functions.
- **PagingAndSortingRepository** provide methods to do pagination and sorting records.
- **JpaRepository** provides some JPA related method such as flushing the persistence context and delete record in a batch.

Because of the inheritance mentioned above, **JpaRepository** will have all the functions of **CrudRepository** and **PagingAndSortingRepository**. So if you don't need the repository to have the functions provided by **JpaRepository** and **PagingAndSortingRepository**, use **CrudRepository**.

What is Spring Data Query Methods Builder Mechanism?

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find...By`, `read...By`, `query...By`, `count...By`, and `get...By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions, such as a `Distinct` to set a distinct flag on the query to be created. However, the first `By` acts as delimiter to indicate the start of the actual criteria. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or`.



No NEED to write custom JPQL queries. This is why I required the use of the **Quarkus Extension for Spring Data JPA API** to enjoy these powerful features of **Spring Data JPA**.

CartStatus

The enumeration class `CartStatus` will look like:

```
public enum CartStatus {  
    NEW, CANCELED, CONFIRMED  
}
```

Address

The class **Address** will look like:

```

@Getter
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Embeddable
public class Address {

    @Column(name = "address_1")
    private String address1;

    @Column(name = "address_2")
    private String address2;

    @Column(name = "city")
    private String city;

    @NotNull
    @Size(max = 10)
    @Column(name = "postcode", length = 10, nullable = false)
    private String postcode;

    @NotNull
    @Size(max = 2)
    @Column(name = "country", length = 2, nullable = false)
    private String country;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Address address = (Address) o;
        return Objects.equals(address1, address.address1) &&
               Objects.equals(address2, address.address2) &&
               Objects.equals(city, address.city) &&
               Objects.equals(postcode, address.postcode) &&
               Objects.equals(country, address.country);
    }

    @Override
    public int hashCode() {
        return Objects.hash(address1, address2, city, postcode, country);
    }
}

```

The class **Address** will be used as **Embeddable** class. Embeddable classes are used to represent the state of an entity but don't have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it. Embeddable classes exist only as the state of another entity.

Category

The **Category** entity:

```

@Getter
@NoArgsConstructor
@ToString(callSuper = true)
@Entity
@Table(name = "categories")
public class Category extends AbstractEntity {

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "description", nullable = false)
    private String description;

    public Category(@NotNull String name, @NotNull String description) {
        this.name = name;
        this.description = description;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Category category = (Category) o;
        return Objects.equals(name, category.name) &&
               Objects.equals(description, category.description);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, description);
    }
}

```

The **CategoryRepository**:

```
@Repository
public interface CategoryRepository extends JpaRepository<Category, Long> {
}
```

Customer

The **Customer** entity:

```
@Getter @Setter
@NoArgsConstructor
@ToString(callSuper = true)
@Entity
@Table(name = "customers")
public class Customer extends AbstractEntity {

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Email
    @Column(name = "email")
    private String email;

    @Column(name = "telephone")
    private String telephone;

    @OneToMany(mappedBy = "customer")
    private Set<Cart> carts;

    @Column(name = "enabled", nullable = false)
    private Boolean enabled;

    public Customer(String firstName, String lastName, @Email String email,
                    String telephone, Set<Cart> carts, Boolean enabled) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.telephone = telephone;
        this.carts = carts;
        this.enabled = enabled;
    }
}
```

```

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Customer customer = (Customer) o;
    return Objects.equals(firstName, customer.firstName) &&
        Objects.equals(lastName, customer.lastName) &&
        Objects.equals(email, customer.email) &&
        Objects.equals(telephone, customer.telephone) &&
        Objects.equals(carts, customer.carts) &&
        Objects.equals(enabled, customer.enabled);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName, email, telephone, enabled);
}
}

```

The **CustomerRepository**:

```

@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
    List<Customer> findAllByEnabled(Boolean enabled);
}

```

Order

The **Order** entity:

```

@Getter
@Setter
@NoArgsConstructor
@ToString(callSuper = true)
@Entity
@Table(name = "orders")
public class Order extends AbstractEntity {

    @NotNull
    @Column(name = "total_price", precision = 10, scale = 2, nullable = false)
    private BigDecimal price;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private OrderStatus status;
}

```

```

@Column(name = "shipped")
private ZonedDateTime shipped;

@OneToOne(cascade = CascadeType.REMOVE)
@JoinColumn(unique = true)
private Payment payment;

@Embedded
private Address shipmentAddress;

@OneToMany(mappedBy = "order", fetch = FetchType.LAZY, cascade = CascadeType.REMOVE)
private Set<OrderItem> orderItems;

@OneToOne
private Cart cart;

public Order(@NotNull BigDecimal price, @NotNull OrderStatus status,
            ZonedDateTime shipped, Payment payment, Address shipmentAddress,
            Set<OrderItem> orderItems, Cart cart) {
    this.price = price;
    this.status = status;
    this.shipped = shipped;
    this.payment = payment;
    this.shipmentAddress = shipmentAddress;
    this.orderItems = orderItems;
    this.cart = cart;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Order order = (Order) o;
    return Objects.equals(price, order.price) && status == order.status &&
           Objects.equals(shipped, order.shipped) &&
           Objects.equals(payment, order.payment) &&
           Objects.equals(shipmentAddress, order.shipmentAddress) &&
           Objects.equals(orderItems, order.orderItems) &&
           Objects.equals(cart, order.cart);
}

@Override
public int hashCode() {
    return Objects.hash(price, status, shipped, payment, shipmentAddress, cart);
}
}

```

The [OrderRepository](#):

```
@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {
    List<Order> findByCartCustomerId(Long customerId);
    Optional<Order> findByPaymentId(Long id);
}
```

OrderItem

The [OrderItem](#) entity:

```
@Getter @NoArgsConstructor
@ToString(callSuper = true)
@Entity @Table(name = "order_items")
public class OrderItem extends AbstractEntity {

    @NotNull
    @Column(name = "quantity", nullable = false)
    private Long quantity;

    @ManyToOne(fetch = FetchType.LAZY)
    private Product product;

    @ManyToOne(fetch = FetchType.LAZY)
    private Order order;

    public OrderItem(@NotNull Long quantity, Product product, Order order) {
        this.quantity = quantity;
        this.product = product;
        this.order = order;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        OrderItem orderItem = (OrderItem) o;
        return Objects.equals(quantity, orderItem.quantity) &&
            Objects.equals(product, orderItem.product) &&
            Objects.equals(order, orderItem.order);
    }

    @Override
    public int hashCode() { return Objects.hash(quantity, product, order); }
}
```

The `OrderItemRepository`:

```
@Repository
public interface OrderItemRepository extends JpaRepository<OrderItem, Long> {
    List<OrderItem> findAllByOrderId(Long id);
}
```

Payment

The `Payment` entity:

```
@Getter @NoArgsConstructor
@ToString(callSuper = true)
@Entity @Table(name = "payments")
public class Payment extends AbstractEntity {

    @Column(name = "paypal_payment_id")
    private String paypalPaymentId;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private PaymentStatus status;

    @NotNull
    @Column(name = "amount", nullable = false)
    private BigDecimal amount;

    public Payment(String paypalPaymentId, @NotNull PaymentStatus status, @NotNull
BigDecimal amount) {
        this.paypalPaymentId = paypalPaymentId;
        this.status = status;
        this.amount = amount;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Payment payment = (Payment) o;
        return Objects.equals(paypalPaymentId, payment.paypalPaymentId);
    }

    @Override
    public int hashCode() { return Objects.hash(paypalPaymentId); }
}
```

The **PaymentRepository**:

```
@Repository
public interface PaymentRepository extends JpaRepository<Payment, Long> {
    List<Payment> findAllByAmountBetween(BigDecimal min, BigDecimal max);
}
```

The **PaymentStatus**:

```
public enum PaymentStatus {
    ACCEPTED, PENDING, REFUSED, ERROR
}
```

Product

The **Product** entity:

```
@Getter
@NoArgsConstructor
@ToString(callSuper = true)
@Entity
@Table(name = "products")
public class Product extends AbstractEntity {

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Column(name = "description", nullable = false)
    private String description;

    @NotNull
    @Column(name = "price", precision = 10, scale = 2, nullable = false)
    private BigDecimal price;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private ProductStatus status;

    @Column(name = "sales_counter")
    private Integer salesCounter;
```

```

@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.REMOVE)
@JoinTable(name = "products_reviews",
           joinColumns = @JoinColumn(name = "product_id"),
           inverseJoinColumns = @JoinColumn(name = "reviews_id"))
private Set<Review> reviews = new HashSet<>();

@ManyToOne
@JoinColumn(name = "category_id")
private Category category;

public Product(@NotNull String name, @NotNull String description,
               @NotNull BigDecimal price, @NotNull ProductStatus status,
               Integer salesCounter, Set<Review> reviews, Category category) {
    this.name = name;
    this.description = description;
    this.price = price;
    this.status = status;
    this.salesCounter = salesCounter;
    this.reviews = reviews;
    this.category = category;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Product product = (Product) o;
    return Objects.equals(name, product.name) &&
           Objects.equals(description, product.description) &&
           Objects.equals(price, product.price) && status == product.status &&
           Objects.equals(salesCounter, product.salesCounter) &&
           Objects.equals(reviews, product.reviews) &&
           Objects.equals(category, product.category);
}

@Override
public int hashCode() {
    return Objects.hash(name, description, price, category);
}
}

```

The **ProductRepository**:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findByCategoryId(Long categoryId);

    Long countAllByCategoryId(Long categoryId);

    @Query("select p from Product p JOIN p.reviews r WHERE r.id = ?1")
    Product findProductByReviewId(Long reviewId);

    void deleteAllByCategoryId(Long id);

    List<Product> findAllByCategoryId(Long id);
}
```

ProductStatus

The enumeration class **ProductStatus** will look like:

```
public enum ProductStatus {
    AVAILABLE, DISCONTINUED
}
```

Review

The **Review** entity:

```

@getter
@NoArgsConstructor
@ToString(callSuper = true)
@Entity
@Table(name = "reviews")
public class Review extends AbstractEntity {

    @NotNull
    @Column(name = "title", nullable = false)
    private String title;

    @NotNull
    @Column(name = "description", nullable = false)
    private String description;

    @NotNull
    @Column(name = "rating", nullable = false)
    private Long rating;

    public Review(@NotNull String title, @NotNull String description, @NotNull Long
rating) {
        this.title = title;
        this.description = description;
        this.rating = rating;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Review review = (Review) o;
        return Objects.equals(title, review.title) &&
               Objects.equals(description, review.description) &&
               Objects.equals(rating, review.rating);
    }

    @Override
    public int hashCode() {
        return Objects.hash(title, description, rating);
    }
}

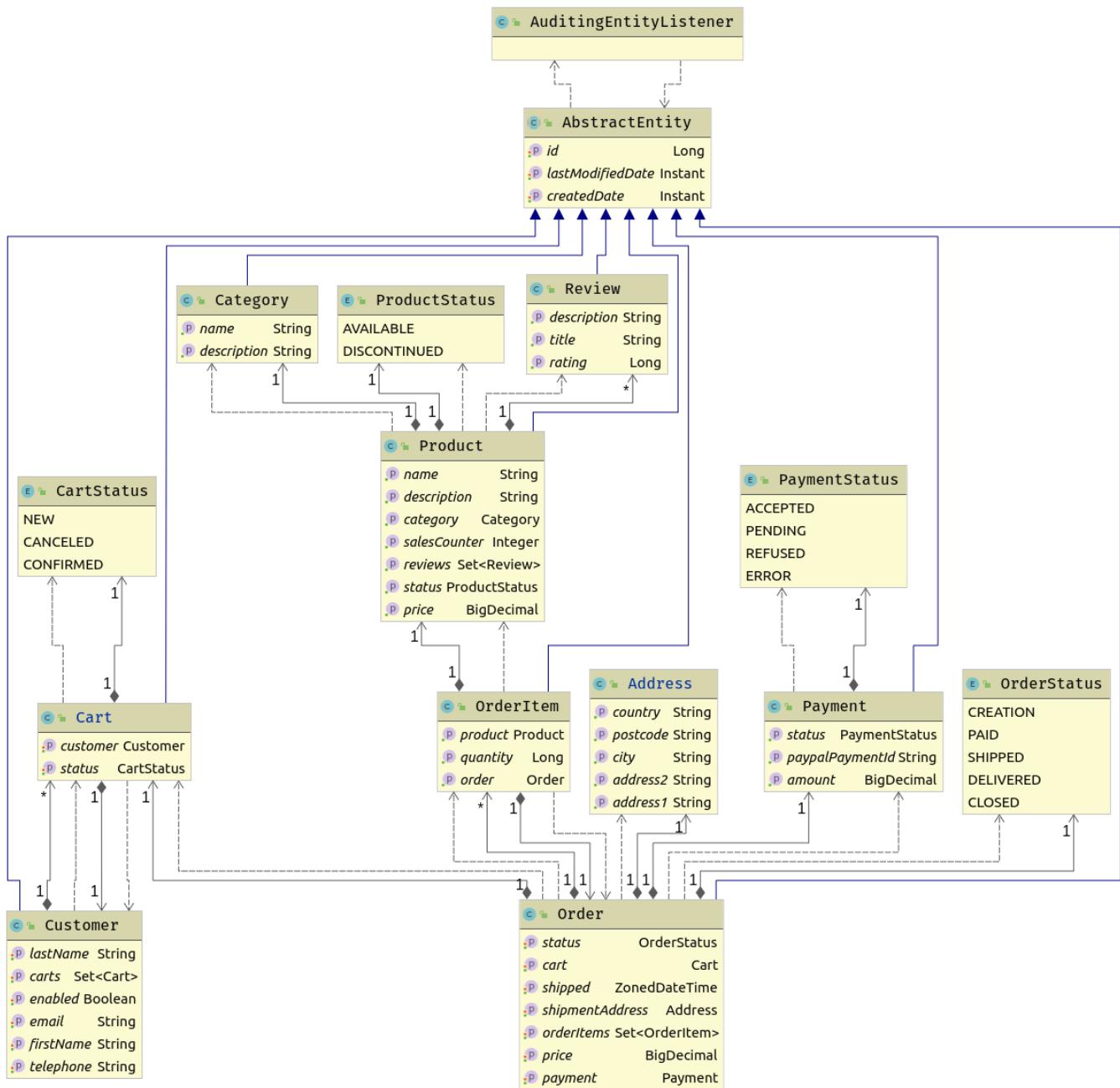
```

The **ReviewRepository**:

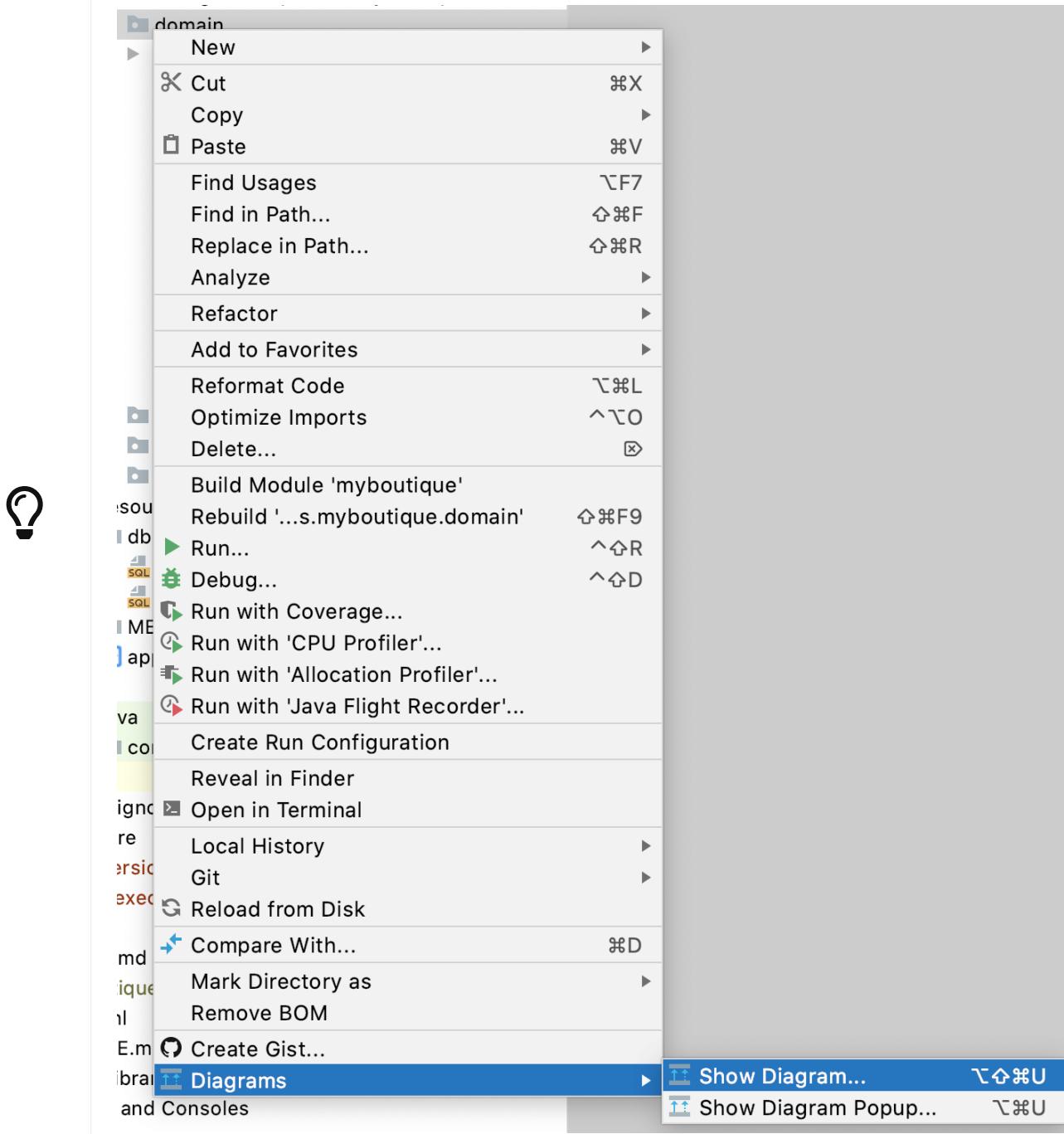
```
@Repository
public interface ReviewRepository extends JpaRepository<Review, Long> {

    @Query("select p.reviews from Product p where p.id = ?1")
    List<Review> findReviewsByProductId(Long id);
}
```

At this stage, we finished creating our **Entities** and **Repositories**. Our **Entities** diagram looks like:



I generated this diagram using **IntelliJ IDEA**. To generate one, just right click on the package containing the targeted classes and choose **Diagrams** → **Show Diagram**. Next, in the list that opens, select **Java Class Diagram**:



Now, we need to provide our Quarkus application with Hibernate/JPA configuration and the database credentials.

Just like **Spring Boot**, **Quarkus** stores its configuration and properties inside **application.properties** file located in **src/main/resources**.

The **application.properties** will store the properties locally. We can override these properties in many ways, for example, thru **environment variables**.

QuarkuShop will need two kinds of configurations now:

- **The Datasource configuration:** all the needed properties and credentials to access the database, like the *Driver Type, URL, username, password, schema name*:

```
# Datasource config properties
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=developer
quarkus.datasource.password=p4SSW0rd
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/demo
```



These credentials are the credentials of our **Dockerized PostgreSQL Database** that we just created before we start hitting the source code 😊

- **The Flyway configuration:** if you go back to the extensions that we selected while generation our application, we picked **Flyway**, which we will be used for versioning the database. So we need to activate it in our application.

```
# Flyway minimal config properties
quarkus.flyway.migrate-at-start=true
```

What is Database Versioning ?

Versioning a database means sharing all changes of a database that are necessary for the application to run properly. Database versioning starts with an initial database schema and optionally with some data. When there is any database changes in a new release of the application, we ship a new patch-file to perform changes on an existing database instead of starting from scratch with an updated dump, so when deploying the new version, the application will run properly. A patch-file describes how to transform an existing database to a new state and how to revert them in order to get back to the old state.

In our case, we will have two **Flyway** scripts - these scripts must be located in the default Flyway folder **src/main/resources/db/migration**:

- **src/main/resources/db/migration/V1.0__Init_app.sql**: Initial skeleton creation script: contains the **SQL queries** to create the tables and their constraints that we already defined in the source code like **Primary Keys, Foreign Keys** between entities and **non nullable columns**. This **Flyway** script also will bring the **Hibernate SQL sequence** needed by our ORM to provide IDs for entities while persisted.
- **src/main/resources/db/migration/V1.1__Insert_samples.sql**: Initial sample data insertion script: contains the sample data that we will insert into our database in order to have sample data to use during the execution to easily use the application.



Notice the versioning counting that we used **V1.0** and **V1.1** - This is used to guarantee the execution order of the **Flyway** scripts.

Now we can move to the next layer: the **Service** layer.

Creating the Service Layer

Now we created our entities, we will create the services.

A **service** in a the component that wraps the **Business Logic**.

At this point, we didn't discussed any **Business Logic**, we just have **CRUD** operations. We will implement these **CRUD** operations in our services.

We will have a separated **service** per **entity** to apply the single responsibility practice for our services.

The **Service** layer is also the glue between the **Persistence** and the **Web** layers. A **Service** will grab the data from the **Repositories** and will apply its business logic on the loaded data, and will encapsulate the calculated data into a wrapper used to transfer the data between the **Service** and the **Web** layers. This wrapper is called **Data Transfer Object** or **DTO**.

Do we really need DTOs ?

We always ask the question: do we need really DTOs in our application ?

The answer is **YES in many cases**.

Let's imagine the case that we have a **Service** that lists the **Users** available in our database. If we don't use the DTOs and instead of it we send back the **User** class, we will transfer to the Webservices, and to the caller behind, the credentials of our **Users** as the password is an encapsulated field of the **User** entity.

Typical Service: CartService

The **CartService** will look like:

```
@Slf4j ①
@ApplicationScoped ②
@Transactional ③
public class CartService {

    @Inject ④
    CartRepository cartRepository;

    @Inject ④
    CustomerRepository customerRepository;

    public List<CartDto> findAll() {
        log.debug("Request to get all Carts");
        return this.cartRepository.findAll()
            .stream()
            .map(CartService::mapToDto)
            .collect(Collectors.toList());
    }
}
```

```

public List<CartDto> findAllActiveCarts() {
    return this.cartRepository.findByStatus(CartStatus.NEW)
        .stream()
        .map(CartService::mapToDto)
        .collect(Collectors.toList());
}

public Cart create(Long customerId) {
    if (this.getActiveCart(customerId) == null) {
        var customer =
            this.customerRepository.findById(customerId).orElseThrow(() ->
                new IllegalStateException("The Customer does not exist!"));

        var cart = new Cart(customer, CartStatus.NEW);

        return this.cartRepository.save(cart);
    } else {
        throw new IllegalStateException("There is already an active cart");
    }
}

public CartDto createDto(Long customerId) {
    return mapToDto(this.create(customerId));
}

@Transactional(SUPPORTS)
public CartDto findById(Long id) {
    log.debug("Request to get Cart : {}", id);
    return this.cartRepository.findById(id).map(CartService::mapToDto).orElse(null);
}

public void delete(Long id) {
    log.debug("Request to delete Cart : {}", id);
    Cart cart = this.cartRepository.findById(id)
        .orElseThrow(() -> new IllegalStateException("Cannot find cart with id " +
+ id));

    cart.setStatus(CartStatus.CANCELED);

    this.cartRepository.save(cart);
}

public CartDto getActiveCart(Long customerId) {
    List<Cart> carts = this.cartRepository

```

```

    .findByStatusAndCustomerId(CartStatus.NEW, customerId);
    if (carts != null) {

        if (carts.size() == 1) {
            return mapToDto(carts.get(0));
        }
        if (carts.size() > 1) {
            throw new IllegalStateException("Many active carts detected !!!");
        }
    }
    return null;
}

public static CartDto mapToDto(Cart cart) {
    return new CartDto(
        cart.getId(),
        CustomerService.mapToDto(cart.getCustomer()),
        cart.getStatus().name()
    );
}
}

```

- ① Lombok annotation used to generate a **logger** in the class. When used, you then have a **static final log** field, initialized to the name of your class, which you can then use to write log statements.
- ② Specifies that this class is **application scoped**.
- ③ **@Transactional** annotation provides the application the ability to declaratively control transaction boundaries.
- ④ The most famous annotation in the **Java EE world** ! Used to request an instance of the annotated field type.

And the **CartDto** class will look like:

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class CartDto {
    private Long id;
    private CustomerDto customer;
    private String status;
}

```

AddressService

The **AddressService** class will look like:

```
@ApplicationScoped
public class AddressService {

    public static Address createFromDto(AddressDto addressDto) {
        return new Address(
            addressDto.getAddress1(),
            addressDto.getAddress2(),
            addressDto.getCity(),
            addressDto.getPostcode(),
            addressDto.getCountry()
        );
    }

    public static AddressDto mapToDto(Address address) {
        return new AddressDto(
            address.getAddress1(),
            address.getAddress2(),
            address.getCity(),
            address.getPostcode(),
            address.getCountry()
        );
    }
}
```

And the **AddressDto** class will look like:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AddressDto {
    private String address1;
    private String address2;
    private String city;
    private String postcode;
    @Size(min = 2, max = 2)
    private String country;
}
```

CategoryService

The **CategoryService** class will look like:

```

@Slf4j
@ApplicationScoped
@Transactional
public class CategoryService {
    @Inject
    CategoryRepository categoryRepository;
    @Inject
    ProductRepository productRepository;

    public static CategoryDto mapToDto(Category category, Long productsCount) {
        return new CategoryDto(
            category.getId(),
            category.getName(),
            category.getDescription(),
            productsCount);
    }

    public List<CategoryDto> findAll() {
        log.debug("Request to get all Categories");
        return this.categoryRepository.findAll()
            .stream().map(category ->
                mapToDto(category,
                    productRepository
                        .countAllByCategoryId(category.getId())))
            .collect(Collectors.toList());
    }

    public CategoryDto findById(Long id) {
        log.debug("Request to get Category : {}", id);
        return this.categoryRepository.findById(id).map(category ->
            mapToDto(category,
                productRepository
                    .countAllByCategoryId(category.getId())))
            .orElse(null);
    }

    public CategoryDto create(CategoryDto categoryDto) {
        log.debug("Request to create Category : {}", categoryDto);
        return mapToDto(this.categoryRepository
            .save(new Category(
                categoryDto.getName(),
                categoryDto.getDescription()))
            , 0L);
    }
}

```

```
public void delete(Long id) {  
    log.debug("Request to delete Category : {}", id);  
    log.debug("Deleting all products for the Category : {}", id);  
    this.productRepository.deleteAllByCategoryId(id);  
    log.debug("Deleting Category : {}", id);  
    this.categoryRepository.deleteById(id);  
}  
  
public List<ProductDto> findProductsByCategoryId(Long id) {  
    return this.productRepository.findAllByCategoryId(id)  
        .stream()  
        .map(ProductService::mapToDto)  
        .collect(Collectors.toList());  
}  
}
```

And the **CategoryDto** class will look like:

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class CategoryDto {  
    private Long id;  
    private String name;  
    private String description;  
    private Long products;  
}
```

CustomerService

The **CustomerService** class will look like:

```

@Slf4j
@ApplicationScoped
@Transactional
public class CustomerService {

    @Inject
    CustomerRepository customerRepository;

    public CustomerDto create(CustomerDto customerDto) {
        log.debug("Request to create Customer : {}", customerDto);
        return mapToDto(this.customerRepository.save(
            new Customer(customerDto.getFirstName(),
                customerDto.getLastName(),
                customerDto.getEmail(),
                customerDto.getTelephone(),
                Collections.emptySet(),
                Boolean.TRUE)
        ));
    }

    public List<CustomerDto> findAll() {
        log.debug("Request to get all Customers");
        return this.customerRepository.findAll()
            .stream()
            .map(CustomerService::mapToDto)
            .collect(Collectors.toList());
    }

    @Transactional
    public CustomerDto findById(Long id) {
        log.debug("Request to get Customer : {}", id);
        return this.customerRepository.findById(id)
            .map(CustomerService::mapToDto).orElse(null);
    }

    public List<CustomerDto> findAllActive() {
        log.debug("Request to get all active customers");
        return this.customerRepository.findAllByEnabled(true)
            .stream().map(CustomerService::mapToDto)
            .collect(Collectors.toList());
    }
}

```

```

public List<CustomerDto> findAllInactive() {
    log.debug("Request to get all inactive customers");
    return this.customerRepository.findAllByEnabled(false)
        .stream().map(CustomerService::mapToDto)
        .collect(Collectors.toList());
}

public void delete(Long id) {
    log.debug("Request to delete Customer : {}", id);

    Customer customer = this.customerRepository.findById(id)
        .orElseThrow(() ->
            new IllegalStateException("Cannot find Customer with id " + id));

    customer.setEnabled(false);
    this.customerRepository.save(customer);
}

public static CustomerDto mapToDto(Customer customer) {
    return new CustomerDto(customer.getId(),
        customer.getFirstName(),
        customer.getLastName(),
        customer.getEmail(),
        customer.getTelephone()
    );
}
}

```

And the `CustomerDto` class will look like:

```

@Data
@NoArgsConstructor
@AllArgsConstructor
public class CustomerDto {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String telephone;
}

```

OrderItemService

The **OrderItemService** class will look like:

```

@Slf4j
@ApplicationScoped
@Transactional
public class OrderItemService {

    @Inject
    OrderItemRepository orderItemRepository;
    @Inject
    OrderRepository orderRepository;
    @Inject
    ProductRepository productRepository;

    public static OrderItemDto mapToDto(OrderItem orderItem) {
        return new OrderItemDto(
            orderItem.getId(),
            orderItem.getQuantity(),
            orderItem.getProduct().getId(),
            orderItem.getOrder().getId()
        );
    }

    public OrderItemDto findById(Long id) {
        log.debug("Request to get OrderItem : {}", id);
        return this.orderItemRepository.findById(id)
            .map(OrderItemService::mapToDto).orElse(null);
    }

    public OrderItemDto create(OrderItemDto orderItemDto) {
        log.debug("Request to create OrderItem : {}", orderItemDto);
        var order =
            this.orderRepository
                .findById(orderItemDto.getOrderId())
                .orElseThrow(() ->
                    new IllegalStateException("The Order does not exist!"));

        var product =
            this.productRepository
                .findById(orderItemDto.getProductId())
                .orElseThrow(() ->
                    new IllegalStateException("The Product does not exist!"));
    }
}

```

```

var orderItem = this.orderItemRepository.save(
    new OrderItem(
        orderItemDto.getQuantity(),
        product,
        order
    ));
order.setPrice(order.getPrice().add(orderItem.getProduct().getPrice()));
this.orderRepository.save(order);

return mapToDto(orderItem);
}

public void delete(Long id) {
    log.debug("Request to delete OrderItem : {}", id);

    var orderItem = this.orderItemRepository.findById(id)
        .orElseThrow(() ->
            new IllegalStateException("The OrderItem does not exist!"));

    var order = orderItem.getOrder();
    order.setPrice(order.getPrice().subtract(orderItem.getProduct().getPrice()));

    this.orderItemRepository.deleteById(id);

    order.getOrderItems().remove(orderItem);

    this.orderRepository.save(order);
}

public List<OrderItemDto> findByOrderId(Long id) {
    log.debug("Request to get all OrderItems of OrderId {}", id);
    return this.orderItemRepository.findAllByOrderId(id)
        .stream()
        .map(OrderItemService::mapToDto)
        .collect(Collectors.toList());
}
}

```

And the `OrderItemDto` class will look like:

```
@Data @NoArgsConstructor @AllArgsConstructor
public class OrderItemDto {
    private Long id;
    private Long quantity;
    private Long productId;
    private Long orderId;
}
```

OrderService

The `OrderService` class will look like:

```
@Slf4j
@ApplicationScoped @Transactional
public class OrderService {

    @Inject OrderRepository orderRepository;
    @Inject PaymentRepository paymentRepository;
    @Inject CartRepository cartRepository;

    public List<OrderDto> findAll() {
        log.debug("Request to get all Orders");
        return this.orderRepository.findAll().stream().map(OrderService::mapToDto)
            .collect(Collectors.toList());
    }

    public OrderDto findById(Long id) {
        log.debug("Request to get Order : {}", id);
        return this.orderRepository.findById(id)
            .map(OrderService::mapToDto).orElse(null);
    }

    public List<OrderDto> findAllByUser(Long id) {
        return this.orderRepository.findByCartCustomerId(id)
            .stream().map(OrderService::mapToDto).collect(Collectors.toList());
    }

    public OrderDto create(OrderDto orderDto) {
        log.debug("Request to create Order : {}", orderDto);

        Long cartId = orderDto.getCart().getId();
        Cart cart = this.cartRepository.findById(cartId)
            .orElseThrow(() -> new IllegalStateException(
                "The Cart with ID[" + cartId + "] was not found !"));
    }
}
```

```

    return mapToDto(this.orderRepository.save(new Order(BigDecimal.ZERO,
        OrderStatus.CREATION, null, null,
        AddressService.createFromDto(orderDto.getShipmentAddress()),
        Collections.emptySet(), cart)));
}

@Transactional
public void delete(Long id) {
    log.debug("Request to delete Order : {}", id);

    Order order = this.orderRepository.findById(id)
        .orElseThrow(() ->
            new IllegalStateException(
                "Order with ID[" + id + "] cannot be found!"));

    Optional.ofNullable(order.getPayment())
        .ifPresent(paymentRepository::delete);

    orderRepository.delete(order);
}

public boolean existsById(Long id) {
    return this.orderRepository.existsById(id);
}

public static OrderDto mapToDto(Order order) {
    Set<OrderItemDto> orderItems = order.getOrderItems()
        .stream().map(OrderItemService::mapToDto).collect(Collectors.toSet());

    return new OrderDto(
        order.getId(),
        order.getPrice(),
        order.getStatus().name(),
        order.getShipped(),
        order.getPayment() != null ? order.getPayment().getId() : null,
        AddressService.mapToDto(order.getShipmentAddress()),
        orderItems,
        CartService.mapToDto(order.getCart())
    );
}
}

```

And the **OrderDto** class will look like:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class OrderDto {
    private Long id;
    private BigDecimal totalPrice;
    private String status;
    private ZonedDateTime shipped;
    private Long paymentId;
    private AddressDto shipmentAddress;
    private Set<OrderItemDto> orderItems;
    private CartDto cart;
}
```

PaymentService

The **PaymentService** class will look like:

```
@Slf4j
@ApplicationScoped @Transactional
public class PaymentService {

    @Inject
    PaymentRepository paymentRepository;
    @Inject
    OrderRepository orderRepository;

    public List<PaymentDto> findByPriceRange(Double max) {
        return this.paymentRepository
            .findAllByAmountBetween(BigDecimal.ZERO, BigDecimal.valueOf(max))
            .stream().map(payment -> mapToDto(payment,
                findOrderByPaymentId(payment.getId()).getId()))
            .collect(Collectors.toList());
    }

    public List<PaymentDto> findAll() {
        return this.paymentRepository.findAll().stream()
            .map(payment -> findById(payment.getId())).collect(Collectors.toList());
    }

    public PaymentDto findById(Long id) {
        log.debug("Request to get Payment : {}", id);
        Order order = findOrderByPaymentId(id).orElseThrow(() ->
            new IllegalStateException("The Order does not exist!"));
    }
}
```

```

        return this.paymentRepository.findById(id)
            .map(payment -> mapToDto(payment, order.getId())).orElse(null);
    }

    public PaymentDto create(PaymentDto paymentDto) {
        log.debug("Request to create Payment : {}", paymentDto);

        Order order = this.orderRepository.findById(paymentDto.getOrderId())
            .orElseThrow(() ->
                new IllegalStateException("The Order does not exist!"));
        order.setStatus(OrderStatus.PAID);

        Payment payment = this.paymentRepository.saveAndFlush(new Payment(
            paymentDto.getPaypalPaymentId(),
            PaymentStatus.valueOf(paymentDto.getStatus()),
            order.getPrice()
        ));

        this.orderRepository.saveAndFlush(order);

        return mapToDto(payment, order.getId());
    }

    private Order findOrderByPaymentId(Long id) {
        return this.orderRepository.findByIdPaymentId(id).orElseThrow(() ->
            new IllegalStateException("No Order exists for the Payment ID " + id));
    }

    public void delete(Long id) {
        log.debug("Request to delete Payment : {}", id);
        this.paymentRepository.deleteById(id);
    }

    public static PaymentDto mapToDto(Payment payment, Long orderId) {
        if (payment != null) {
            return new PaymentDto(
                payment.getId(),
                payment.getPaypalPaymentId(),
                payment.getStatus().name(),
                orderId);
        }
        return null;
    }
}

```

And the **PaymentDto** class will look like:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PaymentDto {
    private Long id;
    private String paypalPaymentId;
    private String status;
    private Long orderId;
}
```

Product Service

The **ProductService** class will look like:

```
@Slf4j
@ApplicationScoped
@Transactional
public class ProductService {

    @Inject
    ProductRepository productRepository;
    @Inject
    CategoryRepository categoryRepository;

    public List<ProductDto> findAll() {
        log.debug("Request to get all Products");
        return this.productRepository.findAll()
            .stream().map(ProductService::mapToDto)
            .collect(Collectors.toList());
    }

    public ProductDto findById(Long id) {
        log.debug("Request to get Product : {}", id);
        return this.productRepository.findById(id)
            .map(ProductService::mapToDto).orElse(null);
    }

    public Long countAll() {
        return this.productRepository.count();
    }

    public Long countByCategoryId(Long id) {
        return this.productRepository.countAllByCategoryId(id);
    }
}
```

```

public ProductDto create(ProductDto productDto) {
    log.debug("Request to create Product : {}", productDto);

    return mapToDto(this.productRepository.save(
        new Product(
            productDto.getName(),
            productDto.getDescription(),
            productDto.getPrice(),
            ProductStatus.valueOf(productDto.getStatus()),
            productDto.getSalesCounter(),
            Collections.emptySet(),
            categoryRepository.findById(productDto.getCategoryId())
                .orElse(null)
        )));
}

public void delete(Long id) {
    log.debug("Request to delete Product : {}", id);
    this.productRepository.deleteById(id);
}

public List<ProductDto> findByCategoryId(Long id) {
    return this.productRepository.findByCategoryId(id).stream()
        .map(ProductService::mapToDto).collect(Collectors.toList());
}

public static ProductDto mapToDto(Product product) {
    return new ProductDto(
        product.getId(),
        product.getName(),
        product.getDescription(),
        product.getPrice(),
        product.getStatus().name(),
        product.getSalesCounter(),
        product.getReviews().stream().map(ReviewService::mapToDto)
            .collect(Collectors.toSet()),
        product.getCategory().getId()
    );
}
}

```

And the **ProductDto** class will look like:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ProductDto {
    private Long id;
    private String name;
    private String description;
    private BigDecimal price;
    private String status;
    private Integer salesCounter;
    private Set<ReviewDto> reviews;
    private Long categoryId;
}
```

ReviewService

The **ReviewService** class will look like:

```
@Slf4j
@ApplicationScoped
@Transactional
public class ReviewService {

    @Inject
    ReviewRepository reviewRepository;

    @Inject
    ProductRepository productRepository;

    public List<ReviewDto> findReviewsByProductId(Long id) {
        log.debug("Request to get all Reviews");
        return this.reviewRepository.findReviewsByProductId(id)
            .stream()
            .map(ReviewService::mapToDto)
            .collect(Collectors.toList());
    }

    public ReviewDto findById(Long id) {
        log.debug("Request to get Review : {}", id);
        return this.reviewRepository
            .findById(id)
            .map(ReviewService::mapToDto)
            .orElse(null);
    }
}
```

```

public ReviewDto create(ReviewDto reviewDto, Long productId) {
    log.debug("Request to create Review : {} ofr the Product {}", reviewDto, productId);

    Product product = this.productRepository.findById(productId)
        .orElseThrow(() ->
            new IllegalStateException(
                "Product with ID:" + productId + " was not found !"));

    Review savedReview = this.reviewRepository.saveAndFlush(
        new Review(
            reviewDto.getTitle(),
            reviewDto.getDescription(),
            reviewDto.getRating()));

    product.getReviews().add(savedReview);
    this.productRepository.saveAndFlush(product);

    return mapToDto(savedReview);
}

public void delete(Long reviewId) {
    log.debug("Request to delete Review : {}", reviewId);

    Review review = this.reviewRepository.findById(reviewId)
        .orElseThrow(() ->
            new IllegalStateException(
                "Product with ID:" + reviewId + " was not found !"));

    Product product = this.productRepository.findProductByReviewId(reviewId);

    product.getReviews().remove(review);

    this.productRepository.saveAndFlush(product);
    this.reviewRepository.delete(review);
}

public static ReviewDto mapToDto(Review review) {
    return new ReviewDto(
        review.getId(),
        review.getTitle(),
        review.getDescription(),
        review.getRating()
    );
}
}

```

And the **ReviewDto** class will look like:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ReviewDto {
    private Long id;
    private String title;
    private String description;
    private Long rating;
}
```

Creating the Web Layer

In this section, we will expose the operations that we implemented in the **Service** classes as **REST webservices**.

In Spring Framework, a **REST webservice** can be implemented using a **RestController**.

REST API Base Path

We want our **Restful Webservices** to be accessed `/api/carts`, `/api/orders`, etc. so we need to define the base-path `/api` as root path, to be reused by all **REST Webservices**.



This is can be configured in **Quarkus** using the property:

```
quarkus.http.root-path=/api
```

Typical RestController: CartResource

The **CartResource** will look like:

```

@ApplicationScoped
@Path("/carts") ①
@Produces(MediaType.APPLICATION_JSON) ②
public class CartResource {

    @Inject CartService cartService;

    @GET
    public List<CartDto> findAll() {
        return this.cartService.findAll();
    }

    @GET @Path("/active")
    public List<CartDto> findAllActiveCarts() {
        return this.cartService.findAllActiveCarts();
    }

    @GET @Path("/customer/{id}")
    public CartDto getActiveCartForCustomer(@PathParam("id") Long customerId) {
        return this.cartService.getActiveCart(customerId);
    }

    @GET @Path("/{id}")
    public CartDto findById(@PathParam("id") Long id) {
        return this.cartService.findById(id);
    }

    @POST @Path("/customer/{id}")
    public CartDto create(@PathParam("id") Long customerId) {
        return this.cartService.createDto(customerId);
    }

    @DELETE @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.cartService.delete(id);
    }
}

```

① Identifies the URI path that a resource class or class method will serve requests for.

② Defines **JSON** as a **MediaType** for the produces content of all the methods inside the annotated class.

In this **Rest** webservice, we are declaring:

- Listing all **Carts**: **HTTP GET** on **/api/carts**
- Listing active **Cart's**: **HTTP GET** on **'/api/carts/active**
- Listing active **Cart** for a customer: **HTTP GET** on **/api/carts/customer/{id}** with **{id}** is a holder of the Customer's ID.
- Listing all details of a **Cart**: **HTTP GET** on **api/carts/{id}** with **{id}** is a holder of the Cart's ID.
- Creating a new **Cart** for a giving **Customer**: **HTTP POST** on **/api/carts/customer/{id}** with **{id}** is a holder of the Customer's ID.
- Deleting a **Cart**: **HTTP DELETE** on **/api/carts/{id}** with **{id}**.

These operations will be described using the **OpenAPI Specification**, to facilitate the consumption of our **REST Webservices** by external callers. The **API description** will be generated automatically 😊

CategoryResource

The **CategoryResource** class will look like:

```
@ApplicationScoped @Path("/categories") @Produces(MediaType.APPLICATION_JSON)
public class CategoryResource {
    @Inject CategoryService categoryService;
    @GET
    public List<CategoryDto> findAll() {
        return this.categoryService.findAll();
    }

    @GET @Path("/{id}")
    public CategoryDto findById(@PathParam("id") Long id) {
        return this.categoryService.findById(id);
    }

    @GET @Path("/{id}/products")
    public List<ProductDto> findProductsByCategoryId(@PathParam("id") Long id) {
        return this.categoryService.findProductsByCategoryId(id);
    }

    @POST @Consumes(MediaType.APPLICATION_JSON)
    public CategoryDto create(CategoryDto categoryDto) {
        return this.categoryService.create(categoryDto);
    }

    @DELETE @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.categoryService.delete(id);
    }
}
```

CustomerResource

The **CustomerResource** class will look like:

```

@ApplicationScoped
@Path("/customers")
@Produces(MediaType.APPLICATION_JSON)
public class CustomerResource {

    @Inject
    CustomerService customerService;

    @GET
    public List<CustomerDto> findAll() {
        return this.customerService.findAll();
    }

    @GET
    @Path("/{id}")
    public CustomerDto findById(@PathParam("id") Long id) {
        return this.customerService.findById(id);
    }

    @GET
    @Path("/active")
    public List<CustomerDto> findAllActive() {
        return this.customerService.findAllActive();
    }

    @GET
    @Path("/inactive")
    public List<CustomerDto> findAllInactive() {
        return this.customerService.findAllInactive();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public CustomerDto create(CustomerDto customerDto) {
        return this.customerService.create(customerDto);
    }

    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.customerService.delete(id);
    }
}

```

OrderItemResource

The `OrderItemResource` class will look like:

```
@ApplicationScoped
@Path("/order-items")
@Produces(MediaType.APPLICATION_JSON)
public class OrderItemResource {

    @Inject
    OrderItemService itemService;

    @GET
    @Path("/order/{id}")
    public List<OrderItemDto> findByOrderId(@PathParam("id") Long id) {
        return this.itemService.findByOrderId(id);
    }

    @GET
    @Path("/{id}")
    public OrderItemDto findById(@PathParam("id") Long id) {
        return this.itemService.findById(id);
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public OrderItemDto create(OrderItemDto orderItemDto) {
        return this.itemService.create(orderItemDto);
    }

    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.itemService.delete(id);
    }
}
```

OrderResource

The **OrderResource** class will look like:

```

@ApplicationScoped
@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
public class OrderResource {

    @Inject
    OrderService orderService;

    @GET
    public List<OrderDto> findAll() {
        return this.orderService.findAll();
    }

    @GET
    @Path("/customer/{id}")
    public List<OrderDto> findAllByUser(@PathParam("id") Long id) {
        return this.orderService.findAllByUser(id);
    }

    @GET
    @Path("/{id}")
    public OrderDto findById(@PathParam("id") Long id) {
        return this.orderService.findById(id);
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public OrderDto create(OrderDto orderDto) {
        return this.orderService.create(orderDto);
    }

    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.orderService.delete(id);
    }

    @GET
    @Path("/exists/{id}")
    public boolean existsById(@PathParam("id") Long id) {
        return this.orderService.existsById(id);
    }
}

```

PaymentResource

The **PaymentResource** class will look like:

```

@ApplicationScoped
@Path("/payments")
@Produces(MediaType.APPLICATION_JSON)
public class PaymentResource {

    @Inject
    PaymentService paymentService;

    @GET
    public List<PaymentDto> findAll() {
        return this.paymentService.findAll();
    }

    @GET
    @Path("/{id}")
    public PaymentDto findById(@PathParam("id") Long id) {
        return this.paymentService.findById(id);
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public PaymentDto create(PaymentDto orderItemDto) {
        return this.paymentService.create(orderItemDto);
    }

    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.paymentService.delete(id);
    }

    @GET
    @Path("/price/{max}")
    public List<PaymentDto> findPaymentsByAmountRangeMax(@PathParam("max") double max) {
        return this.paymentService.findByPriceRange(max);
    }
}

```

ProductResource

The **ProductResource** class will look like:

```

@ApplicationScoped
@Path("/products") @Produces(MediaType.APPLICATION_JSON)
public class ProductResource {

    @Inject ProductService productService;

    @GET
    public List<ProductDto> findAll() {
        return this.productService.findAll();
    }

    @GET @Path("/count")
    public Long countAllProducts() {
        return this.productService.countAll();
    }

    @GET @Path("/{id}")
    public ProductDto findById(@PathParam("id") Long id) {
        return this.productService.findById(id);
    }

    @POST @Consumes(MediaType.APPLICATION_JSON)
    public ProductDto create(ProductDto productDto) {
        return this.productService.create(productDto);
    }

    @DELETE @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.productService.delete(id);
    }

    @GET @Path("/category/{id}")
    public List<ProductDto> findByCategoryId(@PathParam("id") Long id) {
        return this.productService.findByCategoryId(id);
    }

    @GET @Path("/count/category/{id}")
    public Long countByCategoryId(@PathParam("id") Long id) {
        return this.productService.countByCategoryId(id);
    }
}

```

ReviewResource

The **ReviewResource** class will look like:

```
@ApplicationScoped
@Path("/reviews") @Produces(MediaType.APPLICATION_JSON)
public class ReviewResource {

    @Inject ReviewService reviewService;

    @GET @Path("/product/{id}")
    public List<ReviewDto> findAllByProduct(@PathParam("id") Long id) {
        return this.reviewService.findReviewsByProductId(id);
    }

    @GET @Path("/{id}")
    public ReviewDto findById(@PathParam("id") Long id) {
        return this.reviewService.findById(id);
    }

    @POST @Path("/product/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public ReviewDto create(ReviewDto reviewDto, @PathParam("id") Long id) {
        return this.reviewService.create(reviewDto, id);
    }

    @DELETE @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.reviewService.delete(id);
    }
}
```

Automated API documentation

Swagger 2 is an open source project used to describe and document RESTful APIs. **Swagger 2** is language-agnostic and is extensible into new technologies and protocols beyond HTTP. The current version defines a set HTML, JavaScript and CSS assets to dynamically generate documentation from a **Swagger**-compliant API. These files are bundled by the **Swagger UI** project to display the API on browser. Besides rendering documentation, **Swagger UI** allows other API developers or consumers to interact with the API's resources without having any of the implementation logic in place.

The **Swagger 2** specification, which is known as **OpenAPI** specification has several implementations. We will be using the **SmallRye OpenAPI** implementation in our project.

SmallRye OpenAPI is automating the generation of the **API Documentation**. **SmallRye OpenAPI** works by examining an application, once, at runtime to infer **API semantics** based on the **Quarkus** configurations, class structure and various compile time **Java Annotations**.

We already added the **quarkus-smallrye-openapi** dependency to our project. No need to make any additional configuration or development. This extension will be generating the **OpenAPI descriptors** and the **Swagger UI** without any effort !

Hello World Swagger !

To run the **Quarkus** application, you just run: `mvn quarkus:dev`

The application will be running on the **8080** port. To access the **Swagger UI**, just go to <http://localhost:8080/api/swagger-ui/>

The screenshot shows the Swagger UI interface for a Quarkus application. At the top, there's a header with the Swagger logo, the URL '/api/openapi', and a green 'Explore' button. Below the header, the title 'Generated API' is displayed, followed by '1.0' and the 'OAS3' logo. A sub-header '/api/openapi' is shown. The main content area is titled 'default'. It lists various API endpoints categorized by resource type:

- Carts:** GET /api/carts, GET /api/carts/active, GET /api/carts/customer/{id}, POST /api/carts/customer/{id} (highlighted in green), GET /api/carts/{id}, DELETE /api/carts/{id} (highlighted in red).
- Categories:** GET /api/categories, POST /api/categories (highlighted in green), GET /api/categories/{id}, DELETE /api/categories/{id} (highlighted in red).
- Products:** GET /api/categories/{id}/products.

We can check also the **generated OpenAPI descriptor** available on this URL: <http://localhost:8080/api/openapi>:

```
---
openapi: 3.0.1
info:
  title: Generated API
  version: "1.0"
paths:
  /api/carts:
    get:
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ListCartDto'
  /api/carts/active:
    get:
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ListCartDto'
...
...
```

The **Swagger-UI** will be available only in the non-production environment. To enable it permanently, we need to add one parameter in the **application.properties**:

```
# Swagger UI
quarkus.swagger-ui.always-include=true
```

Finally, for more code organization, we will add an **OpenAPI @Tag** annotation to all our **REST APIs** classes to describe each class. This is will be useful to regroup all the methods of belonging to the same **REST API** into one section. Let's take the example of the **CartResource**:

```
@Path("/carts")
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = "cart", description = "All the cart methods")
public class CartResource {
    ...
}
```



Don't forget to add **@Tag** for each class.

Then when we restart the application, and we access the **Swagger-UI** again, you can see that the description appears for each **REST API** and the methods are grouped together under the same Tag name:

The screenshot shows the Swagger-UI interface with two main sections: **cart** and **category**.

- cart** - All the cart methods:
 - GET /api/carts
 - GET /api/carts/active
 - GET /api/carts/customer/{id}
 - POST /api/carts/customer/{id}
 - GET /api/carts/{id}
 - DELETE /api/carts/{id}
- category** - All the category methods:
 - GET /api/categories
 - POST /api/categories
 - GET /api/categories/{id}
 - DELETE /api/categories/{id}
 - GET /api/categories/{id}/products

Customize the Quarkus Banner

The last part in this chapter is to customize the application banner, like we are used to do in **Spring Boot**. In **Quarkus** also the operation is very easy. First of all, we need to create a **banner.txt** file in **src/main/resources**:

src/main/resources/banner.txt

Then, we need to tell the application that we have our custom banner in the `banner.txt` file:

```
# Define the custom banner  
quarkus.banner.path=banner.txt
```

Conclusion

Now, we have all the essential skull of our project: **Source code + API documentation + Database** ☺

In the next chapter, we need to create the necessary tests to protect our code in any future modification or refactoring. We will also dig into building and deploying our application using the best **CI/CD pipelines**.

CHAPTER THREE

Upgrading the Monolithic application

Writing the source code and running it, is not a real win ! The real win is to write a code covered with **tests that will guarantee that the business logic is correctly implemented.**

Tests coverage is a very important metric that shows how much big is the dark side of the code. The bigger the coverage is, the better that we have some insurance that **our code is protected** against any hasty or dirty updates or refactoring.

In our case, the tests will be the protective shield against any problem while **splitting** our **Monolithic application** into **Microservices**.

Implementing QuarkuShop Tests

Introducing Tests libraries in Quarkus

In the Java ecosystem, **JUnit** is the most used testing framework. This is why **Quarkus** brings it automatically as tests dependencies while generating a new project:

```
<dependencies>
...
    <!-- Test dependencies -->
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-junit5</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.rest-assured</groupId>
        <artifactId>rest-assured</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

We also notice that there is **Rest Assured** which is pulled along with **JUnit 5**.

What is Rest Assured ?

Rest Assured is a library used for testing and validating REST Webservices in Java in a very easy way. It has a very rich set of matchers and methods to fetch data and to parse requests/responses. It has a very good integration with build tools (such as **Maven**) and IDEs.

Rest Assured framework will make *API Automation Testing* very simple using **Core Java knowledge**, making it a very desirable thing to do.

I will need another library for my tests: **AssertJ**

What is AssertJ ?

AssertJ an opensource community-driven library provides a rich set of assertions, truly helpful error messages, improves test code readability and is designed to be super easy to use within any IDE or build tool.

The AssertJ Maven Dependency to add to your `pom.xml`:

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <scope>test</scope>
</dependency>
```

To run the tests using **Maven**, just type: `mvn verify`. The application will execute the tests under the `test` profile. To define `test` configuration properties, such as the database that will be used for tests, we need to add these properties to our `application.properties` with a `%test` prefix. This prefix is used to inform the application that these properties are for the `test` profile.

What is an application profile?

The development lifecycle of an application has different stages; the most common ones are *development*, *testing*, and *production*. Quarkus profiles group parts of the application configuration and make it be available only in certain environments.

A profile is a set of configuration settings. Quarkus allows to define profile specific properties using a prefix `%profile` of the property. Then, it automatically loads the properties based on the activated profile.



When no `%profile` is present, then the property is associated to the `dev` profile.



While making the Proof-Of-Concept for the book, I found an issue having multiple `application.properties` as we used to have with Spring Boot. I opened [an issue in the Quarkus GitHub #11072](#). **Georgios Andrianakis** one of the Quarkus Team leaders informed me that **it's strongly recommended having only one `application.properties` file**.

src/main/resources/application.properties

```
...=
# Test Datasource config properties
%test.quarkus.datasource.db-kind=postgresql      ①
%test.quarkus.datasource.username=developer
%test.quarkus.datasource.password=p4SSW0rd
%test.quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/test
# Test Flyway minimal config properties
%test.quarkus.flyway.migrate-at-start=true       ②
```

① We need to define the parameters and the credentials of the dedicated Test database instance.

② We need to activate Flyway for the tests.



No need to copy the Flyway migration scripts in the `src/test` folder. Flyway will seek for them in the `src/main/resources` before checking in `src/test/resources`.

Speaking about databases, we will need a tests dedicated database. We will use **TestContainers** for providing lightweight instances of common databases as Docker containers. We don't need thousands of words to define **TestContainers**. You will discover it and love it thru the practical excercise.

We will start by adding the TestContainers Maven dependencies:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.14.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.14.3</version>
    <scope>test</scope>
</dependency>
```

Next, we will create the Glue class that will bring TestContainers to Quarkus. This is will be a Test Utility class only. So under the test code, inside the package `com.targa.labs.quarkushop.utils`, we will create the `TestContainerResource` class:

`src/test/com.targa.labs.quarkushop.utils.TestContainerResource`

```
public class TestContainerResource implements QuarkusTestResourceLifecycleManager { ①

    private static final PostgreSQLContainer<?> DATABASE =
        new PostgreSQLContainer<?>("postgres:13"); ②

    @Override
    public Map<String, String> start() {

        DATABASE.start(); ③

        Map<String, String> confMap = new HashMap<>(); ④

        confMap.put("quarkus.datasource.jdbc.url", DATABASE.getJdbcUrl()); ④
        confMap.put("quarkus.datasource.username", DATABASE.getUsername()); ④
        confMap.put("quarkus.datasource.password", DATABASE.getPassword()); ④

        return confMap; ④
    }

    @Override
    public void stop() {
        DATABASE.close(); ⑤
    }
}
```

- ① Our `TestContainerResource` will implement `QuarkusTestResourceLifecycleManager`, which manages the lifecycle of a test resource. These resources are started before the first test is run, and are closed at the end of the test suite. These resource is brought to the test class using the `@QuarkusTestResource(TestContainerResource.class)` annotation.
- ② This is the core element of our custom test resource. The parameter `postgres:13` is the name of the **PostgreSQL Docker Image** that we will use.
- ③ When the `start()` method is called, we will begin by starting the DATABASE container.
- ④ Next, we will collect the `Datasource` credentials generated dynamically by **TestContainers** inside a `confMap`. When the `confMap` is returned, these credentials are applied instead of those available in the `application.properties`.
- ⑤ When the `close()` method is called, we will close the DATABASE container.

While **Quarkus** will listen on port **8080** by default, when running tests it defaults to **8081**. This allows us to run tests while having the application running in parallel. This **HTTP Test Port** can be changed, to **9999** for example, in the **application.properties** using the property **quarkus.http.test-port=9999**.

What if I insert **quarkus.http.test-port=8888** and **%test.quarkus.http.test-port=9999** to the **application.properties** ?



Easy ! Here we are dealing with the HTTP Port for the Test profile. So the property with the **%test** will be applied to override any value defined before. When running the tests we will see the tests runtime exposing the **9999** port.

A final configuration we need to have is for **Measuring the tests coverage**, one of a very important metrics for the code quality. We will be using **JaCoCo** for generating code coverage reports.

First of all, we need to add the **JaCoCo agent** dependency to the **pom.xml**:

```
<dependency>
    <groupId>org.jacoco</groupId>
    <artifactId>org.jacoco.agent</artifactId>
    <classifier>runtime</classifier>
    <scope>test</scope>
    <version>0.8.5</version>
</dependency>
```

Next, we need to change the **Surefire** plugin Maven configuration:

```
<build>
...
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${surefire-plugin.version}</version>
    <configuration>
        <excludedGroups>integration</excludedGroups>
        <systemPropertyVariables>
            <jacoco-agent.destfile>
                ${project.build.directory}/jacoco-ut.exec
            </jacoco-agent.destfile>
            <java.util.logging.manager>
                org.jboss.logmanager.LogManager
            </java.util.logging.manager>
            <maven.home>${maven.home}</maven.home>
        </systemPropertyVariables>
    </configuration>
    <executions>
        <execution>
            <id>integration-tests</id>
            <phase>integration-test</phase>
            <goals>
                <goal>test</goal>
            </goals>
            <configuration>
                <excludedGroups>!integration</excludedGroups>
                <groups>integration</groups>
                <systemPropertyVariables>
                    <jacoco-agent.destfile>
                        ${project.build.directory}/jacoco-it.exec
                    </jacoco-agent.destfile>
                </systemPropertyVariables>
            </configuration>
        </execution>
    </executions>
</plugin>
...
</build>
```

Next, we need to the **JaCoCo** Maven plugin and to configure it:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>${jacoco.version}</version>
  <executions>
    <execution>
      <id>instrument-ut</id>
      <goals><goal>instrument</goal></goals>
    </execution>
    <execution>
      <id>restore-ut</id>
      <goals><goal>restore-instrumented-classes</goal></goals>
    </execution>
    <execution>
      <id>report-ut</id>
      <goals><goal>report</goal></goals>
      <configuration>
        <dataFile>${project.build.directory}/jacoco-ut.exec</dataFile>
        <outputDirectory>
          ${project.reporting.outputDirectory}/jacoco-ut
        </outputDirectory>
      </configuration>
    </execution>
    <execution>
      <id>instrument-it</id>
      <phase>pre-integration-test</phase>
      <goals><goal>instrument</goal></goals>
    </execution>
    <execution>
      <id>restore-it</id>
      <phase>post-integration-test</phase>
      <goals><goal>restore-instrumented-classes</goal></goals>
    </execution>
    <execution>
      <id>report-it</id>
      <phase>post-integration-test</phase>
      <goals><goal>report</goal></goals>
      <configuration>
        <dataFile>${project.build.directory}/jacoco-it.exec</dataFile>
        <outputDirectory>
          ${project.reporting.outputDirectory}/jacoco-it
        </outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
<execution>
    <id>merge-results</id>
    <phase>verify</phase>
    <goals><goal>merge</goal></goals>
    <configuration>
        <fileSets>
            <fileSet>
                <directory>${project.build.directory}</directory>
                <includes><include>*.exec</include></includes>
            </fileSet>
        </fileSets>
        <destFile>${project.build.directory}/jacoco.exec</destFile>
    </configuration>
</execution>
<execution>
    <id>post-merge-report</id>
    <phase>verify</phase>
    <goals><goal>report</goal></goals>
    <configuration>
        <dataFile>${project.build.directory}/jacoco.exec</dataFile>
        <outputDirectory>
            ${project.reporting.outputDirectory}/jacoco
        </outputDirectory>
    </configuration>
</execution>
</executions>
</plugin>
```

We will use this **JaCoCo** reporting later to check how much coverage we have in our tests.

Writing the first tests

We will start by testing the first REST API: the **Cart** REST API 

Based on the **CartResource** or the **Swagger-UI**, we can notice that there are six services:

GET	/api/carts
GET	/api/carts/active
GET	/api/carts/customer/{id}
POST	/api/carts/customer/{id}
GET	/api/carts/{id}
DELETE	/api/carts/{id}

So we will have at least six tests, a test per service.

Before starting the tests, let's see how looks a typical Test class in the Quarkus world:

src/test/java/com/targa/labs/quarkushop/web

```
@QuarkusTest          ①
@QuarkusTestResource(TestContainerResource.class) ②
class CartResourceTest {

    @Test           ③
    void testSomeOperationOrFeature() {

    }
}
```

① Annotation that controls the **JUnit 5** testing framework in **Quarkus**.

② Annotation that will make the **TestContainerResource** available for our **CartResourceTest**

③ Used to signal that the annotated method is a Test.

We will make tests to verify that **GIVEN** having the required inputs and **WHEN** doing the correct calls, **THEN** we will get the expected results.

We will start by creating a typical Test: `findAllCarts` used to list all **Carts** using an **HTTP GET** request on `/api/carts`. This **REST API** will return an array of `CartDto`. We can translate this use case to a Test using **JUnit 5** and **REST Assured** easily:

```
@Test
void testfindAll() {
    get("/carts")      ①
        .then()          ②
            .statusCode(OK.getStatusCode()) ③
            .body("size()", greaterThan(0)); ④
}
```

- ① Starts a **REST Assured** Test Case for sending an **HTTP GET** request to the `/carts` **REST API**.
- ② Extracts a **REST Assured ValidatableResponse** of the request made in the previous line.
- ③ Validates that the **Response Status Code** matches `200 OK` which is returned by `OK.getStatusCode()` from `Response.Status` Enumeration coming from **Rest Assured**.
- ④ Validates that the **JSON** or **XML** response body element `size()` conforms to the **Hamcrest** matcher `greaterThan(0)`.

To summarize, this test will verify that doing an **HTTP GET** on `/cart` will return:

- a **header** containing `200` as **Status code**
- a **body** with a non empty array of elements



In the previous call, we just called the path `/carts` and not `/api/carts` because the `/api` root base is added by the property `quarkus.http.root-path=/api` that we added previously in the `application.properties`.

The same style for testing the methods `findAllActiveCarts()` and `getActiveCartForCustomer()`, `findById()` and `deleteById()`:

```
@Test ①
void testfindAllActiveCarts() {
    get("/carts/active").then()
        .statusCode(OK.getStatusCode());
}

@Test ②
void testGetActiveCartForCustomer() {
    get("/carts/customer/3").then()
        .statusCode(OK.getStatusCode())
        .body(containsString("Peter"));
}

@Test ③
void testfindById() {
    get("/carts/3").then()
        .statusCode(OK.getStatusCode())
        .body(containsString("status"))
        .body(containsString("NEW"));

    get("/carts/100").then()
        .statusCode(NO_CONTENT.getStatusCode())
        .body(emptyOrNullString());
}

@Test ④
void testDelete() {
    get("/carts/active").then()
        .statusCode(OK.getStatusCode())
        .body(containsString("Jason"))
        .body(containsString("NEW"));

    delete("/carts/1").then()
        .statusCode(NO_CONTENT.getStatusCode());

    get("/carts/1").then()
        .statusCode(OK.getStatusCode())
        .body(containsString("Jason"))
        .body(containsString("CANCELED"));
}
```

In these tests we verify that:

- ① the response of the **HTTP GET** request on `/carts/active` has **200** as **status code**.
- ② the response of the **HTTP GET** request on `/carts/customer/3` has **200** as **status code** and the body contains "**Peter**". **Peter Quinn** is the Customer with ID 3 and has some active cart: this value is from the sample data that we imported using Flyway in the `V1.1_Insert_samples.sql` script.
- ③ the response of the **HTTP GET** request on `/carts/3` has **200** as **status code** and the body contains "**NEW**" as **Cart status**. While, the response of the **HTTP GET** request on `/carts/100` has **404** as **status code** and its body is empty as **we don't have a cart with ID 100**.
- ④ for a given **active cart with ID 1** and owned by the customer **Jason**, after we execute an **HTTP DELETE** on `/carts/1`, then the cart status will change from "**NEW**" to "**CANCELED**".

No we will go in deeper test cases.

We will verify an incorrect situation: in our business logic, a customer cannot have more than one active cart at any moment. So we will create a test to verify that the application will throw an error when there are two active carts for a given customer. We need to insert a record of an extra active cart for the customer with ID 3. Keep calm ! 😊 We will delete this record in the end of the test to keep our database clean. To do this insertion and deletion SQL queries, we need to access the database from the tests context. To make this interaction possible, we need to get a **Datasource** in the test. **Quarkus** supports this by allowing you to inject **CDI beans** into your tests via the `@Inject` annotation. As the **Datasource** is a **CDI bean** so we can inject it in our test.



Tests in **Quarkus** are full **CDI beans**, so you can enjoy all **CDI features** 😊

The test will look like:

```

@QuarkusTest
@QuarkusTestResource(TestContainerResource.class)
class CartResourceTest {

    private static final String INSERT_WRONG_CART_IN_DB =
        "insert into carts values (999, current_timestamp, current_timestamp, 'NEW', 3)";

    private static final String DELETE_WRONG_CART_IN_DB =
        "delete from carts where id = 999";

    @Inject
    Datasource datasource;

    ...

    @Test
    void testGetActiveCartForCustomerWhenThereAreTwoCartsInDB() {
        executeSql(INSERT_WRONG_CART_IN_DB);

        get("/carts/customer/3").then()
            .statusCode(INTERNAL_SERVER_ERROR.getStatusCode())
            .body(containsString(INTERNAL_SERVER_ERROR.getReasonPhrase()))
            .body(containsString("Many active carts detected !!!"));

        executeSql(DELETE_WRONG_CART_IN_DB);
    }

    private void executeSql(String query) {
        try (var connection = dataSource.getConnection()) {
            var statement = connection.createStatement();
            statement.executeUpdate(query);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    ...
}

```

The test will verify that an HTTP GET request on `/carts/customer/3` will have:

- **Status code** is **500** which stands for an **Internal Server Error**
- **Body** contains "**Internal Server Error**"

The next test will be about creating a new cart.

To create a cart, we need to create a customer, and based on its ID we can create the cart. Our test will have calls to the **Customer API** for customer creation, and next to the **Cart API** to create a cart. For database consistency, by the end of the test, we will delete the created records:

```

@Test
void testCreateCart() {
    var requestParams = new HashMap<>(); ①
    requestParams.put("firstName", "Saul"); ①
    requestParams.put("lastName", "Berenson"); ①
    requestParams.put("email", "call.saul@mail.com"); ①

    var newCustomerId = given()
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
        .body(requestParams).post("/customers").then()
        .statusCode(OK.getStatusCode())
        .extract() ②
        .jsonPath() ②
        .getInt("id"); ②

    var response = post("/carts/customer/" + newCustomerId).then()
        .statusCode(OK.getStatusCode())
        .extract() ③
        .jsonPath() ③
        .getMap("$");

    assertThat(response.get("id")).isNotNull();
    assertThat(response).containsEntry("status", CartStatus.NEW.name());

    delete("/carts/" + response.get("id")).then()
        .statusCode(NO_CONTENT.getStatusCode());

    delete("/customers/" + newCustomerId).then()
        .statusCode(NO_CONTENT.getStatusCode());
}

```

① We are packaging the request parameters into a **Map** that will be serialized to **JSON** by **REST Assured**.

② The `extract().jsonPath().getInt("id")` is used to extract the value for the attribute "**id**" in the response **JSON** body.

③ The `extract().jsonPath().getMap("$")` is used to extract **all** the **JSON** body and deserialize it into a **Map**.

The last test for the **Cart API** is to check that the *API* will refuse to create another cart for the same customer, when he has already an active cart:

```

@Test
void testFailCreateCartWhileHavingAlreadyActiveCart() {

    var requestParams = new HashMap<>();
    requestParams.put("firstName", "Saul");
    requestParams.put("lastName", "Berenson");
    requestParams.put("email", "call.saul@mail.com");

    var newCustomerId = given()
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
        .body(requestParams)
        .post("/customers").then()
        .statusCode(OK.getStatusCode())
        .extract()
        .jsonPath()
        .getLong("id");

    var newCartId = post("/carts/customer/" + newCustomerId).then()
        .statusCode(OK.getStatusCode())
        .extract()
        .jsonPath()
        .getLong("id");

    post("/carts/customer/" + newCustomerId).then()
        .statusCode(INTERNAL_SERVER_ERROR.getStatusCode())
        .body(containsString(INTERNAL_SERVER_ERROR.getReasonPhrase()))
        .body(containsString("There is already an active cart"));

    assertThat(newCartId).isNotNull();

    delete("/carts/" + newCartId).then()
        .statusCode(NO_CONTENT.getStatusCode());

    delete("/customers/" + newCustomerId).then()
        .statusCode(NO_CONTENT.getStatusCode());
}

```

In this test, we verify that, other than the **Status Code 500** and the famous **Internal Server Error**, the response body contains the message "**There is already an active cart**".



I will cover here only the **CartResourceTest** as it's the most panoramic one in the eight test classes. You will find all the code in my [GitHub Repository](#)

We will use **SonarQube** to check the coverage of our tests and to analyze the quality of our code.

Discovering SonarQube

SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on 20+ programming languages. **SonarQube** offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.

SonarQube can record metrics history and provides evolution graphs. **SonarQube** provides fully automated analysis and integration with *Maven*, *Ant*, *Gradle*, *MSBuild* and continuous integration tools (*Atlassian Bamboo*, *Jenkins*, *Hudson*, etc.).

We either need to install **SonarQube** locally on our machine, or to use a hosted version. For example, we can use **SonarCloud** where we have the possibility to analyze our project for free.

What is SonarCloud ?

SonarCloud is the leading online service to catch *Bugs* and *Security Vulnerabilities* in your *Pull Requests* and throughout your code repositories.

SonarCloud is a cloud-based code quality and security service of **SonarQube**. The main features of **SonarCloud** are:

- 23 languages: *Java*, *JS*, *C#*, *C/C++*, *Objective-C*, *TypeScript*, *Python*, *ABAP*, *PLSQL*, *T-SQL and more*.
- Thousands of rules to track down hard-to-find bugs and quality issues thanks to powerful static code analyzers.
- **Cloud CI Integrations**, with Travis, Azure DevOps, BitBucket, AppVeyor and more.
- Deep code analysis, to explore all source files, whether in branches or pull requests, to reach a green **Quality Gate** and promote the build.
- Fast and Scalable

We can create a free account in SonarCloud.

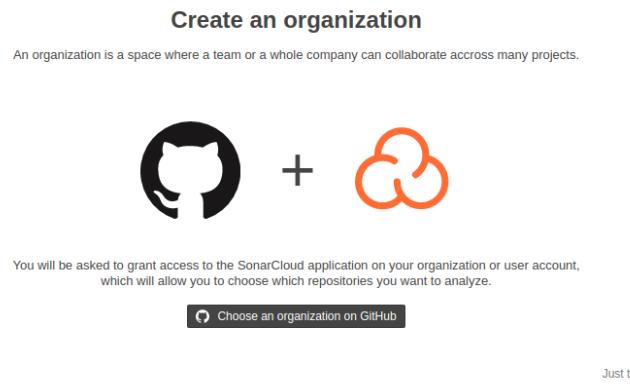
The image shows the SonarCloud landing page. At the top left is the SonarCloud logo. At the top right are links for 'Pricing', 'Explore', and 'Log In'. Below the logo, there's a large orange banner featuring three cartoon characters: a woman pointing upwards, a man in a red shirt with 'KEEP CALM AND WRITE CLEAN CODE' on it, and a man in a purple shirt with 'BUGS NO MORE' on it. To the left of the banner, the text 'Clean Code Rockstar Status' is displayed, along with the subtext 'Eliminate bugs and vulnerabilities. Champion quality code in your projects.' Below this are buttons for GitHub, Bitbucket, Azure DevOps, and GitLab, with a note 'Free for Open-Source Projects'. To the right of the banner, the text 'Enhance Your Workflow with Continuous Code Quality' is shown above a diagram of a software development pipeline. The pipeline starts with a 'Quality Gate Failed' box (containing 2 Bugs, 0 Vulnerabilities, and 1 Code Smell) which leads to a 'Help me fix it' button. This then branches into two paths: one leading to a 'Quality Gate Passed' box (containing 0 Bugs, 0 Vulnerabilities, and 1 Code Smell), and another path that loops back to the failed gate. Below the pipeline, the text 'Maximize your throughput and only release clean code' and 'SonarCloud automatically analyzes branches and decorates pull requests' is displayed.

Next, we can choose to start with **SonarCloud** using a **GitHub** or **Azure DevOps** or even **Bitbucket** or **Gitlab** account. I will use **GitHub** in our case.

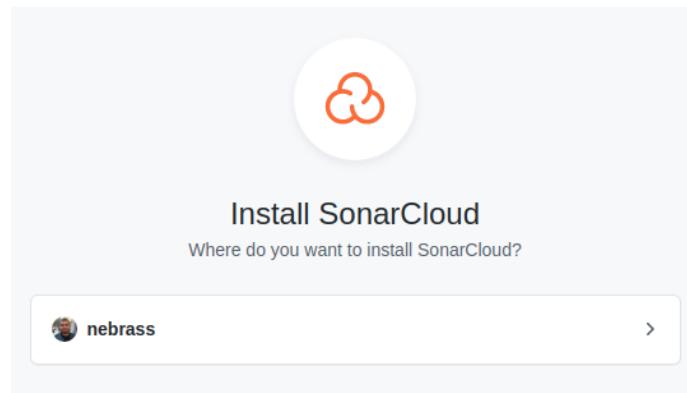
Next, click on ****"create projects manually"**:

The image shows the 'Welcome to SonarCloud' page. At the top center is the SonarCloud logo. Below it, the text 'Welcome to SonarCloud' and 'Let us help you get started in your journey to code quality' is displayed. There are two main sections: 'Analyze your first projects' (with a 'Import projects from GitHub' button) and 'Join an organization' (with a note about asking an administrator to add you manually). A small note at the bottom right says 'Just testing? You can [create projects manually](#)'.

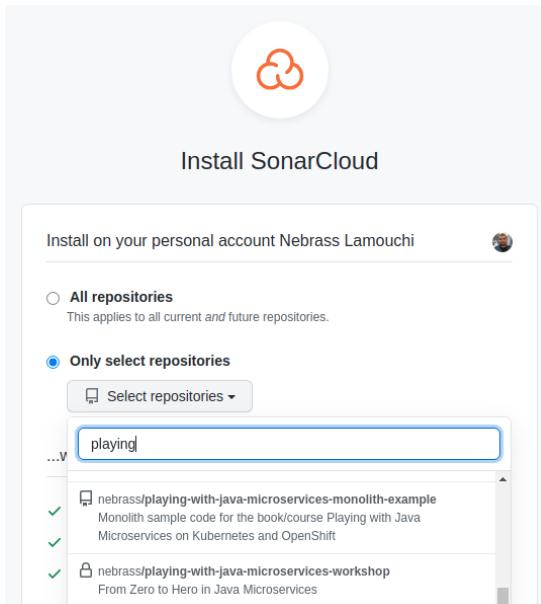
Next, click on ***Choose an organization on GitHub*** to import your organization on GitHub:



Next, choose the organization where you are storing your project source code:



Next, choose the project from the list:



Next, you need to define an organization name:

Create an organization

An organization is a space where a team or a whole company can collaborate across many projects.

- 1 Import organization details**

Import  **Nebrass Lamouchi** into a SonarCloud organization X

Key*
nebrass

Up to 255 characters. All chars must be lower-case letters (a to z), digits or dash (but dash can neither be trailing nor heading).

[Add additional info ▾](#)

 All members from your GitHub organization Nebrass Lamouchi will be added to your SonarCloud organization. As they connect to SonarCloud with their GitHub account, members will automatically have access to your SonarCloud organization and its projects. [See all members on GitHub](#)

[Continue](#)

- 2 Choose a plan**

And choose the plan, you can choose the **Free Plan**, which is suitable for all public repositories and projects:

Create an organization

An organization is a space where a team or a whole company can collaborate across many projects.

- 1 Import organization details**
- 2 Choose a plan**

<p><input type="radio"/> Paid plan from €10</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> Unlimited private projects <input checked="" type="checkbox"/> Strict control over who can view your private data <input checked="" type="checkbox"/> No commitments, cancel anytime <input checked="" type="checkbox"/> 14 days free trial. <p>Learn more</p> <p> Recommended because your organization contains private repositories</p>	<p><input checked="" type="radio"/> Free plan €0</p> <p>All projects you analyze will be public. Anyone can browse source code.</p> <div style="background-color: #fffacd; padding: 5px; border: 1px solid #ffcc99; margin-top: 10px;"> <p> SonarCloud won't allow your private repositories to be imported from GitHub if you choose this plan.</p> </div>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

[Create Organization](#)

Now, you can select the public repositories that you want to analyze, and click on **Set Up**:

Analyze projects - Select repositories

Organization*

Import another organization

Select all available repositories

-  playing-with-cQRS-and-event-sourcing-in-spring-boot-and-axon
-  playing-with-java-microservices-book-code
-  playing-with-java-microservices-monolith-example
-  playing-with-reactive-spring-boot
-  playing-with-spring-boot-and-kafka-on-azure-event-hub
-  playing-with-spring-boot-on-k8s

Don't see your repo? Check your [GitHub app configuration](#).

Analyze private projects with our Paid Plan from €10

- ✓ Unlimited private projects
- ✓ Strict control over who can view your private data
- ✓ No commitments, cancel anytime
- ✓ 14 days free trial.

[Learn more](#)

Just testing? You can [create manually](#).

Setup a [monorepo](#).

We will land next to the project configuration screen:

sonarcloud  My Projects My Issues + 

 Nebrass Lamouchi /  playing-with-java-microservices-monolith-example  master 

[Configure](#) [Issues](#) [Measures](#) [Code](#) [Activity](#) [Administration ▾](#) [PUBLIC](#)  

We do not recommend Automatic Analysis for this project.

90.34% of your code is in languages unsupported by Automatic Analysis 

- ✓ 9.66% Lines of Code can be analyzed (TSQL files)
- ✗ 90.34% Lines of Code will be ignored (Java files)

We recommend using another analysis method to get the most out of SonarCloud.

Choose another analysis method

-  With Travis CI
-  With CircleCI
-  With other CI tools 
-  Manually 

Then, we will chose the **Manual analysis method** and next, we pick Maven as build tool:

Analyze from your local sources

Run analysis on your project

What option best describes your build?

- Maven
- Gradle
- C, C++ or ObjC
- Other (for JS, TS, Go, Python, PHP, ...)

Configure the SONAR_TOKEN environment variable

- ① Name of the environment variable: `SONAR_TOKEN`
- ② Value of the environment variable: `XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX`

Execute the SonarScanner for Maven from your computer

Update your `pom.xml` file with the following properties:

```
<properties>
<sonar.projectKey>nebrass_playing-with-java-microservices-monolith-example</sonar.projectKey>
<sonar.organization>nebrass</sonar.organization>
<sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

Run the following command in the project folder:

```
mvn verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
```

If you wish, you can shorten this command (to `mvn verify sonar:sonar`, for example) by specifying a prefix for the plugin.

See the [SonarScanner for Maven documentation](#) for more details.

Is it done? You should see the page refresh itself in a few moments with your analysis results if everything runs successfully.

Check these useful links while you wait: [What is a Quality Gate?](#), [Configure your Quality Profiles](#)

Now, we will get the custom configuration that we will use to analyze our project on SonarCloud. We will have some properties to add to our `pom.xml`:

```
<properties>
<sonar.projectKey>nebrass_quarkushop</sonar.projectKey>
<sonar.organization>nebrass</sonar.organization>
<sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

And we will need to define an environment variable called **SONAR_TOKEN** with the value generated here. This token will be used to authenticate the **SonarQube Maven Plugin** to the **SonarCloud**.

Now, our project is configured to be analyzed on **SonarCloud**, just by running: `mvn verify sonar:sonar`:

sonarcloud My Projects My Issues

Nebrass Lamouchi Java Dreamer https://blog.nebrass.fr Key: nebrass

Projects Issues Quality Profiles Rules Quality Gates Members Administration

Filters

Quality Gate

Passed	Warning	Failed
1	0	0

Reliability (Bug) Bugs

Perspective: Overall Status Sort by: Name

quarkushop NEW Passed

Last analysis: August 2, 2020, 10:29 PM

1 projects

Bugs: 0 A | Vulnerabilities: 0 A | Code Smells: 0 A | Coverage: 2.2% | Duplications: 0.0% | 1.8k S Java, XML

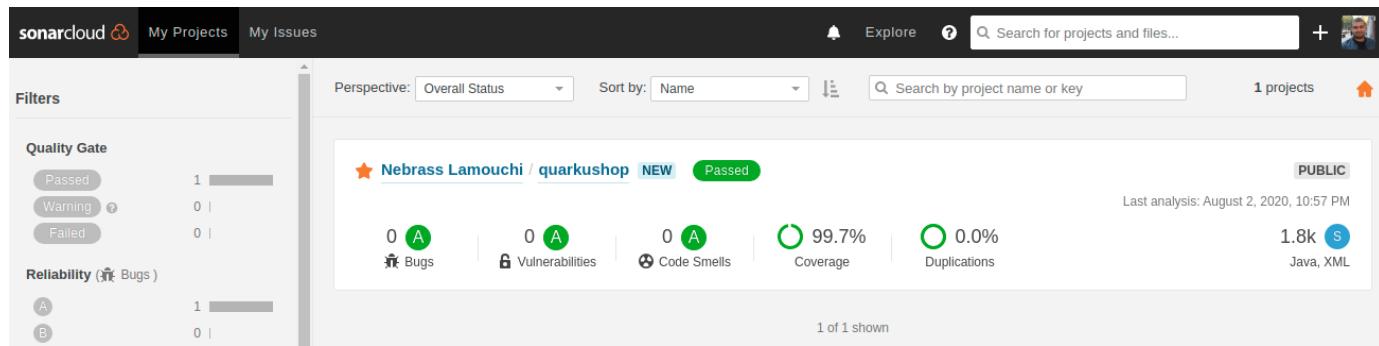
Wow! It's only 2.2% ?? 😬 We made strong enough tests to test everything, but it seems that there is something missing?! The reason is that when using Lombok in the application, we need to add some additional configuration file, in the project root folder:

`lombok.config`

```
config.stopBubbling = true          ①
lombok.addLombokGeneratedAnnotation = true ②
```

- ① tells Lombok that this is your root directory. You can then create `lombok.config` files in any subdirectories (generally representing projects or source packages) with different settings.
- ② Lombok can be configured to add `@lombok.Generated` annotations to all generated nodes where possible which is so useful for `JaCoCo` (which has built in support), or other style checkers and code coverage tools.

Let's run the Sonar Analyzer again: `mvn clean verify sonar:sonar`:



Yoppi! 😊 Now Sonar is aware of the Lombok generated code and the Analysis results are more coherent with what we wrote as tests. 😊

Building and Running QuarkuShop

Building the QuarkuShop

Packaging modes in Quarkus

The **QuarkuShop** is using **Maven** as Build Tool. So, we can use the `mvn package` to build it. This command will build the `quarkushop-1.0.0-SNAPSHOT-runner.jar` file in the `target/` directory. Unfortunately, this **JAR** file cannot be executed anywhere without the `target/lib/` folder where all the required libraries are copied. So if we want to distribute the `quarkushop-1.0.0-SNAPSHOT-runner.jar`, we need to distribute the `lib` folder with it.

To have only a standalone **JAR** file that packages the **QuarkuShop** with all the needed, we can create a **Fat JAR** (aka **Uber JAR**).

What Fat or Uber JARs ?

Fat/Uber JAR Maven and in especially Spring Boot popularized this well-known approach to packaging, which includes everything needed to run the whole app on a standard Java Runtime environment (i.e. so you can run the app with `java -jar myapp.jar`). The amount of extra runtime stuff included in the Uber JAR (and its file size) depends on the framework and runtime features your app uses.

To build the UberJAR for QuarkuShop, just type this command:

```
mvn clean package -Dquarkus.package.type=uber-jar
```

So easy and so simple ! Nothing to configure or to add in our project. Quarkus natively supports the creation of **Uber JARs**.

Quarkus also supports the Native mode, which is the best and the most promoted feature of this great framework.

Meeting the Quarkus Native

Quarkus is coming with the powerful features of **GraalVM** which converts our Java applications into fully compiled binaries using the **native-image** tool. These binaries are also called **Native Images**. **GraalVM** can compile Java bytecode into **Native Images** to get **faster startup** and **smaller footprint** for your applications.

The **native-image** feature is not available by default when install. To install **Native Image** in **GraalVM**, just run the following command :

```
gu install native-image
```



Make sure to have the `GRAALVM_HOME` environment variable configured and pointing to your GraalVM install directory.

To build the native **QuarkuShop** binary:

```
./mvnw clean package -Pnative
```

To build the native executable, we are using the **native** Maven Profile available in the **pom.xml**:

```
<profiles>
  <profile>
    <id>native</id>
    <activation>
      <property><name>native</name></property>
    </activation>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-failsafe-plugin</artifactId>
          <version>${surefire-plugin.version}</version>
          <executions>
            <execution>
              <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
              </goals>
              <configuration>
                <systemProperties>
                  <native.image.path>
                    ${project.build.directory}/${project.build.finalName}-runner
                  </native.image.path>
                  <java.util.logging.manager>
                    org.jboss.logmanager.LogManager
                  </java.util.logging.manager>
                  <maven.home>${maven.home}</maven.home>
                </systemProperties>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
    <properties>
      <quarkus.package.type>native</quarkus.package.type>
    </properties>
  </profile>
</profiles>
```

Running the tests in the Native Mode

You can notice that there are the two goals configured: `integration-test` and `verify`. These are used to execute the tests that are aimed to be running for the Native version of the application. We can reuse the tests that we have in the classic JAR and bring them to the Native Image. This is can be done by creating new Java classes that inherits from each `@QuarkusTest` class. The inheriting class needs to be annotated by `@NativeImageTest` which indicates that this test should be run using a native image, rather than in the JVM.

For example, for the `CartResourceTest`:

```
@QuarkusTest
@QuarkusTestResource(TestContainerResource.class)
class CartResourceTest {
    ...
}
```

We will create the `CartResourceIT` that will run the test in the native image:

```
@NativeImageTest
class CartResourceIT extends CartResourceTest {
```



I'm using the same naming conventions as the official Quarkus documentation. We are using the `Test` suffix for JVM Integration Tests, and we are using the `IT` suffix for Native Image Tests.

After creating the Native Image tests, let's try to run the tests using the native image:

```
mvn verify -Pnative
```

All the tests will pass EXCEPT the Native Test class that inherits from `CartResourceTest` 😱. The error message is very clear:

```
[ERROR] Errors:
[ERROR]   CartResourceIT » JUnit @Inject is not supported in NativeImageTest tests.
Offe...
[INFO]
[ERROR] Tests run: 39, Failures: 0, Errors: 1, Skipped: 0
[INFO]
```

This is due to lack of support of the injecting into Native mode. Although, in the `CartResourceTest`, we are injecting the `DataSource` into the test for database interaction. This is possible in the JVM mode, but not in the Native mode. To disable a specific parent test class in the Native mode, just annotate that class using the `@DisabledOnNativeImage`.

In our case, the `CartResourceTest` will look like:

```
@DisabledOnNativeImage
@QuarkusTest
@QuarkusTestResource(TestContainerResource.class)
class CartResourceTest {
    ...
}
```

Now if we run again the `mvn verify -Pnative` command, we will skip the disabled tests and all the remaining tests will be passing:

```
[INFO] Results:
[INFO]
[WARNINg] Tests run: 39, Failures: 0, Errors: 0, Skipped: 1
```

Packaging and running the Native QuarkuShop

After compiling and running the tests in the native mode. It's time to package it natively. This can be done using the command:

```
mvn package -Pnative
```

This command will build the Native executable ☺ Unfortunately ! This binary cannot be distributed and cannot be executed in other machines: The native executable is specific to your operating system, where it was compiled. ☹

Keep calm ! There is a solution for this brought to you by the wonderful Containerization and the fabulous Quarkus Team ☺ The solution is to build the native binary inside a Docker Container, so it will be isolated from the host OS. We can do this using the command:

```
$ mvn package -Pnative -Dquarkus.native.container-build=true

...
[INFO] --- quarkus-maven-plugin:1.7.0.Final:build (default) @ quarkushop ---
[INFO] [org.jboss.threads] JBoss Threads version 3.1.1.Final
[INFO] [io.quarkus.flyway.FlywayProcessor] Adding application migrations in path
'file:/home/nebrass/java/playing-with-java-microservices-monolith-
example/target/quarkushop-1.0.0-SNAPSHOT.jar!/db/migration' using protocol 'jar'
[INFO] [org.hibernate.Version] HHH000412: Hibernate ORM core version 5.4.19.Final
[INFO] [io.quarkus.deployment.pkg.steps.JarResultBuildStep] Building native image source
jar: ...quarkushop-1.0.0-SNAPSHOT-runner.jar
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Building native image from
...quarkushop-1.0.0-SNAPSHOT-runner.jar
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Checking image status
```

```

quay.io/quarkus/ubi-quarkus-native-image:20.1.0-javal1
20.1.0-javal1: Pulling from quarkus/ubi-quarkus-native-image
57de4da701b5: Pull complete
cf0f3ebe9f53: Pull complete
6d14943d1530: Pull complete
Digest: sha256:abff65d7807d2bc68fd9835748cbfc5066a7a4b76f984c60cb2170ed1d4412c8
Status: Downloaded newer image for quay.io/quarkus/ubi-quarkus-native-image:20.1.0-javal1
quay.io/quarkus/ubi-quarkus-native-image:20.1.0-javal1
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Running Quarkus native-
image plugin on GraalVM Version 20.1.0 (Java Version 11.0.7)
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] docker run \
    -v /home/nebrass/java/playing-with-java-microservices-monolith-
example/target/quarkushop-1.0.0-SNAPSHOT-native-image-source-jar:/project:z \
    --env LANG=C \
    --user 1000:1000 \
    --rm \
    quay.io/quarkus/ubi-quarkus-native-image:20.1.0-javal1 \
    -J-Dsun.nio.ch.maxUpdateArraySize=100 \
    -J-DCoordinatorEnvironmentBean.transactionStatusManagerEnable=false \
    -J-Djava.util.logging.manager=org.jboss.logmanager.LogManager \
    -J-Dvertx.logger-delegate-factory-class
    -name=io.quarkus.vertx.core.runtime.VertxLogDelegateFactory \
    -J-Dvertx.disableDnsResolver=true \
    -J-Dio.netty.leakDetection.level=DISABLED \
    -J-Dio.netty.allocator.maxOrder=1 \
    -J-Duser.language=en \
    -J-Dfile.encoding=UTF-8 \
    --initialize-at-build-time= \
-H:InitialCollectionPolicy=com.oracle.svm.core.genscavenge.CollectionPolicy$BySpaceAndTim
e \
    -H:+JNI -jar quarkushop-1.0.0-SNAPSHOT-runner.jar \
    -H:FallbackThreshold=0 \
    -H:+ReportExceptionStackTraces \
    -H:-AddAllCharsets \
    -H:EnableURLProtocols=http,https \
    --enable-all-security-services \
    --no-server \
    -H:-UseServiceLoaderFeature \
    -H:+StackTrace quarkushop-1.0.0-SNAPSHOT-runner
[quarkushop-1.0.0-SNAPSHOT-runner:24]      classlist:  7,485.73 ms,  1.17 GB
[quarkushop-1.0.0-SNAPSHOT-runner:24]          (cap):   559.13 ms,  1.18 GB
[quarkushop-1.0.0-SNAPSHOT-runner:24]          setup:   2,157.70 ms,  1.18 GB
13:09:33,858 INFO  [org.hib.val.int.uti.Version] HV000001: Hibernate Validator
6.1.5.Final
13:09:33,992 INFO  [org.hib.Version] HHH000412: Hibernate ORM core version 5.4.19.Final
13:09:33,995 INFO  [org.hib.ann.com.Version] HCANN000001: Hibernate Commons Annotations
{5.1.0.Final}

```

```
13:09:34,015 INFO [org.hibernate.dialect] HHH000400: Using dialect:  
io.quarkus.hibernate.orm.runtime.dialect.QuarkusPostgreSQL10Dialect  
13:09:51,740 INFO [org.jboss.threads] JBoss Threads version 3.1.1.Final  
WARNING GR-10238: VarHandle for static field is currently not fully supported. Static  
field private static volatile java.lang.System$Logger  
jdk.internal.event.EventHelper.securityLogger is not properly marked for Unsafe access!  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (clinit): 1,368.20 ms, 5.98 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (typeflow): 38,393.51 ms, 5.98 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (objects): 45,980.37 ms, 5.98 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (features): 1,422.10 ms, 5.98 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] analysis: 92,956.99 ms, 5.98 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] universe: 3,041.57 ms, 6.06 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (parse): 7,648.06 ms, 5.88 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (inline): 4,773.40 ms, 5.92 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] (compile): 38,569.02 ms, 6.53 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] compile: 56,705.30 ms, 6.53 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] image: 8,112.44 ms, 6.59 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] write: 1,002.66 ms, 6.59 GB  
[quarkushop-1.0.0-SNAPSHOT-runner:24] [total]: 171,821.50 ms, 6.59 GB  
[INFO] [io.quarkus.deployment.pkg.steps.NativeImageBuildStep] Execute [objcopy, --strip-  
-debug, /home/nebrass/java/playing-with-java-microservices-monolith-  
example/target/quarkushop-1.0.0-SNAPSHOT-runner]  
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in  
202237ms  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 03:43 min  
[INFO] Finished at: 2020-08-08T15:12:16+02:00  
[INFO] -----
```

In the previous Maven log, there is listed a very long Docker command:

```
docker run \
-v /home/nebrass/java/playing-with-java-microservices-monolith-example/target/quarkushop-1.0.0-SNAPSHOT-native-image-source-jar:/project:z \
--env LANG=C \
--user 1000:1000 \
--rm \
quay.io/quarkus/ubi-quarkus-native-image:20.1.0-java11 \
-J-Dsun.nio.ch.maxUpdateArraySize=100 \
-J-DCoordinatorEnvironmentBean.transactionStatusManagerEnable=false \
-J-Djava.util.logging.manager=org.jboss.logmanager.LogManager \
-J-Dvertx.logger-delegate-factory-class \
-name=io.quarkus.vertx.core.runtime.VertxLogDelegateFactory \
-J-Dvertx.disableDnsResolver=true \
-J-Dio.netty.leakDetection.level=DISABLED \
-J-Dio.netty.allocator.maxOrder=1 \
-J-Duser.language=en \
-J-Dfile.encoding=UTF-8 \
--initialize-at-build-time= \
-H:InitialCollectionPolicy=com.oracle.svm.core.genscavenge.CollectionPolicy$BySpaceAndTime \
-H:+JNI -jar quarkushop-1.0.0-SNAPSHOT-runner.jar \
-H:FallbackThreshold=0 \
-H:+ReportExceptionStackTraces \
-H:-AddAllCharsets \
-H:EnableURLProtocols=http,https \
--enable-all-security-services \
--no-server \
-H:-UseServiceLoaderFeature \
-H:+StackTrace quarkushop-1.0.0-SNAPSHOT-runner
```

This very long command is the one executed to build the native executable inside a Docker container based on the [quay.io/quarkus/ubi-quarkus-native-image:20.1.0-java11](#) image, which has the GraalVM support. So, even if you don't have the GraalVM installed locally, you will be able to build the native executable without any problem 😊

We can select the Containerization Engine explicitly using the commands:



```
# To select Docker
mvn package -Pnative -Dquarkus.native.container-runtime=docker
# To select Podman
mvn package -Pnative -Dquarkus.native.container-runtime=podman
```

The produced executable will be a **64 bit Linux executable**, which we will run in a Docker Container. When we generated QuarkuShop, we got a default `Dockerfile.native` file available in the `src/main/docker` directory with the following content:

src/main/docker/Dockerfile.native

```
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY target/*-runner /work/application
RUN chmod 775 /work
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```

Let's build and run the `Dockerfile.native` - before running the container, be sure that your PostgreSQL container is running 😊

```
$ docker build -f src/main/docker/Dockerfile.native -t nebrass/quarkushop-native .
...
Successfully built b14f563446d1
Successfully tagged nebrass/quarkushop-native:latest
```

```
$ docker run --network host --name quarkushop-native -p 8080:8080 nebrass/quarkushop-native
```

WARNING: Published ports are discarded when using host network mode

Series

Powered by Quarkus

1.7.0.Final

2020-08-08 13:42:37,722 INFO [org.flyway.core.intlic.VersionPrinter] (main) Flyway
Community Edition 6.5.3 by Redgate

2020-08-08 13:42:37,725 INFO [org.fly.cor.int.dat.DatabaseFactory] (main) Database: jdbc:postgresql://localhost:5432/demo (PostgreSQL 9.6)

2020-08-08 13:42:37,729 INFO [org.fly.cor.int.com.DbMigrate] (main) Current version of schema "public": 1.1

2020-08-08 13:42:37,729 INFO [org.fly.cor.int.com.DbMigrate] (main) Schema "public" is up to date. No migration necessary.

2020-08-08 13:42:37,799 INFO [io.quarkus] (main) quarkushop 1.0.0-SNAPSHOT native (powered by Quarkus 1.7.0.Final) started in 0.085s. Listening on: http://0.0.0.0:8080

2020-08-08 13:42:37 799 TNEO [io.quarkus] (main) Profile prod activated

1

Hakuna Matata ! ☺ All good ! The application is running and available on the port **8080**.



You can be asking yourself, why there is (**main**) **Profile prod activated**? ☺ We build the application using the **native** profile ? ☺ Why there is **prod** here? ☺

The shown profile here is the application runtime profile: **prod**. As we said before, the Quarkus application have 3 predefined profiles: **dev**, **test** and **prod**. When we run a packaged application we are in the **prod** profile, although when we run the source code using **mvn quarkus:dev** we are obviously in the dev mode.

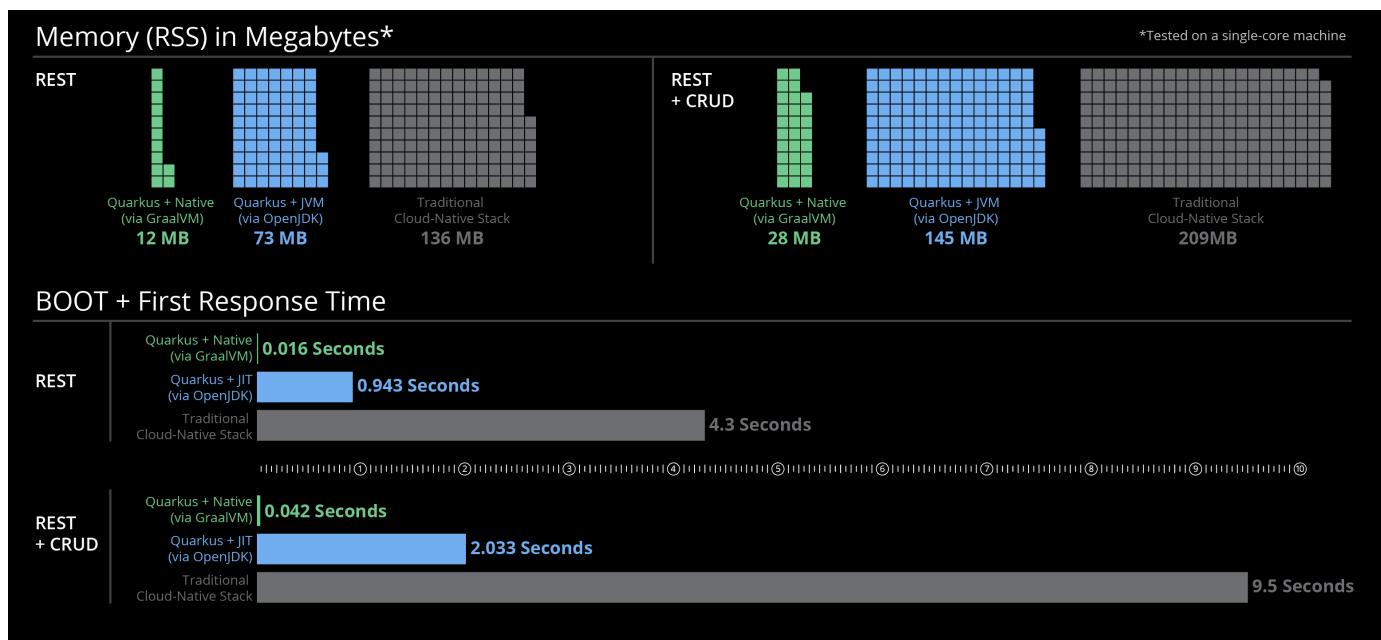
To resume, the Maven Native profile is for building the source code, while the **prod** is the runtime profile. ☺

Good ! We built our Docker Image based on the Native Binary of QuarkusShop: the **supersonic subatomic** Java Binary build with Quarkus. But, how performant is Quarkus ? It's really giving better results than Spring Boot ? Does the Native Image is really optimizing the startup time and is it reducing the memory footprint ?

Difference between JVM and Native modes

Here, we will need to do many scientific researches to check if Quarkus is really so performant ☺ Performance is the first selling point of Quarkus. We will find endless comparison reports that showcases the differences between Quarkus JVM, Quarkus Native and even with many other frameworks like Spring Boot.

We will start by the most famous metrics graphic made by the Quarkus Team that compares Quarkus JVM and Native modes to a traditional Cloud Native Stack (which I think it's Spring Boot ☺)



Memory RSS (which stands for Resident Set Size) is the amount of physical memory currently allocated and used by a process (without swapped out pages). It includes the code, data and shared libraries (which are counted in every process which uses them).

But we all know that this stuff is purely a marketing material ☺ Why not we don't try this ourselves ☺

Before starting this part, I thought of creating a full environment with some **Hello World** REST APIs using **Spring Boot**, **Quarkus JVM** and **Quarkus Native**. Before starting the task, I made a small search in  **GitHub** to check if there is any similar project. Fortunately, I found [an excellent Lab made by Harald Reimüller](#). The Lab makes a benchmarking and metrics collection for sample REST and REST + CRUD applications featuring many frameworks:

- Payara Micro
- Spring Boot
- Quarkus JVM and Native
- and even Python ! Which I will omit as out of scope 😊

I forked the project (which is under [MIT License](#)) and I updated the versions to latest so the benchmarking can be more efficient. You can find it on my  **GitHub Repository** [nebrass/quarkus-performance](#)

I will not explain in details what the lab is going to do, obviously 😊 but I will list the steps globally:

1. All the tasks are executed inside a Docker container based on **CentOs 8**
2. The first step is to install all the required software, like **Maven**, **GraalVM 20.1.0 CE** and even **Enterprise Edition**, **Python** and the tests tools like **Jabba** (Java versions manager similar to **NVM**), etc.
3. Build the Docker Image as run it, and while accessing its bash, we will build the source code of all the sample applications (All Java variants + Python).
4. Then, we will apply a benchmark script on each built binary via a specific runtime. The benchmark script is a load-test based on the **Apache Benchmarking tool**.
5. Then, a Python **Matplotlib** (plotting library for Python) figure is generated for each case. This figures will contain a visualization of the metrics for the **CPU & Memory utilisation**.

What is the difference between GraalVM CE vs EE ?

GraalVM is distributed as **Community** and **Enterprise** editions.

GraalVM Community Edition is open source software built from the sources available on GitHub and distributed under version 2 of the GNU General Public License with the "Classpath" Exception, which are the same terms as for Java.

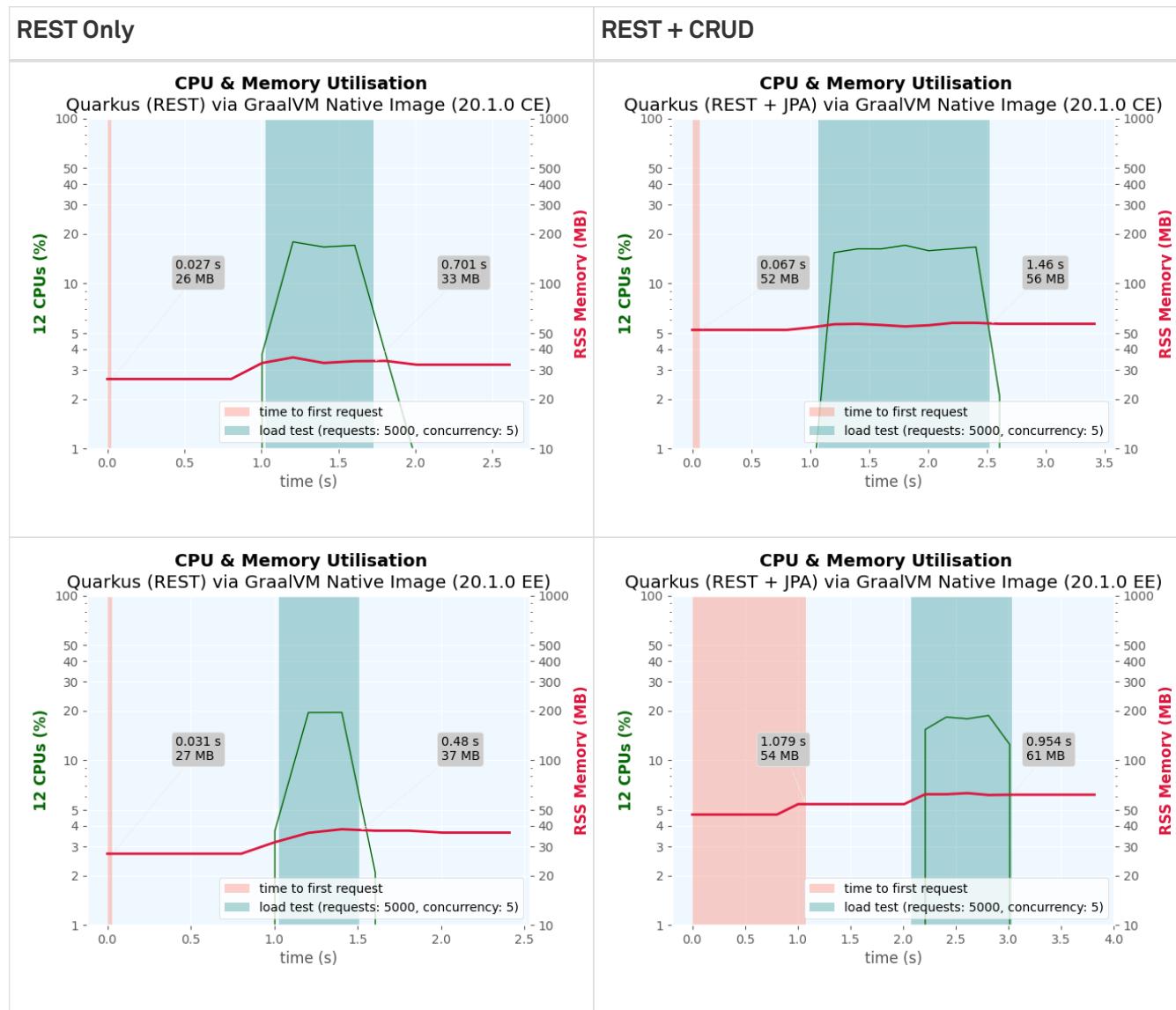
GraalVM Community is free to use for any purpose and comes with no strings attached, but also no guarantees or support.

Oracle GraalVM Enterprise Edition is licensed under either the GraalVM OTN License Agreement, which is free for testing, evaluation, or for developing non-production applications, or under the terms of the Oracle Master License Agreement for customers.

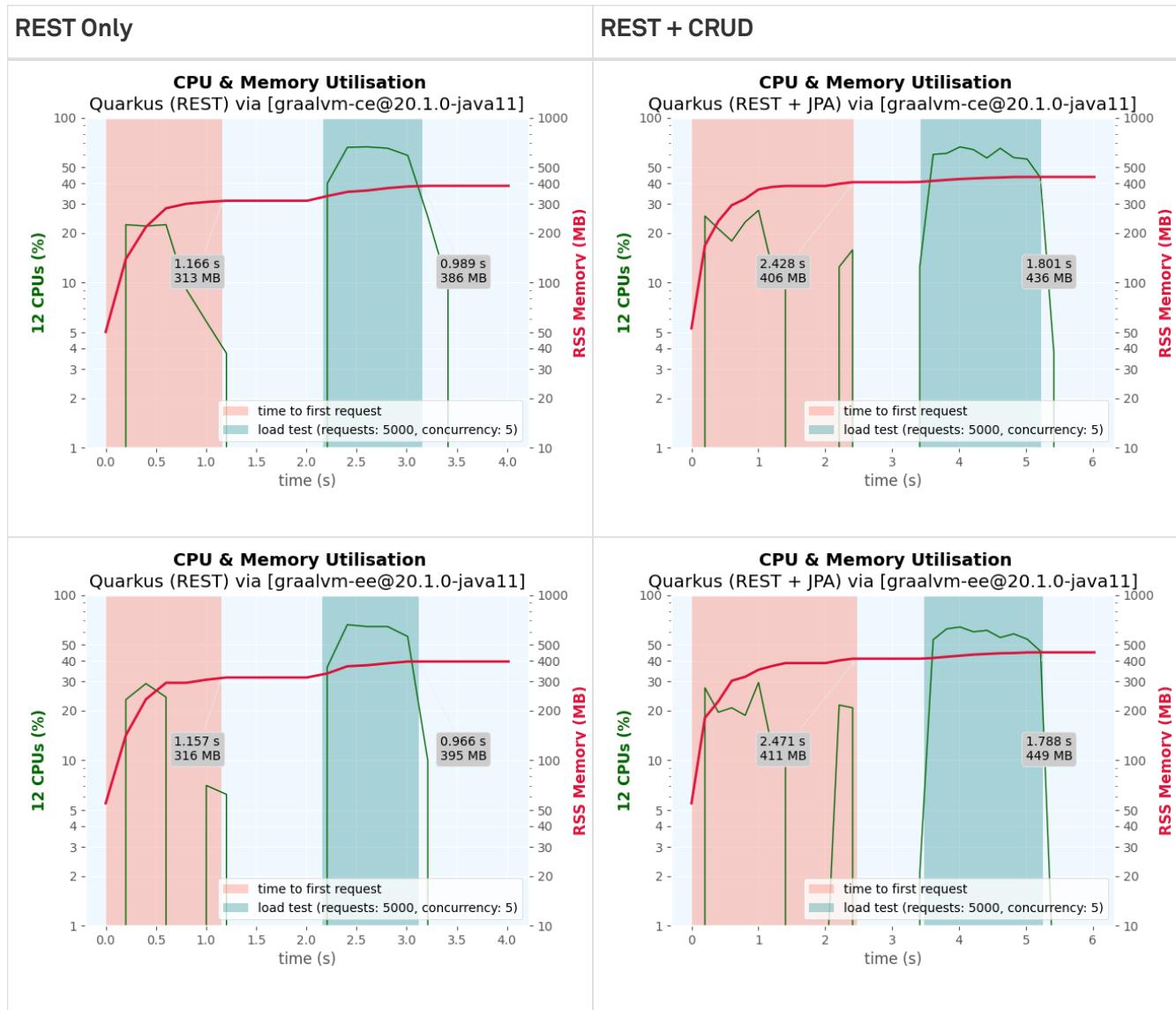
— Official GraalVM Documentation from Oracle, <https://www.graalvm.org/docs/why-graal/>

The benchmark results:

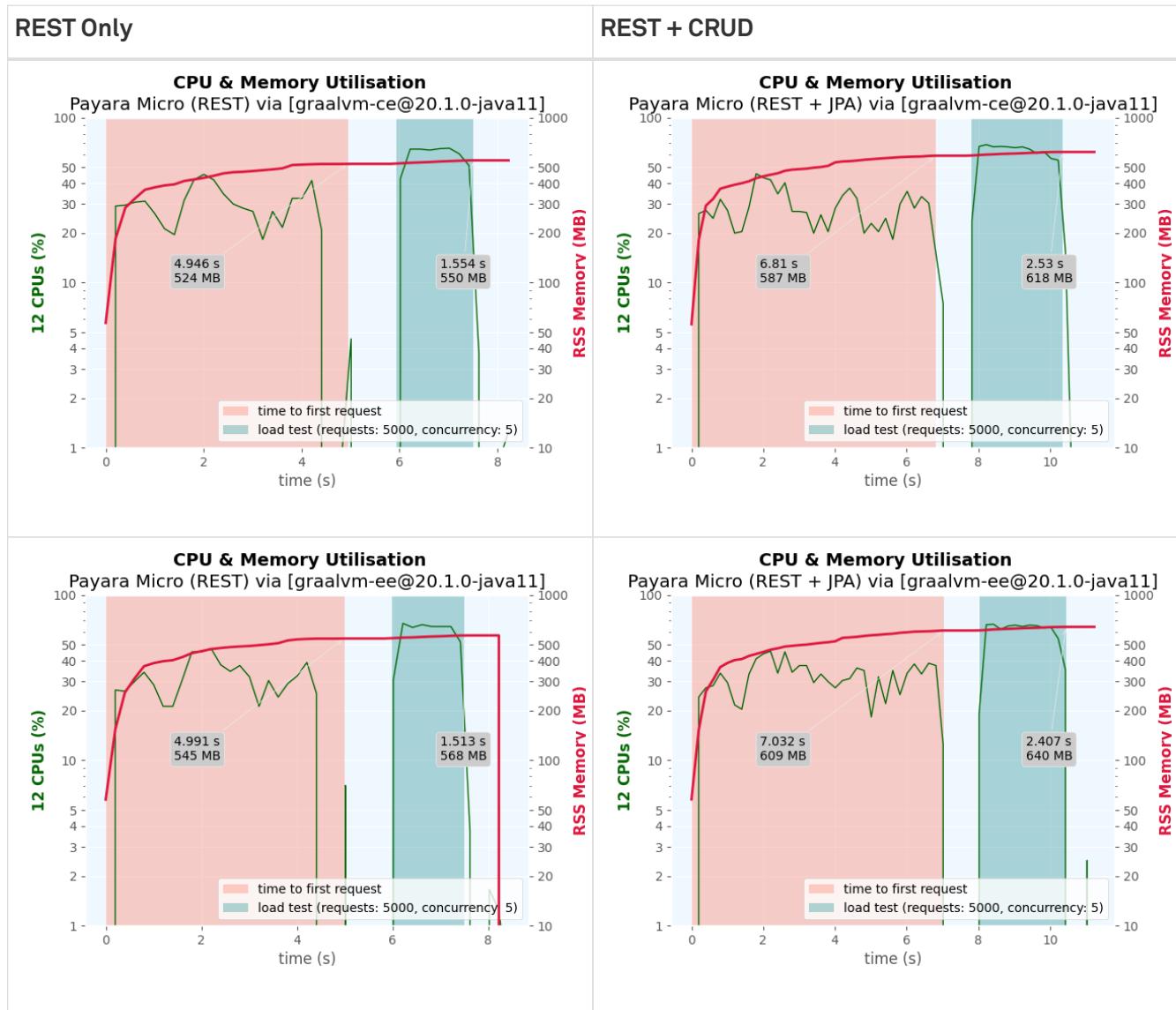
- Quarkus via GraalVM Native Image



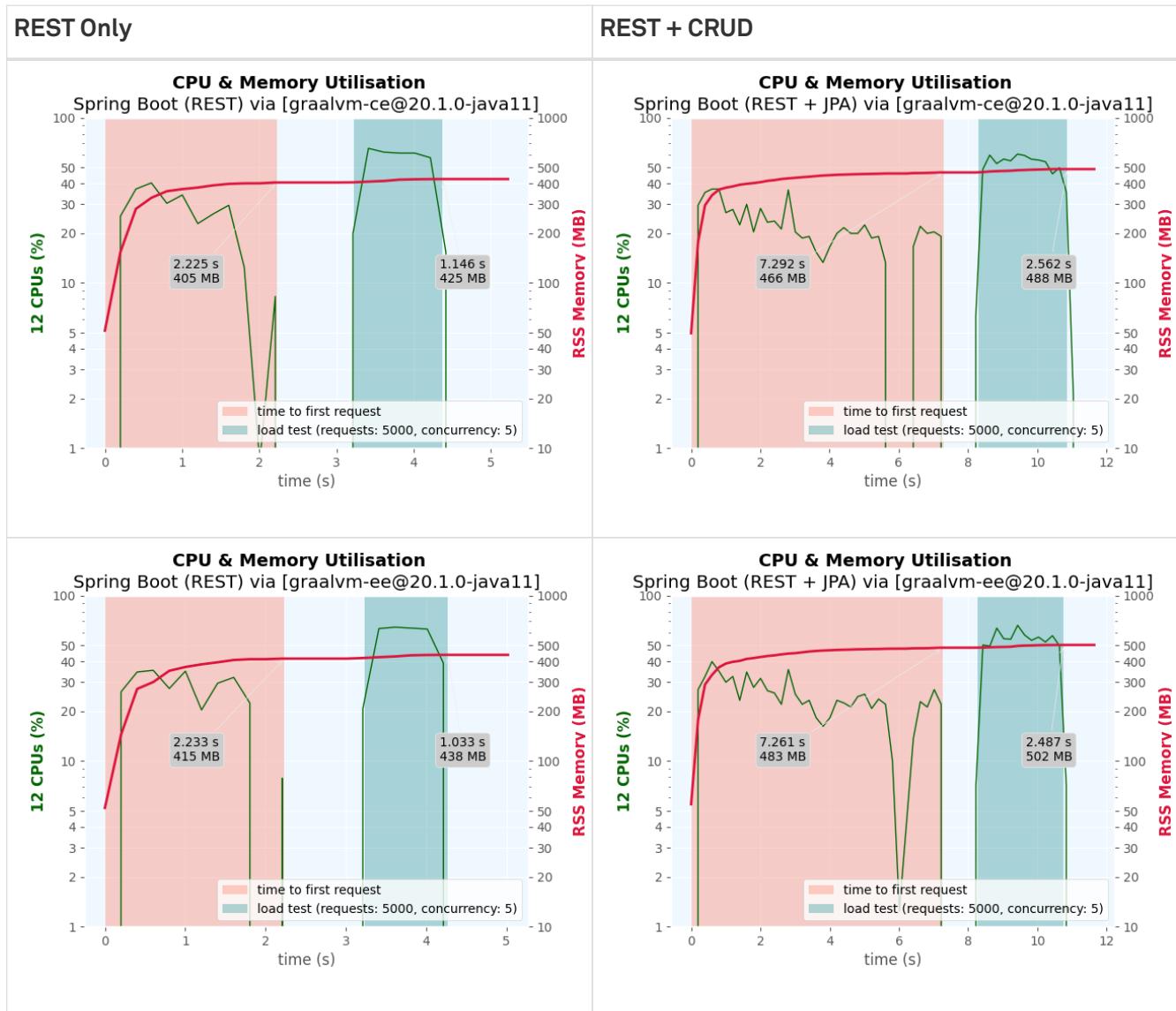
- Quarkus via Java Runtime



- Payara Micro via Java Runtime



- Spring Boot via Java Runtime



For a REST+CRUD app like QuarkuShop, we have for the first request:

- Quarkus Native: Response time: 0.067s and Memory RSS: 52 MB
- Quarkus JVM: Response time: 2.428s and Memory RSS: 406 MB → Quarkus Native is x36 faster and x7 lighter
- Spring Boot: Response time: 7.292s and Memory RSS: 466 MB → Quarkus Native is x108 faster and x8 lighter
- Payara Micro: Response time: 6.81s and Memory RSS: 587 MB → Quarkus Native is x101 faster and x11 lighter

You can easily notice that there is **REALLY** a huge difference in the performance: **Quarkus Native** is really our champion 🏆😊

What is the magic behind the GraalVM's power ?

Running your application inside a Java VM comes with startup and footprint costs.

The image generation process employs static analysis to find any code reachable from the `main()` Java method and then performs full ahead-of-time (AOT) compilation.

What is Ahead-Of-Time (AOT) compilation ?

Ahead-Of-Time (AOT) compiler produces native code in the form of a shared library for the Java methods in specified Java class files. The JVM can load these AOT libraries and use native code from them when corresponding Java methods are called. By using Java AOT Static Compiler `jaotc`, there is no need to wait for the JIT compiler to generate (by compiling bytecode) the fast native code for these Java methods. The code is already generated by `jaotc` and ready to be immediately used. For the same reason, there is no need to execute these methods in the Interpreter because fast compiled native code can be executed instead.

The resulting native binary contains the whole program in machine code form for its immediate execution. It can be linked with other native programs and can optionally include the GraalVM compiler for complementary Just-In-Time (JIT) compilation support to run any GraalVM-based language with High Performance. For additional performance, native images can be built with profile guided optimizations gathered in a previous run of the application.

What is Just-In-Time (JIT) compilation ?

Interpreters are usually a lot slower than native code executing on a real processor, the JVM can run another compiler, which is called JIT compiler which will compile the bytecode into Machine code that can be run by the processor. The JIT compiler is more sophisticated than the **javac** compiler, and it runs complex optimizations to generate high-quality machine code.

In order to speed up the performance, JIT Compiler communicates with JVM at the execution time in order to compile byte code sequences into native machine code. Basically, when using JIT Compiler, the native code is easily executed by the hardware when compared to JVM Interpreter. By doing so, there will be a huge gain in execution speed.

When the JIT Compiler compiles the series of byte code, it also performs certain optimizations such as data analysis, translation from stack operations to register operations, eliminating subexpressions, etc. This makes Java very efficient when it comes to execution and performance.

This powerful combination is making the magic ! The **SUPersonic Subatomic Java** story is made here ! ☺

Conclusion

QuarkuShop is ready to be tested, built and distributed. We will enjoy the power of the GraalVM to produce a powerful and blazing fast Native Binary.

As we are using Maven as a build tool, our application can be built and deployed easily to production using CI/CD pipelines, thru **Jenkins** or **Azure DevOps** for example.

CHAPTER FOUR

Building & Deploying the Monolithic
application

Introduction

In QuarkuShop, we are using **Maven** as a build tool, our application can be built and deployed easily to production using **CI/CD pipelines**, thru **Jenkins** or **Azure DevOps** for example.

Importing the Project in Azure DevOps

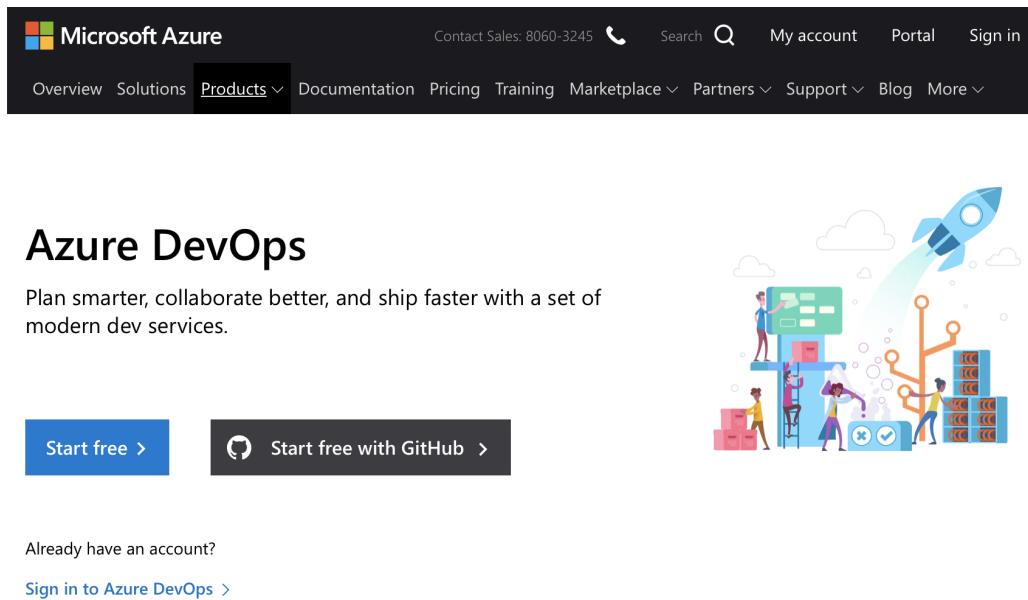
First of all, we need to create a project in **Azure DevOps**. Ok but what is **Azure DevOps** ? 😊

Azure DevOps is a **Software as a Service (SaaS)** from  **Microsoft** that provides many great features for Software Teams – which cover all the lifecycle of a typical application:

- **Azure Pipelines**: CI/CD that works with any language, platform, and cloud (not only **Azure** 😊)
- **Azure Boards**: Powerful work tracking with Kanban boards, backlogs, team dashboards, and custom reporting.
- **Azure Artifacts**: **Maven**, **npm**, and **NuGet** package feeds from public and private sources.
- **Azure Repos**: Unlimited cloud-hosted private Git repos for your project. Collaborative pull requests, advanced file management, and more.
- **Azure Test Plans**: All in one planned and exploratory testing solution.

For the used ☕ Java developers; Yes ! **Azure DevOps** (also known as **ADO**) is a great alternative of **Jenkins/Hudson**. In this chapter, we will showcase how to make easily a full **CI/CD pipeline** for our project.

First of all, go to the **Azure DevOps** portal and Click **[Start free]** to create an account:



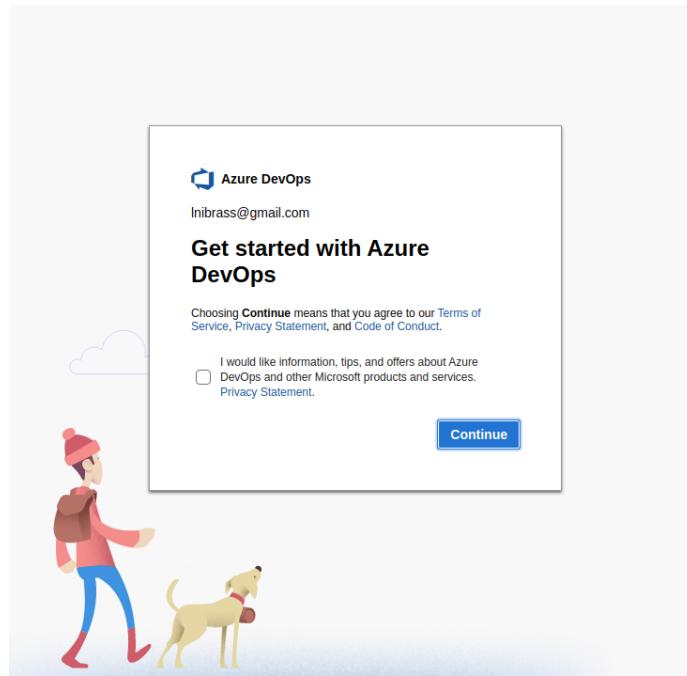
Azure DevOps

Plan smarter, collaborate better, and ship faster with a set of modern dev services.

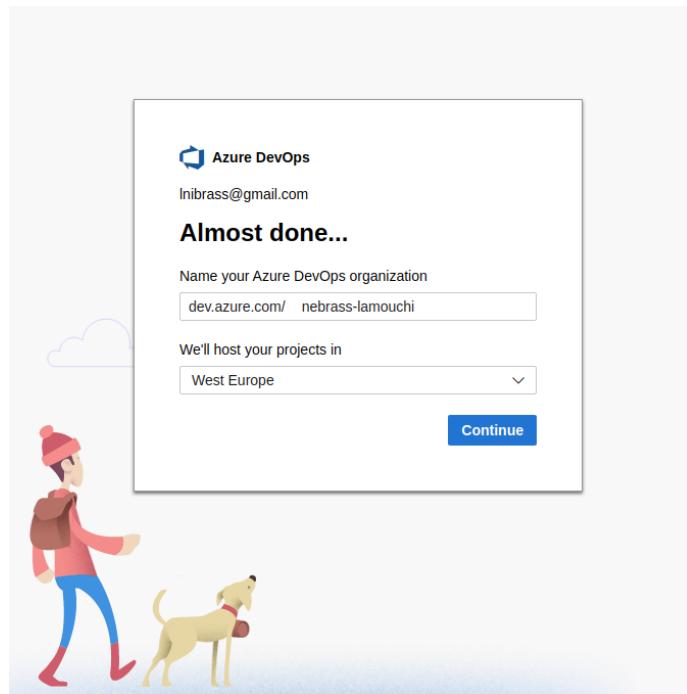
[Start free >](#) [Start free with GitHub >](#)

Already have an account? [Sign in to Azure DevOps >](#)

Next, authenticate to your *Oulook/Hotmail/Live* account (or create a new one ☺) and then confirm enroll:



Next, we need to create an **Azure DevOps Organization**:



Next, create your first **Azure DevOps Project**:

Create a project to get started

Project name *
quarkushop-monolithic-application

Visibility

Public
Anyone on the internet can view the project. Certain features like TFVC are not supported.

Private
Only people you give access to will be able to view this project.

+ Create project

Next, go to **Repos > Files**:

nebrass-lamouchi / quarkushop-monolithic-app... / Repos / Files / quarkushop-monolithic-application

quarkushop-monolithic-application is empty. Add some code!

Clone to your computer

HTTPS SSH https://nebrass-lamouchi@dev.azure.com/nebrass-lamouchi/quarkushop-monolithic-application/ _git/quarkushop-monolithic-application

Generate Git Credentials

Having problems authenticating in Git? Be sure to get the latest version Git for Windows or our plugins for IntelliJ, Eclipse, Android Studio or Windows command line.

Push an existing repository from command line

HTTPS SSH

git remote add origin https://nebrass-lamouchi@dev.azure.com/nebrass-lamouchi/quarkushop-monolithic-application/_git/quarkushop-monolithic-application

Import a repository

Import

Initialize **master** branch with a README or gitignore

Add a README Add a .gitignore: None Initialize

Here we will be interested in "*Push an existing repository from command line*".

ADO generates the **git** commands that you will need to push the local project to the **upstream**. Before running these commands, we just need to make our local project a **git enabled project**.

To do it, just run these commands, which will initiate and add all the files, then it will make the first commit:

```
git init
git add .
git commit -m "first commit"
```

Next, we will run the **git** commands that will push the full project to **ADO**:

```
1 git remote add origin https://nebrass-lamouchi@dev.azure.com/nebrass-
    lamouchi/quarkushop-monolithic-application/_git/quarkushop-monolithic-application
2 git push -u origin --all
```

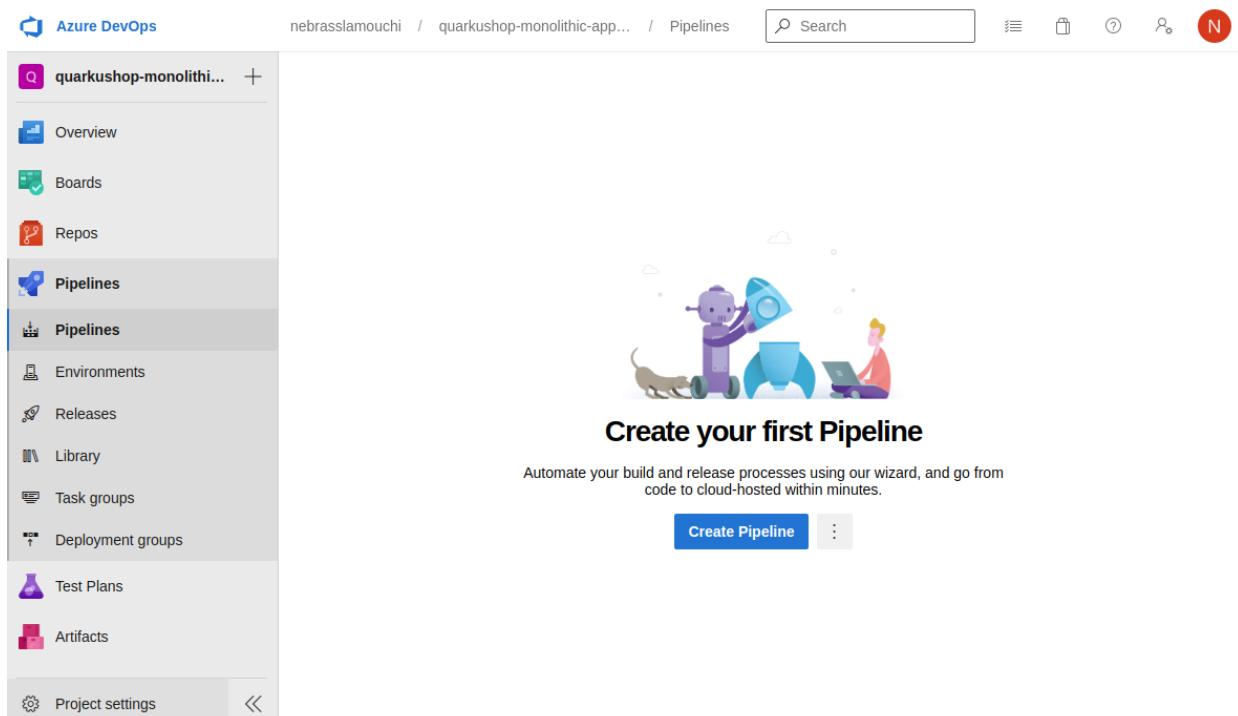
Now, our source code is hosted in **ADO Repositories**. Let's create the **CI/CD Pipelines** ☺

Creating the CI/CD pipelines

The next step will be to configure the **Continuous Integration Pipeline**, which will run each time there is a new code on the main development branch which is the "**Master**" branch in our case.

Creating the Continuous Integration pipeline

To create our first **CI Pipeline**, just go to **[Pipelines]** section and click on **[Create Pipeline]**:



Next, we need to choose where the source code is stored. But we will use the classic editor, so click on **Use the classic editor**:

Azure DevOps nebrasslamouchi / quarkushop-monolithic-app... / Pipelines

Connect Select Configure Review

New pipeline

Where is your code?

- Azure Repos Git YAML
Free private Git repositories, pull requests, and code search
- Bitbucket Cloud YAML
Hosted by Atlassian
- Github YAML
Home to the world's largest community of developers
- Github Enterprise Server YAML
The self-hosted version of GitHub Enterprise
- Other Git
Any generic Git repository
- Subversion
Centralized version control by Apache

Use the classic editor to create a pipeline without YAML.

Next, we select **AzureRepos Git** next, we choose **QuarkuShop Repository** that we just created before:

Azure DevOps nebrasslamouchi / quarkushop-monolithic-app... / Pipelines

✓ Connect **Select** Configure Review

New pipeline

Select a repository

Filter by keywords X

quarkushop-monolithic-application

Pipelines

Then the **Pipeline Configuration** screen will appear. Here we will choose **Maven** pipeline:

The screenshot shows the Azure DevOps interface for pipeline configuration. The left sidebar is titled 'Pipelines' and includes options like Pipelines, Environments, Releases, Library, Task groups, Deployment groups, Test Plans, and Artifacts. The 'Configure' tab is selected at the top. On the right, under 'Configure your pipeline', there are several options: 'Maven' (Build your Java project and run tests with Apache Maven), 'Maven package Java project Web App to Linux on Azure' (Build your Java project and deploy it to Azure as a Linux web app), 'Starter pipeline' (Start with a minimal pipeline that you can customize to build and deploy your code), and 'Existing Azure Pipelines YAML file' (Select an Azure Pipelines YAML file in any branch of the repository). A 'Show more' button is also present.

Then we will insert the most important part, defining the pipeline configuration using definition [YAML](#) file. Then, we will have two choices:

1. Building our **Maven** based Java project directly in **ADO**
2. Building our project using **Docker Multistage builds**

Creating Maven based CI Pipeline

This is the most common case for **Maven** projects that we will build in **ADO**. This is the easiest choice when we think on **Maven** Builds.

Unfortunately, this approach has strong dependencies to the environment. For example, we need to have **JDK 11** and **Maven 3.6.2+**. If one of these dependencies is not found in the host machine, the build will fail obviously



While using this approach, we need to adopt the **CI Platform** for our specific needs and tools.

azure-pipelines.yml

```

trigger:                                ①
- master

pool:                                     ②
  vmImage: 'ubuntu-latest'

steps:
- task: Maven@3
  displayName: 'Maven verify & Sonar Analysis'      ③
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.11'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: true
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'verify sonar:sonar'

- task: Maven@3
  displayName: 'Maven package'                      ④
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.11'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: false
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'package -DskipTests -DskipITs'

- task: CopyFiles@2
  displayName: 'CopyFiles for Target'                ⑤
  inputs:
    SourceFolder: 'target'
    Contents: '*.*'
    TargetFolder: '$(Build.ArtifactStagingDirectory)'

- task: PublishBuildArtifacts@1
  displayName: 'Publish Artifact: drop'              ⑥
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)'
    artifactName: drop

```

① This pipeline will be triggered when there is any update on the **master** branch.

- ② We will be using the latest image of **Ubuntu**. It's **20.04** while writing this book.
- ③ The first task in the **CI scenario** is to run all the tests and generates the **Sonar report**.
- ④ Then, we will package our **Java** application without running tests again, as they were executed in the previous step. We defined the runtime to **Java 11** and we allocated **3Gb** of memory for the **Maven** task.
- ⑤ We will copy the content of the target folder to the predefined **`$(Build.ArtifactStagingDirectory)`**.
- ⑥ Finally, we will upload the content of **`$(Build.ArtifactStagingDirectory)`** as an **Azure DevOps Artifact**.

Creating Docker Multistage based CI Pipeline

This is the trendy way to build a project. This is the best choice to build project that has very specific requirements. Like our **QuarkusShop** application, we need to have **JDK 11 AND GraalVM AND Maven 3.6.2+**. Which cannot be satisfied even in **ADO**, because there is no **GraalVM** in **Azure Pipelines** (at least, until now, while I'm writing these words).

Fortunately, this build approach has no dependency to the environment. Every required component will be installed and configured in the different **Dockerfile** stages.



While using this approach, we bring our specific needed tools to the **CI Platform**.

Let's look to the **Stage 1** of our **Dockerfile.multistage** file:

src/main/docker/Dockerfile.multistage

```
## Stage 1 : build with maven builder image with native capabilities
FROM quay.io/quarkus/centos-quarkus-maven:20.1.0-jav11 AS build
```

In the **Build** stage, we are using on the **centos-quarkus-maven:20.1.0-jav11** Docker Base Image, which comes with **Java 11 with GraalVM, Maven, Podman and Buildah**. Which are exactly what we need as tools and versions.

As you can notice, I'm trying to convince you to adopt this approach 😊 First of all, it's the only possible one, as **ADO** doesn't have **GraalVM**, and moreover, it's the most suitable approach to avoid any unexpected issue. 😊

The `azure-pipelines.yml` for the **Docker Multistage CI Pipeline** looks like this:

`azure-pipelines.yml`

```

trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Maven@3
  displayName: 'Maven verify & Sonar Analysis'
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.11'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: true
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'verify sonar:sonar'

- task: Docker@2
  displayName: 'Docker Multistage Build'          ①
  inputs:
    containerRegistry: 'nebrass@DockerHub'
    repository: 'nebrass/quarkushop-monolithic-application'
    command: 'build'
    Dockerfile: '**/Dockerfile.multistage'
    buildContext: '.'
    tags: |
      $(Build.BuildId)
      latest

- task: Docker@2
  displayName: 'Push Image to DockerHub'         ②
  inputs:
    containerRegistry: 'nebrass@DockerHub'          ③
    repository: 'nebrass/quarkushop-monolithic-application'
    command: 'push'

```

① We are using the first **Docker@2** task to:

- build the **Docker Image** based on the **Dockerfile.multistage** file.
- name the built image to **nebrass/quarkushop-monolithic-application**.
- tag the built image with **\$(Build.BuildId)** which is an **ADO Builds** variables and with the **latest** tag.

- ② We are using the second **Docker@2** task to push the [nebrass/quarkushop-monolithic-application](#) image to my **Docker Hub** account.
- ③ I'm using an **Azure Service Connection** that I called [nebrass@DockerHub](#) which stores my **Docker Hub** credentials.



The @ in **Docker@2** is used to define the selected version of the **Docker** task in **ADO**.



To learn more about creating the **Azure Service Connection** to **SonarCloud**, check [this excellent Azure DevOps Labs tutorial](#).

Don't be surprised! 😊 I didn't delete the **Maven verify & Sonar Analysis** task. Unfortunately, the **Testcontainers** library that we are using for our **Integration Tests** cannot be invoked from inside the **Docker** context. This is why I decided to run the tests using the **Maven** command, and then to complete all the steps inside the **Docker** containers.

Our **CI Pipeline** is missing one requirement: the **SONAR_TOKEN** environment variable, that **Maven** will use to authenticate to **SonarCloud** to publish the analysis report generated in our pipeline.

To define an environment variable in an **Azure Pipeline**, go to:

Pipelines > **Choose your pipeline** > [Edit] > [Variables] > [New variable] then define the environment variable to **SONAR_TOKEN** and give it the **SonarCloud token** that you got while creating your project.

Without the **SONAR_TOKEN** environment variable, we will have the **sonar:sonar** failing with an error message: **java.lang.IllegalStateException: You're not authorized to run analysis. Please contact the project administrator.**

At this level, the **CI Pipeline** is ready. We need now to start looking to the **Continuous Deployment Pipeline**.

Making the Continuous Deployment pipeline

For the deployment part, our **Docker Container** can be deployed to many products and locations:

- **Kubernetes Clusters** 😊 We are not there yet 😊
- managed **Docker** hosting slot (**Azure Container Instance**, **Amazon Container Service**, **Docker Enterprise**, etc.)
- **virtual machines**

In our case, we will be using an **Azure VM** to deploy our **Docker Container**. You can follow the same procedure to make the same **CD Pipeline**.

Create the Virtual Machine

The first step, is to create an **Azure Resource Group**, which is a logical group that will hold our **Azure Resource**, like the **Virtual Machine** that we want to create.

The screenshot shows the 'Create a resource group' wizard in the Microsoft Azure portal. The 'Basics' tab is selected. The 'Project details' section shows a 'Subscription' dropdown set to 'MSDN - Visual Studio Subscription' and a 'Resource group' input field containing 'quarkushop-rg'. The 'Resource details' section shows a 'Region' dropdown set to '(Europe) France Central'. At the bottom, there are buttons for 'Review + create', '< Previous', and 'Next : Tags >'.

Next, we will create the **VM** - by defining:

- **VM Name:** quarkushop-vm
- **Region:** France Central
- **Image:** Ubuntu Server 18.04 LTS
- **Size:** Standard_B2ms 2 vCPUS + 8 GB of RAM

The screenshot shows the Microsoft Azure portal interface for creating a new virtual machine. The top navigation bar is blue with the text "Microsoft Azure". Below it, the breadcrumb navigation shows "Home > Resource groups > quarkushop-rg > New > Create a virtual machine". The main title "Create a virtual machine" is centered above the form.

The "Basics" tab is selected, indicated by a blue underline. Other tabs include Disks, Networking, Management, Advanced, Tags, and Review + create.

The main content area contains instructions: "Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image. Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization. [Learn more](#)".

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription: MSDN - Visual Studio Subscription

Resource group: quarkushop-rg (dropdown with "Create new" option)

Instance details

Virtual machine name: quarkushop-vm

Region: (Europe) France Central

Availability options: No infrastructure redundancy required

Image: Ubuntu Server 18.04 LTS (dropdown with "Browse all public and private images" link)

Azure Spot instance: No (radio button selection)

Size: Standard_B2ms - 2 vcpus, 8 GiB memory (58,11 €/month) (dropdown with "Select size" link)

Next, you will need to define:

- **Authentication type:** Password
- **Username:** nebrass
- **Password** and confirm it - I will not give you mine 😊
- Be sure to keep the **SSH Port:** allowed

Administrator account

Authentication type SSH public key Password

Username *	nebrass	✓
Password *	*****	✓
Confirm password *	*****	✓

Inbound port rules

Select which virtual machine network ports are accessible from the public internet. You can specify more limited or granular network access on the Networking tab.

Public inbound ports * None Allow selected ports

Select inbound ports *

⚠️ This will allow all IP addresses to access your virtual machine. This is only recommended for testing. Use the Advanced controls in the Networking tab to create rules to limit inbound traffic to known IP addresses.

[Review + create](#) < Previous [Next : Disks >](#)

Confirm the creation by clicking on **[Review + create]** then in the validation screen, hit **[Create]**

Delete
 Cancel
 Redeploy
 Refresh

We'd love your feedback! →

✓ Your deployment is complete

Deployment name: CreateVm-Canonical.UbuntuServer-18.04-LTS-...
 Subscription: MSDN - Visual Studio Subscription
 Resource group: quarkushop-rg

Start time: 8/11/2020, 12:07:27 AM
 Correlation ID: e8b051e9-55e6-4b9a-8252-1dddfec5d915

✓ Deployment details ([Download](#))

^ Next steps

Setup auto-shutdown Recommended

Monitor VM health, performance and network dependencies Recommended

Run a script inside the virtual machine Recommended

[Go to resource](#)
[Create another VM](#)

Check the created **VM** by clicking on [**Go to resource**]

Resource group (change)
quarkushop-rg

Status
Running

Location
France Central

Subscription (change)
MSDN - Visual Studio Subscription

Subscription ID
11d4c490-bad4-4b10-9765-69446837aee6

Tags (change)
Click here to add tags

Operating system
Linux (ubuntu 18.04)

Size
Standard B2ms (2 vcpus, 8 GiB memory)

Public IP address
51.103.26.144

Virtual network/subnet
quarkushop-rg-vnet/default

DNS name
Configure

Next, we need to add an exception for the incoming access to the **8080** port (which is the **QuarkuShop HTTP Port**) in the **VM Networking Security Rules**: to do it, go to the [**Networking**] section of the **Azure Virtual Machine**, we need to add [**Add inbound port rule**]:

Search (Ctrl+)/

Attach network interface **Detach network interface**

IP configuration ipconfig1 (Primary)

Network Interface: quarkushop-vm108 Effective security rules

Inbound port rules Outbound port rules Application security group

Priority	Name	Port	Protocol
300	SSH	22	TCP
65000	AllowVnetInBound	Any	Any
65001	AllowAzureLoadBal...	Any	Any
65500	DenyAllInBound	Any	Any

Add inbound security rule

Basic

Source * Any

Source port ranges * Any

Destination * Any

Destination port ranges * 8080

Protocol * TCP

Action * Allow

Priority * 310

Name * Port_8080

Description

Add

You have already here the **IP Address** for the created **VM**. To access it; just open an **SSH session** from the terminal. We will use the defined credentials for accessing the **VM instance**:

```
$ ssh nebrass@51.103.26.144
The authenticity of host '51.103.26.144 (51.103.26.144)' can't be established.
ECDSA key fingerprint is SHA256:bio07HNjtKKgy8g7EAgfR+82Pz4gFyEm10QyMjpLNvK.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '51.103.26.144' (ECDSA) to the list of known hosts.
nebrass@51.103.26.144's password:
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.3.0-1034-azure x86_64)

...
nebrass@quarkushop-vm:~$
```

We will begin by updating the **Virtual Machine**:

```
sudo apt update && sudo apt upgrade
```

We will install **Docker Engine**:

```
$ sudo apt-get install apt-transport-https \
    ca-certificates curl gnupg-agent \
    software-properties-common
```

We will add the official **Docker GPG key**:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Use the following command to set up the stable repository:

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
```

Now, it's time to install (finally 😊) the **Docker Engine**:

```
$ sudo apt-get update && sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Next, we will install **Docker Compose**:

```
$ sudo curl -L \
  "https://github.com/docker/compose/releases/download/1.26.2/docker-compose-$(uname -s) \
  -$(uname -m)" \
  -o /usr/local/bin/docker-compose
```

Apply executable permissions to the binary:

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

Then, create the **docker-compose.yml** file under the **/opt/** folder:

```
version: '3'
services:
  quarkushop:
    image: nebrass/quarkushop-monolithic-application:latest
    environment:
      - QUARKUS_DATASOURCE_JDBC_URL=jdbc:postgresql://postgresql-db:5432/prod
    ports:
      - 8080:8080
  postgresql-db:
    image: postgres:13
    volumes:
      - /opt/postgres-volume:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=developer
      - POSTGRES_PASSWORD=p4SSW0rd
      - POSTGRES_DB=prod
      - POSTGRES_HOST_AUTH_METHOD=trust
    ports:
      - 5432:5432
```

Then, we need to create a local folder in the Azure VM that will be used as Docker volume for PostgreSQL:

```
$ sudo mkdir /opt/postgres-volume
$ sudo chmod 777 /opt/postgres-volume
```

Now, the **Azure VM** is ready to be used as our production runtime ☺ Let's move to the **CD Pipeline**.

Create the Continuous Deployment pipeline

Go back to **Azure DevOps**, then go to **Pipelines > Releases**:

The screenshot shows the Azure DevOps interface with the project 'quarkushop-monolith...' selected. The left sidebar has 'Pipelines' and 'Releases' both highlighted. The main area displays a cartoon illustration of a person and a dog at a launch pad with a rocket. Below the illustration, the text 'No release pipelines found' is centered, followed by the subtext 'Automate your release process in a few easy steps with a new pipeline'. A prominent blue button labeled 'New pipeline' is located below the subtext.

Next, click on **[New pipeline]** and then click on **[Empty job]**

The screenshot shows the 'Select a template' dialog box. The left sidebar is identical to the previous one. The dialog box has a search bar at the top right. It displays a section titled 'Select a template' with the subtext 'Or start with an **Empty job**'. Below this is a 'Featured' section containing several deployment templates: 'Azure App Service deployment', 'Deploy a Java app to Azure App Service', 'Deploy a Node.js app to Azure App Service', 'Deploy a PHP app to Azure App Service and Azure Database for MySQL', 'Deploy a Python app to Azure App Service and Azure database for MySQL', 'Deploy to a Kubernetes cluster', and 'IIS website and SQL database deployment'. Each template has a brief description and an 'Apply' button.

Then, add an **Artifact**, then we will choose **Build** as the source type, and we need to select our Project and Source from the list:

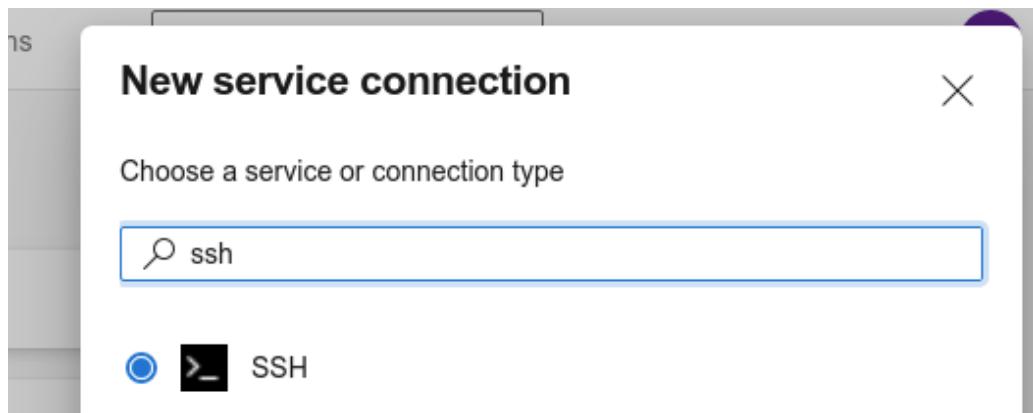
Next, click on the available **Stage 1** and click on the Agent job, change the **Agent Specification** to **ubuntu-20.04**.

Next, add a task to the pipeline:

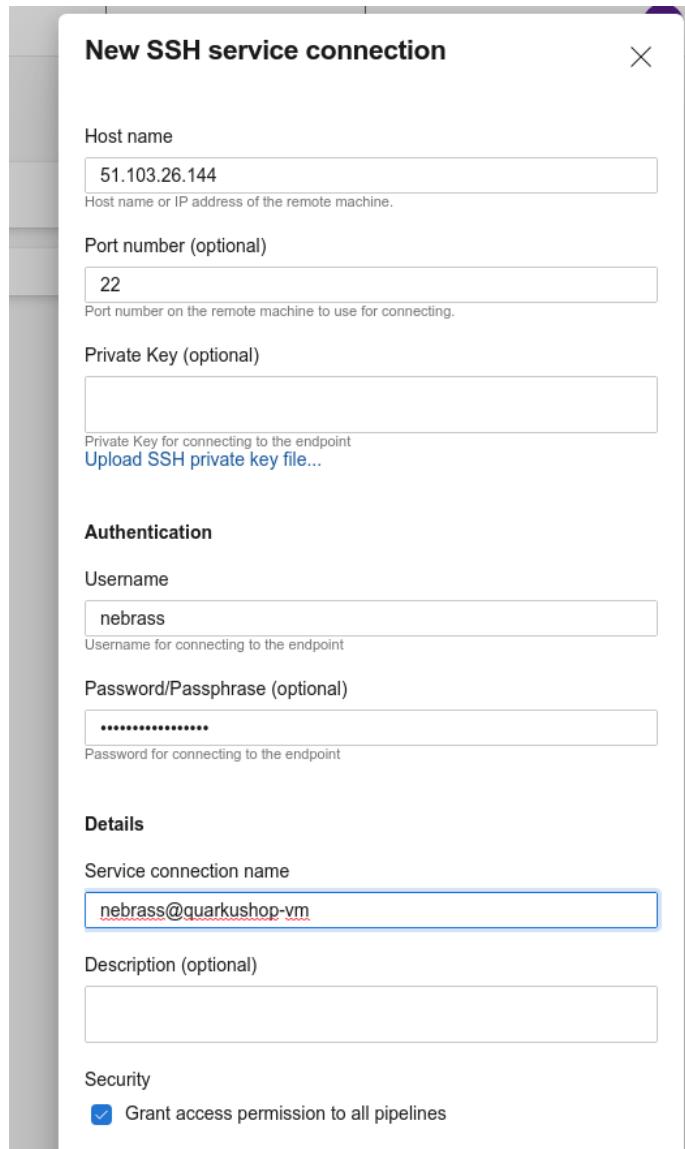
Click on the added task and click **Manage** near **SSH service connection**, to access the **Service Connections** manager:

The screenshot shows the Jenkins interface for managing service connections. On the left, there's a sidebar with various project settings like General, Boards, and Repos. The main content area is titled 'Service connections' and contains a search bar and a 'New service connection' button. A single service connection, 'nebrass@DockerHub', is listed.

Next, click on **New service connection** and search for **SSH** and choose it:



In the next screen, we need to configure the access to the **Azure VM instance**:



Go back to the **release pipeline** screen and hit the refresh button for the **SSH service connection** - and next choose the created **Service Connection**:

Next, in the **Commands** add these **docker-compose** instructions:

```
docker login -u $(docker.username) -p $(docker.password)
docker-compose -f /opt/docker-compose.yml stop
docker-compose -f /opt/docker-compose.yml pull
docker-compose -f /opt/docker-compose.yml rm quarkushop
docker-compose -f /opt/docker-compose.yml up -d
```

These commands will stop all the **Docker Compose** services, pull the latest images, delete the **quarkushop** service and will create it again.



The `$(docker.username)` and `$(docker.password)` are my **Docker Hub** credentials. We will define them as **Environment Variables**.

Then hit [Save]

The screenshot shows the Jenkins Pipeline configuration interface. The pipeline has one stage named "Stage 1" which is a "Deployment process". It contains an "Agent job" task. The task is set to "Run on agent" and uses an "SSH" connection. The "Display name" is "Run shell commands on remote machine". The "SSH service connection" is "nebrass@quarkushop-vm". Under the "Run" section, the "Commands" radio button is selected, and the command entered is:

```
docker login -u $(docker.username) -p $(docker.password)
cd /opt/
docker-compose stop
docker-compose pull
docker-compose rm -f
docker-compose up -d
```

Then go [**Variables**] to define the **environment variables**:

The screenshot shows the Jenkins Pipeline Variables configuration interface. The left sidebar lists "Pipeline variables", "Variable groups", and "Predefined variables". The main area shows a table of predefined variables:

Name	Value	Scope
docker.username	nebrass	Release
docker.password	XXXXXXXXXXXXXX	Release

Building & Deploying the Monolithic application

The last step is to activate the trigger:

The screenshot shows the 'Continuous deployment trigger' configuration in the Azure DevOps Pipelines interface. It includes a section for 'Artifacts' with a build reference to '_quarkushop-monolithic-application'. The 'Enabled' toggle switch is turned on, and a note states: 'Creates a release every time a new build is available.' Below this is a 'Build branch filters' section with a note: 'No filters added.' and a '+ Add' button. A 'Pull request trigger' section follows, with a build reference to '_quarkushop-monolithic-application', a 'Disabled' toggle switch, and a note: 'Enabling this will create a release every time a selected artifact is available as part of a pull request workflow.'

Finally, save the modifications and trigger the release by clicking on **Create release**:

The screenshot shows the 'Create a new release' dialog box overlaid on the pipeline configuration. It includes a 'Pipeline' section with a note: 'Click on a stage to change its trigger from automated to manual.' A 'Stage 1' dropdown menu is open. To the right, there's a 'Artifacts' section where users can select an artifact source, with a note: 'Select the version for the artifact sources for this release.' A table shows a single entry: '_quarkushop-monolithic-applic...' with Version '20200809.2'. At the bottom are 'Release description' and 'Create' and 'Cancel' buttons.

Yopppaaa !! When the **Release Pipeline** execution finishes, just open the **URL** in your browser:
IP_ADDRESS:8080.

For example, in my case **QuarkuShop** is reachable on 51.103.26.144:8080, you will get the default response:

Resource not found

Finally, we can access the predefined **Quarkus index.html** page:

Your new Cloud-Native application is ready!

Congratulations, you have created a new Quarkus application.

Why do you see this?

This page is served by Quarkus. The source is in [src/main/resources/META-INF/resources/index.html](#).

What can I do from here?

If not already done, run the application in *dev mode* using: `mvn compile quarkus:dev`.

- Add REST resources, Servlets, functions and other services in [src/main/java](#).
- Your static assets are located in [src/main/resources/META-INF/resources](#).
- Configure your application in [src/main/resources/application.properties](#).

Do you like Quarkus?

Go give it a star on [GitHub](#).

How do I get rid of this page?

Just delete the [src/main/resources/META-INF/resources/index.html](#) file.

Application

GroupId: com.targa.labs
ArtifactId: quarkushop
Version: 1.0.0-SNAPSHOT
Quarkus Version: 1.6.0.Final

Next steps

[Setup your IDE](#)
[Getting started](#)
[Quarkus Web Site](#)

We have also the **Swagger-UI** in our **QuarkuShop** production environment reachable on 51.103.26.144:8080/api/swagger-ui/

The screenshot shows the Swagger UI interface for the generated API. The main title is "Generated API 1.0 OAS3". Below it, there's a link to "/api/openapi". The "default" endpoint is selected. A list of API endpoints is shown:

- GET /api/carts**
- GET /api/carts/active**
- GET /api/carts/customer/{id}**
- POST /api/carts/customer/{id}** (highlighted in green)
- GET /api/carts/{id}**
- DELETE /api/carts/{id}** (highlighted in red)
- GET /api/categories**
- POST /api/categories** (highlighted in green)

Good! 😊 Let's now change the version listed in this index page and we push the modification, to see if our **CI/CD Pipeline** is working like expected 😊

Your new Cloud-Native application is ready!

Congratulations, you have created a new Quarkus application.

Application

GroupId: com.targa.labs
ArtifactId: quarkushop
Version: 1.0.0-SNAPSHOT
Quarkus Version: 1.7.0.Final

Why do you see this?

This page is served by Quarkus. The source is in [src/main/resources/META-INF/resources/index.html](#).

Next steps

[Setup your IDE](#)
[Getting started](#)
[Quarkus Web Site](#)

What can I do from here?

If not already done, run the application in *dev mode* using: `mvn compile quarkus:dev`.

- Add REST resources, Servlets, functions and other services in `src/main/java`.
- Your static assets are located in `src/main/resources/META-INF/resources`.
- Configure your application in `src/main/resources/application.properties`.

Do you like Quarkus?

Go give it a star on [GitHub](#).

How do I get rid of this page?

Just delete the `src/main/resources/META-INF/resources/index.html` file.

Yoppaaa! Our **CI/CD Factory** is working like a charm ! 😊 Congratulation ! 🎉

Conclusion

QuarkuShop has now a dedicated **CI/CD Pipeline**:

- The **Continuous Integration Pipeline**: On each commit on the master branch, the pipeline will run all the tests and build the **Native Binary**, which will be packaged inside a **Docker Image**.
- The **Continuous Deployment Pipeline**: When the **CI succeeds**, the **Docker Image** will be deployed to the **Azure VM** 😊

In the next chapter, we will implement the more layers:

- **Security**: to avoid that the Unauthenticated visitors to access our application.
- **Monitoring**: to be sure that the application is running correctly and to avoid any bad surprises.

WHY WAIT FOR THE DISASTER TO HAPPEN ? 😊

CHAPTER FIVE

Adding the anti-disasters layers

Introduction

Writing code, unit tests, integration tests, code quality analysis, CI/CD Pipelines. Many developers think that the trip ends here, and a new iteration will start again. We forget the application runtime. No ! 😞 I don't mean **WHERE** to be executed, we already said that we will be running our application in Docker Containers. I'm talking about **HOW** the application will be running ? in deep:

- How users will be playing with QuarkuShop?
- How the users access to the application will be **controlled**?
- Can we handle **unauthorized access**? Do we even know which one to **admit** and which ones to **reject**?
- How the **CPU & Memory** resources consumptions are **measured and tracked**?
- What will happen if the application got **out of resources**?

There are even more questions to be asked about the runtime. These questions reveal two missing layers in QuarkuShop:

- The **Security Layer**: all the **authentication and authorization** engine.
- The **Monitoring Layer**: all the **metrics measurement and tracking** components.

Implementing the Security Layer

Security! One of the most painful topics for developers 😞 But it's probably the most critical subject in any enterprise application. Security has been always a very challenging subject in IT: technology and frameworks keep evolving and obviously, hackers are evolving more and more 😊

For our QuarkuShop, we will use the dedicated Quarkus components and the recommended practices and design choices. We will go in details to implement a typical authentication and authorization engine.



I will be referencing the **authentication and authorization** as **auth²**.

Analyzing the Security requirements and needs

Before starting to write code, we start by making the design, using **UML diagrams** for example. The same for the Security layer; we need to make the design before implementing the code. But which design? The code is there. What we will be designing?

Even the full **QuarkuShop** features have been already implemented, there is a lot to be designed.

I always like to compare building software like building houses. What we already did until now is:

- building the house 🔨 The same as writing the source code.
- validating the conformity of the building with the plans 🔨 The same as writing tests.
- connecting the house to the electricity, water, and sewerage networks 🔨 The same as configuring the access to databases, SonarCloud, etc.
- got the furniture and laid the decoration 🔨 The same as making the CI/CD Pipelines.

Now, the house is ready and its owner wants to have a security system. Then, we will start by checking the windows and doors to locate the possible house access points. It's there where we will put the Locks, and only keys holders will be able to enter, depending on the person, the owner will allocate the keys. For

example, the driver will have only the car garage key; while the gardener will have two keys: the first of the external door and the second for the garden cabin, where we store the tools. While the family members living in the house will have all the keys without exceptions.

We will have also Cameras and Sensors to monitor and audit any access to the house. When we think that there was any illegal access to the house, we can look back to the cameras and check what happened.

This Home Security System deployment process is somehow the same as adding the Security Layer to an application. We will follow the same steps as we did for the house:

1. We will start to analyze and locate all the access points to our application  This process is called "**Attack Surface Analysis**".

Attack Surface Analysis helps you to:

1. identify what functions and what parts of the system you need to review/test for security vulnerabilities
2. identify high-risk areas of code that require defense-in-depth protection - what parts of the system that you need to defend
3. identify when you have changed the attack surface and need to do some kind of threat assessment

— OWASP Cheat Sheet Series,
https://cheatsheetseries.owasp.org/cheatsheets/Attack_Surface_Analysis_Cheat_Sheet.html

2. We will put locks on these access points  These locks are part of the "**Authentication**" process.

Authentication is the process of verifying that an individual, entity, or website is whom it claims to be. Authentication in the context of web applications is commonly performed by submitting a username or ID and one or more items of private information that only a given user should know.

— OWASP Cheat Sheet Series, https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

3. We will define an access control mechanism to be sure that only the allowed person is accessing a given "door"  This process is called **Authorization**.

Authorization is the process where requests to access a particular resource should be granted or denied. It should be noted that authorization is not equivalent to authentication - as these terms and their definitions are frequently confused. Authentication is providing and validating identity. The authorization includes the execution rules that determine which functionality and data the user (or Principal) may access, ensuring the proper allocation of access rights after authentication is successful.

— OWASP Cheat Sheet Series,
https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html

QuarkuShop is a Java Enterprise application that's exposing REST APIs which is the only communication channel with the application users.

The users of **QuarkuShop** can be divided into 3 categories:

- **Visitor** or **Anonymous** who is an unauthenticated customer
- **User** who is an authenticated customer
- **Admin** who is the application super-user

The next step is to define which user category is allowed to access each REST API service. This can be done using the **Authorization Matrix**.

Defining Authorization Matrices for REST APIs

Authorization Matrix for the Cart REST API

Operation	Anonymous	User	Admin
Get all Carts	✗	✓	✓
Get active Carts	✗	✓	✓
Get Carts by Customer ID	✗	✓	✓
Create a new Cart for a given Customer	✗	✓	✓
Get a Cart by ID	✗	✓	✓
Delete a Cart by ID	✗	✓	✓

Authorization Matrix for the Category REST API

Operation	Anonymous	User	Admin
List all Categories	✓	✓	✓
Create new Category	✗	✗	✓
Get Category by ID	✓	✓	✓
Delete Category by ID	✗	✗	✓
Get products by Category ID	✓	✓	✓

Authorization Matrix for the Customer REST API

Operation	Anonymous	User	Admin
Get all Customers	✗	✗	✓
Create a new Customer	✗	✗	✓
Get active Customers	✗	✗	✓
Get inactive Customers	✗	✗	✓

Operation	Anonymous	User	Admin
Get Customer by ID	✗	✗	✓
Delete Customer by ID	✗	✗	✓

Authorization Matrix for the Order REST API

Operation	Anonymous	User	Admin
Get all Orders	✗	✗	✓
Create a new Order	✗	✓	✓
Get orders by Customer ID	✗	✓	✓
Checks if there is an order for a given ID	✗	✓	✓
Get Order by ID	✗	✓	✓
Delete Order by ID	✗	✓	✓

Authorization Matrix for the Order-Item REST API

Operation	Anonymous	User	Admin
Create a new Order-Item	✗	✓	✓
Get Order-Items by Order ID	✗	✓	✓
Get Order-Item by ID	✗	✓	✓
Delete Order-Item by ID	✗	✓	✓

Authorization Matrix for the Payment REST API

Operation	Anonymous	User	Admin
Get all Payments	✗	✗	✓
Create a new Payment	✗	✓	✓
Get Payments for amounts inferior or equal a limit	✗	✓	✓
Get a Payment by ID	✗	✓	✓
Delete a Payment by ID	✗	✗	✓

Authorization Matrix for the Product REST API

Operation	Anonymous	User	Admin
Get all Products	✓	✓	✓
Create a new Product	✗	✗	✓

Operation	Anonymous	User	Admin
Get products by Category ID	✓	✓	✓
Count all Products	✓	✓	✓
Count products by Category ID	✓	✓	✓
Get a Product by ID	✓	✓	✓
Delete a Product by ID	✗	✗	✓

Authorization Matrix for the Review REST API

Operation	Anonymous	User	Admin
Get reviews by Product ID	✓	✓	✓
Create a new review by Product ID	✗	✓	✓
Get a Review by ID	✓	✓	✓
Delete a Review by ID	✗	✗	✓

Implementing the Security Layer

We will be using dedicated Identity storage for handling the **QuarkuShop** users' credentials. We made the choice of **Keycloak** for this purpose.

What is Keycloak?

Keycloak is an open-source **Identity and Access Management solution** aimed at modern applications and services. It makes it easy to secure applications and services with little to no code.



Users authenticate with **Keycloak** rather than individual applications. This means that your applications don't have to deal with **login forms**, **authenticating users**, and **storing users**. Once logged-in to **Keycloak**, users don't have to login again to access a different application. This also applies to logout.

Keycloak provides **Single-Sign Out**, which means users only have to logout once to be logged out of all applications that use **Keycloak**.

Keycloak is based on standard protocols and provides support for **OpenID Connect**, **OAuth 2.0**, and **SAML**.

If role-based authorization doesn't cover your needs, **Keycloak** provides fine-grained authorization services as well. This allows you to manage permissions for all your services from the **Keycloak** admin console and gives you the power to define exactly the policies you need.

We will proceed into implementing this Security Policy in four steps:

1. Preparing and configuring **Keycloak**.
2. Implementing the **auth²** Java Components in QuarkuShop.
3. Updating our **Integration Tests** to support **auth²**.
4. Adding **Keycloak** to our **Production** environment.

Preparing and configuring Keycloak

The first step in Security implementation is to have a **Keycloak** instance. We will be going step by step in creating and configuring **Keycloak**.



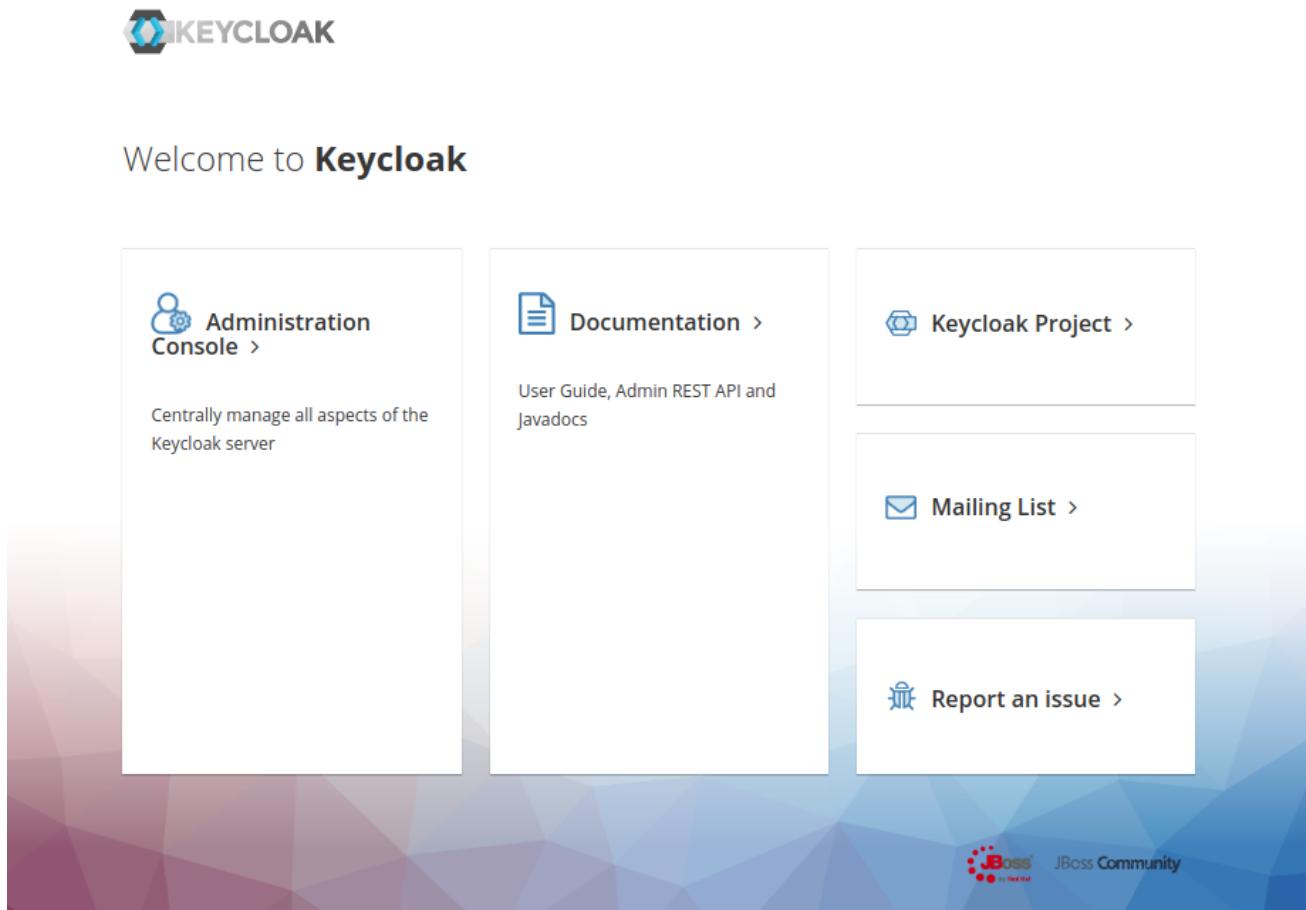
In many tutorials, there are prepared configurations that can be imported to **Keycloak** to get started easily. We will not be doing this here. We will be going step by step to make all the needed configuration by ourselves. This is the best way to learn: **Learning by doing**.

We will start by creating the **Keycloak** instance in **Docker Container**:

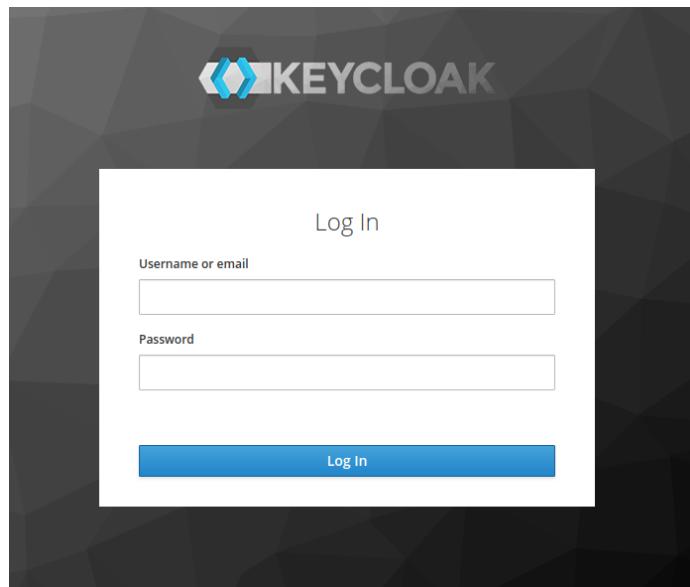
```
docker run -d --name docker-keycloak \
-e KEYCLOAK_USER=admin \
-e KEYCLOAK_PASSWORD=admin \
-e DB_VENDOR=h2 \
-p 9080:8080 \
-p 8443:8443 \
-p 9990:9990 \
jboss/keycloak:11.0.0
```

- ① By default, there is no **admin** user created so you won't be able to **login** to the **admin** console. To create an **admin** account you need to use environment variables to pass in an initial **username** and **password**.
- ② We will be using **H2** as **Keycloak** database
- ③ Listing the exposed ports
- ④ we will be based on **Keycloak 11.0.0**

Then open the <http://localhost:9080> to access the welcome page:



Next, click on **[Administration Console]** to authenticate to the console and use the credentials [**username** and **password** are **admin**]:



Next, we need to create a new **REALM** - start by clicking on [**Add realm**]:

What is a Keycloak Realm?

Realm is the core concept in Keycloak. A realm manages a set of users, credentials, roles, and groups. A user belongs to and logs into a realm. Realms are isolated from one another and can only manage and authenticate the users that they control.

When you boot **Keycloak** for the first time **Keycloak** creates a pre-defined realm for you. This initial realm is the **master** realm. It is the highest level in the hierarchy of realms. Admin accounts in this realm have permissions to view and manage any other realm created on the server instance. When you define your initial admin account, you create an account in the **master** realm. Your initial login to the admin console will also be via the **master** realm.

We will land next on the first step of creating a new realm that we will call **quarkushop-realm** then hit [**Create**]:

Next, we will get the **quarkushop-realm** configuration page:

The screenshot shows the Keycloak admin interface for the 'Quarkushop-realm'. The left sidebar has a dark theme with various management options. The main panel is titled 'Quarkushop-realm' and shows the 'General' tab selected. The 'Name' field is filled with 'quarkushop-realm'. Other fields like 'Display name', 'HTML Display name', 'Frontend URL', and 'Enabled' are empty or off. The 'Endpoints' section shows 'OpenID Endpoint Configuration' and 'SAML 2.0 Identity Provider Metadata'. The 'Save' button is visible at the bottom.

Now, in our **quarkushop-realm**, we need to define the **Roles** that we will be using: **user** and **admin** roles. To do it just go **Roles > Add Role**:

The screenshot shows the Keycloak admin interface for the 'Quarkushop-realm'. The left sidebar has a dark theme with various management options. The main panel is titled 'Roles' and shows the 'Realm Roles' tab selected. It lists two roles: 'offline_access' and 'uma_authorization'. Both are marked as 'Composite' and 'False' in the 'Description' column. There are 'Edit' and 'Delete' buttons for each role.

We will start by creating the **admin** role:

Roles > Add Role

Add Role

* Role Name	<input type="text" value="admin"/>
Description	<input type="text" value="Administrator role"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

And create also the **user** role:

Roles > Add Role

Add Role

* Role Name	<input type="text" value="user"/>
Description	<input type="text" value="User role"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

Now that we created the **Roles**, we need to create the **Users** - Just go to the **Users** menu:



Sometimes, on this screen, the list of users doesn't get loaded when opening the page. If so just hit **[View all users]** to load the list.

Click on **[Add user]** to create the users [Nebrass, Jason and Marie]:

Users > Add user

Add user

ID	<input type="text"/>
Created At	<input type="text"/>
Username *	<input type="text" value="nebrass"/>
Email	<input type="text" value="nebrass@quarkushop.store"/>
First Name	<input type="text" value="Nebrass"/>
Last Name	<input type="text" value="Lamouchi"/>
User Enabled	<input checked="" type="checkbox"/>
Email Verified	<input checked="" type="checkbox"/>
Required User Actions	<input type="text" value="Select an action..."/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

Then after hitting **[Save]**, click on the **[Credentials]** tab - here we will define the user password.

⚠ Don't forget to define the **Temporary** to **OFF** to prevent **Keycloak** from asking you to update the password on the first login. Then, click on **[Set Password]**:

Users > nebrass

Nebrass

- [Details](#)
- [Attributes](#)
- [Credentials](#)
- [Role Mappings](#)
- [Groups](#)
- [Consents](#)
- [Sessions](#)

Manage Credentials

Position	Type	User Label	Data	Actions

Set Password

Password	<input type="text" value="*****"/>	<input type="button" value=""/>
Confirmation	<input type="text" value="*****"/>	<input type="button" value=""/>
Temporary	<input checked="" type="checkbox"/>	<input type="button" value="OFF"/>
<input type="button" value="Set Password"/>		

Credential Reset

Reset Actions	<input type="text" value="Select an action..."/>
Expires In	<input type="text" value="12"/> Hours
Reset Actions Email	<input type="button" value="Send email"/>

Next, go to the **Role Mappings** tab - and add all roles to Nebrass 😊 Yes! I'm the Admin of my App 😊

Same here, create user **Jason** and define its password:

ID	<input type="text"/>
Created At	<input type="text"/>
Username *	jason
Email	jason@quarkushop.store
First Name	jason
Last Name	Bourne
User Enabled	<input checked="" type="checkbox"/> ON
Email Verified	<input checked="" type="checkbox"/> ON
Required User Actions	<input type="text"/> Select an action...
<input type="button"/> Save <input type="button"/> Cancel	

Add the **user** role to **Jason**:

Users > jason

Jason

Role Mappings

Details Attributes Credentials Groups Consents Sessions

Realm Roles	Available Roles	Assigned Roles	Effective Roles
	admin	offline_access uma_authorization user	offline_access uma_authorization user
	Add selected >	« Remove selected	
Client Roles	Select a client...		

Same here, create user **Marie** and define its password:

Users > Add user

Add user

ID	
Created At	
Username *	marie
Email	marie@quarkushop.store
First Name	Marie
Last Name	Hugo
User Enabled	<input checked="" type="checkbox"/> ON
Email Verified	<input checked="" type="checkbox"/> ON
Required User Actions	Select an action...
Save Cancel	

Add the **user** role to **Marie**:

Users > marie

Role Mappings

Details Attributes Credentials Role Mappings Groups Consents Sessions

Realm Roles	Available Roles	Assigned Roles	Effective Roles
	admin	offline_access uma_authorization user	offline_access uma_authorization user
	Add selected >	« Remove selected	
Client Roles	Select a client...		

We finished configuring **Roles** and **Users**! Now, click on menu:[Clients] to list the **Realm Clients**.

KEYCLOAK

Admin

Quarkushop-realm

Clients

Client ID	Enabled	Base URL	Actions		
account	True	http://localhost:9080/auth/realms/quarkushop-realm/account/	Edit	Export	Delete
account-console	True	http://localhost:9080/auth/realms/quarkushop-realm/account/	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	http://localhost:9080/auth/admin/quarkushop-realm/console/	Edit	Export	Delete

What is a Realm Client?

Clients are entities that can request **Keycloak** to authenticate a user. Most often, clients are applications and services that want to use **Keycloak** to secure themselves and provide a **Single Sign-On** solution. **Clients** can also be entities that just want to request **identity information** or an **access token** so that they can securely invoke other services on the network that are secured by **Keycloak**.

Click on **[Create]** to add a new Client:

Clients > Add Client

Add Client

Import	<input type="button" value="Select file"/>
Client ID *	quarkushop
Client Protocol	openid-connect
Root URL	http://localhost:8080
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

- Client ID: **quarkushop**
- Client protocol: **openid-connect**
- Root URL: <http://localhost:8080>

Hit **[Save]** to get the **Client** configuration screen:

Then, it's mandatory to define a Protocol Mapper for our Client.

What is Protocol Mapper?

Protocol mappers perform the transformation on tokens and documents. They can do things like map user data into protocol claims, or just transform any requests going between the client and auth server.

Click on the **[Mappers]** tab to configure a Mapper:

Then click on **[Create]** to add a new Mapper:

Clients > quarkushop > Mappers > Create Protocol Mappers

Create Protocol Mapper

Protocol	openid-connect
Name	realm-roles-mapper
Mapper Type	User Realm Role
Realm Role prefix	
Multivalued	<input checked="" type="checkbox"/>
Token Claim Name	groups
Claim JSON Type	String
Add to ID token	<input checked="" type="checkbox"/>
Add to access token	<input checked="" type="checkbox"/>
Add to userinfo	<input checked="" type="checkbox"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

Why do we define Token Claim Name as groups?

We will use the Protocol Mapper to map the **User Realm Role** (which can be **user** or **admin**) to a property called **groups**, that we will add as **plain String** to the **ID token**, the **access token**, and the **userinfo**.

The choice of **groups** is based on the **MpJwtValidator** Java Class from the **Quarkus SmallRye JWT** library (which is the Quarkus implementation for managing the JWT); where we are defining the **SecurityIdentity Roles** using **jwtPrincipal.getGroups()**.



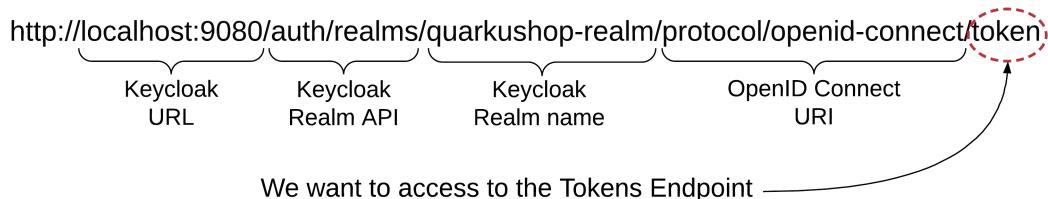
```
JsonWebToken jwtPrincipal = parser.parse(request.getToken().getToken());
uniEmitter.complete(
    QuarkusSecurityIdentity.builder().setPrincipal(jwtPrincipal)
        .addRoles(jwtPrincipal.getGroups())
        .addAttribute(SecurityIdentity.USER_ATTRIBUTE, jwtPrincipal)
        .build()
);
```

This is why we defined our **Mapper** to affect the **User Roles** to the **Groups** property embedded into the **JWT Token** ☺

We can now test if our **Keycloak** instance is running as expected. We will request an **access_token** using **cURL** command:

```
curl -X POST http://localhost:9080/auth/realms/quarkushop-realm/protocol/openid-
connect/token \
-H 'content-type: application/x-www-form-urlencoded' \
-d 'client_id=quarkushop' \
-d 'username=nebrass' \
-d 'password=password' \
-d 'grant_type=password' | jq '.'
```

The URL <http://localhost:9080/auth/realms/quarkushop-realm/protocol/openid-connect/token> is composed of:



You can use the **jq** is a lightweight and flexible command-line JSON processor that I'm using to format the JSON Output.

We will get a JSON Response like this:

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSlldUIiwia2lkIiA6ICJKcmIxWGQzYVBNOS13djhXUmVJekZkRnRJa3Z1WG5uNDd4a0JmT195R19zIn0.eyJleHAiOjE10tC00TEwNTIsImhdCI6MTU5NzQ5Mdc1MiwanRpIjoizmIxZmQx0WMtNWJlMC00YTgwLWExOTUtOTAxZjFkOTI3NDI5IiwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo5MDgwL2F1dGgvcmVhbG1zL3F1YXJrdXnob3AtcmVhbG0iLCJhdWQiOijhY2NvdW50Iiwic3ViIjoimZU10Dc3YWQtMjY3Ny00ODJiLWE5NWYtYTI4ZjdmZGI10Tk5IiwidHlwIjoiQmVhcmVyiwiYXpwIjoiXVhcmt1c2hvcCIsInNlc3Npb25fc3RhGUoijjM2E4ZmU3Mi02MzRmLTRiNmUtYTZkMS03MTkyOGI2YTBlN2YiLCJhY3Ii0iIxIiwiYWxsB3dlZC1vcmlnaW5zIjpbImh0dHA6Ly9sb2NhbGhvc3Q6ODA4MCJdLCJyZWFBsV9hY2Nlc3MiOnsicm9sZXMi0lsib2ZmbGluZV9hY2Nlc3MiLCJhZG1pbisInVtYY9hdXR0b3JpemF0aW9uIiwidXNlcjdfSwicmVzb3VyY2VfYWNjZXNzIjp7ImFjY291bnQiOnsicm9sZXMi0lsibWFuYWd1LLWFjY291bnQiLCJtYW5hZ2UtYWNjb3VudC1saW5rcyIsInZpZXcteHJvZmlsZSJdfx0sInNjb3B1IjoiZW1haWwgHJvZmlsZSIsImVtYWlsX3ZlcmlmaWvkIjp0cnVllCJuYW1lIjoiTmVicmFzcyBMYW1vdWNoaSIsImdyb3VwcyI6WyJvZmZsaW5lX2FjY2VzcyIsImFkbWluIiwidW1hX2F1dGhvcmL6YXRpb24iLCJ1c2VyIl0sInByZWZlcnJlZF91c2VybmtZSI6Im51YnJhc3MiLCJnaXZlbl9uYW1lIjoiTmVicmFzcyIsImZhbWlseV9uYW1lIjoiTGftb3VjaGkiLCJlbWFpbCI6Im51YnJhc3NAcXVhcmt1c2hvcC5zdG9yZSJ9.HZmicWhE9V8g74of9KGcZOVGvwC_oo2zs4-E1BBuV6XSWDUoiFLJVkSUzOV4WFzwvsM7V7_aZRzihZqq6QTtezweyhZIauo3pjmmtbMnq16WUFV-4oJWzk3P_6T5y74sh93aPuQtnw5hSQ4L68RjwQ6HIcaHJFkqrh6fX7uy0ZiHuPnRzhv38uQrD9YMC_z3tApWKTs2TA9igizZrlJCDfTdfiThUDuXEg0mw-pffYx1BAsfL1400c0apGPqirNkSgSrCpuFvikXlRdeu3YnI1JQ6S7Jn-qQI-bdCD5M0_ynaUiJn_p6sZqI6ioSmLGyA__S5J7nj_B0--fdI10lUA",
  "expires_in": 300,
  "refresh_expires_in": 1800,
  "refresh_token": "...",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "c3a8fe72-634f-4b6e-a6d1-71928b6a0e7f",
  "scope": "email profile"
}
```



For a pretty output, I omitted the value of `refresh_token` as it's very long and useless.

Copy the **access_token** value. Then go to jwt.io and paste the JWT **access_token** there to decode it:

The screenshot shows the jwt.io interface. On the left, under 'Encoded', is the long base64-encoded JWT string:

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJKcmlxWGQzYVBN
OS13djhXUmVjekZkRnRJa3Z1W5uNDd4a0JmT195R192In0.eyJleHA10jE1OTc0
OTEwNTIsImlhC16MTU5NzQ5MDc1MiwiRpijoiZmIxZmQxOWMtNWJ1MC00YTgw
LWExOTU0TAXZjFKOTI3NDI5IiwiXNzIjjoiaHR0cDovL2xvY2fsaG9zdDo5MDgw
L2F1dGvcmVhbG1zL3F1YXJrdXNob3AtcmVhbG0iLCJhdWQ1oIjhY2NvdW50Tiwi
c3ViIjoiMzU10dc3YWQtMjY3Ny00ODJlWE5NWYYtT14ZjdmZG10Tk5IiwiidHlw
IjoiQmVhcmVYIiwiYXpwIjoiicXvhcm1c2hvcCIsInNcNpb25fc3RhGU01Jj
M2E4Zm03M102MzRmLTr1NmUtYtZkMS83MTkyOGI2TB1N2Y1lCJh3Ii10iixIiwi
YWxsbsd1ZC1vcmlnaW5zIjpbImh0dHA6Ly9sb2Nhbgv3Q60DA4MCJdLCJyZWFs
bV9hY2Nlc3Mi0nsicm9sZXMiolsibZmbGluzV9hY2Nl3MiLCJhZG1pbisInVt
VV9hdXR0b3JpemF0aW9uIiwydfSwicmVzb3ImJp7ImFj
Y291bnQ10nsicm9sZXMi0lsbWFuYWdlWFjY291bnQ1lCJtYW5hZ3UyYWNjb3Vu
dc1saW5rcyIsInZpZctchJyV2mIsZSjdfX0sInNjbj3B1IjoiZWIhaWwgchJyVzmls
ZSiIsImVtYWlsX3ZlcmlmaWVkJip0cnVlLCJuYW1lIjoiTmVicmFzcycBMYW1vdWNo
aSiSmidyb3VwcyI6WjVzMsasW51X2fjY2VzcyIsImFkbwluIiwidW1hX2F1dghv
cml6YXRp24iLCJ1c2VyIl0sInByZWLcnJlZF91c2VybmfZSI6Im5lYnhc3Mi
LCJnaXlbl9uYW1lIjoiTmVicmFzcycIsimZhbWiseV9uYW1lIjoiTGftb3VjaGki
LCJ1bWFpbCI6Im5lYnhc3NAcXvhcm1c2hvcC5zG9yZSJ9.HZmicWhE9V8g4o
f9KGcZOVGwC_o02zs4-
E1BBuV6XSWDUo1FLJVksUz0V4WFzwsM7V7_aZRzihZqq6QTtezweyhZIauo3pj
mtbMnq16UFV-
4oJWzk3P_6T5y74sh93aPuQtnw5hSQ4L68RjwQ6HICAjHFkqrh6fx7uy0Z1huPnR
zhv38uQrD9YMC_z3tApWKTS2TA9igizZrlJCdfTdfiThUDuXEG0mw-
pffYx1BASfl1400c0ap0Pq1rNk5gSrCpuFv1kXlRdeu3YnIIJQ6S7Jn-qQI-
bdCD5M0_ynaUiJn_p6sZqI6ioSmLGyA__S5J7nj_B0--fdI101UA
```

On the right, under 'Decoded', is the JSON payload:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "JriqXd3aPM9-wv8WReIzFdFtIkvuXnn47xbfN_yG_s"
}

PAYLOAD: DATA

{
  "exp": 1597491052,
  "iat": 1597498752,
  "jti": "f1fd19c5be8-a480-a195-901f1d927429",
  "iss": "http://localhost:9080/auth/realm/quarkushop-realm",
  "aud": "account",
  "sub": "355877ad-2677-482b-a95f-a28f7fdb5999",
  "typ": "Bearer",
  "azp": "quarkushop",
  "session_state": "c3a8fe72-634f-4b6e-a6d1-71928b6a0e7f",
  "acr": "1",
  "allowed_origins": [
    "http://localhost:8080"
  ],
  "realm_access": {
    "roles": [
      "offline_access",
      "admin",
      "uma_authorization",
      "user"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    },
    "scope": "email profile",
    "email_verified": true,
    "name": "Nebrass Lamouchi",
    "groups": [
      "offline_access",
      "admin",
      "uma_authorization",
      "user"
    ],
    "preferred_username": "nebrass",
    "given_name": "Nebrass",
    "family_name": "Lamouchi",
    "email": "nebrass@quarkushop.store"
  }
}
```

You notice that there is a **JSON attribute** called **groups** that holds the **Roles** array that we assigned to the user **Nebrass** in **Keycloak**.

Good! ☺ our **Keycloak** is running and working like expected. We will start now the implementing Security on the Java side.

Implementing the auth² Java Components in QuarkuShop

Java Configuration side

The first step is to add the **Quarkus SmallRye JWT** dependency to QuarkuShop:

```
./mvnw quarkus:add-extension -Dextensions="io.quarkus:quarkus-smallrye-jwt"
```

Next, we will add the Security configuration to the `application.properties`:

```

1 ### Security
2 quarkus.http.cors=true ①
3 # MP-JWT Config
4 mp.jwt.verify.issuer=http://localhost:9080/auth/realm/quarkushop-realm ②
5 mp.jwt.verify.publickey.location=http://localhost:9080/auth/realm/quarkushop-
    realm/protocol/openid-connect/certs ③
6 # Keycloak Configuration
7 keycloak.credentials.client-id=quarkushop ④

```

① Enables the HTTP CORS Filter.

What is CORS ?

Cross-Origin Resource Sharing (aka **CORS**) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

- ② Config property specifies the value of the `iss` (issuer) claim of the **JWT Token** that the server will accept as valid. We already got this value in the `iss` field of the decrypted **JWT Token**.
- ③ Config property allows for an external or internal location of **Public Key** to be specified. The value may be a relative path or a URL. When using **Keycloak** as Identity manager, the value will be the `mp.jwt.verify.issuer + /protocol/openid-connect/certs`
- ④ A custom property with a key I called `keycloak.credentials.client-id` which holds the value `quarkushop` which is our Realm Client ID.

Java Source Code side

In the Java code, we need to add the needed protection to the **REST APIs** based on the **Authorization Matrix** that we made for each service.

We will begin by the **Carts REST API**, where all operations are allowed for **Users** and **Admins**. Only the **authenticated subjects** are allowed on the **Carts REST API**. To satisfy this requirement, we have an `@Authenticated` annotation, from the `io.quarkus.security` package, that will grant access only to **authenticated subjects**.



To make a difference between an *application user* and the *Role User*, I will make reference to an application user as **Subject**. The term **Subject** came in the old good **Java Authentication and Authorization Service (JAAS)** to represent the caller that is making a request to access a resource. A **subject** may be any *entity*, such as a *person* or *service*.

As all the operations in the **CartResource** require an authenticated subject, we will then annotote the class **CartResource** by **@Authenticated**, so it will be applied to all the operations:

`com.targa.labs.quarkushop.web.CartResource`

```
@Authenticated
@Path("/carts")
@Produces(MediaType.APPLICATION_JSON)
@Tags(value = @Tag(name = "cart", description = "All the cart methods"))
public class CartResource {

    ...
}
```

The next **REST API** is the **Category API**, where only the Admin can create and delete categories, while all the other operations are allowed everyone (**Admin**, **User**, and **Anonymous**). To grant access only for a specific role, we have the annotation **@RolesAllowed** from the `javax.annotation.security` package.

JavaDoc of javax.annotation.security.RolesAllowed

@RolesAllowed: Specifies the list of security roles permitted to access the method(s) in an application.

The value of the **@RolesAllowed** annotation is a list of security role names. This annotation can be specified on a class or on method(s):



- Specifying it at a class level means that it applies to all the operations in the class.
- Specifying it on a method means that it applies to that method only.
- If applied at both the class and operations level, the method value overrides the class value if the two conflict.

Based on the **JavaDoc**, we will pass the desired role as value of the annotation We will apply the **@RolesAllowed("admin")** annotation on the operations where we create and delete categories:

```

@Path("/categories")
@Produces(MediaType.APPLICATION_JSON)
@Tags(value = @Tag(name = "category", description = "All the category methods"))
public class CategoryResource {

    ...

    @RolesAllowed("admin")
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public CategoryDto create(CategoryDto categoryDto) {
        return this.categoryService.create(categoryDto);
    }

    @RolesAllowed("admin")
    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.categoryService.delete(id);
    }
}

```

For the **Customer REST API**, only the Admin is allowed to access all the services Annotating the **CustomerResource** class by **@RolesAllowed("admin")** will apply this policy.

The next one is for the **Order REST API**, where the authenticated subjects can access all the operations, except the method **findAll()** which will be allowed only for **admin**. So here we will combine the two annotations **@Authenticated** in the *Class Level* and **@RolesAllowed** on the *Method* requiring specific **Role**:

```

@Authenticated
@Path("/orders")
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = "order", description = "All the order methods")
public class OrderResource {

    ...

    @RolesAllowed("admin")
    @GET
    public List<OrderDto> findAll() {
        return this.orderService.findAll();
    }

    ...
}

```



While we are using `@RolesAllowed` in the method, we are overriding locally on the annotated method the policy applied by the class level `@Authenticated`.

After `OrderResource`, we will deal with the **OrderItem REST API**, where only authenticated subjects are allowed to access all the services ↗ Annotating the `OrderItemResource` class by `@Authenticated` will apply this policy.

The `PaymentResource` is granting access only for authenticated subjects, except deleting and listing all payments are granted only for `admin`:

```

@Authenticated
@Path("/payments")
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = "payment", description = "All the payment methods")
public class PaymentResource {

    ...

    @RolesAllowed("admin")
    @GET
    public List<PaymentDto> findAll() {
        return this.paymentService.findAll();
    }

    @RolesAllowed("admin")
    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.paymentService.delete(id);
    }

    ...
}

```

For the **Product REST API**, all operations are allowed for everyone, except creating and deleting products are allowed only for **admin**:

```
@Path("/products")
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = "product", description = "All the product methods")
public class ProductResource {

    ...

    @RolesAllowed("admin")
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public ProductDto create(ProductDto productDto) {
        return this.productService.create(productDto);
    }

    @RolesAllowed("admin")
    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.productService.delete(id);
    }

    ...
}
```

Finally, the last REST API is **ReviewResource**:

```
@Path("/reviews")
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = "review", description = "All the review methods")
public class ReviewResource {

    . . .

    @Authenticated
    @POST
    @Path("/product/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public ReviewDto create(ReviewDto reviewDto, @PathParam("id") Long id) {
        return this.reviewService.create(reviewDto, id);
    }

    @RolesAllowed("admin")
    @DELETE
    @Path("/{id}")
    public void delete(@PathParam("id") Long id) {
        this.reviewService.delete(id);
    }
}
```

Great! We applied the first level authorization layer! 😊

Don't be confident 😊 This implementation leaks more levels of authorization. For example, users are authorized to delete **Orders** or **OrderItems**, but we don't have any verification that the authenticated subject **Peter** is deleting his own **Orders** or **OrderItems** and not those of the user **Saul**. Until now, we know how to define access rules based on the role, but how can we gather more information about the authenticated subject.

For this purpose, we will create a new **REST API** that we will call **UserResource**. When invoked, **UserResource** will return the current authenticated subject information as a response.

The **UserResource** will look like:

```

@Path("/user")
@Authenticated
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = " user", description = "All the user methods")
public class UserResource {

    @Inject
    JsonWebToken jwt;                                ①

    @GET
    @Path("/current/info")
    public JsonWebToken getCurrentUserInfo() {        ②
        return jwt;
    }

    @GET
    @Path("/current/info/claims")
    public Map<String, Object> getCurrentUserInfoClaims() { ③
        return jwt.getClaimNames()
            .stream()
            .map(name -> Map.entry(name, jwt.getClaim(name)))
            .collect(Collectors.toMap(
                entry -> entry.getKey(),
                entry -> entry.getValue()))
    };
}
}

```

- ① We inject a **JsonWebToken**: which is an implementation of the specification that defines the use of JWT as bearer token. This injection will make an instance of the JWT Token used in the incoming request.
- ② The **JsonWebToken** will return the injected JWT Token.
- ③ The **getCurrentUserInfoClaims()** method will return the list of the available claims and their respective values embedded in the JWT token. The claims will be extracted from the JWT Token instance injected in the **UserResource**.

Let's test our new REST API. Start by getting a new **access_token**:

```
export access_token=$(curl -X POST http://localhost:9080/auth/realms/quarkushop-realm/protocol/openid-connect/token \
-H 'content-type: application/x-www-form-urlencoded' \
-d 'client_id=quarkushop' \
-d 'username=nebrass' \
-d 'password=password' \
-d 'grant_type=password' | jq --raw-output '.access_token')
```

Then, let's invoke the **/user/current/info** REST API:

```
curl -X GET -H "Authorization: Bearer $access_token"
http://localhost:8080/api/user/current/info | jq .'
```

Then we will get as response the JWT Token:

```
{
  "audience": [
    "account"
  ],
  "claimNames": [
    "sub", "resource_access", "email_verified", "allowed-origins", "raw_token", "iss",
    "groups", "typ", "preferred_username", "given_name", "aud", "acr", "realm_access",
    "azp", "scope", "name", "exp", "session_state", "iat", "family_name", "jti", "email"
  ],
  "expirationTime": 1597530481,
  "groups": ["offline_access", "admin", "uma_authorization", "user"],
  "issuedAtTime": 1597530181,
  "issuer": "http://localhost:9080/auth/realms/quarkushop-realm",
  "name": "nebrass",
  "rawToken": "eyJhbGci...L5A",
  "subject": "355877ad-2677-482b-a95f-a28f7fdb5999",
  "tokenID": "7416ee6e-e74c-45ae-bf85-8889744eaacf"
}
```

Good, we have the available claims, let's get their respective values:

```
{
  "sub": "355877ad-2677-482b-a95f-a28f7fdb5999",
  "resource_access": {
    "account": {
      "roles": ["manage-account", "manage-account-links", "view-profile"]
    }
  },
  "email_verified": true,
  "allowed-origins": ["http://localhost:8080"],
  "raw_token": "eyJhbGci...L5A",
  "iss": "http://localhost:9080/auth/realms/quarkushop-realm",
  "groups": ["offline_access", "admin", "uma_authorization", "user"],
  "typ": "Bearer",
  "preferred_username": "nebrass",
  "given_name": "Nebrass",
  "aud": ["account"],
  "acr": "1",
  "realm_access": {
    "roles": ["offline_access", "admin", "uma_authorization", "user"]
  },
  "azp": "quarkushop",
  "scope": "email profile",
  "name": "Nebrass Lamouchi",
  "exp": 1597530481,
  "session_state": "68069099-f534-434f-8d08-d8d75b8ff1c6",
  "iat": 1597530181,
  "family_name": "Lamouchi",
  "jti": "7416ee6e-e74c-45ae-bf85-8889744eaacf",
  "email": "nebrass@quarkushop.store"
}
```

Wonderful! 😊 Now we know how to access the Security details that you can use to identify exactly the authenticated subject. By the way, we can have the same information about the JWT Token. We can get the same JWT Token in an alternative way:

```
@GET()
@Path("/current/info-alternative")
public Principal getCurrentUserInfoAlternative(@Context SecurityContext ctx) {
    return ctx.getUserPrincipal();
}
```

The `@Context SecurityContext ctx` passed as the method parameter is used to inject the `SecurityContext` to the current context.



SecurityContext: An injectable interface that provides access to security-related information like the **Principal**.



Principal: This interface represents the abstract notion of a principal, which can be used to represent any entity, such as an individual, a corporation, and a login id. In our case, the Principal is the JWT Token.



I will not dig into the full implementation of the authorization layer. I will stop here, otherwise, I will spend two more chapters just in this content.

The last step in the security part is to add a **REST API** that returns an `access_token` for a given `username` and `password`. This is can be useful especially from the **Swagger UI**. Speaking about **Swagger UI**, we need to make it aware of our **Security Layer**.

We will start by creating the `TokenService`, which will be the **REST Client** that will be making the request of the `access_token` from **Keycloak**.

As we are using **Java 11**, we will enjoy one of its great new features: the brand new **HTTP Client**.

As we did before using the cURL command, we need to use the **Java 11** HTTP Client to request an `access_token`.

```

@RequestScoped
public class TokenService {

    @ConfigProperty(name = "mp.jwt.verify.issuer", defaultValue = "undefined")
    Provider<String> jwtIssuerUrlProvider; ①

    @ConfigProperty(name = "keycloak.credentials.client-id", defaultValue = "undefined")
    Provider<String> clientIdProvider; ②

    public String getAccessToken(String userName, String password)
        throws IOException, InterruptedException {

        String keycloakTokenEndpoint =
            jwtIssuerUrlProvider.get() + "/protocol/openid-connect/token";

        String requestBody = "username=" + userName + "&password=" + password +
            "&grant_type=password&client_id=" + clientIdProvider.get();

        if (clientSecret != null) {
            requestBody += "&client_secret=" + clientSecret;
        }

        HttpClient client = HttpClient.newBuilder().build();
    }
}

```

```

HttpRequest request = HttpRequest.newBuilder()
    .POST(BodyPublishers.ofString(requestBody))
    .uri(URI.create(keycloakTokenEndpoint))
    .header("Content-Type", "application/x-www-form-urlencoded")
    .build();

HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers
.ofString());

String accessToken = "";

if (response.statusCode() == 200) {
    ObjectMapper mapper = new ObjectMapper();
    try {
        accessToken = mapper.readTree(response.body()).get("access_token"
).textValue();
    } catch (IOException e) {
        e.printStackTrace();
    }
} else {
    throw new UnauthorizedException();
}

return accessToken;
}
}

```

- ① Gets the `mp.jwt.verify.issuer` property value to build the **Keycloak** Tokens Endpoint URL.
- ② Gets the `keycloak.credentials.client-id` property value, which is needed for requesting the `access_token`.



We are using getting the property value as `Provider<String>` and not `String` to avoid failing the build of the native image.

Next, we will add the `getAccessToken()` method:

```
@Path("/user")
@Authenticated
@Produces(MediaType.APPLICATION_JSON)
@Tag(name = " user", description = "All the user methods")
public class UserResource {

    @Inject JsonWebToken jwt;

    @POST
    @PermitAll
    @Path("/access-token")
    @Produces(MediaType.TEXT_PLAIN)
    public String getAccessToken(@QueryParam("username") String username,
                                @QueryParam("password") String password)
        throws IOException, InterruptedException {
        return tokenService.getAccessToken(username, password);
    }
    ...
}
```

Excellent! 😊 We can test the new method from the **Swagger UI**:

Yippi! 😊 Now, we need to find a way to make **Swagger UI** apply the `access_token` to request all the REST APIs.

I thought this will be a long trip, but it was a very easy task - Just 10 lines of code are enough to make the party 😊

```

@SecurityScheme(
    securitySchemeName = "jwt",           ①
    description = "JWT authentication with bearer token",
    type = SecuritySchemeType.HTTP,      ①
    in = SecuritySchemeIn.HEADER,        ①
    scheme = "bearer",                  ①
    bearerFormat = "Bearer [token]")     ①
@OpenAPIDefinition(
    info = @Info(                      ②
        title = "QuarkuShop API",
        description = "Sample application for the book 'Playing with Java
Microservices with Quarkus and Kubernetes'",
        contact = @Contact(name = "Nebrass Lamouchi", email =
"lnibrass@gmail.com", url = "https://blog.nebrass.fr"),
        version = "1.0.0-SNAPSHOT"
    ),
    security = @SecurityRequirement(name = "JWT") ③
)
public class OpenApiConfig extends Application {
}

```

① The `@SecurityScheme` defines a security scheme that can be used by the **OpenAPI operations**.

② We are adding a description for the **OpenAPI Definition**.

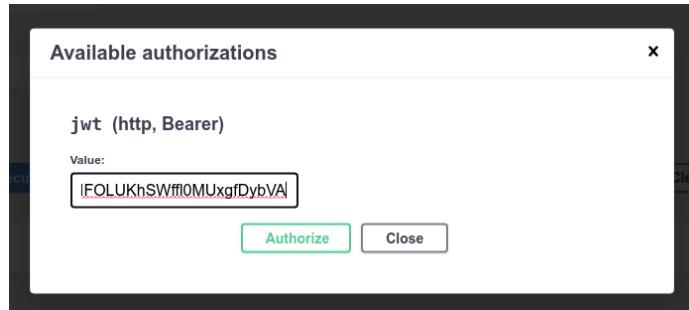
③ We are linking the created security schema `jwt` to the **OpenAPI Definition**.

Now, we will check again **Swagger UI**:

The screenshot shows the QuarkuShop API Swagger UI interface. At the top, it displays the title "QuarkuShop API 1.0.0-SNAPSHOT OAS3" and a note about the sample application being for the book "Playing with Java Microservice with Quarkus and Kubernetes". Below this, there's a link to "Nebrass Lamouchi - Website" and an option to "Send email to Nebrass Lamouchi". On the right side, there's a green "Authorize" button with a lock icon. The main content area is divided into sections: "user" and "cart". The "user" section contains four operations: "POST /api/user/access-token" (green background), "GET /api/user/current/info" (blue background), "GET /api/user/current/info-alternative" (blue background), and "GET /api/user/current/info/claims" (blue background). The "cart" section contains two operations: "GET /api/carts" (blue background) and "GET /api/carts/active" (blue background). Each operation row has a lock icon on the right.

You can notice that there is the lock icon .

Use the `getAccessToken()` operation to ken an `access_token` then click on [**Authorize**] to pass the generated `access_token` to the **SecurityScheme** and click on [**Authorize**]:



Now, when you click on any operation, **Swagger UI** will include the `access_token` as bearer to every request.

Excellent! Now, we have the **Keycloak** and the **Security Java Components**. Now we need to refactor our tests to be aware of the **Security Layer**.

Updating our Integration Tests to support auth²

For the tests, we need to provision dynamically the **Keycloak** instance, as we did for the Database using the **Testcontainers** library.

We will use the same library to provision Keycloak. We will not use a plain Docker container as we did with PostgreSQL, because there, Flyway is populating the database for us, while for Keycloak, we don't have any built-in mechanism that will create the **Keycloak** Realms for us. The only possible solution is to use a Docker Compose file that will import a sample **Keycloak** realms file. Fortunately, **Testcontainers** has great support to provision **Docker-Compose** files.

To provision the containers from a Docker Compose file using **TestContainers**:

```
public static DockerComposeContainer KEYCLOAK = new DockerComposeContainer(
    new File("src/main/docker/keycloak-test.yml"))
    .withExposedService("keycloak_1", 9080,
        Wait.forListeningPort().withStartupTimeout(Duration.ofSeconds(30)));
```

We will use a sample **Keycloak** Realm that contains:

- Two users:
 - Admin has username: **admin** - password: **test** - role: **admin**
 - Test has username: **user** - password: **test** - role: **user**
- The **Keycloak Client** has an **client-id=quarkus-client**: and `secret=mysecret

Our **QuarkusTestResourceLifecycleManager** will be communicating with the provisioned **Keycloak** instance, using the **TokenService** that we created, and will use the sample Realm credentials to get **access_tokens** that we will use in the integration tests.

We will be getting two **access_tokens**: one for the **admin** role and the second for the **user** role. We will be storing them, along with the **mp.jwt.verify.publickey.location** and **mp.jwt.verify.issuer**, as system properties in the current test scope.

Our custom **QuarkusTestResourceLifecycleManager** will look like:

```
public class KeycloakRealmResource implements QuarkusTestResourceLifecycleManager {

    @ClassRule
    public static DockerComposeContainer KEYCLOAK = new DockerComposeContainer(
        new File("src/main/docker/keycloak-test.yml"))
        .withExposedService("keycloak_1",
            9080,
            Wait.forListeningPort().withStartupTimeout(Duration.ofSeconds(30)));

    @Override
    public Map<String, String> start() {
        KEYCLOAK.start();

        String jwtIssuerUrl = String.format(
            "http://%s:%s/auth/realms/quarkus-realm",
            KEYCLOAK.getServiceHost("keycloak_1", 9080),
            KEYCLOAK.getServicePort("keycloak_1", 9080)
        );

        TokenService tokenService = new TokenService();
        Map<String, String> config = new HashMap<>();
```

```

try {
    String adminAccessToken = tokenService.getAccessToken(jwtIssuerUrl,
        "admin", "test", "quarkus-client", "mysecret"
    );

    String testAccessToken = tokenService.getAccessToken(jwtIssuerUrl,
        "test", "test", "quarkus-client", "mysecret"
    );

    config.put("quarkus-admin-access-token", adminAccessToken);
    config.put("quarkus-test-access-token", testAccessToken);

} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}

config.put("mp.jwt.verify.publickey.location", jwtIssuerUrl + "/protocol/openid-
connect/certs");
config.put("mp.jwt.verify.issuer", jwtIssuerUrl);

return config;
}

@Override
public void stop() {
    KEYCLOAK.stop();
}
}

```

Then, we will use it with the `@QuarkusTestResource(KeycloakRealmResource.class)` annotation.

In the `@BeforeAll` method, we will get the access_tokens from the Systems Properties to make them ready to be used in tests. A typical test skull will look like:

```
@QuarkusTest
@QuarkusTestResource(TestContainerResource.class)
@QuarkusTestResource(KeycloakRealmResource.class)
class CategoryResourceTest {

    static String ADMIN_BEARER_TOKEN;
    static String USER_BEARER_TOKEN;

    @BeforeAll
    static void init() {
        ADMIN_BEARER_TOKEN = System.getProperty("quarkus-admin-access-token");
        USER_BEARER_TOKEN = System.getProperty("quarkus-test-access-token");
    }

    ...
}
```

To use these tokens in a Test:

```
@Test
void testFindAllWithAdminRole() {
    given().when()
        .header(HttpHeaders.AUTHORIZATION, "Bearer " + ADMIN_BEARER_TOKEN)
        .get("/carts")
        .then()
        .statusCode(OK.getStatusCode())
        .body("size()", greaterThan(0));
}
```

We need to test and verify the authorization rules in tests - for example to verify that a given profile is not **Unauthorized**:

```
@Test
void testFindAll() {
    get("/carts").then()
        .statusCode(UNAUTHORIZED.getStatusCode());
}
```

To verify that for a given request, the subject is not allowed on a REST API:

```
@Test
void testDeleteWithUserRole() {
    given().when()
        .header(HttpHeaders.AUTHORIZATION, "Bearer " + USER_BEARER_TOKEN)
        .delete("/products/1")
        .then()
        .statusCode(FORBIDDEN.getStatusCode());
}
```

Good! I implemented all the tests, you will find them in my [GitHub Repository](#)

Adding Keycloak to our Production environment

The last step is to add the production **Keycloak** entries to our **Docker Compose** in the Production VM. We need also to add our **Production Realms**.

We can export the Realm from our *local Keycloak* instance, in very few steps:

To export the Realm from the local **Keycloak** container (called `docker-keycloak`):

```
$ docker exec -it docker-keycloak bash

bash-4.4$ cd /opt/jboss/keycloak/bin/

bash-4.4$ mkdir backup

bash-4.4$ ./standalone.sh -Djboss.socket.binding.port-offset=1000 \
    -Dkeycloak.migration.realmName=quarkushop-realm \
    -Dkeycloak.migration.action=export \
    -Dkeycloak.migration.provider=dir \
    -Dkeycloak.migration.dir=./backup/
```

To copy the stored Realm from the `docker-keycloak` Container to a local directory:

```
$ mkdir ~/keycloak-realms

$ docker cp docker-keycloak:/opt/jboss/keycloak/bin/backup ~/keycloak-realms
```

We will get two Keycloak Realm files `quarkushop-realm-realm.json` and `quarkushop-realm-users-0.json` in the `~/keycloak-realms`.

We need to edit the `quarkushop-realm-realm.json` file - and change the `sslRequired` from `external` to `none`:

```
{
  ...
  "sslRequired": "none",
  ...
}
```



The `"sslRequired": "none"` property will disable the required SSL certificate for any request.

Then, we need to copy the two files `quarkushop-realm-realm.json` and `quarkushop-realm-users-0.json` to the `/opt/realms` directory, in the *Azure VM instance*, which is our *Production environment*.

Here we are in the *Azure VM instance* ⑩ we will finish the last step: adding the **Keycloak** service to the `/opt/docker-compose.yml` file:

```

1 version: '3'
2 services:
3   quarkushop:
4     image: nebrass/quarkushop-monolithic-application:latest
5     environment:
6       - QUARKUS_DATASOURCE_JDBC_URL=jdbc:postgresql://postgresql-db:5432/demo
7       -
8         MP_JWT_VERIFY_PUBLICKEY_LOCATION=http://51.103.50.23:9080/auth/realms/quarkushop-
9           realm/protocol/openid-connect/certs  ①
10        - MP_JWT_VERIFY_ISSUER=http://51.103.50.23:9080/auth/realms/quarkushop-realm
11      ports:
12        - 8080:8080
13      postgresql-db:
14        image: postgres:13
15        volumes:
16          - /opt/postgres-volume:/var/lib/postgresql/data
17        environment:
18          - POSTGRES_USER=developer
19          - POSTGRES_PASSWORD=p4SSW0rd
20          - POSTGRES_DB=demo
21          - POSTGRES_HOST_AUTH_METHOD=trust
22        ports:
23          - 5432:5432
24      keycloak: ②
25        image: jboss/keycloak:latest
26        command:
27          [
28            '-b', '0.0.0.0',
29
```

```

27
28     '-Dkeycloak.migration.action=import',
29     '-Dkeycloak.migration.provider=dir',
30     '-Dkeycloak.migration.dir=/opt/jboss/keycloak/realm',
31     '-Dkeycloak.migration.strategy=OVERWRITE_EXISTING',
32     '-Djboss.socket.binding.port-offset=1000',
33     '-Dkeycloak.profile.feature.upload_scripts=enabled',
34 ]
35
36
37 volumes:
38   - ./realms:/opt/jboss/keycloak/realm
39 environment:
40   - KEYCLOAK_USER=admin          ③
41   - KEYCLOAK_PASSWORD=admin      ③
42   - DB_VENDOR=POSTGRES          ④
43   - DB_ADDR=postgresql-db       ⑤
44   - DB_DATABASE=demo            ⑥
45   - DB_USER=developer           ⑥
46   - DB_SCHEMA=public             ⑥
47   - DB_PASSWORD=p4SSW0rd         ⑥
48 ports:
49   - 9080:9080
50 depends_on:
51   - postgresql-db               ⑦

```

- ① Overrides the `MP_JWT_VERIFY_PUBLICKEY_LOCATION` and `MP_JWT_VERIFY_ISSUER` properties to make the application points on the `Azure VM Instance` instead of the `localhost`.
- ② Adds the **Keycloak** Docker Service.
- ③ Defines the **Keycloak** cluster `username` and `password`.
- ④ Defines the **Keycloak Database** vendor as **PostgreSQL**, as we have already a **PostgreSQL DB** instance in our services.
- ⑤ Defines the **DB Host** as the `postgresql-db`, which will be resolved dynamically by Docker with the service IP.
- ⑥ Defines the **Keycloak Database** credentials the same as the **PostgreSQL** Credentials.
- ⑦ Express dependency between the **Keycloak** and the **PostgreSQL** services.



As you can notice, all the credentials are listed as plain text in the `docker-compose.yml` file.

We can make protect them, using Docker Secrets. Each password will be stored in a Docker Secret.

```
docker service create --name POSTGRES_USER --secret developer
docker service create --name POSTGRES_PASSWORD --secret p4SSW0rd
docker service create --name POSTGRES_DB --secret demo
docker service create --name KEYCLOAK_USER --secret admin
docker service create --name KEYCLOAK_PASSWORD --secret admin
```



Unfortunately, this feature is available only for **Docker Swarm** clusters.

To protect the credentials, we will store them in an `~/.env` file on the Azure VM Instance:

```
POSTGRES_USER=developer
POSTGRES_PASSWORD=p4SSW0rd
POSTGRES_DB=demo
KEYCLOAK_USER=admin
KEYCLOAK_PASSWORD=admin
```

Then, we will need to change the `docker-compose.yml` to use the `~/.env` elements:

```
version: '3'
services:
  quarkushop:
    image: nebrass/quarkushop-monolithic-application:latest
    environment:
      - QUARKUS_DATASOURCE_JDBC_URL=jdbc:postgresql://postgresql-db:5432/${POSTGRES_DB}
      - MP_JWT_VERIFY_PUBLICKEY_LOCATION=http://51.103.50.23:9080/auth/realm/quarkushop-realm/protocol/openid-connect/certs
      - MP_JWT_VERIFY_ISSUER=http://51.103.50.23:9080/auth/realm/quarkushop-realm
    ports:
      - 8080:8080
  postgresql-db:
    image: postgres:13
    volumes:
      - /opt/postgres-volume:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
      - POSTGRES_HOST_AUTH_METHOD=trust
    ports:
      - 5432:5432
  keycloak:
    image: jboss/keycloak:latest
```

```

command:
[
  '-b',
  '0.0.0.0',
  '-Dkeycloak.migration.action=import',
  '-Dkeycloak.migration.provider=dir',
  '-Dkeycloak.migration.dir=/opt/jboss/keycloak/realm',
  '-Dkeycloak.migration.strategy=OVERWRITE_EXISTING',
  '-Djboss.socket.binding.port-offset=1000',
  '-Dkeycloak.profile.feature.upload_scripts=enabled',
]
volumes:
- ./realms:/opt/jboss/keycloak/realm
environment:
- KEYCLOAK_USER=${KEYCLOAK_USER}
- KEYCLOAK_PASSWORD=${KEYCLOAK_PASSWORD}
- DB_VENDOR=POSTGRES
- DB_ADDR=postgresql-db
- DB_DATABASE=${POSTGRES_DB}
- DB_USER=${POSTGRES_USER}
- DB_SCHEMA=public
- DB_PASSWORD=${POSTGRES_PASSWORD}
ports:
- 9080:9080
depends_on:
- postgresql-db

```

Good! The **Docker Compose** services are ready to go! 😊

We need just to add an exception in the **Azure VM instance Networking Rules** for the **Keycloak** port. In the **Azure VM instance**, go to the **Networking** section and add the exception for port **9080**:

Priority	Name	Port	Protocol
300	SSH	22	TCP
310	Port_8080	8080	Any
65000	AllowVnetInBound	Any	Any
65001	AllowAzureLoadBalancerInBound	Any	Any
65500	DenyAllInBound	Any	Any

Source	<input type="radio"/> Any
Source port ranges	*
Destination	<input type="radio"/> Any
Destination port ranges	9080
Protocol	<input type="radio"/> Any <input checked="" type="radio"/> TCP <input type="radio"/> UDP <input type="radio"/> ICMP
Action	<input type="radio"/> Allow <input checked="" type="radio"/> Deny
Priority	330
Name	Port_9080
Description	

Excellent! 🎉 Our production environment has all the needed elements to deploy a new version of the **QuarkuShop** container without risking the **PostgreSQL DB** or the **Keycloak** cluster.

Now, go to the production **Swagger UI** and enjoy QuarkuShop: the greatest online store! 😊

The screenshot shows the Swagger UI interface for the QuarkuShop API. The top navigation bar includes tabs for 'Swagger UI' and 'Explore', and a status bar indicating 'Not secure' and the URL '51.103.50.23:8080/api/swagger-ui/#/%20user/get_api_user_current_info_claims'. The main content area displays the 'QuarkuShop API' documentation, version 1.0.0-SNAPSHOT, OAS3. It features a 'User' section with methods like 'POST /api/user/access-token'. The 'Parameters' tab for this endpoint shows fields for 'password' (string, query) and 'username' (string, query). Below the parameters are 'Execute' and 'Clear' buttons. The 'Responses' section contains a 'Curl' command and a 'Request URL'. The 'Server response' section shows a detailed JSON response body, which is partially redacted. A 'Download' button is located at the bottom right of the response area.

Good! 😊 We will move now to the next anti-disasters layer: the 🔍 **MONITORING** 😃

Implementing the Monitoring Layer

Security is not the only critical extra layer in an application. Metrics monitoring is also a very important layer that can prevent disasters. Let's imagine this situation: we have a super-secure powerful application deployed on a server in the cloud, if we don't measure up regularly the application metrics, our application may get out of resources without even that we know.

Application monitoring is not just a mechanism of getting metrics, but it includes also analyzing performance and the behavior of the different components. In this chapter, I will not cover, obviously, all the different monitoring tools and practices.

I will implement just two components:

- The **application status indicator**, which is known as the **Health Check indicator**.
- The **application metrics service**, which will be used to **provide various metrics and statistics** about the application.

Implementing the Health Check

Implementing **Health Checks** in Quarkus is a very easy task: we need to add only the **SmallRye Health** extension. Yes ! we need just to add one library to our **pom.xml**! The magic will happen automatically!

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

The **SmallRye Health** will add automatically the **Health Check** endpoints to our Quarkus application.



For those who are used to **Spring Boot**, **SmallRye Health** is the equivalent of **Actuator**.

The new **Health Check** endpoints are:

- **/health/live**: The application is up and running.
- **/health/ready**: The application is ready to serve requests.
- **/health**: Accumulating all health check procedures in the application.

Let's run the application and do a **cURL GET** on the <http://localhost:8080/api/health>:

```
curl -X GET http://localhost:8080/api/health | jq '.'
```

We will get a **JSON response**:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "Database connections health check",
      "status": "UP"
    }
  ]
}
```

This JSON response confirms that the application is correctly running, and there is one **check** that was made confirming that the Database is **UP**.

All the **health REST Endpoints** return a simple **JSON** object with two fields:

- **status** - the overall result of all the health check procedures
- **checks** - an array of individual checks

We have already one check for the Database. Let's create a Health Check for the Keycloak instance:

```
@Liveness
@ApplicationScoped
public class KeycloakConnectionHealthCheck implements HealthCheck {

  @ConfigProperty(name = "mp.jwt.verify.publickey.location", defaultValue = "false")
  private Provider<String> keycloakUrl;                                ①

  @Override
  public HealthCheckResponse call() {

    HealthCheckResponseBuilder responseBuilder =
      HealthCheckResponse.named("Keycloak connection health check");       ③

    try {
      keycloakConnectionVerification();
      responseBuilder.up();                                                 ④
    } catch (IllegalStateException e) {
      // cannot access keycloak
      responseBuilder.down().withData("error", e.getMessage());            ⑤
    }

    return responseBuilder.build();                                         ⑥
  }
}
```

```

private void keycloakConnectionVerification() {
    HttpClient httpClient = HttpClient.newBuilder()
        .connectTimeout(Duration.ofMillis(3000))
        .build();

    HttpRequest request = HttpRequest.newBuilder()
        .GET()
        .uri(URI.create(keycloakUrl.get()))
        .build();

    HttpResponse<String> response = null;

    try {
        response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
        Thread.currentThread().interrupt();
    }

    if (response == null || response.statusCode() != 200) {
        throw new IllegalStateException("Cannot contact Keycloak");
    }
}
}

```

① We get the `mp.jwt.verify.publickey.location` property, which we will use as the Keycloak URL.

② We instantiate the **Java 11 HttpClient** with **3000 milliseconds** of Timeout.

③ We define the name of the Health Check as **Keycloak connection health check**.

④ We will verify that the Keycloak URL is reachable and the response status code is HTTP 200.

⑤ If the `keycloakConnectionVerification()` throws an exception, the Health Check status will be **down**.

⑥ We will build the Health Check response and send it back to the caller.

Now, let's **cURL** again the `/health` endpoint:

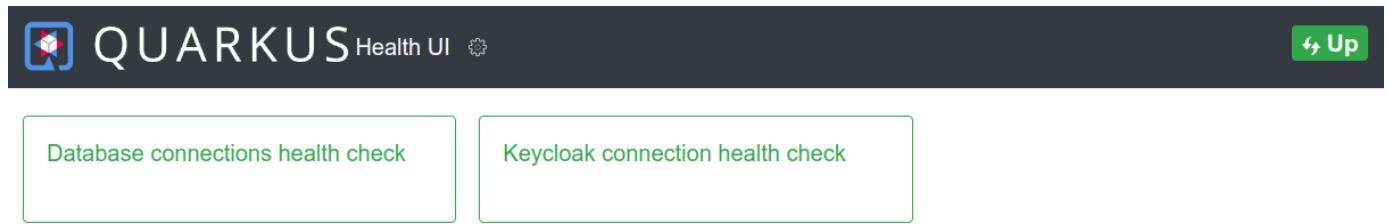
```
curl -X GET http://localhost:8080/api/health | jq .'
```

The new **JSON response** will be 😊

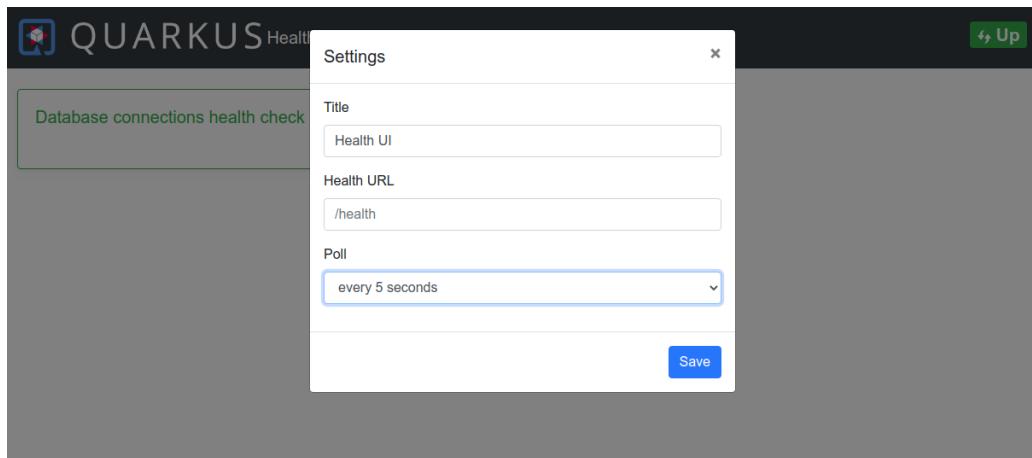
```
{
  "status": "UP",
  "checks": [
    {
      "name": "Keycloak connection health check",
      "status": "UP"
    },
    {
      "name": "Database connections health check",
      "status": "UP"
    }
  ]
}
```

Good! 😊 as simple as that!

There is more! The **SmallRye Health** provides a very useful **Health UI** reachable on the <http://localhost:8080/api/health-ui/>:



There is even a pooling service that can be configured to refresh the page every lap of time:



This **Health UI** is very useful but unfortunately is not activated by default in the **prod** profile. To always enable it in every profile/environment, use this property:

```
### Health Check
quarkus.smallrye-health.ui.always-include=true
```

Excellent 😊 Now we will move to the next step to implement the **Metrics Service**.

Implementing the Metrics Service

Metrics are a very important and critical monitoring data. Quarkus has a dedicated library to expose metrics and a very rich toolset to build custom application metrics. No surprise, the library is also from **SmallRye** - its Maven dependency :

```
<!-- Metrics -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-metrics</artifactId>
</dependency>
```

What is SmallRye?

SmallRye is a project to share and collaborate on implementing specifications that are part of **Eclipse MicroProfile**.

SmallRye implementation includes many components:

- *SmallRye OpenAPI*
- *SmallRye JWT*
- *SmallRye Health*
- *SmallRye Metrics*

and many more! 😊

After adding the **SmallRye Metrics** library, do a cURL GET on the <http://localhost:8080/metrics>:

```
curl -s -H "Accept: application/json" localhost:8080/api/metrics | jq .'
```

The JSON Response looks like:

```
{
  "base": { ①
    "cpu.systemLoadAverage": 2.17,
    "thread.count": 59,
    "classloader.loadedClasses.count": 14535,
    "classloader.unloadedClasses.total": 0,
    "gc.total;name=G1 Young Generation": 8,
    "jvm.uptime": 1198600,
    "thread.max.count": 70,
    "memory.committedHeap": 715128832,
    "classloader.loadedClasses.total": 14535,
    "cpu.availableProcessors": 12,
    "thread.daemon.count": 11,
    "gc.total;name=G1 Old Generation": 0,
    "memory.maxHeap": 8371830784,
    "cpu.processCpuLoad": 0.001836125774615561,
    "memory.usedHeap": 248072000,
    "gc.time;name=G1 Young Generation": 103
  },
  "vendor": { ②
    "memoryPool.usage.max;name=G1 Survivor Space": 39845888,
    "memory.freePhysicalSize": 10268975104,
    "memoryPool.usage.max;name=CodeHeap 'non-profiled nmethods)": 4977920,
    "memoryPool.usage;name=Metaspace": 77818376,
    "memoryPool.usage;name=CodeHeap 'non-profiled nmethods)": 4979456,
    "memoryPool.usage;name=CodeHeap 'profiled nmethods)": 24434816,
    "memoryPool.usage.max;name=CodeHeap 'non-nmethods)": 1458432,
    "memoryPool.usage.max;name=G1 Old Gen": 44376240,
    "cpu.processCpuTime": 25400000000,
    "memory.committedNonHeap": 123338752,
    "memoryPool.usage.max;name=Compressed Class Space": 9450504,
    "memoryPool.usage.max;name=G1 Eden Space": 310378496,
    "memory.freeSwapSize": 4000313344,
    "memoryPool.usage.max;name=Metaspace": 77818376,
    "cpu.systemCpuLoad": 0.1020314472627109,
    "memory.usedNonHeap": 118139264,
    "memoryPool.usage;name=CodeHeap 'non-nmethods)": 1451392,
    "memoryPool.usage;name=G1 Survivor Space": 20971520,
    "memoryPool.usage;name=Compressed Class Space": 9450504,
    "memory.maxNonHeap": -1,
    "memoryPool.usage.max;name=CodeHeap 'profiled nmethods)": 24429568
  },
  "application": {} ③
}
```

As you can notice, the **JSON Payload** has 3 attributes - each attribute represents a separate **Scope**:

① **base** contains all the core information of the server, like:

- General JVM Stats
- Thread JVM Stats
- Thread Pool Stats
- ClassLoading JVM Stats
- Operating System
- and even more!

② **vendor** contains vendor specific metrics that are generated by application frameworks, but not directly declared in application code.

③ **application** contains application specific metrics - we will create our custom application metrics provider.

To get more information about the provided values in a specific SCOPE: just make a cURL OPTIONS command on the `/metrics/SCOPE` endpoint.

For example, a `curl -X OPTIONS -H "Accept: application/json" localhost:8080/api/metrics/base | jq '.'` response will look like:

```
{
  "classloader.loadedClasses.count": {
    "unit": "none",
    "type": "gauge",
    "description": "Displays the number of classes that are currently loaded in the Java virtual machine.",
    "displayName": "Current Loaded Class Count",
    "tags": [
      []
    ]
  },
  "classloader.loadedClasses.total": {
    "unit": "none",
    "type": "counter",
    "description": "Displays the total number of classes that have been loaded since the Java virtual machine has started execution.",
    "displayName": "Total Loaded Class Count",
    "tags": [
      []
    ]
  },
  ...
}
```

Great! ② This is a full description of the **Metrics JSON Payload**.

Now, we will create our own custom application Metrics provider for the **TokenService**:

```
@Slf4j
@RequestScoped
public class TokenService {

    ...

    @Counted(name = "accessTokensRequestsCounter", description = "How many access_tokens
    have been requested.") ①
    @Timed(name = "getAccessTokenRequestsTimer", description = "Measuring time to get an
    access_token.", unit = MetricUnits.MILLISECONDS) ②
    public String getAccessToken(String userName, String password) throws IOException,
    InterruptedException {
        return getAccessToken(jwtIssuerUrlProvider.get(), userName, password,
        clientIdProvider.get(), null);
    }

    ...
}
```

- ① The **@Counted** annotation will create a counter which will increment each time the **getAccessToken()** method is invoked.
- ② The **@Timed** annotation will create a time recorder for the duration of executing the **getAccessToken()** method.

Go to the **Swagger UI** and request an **access_token** many times to generate some metrics, then get the **application metrics**:

```
curl -H "Accept: application/json" localhost:8080/api/metrics/application | jq .'
```

The response will look like:

```
{
  "com.targa.labs.quarkushop.security.TokenService.accessTokensRequests": 33,
  "com.targa.labs.quarkushop.security.TokenService.getAccessTokenTimer": {
    "p99": 107.523443,
    "min": 60.078865,
    "max": 107.523443,
    "mean": 64.48136463077354,
    "p50": 62.960031,
    "p999": 107.523443,
    "stddev": 8.161844154899923,
    "p95": 83.379302,
    "p98": 107.523443,
    "p75": 63.932886,
    "fiveMinRate": 0.017389050867549757,
    "fifteenMinRate": 0.019825628505551772,
    "meanRate": 0.05777492089175027,
    "count": 33,
    "oneMinRate": 5.4358945758975695e-05
  }
}
```

- We have "**accessTokensRequests**": 33 ↗ We recorder 33 requests.
- The **getAccessTokenTimer** object had many attributes:
 - **min**: The shortest duration it took to perform a primality test, probably it was performed for a small number.
 - **max**: The longest duration, probably it was with a large prime number.
 - **mean**: The mean value of the measured durations.
 - **stddev**: The standard deviation.
 - **count**: The number of observations (so it will be the same value as **accessTokensRequests**).
 - **p50, p75, p95, p99, p999**: Percentiles of the durations. For example the value in p95 means that 95 % of the measurements were faster than this duration.
 - **meanRate, oneMinRate, fiveMinRate, fifteenMinRate**: Mean throughput and one-, five-, and fifteen-minute exponentially-weighted moving average throughput.

Although these metrics are so rich, the results format is not compliant with the **Micrometer** framework. To apply the ***Micrometer* compatibility mode**, just add this property to the application.properties:

```
quarkus.smallrye-metrics.micrometer.compatibility=true
```

What is the *Micrometer* compatibility mode?

In the ***Micrometer* compatibility mode**, Quarkus will expose the same 'jvm' metrics that **Micrometer** does instead of the regular **base** and **vendor** metrics. The **application** metrics are unaffected by this mode. The usecase is to facilitate migration from Micrometer-based metrics, because original dashboards for JVM metrics will continue working without having to rewrite them.

Our metrics with the ***Micrometer* compatibility mode**, looks like:

```
{
  "base": {
    "jvm.threads.states;state=new": 0,
    "jvm.memory.max;area=nonheap;id=Metaspace": -1,
    "jvm.buffer.count;id=mapped": 0,
    "jvm.memory.committed;area=heap;id=G1 Eden Space": 308281344,
    "jvm.buffer.total.capacity;id=mapped": 0,
    "jvm.memory.used;area=nonheap;id=Compressed Class Space": 9378640,
    "jvm.memory.committed;area=heap;id=G1 Survivor Space": 25165824,
    "jvm.memory.max;area=nonheap;id=Compressed Class Space": 1073741824,
    "jvm.threads.states;state=runnable": 69,
    "jvm.memory.used;area=nonheap;id=CodeHeap 'non-profiled nmethods)": 4034432,
    "jvm.threads.states;state=timed-waiting": 122,
    "jvm.memory.used;area=nonheap;id=Metaspace": 76176672,
    "jvm.gc.memory.allocated": 0,
    "jvm.classes.loaded": 14344,
    "jvm.classes.unloaded": 0,
    "jvm.memory.committed;area=nonheap;id=CodeHeap 'non-nmethods)": 2555904,
    "jvm.threads.daemon": 171,
    "jvm.memory.committed;area=nonheap;id=Metaspace": 79167488,
    "jvm.memory.used;area=heap;id=G1 Old Gen": 38630976,
    "jvm.memory.max;area=heap;id=G1 Survivor Space": -1,
    "jvm.memory.used;area=heap;id=G1 Survivor Space": 25165824,
    "jvm.buffer.count;id=direct": 154,
    "jvm.memory.used;area=nonheap;id=CodeHeap 'non-nmethods)": 1458944,
    "jvm.memory.committed;area=heap;id=G1 Old Gen": 478150656,
    "jvm.gc.max.data.size": 0,
    "jvm.memory.committed;area=nonheap;id=CodeHeap 'profiled nmethods)": 21561344,
    "process.start.time": 1597857576207,
    "jvm.memory.max;area=heap;id=G1 Old Gen": 8371830784,
    "jvm.memory.max;area=nonheap;id=CodeHeap 'non-nmethods)": 5836800,
    "jvm.memory.max;area=nonheap;id=CodeHeap 'non-profiled nmethods)": 122912768,
    "jvm.threads.peak": 209,
    "jvm.gc.live.data.size": 0,
    "jvm.threads.live": 209,
    "jvm.threads.states;state=blocked": 0,
    "jvm.memory.committed;area=nonheap;id=CodeHeap 'non-profiled nmethods)": 4063232,
```

```

    "jvm.gc.memory.promoted": 0,
    "jvm.buffer.memory.used;id=direct": 805025,
    "jvm.memory.used;area=heap;id=G1 Eden Space": 39845888,
    "jvm.memory.used;area=nonheap;id=CodeHeap 'profiled nmethods)": 21506688,
    "jvm.buffer.memory.used;id=mapped": 0,
    "jvm.memory.max;area=heap;id=G1 Eden Space": -1,
    "process.runtime": 314569,
    "jvm.buffer.total.capacity;id=direct": 805023,
    "jvm.memory.committed;area=nonheap;id=Compressed Class Space": 10485760,
    "jvm.threads.states;state=terminated": 0,
    "jvm.threads.states;state=waiting": 18,
    "jvm.memory.max;area=nonheap;id=CodeHeap 'profiled nmethods)": 122908672
},
"vendor": {},
"application": {
    "com.targa.labs.quarkushop.security.TokenService.getAccessTokenRequestsTimer": {
        "p99": 302.685563,
        "min": 59.226498,
        "max": 302.685563,
        "mean": 69.31378190758618,
        "p50": 63.849377,
        "p999": 302.685563,
        "stddev": 32.88389317091154,
        "p95": 71.019678,
        "p98": 104.676127,
        "p75": 67.422129,
        "fiveMinRate": 0.16007011196502285,
        "fifteenMinRate": 0.05407873859038571,
        "meanRate": 0.15685503462475095,
        "count": 49,
        "oneMinRate": 0.7391211558363393
    },
    "com.targa.labs.quarkushop.security.TokenService.accessTokensRequestsCounter": 49
}
}

```



You can notice that the **vendor** section is empty now.

Great! 🎉 We have now the basic health check & monitoring components that will help us to survive and to be sure our application is healthy and has all the required resources.

Conclusion

I consider that the Security and Monitoring are really anti-disasters layers, but even with the application of these components, the application is still facing many risks. One of them is the **High Availability**.

CHAPTER SIX

Microservices Architecture Pattern

Introduction

We developed **QuarkuShop** as a single-block application. All the components are packaged in a single package. This single block is called **Monolith**.

Based on the **Oxford** English dictionary*, a monolith is "*a single, very large organization that is very slow to change*". In the software architecture world, a **Monolith** or **Monolithic Application** is a single-block application where all its components are combined into a single package.

In the **QuarkuShop** project, if a developer wants to update the definition of a **Product** entity, he will access the same code base, as the other developer who is working on the **PaymentService**. After these updates, we must rebuild and redeploy the updated version of our application.

During runtime, **QuarkuShop** is deployed as into one package, the **Native JAR** that we packaged in a **Docker Container** using the **CI/CD pipelines**. All the Java code and the **HTML/CSS** resources (we have a web page in the **src/main/resources/META-INF/resources**), and even if we are embedding some **React** or **Angular** code in the JAR, everything will be running in the same **BLOCK**.

QuarkuShop will handle HTTP requests, execute some domain-specific logic, retrieve and update data from the database, and handle payloads to be sent to the REST Client.

For all of these points, **QuarkuShop** is a **MONOLITH**. ☺

But is this a good or a bad thing?? ☺

Always in training or discussions when I qualify an Application as **MONOLITH**, many persons start thinking that this is maybe a bad thing ☺ I really cannot understand what is bad in this word, which afraid many developers. I will describe the pros/cons of this architecture, and I will let you decide if it's **good** or **bad**, or **good+bad** (both) ☺

The primary benefit of a monolithic application is **SIMPLICITY**: It is easy to develop, deploy, and scale only one block.

Since the entire application's codebase is in one place, only one slot has to be configured for building and deploying the application.

Monoliths are very easy to work on, especially in the beginning, and it's the default choice while starting a new project. While the complexity will grow over time, agile management of the code base will help to maintain productivity over the lifetime of a monolithic application. Keeping an eye on how the code is written and how the architecture is evolving will protect the project of being a big spaghetti pan.



Although I'm telling you that agility will guarantee that the project will remain clean and easy to work on, this is not **RECIPROCAL**: monolithic application components are very tightly coupled and can become so complex as the product evolves. Thus, it can be extremely difficult for developers to manage over time.

The way to build a complex system that works is to build it from very simple systems that work.

— Kevin Kelly

In real-world projects, in medium or huge projects, it's very common that a developer is working only on a specific part of the application. Also, it is common for each developer to understand only part of a monolith,

meaning very few developers can explain the entirety of the application. This can be the case when it's a small or a medium application, but it will never be the case when it's a huge application. Since the monoliths must be developed and deployed as one unit, it can be difficult to break up development efforts into independent teams. Each code change must be carefully coordinated, which slows down development.

This situation can be a little bit difficult for new developers, who don't hope to deal with a massive code base that has evolved over the years. As a result, more time is spent to find the correct line of code and making sure it doesn't have side effects. So, less time is spent writing new features that will improve the application.

Adopting new technology in a monolith can mean rewriting the whole application, which is a costly and time-intensive drudgery, that doesn't always can be done.

Monoliths are popular because they are simpler to begin building than their alternative, μ services.

Microservices Architecture

What is Microservices Architecture?

While a monolith is a single, large unit, a μ service architecture uses small, modular units of code that can be deployed independently of the rest of a product's components.

Simple can be harder than complex. You have to work hard to get your thinking clean to make it simple.

— Steve Jobs

Microservices architecture allows a big application to divide into small, loosely coupled services. It comes up with a lot of benefits:

1. We have our **QuarkuShop** application which was developed using monolithic architecture, and over the time it has grown with a lot of new features. Now our codebase became huge and some people are going to join our team. It is very difficult for them to get started with a given application as there is no clear separation between various components of the application. Microservices allows us to see our application as small, loosely coupled components, anyone can get started with existing application in a short period and, it wouldn't be too difficult to add a new fix to the codebase.
2. Imagine today we have **Black Friday**, so we will get a huge amount of traffic. And somehow our **Product Catalog** will be highly requested. It will affect the whole application, and it will cause the whole application to go down. If the same happens in μ services architecture, we won't face such failures as it runs multiple services independently so even if one goes down, it won't affect other services.
3. We are some developers who spent a lot of time in terms of understanding our application codebase. We are excited about adding new features for example **products search engine**. You have to redeploy the whole application to show these features to end-users. And we would be working on many such features, just think about the time it is going to take every time in deployment. So if our application is huge, it takes a lot of effort and time which definitely will cause loss of productivity. Nobody likes to wait to see the result of code changes. In a μ services architecture, we will try to make codebase as small as possible in each service so it doesn't take so much time in terms of building, deployments, etc.
4. Today we are getting a huge amount of traffic, and nothing got broken, servers are up. Let's say we need to scale some components of our application, the **Product Catalog** for example. We can't scale individual components in a monolithic architecture. But, in μ services architecture, we can scale any component separately by adding more nodes dynamically.

- Tomorrow, we want to move to a new framework or technology stack. It is so hard to upgrade the monolith as it has become very big, complex over time. It is not that difficult to achieve in **μservices** architecture, as components are small and flexible.

That's how **μservices** architecture makes our life easy. There are various strategies available that help us in dividing the application into small services ex. decompose by domain driven design, decompose by business capability, etc.

What is REALLY a Microservice?

A Microservice, aka service, typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each **μservice** is a mini-application that has its own business logic along with its boundaries, such as REST webservices. Some **μservices** would expose an API that's consumed by other **μservices** or by the application's clients. Other **μservices** might implement a web UI.

Each functional area of the application is now implemented by its own **μservice**. Moreover, the web application is split into a set of smaller web applications.

Each backend service generally exposes a REST API and most services consume APIs provided by other services. The UI services invoke other services to render web pages. Services might also use asynchronous message-based communication.

Some REST APIs are also exposed to client apps, that don't have direct access to the backend services. Instead, communication is mediated by an intermediary known as an API Gateway. The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring.

The Microservices Architecture pattern significantly impacts the relationship between the application and the database. Rather than sharing a single database schema with other services, each service has its database schema. On the one hand, this approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data. However, having a database schema per service is essential if you want to benefit from **μservices** because it ensures loose coupling.

Each of the services has its database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture.

On the surface, the Microservices Architecture pattern is similar to SOA. With both approaches, the architecture consists of a set of services. However, one way to think about the Microservices Architecture pattern is that it's SOA without the commercialization and perceived baggage of **Webservice Specifications (WS)** and an **Enterprise Service Bus (ESB)**. Microservice-based applications favor simpler, lightweight protocols such as REST, rather than heavy protocols. They also very much avoid using ESBs and instead implement ESB-like functionality in the **μservices** themselves.

Making the Switch

So is it possible to switch from a monolithic to **μservices** without starting from scratch? Is it worth it? ☺ Big names like **Netflix**, **Amazon**, and **Twitter** have all shifted from monolithic to **μservices** and have no intention of going back.

Changing architectures can be done in stages. Extracting **μservices** one-by-one out of your monolithic application until you are done. ☺ A good architecture choice can transform your applications and organization for the better. If you're thinking about switching over, you will get a great migration guide by the incoming chapters.

CHAPTER SEVEN

Splitting the Monolith: Bombarding
the domain

Introduction

We've already defined what is a `μservice` and what is the problems solved by this architecture. We listed also many the benefits of adopting `μservices` architecture. But how can I migrate my monolithic application to `μservices` architecture? How can we apply this pattern? How can we split our monolith without rewriting all the application?

We will use Domain-Driven Design as an approach to split our monolith. Domain-Driven Design (DDD), is an approach to software development that simplifies the software modeling and design.

What is Domain-Driven Design ?

The Domain-Driven Design has many advantages:

- Focus on the core business domain and business logic.
- Best way to ensure that the design is based on a model of the domain.
- A tight collaboration way between technical and business teams.

To understand Domain-Driven Design, we need to explain and define many concepts.

Context

The circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood.

Domain

The subject area to which the user applies a program is the domain of the software.

Model

A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

Ubiquitous Language

A common language to communicate within a project. As we have seen, designing a model is the collective effort of software designers, domain experts, and developers; therefore, it requires a common language to communicate with. DDD makes it necessary to use ubiquitous language. Domain models use ubiquitous language in their diagrams, descriptions, presentations, speeches, and meetings. It removes the misunderstanding, misinterpretation, and communication gap among them. Therefore, it must be included in all diagrams, description, presentations, meetings, and so on—in short, in everything.

Strategic Design

Domain-Driven Design addresses the strategy behind the direction of the business and not just the technical aspects. It helps define the internal relationships and early warning feedback systems. On the technical side, strategic design protects each business service by providing the motivation for how an service-oriented architecture should be achieved.

There are various principles that could be followed to maintain the integrity of the domain model, and these

are listed as follows:

Bounded context

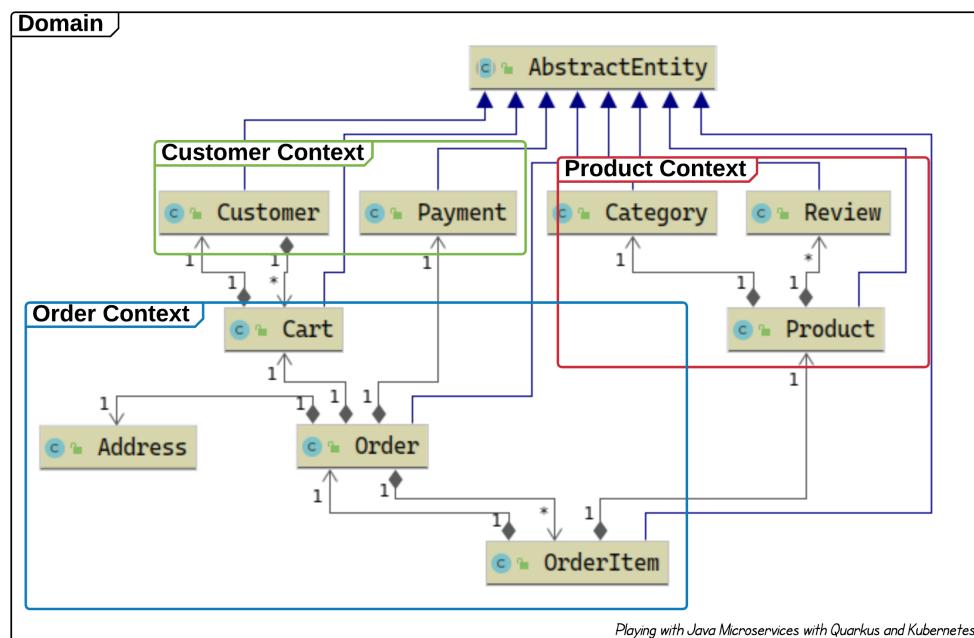
When you have different submodels, it is difficult to maintain the code when all submodels are combined. You need to have a small model that can be assigned to a single team. You might need to collect the related elements and group them. Context keeps and maintains the meaning of the domain term defined for its respective submodel by applying this set of conditions.

These domain terms define the scope of the model that creates the boundaries of the context.

Bounded context seems very similar to the module that you learned about in the previous section. In fact, the module is part of the bounded context that defines the logical frame where a submodel takes place and is developed. Whereas, the module organizes the elements of the domain model, and is visible in the design document and the code.

Now, let's examine the table reservation example we've been using. When you started designing the system, you would have seen that the guest would visit the application, and would request a table reservation at a selected restaurant, date, and time. Then, there is the backend system that informs the restaurant about the booking information, and similarly, the restaurant would keep their system updated in regard to table bookings, given that tables can also be booked by the restaurant themselves. So, when you look at the system's finer points, you can see three domain models:

- The Product domain model
- The Order domain model
- The Customer domain model



They have their own bounded context and you need to make sure that the interface between them works fine.

Bombarding QuarkuShop

Splitting our application will be done in many steps.

Codebase

We will start by creating packages for every bounded context, and then move related classes into them. With NetBeans (and other IDEs) we can move and refactor code easily with some clicks. While moving code, you can find some new bounded context that you may have missed.

At this point, the only way to guarantee that our refactoring did not break our application, is **TESTS !!**

Doing the refactor and moving the code in a small codebase, like our application, is very easy and will take some minutes or even hours. But when dealing with huge applications will be hard and will take several weeks or months. You may not need to sort all code into domain-oriented packages before splitting out your first service, and indeed it can be more valuable to concentrate your effort in one place. There is no need for this to be a big-bang approach. It is something that can be done bit by bit, day by day, and we have a lot of tools at our disposal to track our progress.

We can use structure analysis tools like Stan4j and Structure101.

Dependencies and Commons

While we are moving parts of code to the corresponding package, we will meet some common classes (DTOs or Utilities).. these commonly used classes must be moved to a dedicated **COMMONS** package.

The structure analysis tools are very useful at this point.

Next, we will be talking about the most important element of the Domain Splitting: Entities and Relationships

Entities

Until now, we had grouped our code into packages representing our bounded contexts; we want to do the same for our database access code.

For our database access code, we have already a **Repository** for each **Entity**. The relationships between entities shows us the foreign key relationships between tables, which is database-level constraints.

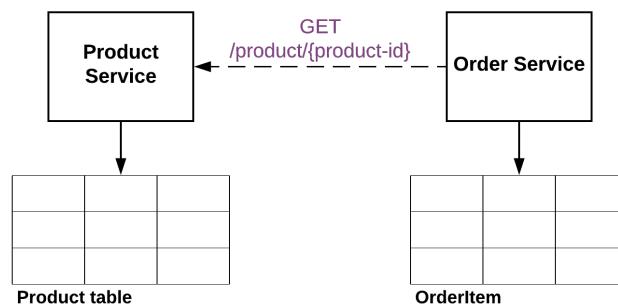
All this helps you understand the coupling between tables that may span what will eventually become service boundaries. But how do we cut those ties? And what about cases where the same tables are used from multiple different bounded contexts? Handling problems like these is not easy, and there are many answers, but it is doable.

Coming back to some concrete examples, let's consider our **QuarkuShop** again. We have identified three bounded contexts, and want to move forward with making them three distinct, interacting services. We are going to look at the problems we might face, and what will be the potential solutions.

Example: Breaking Foreign Key Relationships

In our application, we are storing the products information in a dedicated table, which is mapped and managed by the ORM. The OrderItem object stores in his dedicated table, the reference of the ordered product with the ordered quantity. The reference of the Product record in the OrderItem table consists a foreign key constraint.

So how can we break this relationship? Well, we need to make a change in two places. First, we need to stop the Order code from reaching into the Product table, as this table really belongs to the Product Bounded Context, and we don't want database integration happening once Product and Order are services in their own rights. The quickest way to address this is rather than having the code in Order reach into the Product table, we'll expose the data via an API call in the Product service that the Order code can call. This API call will be the forerunner of a call we will make over the wire, as we see in Figure 5-3.



We have two database calls instead of one, as these two services are separated, while we have only one call in the monolith. So for sure you are thinking now about performance issues?! What I can told you, that the extra call will take some extra time, but will not so huge. You can test performance before splitting, then you should have visibility while making changes.

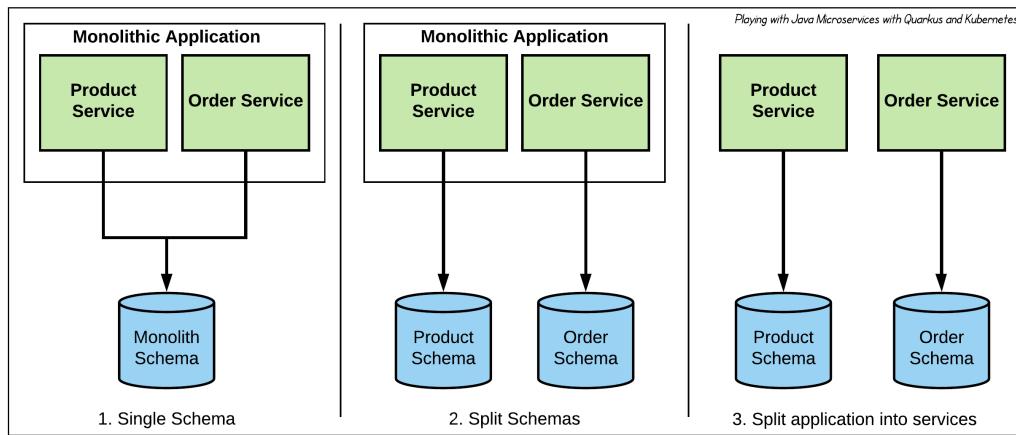
But what about the foreign key relationship? Well, we lose this altogether. This becomes a constraint we need to now manage in our resulting services rather than in the database level. This may mean that we need to implement our own consistency check across services, or else trigger actions to clean up related data. Whether or not this is needed is often not a technologist's choice to make. For example, if our Order service contains a list of IDs for Products, what happens if a Product is removed and an Order now refers to an invalid Product ID? Should we allow it? If we do, then how is this represented in the order when it is displayed? If we don't, then how can we check that this isn't violated? These are questions you'll need to get answered by the people who define how your system should behave for its users.

Refactoring Databases

Staging the Break

We used the defined bounded contexts to define the splitting plans of the database.

As any refactoring must be in steps, no big-bang effect will happen in our migration from a monolithic application with one database schema to many services having its own schema each. We recommend that you split out the schema but keep the service together before splitting the application code out into separate uservices.



With a separate schema, we'll be potentially increasing the number of database calls to perform a single action. Where before we might have been able to have all the data we wanted in a single **SELECT** statement, now we may need to pull the data back from two locations and join in memory. Also, we end up breaking transactional integrity when we move to two schemas, which could have significant impact on our applications; we'll be discussing this next. By splitting the schemas out but keeping the application code together, we give ourselves the ability to revert our changes or continue to tweak things without impacting any consumers of our service. Once we are satisfied that the DB separation makes sense, we can then think about splitting out the application code into two services.

Transactional Boundaries

Transactions are useful things. They allow us to say **these events either all happen together, or none of them happen**. They are very useful when we're inserting data into a database; they let us update multiple tables at once, knowing that if anything fails, everything gets rolled back, ensuring our data doesn't get into an inconsistent state. Simply put, a transaction allows us to group together multiple different activities that take our system from one consistent state to another—everything works, or nothing changes.

Transactions don't just apply to databases, although we most often use them in that context. Message brokers, for example, have long allowed you to post and receive messages within transactions too.

With a monolithic schema, all our create or updates will probably be done within a single transactional boundary. When we split apart our databases, we lose the safety afforded to us by having a single transaction.

Within a single transaction in our existing monolithic schema, inserting records happens within a single transaction.

But if we have pulled apart the schema into two separate schemas, we have lost this transactional safety. We now span many separate transactional boundaries. If one insertion fails, we can clearly stop everything, leaving us in a consistent state. But what happens when the insert into the first table works, but the insert into the second one fails?

Try Again Later

The fact that the insertion was done in the first table might be enough for us, and we may decide to retry the insertion into the second table at a later date. We could queue up this part of the operation in a queue or logfile, and try again later. For some sorts of operations this makes sense, but we have to assume that a retry would fix it.

In many ways, this is another form of what is called eventual consistency. Rather than using a transactional boundary to ensure that the system is in a consistent state when the transaction completes, instead we accept that the system will get itself into a consistent state at some point in the future. This approach is especially useful with business operations that might be long-lived.

Abort the Entire Operation

Another option is to reject the entire operation. In this case, we have to put the system back into a consistent state. The second table is easy, as that insert failed, but we have a committed transaction in the first table. We need to unwind this. What we have to do is issue a compensating transaction, kicking off a new transaction to wind back what just happened. For us, that could be something as simple as issuing a **DELETE** statement to remove the records inserted in the first table. Then we'd also need to report back via the UI that the operation failed. Our application could handle both aspects within a monolithic system, but we'd have to consider what we could do when we split up the application code. Does the logic to handle the compensating transaction live in the customer service, the order service, or somewhere else?

But what happens if our compensating transaction fails? It's certainly possible. In this situation, you'd either need to retry the compensating transaction, or allow some backend process to clean up the inconsistency later on. This could be something as simple as a maintenance screen that admin staff had access to, or an automated process.

Now think about what happens if we have not one or two operations we want to be consistent, but three, four, or five. Handling compensating transactions for each failure mode becomes quite challenging to comprehend, let alone implement.

Distributed Transactions

An alternative to manually orchestrating compensating transactions is to use a distributed transaction. Distributed transactions try to span multiple transactions within them, using some overall governing process called a transaction manager to orchestrate the various transactions being done by underlying systems. Just as with a normal transaction, a distributed transaction tries to ensure that everything remains in a consistent state, only in this case it tries to do so across multiple different systems running in different processes, often communicating across network boundaries.

The most common algorithm for handling distributed transactions—especially short-lived transactions, as in the case of handling our customer order—is to use a two-phase commit. With a two-phase commit, first comes the voting phase. This is where each participant (also called a cohort in this context) in the distributed transaction tells the transaction manager whether it thinks its local transaction can go ahead. If the transaction manager gets a yes vote from all participants, then it tells them all to go ahead and perform their commits. A single no vote is enough for the transaction manager to send out a rollback to all parties.

This approach relies on all parties halting until the central coordinating process tells them to proceed. This

means we are vulnerable to outages. If the transaction manager goes down, the pending transactions never complete. If a cohort fails to respond during voting, everything blocks. And there is also the case of what happens if a commit fails after voting. There is an assumption implicit in this algorithm that this cannot happen: if a cohort says yes during the voting period, then we have to assume it will commit. Cohorts need a way of making this commit work at some point. This means this algorithm isn't foolproof—rather, it just tries to catch most failure cases.

This coordination process also mean locks; that is, pending transactions can hold locks on resources. Locks on resources can lead to contention, making scaling systems much more difficult, especially in the context of distributed systems.

Distributed transactions have been implemented for specific technology stacks, such as Java's Transaction API, allowing for disparate resources like a database and a message queue to all participate in the same, overarching transaction. The various algorithms are hard to get right, so I'd suggest you avoid trying to create your own. Instead, do lots of research on this topic if this seems like the route you want to take, and see if you can use an existing implementation.

So What to Do?

All of these solutions add complexity. As you can see, distributed transactions are hard to get right and can actually inhibit scaling. Systems that eventually converge through compensating retry logic can be harder to reason about, and may need other compensating behavior to fix up inconsistencies in data.

When you encounter business operations that currently occur within a single transaction, ask yourself if they really need to. Can they happen in different, local transactions, and rely on the concept of eventual consistency? These systems are much easier to build and scale (we'll discuss this more in Chapter 11).

If you do encounter state that really, really wants to be kept consistent, do everything you can to avoid splitting it up in the first place. Try really hard. If you really need to go ahead with the split, think about moving from a purely technical view of the process (e.g., a database transaction) and actually create a concrete concept to represent the transaction itself. This gives you a handle, or a hook, on which to run other operations like compensating transactions, and a way to monitor and manage these more complex concepts in your system. For example, you might create the idea of an "in-process-order" that gives you a natural place to focus all logic around processing the order end to end (and dealing with exceptions).

Summary

We decompose our system by focusing on **Bounded Contexts**, and this can be an incremental approach. By getting good at finding these boundaries and working to reduce the cost of splitting out services in the first place, we can continue to grow and evolve our systems to meet whatever requirements come down the road. As you can see, some of this work can be so hard. But the very fact that it can be done incrementally means there is no need to fear this work.

So now we can split our services out, but we've introduced some new problems too. We have many more moving parts to get into production now! So next up we'll dive into the world of deployment.

CHAPTER EIGHT

Applying DDD to the code

Introduction

We saw, in the previous chapter, how to use **Domain Driven Design** to split our **Domain** into **Bounded Contexts**.

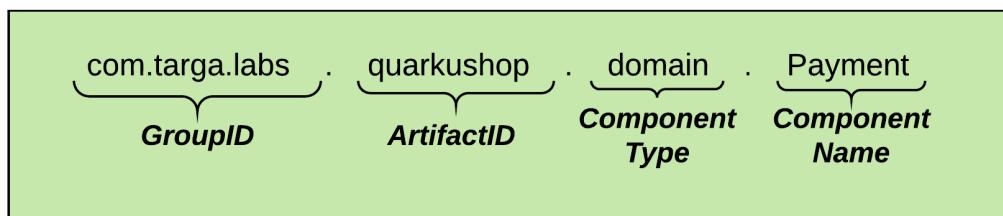
Now we will apply the splitting of our code, using the cut lines that we defined in the previous chapter.

Applying Bounded Contexts to Java Packages

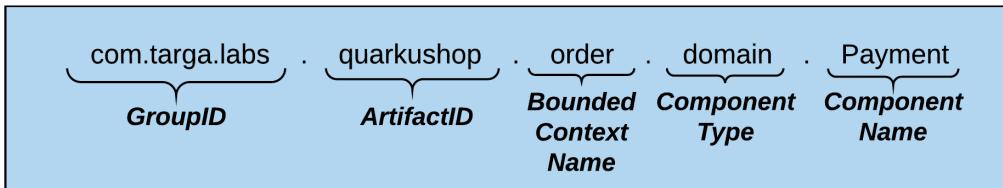
Our classes are already classified by the **component type**, such as *Repository*, *Service*, etc.

We will move every component in a new package, that includes the **Bounded Context** name.

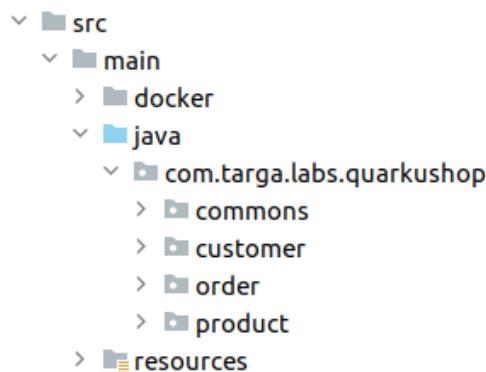
The name of component is actually has the format:



So, after the refactoring, the name will be:



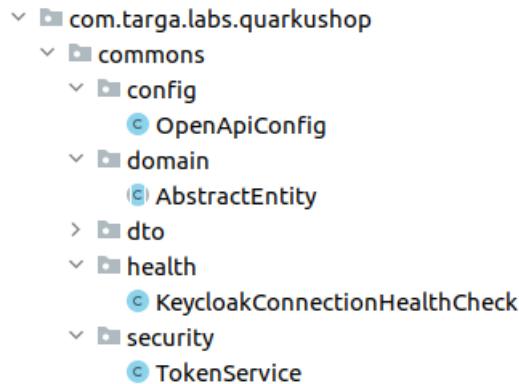
After renaming all the components, we will get a *Project Tree* looking like this:



Wait ! Wait ! What is the content of that **commons** and **configuration** packages ?

The birth of the commons package

I didn't tell you ? Euh, ok :) I'm sure that we said that the common components have to be collected into a dedicated **commons** package, that will be shared by all the **BCs**. Ok, I will show you what I have in my **commons** package:



Ohhhh ! What is this **dto** package? We already said that components have to be moved to their corresponding **Bounded Context**? So if the **Payment** class belongs exclusively to the **Order BC**, then automatically the **PaymentDTO** class has to reside in the **order** package and not the **commons**. So why our **DTOS** are not respecting this recommendation? Are we kidding ?

The answer is obviously NO! We are not kidding. I did this **intentionally** and not by mistake. The reason is very simple: When we was talking about **μservices**, we said that after the splitting, our **μservices** will be talking with each other. So, let we imagine the case: The **Order** wants to get a **Product** from the **Product** **μservice** using a given Product ID. The REST API in the **Product** will return a **ProductDTO** object populated by the data of a **Product** record, which will be serialized to JSON and will be returned back to the requesting service, which is the **Order**. But how can handle the received JSON if its format is unknown to him ? The answer is **it can't**, when the **ProductDTO** is exclusively stored in the **Product** **μservice**.

An other idea, is to store the **DTO** classes in both services, but this idea seems to inefficient for me, as it will result of duplicated code to handle. Imagine the case that we need to update the **DTO** classes, in this case we will be doing the same work twice..

So the best idea that I found is to store the **DTOS** in the **commons**. This solution will make sharing and upgrades so easy.

Other than the **dto** package, we find also a **domain** package.

What ? a Domain in the common area?

No, it's not really a domain, it has only **AbstractEntity** class which a common class, used a parent class for all the **Entity** classes in every **BC**. As the entities belong to the Domain, I thought that the most suitable place to put the common **AbstractEntity** is the **commons.domain** package.

Finally, there is also: **utils** package, that will host the **Utility** classes, for example we have actually a **Class** that contains **constants** which looks like this:

- **config**: contains the **OpenApiConfig** used for configuring the OpenAPI implementation & the Swagger UI
- **health**: contains the custom Keycloak Health Check
- **security**: contains the **TokenService** used to grab the JWT Token.

Locating & breaking the BC Relationships

Locating the BC Relationships

This step aims to break dependencies between **Bounded Contexts**. To be able to break them, we need first to find these dependencies. There are many ways to find them. For example reviewing the **Class Diagrams**, the source code, etc. In the previous chapter we talked about tools that can help us highlight the dependencies between blocs in our monolith.

I am mainly using **STAN** (Aka STAN4J), a powerful structure analysis for Java. **STAN** supports a set of carefully selected metrics, suitable to cover the most important aspects of structural quality. Special focus has been set on visual dependency analysis, a key to structure analysis.

STAN is available in two variants:

1. as a standalone application for Windows and Mac OS, which is targeted to architects and project managers who are typically not using the IDE
2. as an extension to the Eclipse Integrated Development Environment (IDE), that allows the developer to quickly explore the structure for any bunch of code she desires.

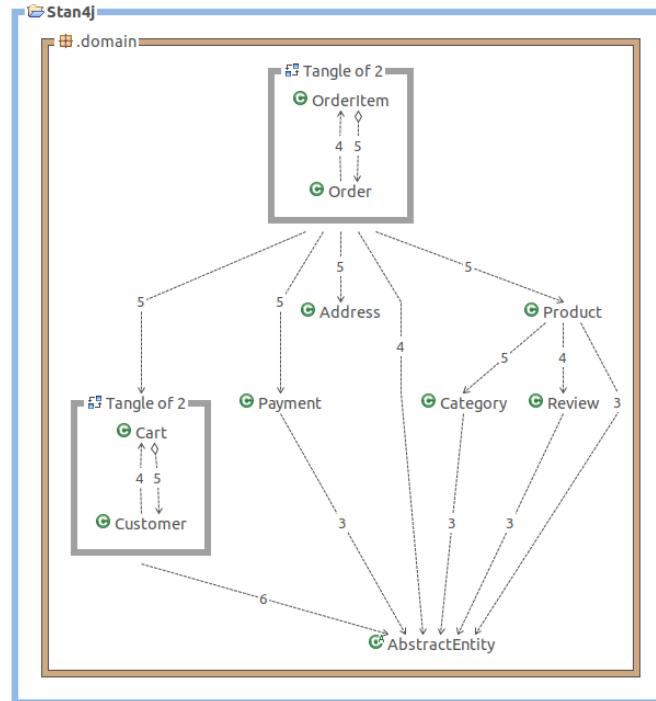
Yeah! You are right! We will be using the second choice: I got a fresh install of Eclipse IDE and I installed the IDE Extension as described in <http://stan4j.com/download/ide/>.

Our monolith is a Maven-based project, so it can be easily imported into Eclipse IDE. After doing the import, just do a :

1. Right click on the project
2. **Run As > Maven Build**
3. Choose the Maven configuration, if you already have one
4. In Goals, just enter: **clean install -DskipTests** and Run
5. Done !

Next, to make a structure analysis of our project:

1. Right click on the project
2. **Run As > Structure Analysis**
3. Yoppi ! ☺ The structure diagram is here !



Breaking the BC Relationships

Now, we will break the relationship between the Bounded Contexts. We can start by the relations between entities. Generally the relations are more effective between tables in the database. Here the entity classes are treated as relational tables (concept of JPA), therefore the relationships between Entity classes are as follows:

- **@ManyToOne** Relation
- **@OneToMany** Relation
- **@OneToOne** Relation
- **@ManyToMany** Relation

To be more precise, we will not break relationships between entities that belong to the same **BC**, we will break **inter-BC** relationships. For example, we have the relationship between the **OrderItem**, which belongs to the **Order Context**, and the **Product**, which belongs to the **Product Context**.

In Java, this relationship is represented by:

```
public class OrderItem extends AbstractEntity {
    @NotNull
    @Column(name = "quantity", nullable = false)
    private Long quantity;

    @ManyToOne
    private Product product;

    @ManyToOne
    private Order order;
}
```

The `@ManyToOne` annotating the `product` field is our target here.

How to break the relationship ? It's simple, the bloc :

```
@ManyToOne
private Product product;
```

Will be changed by :

```
private Long productId;
```

Ok, why we picked up the `Long` type to replace the `Product` type ? It's simple ! `Long` is the type of the `Product`'s ID.

Great ! We have to do the same in the `Product` class, if the relationship between `OrderItem` and `Product` is a bidirectional relationship.

Ok :) When we try to build the project, using `mvn clean install`, we will get a Compilation problems. This is obvious, as we edited our `OrderItem`, many components using this class has to be aware about these modifications. Here, it is the `OrderItemService` that is making trouble.

Let's see the blocks of the `OrderItemService` class that have errors, the first one:

```
public OrderItemDto create(OrderItemDto orderItemDto) {
    log.debug("Request to create OrderItem : {}", orderItemDto);

    var order = this.orderRepository
        .findById(orderItemDto.getOrderId())
        .orElseThrow(() -> new IllegalStateException("The Order does not exist!"));

    var product = this.productRepository
        .findById(orderItemDto.getProductId())
        .orElseThrow(() -> new IllegalStateException("The Product does not exist!"));

    return mapToDto(
        this.orderItemRepository.save(
            new OrderItem(
                orderItemDto.getQuantity(),
                product,
                order
            )));
}
```

Here we see that we are getting a `Product` instance from the `ProductRepository`, and we are passing it to the `OrderItem` constructor. As we have no more the `Product` field in the `OrderItem` class, and it has been changed to `Product ID`, we need to pass the ID to the `OrderItem` constructor, instead of the `Product`.

But we already said that we will have no more the `Product` object in the `Order Bounded Context`. How can we talk about dealing with `Product` and `ProductRepository` classes until now?

Did you forget? We have in the scope of `Order Bounded Context`, the `commons` module, which contains the `ProductDTO` and we can use for free :D

For the `ProductRepository`, it's obvious, it will be replaced by a REST Client that will gather the `Product` data from the `Product` μservice, and populate the known `ProductDTO` object when needed. But here, we need only the `Product ID` to create the `OrderItem` instance. So, in this case, we dont need to get the Product record.

The second block is the method that maps the `OrderItem` to an `OrderItemDto`:

```
public static OrderItemDto mapToDto(OrderItem orderItem) {
    return new OrderItemDto(
        orderItem.getId(),
        orderItem.getQuantity(),
        orderItem.getProductId(),
        orderItem.getOrder().getId()
    );
}
```

The reference of the **Product** has to be deleted from this method, as it was removed from the **OrderItem** class. We don't have no more a product as member, but we have the Product ID in the **OrderItem** class. So the instruction `orderItem.getProduct().getId()` has to be changed to `orderItem.getProductId()`. To get the product price to add/subtract from the order total price, we will inject the **ProductRepository** in our **OrderItemService**.

The resulting **OrderItemService** will look like:

```

@Slf4j
@ApplicationScoped
@Transactional
public class OrderItemService {

    @Inject OrderItemRepository orderItemRepository;
    @Inject OrderRepository orderRepository;
    @Inject ProductRepository productRepository;

    public static OrderItemDto mapToDto(OrderItem orderItem) {
        return new OrderItemDto(orderItem.getId(),
            orderItem.getQuantity(), orderItem.getProductId(),
            orderItem.getOrder().getId());
    }

    public OrderItemDto findById(Long id) {
        log.debug("Request to get OrderItem : {}", id);
        return this.orderItemRepository.findById(id)
            .map(OrderItemService::mapToDto).orElse(null);
    }

    public OrderItemDto create(OrderItemDto orderItemDto) {
        log.debug("Request to create OrderItem : {}", orderItemDto);
        var order = this.orderRepository
            .findById(orderItemDto.getOrderId()).orElseThrow(() ->
            new IllegalStateException("The Order does not exist!"));

        var orderItem = this.orderItemRepository.save(
            new OrderItem(orderItemDto.getQuantity(),
                orderItemDto.getProductId(), order
            ));

        var product = this.productRepository.getOne(orderItem.getProductId());

        order.setPrice(order.getPrice().add(product.getPrice()));
        this.orderRepository.save(order);

        return mapToDto(orderItem);
    }
}

```

```

public void delete(Long id) {
    log.debug("Request to delete OrderItem : {}", id);

    var orderItem = this.orderItemRepository.findById(id)
        .orElseThrow(() ->
            new IllegalStateException("The OrderItem does not exist!"));

    var order = orderItem.getOrder();
    var product = this.productRepository.getOne(orderItem.getProductId());

    order.setPrice(order.getPrice().subtract(product.getPrice()));

    this.orderItemRepository.deleteById(id);

    order.getOrderItems().remove(orderItem);
    this.orderRepository.save(order);
}

public List<OrderItemDto> findByOrderId(Long id) {
    log.debug("Request to get all OrderItems of OrderId {}", id);
    return this.orderItemRepository.findAllByOrderId(id)
        .stream()
        .map(OrderItemService::mapToDto)
        .collect(Collectors.toList());
}
}

```

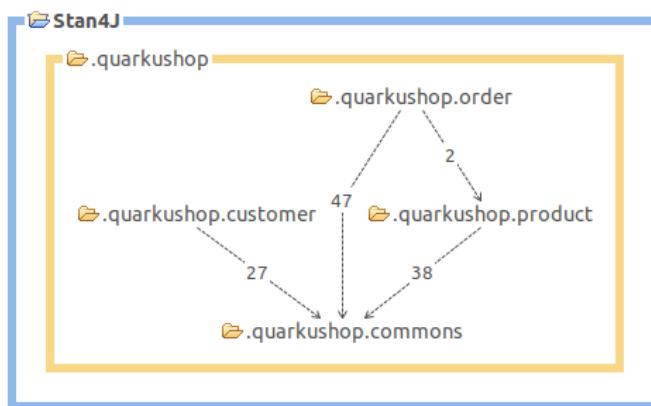
After this refactoring, we can use our great tool **STAN** to check how our modifications changed the structure of our project.

After some checks, the remaining modifications to do are:

1. in the **customer** package:
 1. Cart: the **private Order order;** will be changed by **private Long orderId;**
 2. CartService: the **Order** reference will be changed by **OrderDto**
2. in the **order** package:
 1. Order: the **private Cart cart;** will be changed by **private Long cartId;**
 2. OrderService: the **Cart** reference will be changed by **CartDto**

After these modifications, we will still have an **OrderService** reference in the **CartService**. As we said before, this **OrderService** will be replaced by a REST Client that will make calls to the API exposed by the **Order** **UserService**. You can comment the lines referencing the **OrderService** before analyzing the structure.

After this refactoring, we can check how our modifications changed the structure of our project:



We didn't break totally the link between **order** and **product** packages. This is because to calculate an order total amount, we need to have the product price. We will see how to deal with this link in the future steps.

This is good ! We just finished one of the heaviest tasks in our migration trip !

Now, we can start building our standalone µservices. But, we need to learn more about the best practices and µservices patterns.

CHAPTER NINE

Meeting the microservices concerns
and patterns

Introduction

When we deal with architecture and design, we immediately start thinking about recipes and patterns. These design patterns are useful for building reliable, scalable, secure applications in the cloud.

A pattern is a reusable solution to a problem that occurs in particular context. It's an idea, which has its origins in real-world architecture, that has proven to be useful in software architecture and design.

The Microservices architecture is being proclaimed as the most powerful architectural pattern. We already discussed this pattern in details in **Chapter 8**.

When presenting a pattern, we will start by defining the context and the problem that we can face, and the solution given by the pattern.

Cloud Patterns

We will deal with these patterns:

- Externalized configuration
- Service discovery and registration
- Circuit Breaker
- Database per service
- API gateway
- CQRS
- Event sourcing
- Log aggregation
- Distributed tracing
- Audit logging
- Application metrics
- Health check API

Service discovery and registration

Context and problem

In the μ services world, Service Registry and Discovery plays an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers. In addition to that, in Cloud environments service instances may come up and go down anytime. So we need some automatic service registration and discovery mechanism.

In a monolithic application, calls between components are made through language-level calls. But, in μ services architecture, services typically need to call each another using HTTP/REST (or other) calls. In order to make a request, a service needs to know the network location (IP address and port) of a given service instance. As we said in previous chapters, μ services have dynamically assigned network locations, due to many factors such as, different deployment frequencies for example. Moreover, a service can have more than one instance that can keep change dynamically because of autoscaling, failures, etc..

So, we must implement a mechanism that enables the clients of a given service to make requests to a

dynamically changing set of the service instances.

How does the client of a service discover the location of a service instance? How does the client of a service know about the available instances of a service?

Solution

We can create a service registry, which is a database of available service instances. Which works like this:

- The network location of a μ service instance is registered with the service registry when the instance starts up.
- The network location of a μ service instance is removed from the service registry when the instance terminates.
- The μ service instance availability is typically refreshed periodically using a heartbeat mechanism.

We have two varieties of this pattern:

- **Client-side Discovery pattern:** The requesting service (client) is responsible for seeking the network locations of available service instances. The client queries a service registry and then selects, using some **load-balancing** algorithms, one of the available service instances and makes a request. This mechanism follows the **client-side discovery pattern**.
- **Server-side Discovery pattern:** The **server-side discovery pattern** advises that the client makes a request to a service via a **load balancer**. It's the responsibility of the load balancer to query the service registry and to forward each request to an available service instance. So, in case we want to follow this pattern, we need to have (or implement) the discussed **load balancer**.

Externalized configuration

Context and problem

An application typically uses one or more infrastructure (a message broker and a database server, etc..) and 3rd party services (a payment gateway, email and messaging, etc..). These services need configuration information (credentials for example) to be used. This configuration information is stored in files that are deployed with the application.

In some cases, it's possible to edit these files to change the application behavior after it's been deployed. However, changes to the configuration require the application be redeployed, often resulting in unacceptable downtime and other administrative overhead.

Local configuration files also limit the configuration to a single application, but sometimes it would be useful to share configuration settings across multiple applications. Examples include database connection strings or the URLs of queues and storage used by a related set of applications.

Our monolith is divided into many μ services. All these μ services need the configuration information that was provided to the monolith. Imagine that we need to update the database url. This task needs to be done for all the μ services. If we forget to update the data somewhere, it can result in instances using different configuration settings while the update is being deployed.

Solution

Externalize all application configuration including the database credentials and network location. We can for example store the configuration information externally, and provide an interface that can be used to quickly and efficiently read and update configuration settings. We can call this configuration store as **Configuration**

Server.

When a `μservice` starts up, it reads the configuration from the given **Configuration Server**.

Circuit Breaker

Context and problem

Microservices are communicating using mainly HTTP REST requests. When a `μservice` synchronously another one, there is always the risk that the other service is unavailable or unreachable high latency it is essentially unusable. The unsuccessful calls might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.

Solution

A requesting `μservice` should invoke a remote service via a proxy that works in a similar mechanism to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

The Circuit Breaker pattern, popularized by **Michael Nygard** in his book, **Release It!**, can prevent an application from repeatedly trying to execute an operation that's likely to fail. Allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long lasting. The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.

Database per service

Context and problem

In the `μservices` architecture world, the services must be loosely coupled so that they can be developed, deployed and scaled independently.

Most services need to persist data in some kind of database. In our application, , the **Order Service** stores information about orders and the **Customer Service** stores information about customers.

What's the database architecture in a `μservices` application?

Solution

Keep each `μservice`'s persistent data private to that service and accessible only via its API.

The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

There are a few different ways to keep a service's persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

- Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
- Schema-per-service – each service has a database schema that's private to that service

- Database-server-per-service – each service has its own database server.

Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service's API and access its data directly.

API gateway

Context and problem

For our boutique, imagine that we are implementing the product details page. Let's imagine that we need to develop multiple versions of the product details user interface:

- HTML5/JavaScript-based UI for desktop and mobile browsers - HTML is generated by a server-side web application
- Native Android and iPhone clients - these clients interact with the server via REST APIs

In addition, **QuarkuShop** must expose product details via a REST API for use by 3rd party applications.

A product details UI can display a lot of information about a product. For example:

- Basic information about the product such as name, description, price, etc..
- Your purchase history for the product
- Availability
- Buying options
- Other items that are frequently bought with this product
- Other items bought by customers who bought this product
- Customer reviews

Since **QuarkuShop** follows the Microservice architecture pattern, the product details data is spread over multiple services:

- Product Service - basic information about the product such as name, description, price, customer reviews, product availability..
- Order service - purchase history for product..
- Customer service - customers, carts..

Consequently, the code that displays the product details needs to fetch information from all of these services.

How do the clients of a Microservices-based application access the individual services?

Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in

one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. The API gateway might also implement security, e.g. verify that the client is authorized to perform the request.

CQRS

Context and problem

In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository. These entities can be a subset of the rows in one or more tables in a relational database such as SQL Server.

Typically in these systems, all create, read, update, and delete (CRUD) operations are applied to the same representation of the entity. For example, a data transfer object (DTO) representing a customer is retrieved from the data store by the data access layer (DAL) and displayed on the screen. A user updates some fields of the DTO (perhaps through data binding) and the DTO is then saved back in the data store by the DAL. The same DTO is used for both the read and write operations.

Traditional CRUD designs work well when only limited business logic is applied to the data operations. Scaffold mechanisms provided by development tools can create data access code very quickly, which can then be customized as required.

However, the traditional CRUD approach has some disadvantages:

- It often means that there's a mismatch between the read and write representations of the data, such as additional columns or properties that must be updated correctly even though they aren't required as part of an operation.
- It risks data contention when records are locked in the data store in a collaborative domain, where multiple actors operate in parallel on the same set of data. Or update conflicts caused by concurrent updates when optimistic locking is used. These risks increase as the complexity and throughput of the system grows. In addition, the traditional approach can have a negative effect on performance due to load on the data store and data access layer, and the complexity of queries required to retrieve information.
- It can make managing security and permissions more complex because each entity is subject to both read and write operations, which might expose data in the wrong context.

Solution

Command and Query Responsibility Segregation (CQRS) is a pattern that segregates the operations that read data (queries) from the operations that update data (commands) by using separate interfaces. This means that the data models used for querying and updates are different. The models can then be isolated.

Compared to the single data model used in CRUD-based systems, the use of separate query and update models for the data in CQRS-based systems simplifies design and implementation. However, one disadvantage is that unlike CRUD designs, CQRS code can't automatically be generated using scaffold mechanisms.

The query model for reading data and the update model for writing data can access the same physical store, perhaps by using SQL views or by generating projections on the fly.

The read store can be a read-only replica of the write store, or the read and write stores can have a different

structure altogether. Using multiple read-only replicas of the read store can greatly increase query performance and application UI responsiveness, especially in distributed scenarios where read-only replicas are located close to the application instances.

Event sourcing

Context and problem

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it. For example, in the traditional create, read, update, and delete (CRUD) model a typical data process is to read data from the store, make some modifications to it, and update the current state of the data with the new values—often by using transactions that lock the data.

The CRUD approach has some limitations:

- CRUD systems perform update operations directly against a data store, which can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires.
- In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data.
- Unless there's an additional auditing mechanism that records the details of each operation in a separate log, history is lost.

Solution

The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted. Each event represents a set of changes to the data (such as [AddedItemToOrder](#)).

The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. As the application adds new orders, adds or removes items on the order, and adds shipping information, the events that describe these changes can be handled and used to update the materialized view.

In addition, at any point it's possible for applications to read the history of events, and use it to materialize the current state of an entity by playing back and consuming all the events related to that entity. This can occur on demand to materialize a domain object when handling a request, or through a scheduled task so that the state of the entity can be stored as a materialized view to support the presentation layer.

Log aggregation

Context and problem

In the `μ`services architecture, our application consists of multiple services and service instances that are running on different servers and locations. Requests often cross multiple service instances.

When we had the monolith, the application is generating one log stream, which is generally stored in one log file/directory. Now, each service instance generates its own log file.

How to understand the behavior of an application and troubleshoot problems when log is splitted ?

Solution

Use a centralized logging service that aggregates logs from each service instance. When the log is aggregated, the users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs.

Distributed tracing

Context and problem

In the μ services architecture, requests often span multiple services. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, etc.

When a request fails, how to understand the behaviour and troubleshoot the problems ?

Solution

Instrument services with code that

- Assigns each external request a unique external Request ID
- Passes the external Request ID to all services that are involved in handling the request
- Includes the external Request ID in all log messages
- Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

Audit logging

Context and problem

In a μ services architecture, we need more visibility about our services and how things are going, in addition to the logging system.

How to understand the behavior of users and the application and troubleshoot problems?

Solution

We can for example record the user activity in a database, or some special dedicated logging system.

Application metrics

Context and problem

In a μ services architecture, we need more visibility about our services and what's going on, in addition to indicators that we already have.

How to understand and articulate the application behavior ?

Solution

The recommended solution is to have a centralized metrics service that gathers and stocks the decision-enabling statistics of each of the service operations. Microservices can push their metrics information to the metrics service. On the other side, the metrics service can pull metrics from the μservice.

Health check API

Context and problem

It's a good practice, and often a business requirement, to monitor web applications and back-end services, to ensure they're available and performing correctly. However, it's more difficult to monitor services running in the cloud than it is to monitor on-premises services. There are many factors that affect applications such as network latency, the performance and availability of the underlying compute and storage systems, and the network bandwidth between them. The service can fail entirely or partially due to any of these factors. Therefore, you must verify at regular intervals that the service is performing correctly to ensure the required level of availability.

Solution

Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.

A health monitoring check typically combines two factors:

- The checks (if any) performed by the application or service in response to the request to the health verification endpoint.
- Analysis of the results by the tool or framework that performs the health verification check.

The response code indicates the status of the application and, optionally, any components or services it uses. The latency or response time check is performed by the monitoring tool or framework.

Security between services: Access Token

Context and problem

In a μservices architecture and with the use of the API Gateway pattern. The application is composed of numerous services. The API gateway is the single entry point for client requests. It authenticates requests, and forwards them to other services, which might in turn invoke other services.

How to communicate the identity of the requestor to the services that handle the request?

Solution

The API Gateway authenticates the request and passes an access token (e.g. JSON Web Token) that securely identifies the requestor in each request to the services. A service can include the access token in requests it makes to other services.

What's next?

Now that we splitted our monolith and we discovered some useful patterns, we can start building our standalone μservices.

CHAPTER TEN

Getting started with Kubernetes

Introduction

To deploy our (so huge, so big) application 😊 we will be using 🚤 **Docker**. We will deploy our code in a container, so we can enjoy the great feature provided by 🚤 **Docker**.

🚢 **Docker** has become the standard to develop and run containerized applications.

This is great ! Using 🚤 **Docker** is quietly simple, especially in development stages. Deploying containers in the same server (**docker-machine**) is simple, but when start thinking to deploy many containers to many servers, things become complicated (managing the servers, managing the container state, etc).

Here come the orchestration system, which provides many features like orchestrating computing, networking and storage infrastructure on behalf of user workloads:

- Scheduling: matching containers to machines based on many factors like resources needs, affinity requirements, etc.
- Replications
- Handling failures
- Etc.

For our tutorial, we will choose ⚙️ **Kubernetes**, the star of container orchestration.

What is Kubernetes ?

⚙️ **Kubernetes** (aka **K8s**) was a project spun out of **G Google** as a open source next-gen container scheduler designed with the lessons learned from developing and managing *Borg* and *Omega*.



Kubernetes is designed to have loosely coupled components centered around deploying, maintaining and scaling applications. **K8s** abstracts the underlying infrastructure of the nodes and provides a uniform layer for the deployed applications.

Kubernetes Architecture

In the big plan, a **Kubernetes** cluster is composed of two items:

- **Master Nodes**: The main control plane for **Kubernetes**. It contains an *API Server*, a *Scheduler*, a *Controller Manager* (**K8s** cluster manager) and a datastore to save the cluster state called **Etcd**.
- **Worker Nodes**: A single host, physical or virtual machine, able to run a **POD**. They are managed by the **Master nodes**.

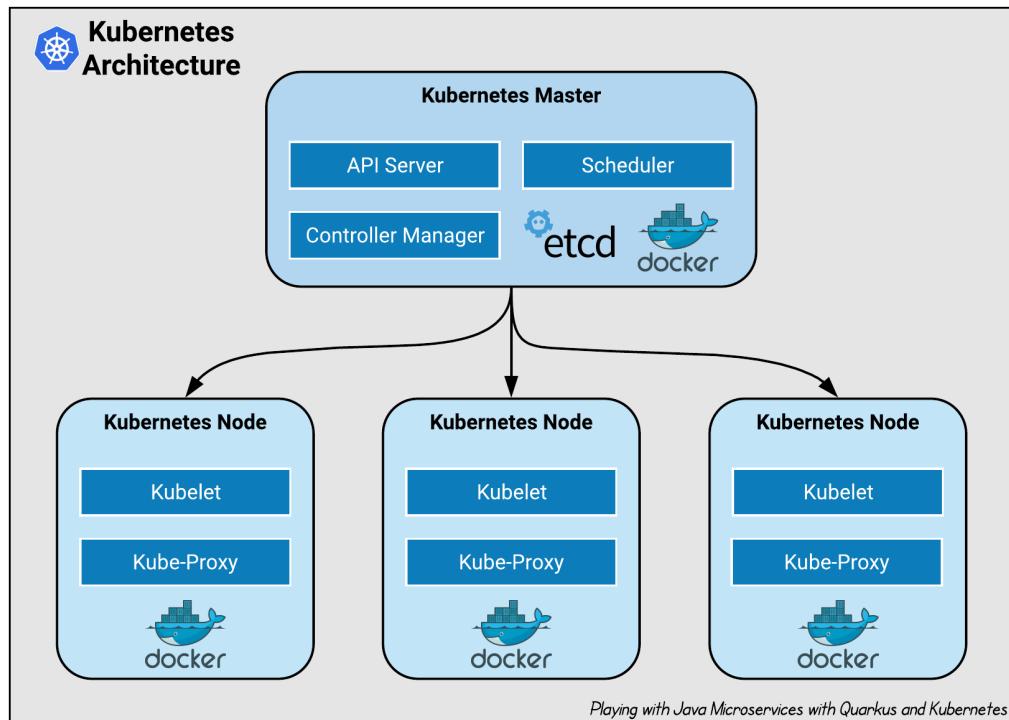
Let's have a look inside a **Master node**:

- **Kube API-Server**: allows the communication, thru *REST APIs*, between the **Master Node** and all its clients such as *Worker Nodes*, **kube-cli**, etc.

- **Kube Scheduler:** a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity to assign a Node to a newly created **POD**.
- **Kube Controller Manager:** a daemon that embeds the core control loops shipped with **Kubernetes**. A control loop is a permanent listener that regulates the state of the system. In **Kubernetes**, a controller is a control loop that watches the shared state of the cluster through the API-Server and makes changes attempting to move the current state towards the desired state.
- **Etcd:** a strong, consistent and highly available key-value store used for persisting the cluster state.

Then, what about a **Worker node**?

- **Kubelet:** an agent that runs on each node in the cluster. It makes sure that containers are correctly running in a **POD**.
- **Kube-Proxy:** enables the **Kubernetes** service abstraction by maintaining network rules on the host and perform networking actions.



The Container Runtime that we will use is **Docker**. **Kubernetes** is compatible with many others like **cri-o**, **rkt**, etc.

Kubernetes Core Concepts

The **K8s** ecosystem covers many concepts and components. We will try to introduce them briefly.

Kubectl

The **kubectl** is a command line interface for running commands against **Kubernetes** clusters.

Cluster

A collection of hosts that aggregate their resources (*CPU, Ram, Disk, etc.*) into one common usable pool, that can be shared (and controlled) between the cluster resources.

Namespace

A logical partitioning capability that enable one **Kubernetes** cluster to be used by multiple users, teams of users, or a single user with multiple applications without concern for undesired interaction. Each user, team of users, or application may exist within its Namespace, isolated from every other user of the cluster and operating as if it were the sole user of the cluster.

List all Namespace: `kubectl get namespace` or `kubectl get ns`

Label

Key-value pairs that are used to identify and select related sets of objects. Labels have a strict syntax and defined character set.

Annotation

Key-value pairs that contain non-identifying information or metadata. Annotations do not have the syntax limitations as labels and can contain structured or unstructured data.

Selector

Selectors use labels to filter or select objects. Both equality-based (=, ==, !=) or simple key-value matching selectors are supported.

Use case of Annotations, Labels and Selectors.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
      ports:
        - containerPort: 80
```

Pod

It is the basic unit of work for **Kubernetes**. Represent a collection of containers that share resources, such as IP addresses and storage.

Pod Example

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

List all Pods: `kubectl get pod` or `kubectl get po`

ReplicationController

A framework for defining pods that are meant to be horizontally scaled. A replication controller includes a pod definition that is to be replicated, and the pods created from it can be scheduled to different nodes.

ReplicationController Example

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

List all ReplicationControllers: `kubectl get replicationcontroller` or `kubectl get rc`

ReplicaSet

An upgraded version of ReplicationController that supports set-based selectors.

ReplicaSet Example

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
          env:
            - name: GET_HOSTS_FROM
              value: dns
          ports:
            - containerPort: 80
```

List all ReplicaSets: `kubectl get replicaset` or `kubectl get rs`

Deployment

Includes a Pod template and a replicas field. Kubernetes will make sure the actual state (amount of replicas, Pod template) always matches the desired state. When you update a Deployment it will perform a “rolling update”.

Deployment Example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
        ports:
          - containerPort: 80
```

List all Deployments: `kubectl get deployment`

StatefulSet

A controller that aims to manage Pods that must persist or maintain state. Pod identity including hostname, network, and storage will be persisted.

StatefulSet Example

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx
          image: k8s.gcr.io/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: "my-storage-class"
        resources:
          requests:
            storage: 1Gi

```

List all StatefulSets: `kubectl get statefulset`

DaemonSet

Ensures that an instance of a specific pod is running on all (or a selection of) nodes in a cluster.

DaemonSet Example

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
        - key: node-role.kubernetes.io/master
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: gcr.io/google-containers/fluentd-elasticsearch:1.20
  terminationGracePeriodSeconds: 30
```

List all DaemonSets: `kubectl get daemonset` or `kubectl get ds`

Service

Define a single IP/port combination that provides access to a group of pods. It uses label selectors to map groups of pods and ports to a cluster-unique virtual IP.

Service Example

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
```

List all Services: `kubectl get service` or `kubectl get svc`

Ingress

An ingress controller is the primary method of exposing a cluster service (usually `http`) to the outside world. These are load balancers or routers that usually offer `SSL` termination, name-based virtual hosting, etc.

Ingress Example

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - http:
        paths:
          - path: /testpath
            backend:
              service:
                name: test
                port:
                  number: 80
```

List all Ingress: `kubectl get ingress`

Volume

Storage that is tied to the Pod Lifecycle, consumable by one or more containers within the pod.

PersistentVolume

A PersistentVolume (PV) represents a storage resource. PVs are commonly linked to a backing storage resource, NFS, GCEPersistentDisk, RBD etc. and are provisioned ahead of time. Their lifecycle is handled independently from a pod.

List all PersistentVolumes: `kubectl get persistentvolume` or `kubectl get pv`

PersistentVolumeClaim

A PersistentVolumeClaim (PVC) is a request for storage that satisfies a set of requirements. Commonly used with dynamically provisioned storage.

List all PersistentVolumeClaims: `kubectl get persistentvolumeclaim` or `kubectl get pvc`

StorageClass

Storage classes are an abstraction on top of an external storage resource. These will include a provisioner, provisioner configuration parameters as well as a PV reclaimPolicy.

List all StorageClasses: `kubectl get storageclass` or `kubectl get sc`

Job

The job controller ensures one or more pods are executed and successfully terminates. It will do this until it satisfies the completion and/or parallelism condition.

List all Jobs: `kubectl get job`

CronJob

An extension of the Job Controller, it provides a method of executing jobs on a cron-like schedule.

List all CronJobs: `kubectl get cronjob`

ConfigMap

Externalized data stored within **Kubernetes** that can be referenced as a commandline argument, environment variable or injected as a file into a volume mount. Ideal for implementing the External Configuration Store pattern.

List all ConfigMaps: `kubectl get configmap` or `kubectl get cm`

Secret

Functionally identical to ConfigMaps, but stored encoded as base64, and encrypted at rest (if configured).

List all Secrets: `kubectl get secret`

Run Kubernetes locally

For our tutorial we will not build a real **Kubernetes** Cluster. We will use **Minikube**.

Minikube is a tool that makes it easy to run **Kubernetes** locally. Minikube runs a single-node **Kubernetes** cluster inside a VM on your laptop for users looking to try out **Kubernetes** or develop with it day-to-day.

For **Minikube** installation: <https://github.com/kubernetes/minikube>

After the installation, to start Minikube:

```
minikube start
```

The **minikube start** command creates a **kubectl context** called **minikube**. This context contains the configuration to communicate with your minikube cluster.

Minikube sets this context to default automatically, but if you need to switch back to it in the future, run:

```
kubectl config use-context minikube
```

To access the **Kubernetes Dashboard**:

```
minikube dashboard
```

The **Dashboard** will be opened in your default browser:

Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created
kubernetes	default	component:apiserver provider:kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	13 hours ago

Name	Namespace	Labels	Type	Created
default-token-2x8tt	default	-	kubernetes.io/service-account-token	13 hours ago

The `minikube stop` command can be used to stop your cluster. This command shuts down the minikube virtual machine, but preserves all cluster state and data. Starting the cluster again will restore it to its previous state.

The `minikube delete` command can be used to delete your cluster. This command shuts down and deletes the minikube virtual machine. No data or state will be preserved.

To allocate resources for `minikube`, you can do it using these commands:

```
minikube config set memory 8192          ①
minikube config set cpus 4                ②
minikube config set kubernetes-version 1.16.2 ③
minikube config set vm-driver kvm2        ④
minikube config set container-runtime crio ⑤
```

The line defines:

- ① the allocated memory to 8192 Mb = 8Gb
- ② the allocated CPUs to 4
- ③ Kubernetes version to **1.16.2**
- ④ the VM Driver to **kvm2**
- ⑤ the container runtime to use **crio** instead of Docker (default choice)



We need to delete the current minikube cluster instance before setting new configuration.

To check if the configuration is correctly saved, just do `minikube config view`:

```
$ minikube config view

- cpus: 4
- kubernetes-version: 1.16.2
- memory: 8192
- vm-driver: kvm2
- container-runtime: crio
```

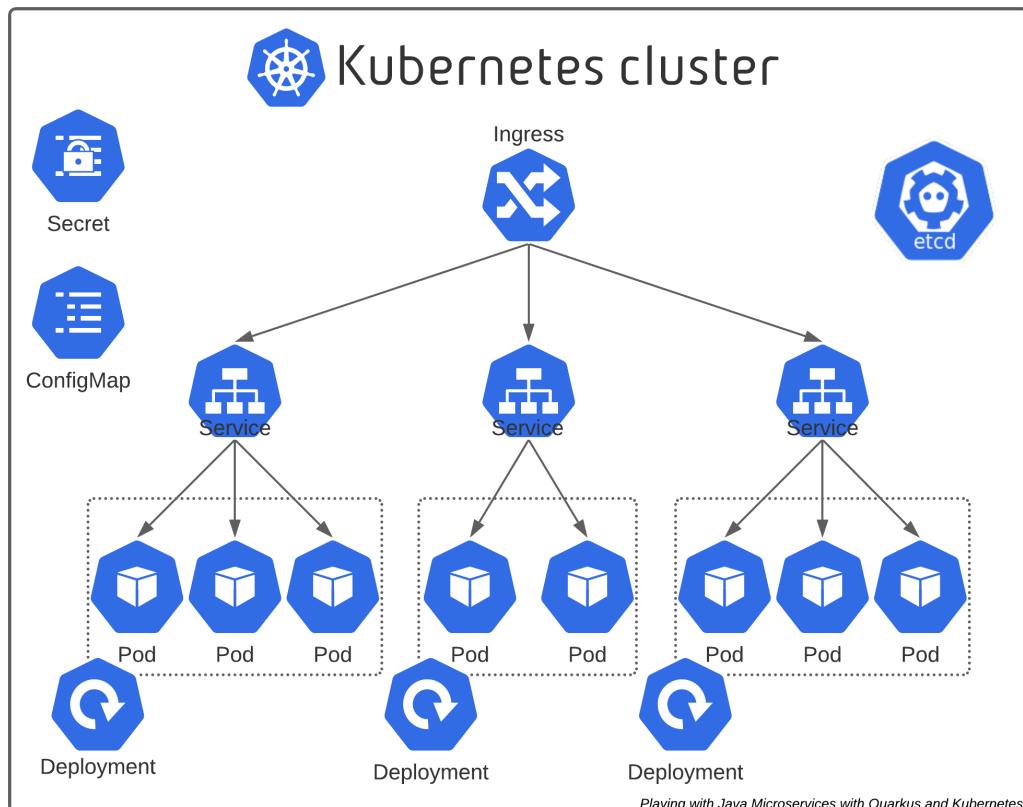
Practical Summary & Conclusion

Kubernetes is the best containers orchestrator available in the market. 😊 Many solutions are now based on it like **OpenShift Container Platform** & **Rancher Kubernetes Engine**. **Kubernetes** became the standard of the *Cloud Native* architecture & infrastructure. Almost all the cloud providers have *Managed Kubernetes hosting*:

- Azure Kubernetes Service
- Amazon Elastic Kubernetes Service
- Google Kubernetes Engine
- IBM Cloud Kubernetes Service
- Oracle Container Engine for Kubernetes
- Alibaba Container Service for Kubernetes

Even **OVH** and **DigitalOcean** have joint the race to provide managed **Kubernetes** hosting solutions 😊

As we listed in the first part of this chapter, there are many **Kubernetes** objects, each of them can be used to satisfy a specific need. In this section we will try to check which of these objects can fit our needs.



QuarkuShop is packaged as a **Docker** container. In **Kubernetes**, our application will be running inside a **POD** object: the most elementary **Kubernetes** object! We will maybe want to have more than one **POD** for a given **UserService**. To avoid managing them manually, we will use the **Deployment** object, which will be handling each **POD** set. It will also make sure that there is the desired number of instances in case that we wanted to have many instances of a specific **POD**.

To store the properties we can use the **ConfigMap** and for storing the credentials, we can use the **SECRET** object. To access them, we will communicate with the **Kubernetes API Server** to get/read the desired data.

When we splitted the monolithic application into **services**, we said that the communication between them will be based on the **HTTP protocol** (directly or indirectly). Each **service** is running inside a **POD** which will have a dedicated **IP address**, which is totally dynamic. So if the **Order service** will communicate with the **Product service**, it cannot guess the **IP address** of its target. The only way is to use DNS-like solution, to use a domain name instead of an **IP address**, and to let the system resolve the domain name dynamically. This is exactly what a K8s **SERVICE** is doing: **Kubernetes** gives **POD** their own **IP addresses** and a **single DNS name** for a set of **PODS**, and **can load-balance across them**.

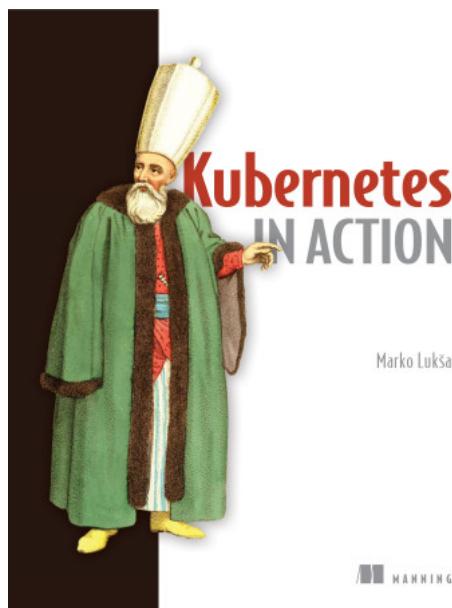


Other than *dynamic IP address* resolution, the **SERVICE** include also the **Load Balancing** feature.

To expose the **QuarkuShop** to outside the cluster, we will use the **INGRESS** object: An API object that manages external access to the services in a cluster, typically **HTTP**. It may also provide **Load Balancing**, **SSL termination** and **Name-based Virtual Hosting**.

Additional reading

For sure one or few chapters cannot cover **Kubernetes**, I just wanted this chapter to be a small introduction to the **Kubernetes** World. I suggest that you read **Kubernetes in Action** written by **Marko Lukša** and published by **Manning**: personally I consider it the best **Kubernetes** book ever made.



If you are more comfortable with videos, I suggest this great [Kubernetes Learning YouTube playlist](#), made by my friend **Houssem Dellai**, Cloud Engineer at Microsoft.

Learning Kubernetes

35 videos • 5,246 views • Last updated on Oct 7, 2020

PLAY ALL

SUBSCRIBE

1 Kubernetes architecture explained Houssem Dellai 15:57

2 From traditional app to Docker and Kubernetes Houssem Dellai 18:03

3 Kubernetes Objects: Deployment | Service | Secret | PersistentVolume Houssem Dellai 19:16

4 Logging with EFK in Kubernetes Houssem Dellai 15:49

5 Installing EFK in Kubernetes Houssem Dellai 19:30

6 Kubernetes Service Catalog Houssem Dellai 30:23

7 Monitoring Kubernetes with Prometheus & Grafana (1/5) Houssem Dellai 14:57

8 Exporting Prometheus metrics in ASP.NET Core (2/5) Houssem Dellai 13:04

CHAPTER ELEVEN

Implementing the Cloud Patterns

Introduction

We already know that we will use **Kubernetes** as our *Cloud platform*. In the previous chapter, we listed the **Kubernetes Objects** that will use for **QuarkuShop**. In this chapter, we will start implementing some *Cloud Patterns* and we will bring the monolithic universe (the monolithic application + PostgreSQL + Keycloak) to **Kubernetes**.

Bringing the Monolithic Universe to Kubernetes

Before starting to work on the application code, we will bring the  **PostgreSQL DB** and **Keycloak** to the **Kubernetes Cluster**.

Deploying PostgreSQL to Kubernetes

To deploy a  **PostgreSQL Database** instance to **Kubernetes**, we will use:

- a **ConfigMap** that stores the PostgreSQL Username and Database Name
- a **Secret** that stores the PostgreSQL Password
- a **PersistentVolumeClaim** that will request a storage space for the PostgreSQL PODs
- a **Deployment** that will bring the description of the wanted the PostgreSQL PODs
- a **Service** that will be used as a DNS name pointing on the PostgreSQL PODs

The **ConfigMap** for PostgreSQL:

postgres-cm.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-config
  labels:
    app: postgres
data:
  POSTGRES_DB: demo
  POSTGRES_USER: developer
```

The PostgreSQL password will be stored in a **Secret**. The value needs to be encoded as **Base64**. You can encode the **p4SSW0rd** string using the **openssl** library locally:

```
echo -n 'p4SSW0rd' | openssl base64
```

The result will be: `cDRTU1cwcmQ=` - we will use it inside our `Secret` object:

`postgres-secret.yaml`

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
  labels:
    app: postgres
type: Opaque
data:
  POSTGRES_PASSWORD: cDRTU1cwcmQ=
```

The `PersistentVolumeClaim` that will be used to ask storage access for PostgreSQL:

`postgres-pvc.yaml`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
  labels:
    app: postgres
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
```

The PostgreSQL Deployment file:

postgres-deployment.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: postgres-pvc
      containers:
        - name: postgres
          image: postgres:12.3
          envFrom:
            - configMapRef:
                name: postgres-config
            - secretRef:
                name: postgres-secret
          ports:
            - containerPort: 5432
      volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
          subPath: postgres
      resources:
        requests:
          memory: '512Mi'
          cpu: '500m'
        limits:
          memory: '1Gi'
          cpu: '1'

```

The PostgreSQL Service file looks like:

postgres-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  labels:
    app: postgres
spec:
  selector:
    app: postgres
  ports:
    - port: 5432
  type: LoadBalancer
```

Now, we will test the deployed **PostgreSQL** instance using **IntelliJ Database Explorer**.

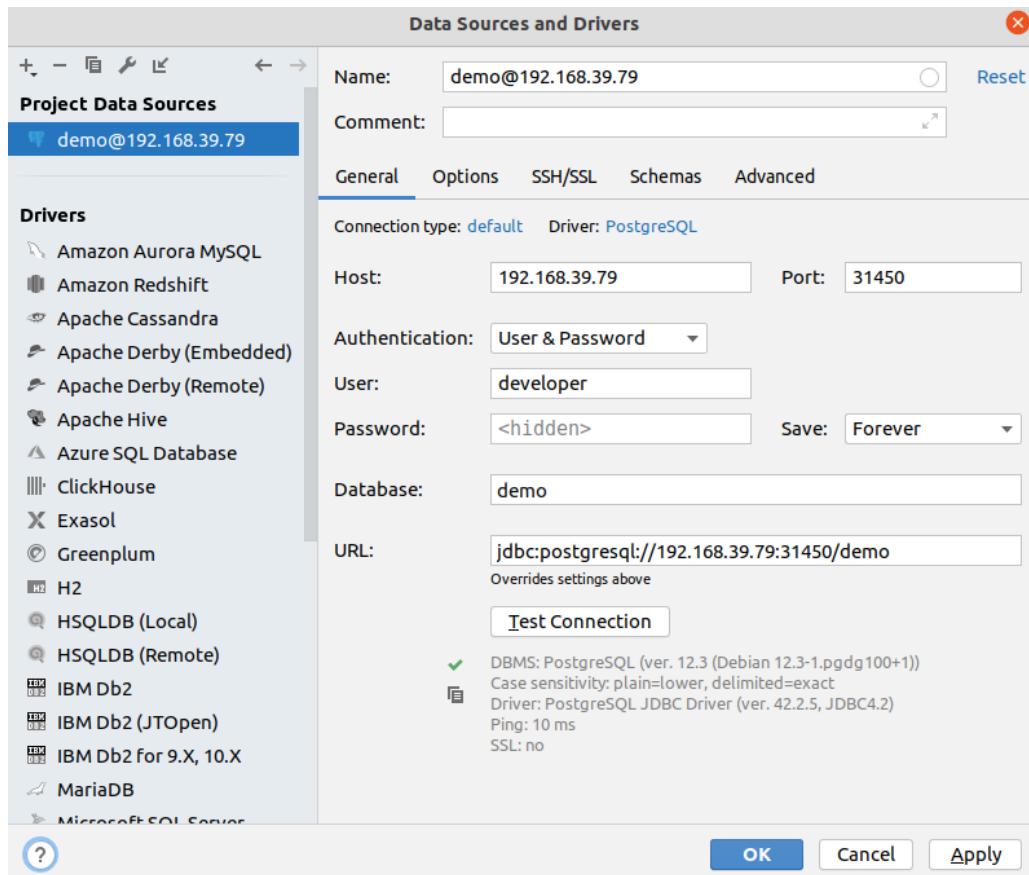
We will start by getting the **PostgreSQL Kubernetes Service URL**: we will use this command to get the Kubernetes Service URL from the **Minikube** cluster:

```
$ minikube service postgres --url
http://192.168.39.79:31450
```

We will use :

- 192.168.39.79:31450 as *Database URL*
- developer as *User*
- p4SSW0rd as *Password*
- demo as *Database name*

Just hit [Test Connection] to verify that everything is ok. If ok you will get a ✓ that confirms that the connection was successful.



Deploying Keycloak to Kubernetes

For deploying Keycloak to our Kubernetes Cluster, we will use **Helm**.

What is Kubernetes Helm?

Helm is a package manager for **Kubernetes** that allows developers and operators to more easily package, configure, and deploy applications and services onto **Kubernetes** clusters.

Helm is now an official **Kubernetes** project and is part of the **Cloud Native Computing Foundation**, a non-profit that supports open source projects in and around the **Kubernetes** ecosystem.

Helm can:

- Install software.
- Automatically install software dependencies.
- Upgrade software.
- Configure software deployments.
- Fetch software packages from repositories.

Helm provides this functionality through the following components:

- A command line tool, **helm**, which provides the user interface to all **Helm** functionality.
- A companion server component, **tiller**, that runs on your **Kubernetes** cluster, listens for commands from **helm**, and handles the configuration and deployment of software releases on the cluster.
- The **Helm** packaging format is called **charts**.

What is Helm Chart ?

Helm packages are called **charts**, and they consist of a few **YAML** configuration files and some templates that are rendered into **Kubernetes** manifest files.

The **helm** command can install a **chart** from a local directory, or from a **.tar.gz** packaged version of this directory structure. These packaged **charts** can also be automatically downloaded and installed from **chart** repositories or repos.

To install the **Helm CLI**, just go to this guide: helm.sh/docs/intro/install

After installing the **Helm CLI**, we will proceed to install the Keycloak **Helm Chart** into our **Kubernetes Cluster**.

The **Keycloak Helm Chart** that we wish to install is available in the **codecentric** repository, so we need to add it in the **Helm Repositories**:

```
helm repo add codecentric https://codecentric.github.io/helm-charts
```

Next, we will create a **Kubernetes Namespace** where we will install **Keycloak**:

```
$ kubectl create namespace keycloak
namespace/keycloak created
```

Next, we will install **Keycloak** using this **Helm** command:

```
helm install keycloak --namespace keycloak codecentric/keycloak
```

To list the created objects in the **keycloak** namespace using the command **kubectl get all -n keycloak**:

NAME	READY	STATUS	RESTARTS
pod/keycloak-0	1/1	Running	0
pod/keycloak-postgresql-0	1/1	Running	0

NAME	TYPE	CLUSTER-IP	PORT(S)
service/keycloak-headless	ClusterIP	None	80/TCP
service/keycloak-http	ClusterIP	10.96.104.85	80/TCP 8443/TCP
9990/TCP			
service/keycloak-postgresql	ClusterIP	10.107.175.90	5432/TCP
service/keycloak-postgresql-headless	ClusterIP	None	5432/TCP

NAME	READY
statefulset.apps/keycloak	1/1
statefulset.apps/keycloak-postgresql	1/1

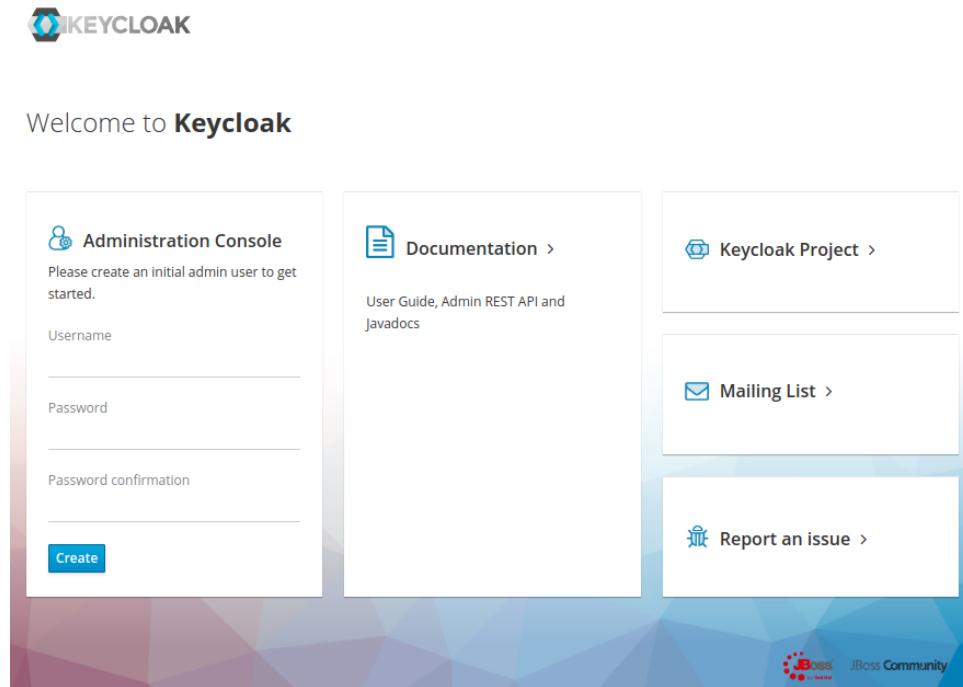
You can notice that we have two **PODs**:

- **keycloak-0**: contains the **Keycloak** application
- **keycloak-postgresql-0**: contains a dedicated **PostgreSQL Database** instance for the **Keycloak** application

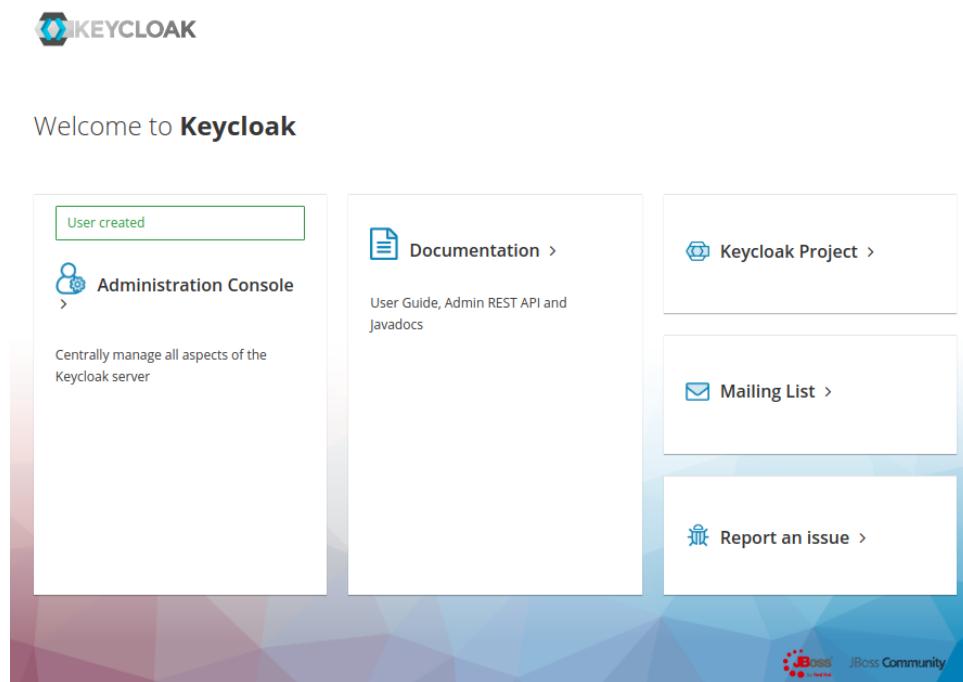
Now, we need to configure the **Keycloak** instance as we did before in *Chapter 5*. To access the **Keycloak POD**, we will use the **port-forwarding** from the **POD** to the **localhost**:

```
kubectl -n keycloak port-forward service/keycloak-http 8080:80
```

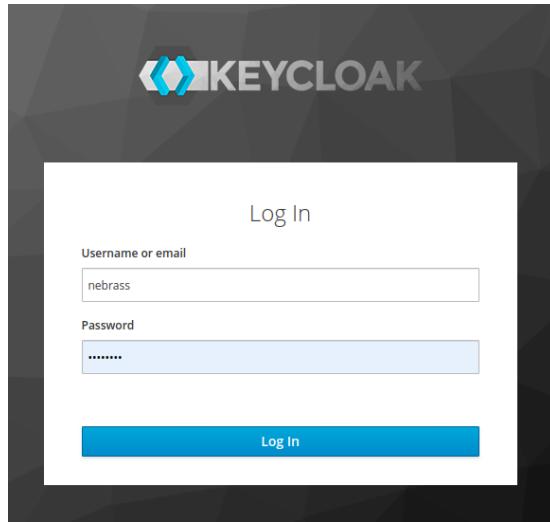
Now, we will access the **Keycloak Console** on: localhost:8080/auth:



In this screen, we will create the **Administrator** user:



Then, click on the **[Administration Console]** to access the login screen:



After login, we will land to the **Keycloak Administration Console**:

A screenshot of the Keycloak administration console. The left sidebar shows navigation options like "Master", "Configure", "Realm Settings" (which is selected), and "Manage". The main content area is titled "Master" and shows the "General" tab selected. It displays configuration for the "master" realm, including fields for "Name" (master), "Display name" (Keycloak), "HTML Display name" (<div class="kc-logo-text">Keycloak</div>), "Frontend URL" (empty), "Enabled" (ON), "User-Managed Access" (OFF), and "Endpoints" (OpenID Endpoint Configuration, SAML 2.0 Identity Provider Metadata). There are "Save" and "Cancel" buttons at the bottom.

Then, we can now apply the same steps as *Chapter 5* to create the **Keycloak Realm**.

Deploying the monolithic QuarkuShop to Kubernetes

After bringing the **PostgreSQL DB** and **Keycloak** to **Kubernetes**, we can now proceed to settling the monolithic **QuarkuShop** application to **Kubernetes**. This exercise is very useful to learn how to deploy only one application to **Kubernetes**.

We will start by adding two dependencies to the **QuarkusShop pom.xml**:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-kubernetes</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-container-image-jib</artifactId>
</dependency>
```

When building the project using **Maven**, these two libraries will:

- Create the **Docker Image** for the application
- Generate the **Kubernetes Objects Descriptors** required for deploying the application

To verify that I'm telling the truth ☺ Let's build the application and check:

```
$ mvn clean install -Dquarkus.container-image.build=true
```

In the **target** directory, there are two new folders **jib** and **kubernetes**. To check their content:

```
$ ls -l target/jib target/kubernetes

target/jib:          ①
total 32
-rw-rw-r-- 1 nebrass nebrass 1160 sept. 8 21:01 application.properties
-rw-rw-r-- 1 nebrass nebrass 427 sept. 8 21:01 banner.txt
drwxrwxr-x 3 nebrass nebrass 4096 sept. 8 21:01 com
drwxrwxr-x 3 nebrass nebrass 4096 sept. 8 21:01 db
drwxrwxr-x 7 nebrass nebrass 4096 sept. 8 21:01 io
drwxrwxr-x 7 nebrass nebrass 4096 sept. 8 21:01 javax
drwxrwxr-x 7 nebrass nebrass 4096 sept. 8 21:01 META-INF
drwxrwxr-x 4 nebrass nebrass 4096 sept. 8 21:01 org

target/kubernetes: ②
total 12
-rw-rw-r-- 1 nebrass nebrass 5113 sept. 8 21:01 kubernetes.json
-rw-rw-r-- 1 nebrass nebrass 3478 sept. 8 21:01 kubernetes.yml
```

① directory used by **JIB** while generating the **Docker Image**

② directory used to store the generated **Kubernetes** descriptors - the same content is generated in two formats **YAML** and **JSON**

If we want to build the Docker Image based on the Native Binary:

```
mvn clean install -Pnative -Dquarkus.native.container-build=true -Dquarkus.container-image.build=true
```

We can verify also that there is a newly created **Docker Image** locally:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nebrass/quarkushop	1.0.0-SNAPSHOT	eb2c67d7fa27	21 minutes ago	244MB

Now, we will push this image to **Docker Hub**:

```
$ docker push nebrass/quarkushop:1.0.0-SNAPSHOT
```

Before importing the **Kubernetes** descriptor, we need to make a small modification: Our application currently is pointing on a *LOCAL PostgreSQL* and **Keycloak** instances. We have two possibilities:

- change the hardcoded properties
- override the properties using the environment variables  this is the best solution as it doesn't need any new build or release

To override the properties we will use environment variables. For example, to override the value of the **quarkus.http.port** property to **9999**, we will create an environment variable called **QUARKUS_HTTP_PORT=9999**. As you can notice it's the same capitalized string with an underscore instead of the separating dot **.** In our case, we want to override these properties:

- **quarkus.datasource.jdbc.url** which is pointing on the **PostgreSQL** URL its corresponding environment variable will be **QUARKUS_DATASOURCE_JDBC_URL**
- **mp.jwt.verify.publickey.location** which is pointing on the **Keycloak** URL its corresponding environment variable will be **MP_JWT_VERIFY_PUBLICKEY_LOCATION**
- **mp.jwt.verify.issuer** which is pointing on the **Keycloak** URL its corresponding environment variable will be **MP_JWT_VERIFY_ISSUER**

These three properties are pointing to **localhost** as **Host** for the PostgreSQL and Keycloak instances. We want them now to point on the **PostgreSQL** and **Keycloak** respective **PODs**.

As we said in the previous chapter, we have the **K8s Service** objects that can be used as DNS Name for each set of **PODs**.

Let's list the available **Service** objects that we already have in our cluster:

```
$ kubectl get svc --all-namespaces
```

NAMESPACE	NAME	PORT(S)	
default	kubernetes	443/TCP	
default	postgres	5432:31450/TCP	①
keycloak	keycloak-headless	80/TCP	②
keycloak	keycloak-http	80/TCP, 8443/TCP, 9990/TCP	②
keycloak	keycloak-postgresql	5432/TCP	②
keycloak	keycloak-postgresql-headless	5432/TCP	②
kube-system	kube-dns	53/UDP, 53/TCP, 9153/TCP	
kubernetes-dashboard	dashboard-metrics-scraper	8000/TCP	
kubernetes-dashboard	kubernetes-dashboard	80/TCP	

① The **PostgreSQL Service** is available in the **default** namespace.

② We have several **Keycloak Service** objects in the **keycloak** namespace:

- **keycloak-headless**: a **Headless Kubernetes Service** pointing to **Keycloak PODs**. A **Headless service** is used to enable a direct communication to the matching **PODs** without any intermediate layer - there is no **proxy** or any routing layer involved in the communication. A **Headless Service** is listing all the selected backing **PODs** while the other **Service** types are forwarding the calls to a **RANDOMLY selected POD**.
- **keycloak-http**: **Kubernetes Service** which offers a **LoadBalancing** Router to **Keycloak PODs**
- **keycloak-postgresql**: **Kubernetes Service** which offers a **LoadBalancing** Router to **PostgreSQL PODs**
- **keycloak-postgresql-headless**: **Headless Kubernetes Service** pointing to **PostgreSQL PODs**

In our case we will not be using the **Headless Service** as we want to have **Requests LoadBalancing** and we don't have any known (yet) need to communicate directly with the **PODs**.

Actually the `quarkus.datasource.jdbc.url` property contains `jdbc:postgresql://localhost:5432/demo`. We will use **postgres Service** with port **5432** from the **default** namespace, instead of **localhost**. So the new property definition will be:

```
quarkus.datasource.jdbc.url=jdbc:postgresql://postgres:5432/demo
```

The environment variable will be:

```
QUARKUS_DATASOURCE_JDBC_URL=jdbc:postgresql://postgres:5432/demo
```

Good! We will apply the same logic for the `mp.jwt.verify.publickey.location` and `mp.jwt.verify.issuer` properties.

```
1 mp.jwt.verify.publickey.location=http://localhost:9080/auth/realm/quarkushop-realm/protocol/openid-connect/certs
2 mp.jwt.verify.issuer=http://localhost:9080/auth/realm/quarkushop-realm
```

Instead of the `localhost` we will use the `keycloak-http` with the port `80` from the `keycloak` namespace. The new `properties` values will be:

```
1 mp.jwt.verify.publickey.location=http://keycloak-http.keycloak/auth/realm/quarkushop-realm/protocol/openid-connect/certs
2 mp.jwt.verify.issuer=http://keycloak-http.keycloak/auth/realm/quarkushop-realm
```



You can notice that for **VISIBILITY** we appended the `namespace name` after the `service name` for the **Keycloak Services**, while we didn't do that for the **PostgreSQL Service**. This is due to the presence of the **PostgreSQL Service** inside the `default` namespace, where we will deploy our application, while the **Keycloak Services** are available inside the `keycloak` namespace.

The **MP JWT** environment variables will be:

```
1 MP_JWT_VERIFY_PUBLICKEY_LOCATION=http://keycloak-http.keycloak/auth/realm/quarkushop-realm/protocol/openid-connect/certs
2 MP_JWT_VERIFY_ISSUER=http://keycloak-http.keycloak/auth/realm/quarkushop-realm
```

Now we need to add these new environment variables to the Kubernetes descriptor, that was generated by the **Quarkus Kubernetes Extension** inside the `target/kubernetes/` directory.

In the Kubernetes world, if we want to pass environment variables to a **POD**, we pass them using the **Kubernetes Deployment Object**, in the `spec.template.spec.containers.env` section.

The updated **Deployment** object in the **Kubernetes.json** descriptor file will look like:

target/kubernetes/kubernetes.json

```
{
  "apiVersion": "apps/v1",
  "kind": "Deployment",
  "metadata": { ...,
    "name": "quarkushop"
  },
  "spec": {
    "replicas": 1,
    "selector": ...,
    "template": {
      "metadata": {
        "annotations": ...,
        "labels": {
          "app.kubernetes.io/name": "quarkushop",
          "app.kubernetes.io/version": "1.0.0-SNAPSHOT"
        }
      },
      "spec": {
        "containers": [
          {
            "env": [
              { "name": "KUBERNETES_NAMESPACE",
                "valueFrom": {
                  "fieldRef": { "fieldPath": "metadata.namespace" }
                }
              },
              { "name": "QUARKUS_DATASOURCE_JDBC_URL",
                "value": "jdbc:postgresql://postgres:5432/demo"
              },
              { "name": "MP_JWT_VERIFY_PUBLICKEY_LOCATION",
                "value": "http://keycloak-http.keycloak/auth/realms/quarkushop-
realm/protocol/openid-connect/certs"
              },
              { "name": "MP_JWT_VERIFY_ISSUER",
                "value": "http://keycloak-http.keycloak/auth/realms/quarkushop-realm"
              }
            ],
            "image": "nebrass/quarkushop:1.0.0-SNAPSHOT",
            "imagePullPolicy": "IfNotPresent",
            ...
          }
        ]
      }
    }
  }
}
```

It's annoying method to add these values each time, especially that the target folder will be deleted each time we build the code. This is why the **Quarkus** team thought of adding a new mechanism of environment variables definition for **Kubernetes** descriptors.

We can add the same environment variables using the **application.properties** file with the prefix **quarkus.kubernetes.env-vars.** for each env variable with **.value** as suffix for each environment variable:

```
1 quarkus.kubernetes.env-vars.QUARKUS-DATASOURCE-JDBC-URL.value
= jdbc:postgresql://postgres:5432/demo
2 quarkus.kubernetes.env-vars.MP-JWT-VERIFY-PUBLICKEY-LOCATION.value=http://keycloak-
http.keycloak/auth/realms/quarkushop-realm/protocol/openid-connect/certs
3 quarkus.kubernetes.env-vars.MP-JWT-VERIFY-ISSUER.value=http://keycloak-
http.keycloak/auth/realms/quarkushop-realm
```

To import the generated **Kubernetes** descriptor:

```
$ kubectl apply -f target/kubernetes/kubernetes.json

serviceaccount/quarkushop created
service/quarkushop created
deployment.apps/quarkushop created
```

To verify that **QuarkuShop** is correctly deployed to **Kubernetes**, just list all the objects in the current (**default**) namespace:

```
$ kubectl get all

NAME                               READY   STATUS    RESTARTS
pod/postgres-69c47c748-pnbbf      1/1     Running   3
pod/quarkushop-78c67844ff-7fzv   1/1     Running   0

NAME                  TYPE        CLUSTER-IP   PORT(S)
service/kubernetes   ClusterIP   10.96.0.1   443/TCP
service/postgres     LoadBalancer 10.106.7.15  5432:31450/TCP
service/quarkushop   ClusterIP   10.97.230.13 8080/TCP

NAME                  READY   UP-TO-DATE   AVAILABLE
deployment.apps/postgres  1/1     1           1
deployment.apps/quarkushop 1/1     1           1

NAME                  DESIRED  CURRENT   READY
replicaset.apps/postgres-69c47c748  1        1         1
replicaset.apps/quarkushop-78c67844ff 1        1         1
```

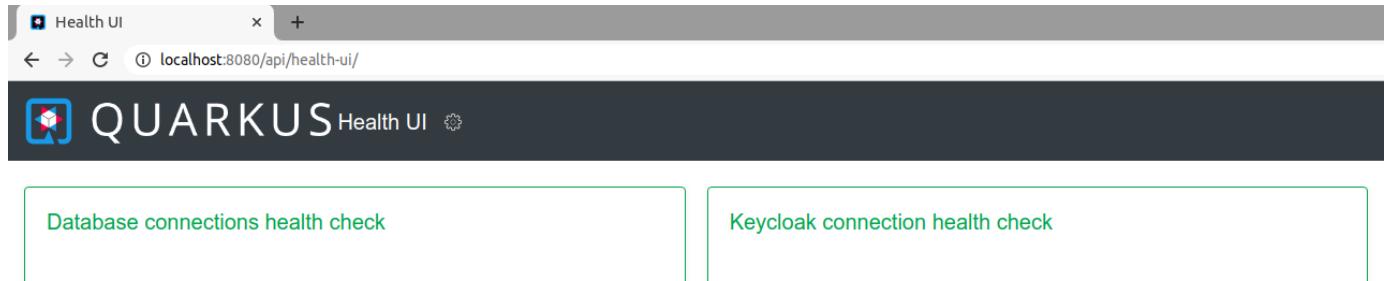
In the resource list, there is a **POD** called **quarkushop-78c67844ff-7fzbv**. To check that everything is **OK**, we need to access the application packaged inside it. To do that, we will do a **port-forward** from the port **8080** of the **quarkushop-78c67844ff-7fzbv POD** to the **8080** of the **localhost**.

```
$ kubectl port-forward quarkushop-78c67844ff-7fzbv 8080:8080
```

Forwarding from 127.0.0.1:8080 -> 8080

Forwarding from [::1]:8080 -> 8080

Now open the <http://localhost:8080/api/health-ui>:



We can see that the **Health Checks** verified that the **PostgreSQL Database** and **Keycloak** are reachable.

There is an other alternative to provide addional **properties** to the application using the **ConfigMaps**. To do it, we will need an **OTHER** Maven Dependency 😊

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-kubernetes-config</artifactId>
</dependency>
```

We will need to remove the **ENV VARS properties** that we added before:

```
quarkus.kubernetes.env-vars.QUARKUS-DATASOURCE-JDBC-URL.value
= jdbc:postgresql://postgres:...
quarkus.kubernetes.env-vars.MP-JWT-VERIFY-PUBLICKEY-LOCATION.value=http://keycloak-
http.k...
quarkus.kubernetes.env-vars.MP-JWT-VERIFY-ISSUER.value=http://keycloak-http.keycloak/a...
```

Next, we will need to enable the **Kubernetes ConfigMap** access and to tell the application which **ConfigMap** has our needed properties:

```
quarkus.kubernetes-config.enabled=true
quarkus.kubernetes-config.config-maps=quarkushop-monolith-config
```

Now, we need to define our new **ConfigMap** called **quarkushop-monolith-config**:

quarkushop-monolith-config.yml

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: quarkushop-monolith-config
5 data:
6   application.properties: |-
7     quarkus.datasource.jdbc.url=jdbc:postgresql://postgres:5432/demo
8     mp.jwt.verify.publickey.location=http://keycloak-
9       http.keycloak/auth/realm/quarkushop-realm/protocol/openid-connect/certs
9     mp.jwt.verify.issuer=http://keycloak-http.keycloak/auth/realm/quarkushop-realm
```

We need just to import the **quarkushop-monolith-config.yml** to our **Kubernetes Cluster**:

```
kubectl apply -f quarkushop-monolith-config.yml
```

Now, if we build the **QuarkuShop** application again, we will notice that in the generated **Kubernetes** descriptors, there is a new **RoleBinding** object ⓘ This object is generated by the **quarkus-kubernetes-config Quarkus Extension**.

```
{
  "apiVersion" : "rbac.authorization.k8s.io/v1",
  "kind" : "RoleBinding",                                     ①
  "metadata" : {
    "annotations" : {
      "prometheus.io/path" : "/metrics",
      "prometheus.io/port" : "8080",
      "prometheus.io/scrape" : "true"
    },
    "labels" : {
      "app.kubernetes.io/name" : "quarkushop",
      "app.kubernetes.io/version" : "1.0.0-SNAPSHOT"
    },
    "name" : "quarkushop:view"                                ②
  },
  "roleRef" : {
    "kind" : "ClusterRole",                                    ③
    "apiGroup" : "rbac.authorization.k8s.io",
    "name" : "view"
  },
  "subjects" : [ {
    "kind" : "ServiceAccount",                               ④
    "name" : "quarkushop"                                    ④
  } ]
}
```

① The current object is **RoleBinding**

② The **RoleBinding** object name is **quarkushop:view**

③ This will bind the **ClusterRole** with the **view** role just to be able to read **ConfigMaps** and **Secrets**

④ This **RoleBinding** will be applied on the **ServiceAccount** called **quarkushop**, which is generated by the **quarkus-kubernetes Extension**.

Let's build and package the application image and deploy it again to the **K8s Cluster**:

```
mvn clean install -DskipTests -DskipITs -Pnative \
-Dquarkus.native.container-build=true \
-Dquarkus.container-image.build=true
```

Next, push the image:

```
docker push nebrass/quarkushop:1.0.0-SNAPSHOT
```

Next, deploy the application again to Kubernetes:

```
kubectl apply -f target/kubernetes/kubernetes.json
```

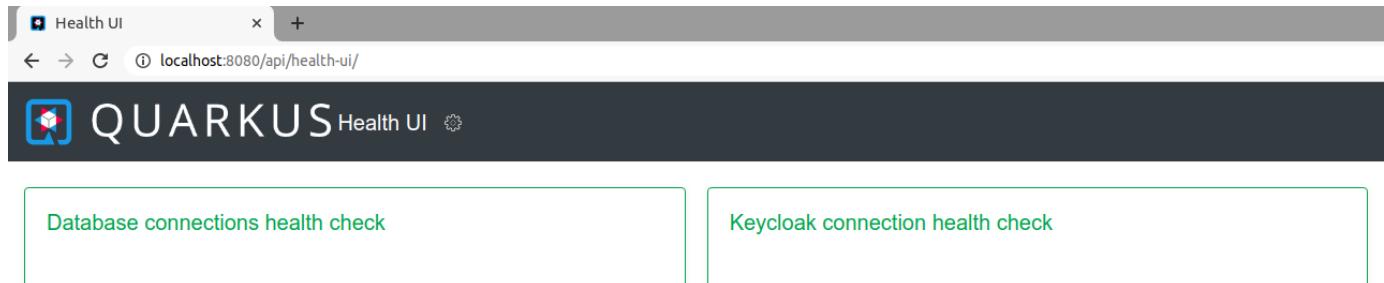
To test the application now just do a **port-forward** on the **QuarkuShop POD**:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
postgres-69c47c748-pnbbf	1/1	Running	5	19d
quarkushop-77dcfc7c45-tzmbs	1/1	Running	0	73m

```
$ kubectl port-forward quarkushop-77dcfc7c45-tzmbs 8080:8080
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

Then open the <http://localhost:8080/api/health-ui/>:



Excellent! We have landed now our monolithic application into **Kubernetes** In the next chapter, we will be doing the same steps to create and land our **microservices** in our **Kubernetes Cluster**.

Conclusion

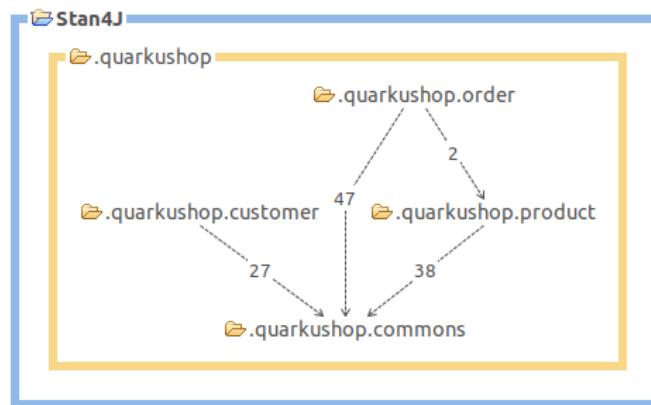
In this chapter, I tried to introduce some **Cloud Patterns** based on the **Kubernetes** ecosystem. The exercise of bringing the monolithic application and its dependencies into Kubernetes is the first step while creating and deploying our **microservices**.

CHAPTER TWELVE

Building the Kuberntized
Microservices

Introduction

Previously, in chapter 8, we applied the DDD to the monolithic application. We broke the relations between the Boundary Contexts that we revealed while analyzing the project. In Stan4J, the final code structure looks like:



In this chapter, we will implement the three µservices **product**, **order** and **customer**. As you can notice, all the packages depend on the **commons** package. So, we need to start to implement it before implementing the three µservices.

Creating the Commons Library

The **commons** JAR will be the library that will be wrapping all the content of the **commons** package.

We will generate a simple Maven Project, where we will copy all the content of the **commons** package.

```
mvn archetype:generate -DgroupId=com.targa.labs.commons -DartifactId=quarkushop-commons \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.4 -DinteractiveMode=false
```

This command will generate a simple project with a content:

```
project
|-- pom.xml
\-- src
    |-- main/java
    |    '-- com.targa.labs.common
    |         '-- App.java
    '-- test/java
        '-- com.targa.labs.common
            '-- AppTest.java
```



We will remove the **App.java** and **AppTest.java** as we will not need them.

Then, we will copy/paste the content of the `commons` package from the monolithic application to our `quarkushop-commons` project.

Don't be scared! I know that there are many errors and warnings that appeared when you pasted the copied classes 😬 our next step is to add the missing dependencies to make the IDE happy 😊

Let's open the `pom.xml` and start making the changes:

1. We will start by changing the `maven.compiler.source` and `maven.compiler.target` from **1.7** to **11**
2. We will define the dependencies as:

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.30</version>
    </dependency>
    <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>2.0.1.Final</version>
    </dependency>
    <dependency>
        <groupId>org.eclipse.microprofile.openapi</groupId>
        <artifactId>microprofile-openapi-api</artifactId>
        <version>1.1.2</version>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javaee.ws.rs</groupId>
        <artifactId>jboss-jaxrs-api_2.1_spec</artifactId>
        <version>2.0.1.Final</version>
    </dependency>
    <dependency>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>2.2.3</version>
    </dependency>
    <dependency>
        <groupId>jakarta.enterprise</groupId>
        <artifactId>jakarta.enterprise.cdi-api</artifactId>
        <version>2.0.2</version>
    </dependency>
    <dependency>
        <groupId>org.eclipse.microprofile.health</groupId>
        <artifactId>microprofile-health-api</artifactId>
        <version>2.2</version>
    </dependency>
```

```

<dependency>
    <groupId>org.eclipse.microprofile.config</groupId>
    <artifactId>microprofile-config-api</artifactId>
    <version>1.4</version>
</dependency>
<dependency>
    <groupId>org.eclipse.microprofile.metrics</groupId>
    <artifactId>microprofile-metrics-api</artifactId>
    <version>2.3</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.11.2</version>
</dependency>
<dependency>
    <groupId>io.quarkus.security</groupId>
    <artifactId>quarkus-security</artifactId>
    <version>1.1.2.Final</version>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.14.3</version>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-test-common</artifactId>
    <version>1.8.2.Final</version>
</dependency>
</dependencies>

```

Wow! 😮 From where did you get these dependencies?? I'm sure that you are like me, I refuse to write things that I don't understand 😬 Keep calm! I will tell you from where they came?

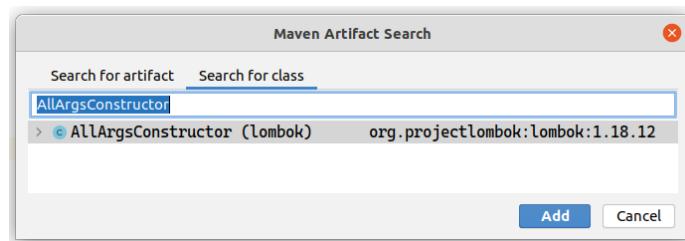
All these dependencies are those of the **Quarkus** framework. I used the IDE to add the missing dependency:

```

11  /**
12   * @author Nebrass Lamouchi
13  */
14  @Data
15  @NoArgsConstructor
16  @AllArgsConstructor
17  public class OrderDto {
18      private L...
19      private B...
20      private S...
21      private Z...
22      private L...
23      private AddressDto shipmentAddress;
24      private Set<OrderItemDto> orderItems;
25      private CartDto cart;
26  }
27

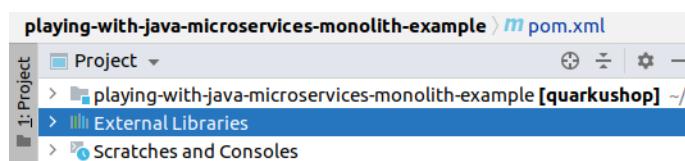
```

Then, the **Maven Artifact Search** window will appear:

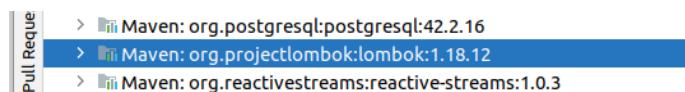


You will be asking yourself which version of a given dependency will you choose?

The answer is easy, we will be using IntelliJ to check which External Libraries are used in the monolithic application:



Expand the section and scroll to find the desired library and the version will be show after the **groupId** and the **artifactId**:



Here we see that the monolithic application is using the **Lombok v1.18.12**, so in the **Commons** project we need to select the same version.



Be sure that the **quarkus-test-common** dependency has the same Quarkus version as the **userservices** in order to avoid conflicts 😊

Finally, we need to build the Maven project using: `mvn clean install` which will build the JAR and will make it available in the local `.m2` directory. This will make possible to use it as a dependency in the future steps.

Wait! It's not finished yet! We need to think about the **TESTs!** Yes! We need to copy the **utils** package from the **Test classes** to the **Main classes** of the **quarkushop-commons** project.

But why are we moving in to `main`, while they belong to the **Tests**? The answer is simple: to be able to reuse these classes outside the **quarkushop-commons** library, we need to have them in the **main directory**, as any other ordinary class. The classes belonging to the **test directory** are just for testing purposes, and they are not intended to be reused.

Implementing the Product µservice

Now, we will start to do the serious job: we will create the **Product** µservice. Let's generate a new **Quarkus** application called **quarkushop-product** from the [code.quarkus.io](#):

The screenshot shows the Quarkus application generator interface. At the top, it displays "QUARKUS 1.8.2.Final". Below that, the "Configure your application details" section shows the Group as "com.targa.labs", Artifact as "quarkushop-product", Version as "1.0.0-SNAPSHOT", and Build Tool as "Maven". There are buttons for "Back to quarkus.io" and "Available with Enterprise Support". A "CLOSE" button and a "Generate your application (alt + ⌘)" button are also present.

The main area is titled "Pick your extensions". On the left, there's a search bar with "RESTEasy, Hibernate ORM, Web..." and a sidebar with categories like RESTEasy, SmallRye, Flyway, and Kubernetes. On the right, under the "Web" category, a list of extensions is shown:

Extension	Description	Actions
<input checked="" type="checkbox"/> RESTEasy JAX-RS	REST endpoint framework implementing JAX-RS and more	⋮
<input checked="" type="checkbox"/> RESTEasy JSON-B	JSON-B serialization support for RESTEasy	⋮
<input type="checkbox"/> RESTEasy Jackson	Jackson serialization support for RESTEasy	⋮
<input checked="" type="checkbox"/> Hibernate Validator	Validate object properties (field, getter) and method parameters fo...	⋮
<input type="checkbox"/> REST Client	Call REST services	⋮
<input type="checkbox"/> REST Client JAXB	Enable XML serialization for the REST Client	⋮
<input type="checkbox"/> REST Client JSON-B	Enable JSON-B serialization for the REST client	⋮
<input type="checkbox"/> REST Client Jackson	Enable Jackson serialization for the REST Client	⋮
<input type="checkbox"/> REST resources for Hibernate ORM with P...	EXPERIMENTAL Generate JAX-RS resources for your Hibernate Panache entities an...	⋮
<input type="checkbox"/> RESTEasy JAXB	XML serialization support for RESTEasy	⋮
<input type="checkbox"/> RESTEasy Multipart	Multipart support for RESTEasy	⋮
<input type="checkbox"/> RESTEasy Mutiny	Mutiny support for RESTEasy	⋮
<input type="checkbox"/> RESTEasy Quie	EXPERIMENTAL Quie Templating integration for RESTEasy	⋮
<input type="checkbox"/> Reactive Routes	REST framework offering the route model to define non blocking e...	⋮
<input type="checkbox"/> SmallRye GraphQL	PREVIEW Create GraphQL Endpoints using the code-first approach from Mic...	⋮
<input checked="" type="checkbox"/> SmallRye JWT	Secure your applications with JSON Web Token	⋮
<input checked="" type="checkbox"/> SmallRye OpenAPI	Document your REST APIs with OpenAPI - comes with Swagger UI	⋮

The list of selected extensions:

- RESTEasy JAX-RS
- RESTEasy JSON-B
- SmallRye OpenAPI
- Hibernate ORM
- Hibernate Validator
- JDBC Driver - PostgreSQL
- Flyway
- Quarkus Extension for Spring Data JPA API
- SmallRye JWT

- SmallRye Health
- SmallRye Metrics
- Kubernetes
- Kubernetes Config
- Container Image Jib

Now, download the generated application skull by clicking on [\[Generate your application\]](#).

Then import the code to your **IDE** and open the **pom.xml** and add the **Lombok** and **TestContainers** dependencies:

```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>

    <dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>1.14.3</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>postgresql</artifactId>
        <version>1.14.3</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.assertj</groupId>
        <artifactId>assertj-core</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Next, we will add the most important dependency ☺ the **quarkushop-commons**:

```
<dependency>
    <groupId>com.targa.labs</groupId>
    <artifactId>quarkushop-commons</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

Then, we will copy the code from the `product` package to the `quarkushop-product` `μservice`.



Don't forget to copy the `banner.txt` file from the **monolithic application** to the `src/main/resources` directory of the **Product μservice**.

The next step is to populate the `application.properties`:

```

1 # Datasource config properties
2 quarkus.datasource.db-kind=postgresql
3 quarkus.datasource.username=developer
4 quarkus.datasource.password=p4SSW0rd
5 quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/product
6 # Flyway minimal config properties
7 quarkus.flyway.migrate-at-start=true
8 # HTTP config properties
9 quarkus.http.root-path=/api
10 quarkus.http.access-log.enabled=true
11 %prod.quarkus.http.access-log.enabled=false
12 # Swagger UI
13 quarkus.swagger-ui.always-include=true
14 # Datasource config properties
15 %test.quarkus.datasource.db-kind=postgresql
16 # Flyway minimal config properties
17 %test.quarkus.flyway.migrate-at-start=true
18 # Define the custom banner
19 quarkus.banner.path=banner.txt
20 ### Security
21 quarkus.http.cors=true
22 quarkus.smallrye-jwt.enabled=true
23 # Keycloak Configuration
24 keycloak.credentials.client-id=quarkushop
25 # MP-JWT Config
26 mp.jwt.verify.publickey.location=http://localhost:9080/auth/realm/quarkushop-
    realm/protocol/openid-connect/certs
27 mp.jwt.verify.issuer=http://localhost:9080/auth/realm/quarkushop-realm
28 ### Health Check
29 quarkus.smallrye-health.ui.always-include=true
30 ### Metrics Monitoring
31 quarkus.smallrye-metrics.micrometer.compatibility=true
32 # Kubernetes ConfigMaps
33 quarkus.kubernetes-config.enabled=true
34 quarkus.kubernetes-config.config-maps=quarkushop-product-config

```

The properties are almost the same of the monolithic application, which is quite logical as the `μservice` is a slice cut from the monolithic application.



I changed the names of the DB (line 5) and the ConfigMap (line 34) !

We said already that we changed the **ConfigMap**, so we need to create it:

quarkushop-product-config.yml

```

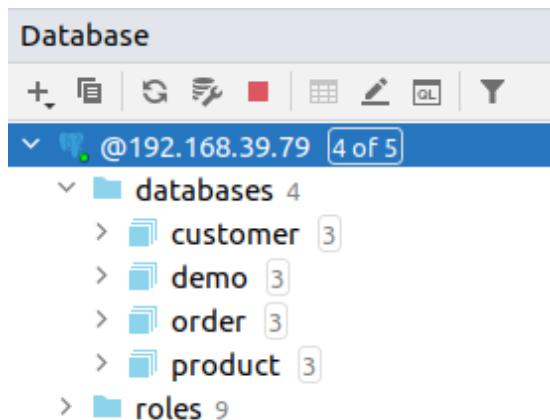
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: quarkushop-product-config
5 data:
6   application.properties: |-
7     quarkus.datasource.jdbc.url=jdbc:postgresql://postgres:5432/product
8     mp.jwt.verify.publickey.location=http://keycloak-
9       http.keycloak/auth/realm/quarkushop-realm/protocol/openid-connect/certs
10    mp.jwt.verify.issuer=http://keycloak-http.keycloak/auth/realm/quarkushop-realm

```



I've also changed the name of the DB (line 7) in this **ConfigMap**

But why did I changed the DB name ? 😊 The answer is in the **Chapter 9**: having a Database per **μservice** is a recommended pattern 😊 This is why I created a dedicated schema for each **μservice**:



As we are in the Database context, we need to copy the **Flyway** scripts **V1.0__Init_app.sql** and **V1.1__Insert_samples.sql** from the **monolithic application** to the **src/main/resources/db/migration** directory of the **Product μservice**. For sure, then, we need to clean up the **SQL Scripts** to keep only the **Product Bounded Context** related objects and sample data.



Be sure that you clean up correctly the scripts, otherwise the deployment will fail during the application boot process.

Next, there is a very important task to do: identifying the [quarkushop-commons](#) project to the **Quarkus Index** of the [quarkushop-product](#).

What is the Quarkus Indexing ?

Quarkus automatically indexes the current module but, when we have external modules containing CDI beans, entities, objects serialized as JSON, we need to explicitly index them.

The indexing can be done in many ways:

- Using the **Jandex Maven plugin**
- Adding an empty `META-INF/beans.xml` file
- Using **Quarkus Index Dependency** properties  this is my favourite choice

The indexing can be done using some [application.properties](#) values:

```
quarkus.index-dependency.common.group-id=com.targa.labs
quarkus.index-dependency.common.artifact-id=quarkushop-commons
```



Without this **index-dependency** configuration, we will not be able to build the native binary of the application.

Before building the project, we need to copy the related **TESTS** from the monolith to the [quarkushop-product](#) **μservice**:

- CategoryResourceIT
- CategoryResourceTest
- ProductResourceIT
- ProductResourceTest
- ReviewResourceIT
- ReviewResourceTest

We also need to copy the **Keycloak Docker** files from the `src/main/docker` to the [quarkushop-product](#) **μservice**:

- the `keycloak-test.yml` file
- the `realms` directory

Then, in order to be able to execute tests; we need to disable the **Kubernetes** support during in the **test** environment/profile:

```
%test.quarkus.kubernetes-config.enabled=false
quarkus.test.native-image-profile=test
```



We are defining that the **Native Image Test Profile** as **test** in order to have also the **Kubernetes** support disabled to for **Native Image Tests**

Now, we need to execute the tests, build and push the **quarkushop-product** image:

```
mvn clean install -Pnative \
-Dquarkus.native.container-build=true \
-Dquarkus.container-image.build=true
```

Next, we need to push the **quarkushop-product** image to the container registry:

```
docker push nebrass/quarkushop-product:1.0.0-SNAPSHOT
```

Now, we need to create the **quarkushop-product-config** ConfigMap:

```
kubectl apply -f quarkushop-product/quarkushop-product-config.yml
```

Now, we will deploy the **Product μservice** to the **Kubernetes** cluster:

```
kubectl apply -f quarkushop-product/target/kubernetes/kubernetes.json
```

Excellent! Now we can list the PODs:

NAME	READY	STATUS
postgres-69c47c748-pnbbf	1/1	Running
quarkushop-product-7748f9f74c-dqnqk	1/1	Running

We will test the application using a **port-forward** on the **quarkushop-product**

```
$ kubectl port-forward quarkushop-product-7748f9f74c-dqnqk 8080:8080
```

Forwarding from 127.0.0.1:8080 -> 8080

Forwarding from [::1]:8080 -> 8080

Handling connection for 8080

Handling connection for 8080

Then a **curl** command can count the Products stored on the DB:

```
$ curl -X GET "http://localhost:8080/api/products/count"
```

4

Good! The access to the **Database** is correctly working ☺ we can check also the **Health Check** to be sure that the **Keycloak** is correctly reached using a **curl -X GET "http://localhost:8080/api/health"** command:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "Keycloak connection health check",
      "status": "UP"
    },
    {
      "name": "Database connections health check",
      "status": "UP"
    }
  ]
}
```

Excellent! Everything is working as expected! ☺☺ We can move now to the **Order µservice**.

Implementing the Order μservice

Good! Now, we will generate the **Order μservice** with the same *Extensions* as the **Product μservice**, with an extra one: the **REST Client Extension**.

We know that the **Order μservice** has a communication dependency on the **Product μservice**. This communication can be implemented as **REST API** calls from the **Order μservice** to the **Product μservice**. This is why we included the **REST Client** extension to the selected dependencies.

After generating the project, we will do the same tasks as we did previously with the **Product μservice**:

- Copying the code from the `order` package in the monolithic application to the new **Order μservice**
- Adding the **Lombok**, **AssertJ** and **TestContainers** dependencies
- Adding the `quarkushop-commons` dependency
- Copying the `banner.txt` from the monolith
- Adding the `application.properties` and changing the **Database** and **ConfigMap** names
- Creating the `quarkushop-product-config` **ConfigMap** file
- Copying the **Flyway** scripts and cleaning up the unrelated objects and data
- Adding the **Quarkus Index Dependency** for `quarkushop-commons` in the `application.properties`

At this level, we need to fix the code, as we still have a `ProductRepository` reference inside the `OrderItemService` class:

The `OrderItemService` uses the `ProductRepository` to find a product using a given **ID**. This programmatic call will be replaced by a call **REST API** to the **Product μservice**. For this we need to create a `ProductRestClient` class that will fetch the **Product** data using a given **ID**:

```
@Path("/products")                                ①
@registerRestClient                            ②
public interface ProductRestClient {
    @GET
    @Path("/{id}")
    ProductDto findById(@PathParam Long id);      ③
}
```

① The `ProductRestClient` will be pointing on the URI `/products`

② The `@RegisterRestClient` allows **Quarkus** to know that this interface is meant to be available for **CDI injection** as a **REST Client**

③ The `findById()` method will do an **HTTP GET** on the URI `/products`

But what's the base URL of the **Product μservice** API? Good point! The **ProductRestClient** needs to be configured in order to work correctly. The configuration can be done using the properties:

```
1 product-service.url=http://quarkushop-product:8080/api
2 com.targa.labs.quarkushop.order.client.ProductRestClient/mp-rest/url=${product-
    service.url} ①
3 com.targa.labs.quarkushop.order.client.ProductRestClient/mp-
    rest/scope=javax.inject.Singleton ②
```

① The base URL configuration of the **ProductRestClient**

② Define the scope of the **ProductRestClient** bean as **Singleton**

We will refactor then the **OrderItemService** class to change the:

```
@Inject
ProductRepository productRepository;
```

by the new:

```
@Inject
@RestClient
ProductRestClient productRestClient;
```



The **@RestClient** is used along with **@Inject** to inject a **Rest Client**.

Then, we will change the **productRepository.getOne()** call by the **productRestClient.findById()** 😊 Now the JPA Repository fetches are replaced by a REST API call.

As we have an external dependency to the **Product μservice**, we need to create a **Health Check** that will verify that the **Product μservice** is reachable, the same way as we did with the **PostgreSQL** and **Keycloak**. The **ProductServiceHealthCheck** looks like:

```
@Slf4j
@Liveness
@ApplicationScoped
public class ProductServiceHealthCheck implements HealthCheck {

    @ConfigProperty(name = "product-service.url", defaultValue = "false")
    Provider<String> productServiceUrl;

    @Override
    public HealthCheckResponse call() {

        HealthCheckResponseBuilder responseBuilder =
            HealthCheckResponse.named("Product Service connection health check");
```

```
try {

    productServiceConnectionVerification();
    responseBuilder.up();

} catch (IllegalStateException e) {
    responseBuilder.down().withData("error", e.getMessage());
}

return responseBuilder.build();
}

private void productServiceConnectionVerification() {
    HttpClient httpClient = HttpClient.newBuilder()
        .connectTimeout(Duration.ofMillis(3000))
        .build();

    HttpRequest request = HttpRequest.newBuilder()
        .GET()
        .uri(URI.create(productServiceUrl.get() + "/health"))
        .build();

    HttpResponse<String> response = null;

    try {
        response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
    } catch (IOException e) {
        log.error("IOException", e);
    } catch (InterruptedException e) {
        log.error("InterruptedException", e);
        Thread.currentThread().interrupt();
    }

    if (response == null || response.statusCode() != 200) {
        throw new IllegalStateException("Cannot contact Product Service");
    }
}
}
```

Then, we need to copy the related **TESTS** from the monolith to the **quarkushop-order** μservice:

- AddressServiceUnitTest
- CartResourceIT
- CartResourceTest
- OrderItemResourceIT
- OrderItemResourceTest
- OrderResourceIT
- OrderResourceTest

We also need to copy the **Keycloak Docker** files from the **src/main/docker** to the **quarkushop-order** μservice:

- the **keycloak-test.yml** file
- the **realms** directory

Then we need to add the needed properties for Tests execution:

```
%test.quarkus.kubernetes-config.enabled=false
quarkus.test.native-image-profile=test
```

The **quarkushop-product** we don't rely on any other microservice, so until this point the modifications that we did for the **quarkushop-order** until now are enough for **quarkushop-product**. But the **quarkushop-order** reply on the **quarkushop-product** μservice ↗ which means, obviously, the **Tests** also rely on the **quarkushop-product** μservice.

We have many solutions here - my two choices:

1. Mocking the RestClient
2. Add a Testing Instance of **quarkushop-product**

I will go for the second choice for the **quarkushop-order** and I will be **Mocking the RestClient** for **quarkushop-customer** as he is replying on an other μservice.

If you (still) remember, we used **TestContainers** framework to provision a **Keycloak** instance for our **Integration Tests**. The provision was made using a **docker-compose** file, where a **keycloak** service was created. I got the idea to go on this way to provision a **quarkushop-product** instance, using the same **docker-compose** file. But I must create a new **QuarkusTestResourceLifecycleManager** class instead of the **KeycloakRealmResource** as it's made to provision Keycloak only.

I will rename the **src/main/docker/keycloak-test.yml** file to **src/main/docker/context-test.yml** - Then I will add two services: **quarkushop-product** and **postgresql-db**



Why do we add **postgresql-db** service? The answer is easy 😊 it's needed by the **quarkushop-product** - our μservices need DB storage + Keycloak tenant to work ☺

The content of `src/main/docker/context-test.yml` will be:

```

version: '3'
services:
  keycloak:
    image: jboss/keycloak:latest
    command:
      [
        '-b','0.0.0.0',
        '-Dkeycloak.migration.action=import',
        '-Dkeycloak.migration.provider=dir',
        '-Dkeycloak.migration.dir=/opt/jboss/keycloak/realm',
        '-Dkeycloak.migration.strategy=OVERWRITE_EXISTING',
        '-Djboss.socket.binding.port-offset=1000',
        '-Dkeycloak.profile.feature.upload_scripts=enabled',
      ]
    volumes:
      - ./realms-test:/opt/jboss/keycloak/realm
    environment:
      - KEYCLOAK_USER=admin
      - KEYCLOAK_PASSWORD=admin
      - DB_VENDOR=h2
    ports:
      - 9080:9080
      - 9443:9443
      - 10990:10990
  quarkushop-product:
    image: nebrass/quarkushop-product:1.0.0-SNAPSHOT
    environment:
      - QUARKUS_PROFILE=test
      - QUARKUS_DATASOURCE_JDBC_URL=jdbc:postgresql://postgresql-db:5432/product
      - MP_JWT_VERIFY_PUBLICKEY_LOCATION=http://keycloak:9080/auth/realm/quarkushop-realm/protocol/openid-connect/certs
      - MP_JWT_VERIFY_ISSUER=http://keycloak:9080/auth/realm/quarkushop-realm
    depends_on:
      - postgresql-db
      - keycloak
    ports:
      - 8080:8080
  postgresql-db:
    image: postgres:13
    volumes:
      - /opt/postgres-volume:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=developer
      - POSTGRES_PASSWORD=p4SSW0rd
      - POSTGRES_DB=product

```

```
- POSTGRES_HOST_AUTH_METHOD=trust
ports:
- 5432:5432
```



You can notice that I took the opportunity of the available Keycloak instance to use it thru environment variables for `quarkushop-product` ☺

Excellent! Now, **TestContainers** will provision a **Keycloak + PostgreSQL + quarkushop-product** instances ☺ which is what the `quarkushop-order` is dreaming of for his **Integration Tests** ☺

Next, we need to create the new `QuarkusTestResourceLifecycleManager` class that I will call `ContextTestResource` - This class will provision **Keycloak + quarkushop-product** and then needs to pass their properties to the application:

`src/test/java/com/targa/labs/quarkushop/order/util/ContextTestResource.java`

```
public class ContextTestResource implements QuarkusTestResourceLifecycleManager {

    @ClassRule
    public static DockerComposeContainer ECOSYSTEM = new DockerComposeContainer(
        new File("src/main/docker/context-test.yml"))
        .withExposedService("quarkushop-product_1", 8080, ①
            Wait.forListeningPort().withStartupTimeout(Duration.ofSeconds(30)))
        .withExposedService("keycloak_1", 9080, ①
            Wait.forListeningPort().withStartupTimeout(Duration.ofSeconds(30)));

    @Override
    public Map<String, String> start() {
        ECOSYSTEM.start();

        String jwtIssuerUrl = String.format("http://%s:%s/auth/realms/quarkus-realm",
            ECOSYSTEM.getServiceHost("keycloak_1", 9080),
            ECOSYSTEM.getServicePort("keycloak_1", 9080)
        );

        TokenService tokenService = new TokenService();
        Map<String, String> config = new HashMap<>();

        try {

            String adminAccessToken = tokenService.getAccessToken(jwtIssuerUrl,
                "admin", "test", "quarkus-client", "mysecret");
            String testAccessToken = tokenService.getAccessToken(jwtIssuerUrl,
                "test", "test", "quarkus-client", "mysecret");

            config.put("quarkus-admin-access-token", adminAccessToken);
            config.put("quarkus-test-access-token", testAccessToken);
        }
    }
}
```

```

    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }

    config.put("mp.jwt.verify.publickey.location", jwtIssuerUrl
               + "/protocol/openid-connect/certs");      ②
    config.put("mp.jwt.verify.issuer", jwtIssuerUrl);      ②

    String productServiceUrl = String.format("http://%s:%s/api",
                                              ECOSYSTEM.getServiceHost("quarkushop-product_1", 8080),
                                              ECOSYSTEM.getServicePort("quarkushop-product_1", 8080)
                                            );
    config.put("product-service.url", productServiceUrl);      ②

    return config;
}

@Override
public void stop() {
    ECOSYSTEM.stop();
}
}

```

① Defining the provisioned services

② Defining the needed properties of **Keycloak** and **quarkushop-product**

Then we need to refactor the **Tests** to include the **ContextTestResource**:

```

@DisabledOnNativeImage
@QuarkusTest
@QuarkusTestResource(TestContainerResource.class)
@QuarkusTestResource(ContextTestResource.class)
class CartResourceTest {
    ...
}

```



When walking into the refactoring steps, I removed the customer creation - I'm using now a randomly generated customer ID.

We can pursue the μservice creation process 😊

- Building the **quarkushop-order** image and will push it to the container registry

```
$ mvn clean install -Pnative \
-Dquarkus.native.container-build=true \
-Dquarkus.container-image.build=true

$ docker push nebrass/quarkushop-order:1.0.0-SNAPSHOT
```

- Creating the **quarkushop-order-config ConfigMap** file

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: quarkushop-order-config
5 data:
6   application.properties: |- 
7     quarkus.datasource.jdbc.url=jdbc:postgresql://postgres:5432/order
8     mp.jwt.verify.publickey.location=http://keycloak-
      http.keycloak/auth/realm/quarkushop-realm/protocol/openid-connect/certs
9     mp.jwt.verify.issuer=http://keycloak-http.keycloak/auth/realm/quarkushop-realm
```

- Creating the **quarkushop-order-config ConfigMap** in Kubernetes:

```
kubectl apply -f quarkushop-order/quarkushop-order-config.yml
```

- Deploying the **quarkushop-order** application to Kubernetes

```
kubectl apply -f quarkushop-order/target/kubernetes/kubernetes.json
```

- Checking the `quarkushop-order` POD using the **Health Checks** using a `port-forward` and a simple `curl` GET command to its `/health` API:

```
{  
    "status": "UP",  
    "checks": [  
        {  
            "name": "Keycloak connection health check",  
            "status": "UP"  
        },  
        {  
            "name": "Product Service connection health check",  
            "status": "UP"  
        },  
        {  
            "name": "Database connections health check",  
            "status": "UP"  
        }  
    ]  
}
```

Good! The **Order µservice** is working as expected. We will do the same job with the **Customer µservice** 😊

Implementing the Customer μservice

To implement the **Customer μservice** we will apply the same steps as we did with the **Order μservice**, again we will:

- Copying the code from the `customer` package in the monolithic application to the new **Customer μservice**
- Adding the **Lombok**, **AssertJ** and **TestContainers** dependencies
- Adding the `quarkushop-commons` dependency
- Copying the `banner.txt` from the monolith
- Adding the `application.properties` and changing the **Database** and **ConfigMap** names
- Copying the **Flyway** scripts and cleaning up the unrelated objects and data
- Adding the **Quarkus Index Dependency** for `quarkushop-commons` in the `application.properties`

Then, we need to copy the related **TESTS** from the monolith to the `quarkushop-customer` μservice:

- `CustomerResourceIT`
- `CustomerResourceTest`
- `PaymentResourceIT`
- `PaymentResourceTest`

We also need to copy the **Keycloak Docker** files from the `src/main/docker` to the `quarkushop-order` μservice:

- the `keycloak-test.yml` file
- the `realms` directory

Then we need to add the needed properties for **Tests** execution:

```
%test.quarkus.kubernetes-config.enabled=false
quarkus.test.native-image-profile=test
```

Then, we will **Mock** the `OrderRestClient` - We will be creating a new *Class* called `MockOrderRestClient`:

`src/test/java/com/targa/labs/quarkushop/customer/utils/MockOrderRestClient.java`

```

@Mock          ①
@ApplicationScoped
@RestClient
public class MockOrderRestClient implements OrderRestClient { ②

    @Override
    public Optional<OrderDto> findById(Long id) { ③
        OrderDto order = new OrderDto();
        order.setId(id);
        order.setTotalPrice(BigDecimal.valueOf(1000));
        return Optional.of(order);
    }

    @Override
    public Optional<OrderDto> findByPaymentId(Long id) {
        OrderDto order = new OrderDto();
        order.setId(5L);
        return Optional.of(order);
    }

    @Override
    public OrderDto save(OrderDto order) {
        return order;
    }
}

```

① Annotation used to mock beans injected in tests

② The **Mock** class implements the `RestClient` interface

③ The **Mock** methods were implemented to return results suitable for tests

That's all tale! Mocking is very easy! The Mocked component will be automatically injected to the tests ☺

- Building the `quarkushop-customer` image and will push it to the container registry
- Creating the `quarkushop-customer-config` **ConfigMap** file
- Creating the `quarkushop-customer-config` **ConfigMap** in Kubernetes
- Checking the `quarkushop-customer` POD using the **Health Checks** using a `port-forward` and a simple `curl` GET command to its `/health` API



You will find all the needed code and resources in the book's [GitHub Repositories](#).

Implementing the User µservice

Oppa! What is this extra µservice?? Hey?! You never told us about this one?! 😊😊

Keep cool! 😊 This is not a huge µservice. It's only a µservice that will be used for the authentication. It will hold mainly the **user** section REST APIs listed in the **monolith Quarkushop Swagger-UI**:

user All the user methods		
POST	/api/user/access-token	🔒
GET	/api/user/current/info	🔒
GET	/api/user/current/info-alternative	🔒
GET	/api/user/current/info/claims	🔒

Let's generate a new Quarkus application with these extensions:

- RESTEasy JAX-RS
- RESTEasy JSON-B
- SmallRye OpenAPI
- SmallRye JWT
- SmallRye Health
- SmallRye Metrics
- Kubernetes
- Kubernetes Config
- Container Image Jib



There is no persistence related extensions as we don't interact with the DB in this µservice

The steps to follow in order to implement the **User µservice**, we will be:

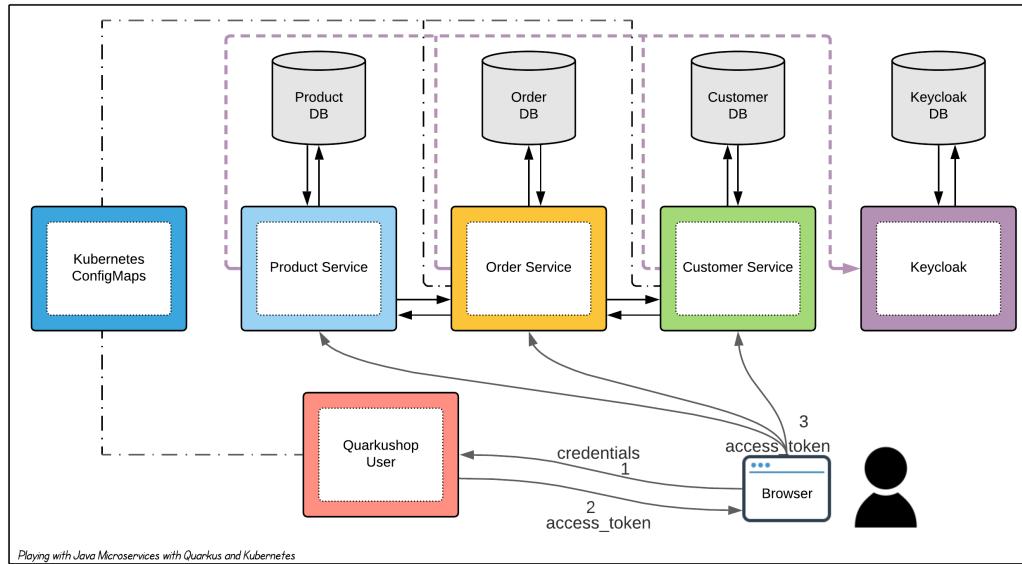
- Copying the User related content: as this µservice is very minimal: it will only hold the **UserResource** class
- Adding the **Lombok** dependency
- Adding the **quarkushop-commons** dependency
- Copying the **banner.txt** from the monolith
- Adding the **application.properties**, changing the **ConfigMap** name and removing all data persistence properties (*Flyway, JPA, etc.*)
- Adding the **Quarkus Index Dependency** for **quarkushop-commons** in the **application.properties**
- Building the **quarkushop-user** image and will push it to the container registry
- Creating the **quarkushop-user-config ConfigMap** file
- Creating the **quarkushop-user-config ConfigMap** in Kubernetes

- Checking the **quarkushop-user** POD using the **Health Checks** using a **port-forward** and a simple **curl** GET command to its **/health** API



You will find all the needed code and resources in the book's [GitHub Repositories](#).

After deploying the **quarkushop-user** μservice, we will be using it to get an **access_token** in order to be able to communicate with the secured APIs of the other three μservices:



Good! Our μservices are here and correctly deployed and working 😊

Conclusion

We successfully created our μservices and deployed them to Kubernetes. During this heavy task, we implemented some patterns from those introduced in **Chapter 9**. But we did not finish yet to implement the most wanted ones. This is what we will be doing in the next chapter 😊

CHAPTER THIRTEEN

Flying all over the Sky with Quarkus
and Kubernetes

Introduction

Starting from **Chapter 9**, we started digging into the **Cloud Native Patterns** and even implemented some of them:

- **Service discovery and registration**: made using the **Kubernetes Deployment & Service** objects
- **Externalized configuration**: made using the **Kubernetes ConfigMap** and **Secret** objects
- **Database per service**: made while splitting the monolith codebase using DDD concepts
- **Application metrics**: implemented using the **SmallRye Metrics Quarkus Extension**
- **Health Check API**: implemented using the **SmallRye Health Quarkus Extension**
- **Security Between Services**: implemented using the **SmallRye JWT Quarkus Extension** and **Keycloak**

In this chapter we will implement more popular patterns:

- **Circuit Breaker**
- **Log Aggregation**
- **Distributed Tracing**
- **API Gateway**

Implementing the Circuit Breaker pattern

The **Circuit Breaker** pattern is useful for making resilient µservices, which are using faulty communication protocols (like **HTTP**). The idea of the pattern is to handle the communication problem between µservices



The implementation of the **Circuit Breaker** pattern will impact only the **Order** and **Customer** µservices - where we used the **Rest Client** to make external calls

Implementing this pattern in a **Quarkus** based application is very easy:

The first step is to add this dependency to the **pom.xml**:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-fault-tolerance</artifactId>
</dependency>
```

Now we will enable the **Circuit Breaker** feature on our **Rest Client** components:

Let's say that we want our Rest Clients to **stop making the API Calls for 15 seconds, if we have a 50% of failing request from the last 10 requests**. This is can be done using this line of code:

```
@CircuitBreaker(requestVolumeThreshold = 10, failureRatio = 0.5, delay = 15000)
```

In the **Order μservice**:

```
@Path("/products")
@RegisterRestClient
public interface ProductRestClient {

    @GET
    @Path("/{id}")
    @CircuitBreaker(requestVolumeThreshold = 10, failureRatio = 0.5, delay = 15000)
    ProductDto findById(@PathParam Long id);
}
```

@CircuitBreaker has many attributes:

- **failOn**: The list of exception types which should be considered failures - the default value is `{Throwable.class}` ↗ All Exceptions inheriting from `Throwable`, thrown by the annotated method, are considered as failures
- **skipOn**: The list of exception types which should not be considered failures - the default value is `{}`
- **delay**: The delay after which an open circuit will transition to half-open state - the default value is `5000`
- **delayUnit**: The unit of the delay - the default value is `ChronoUnit.MILLIS`
- **requestVolumeThreshold**: The number of consecutive requests in a rolling window - the default value is `20`
- **failureRatio**: The ratio of failures within the rolling window that will trip the circuit to open - the default value is `0.50`
- **successThreshold**: The number of successful executions, before a half-open circuit is closed again - the default value is `1`

The **Circuit Breaker** has three states:



- **Closed**: all requests are made normally
- **Half-open**: transition state; where verifications are made to check if the problem is really occurring or no more
- **Open**: all requests are disabled until the delay is due

In the **Customer** µservice:

```
@Path("/orders")
@RegisterRestClient
public interface OrderRestClient {

    @GET
    @Path("/{id}")
    @CircuitBreaker(requestVolumeThreshold = 10, delay = 15000)
    Optional<OrderDto> findById(@PathParam Long id);

    @GET
    @Path("/payment/{id}")
    @CircuitBreaker(requestVolumeThreshold = 10, delay = 15000)
    Optional<OrderDto> findByPaymentId(Long id);

    @POST
    @CircuitBreaker(requestVolumeThreshold = 10, delay = 15000)
    OrderDto save(OrderDto order);
}
```

The **SmallRye Fault Tolerance** extension offers many other useful options to deal with faults in order to be strong resilient µservices. We have for example a :

1. **Retry mechanism:** used to make a number of retries in case of an invocation fails - Let's do it:

```
@Path("/products")
@RegisterRestClient
public interface ProductRestClient {

    @GET
    @Path("/{id}")
    @CircuitBreaker(requestVolumeThreshold = 10, delay = 15000)
    @Retry(maxRetries = 4)
    ProductDto findById(@PathParam Long id);
}
```



The `@Retry(maxRetries = 4)` will make up to **4** retries in case of the invocation fails

2. **Timeout mechanism:** used to define a method execution timeout. It can be easily implemented:

```
@Path("/products")
@RegisterRestClient
public interface ProductRestClient {

    @GET
    @Path("/{id}")
    @CircuitBreaker(requestVolumeThreshold = 10, delay = 15000)
    @Retry(maxRetries = 4)
    @Timeout(500)
    ProductDto findById(@PathParam Long id);
}
```



The `@Timeout(500)` will make the application throw a `TimeoutException` in case of the `findById()` invocation took more than 500 milliseconds.

3. **Fallback mechanism:** used to invoke a fallback (or backup) method, in case of the main method failed - An annotation is here to do the job:

```
public class SomeClass {

    @Inject
    @RestClient
    ProductRestClient productRestClient;

    @Fallback(fallbackMethod = "fallbackFetchProduct")
    List<ProductDto> findProductsByCategory(String category){
        return productRestClient.findProductsByCategory(category);
    }

    public List<ProductDto> fallbackFetchProduct(String category) {
        return Collections.emptyList();
    }
}
```



In case of the `productRestClient.findProductsByCategory()` fails, we will get the response from `fallbackFetchProduct()` method instead of the `findProductsByCategory()`. We can tune more this powerful mechanism. We can, for example, configure it to switch to the fallback method in case of defined Exceptions only or after some timeout.

You can notice that the Circuit Breaker pattern or even all the Fault Tolerance patterns are perfectly implemented in the Quarkus framework.

Implementing the Log Aggregation pattern

For the **Log aggregation**, we will use the famous **ELK Stack**. What is the **ELK**?

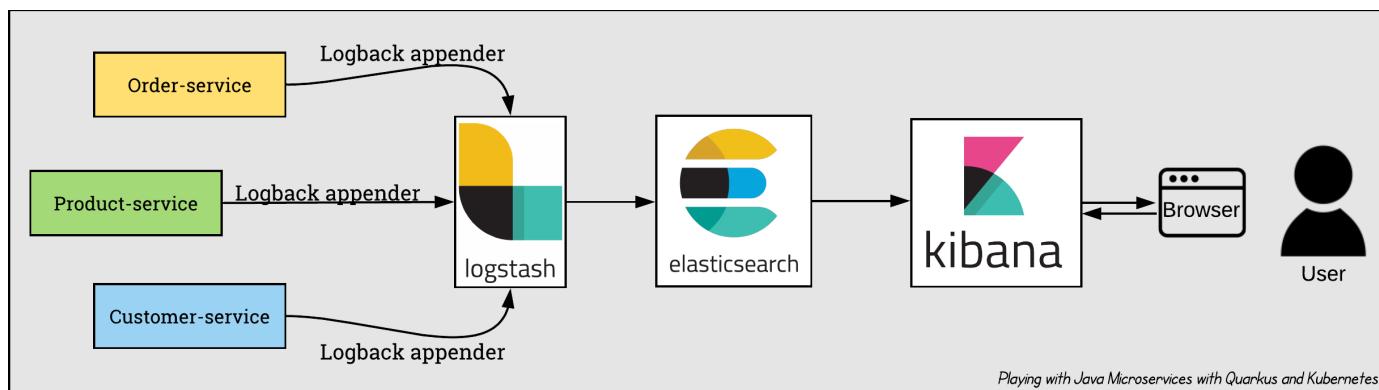
"**ELK**" is the acronym for three open source projects: **Elasticsearch**, **Logstash**, and **Kibana**.



- **Elasticsearch** is a search and analytics engine.
- **Logstash** is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like **Elasticsearch**.
- **Kibana** lets users visualize data with charts and graphs in **Elasticsearch**.

Together, these tools are most commonly used for centralizing and analyzing logs in distributed systems. The **ELK Stack** is popular because it fulfills a need in the log analytics space.

Our use case of the **ELK Stack** will be like:



All our services will push their respective logs to **Logstash**, which will use **Elasticsearch** to index them. The indexed logs can be consumed later by **Kibana**.

Quarkus has a great extension called **quarkus-logging-gelf**, which is described by "Log using the **Graylog Extended Log Format** and centralize your logs in **ELK** or **EFK**" ☺

What is the "Graylog Extended Log Format" ?

Based on the Graylog.org website: The **Graylog Extended Log Format (GELF)** is a uniquely convenient log format created to deal with all the shortcomings of classic plain **Syslog**. This enterprise feature allows you to collect structured events from anywhere, and then compress and chunk them in the blink of an eye.

Good! **Logstash** is supporting natively the **Graylog Extended Log Format**. We need just to activate it during the configuration process.

Step 1: Deploying the ELK Stack to Kubernetes

How to install the **ELK Stack** in our **Kubernetes** Cluster ? 🤔 Huge question!

Keep calm! It's an extremely easy task: as we did with **Keycloak**, we will use **Helm** for installing the **ELK Stack** 😊

We will start by adding the **official ELK Helm Charts Repository** in our **Helm** client:

```
helm repo add elastic https://helm.elastic.co
```

Next, we need to update the references:

```
helm repo update
```

If you are on **Minikube** like me, you need to create an **elasticsearch-values.yaml** file that we will use in customizing the **helm install**:

elasticsearch-values.yaml

```
# Permit co-located instances for solitary minikube virtual machines.
antiAffinity: "soft"

# Shrink default JVM heap.
esJavaOpts: "-Xmx128m -Xms128m"

# Allocate smaller chunks of memory per pod.
resources:
  requests:
    cpu: "100m"
    memory: "512M"
  limits:
    cpu: "1000m"
    memory: "512M"

# Request smaller persistent volumes.
volumeClaimTemplate:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: "standard"
  resources:
    requests:
      storage: 100M
```



This configuration file is the recommended configuration to use while installing **Elasticsearch** on **Minikube**: <https://github.com/elastic/helm-charts/blob/master/elasticsearch/examples/minikube/values.yaml>

Now, we will install **Elasticsearch**:

```
helm install elasticsearch elastic/elasticsearch -f ./elasticsearch-values.yaml
```

We can check what was created by this **chart** using this command:

```
$ kubectl get all -l release=elasticsearch

NAME                               READY   STATUS    RESTARTS
pod/elasticsearch-master-0        1/1     Running   0
pod/elasticsearch-master-1        1/1     Running   0
pod/elasticsearch-master-2        1/1     Running   0

NAME                           TYPE      CLUSTER-IP   PORT(S)
service/elasticsearch-master      ClusterIP  10.103.91.46  9200/TCP,9300/TCP
service/elasticsearch-master-headless  ClusterIP  None        9200/TCP,9300/TCP

NAME                               READY
statefulset.apps/elasticsearch-master  3/3
```



The three masters are here for **High Availability** purposes. I know that we don't have the problem now, but think in the **Black Friday!** 😊

Our **PODs** are in the running status, so we can test the famous **Elasticsearch 9200** Port - We can do a **port-forward** and a **curl**:

```
$ kubectl port-forward service/elasticsearch-master 9200
Forwarding from 127.0.0.1:9200 -> 9200
Forwarding from [::1]:9200 -> 9200

$ curl localhost:9200
{
  "name" : "elasticsearch-master-1",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "UkYbL4KsSeK4boVr4r0e2w",
  "version" : {
    "number" : "7.9.2",
    ...
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Excellent! We can move to deploy **Kibana**:

```
helm install kibana elastic/kibana --set fullnameOverride=quarkushop-kibana
```

We can check what was created by this **chart** using this command:

```
$ kubectl get all -l release=kibana

NAME                                     READY   STATUS
pod/quarkushop-kibana-696f869668-5tcvz  1/1    Running

NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
service/quarkushop-kibana   ClusterIP   10.107.223.6   <none>        5601/TCP

NAME           READY   UP-TO-DATE   AVAILABLE
deployment.apps/quarkushop-kibana  1/1     1           1

NAME           DESIRED  CURRENT  READY
replicaset.apps/quarkushop-kibana-696f869668  1       1       1
```

We will do a **port-forward** on the **Kibana** Service:

```
kubectl port-forward service/quarkushop-kibana 5601
```

Then open the **Kibana UI** to check that everything is working correctly:

The screenshot shows the Kibana UI homepage at `localhost:5601/app/home#`. The interface is divided into several sections:

- Observability** section:
 - APM**: Describes APM automatically collecting in-depth performance metrics and errors from inside your applications. Includes a [Add APM](#) button.
 - Logs**: Describes ingesting logs from popular data sources and visualizing them in preconfigured dashboards. Includes a [Add log data](#) button.
 - Metrics**: Describes collecting metrics from the operating system and services running on your servers. Includes a [Add metric data](#) button.
- Security** section:
 - SIEM + Endpoint Security**: Protect hosts, analyze security information and events, hunt threats, automate detections, and create cases. Includes a [Add events](#) button.
- Add sample data** and **Use Elasticsearch data** buttons.
- Visualize and Explore Data** section:
 - APM**: Automatically collect in-depth performance metrics and errors from inside your applications.
 - App Search**: Leverage dashboards, analytics, and APIs for advanced application search made simple.
- Manage and Administer the Elastic Stack** section:
 - Console**: Skip cURL and use this JSON interface to work with your data directly.
 - Rollups**: Summarize and store historical data in a smaller index for future analysis.

Good! We need to install **Logstash** - but we will need to customize the installation using a Helm **logstash-values.yaml** file:

logstash-values.yaml

```
logstashConfig:
  logstash.yml: |
    http.host: "0.0.0.0"
    xpack.monitoring.elasticsearch.hosts: [ "http://elasticsearch-master:9200" ]
    xpack.monitoring.enabled: true
  pipelines.yml: |
    - pipeline.id: custom
      path.config: "/usr/share/logstash/pipeline/logstash.conf"
logstashPipeline:
  logstash.conf: |
    input {
      gelf {
        port => 12201
        type => gelf
      }
    }

    output {
      stdout {}
      elasticsearch {
        hosts => ["http://elasticsearch-master:9200"]
        index => "logstash-%{+YYYY-MM-dd}"
      }
    }
service:
  annotations: {}
  type: ClusterIP
  ports:
    - name: filebeat
      port: 5000
      protocol: TCP
      targetPort: 5000
    - name: api
      port: 9600
      protocol: TCP
      targetPort: 9600
    - name: gelf
      port: 12201
      protocol: UDP
      targetPort: 12201
```

This **values.yaml** file is used to configure:

1. the **Logstash Pipeline**: enabling the **gelf** plugin and exposing the default **12201** port + defining the Logstash output pattern and flow to our **Elasticsearch** instance
2. the **Logstash Service** definition and exposed ports

Now, let's install the **Logstash**:

```
helm install -f ./logstash-values.yaml logstash elastic/logstash \
--set fullnameOverride=quarkushop-logstash
```

To list the created objects:

```
$ k get all -l chart=logstash

NAME                      READY   STATUS    RESTARTS
pod/quarkushop-logstash-0 1/1     Running   0

NAME                           TYPE      CLUSTER-IP      PORT(S)
service/quarkushop-logstash   ClusterIP  10.107.204.49
5000/TCP,9600/TCP,12201/UDP
service/quarkushop-logstash-headless   ClusterIP  None
5000/TCP,9600/TCP,12201/UDP

NAME                      READY
statefulset.apps/quarkushop-logstash 1/1
```

Excellent! Now the **ELK Stack** is correctly deployed - we will proceed next to configure the µservices to log into the **ELK Stack**.

Step 2: Configuring the Microservices to Log into the ELK Stack

The modifications that we will do in this step, are applicable for:

- quarkushop-product
- quarkushop-order
- quarkushop-customer
- quarkushop-user

Good! Let's add the extension to the `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-logging-gelf</artifactId>
</dependency>
```

Then, we need to define the **Logstash** server `properties` to each `ConfigMap` file:

`quarkushop-order-config.yml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: quarkushop-order-config
data:
  application.properties: |-
    quarkus.datasource.jdbc.url=jdbc:postgresql://postgres:5432/order
    mp.jwt.verify.publickey.location=http://keycloak-
    http.keycloak/auth/realms/quarkushop-realm/protocol/openid-connect/certs
    mp.jwt.verify.issuer=http://keycloak-http.keycloak/auth/realms/quarkushop-realm
    quarkus.log.handler.gelf.enabled=true
    quarkus.log.handler.gelf.host=quarkushop-logstash
    quarkus.log.handler.gelf.port=12201
```

Good let's build, push the containers and deploy again them all again to our **Kubernetes Cluster**. For sure, we need to import again the `ConfigMaps` in order to update them.

Step3: Collecting logs !

Once everything is deployed and configured, we need to access **Kibana UI** to parse the collected logs:

```
kubectl port-forward service/quarkushop-kibana 5601
```

Then we need to go **Management > Stack Management > Index Management > Kibana > Index Patterns**:

The screenshot shows the Kibana UI navigation sidebar. The 'Index patterns' section is highlighted in the 'Management' category. Other categories like 'Observability', 'Security', and 'Stack' are also visible.

- Observability**: Overview, Logs, Metrics, APM, Uptime.
- Security**: Overview, Detections, Hosts, Network, Timelines, Cases, Administration.
- Management**: Dev Tools, Ingest Manager, Stack Monitoring, Stack Management (selected).
- Ingest**: Ingest Node Pipelines.
- Data**: Index Management, Index Lifecycle Policies, Snapshot and Restore, Rollup Jobs, Transformations, Remote Clusters.
- Alerts and Insights**: Alerts and Actions, Reporting.
- Kibana**: Index Patterns (selected), Saved Objects, Spaces, Advanced Settings.
- Stack**: License Management, 8.0 Upgrade Assistant.

Here click on **[Create index pattern]** to create a new *Index Pattern* - Then a new screen will be shown:

The screenshot shows the 'Create index pattern' step 1 of 2 interface. The user has entered 'index-name-*' as the index pattern name. The 'Include system and hidden indices' checkbox is checked. A note says 'Your index pattern can match your 1 source.' Below the form, there is a table with one row labeled 'logstash-2020-10-13'.

Index
logstash-2020-10-13

At the bottom, there is a 'Rows per page: 10' dropdown.

In the **Index pattern name** field, just fill it with **logstash-*** and hit next step:

Create index pattern

An index pattern can match a single source, for example, `filebeat-4-3-22`, or **multiple** data sources, `filebeat-*`.

[Read documentation](#)

Step 1 of 2: Define index pattern

Index pattern name

logstash-*

Use an asterisk (*) to match multiple indices. Spaces and the characters \, /, ?, *, <, >,] are not allowed.

Include system and hidden indices

✓ Your index pattern matches 1 source.

logstash-2020-10-13

Index

Rows per page: 10

In the next screen, just select **@timestamp** for the Time Field and hit [**Create index pattern**]:

Create index pattern

An index pattern can match a single source, for example, `filebeat-4-3-22`, or **multiple** data sources, `filebeat-*`.

[Read documentation](#)

Step 2 of 2: Configure settings

logstash-*

Select a primary time field for use with the global time filter.

Time field Refresh

@timestamp

> Show advanced options

Back Create index pattern

Then a new **Index Pattern** is created - a confirmation screen is shown:

★ logstash-*

Time Filter field name: '@timestamp' Default

This page lists every field in the `logstash-*` index and the field's associated core type as recorded by Elasticsearch. To change a field type, use the Elasticsearch [Mapping API](#).

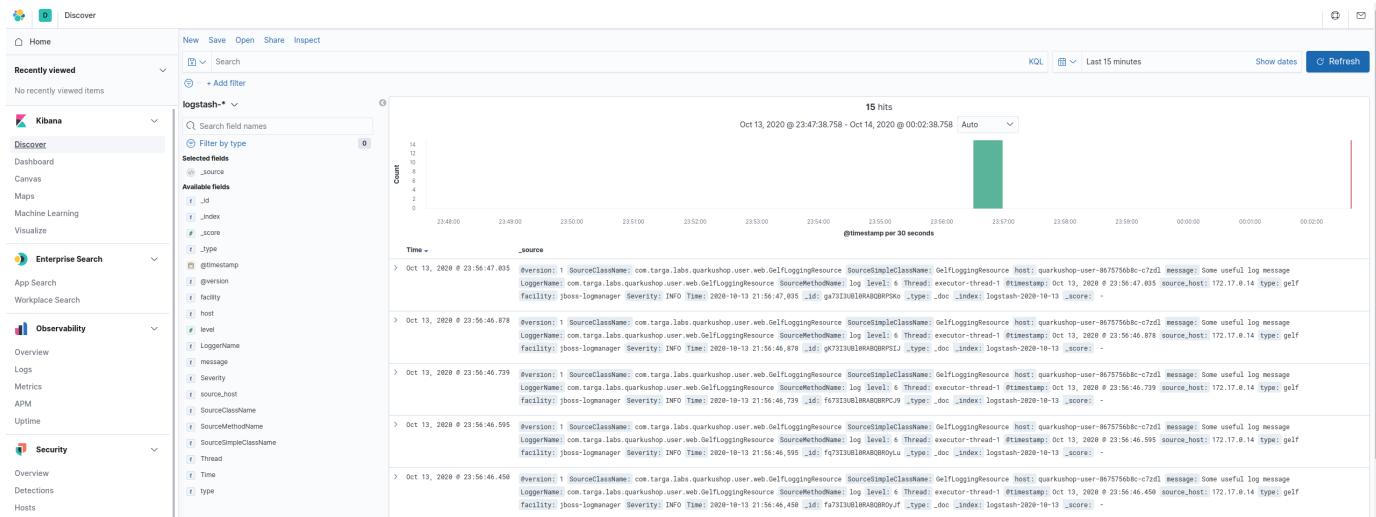
Fields (35) Scripted fields (0) Source filters (0)

All field types

Name	Type	Format	Searchable	Aggregatable	Excluded
@timestamp	date		●	●	○
@version	string		●	●	○
LoggerName	string		●		○
LoggerName.keyword	string		●	●	○
Severity	string		●		○
Severity.keyword	string		●	●	○
SourceClassName	string		●		○
SourceClassName.keyword	string		●	●	○
SourceMethodName	string		●		○
SourceMethodName.keyword	string		●	●	○

Rows per page: 10

Now, if you click on the **Discover** menu in the **Kibana** section, you will see the logs listing ☺



Excellent! ☺ Now we can keep an eye on all the logs produced in all our µservices. We can enjoy the powerful features of the **ELK Stack**. For example, we can make a custom query to monitor a specific kind of errors in the log streams.

Implementing the Distributed Tracing pattern

The **Distributed Tracing** pattern has dedicated extension in the **Quarkus** called **quarkus-smallrye-opentracing** ☺ Just like **Log Aggregation**, for the **Distributed Tracing**, we will need to have a **Distributed Tracing System**.

A **Distributed Tracing System** is used to collect and store the timing data needed to monitor communication requests in a µservices architecture in order to detect latency problems. There are many available **Distributed Tracing Systems** in the market, like **Zipkin** and **Jaeger**. In this book, we will use **Jaeger** as it is the default tracer supported by the **quarkus-smallrye-opentracing** extension.

Now, we need to install **Jaeger** and configure our µservices to support it in order to collect request traces.

Step 1: Deploying the Jaeger to Kubernetes

We will start by creating the **jaeger-deployment.yml** with this content:

jaeger/jaeger-deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: jaeger
    app.kubernetes.io/component: all-in-one
    app.kubernetes.io/name: jaeger
  name: jaeger
spec:
  progressDeadlineSeconds: 2147483647
  replicas: 1
```

```
revisionHistoryLimit: 2147483647
selector:
  matchLabels:
    app: jaeger
    app.kubernetes.io/component: all-in-one
    app.kubernetes.io/name: jaeger
strategy:
  type: Recreate
template:
  metadata:
    annotations:
      prometheus.io/port: "16686"
      prometheus.io/scrape: "true"
    labels:
      app: jaeger
      app.kubernetes.io/component: all-in-one
      app.kubernetes.io/name: jaeger
spec:
  containers:
    - env:
        - name: COLLECTOR_ZIPKIN_HTTP_PORT
          value: "9411"
      image: jaegertracing/all-in-one
      imagePullPolicy: Always
      name: jaeger
      ports:
        - containerPort: 5775
          protocol: UDP
        - containerPort: 6831
          protocol: UDP
        - containerPort: 6832
          protocol: UDP
        - containerPort: 5778
          protocol: TCP
        - containerPort: 16686
          protocol: TCP
        - containerPort: 9411
          protocol: TCP
      readinessProbe:
        failureThreshold: 3
        httpGet:
          path: /
          port: 14269
          scheme: HTTP
        initialDelaySeconds: 5
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 1
```

```

resources: {}
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
dnsPolicy: ClusterFirst
restartPolicy: Always
schedulerName: default-scheduler
securityContext: {}
terminationGracePeriodSeconds: 30

```

Now, let's import it in our **Kubernetes Cluster**:

```
kubectl apply -f jaeger/jaeger-deployment.yml
```

Then, we need to create a **LoadBalanced Kubernetes Service** object called **jaeger-query**:

jaeger/jaeger-query-service.yml

```

apiVersion: v1
kind: Service
metadata:
  name: jaeger-query
  labels:
    app: jaeger
    app.kubernetes.io/name: jaeger
    app.kubernetes.io/component: query
spec:
  ports:
    - name: query-http
      port: 80
      protocol: TCP
      targetPort: 16686
  selector:
    app.kubernetes.io/name: jaeger
    app.kubernetes.io/component: all-in-one
  type: LoadBalancer

```

And an other **Service** called **jaeger-collector**:

jaeger/jaeger-collector-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: jaeger-collector
  labels:
    app: jaeger
    app.kubernetes.io/name: jaeger
    app.kubernetes.io/component: collector
spec:
  ports:
    - name: jaeger-collector-tchannel
      port: 14267
      protocol: TCP
      targetPort: 14267
    - name: jaeger-collector-http
      port: 14268
      protocol: TCP
      targetPort: 14268
    - name: jaeger-collector-zipkin
      port: 9411
      protocol: TCP
      targetPort: 9411
  selector:
    app.kubernetes.io/name: jaeger
    app.kubernetes.io/component: all-in-one
  type: ClusterIP
```

And a the last one, which is called **jaeger-agent**:

jaeger/jaeger-agent-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: jaeger-agent
  labels:
    app: jaeger
    app.kubernetes.io/name: jaeger
    app.kubernetes.io/component: agent
spec:
  ports:
    - name: agent-zipkin-thrift
      port: 5775
      protocol: UDP
      targetPort: 5775
    - name: agent-compact
      port: 6831
      protocol: UDP
      targetPort: 6831
    - name: agent-binary
      port: 6832
      protocol: UDP
      targetPort: 6832
    - name: agent-configs
      port: 5778
      protocol: TCP
      targetPort: 5778
  clusterIP: None
  selector:
    app.kubernetes.io/name: jaeger
    app.kubernetes.io/component: all-in-one
```

Now let's create the **Kubernetes** objects:

```
kubectl apply -f jaeger/jaeger-query-service.yml
kubectl apply -f jaeger/jaeger-collector-service.yml
kubectl apply -f jaeger/jaeger-agent-service.yml
```

Excellent now, let's check if **Jaeger** is correctly installed. We need to do a **port-forward** on the **jaeger-query Service**:

```
kubectl port-forward service/jaeger-query 8888:80
```

Then open the **localhost:8888**:

Excellent! We can move to the µservices configuration 😊

Step 2: Enabling the Jaeger support in our Microservices

The first step to do is to add the **quarkus-smallrye-opentracing** dependency to our µservices:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

Then, we need to define the **Jaeger** configuration in every μservice's **ConfigMap**:

```

1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: quarkushop-user-config
5 data:
6   application.properties: |-
7     mp.jwt.verify.publickey.location=http://keycloak-
8       http.keycloak/auth/realms/quarkushop-realm/protocol/openid-connect/certs
9     mp.jwt.verify.issuer=http://keycloak-http.keycloak/auth/realms/quarkushop-realm
10    quarkus.log.handler.gelf.enabled=true
11    quarkus.log.handler.gelf.host=quarkushop-logstash
12    quarkus.log.handler.gelf.port=12201
13    quarkus.jaeger.service-name=quarkushop-user
14    quarkus.jaeger.sampler-type=const
15    quarkus.jaeger.sampler-param=1
16    quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId},
      spanId=%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%n
17    quarkus.jaeger.agent-host-port=jaeger-agent:6831

```

The new **Jaeger properties** are:

- **quarkus.jaeger.service-name**: The service name ↗ the name used by the μservice for presenting himself to the **Jaeger** server
- **quarkus.jaeger.sampler-type**: The sampler type is our example is **constant** ↗ we will constantly send the quota defined in the **quarkus.jaeger.sampler-param**
- **quarkus.jaeger.sampler-param**: Sampler quota defined between **0** et **1** - where **1** is **100%** of requests
- **quarkus.log.console.format**: Add **Trace** IDs into log message
- **quarkus.jaeger.agent-host-port**: The **hostname** and **port** for communicating with the **Jaeger Agent** via **UDP** ↗ we make it point on the **jaeger-agent** as **Host** and **6831** as **Port**

Good let's build, push the containers and deploy again them all again to our **Kubernetes** Cluster. For sure, we need to import again the **ConfigMaps** in order to update them.

Step 3: Collecting traces !

After deploying the **Jaeger** server and the updates μservices, we need to make some requests to generate traces. Then we can check what **Jaeger** is already catching.

For example, I will use the **quarkushop-user** for requesting an **access_token** from **Keycloak**:

```
kubectl port-forward service/quarkushop-user 8080
```

Then let's do a `curl` to request the `access_token` from `quarkushop-user` μservice:

```
curl -X POST "http://localhost:8080/api/user/access-
token?password=password&username=nebrass"
```

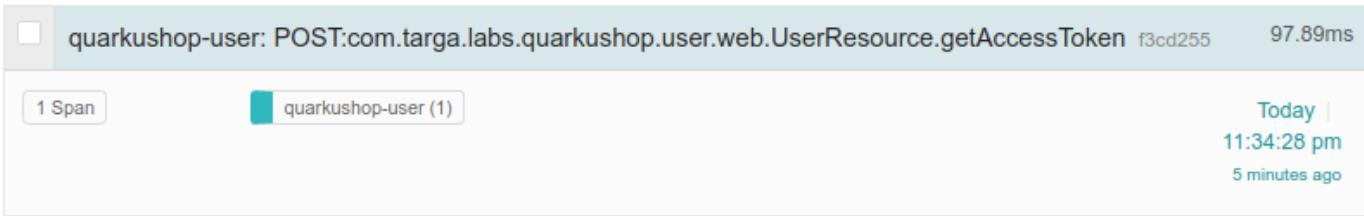
Good! Let's do a `port-forward` and access the **Jaeger** to check what's going on there:

```
kubectl port-forward service/jaeger-query 8888:80
```

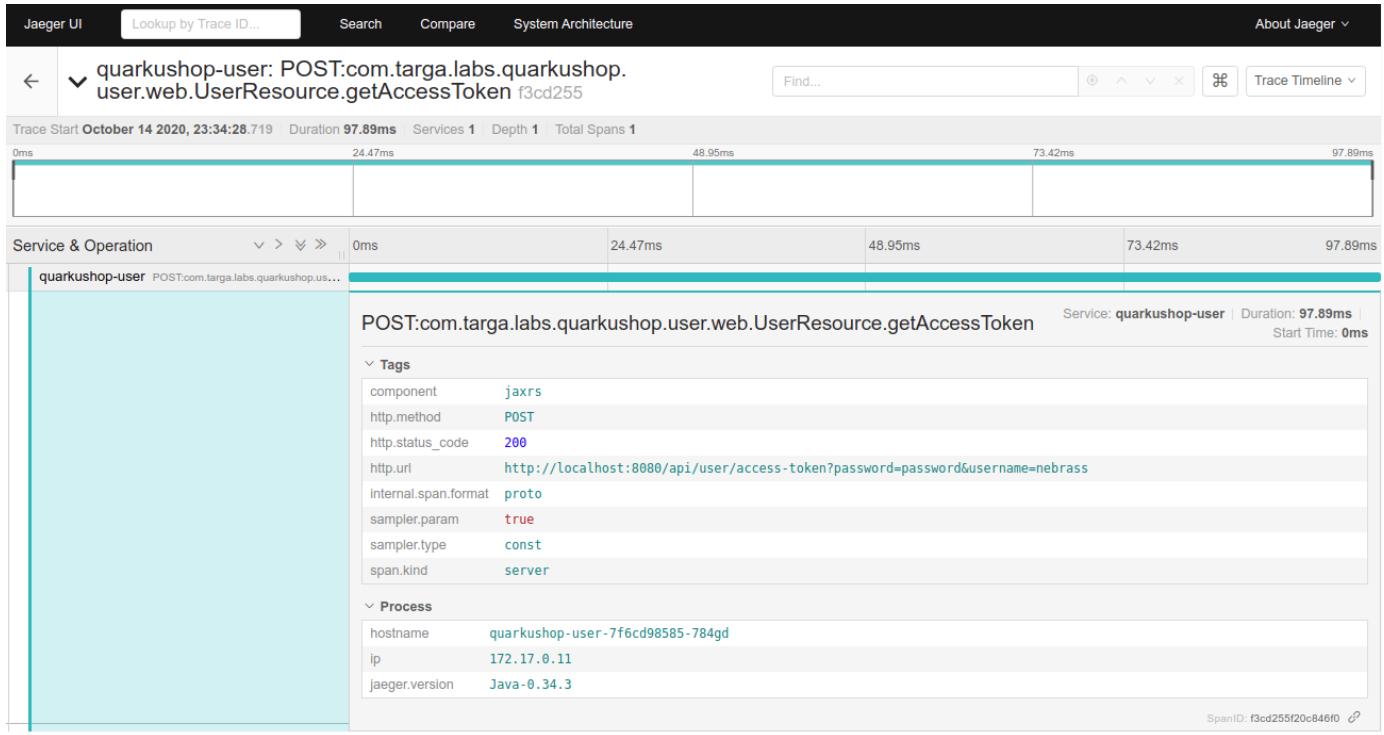
Then open the **localhost:8888**:

As you can notice, in the **Service** section, there are 3 elements - just select `quarkushop-user` and hit **[Find Traces]**:

Here there is **1 Span** shown:



If you click on it, more details will be shown:



Here you can see that all the details of the request are shown here:

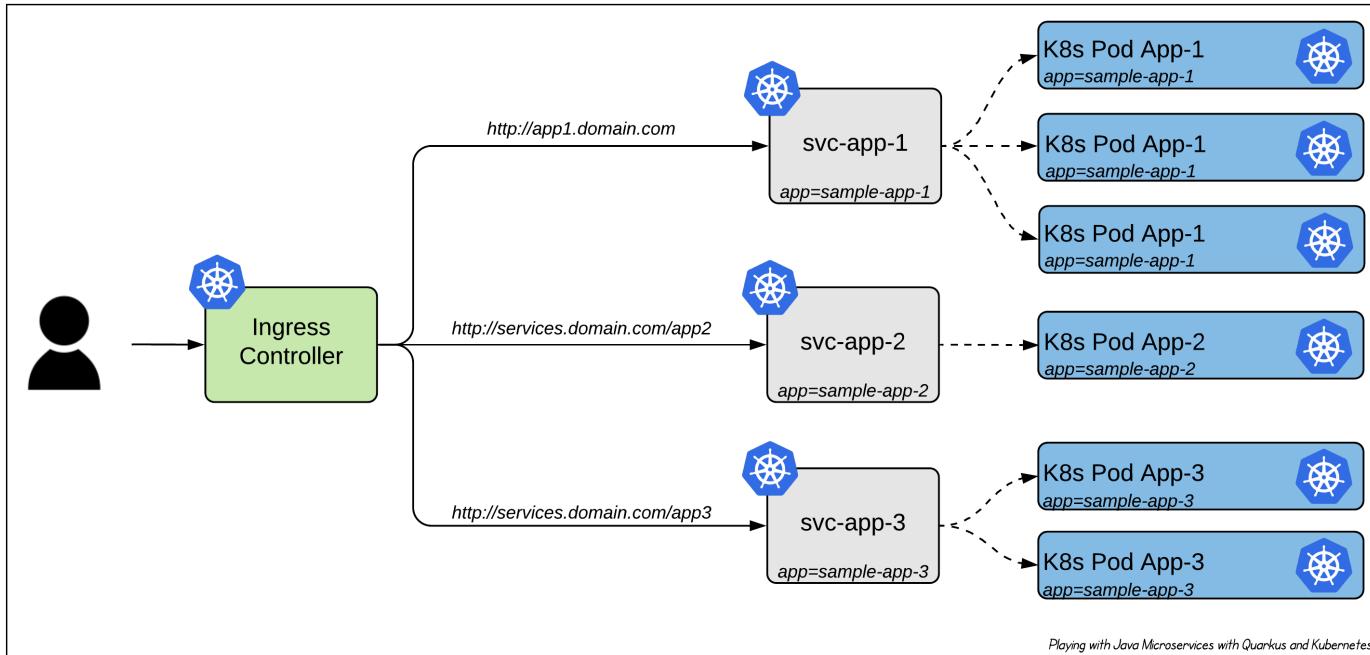
- the **URL**
- the **HTTP Verb**
- the **Duration**
- etc. *Every thing is here 😊*

Done! We finished implementing the **Distributed Tracing** pattern in **Quarkus** in a very efficient and easy way! I'm really happy! 😊

Implementing the API Gateway pattern

An **API Gateway** is a programming facade that sits in front of APIs and acts as a single point of entry for a defined group of services.

For implementing **Kubernetes Ingress** manages external access to the services in a cluster, typically *facadeHTTP*. **Ingress** can provide *load balancing, SSL termination and name-based virtual hosting*.



Playing with Java Microservices with Quarkus and Kubernetes

An **Ingress** is a collection of rules that allow inbound connections to reach the cluster services. It can be configured to give services *externally-reachable URLs, load balance traffic, terminate SSL, offer name based virtual hosting, and more*.

An **Ingress Controller** is responsible for fulfilling the **Ingress**, usually with a **LoadBalancer**, though it may also configure your **edge router** or additional frontends to help handle the traffic in an *High Availability* manner.

Let's bring our **API Gateway** to **Kubernetes** 😊

Step 1: Enable the Ingress support in Minikube

The first step is to enable the **Ingress** support in **Minikube**. This step is not required for those (lucky people) using real **Kubernetes Clusters** 😊

To enable the **Ingress** support in **Minikube**, just start your **minikube** instance and then do:

```
minikube addons enable ingress
```



Ingress is available as an Addon for **Minikube** 😊

We need a domain name for the **ingress**; we want to use the **quarkushop.io** domain name.

So, we need to get the IP address of **Minikube** by typing:

```
$ minikube ip
192.168.39.243
```

Then, we need to add a new entry to the **/etc/hosts** file to this IP address, in order to use it for our custom domain **quarkushop.io**:

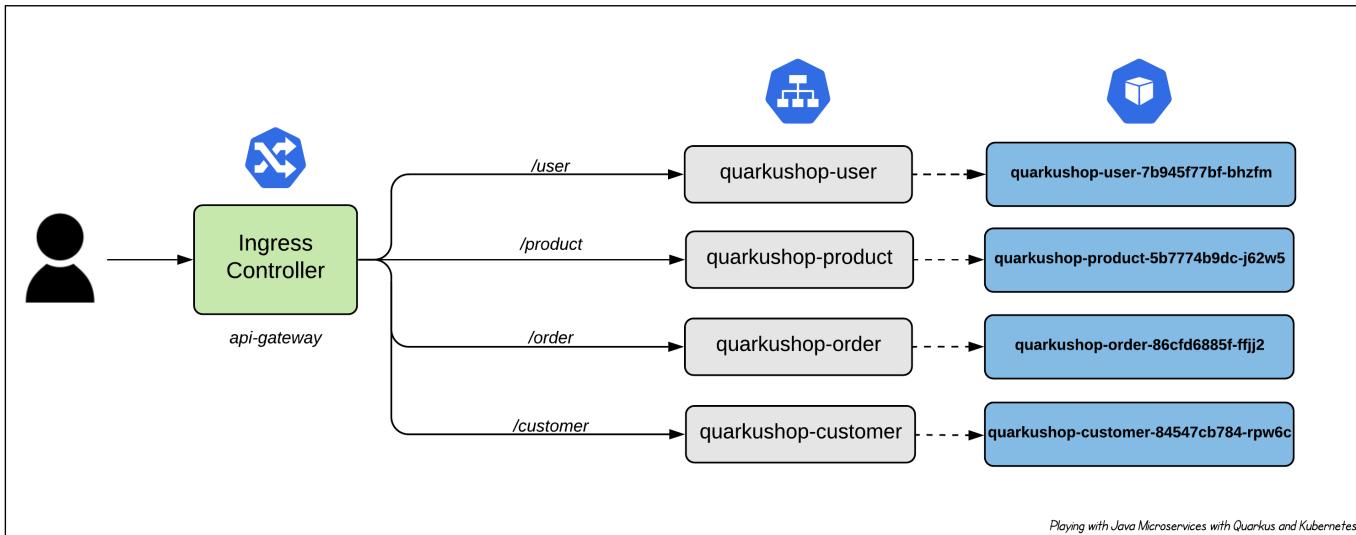
```
192.168.39.243 quarkushop.io
```

Now this custom internal *DNS entry*, will make any call to **quarkushop.io** be targeting **192.168.39.243** ⓘ

Step 2: Create the API Gateway Ingress

Our **Ingress** will point on our 4 µservices:

- **quarkushop-product**
- **quarkushop-order**
- **quarkushop-customer**
- **quarkushop-user**



Our **Ingress** descriptor will look like:

api-gateway-ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-gateway
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
  - http:
    paths:
    - path: /user
      pathType: Prefix
      backend:
        service:
          name: quarkushop-user
          port:
            number: 8080
    - path: /product
      pathType: Prefix
      backend:
        service:
          name: quarkushop-product
          port:
            number: 8080
    - path: /order
      pathType: Prefix
      backend:
        service:
          name: quarkushop-order
          port:
            number: 8080
    - path: /customer
      pathType: Prefix
      backend:
        service:
          name: quarkushop-customer
          port:
            number: 8080
```

Just save this content to **api-gateway-ingress.yml** and to create the resource with the command:

```
kubectl create -f api-gateway-ingress.yml
```

The **Ingress** is created successfully! Let's check it:

```
$ kubectl get ingress
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
api-gateway	<none>	quarkushop.io	192.168.39.243	80	7m46s

Excellent! As you can see the **ADDRESS** is the same as the **Minikube IP** 😊

Step 3: Test the Ingress

Now, we can enjoy our **Ingress**, we can use it to request an **access_token** from the **quarkushop-user** μservice:

```
$ curl -X POST "http://quarkushop.io/user/api/user/access-
token?password=password&username=nebrass"
eyJhbGciOiJSUzI1NiIsIn...
```

Ouppa! We got successfully the request **access_token**! 😊 The **Ingress** is working like a charm! 😊

Conclusion

In this chapter, we implemented many patterns using different **Quarkus Extensions** coupled with **Kubernetes Objects**. This task was quite easy especially that we delegated many tasks to **Kubernetes**.

Hakuna matata! 😊 We implemented successfully our **Cloud Native μservices**. I'm really so happy by the **Extensions** available and by the great documentation offered.

In the next chapter, we will discover how to increase our productivity and efficiency using  **Microsoft Azure**. 😊

CHAPTER FOURTEEN

Playing with Quarkus in Azure

To be Done Soon 😊

CHAPTER FIFTEEN

Bringing Dapr into the game

To be Done Soon 😊

Final words & thoughts

I hope that enjoyed reading this book. I tried to share with you my personal experience with the **Quarkus Framework**.

When I started writing this book, I just wanted to dig more into this new framework. But I never thought that **Quarkus** would be excellent! I really appreciated the wide range of available libraries implementing many patterns.

The **GraalVM** support in **Quarkus** is just incredible! I'm really excited by the performance and by the implementation abstraction of the **Native Image**.

About the author



Nebrass is a *Senior Software Engineer* at **Microsoft**, addicted to *Java* and *Cloud technologies*. He is a former **NetBeans Dream Team** member until December 2017.

Nebrass was one of the happy four winners of the **Oracle Groundbreaker Awards** in May 2019.

He is also a *Project Leader* in the **OWASP Foundation**, since March 2013, on the **Barbarus Project**.

He is the author of the books:

- **Playing with Java Microservices with Quarkus and Kubernetes** published with *Leanpub* in December 2020.
- **Playing with Java Microservices on Kubernetes and OpenShift** published with *Leanpub* in November 2018.
- **Pairing Apache Shiro with Java EE 7** published with *InfoQ* in May 2016.

Nebrass is graduated with an *M.Sc Degree* in *Information Systems Security* and a *Bachelor's Degree* in *Computing sciences & Management* from the *Higher Institute of Management of Tunis, Tunisia*.

Over the past 8 years, he has been working on *Java SE/EE projects*, in many sectors, including *Business Management, Petroleum, Finance & Banking, Medical & healthcare, and Defence & Space*. He has developed applications using many frameworks and *Java-related technologies*, such as *native Java EE APIs* and *3rd-party frameworks & tools* (*Spring, Quarkus, Hibernate, Primefaces, JBoss Forge*). He has been managing and using *infrastructure & programming tools* such as *DBMS, Java EE servers (Glassfish and JBoss), Quality & Continuous Integration/Deployment tools (Sonar, Jenkins and Azure DevOps), Docker & Kubernetes & Openshift...*

