Quarkus Workshop

Emmanuel Bernard, Clement Escoffier, Antonio Goncalves

Contents

Welcome

Let's start from the beginning... Quarkus. What's Quarkus? That's a pretty good question, and probably a good start. If you go on the Quarkus web site, Quarkus is "*A Kubernetes Native Java stack tailored for OpenJDK HotSpot & GraalVM, crafted from the best of breed Java libraries and standards*". This description is rather unclear, but does a very good job at using bankable keywords, right? It's also written: "Supersonic Subatomic Java". Still very foggy. In practice, Quarkus is an Open Source stack to write Java applications, specifically backend applications. In this lab, we are going to explain what Quarkus is, and, because the best way to understand Quarkus is to use it, build a set of microservices with it. Don't be mistaken, Quarkus is not limited to microservices, and we are going to learn about this in the workshop.

This lab offers attendees an intro-level, hands-on session with Quarkus, from the first line of code to making services, to consuming them, and finally to assembling everything in a consistent system. But, what are we going to build? Well, it's going to be a set of microservices (we want to be trendy):

- using Quarkus
- using HTTP and events (Kafka)
- with some parts of the dark side of microservices (monitoring (Prometheus), resilience)
- answer the ultimate question: are super heroes stronger than super villains?

This workshop is a BYOL (Bring Your Own Laptop) session, so bring your Windows, OSX, or Linux laptop. You need JDK 8+ on your machine, Apache Maven (3.6+), and Docker. On Windows, some parts may be qualified as *experimental*. On Mac and Windows, Docker for *x* is recommended instead of the Docker toolbox setup.

What you are going to learn:

- What is Quarkus and how you can use it
- How to build an HTTP endpoint (REST API) with Quarkus
- How to access a database
- How you can use Swagger and OpenAPI
- How you test your microservice
- How you improve the resilience of your service
- How to build event-driven and reactive microservices with Kafka
- How to build native executable
- How to extend Quarkus with extensions
- And many more...

Ready? Here we go!

Presenting the Workshop

What Is This Workshop About?

This workshop should give you a practical introduction to Quarkus. You will first install all the needed tools to then develop an entire microservice architecture, mixing classical HTTP microservices and event-based microservices. You will finish by extending the capabilities of Quarkus and learn more about the ability to create native executables.

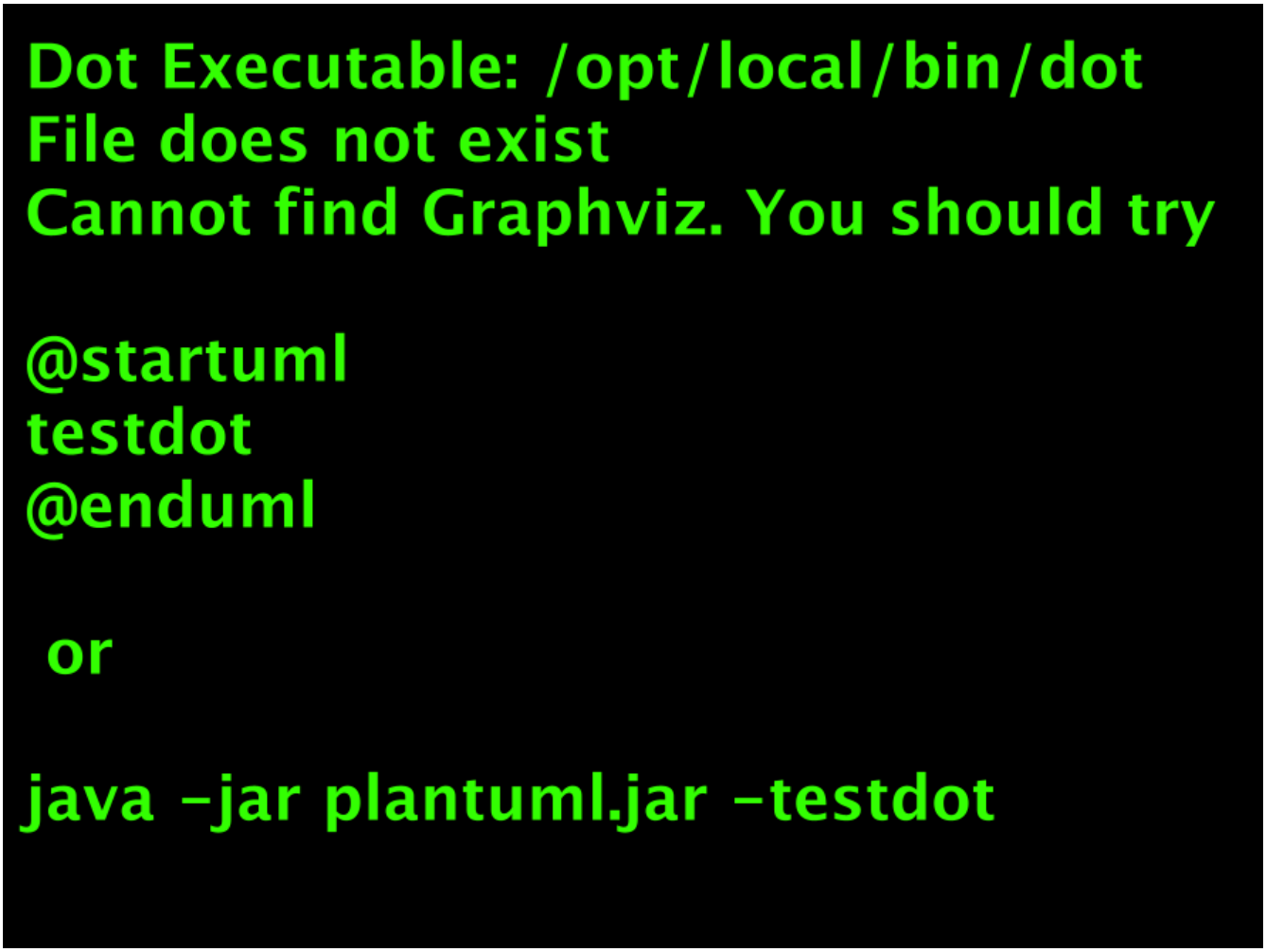
The idea is that you leave this workshop with a good understanding of what Quarkus is, what it is not, and how it can help you in your projects. Then, you'll be prepared to investigate a bit more and, hopefully, contribute.

NOTE	Get this workshop from https://github.com/quarkusio/quarkus-workshops/tree/master/quarkus-workshop-super-heroes

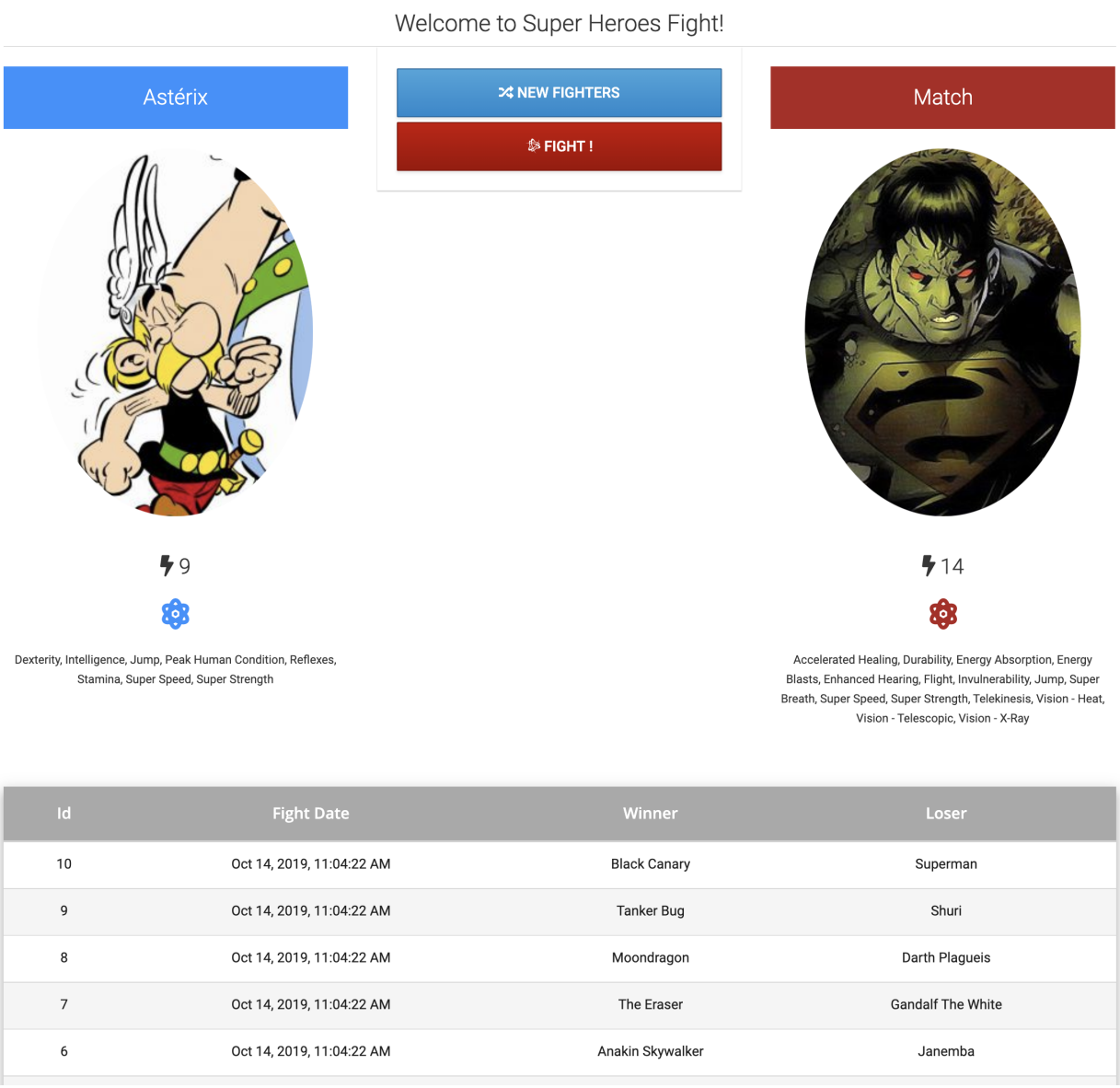
What Will You Be Developing?

In this workshop you will develop an application that allows super-heroes to fight against villains. Being a workshop about microservices, you will be developing several microservices communicating either synchronously via REST or asynchronously using Kafka:

- *Super Hero UI*: an Angular application allowing you to pick up a random super-hero, a random villain and makes them fight. The Super Hero UI is exposed via Quarkus and invokes the Fight REST API
- *Hero REST API*: Allows CRUD operations on Heroes which are stored in a Postgres database
- *Villain REST API*: Allows CRUD operations on Villains which are stored in a Postgres database
- *Fight REST API*: This REST API invokes the Hero and Villain APIs to get a random super-hero and a random villain. Each fight is stored in a Postgres database
- *Statistics*: Each fight is asynchronously sent (via Kafka) to the Statistics microservice. It has a HTML + JQuery UI displaying all the statistics.
- *Promotheus* polls metrics from the three microservices Fight, Hero and Villain



The main UI allows you to pick up one random Hero and Villain by clicking on "New Fighters". Then it's just a matter of clicking on "Fight!" to get them to fight. The table at the bottom shows the list of the previous fights.



How Does This Workshop Work?

You have this material in your hands (either electronically or printed) and you can now follow it step by step. The structure of this workshop is as follow :

- *Installing all the needed tools*: in this section you will install all the tools and code to be able to develop, compile and execute our application
- *Developing with Quarkus*: in this section you will develop a microservice architecture by creating several Maven projects, write some Java code, add JPA entities, JAX-RS REST endpoints, write some tests, use an Angular web application, and all that on Quarkus
- Extending Quarkus: in this section you will create a Quarkus extension

If you already have the tools installed, skip the *Installing all the needed tools* section and jump to the sections *Developing with Quarkus* and *Extending Quarkus*, and start hacking some code and addons. This "à la carte" mode allows you to make the most of this 6 hours long hands on lab.

What Do You Have to Do?

This workshop should be as self explanatory as possible. So your job is to follow the instructions by yourself, do what you are supposed to do, and do not hesitate to ask for any clarification or assistance, that's why the team is here. Oh, and be ready to have some fun!

Software Requirements

First of all, make sure you have a 64bits computer with admin rights (so you can install all the needed tools) and at least 8Gb of RAM (as some tools need a few resources).

WARNING	If you are using Mac OS X make sure the version is greater than 10.11.x (Captain).

This workshop will make use of the following software, tools, frameworks that you will need to install and now (more or less) how it works:

- Any IDE you feel comfortable with (eg. Intellij IDEA, Eclipse IDE, VS Code..)
- JDK 11
- GraalVM 21.0.0.2
- Maven 3.6.x
- Docker
- cURL (or any other command line HTTP client)
- Node JS (optional, only if you are in a *frontend* mood)

The next section focuses on how to install and setup the needed software. You can skip the next section if you have already installed all the prerequisites.

WARNING	This workshop assumes a bash shell. If you run on Windows in particular, adjust the commands accordingly.

Installing Software

JDK 11

Essential for the development and execution of this workshop is the *Java Development Kit* (JDK).[1] The JDK includes several tools such as a compiler (`javac`), a virtual machine, a documentation generator (`javadoc`), monitoring tools (Visual VM) and so on.[2] The code in this workshop uses JDK 11.

Installing the JDK

To install the JDK 11, go to the official website, select the appropriate platform and language, and download the distribution.[3] For example, if you are running on Mac OS X, download the DMG file (you should check out the *Accept License Agreement* check box before hitting the download link to let the download start). If you are not on Mac, the download steps are still pretty similar.

Instead of the Oracle distribution, you can use AdoptOpenJDK and download the JDK from https://adoptopenjdk.net. Follows the instructions from https://adoptopenjdk.net/installation.html to download and install the JDK for your platform.

There is also an easier way to download and install Java if you are on Mac OS X. You can use Homebrew to install JDK 11 using the following commands.[4]

```
$ brew tap caskroom/versions
$ brew cask install java8
```

Checking for Java Installation

Once the installation is complete, it is necessary to set the `JAVA_HOME` variable and the `$JAVA_HOME/bin` directory to the `PATH` variable. Check that your system recognises Java by entering `java -version` as well as the Java compiler with `javac -version`.

```
$ java -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
$ javac -version
javac 1.8.0_201
```

GraalVM

GraalVM is an extension of the *Java Virtual Machine* (JVM) to support more languages and several execution modes.[5] It supports a large set of languages: Java of course, other JVM-based languages (such as Groovy, Kotlin etc.) but also JavaScript, Ruby, Python, R and C/C++. It includes a new high performance Java compiler, itself called *Graal*, which can be used in a *Just-In-Time* (JIT) configuration on the HotSpot VM, or in an *Ahead-Of-Time* (AOT) configuration on the Substrate VM.[6] One objective of Graal is to improve the performance of Java virtual machine-based languages to

match the performance of native languages.

Prerequisites for GraalVM

On Linux, you need GCC, and the glibc and zlib headers. Examples for common distributions:

```
# dnf (rpm-based)
sudo dnf install gcc glibc-devel zlib-devel
# Debian-based distributions:
sudo apt-get install build-essential libz-dev zlib1g-dev
```

On MacOS X, XCode provides the required dependencies to build native executables:

```
xcode-select --install
```

Installing GraalVM

GraalVM installed from the GraalVM web site.[7] Using the community edition is enough. Version 21.0.0.2 is required.

Once installed, make sure the `GRAALVM_HOME` environment variable configured appropriately and points to the directory where GraalVM is installed (eg. on Mac OS X it will be /Library/Java/JavaVirtualMachines/graalvm-ce-21.0.0.2/Contents/Home) The `native-image` tool must be installed; this can be done by running `gu install native-image` from your GraalVM directory.

NOTE

Mac OS X - Catalina

On Mac OS X catalina, the installation of the `native-image` executable may fail. GraalVM binaries are not (yet) notarized for Catalina. To bypass the issue, it is recommended to run the following command instead of disabling macOS Gatekeeper entirely:

```
xattr -r -d com.apple.quarantine ${GRAAL_VM}
```

Checking for GraalVM Installation

Once installed and setup, you should be able to run the following command and get the following output.

```
$ $GRAALVM_HOME/bin/native-image --version
GraalVM Version 19.2.1 CE
```

Maven 3.6.x

All the examples of this workshop are built and tested using Maven.[8] Maven offers a building solution, shared libraries, and a plugin platform for your projects, allowing you to do quality

control, documentation, teamwork and so forth. Based on the "convention over configuration" principle, Maven brings a standard project description and a number of conventions such as a standard directory structure. With an extensible architecture based on plugins, Maven can offer many different services.

Installing Maven

The examples of this workshop have been developed with Apache Maven 3.6.x. Once you have installed JDK 11, make sure the `JAVA_HOME` environment variable is set. Then, download Maven from http://maven.apache.org/, unzip the file on your hard drive, and add the `apache-maven/bin` directory to your `PATH` variable. More details about the installation process is available on https://maven.apache.org/install.html.

But of course, if you are on Mac OS X and use Homebrew, just install Maven with the following command:

```
$ brew install maven
```

Checking for Maven Installation

Once you've got Maven installed, open a command line and enter `mvn -version` to validate your installation. Maven should print its version and the JDK version it uses (which is handy as you might have different JDK versions installed on the same machine).

```
$ mvn -version
Apache Maven 3.6.2
Maven home: /usr/local/Cellar/maven/3.6.2/libexec
Java version: 1.8.0_201, vendor: Oracle Corporation
OS name: "mac os x", version: "10.14.2", arch: "x86_64", family: "mac"
```

Be aware that Maven needs Internet access so it can download plugins and project dependencies from the Maven Central and/or other remote repositories.[9]

Some Maven Commands

Maven is a command line utility where you can use several parameters and options to build, test or package your code. To get some help on the commands you can type, use the following command:

```
$ mvn --help

usage: mvn [options] [<goal(s)>] [<phase(s)>]
```

Here are some commands that you will be using to run the examples in the workshop. Each invoke a different phase of the project life cycle (clean, compile, install etc.) and use the `pom.xml` to download libraries, customise the compilation, or extend some behaviours with plugins:

- `mvn clean`: Deletes all generated files (compiled classes, generated code, artifacts etc.).

- `mvn compile`: Compiles the main Java classes.
- `mvn test-compile`: Compiles the test classes.
- `mvn test`: Compiles the main Java classes as well as the test classes and executes the tests.
- `mvn package`: Compiles, executes the tests and packages the code into an archive.
- `mvn install`: Builds and installs the artifacts in your local repository.
- `mvn clean install`: Cleans and installs (note that you can add several commands separated by a space, like `mvn clean compile test`).

cUrl

To invoke the REST Web Services described in this workshop, we often use cURL.[10] cURL is a command line tool and library to do reliable data transfers with various protocols, including HTTP. It is free, open source (available under the MIT Licence) and has been ported to several operating systems.

Installing cURL

If you are on Mac OS X and if you have installed Homebrew, then installing cURL is just a matter of a single command.[11] Open your terminal and install cURL with the following command:

```
$ brew install curl
```

Checking for cURL Installation

Once installed, check for cURL by running `curl --version` in the terminal. It should display cURL version:

```
$ curl --version
curl 7.54.0 (x86_64-apple-darwin17.0) libcurl/7.54.0 LibreSSL/2.0.20 zlib/1.2.11
nghttp2/1.24.0
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtsp
smb smbs smtp smtps telnet tftp
Features: AsynchDNS IPv6 Largefile GSS-API Kerberos SPNEGO NTLM NTLM_WB SSL libz HTTP2
UnixSockets HTTPS-proxy
```

Some cURL Commands

cURL is a command line utility where you can use several parameters and options to invoke URLs. You invoke `curl` with zero, one or several command-line options to accompany the URL (or set of URLs) you want the transfer to be about. cURL supports over two hundred different options and I would recommend reading the documentation for more help.[12] To get some help on the commands and options you can type, use the following command:

```
$ curl --help

Usage: curl [options...] <url>
```

You can also opt to use `curl --manual` which will output the entire man page for cURL plus an appended tutorial for the most common use cases.

Here are some commands that you will be using to invoke the RESTful web service examples in this workshop.

- `curl http://localhost:8083/api/heroes/hello`: HTTP GET on a given URL.
- `curl -X GET http://localhost:8083/api/heroes/hello`: Same effect as the previous command, an HTTP GET on a given URL.
- `curl -v http://localhost:8083/api/heroes/hello`: HTTP GET on a given URL with verbose mode on.
- `curl -H 'Content-Type: application/json' http://localhost:8083/api/heroes/hello`: HTTP GET on a given URL passing the JSON Content Type in the HTTP Header.
- `curl -X DELETE http://localhost:8083/api/heroes/1`: HTTP DELETE on a given URL.

Formatting the cURL JSON Output

Very often when using cURL to invoke a RESTful web service, we get some JSON payload. cURL does not format this JSON, so you will get a flat String such as:

```
$ curl http://localhost:8083/api/heroes
[{"id":"1","name":"Chewbacca","level":"14"},{"id":"2","name":"Wonder
Woman","level":"15"},{"id":"3","name":"Anakin Skywalker","level":"8"}]
```

But what we really want is to format the JSON payload so it is easier to read. For that, there is a neat utility tool called jq that we could use. jq is a tool for processing JSON inputs, applying the given filter to its JSON text inputs and producing the filter's results as JSON on standard output.[13] You can install it on Mac OSX with a simple `brew install jq`. Once installed, it's just a matter of piping the cURL output to jq like this:

```
$ curl http://localhost:8083/api/heroes | jq
[
  {
    "id": "1",
    "name": "Chewbacca",
    "lastName": "14"
  },
  {
    "id": "2",
    "name": "Wonder Woman",
    "lastName": "15"
  },
  {
    "id": "3",
    "name": "Anakin Skywalker",
    "lastName": "8"
  }
]
```

Docker

Docker is a set of platform-as-a-service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels.

Installing Docker

Our infrastructure is going to use Docker to ease the installation of the different technical services (database, monitoring...). So for this, we need to install `docker` and `docker-compose` Installation instructions are available on the following page:

- Mac OS X - https://docs.docker.com/docker-for-mac/install/ (version 18.03+)
- Windows - https://docs.docker.com/docker-for-windows/install/ (version 18.03+)
- CentOS - https://docs.docker.com/install/linux/docker-ce/centos/
- Debian - https://docs.docker.com/install/linux/docker-ce/debian/
- Fedora - https://docs.docker.com/install/linux/docker-ce/fedora/
- Ubuntu - https://docs.docker.com/install/linux/docker-ce/ubuntu/

On Linux, don't forget the post-execution steps described on https://docs.docker.com/install/linux/linux-postinstall/.

Checking for Docker Installation

Once installed, check that both `docker` and `docker-compose` are available in your `PATH`:

```
$ docker version
Client: Docker Engine - Community
Version:           19.03.2
API version:       1.40
Go version:        go1.12.8
Git commit:        6a30dfc
Built:             Thu Aug 29 05:26:49 2019
OS/Arch:           darwin/amd64
Experimental:      false

Server: Docker Engine - Community
Engine:
Version:          19.03.2
API version:      1.40 (minimum version 1.12)
Go version:       go1.12.8
Git commit:       6a30dfc
Built:            Thu Aug 29 05:32:21 2019
OS/Arch:          linux/amd64
Experimental:     false
containerd:
Version:          v1.2.6
GitCommit:        894b81a4b802e4eb2a91d1ce216b8817763c29fb
runc:
Version:          1.0.0-rc8
GitCommit:        425e105d5a03fabd737a126ad93d62a9eeede87f
docker-init:
Version:          0.18.0
GitCommit:        fec3683

$ docker-compose version
docker-compose version 1.24.1, build 4667896b
docker-py version: 3.7.3
CPython version: 3.6.8
OpenSSL version: OpenSSL 1.1.0j  20 Nov 2018
```

Finally, run your first container as follows:

```
$ docker run hello-world

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.
(amd64)
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.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Some Docker Commands

Docker is a command line utility where you can use several parameters and options to start/stop a container. You invoke `docker` with zero, one or several command-line options with the container or image ID you want to work with. Docker comes with several options that are described in the documentation if you need more help.[14] To get some help on the commands and options you can type, use the following command:

```
$ docker help

Usage:  docker [OPTIONS] COMMAND

$ docker help attach

Usage:  docker attach [OPTIONS] CONTAINER

Attach local standard input, output, and error streams to a running container
```

Here are some commands that you will be using to start/stop containers in this workshop.

- `docker container ls`: Lists containers.
- `docker container start CONTAINER`: Starts one or more stopped containers.
- `docker-compose -f docker-compose.yaml up -d`: Starts all containers defined in a Docker Compose file.

- `docker-compose -f docker-compose.yaml down`: Stops all containers defined in a Docker Compose file.

Recap

Just make sure the following commands work on your machine

```
$ java -version
$ $GRAALVM_HOME/bin/native-image --version
$ mvn -version
$ curl --version
$ docker version
$ docker-compose version
```

Preparing for the Workshop

This workshop needs internet access to download all sorts of Maven artifacts, Docker images and even pictures. Some of these artifacts are large, and because we have to share internet connexions at the workshop, it is better to download them prior to the workshop. Here are a few commands that you can execute before the workshop.

Download the workshop scaffolding

[hand o right] Call to action

First, download the zip file https://raw.githubusercontent.com/dcavanau/quarkus-workshops/master/quarkus-workshop-super-heroes/dist/quarkus-super-heroes-workshop.zip, and unzip it wherever you want.

In this workshop you will be developing an application dealing with Super Heroes (and Super Villains) as well as Quarkus extensions. The code will be separated into two different directories:

```
quarkus-workshop-super-heroes
├── extensions                The Quarkus extensions
└── super-heroes              The entire Super Hero application
```

Super Heroes Application

Under the `super-heroes` directory you will find the entire Super Hero application spread throughout a set of subdirectories, each one containing a microservice or some tooling. The final structure will be the following:

```
quarkus-workshop-super-heroes
└── super-heroes
    ├── infrastructure        All the needed infrastructure (Postgres, Kafka...)
    ├── event-statistics      UI and application dealing with fight statistics (you will create it)
    ├── load-super-heroes     Stress tool loading heroes, villains and fights
    ├── rest-fight            REST API allowing super heroes to fight (you will create it)
    ├── rest-hero             REST API for CRUD operations on Heroes (you will create it)
    ├── rest-villain          REST API for CRUD operations on Villains
    └── ui-super-heroes       Angular application so we can fight visually
```

Most of theses subdirectories are Maven projects and follow the Maven directory structure:

```
quarkus-workshop-super-heroes
└── super-heroes
    └── rest-hero
        └── src
            ├── main
            │   ├── docker
            │   ├── java
            │   └── resources
            └── test
                └── java
```

Quarkus Extensions

Under the `extensions` directory you will find quarkus extensions. By the end of the workshop, you will get:

```
quarkus-workshop-super-heroes
  extensions
    extension-banner          Extension displaying a startup banner (you will create it)
    extension-fault-injector  Injects network latency and HTTP faults
```

Checking Ports

During this workshop we will use several ports.

[hand o right] Call to action

Just make sure the following ports are free so you don't run into any conflicts

```
$ lsof -i tcp:8080    // UI
$ lsof -i tcp:8082    // Fight REST API
$ lsof -i tcp:8083    // Hero REST API
$ lsof -i tcp:8084    // Villain REST API
$ lsof -i tcp:5432    // Postgres
$ lsof -i tcp:9090    // Prometheus
$ lsof -i tcp:2181    // Zookeeper
$ lsof -i tcp:9092    // Kafka
```

Warming up Maven

Now that you have the initial structure in place, navigate to the the root directory and run:

[hand o right] Call to action

```
mvn clean install
```

By running this command, it downloads all the required dependencies.

Warming up Docker

[hand o right] Call to action

To warm up your Docker image repository, navigate to the `quarkus-workshop-super-heroes/super-heroes/infrastructure` directory. Here, you will find a `docker-compose.yaml`/`docker-compose-linux.yaml` files which defines all the needed Docker images. Notice that there is a `db-init` directory with a `initialize-databases.sql` script which sets up our databases and a `monitoring` directory (all that will be explained later).

Linux User

If you are on Linux, use `docker-compose-linux.yaml` instead of `docker-compose`.yaml

Then execute `docker-compose -f docker-compose.yaml up -d` or `docker-compose -f docker-compose-linux.yaml up -d` on Linux. This will download all the Docker images and start the containers.

If you have an issue creating the roles for the database with the `initialize-databases.sql` file, you have to execute the following commands:

```
docker exec -it --user postgres super-database psql -c "CREATE ROLE
superman LOGIN PASSWORD 'superman' NOSUPERUSER INHERIT NOCREATEDB
NOCREATEROLE NOREPLICATION"
docker exec -it --user postgres super-database psql -c "CREATE ROLE
superbad LOGIN PASSWORD 'superbad' NOSUPERUSER INHERIT NOCREATEDB
NOCREATEROLE NOREPLICATION"
docker exec -it --user postgres super-database psql -c "CREATE ROLE
superfight LOGIN PASSWORD 'superfight' NOSUPERUSER INHERIT NOCREATEDB
NOCREATEROLE NOREPLICATION"
```

Once all the containers are up and running, you can shut them down with the commands:

```
docker-compose -f docker-compose.yaml down
docker-compose -f docker-compose.yaml rm
```

What's this infra?

Any microservice system is going to rely on a set of technical services. In our context, we are going to use PostgreSQL as the database, Prometheus as the monitoring tool, and Kafka as the event/message bus. This infrastructure starts all these services, so you don't have to worry about them.

Ready?

Prerequisites has been installed, the different components have been warmed up, it's now time to write some code!

[1] Java http://www.oracle.com/technetwork/java/javase

[2] Visual VM https://visualvm.github.io

[3] Java Website http://www.oracle.com/technetwork/java/javase/downloads/index.html

[4] Homebrew https://brew.sh

[5] GraalVM https://www.graalvm.org

[6] SubstrateVM https://github.com/oracle/graal/tree/master/substratevm

[7] GraalVM Download https://www.graalvm.org/downloads

[8] Maven https://maven.apache.org

[9] Maven Central https://search.maven.org

[10] cURL https://curl.haxx.se

[11] Homebrew https://brew.sh

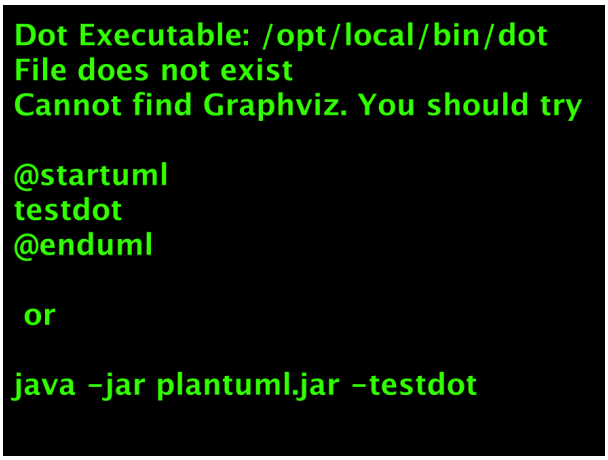
[12] cURL commands https://ec.haxx.se/cmdline.html

[13] jq https://stedolan.github.io/jq

[14] Docker commands https://docs.docker.com/engine/reference/commandline/cli

Creating a REST/HTTP Microservice

At the heart of the Super Hero application comes Heroes. We need to expose a REST API allowing CRUD operations on Super Heroes. This microservice is, let's say, a *classical* microservice. It uses HTTP to expose a REST API and internally store data into a database. This service will be used by the *fight* microservice.



In the following sections, you learn:

- how to create a new Quarkus application
- how to implement REST API using JAX-RS
- how to compose your application using CDI beans
- how to access your database using Hibernate with Panache
- how to use transactions
- how to enable OpenAPI and Swagger-UI

But first, let's describe our service. The Super Heroes microservice stores super-heroes, with their names, powers, and so on. The REST API allows adding, removing, listing, and picking a random hero from the stored set. Nothing outstanding but a good first step to discover Quarkus.

Hero Microservice

First thing first, we need a project. That's what your are going to see in this section.

Bootstrapping the Hero REST Endpoint

The easiest way to create a new Quarkus project is to use the Quarkus Maven plugin. We have created the project structure earlier, so we will move to the `rest-hero` directory and run the project creation command. Open a terminal and run the following command:

[hand o right] Call to action

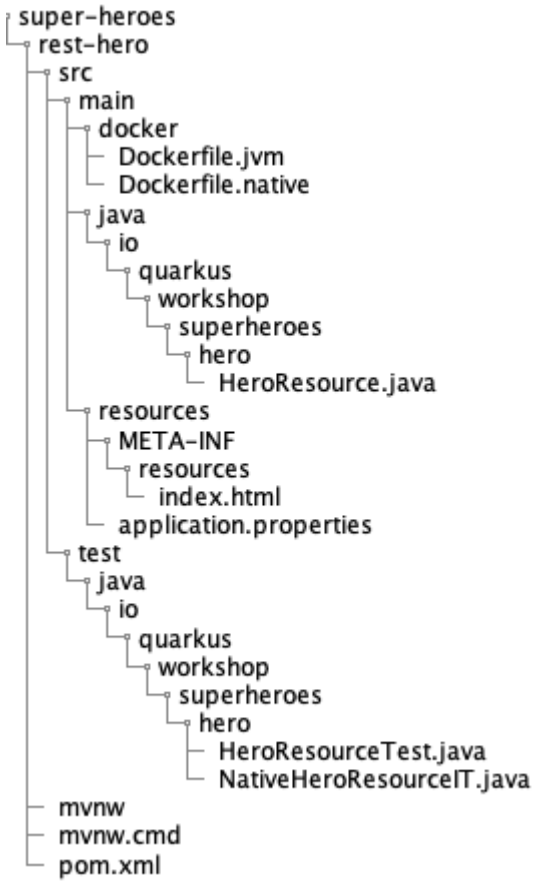
```
cd quarkus-workshop-super-heroes/super-heroes
mvn io.quarkus:quarkus-maven-plugin:1.13.0.Final:create \
    -DprojectGroupId=io.quarkus.workshop.super-heroes \
    -DprojectArtifactId=rest-hero \
    -DclassName="io.quarkus.workshop.superheroes.hero.HeroResource" \
    -Dpath="api/heroes"
```

Preferring Web UI

Instead of the Maven command, you can use https://code.quarkus.io.

Directory Structure

Once you bootstrap the project, you get the following directory structure with a few Java classes and other artifacts :



The Maven archetype generates the following `rest-hero` sub-directory:

- the Maven structure with a `pom.xml`
- an `io.quarkus.workshop.superheroes.hero.HeroResource` resource exposed on `/api/heroes`
- an associated unit test `HeroResourceTest`
- the landing page `index.html` that is accessible on http://localhost:8080 after starting the application
- example `Dockerfile` files for both native and jvm modes in `src/main/docker`
- the `application.properties` configuration file

Once generated, look at the `pom.xml`. You will find the import of the Quarkus BOM, allowing you to omit the version on the different Quarkus dependencies. In addition, you can see the `quarkus-maven-plugin` responsible of the packaging of the application and also providing the development mode.

If we focus on the dependencies section, you can see the extension allowing the development of REST applications:

RESTEasy

You may not be familiar with RESTEasy.[15] It's an implementation of JAX-RS and it uses to implement RestFul services in Quarkus.

What's this Quarkus Universe thingy?

The Quarkus Universe includes Quarkus as well as third-party extensions, like Apache Camel.

The JAX-RS Resource

During the project creation, the `HeroResource.java` file has been created with the following content:

```
package io.quarkus.workshop.superheroes.hero;

@Path("/api/heroes")
public class HeroResource {

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

It's a very simple REST endpoint, returning "hello" to requests on `/api/heroes`.

Running the Application

Now we are ready to run our application.

[hand o right] Call to action

Use: `./mvnw quarkus:dev`:

```
$ ./mvnw quarkus:dev
[INFO] Scanning for projects...
[INFO]
[INFO] -------------< io.quarkus.workshop.super-heroes:rest-hero >-------------
[INFO] Building rest-hero 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- quarkus-maven-plugin:1.9.2.Final:dev (default-cli) @ rest-hero ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/clement/Downloads/quarkus-super-heroes-
workshop/super-heroes/rest-hero/target/classes
Listening for transport dt_socket at address: 5005
2020-01-28 15:09:02,111 INFO  [io.quarkus] (main) rest-hero 1.0-SNAPSHOT (running on
Quarkus 1.9.2.Final) started in 2.161s. Listening on: http://0.0.0.0:8080
2020-01-28 15:09:02,123 INFO  [io.quarkus] (main) Profile dev activated. Live Coding
activated.
2020-01-28 15:09:02,124 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
```

Then check that the endpoint returns `hello` as expected:

[source,shell]Now we are ready to run our application

```
$ curl http://localhost:8080/api/heroes
hello
```

Alternatively, you can open http://localhost:8080/api/heroes in your browser.

Development Mode

`quarkus:dev` runs Quarkus in development mode. This enables hot deployment with background compilation, which means that when you modify your Java files and/or your resource files and invoke a REST endpoint (i.e. cUrl command or refresh your browser), these changes will automatically take effect. This works too for resource files like the configuration property and HTML files. Refreshing the browser triggers a scan of the workspace, and if any changes are detected, the Java files are recompiled and the application is redeployed; your request is then serviced by the redeployed application. If there are any issues with compilation or deployment an error page will let you know.

The development mode also allows debugging and listens for a debugger on port 5005. If you want to wait for the debugger to attach before running you can pass `-Dsuspend=true` on the command line. If you don't want the debugger at all you can use `-Ddebug=false`.

Alright, time to change some code. Open your favorite IDE and import the project. To check that the hot reload is working, update the method `HeroResource.hello()` by returning the String "hello hero". Now, execute the cUrl command again, the output has changed without you to having to stop and restart Quarkus:

[hand o right] Call to action

```
$ curl http://localhost:8080/api/heroes
hello hero
```

Testing the Application

All right, so far so good, but wouldn't it be better with a few tests, just in case.

In the generated `pom.xml` file, you can see 2 test dependencies:

```
```

Quarkus supports Junit 4 and Junit 5 tests. In the generated project, we use Junit 5. Because of this, the version of the Surefire Maven Plugin must be set, as the default version does not support Junit 5:

```
```

We also set the `java.util.logging` system property to make sure tests will use the correct log manager.

The generated project contains a simple test in `HeroResourceTest.java`.

```
package io.quarkus.workshop.superheroes.hero;

@QuarkusTest
public class HeroResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
          .when().get("/api/heroes")
          .then()
             .statusCode(200)
             .body(is("hello"));
    }
}
```

By using the `QuarkusTest` runner, the `HeroResourceTest` class instructs JUnit to start the application before the tests. Then, the `testHelloEndpoint` method checks the HTTP response status code and content. Notice that these tests use RestAssured, but feel free to use your favorite library.[16]

[hand o right] Call to action

Execute it with `./mvnw test` or from your IDE. It fails! It's expected, you changed the output of `HeroResource.hello()` earlier. Adjust the test body condition accordingly.

Packaging and Running the Application

[hand o right] Call to action

The application is packaged using `./mvnw package`. It produces 2 jar files in `/target`:

- `rest-hero-1.0-SNAPSHOT.jar` : containing just the classes and resources of the projects, it's the regular artifact produced by the Maven build;
- `rest-hero-1.0-SNAPSHOT-runner.jar` : being an executable jar. Be aware that it's not an über-jar as the dependencies are copied into the `target/lib` directory.

You can run the application using: `java -jar target/rest-hero-1.0-SNAPSHOT-runner.jar`.

Before running the application, don't forget to stop the hot reload mode (hit CTRL+C), or you will have a port conflict.

Troubleshooting

You might come across the following error while developing:

```
WARN  [io.qu.ne.ru.NettyRecorder] (Thread-48) Localhost lookup took more than one
second, you need to add a /etc/hosts entry to improve Quarkus startup time. See
https://thoeni.io/post/macos-sierra-java/ for details.
```

If this is the case, it's just a matter to add the node name of your machine to the /etc/hosts. For that, first get the name of your node with the following command:

```
$ uname -n
my-node.local
```

Then `sudo vi /etc/hosts` so you have the rights to edit the file and add the following entry

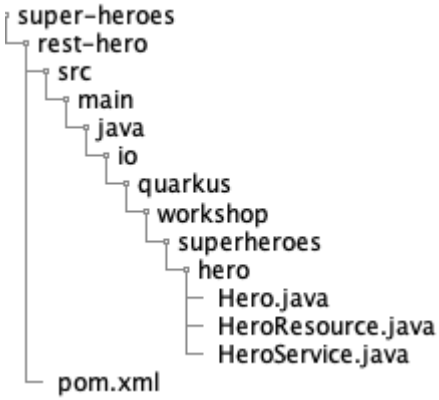
```
127.0.0.1 localhost my-node.local
```

Transactions and ORM

The Hero API's role is to allow CRUD operations on Super Heroes. In this module we will create a Hero entity and persist/update/delete/retrieve it from a Postgres database in a transactional way.

Directory Structure

In this module we will add extra classes to the Hero API project. You will end-up with the following directory structure:



Installing the PostgreSQL Dependency, Hibernate with Panache and Hibernate Validator

This microservice:

- interacts with a PostGreSQL database - so it needs a driver
- uses Hibernate with Panache - so need the dependency on it
- validates payloads and entities - so need a validator
- consumes and produces JSON - so we need a mapper

Hibernate ORM is the de-facto JPA implementation and offers you the full breadth of an Object Relational Mapper. It makes complex mappings possible, but it does not make simple and common mappings trivial. Hibernate ORM with Panache focuses on making your entities trivial and fun to write in Quarkus.[17]

Because JPA and Bean Validation work well together, we will use Bean Validation to constrain our business model.

To add the required dependencies, just run the following command:

[hand o right] Call to action

```
$ ./mvnw quarkus:add-extension -Dextensions="jdbc-postgresql,hibernate-orm
-panache,hibernate-validator,resteasy-jsonb"
```

This will add the following dependencies in the `pom.xml` file:

```
```

From now on, you can choose to either edit your pom directly or use the `quarkus:add-extension` command.

Running the Infrastructure

Before going further, be sure to run the infrastructure. To execute this service, you need a database (and later on we will need Prometheus and Kafka). Let's use Docker and docker compose to ease the installation of such infrastructure.

You should already have installed the infrastructure into the `infrastructure` directory. Now, just execute `docker-compose -f docker-compose.yaml up -d`. You should see a few logs going on and then all the containers get started.

[hand o right] Call to action

On Linux, use the `docker-compose-linux.yaml`:

Listing 1. On Linux

```
docker-compose -f docker-compose-linux.yaml up -d
```

During the workshop, just leave all the containers up and running. Then, after the workshop, remember to shut them down using: `docker-compose -f docker-compose.yaml down` or `docker-compose -f docker-compose-linux.yaml down` on Linux.

Hero Entity

[hand o right] Call to action

To define a Panache entity, simply extend `PanacheEntity`, annotate it with `@Entity` and add your columns as public fields (no need to have getters and setters). The `Hero` entity should look like this:

```
```

Notice that you can put all your JPA column annotations and Bean Validation constraint annotations on the public fields.

Adding Operations

Thanks to Panache, once you have written the `Hero` entity, here are the most common operations you will be able to do:

```
// creating a hero
Hero hero = new Hero();
hero.name = "Superman";
hero.level = 9;

// persist it
hero.persist();

// getting a list of all Hero entities
List<Hero> heroes = Hero.listAll();

// finding a specific hero by ID
hero = Hero.findById(id);

// counting all heroes
long countAll = Hero.count();
```

But we are missing a business method: we need to return a random hero. For that it's just a matter to add the following method to our `Hero.java` entity:

```

```

You would need to add the following import statement if not done automatically by your IDE `import java.util.Random;`

Configuring Hibernate

Quarkus development mode is really useful for applications that mix front end or services and database access. We use `quarkus.hibernate-orm.database.generation=drop-and-create` in conjunction with `import.sql` so every change to your app and in particular to your entities, the database schema will be properly recreated and your data (stored in `import.sql`) will be used to repopulate it from scratch. This is best to perfectly control your environment and works magic with Quarkus live reload mode: your entity changes or any change to your `import.sql` is immediately picked up and the schema updated without restarting the application!

[hand o right] Call to action

For that, make sure to have the following configuration in your `application.properties` (located in `src/main/resources`):

```

```

HeroService Transactional Service

To manipulate the `Hero` entity we will develop a transactional `HeroService` class. The idea is to wrap methods modifying the database (e.g. `entity.persist()`) within a transaction. Marking a CDI bean method `@Transactional` will do that for you and make that method a transaction boundary.

`@Transactional` can be used to control transaction boundaries on any CDI bean at the method level or at the class level to ensure every method is transactional. You can control whether and how the transaction is started with parameters on `@Transactional`:

- `@Transactional(REQUIRED)` (default): starts a transaction if none was started, stays with the existing one otherwise.
- `@Transactional(REQUIRES_NEW)`: starts a transaction if none was started ; if an existing one was started, suspends it and starts a new one for the boundary of that method.
- `@Transactional(MANDATORY)`: fails if no transaction was started ; works within the existing transaction otherwise.
- `@Transactional(SUPPORTS)`: if a transaction was started, joins it ; otherwise works with no transaction.
- `@Transactional(NOT_SUPPORTED)`: if a transaction was started, suspends it and works with no transaction for the boundary of the method ; otherwise works with no transaction.
- `@Transactional(NEVER)`: if a transaction was started, raises an exception ; otherwise works with no transaction.

[hand o right] Call to action

Creates a new `HeroService.java` file in the same package with the following content:

Notice that both methods that persist and update a hero, pass a `Hero` object as a parameter. Thanks to the Bean Validation's `@Valid` annotation, the `Hero` object will be checked to see if it's valid or not. It it's not, the transaction will be rollback-ed.

Configuring the Datasource

Our project now requires a connection to a PostgreSQL database. The main way of obtaining connections to a database is to use a datasource. In Quarkus, the out of the box datasource and connection pooling implementation is Agroal.[18]

This is done in the `src/main/resources/application.properties` file.

[hand o right] Call to action

Just add the following datasource configuration:

HeroResource Endpoint

The `HeroResource` Endpoint was bootstrapped with only one method `hello()`. We need to add extra methods that will allow CRUD operations on heroes.

[hand o right] Call to action

Here are the new methods to add to the `HeroResource` class:

```
```

Dependency Injection

Dependency injection in Quarkus is based on ArC which is a CDI-based dependency injection solution tailored for Quarkus' architecture.[19] You can learn more about it in the Contexts and Dependency Injection guide.[20]

ArC comes as a dependency of `quarkus-resteasy` so you already have it handy. That's why you were able to use `@Inject` in the `HeroResource` to inject a reference to `HeroService`.

Adding Data

[hand o right] Call to action

To load some SQL statements when Hibernate ORM starts, add the following `import.sql` in the root of the `resources` directory. It contains SQL statements terminated by a semicolon. This is useful to have a data set ready for the tests or demos.

```
INSERT INTO hero(id, name, otherName, picture, powers, level)
VALUES (nextval('hibernate_sequence'), 'Chewbacca', '',
'https://www.superherodb.com/pictures2/portraits/10/050/10466.jpg', 'Agility,
Longevity, Marksmanship, Natural Weapons, Stealth, Super Strength, Weapons Master', 5
);
INSERT INTO hero(id, name, otherName, picture, powers, level)
VALUES (nextval('hibernate_sequence'), 'Angel Salvadore', 'Angel Salvadore Bohusk',
'https://www.superherodb.com/pictures2/portraits/10/050/1406.jpg', 'Animal Attributes,
Animal Oriented Powers, Flight, Regeneration, Toxin and Disease Control', 4);
INSERT INTO hero(id, name, otherName, picture, powers, level)
VALUES (nextval('hibernate_sequence'), 'Bill Harken', '',
'https://www.superherodb.com/pictures2/portraits/10/050/1527.jpg', 'Super Speed, Super
Strength, Toxin and Disease Resistance', 6);
```

Ok, but that's just a few entries. Download the SQL file import.sql and copy it under `src/main/resources`. Now, you have around 500 heroes that will be loaded in the database.

If you didn't yet, start the application in dev mode:

```
$./mvnw quarkus:dev
```

Then, open your browser to http://localhost:8080/api/heroes. You should see lots of heroes...

CRUD Tests in HeroResourceTest

To test the `HeroResource` endpoint, we will be using a `QuarkusTestResource` that will fire a Postgres

database and then test CRUD operations. The `QuarkusTestResource` is a test extension that can configure the environment before running the application. In our context, we will be using TestContainers to start our database.[21]

[hand o right] Call to action

For that we will install the TestContainers dependency in our `pom.xml` as well as some extra test dependencies:

```

```

[hand o right] Call to action

Then, you need to create the `QuarkusTestResource`. Create the `io.quarkus.workshop.superheroes.hero.DatabaseResource` classes with the following content:

```
package io.quarkus.workshop.superheroes.hero;

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;
import org.testcontainers.containers.PostgreSQLContainer;

import java.util.Collections;
import java.util.Map;

public class DatabaseResource implements QuarkusTestResourceLifecycleManager {

    private static final PostgreSQLContainer DATABASE = new PostgreSQLContainer<>(
"postgres:10.5")
        .withDatabaseName("heroes_database")
        .withUsername("superman")
        .withPassword("superman")
        .withExposedPorts(5432);

    @Override
    public Map<String, String> start() {
        DATABASE.start();
        return Collections.singletonMap("quarkus.datasource.jdbc.url", DATABASE
.getJdbcUrl());
    }

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

This code just starts the database before the application, and set the `quarkus.datasource.jdbc.url` system property to indicate to the application where is the test database. Note that the database would use a random port.

[hand o right] Call to action

Then, in `io.quarkus.workshop.superheroes.hero.HeroResourceTest`, you will add the following test methods to the `HeroResourceTest` class:

- `shouldNotGetUnknownHero`: giving a random Hero identifier, the `HeroResource` endpoint should return a 204 (No content)
- `shouldGetRandomHero`: checks that the `HeroResource` endpoint returns a random hero
- `shouldNotAddInvalidItem`: passing an invalid `Hero` should fail when creating it (thanks to the `@Valid` annotation)
- `shouldGetInitialItems`: checks that the `HeroResource` endpoint returns the list of heroes
- `shouldAddAnItem`: checks that the `HeroResource` endpoint creates a valid `Hero`
- `shouldUpdateAnItem`: checks that the `HeroResource` endpoint updates a newly created `Hero`
- `shouldRemoveAnItem`: checks that the `HeroResource` endpoint deletes a hero from the database

The code is as follow:

```
```

Let's have a look to the 2 annotations used on the `HeroResourceTest` class. `@QuarkusTest` indicates that this test class is checking the behavior of a Quarkus application. The test framework starts the application before the test class and stops it once all the tests have been executed. The tests and the application runs in the same JVM, meaning that the test can be injected with application *beans*. This feature is very useful to test specific parts of the application. However in our case, we just execute HTTP requests to check the result.

Notice also the `@QuarkusTestResource(DatabaseResource.class)`. It is how the `QuarkusTestResource` are attached to a test class.

[hand o right] Call to action

With this code written, execute the test using `./mvnw test`. The test should pass.

Configuring the Hero Microservice

Hardcoded values in our code are a no go (even if we all did it at some point ;-)). In this guide, we learn how to configure our Hero API as well as some parts of Quarkus.

Configuring Logging

Run time configuration of logging is done through the normal `application.properties` file.

Configuring Quarkus Listening Port

Because we will end-up running several microservices, let's configure Quarkus so it listens to a different port than 8080: This is quite easy as we just need to add one property in the `application.properties` file:

[hand o right] Call to action

Changing the port is one of the rare configuration that cannot be done while the application is running.

[hand o right] Call to action

You would need to restart the application to change the port.

Hit `CTRL+C` to stop the application and restart it with: `./mvnw quarkus:dev`

Injecting Configuration Value

When we persist a new hero we want to multiply its level by a value that can be configured. For this, Quarkus uses MicroProfile Config to inject the configuration in the application.[22] The injection uses the `@ConfigProperty` annotation.

When injecting a configured value, you can use `@Inject @ConfigProperty` or just `@ConfigProperty`. The `@Inject` annotation is not necessary for members annotated with `@ConfigProperty`, a behavior which differs from MicroProfile Config.

[hand o right] Call to action

Edit the `HeroService`, and introduce the following configuration properties:

You may need to add the following import statement if your IDE does not do it automatically: `import org.eclipse.microprofile.config.inject.ConfigProperty;`

- If you do not provide a value for this property, the application startup fails with `javax.enterprise.inject.spi.DeploymentException: No config value of type [int] exists for: level.multiplier`
- A default value (property `defaultValue`) is injected if the configuration does not provide a value for `level.multiplier`

[hand o right] Call to action

Now, modify the `HeroService.persistHero()` method to use the injected properties:

```
```

Create the Configuration

By default, Quarkus reads `application.properties`.

[hand o right] Call to action

Edit the `src/main/resources/application.properties` with the following content:

```
```

Running and Testing the Application

[hand o right] Call to action

If you didn't already, start the application with `./mvnw quarkus:dev`. Once started, create a new hero with the following cUrl command:

```
$ curl -X POST -d  '{"level":5, "name":"Chewbacca", "powers":"Agility, Longevity"}'
-H "Content-Type: application/json" http://localhost:8083/api/heroes -v

< HTTP/1.1 201 Created
< Location: http://localhost:8083/api/heroes/952
```

As you can see, we've passed a level of 5 to create this new hero. The cUrl command returns the location of the newly created hero. Take this URL and do an HTTP GET on it. You will see that the level has been increased.

```
$ curl http://localhost:8083/api/heroes/952 | jq

{
  "id": 957,
  "level": 15,
  "name": "Chewbacca",
  "powers": "Agility, Longevity"
}
```

You may not know `jq`. It's an amazing tool to manipulate JSON in the shell. More info on: https://stedolan.github.io/jq/

Hey! Wait a minute! Tests are failing now! Indeed, they don't know the multiplier.

[hand o right] Call to action

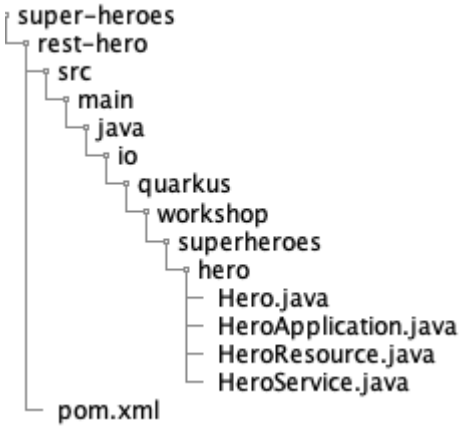
In the `application.properties` file, add: `%test.level.multiplier=1` which set the multiplier to 1 when running the tests. We will cover the `%test` syntax soon.

Open API

By default, a Quarkus application exposes its API description through an OpenAPI specification. Quarkus also lets you test it via a user-friendly UI named Swagger UI.

Directory Structure

In this module we will add extra class (`HeroApplication`) to the Hero API project. You will end-up with the following directory structure:



Installing the OpenAPI Dependency

Quarkus proposes a `smallrye-openapi` extension compliant with the Eclipse MicroProfile OpenAPI specification in order to generate your API OpenAPI v3 specification.[23]

[hand o right] Call to action

To install the OpenAPI dependency, just run the following command:

```
$ ./mvnw quarkus:add-extension -Dextensions="smallrye-openapi"
```

This will add the following dependency in the `pom.xml` file:

```
```

Open API

Now, you *curl* `http://localhost:8083/openapi` endpoint:

```
$ curl http://localhost:8083/openapi
---
openapi: 3.0.1
info:
  title: Generated API
  version: "1.0"
paths:
```

```
  /api/heroes:
    get:
      responses:
        200:
          description: OK
    put:
      responses:
        200:
          description: OK
    post:
      responses:
        200:
          description: OK
  /api/heroes/hello:
    get:
      responses:
        200:
          description: OK
          content:
            text/plain:
              schema:
                $ref: '#/components/schemas/String'
  /api/heroes/random:
    get:
      responses:
        200:
          description: OK
  /api/heroes/{id}:
    get:
      parameters:
      - name: id
        in: path
        required: true
        schema:
          $ref: '#/components/schemas/Long'
      responses:
        200:
          description: OK
    delete:
      parameters:
      - name: id
        in: path
        required: true
        schema:
          $ref: '#/components/schemas/Long'
      responses:
        200:
          description: OK
components:
  schemas:
    Long:
```

```
      format: int64
      type: integer
    String:
      type: string
```

This contract lacks of documentation. The Eclipse MicroProfile OpenAPI allows you to customize the methods of your REST endpoint as well as the application.

Customizing Methods

The MicroProfile OpenAPI has a set of annotations to customize each REST endpoint method so the OpenAPI contract is richer and clearer for consumers:

- `@Operation`: Describes a single API operation on a path.
- `@APIResponse`: Corresponds to the OpenAPI Response model object which describes a single response from an API Operation
- `@Parameter`: The name of the parameter.
- `@RequestBody`: A brief description of the request body.

This is what the `HeroResource` endpoint looks like once annotated

```

```

Customizing the Application

The previous annotations allow you to customize the contract for a given REST Endpoint. But it's also important to customize the entire application. The Microprofile OpenAPI also has a set of annotation to do so. The difference is that these annotations cannot be used on the Endpoint itself, but instead on another Java class configuring the entire application.

[hand o right] Call to action

For this, you need to create the `src/main/java/io/quarkus/workshop/superheroes/hero/HeroApplication` class with the following content:

```

```

Customized Contract

If you go back to the `http://localhost:8083/openapi` endpoint you will see the following OpenAPI contract:

```
---
openapi: 3.0.1
info:
  title: Hero API
  description: This API allows CRUD operations on a hero
```

```
  contact:
    name: Quarkus
    url: https://github.com/quarkusio
  version: "1.0"
externalDocs:
  description: All the Quarkus workshops
  url: https://github.com/quarkusio/quarkus-workshops
servers:
- url: http://localhost:8083
tags:
- name: api
  description: Public that can be used by anybody
- name: heroes
  description: Anybody interested in heroes
paths:
  /api/heroes:
    get:
      summary: Returns all the heroes from the database
      responses:
        204:
          description: No heroes
        200:
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Hero'
    put:
      summary: Updates an exiting  hero
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Hero'
        required: true
      responses:
        200:
          description: The updated hero
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Hero'
    post:
      summary: Creates a valid hero
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Hero'
```

```
        required: true
      responses:
        201:
          description: The URI of the created hero
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/URI'
  /api/heroes/hello:
    get:
      responses:
        200:
          description: OK
          content:
            text/plain:
              schema:
                $ref: '#/components/schemas/String'
  /api/heroes/random:
    get:
      summary: Returns a random hero
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                description: The hero fighting against the villain
                required:
                - level
                - name
                type: object
                properties:
                  id:
                    format: int64
                    type: integer
                  level:
                    format: int32
                    minimum: 1
                    type: integer
                    nullable: false
                  name:
                    maxLength: 50
                    minLength: 3
                    type: string
                    nullable: false
                  otherName:
                    type: string
                  picture:
                    type: string
                  powers:
                    type: string
```

```
  /api/heroes/{id}:
    get:
      summary: Returns a hero for a given identifier
      parameters:
      - name: id
        in: path
        description: Hero identifier
        required: true
        schema:
          $ref: '#/components/schemas/Long'
      responses:
        204:
          description: The hero is not found for a given identifier
        200:
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Hero'
    delete:
      summary: Deletes an exiting hero
      parameters:
      - name: id
        in: path
        description: Hero identifier
        required: true
        schema:
          $ref: '#/components/schemas/Long'
      responses:
        204:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Hero'
components:
  schemas:
    Hero:
      description: The hero fighting against the villain
      required:
      - level
      - name
      type: object
      properties:
        id:
          format: int64
          type: integer
        level:
          format: int32
          minimum: 1
          type: integer
          nullable: false
```

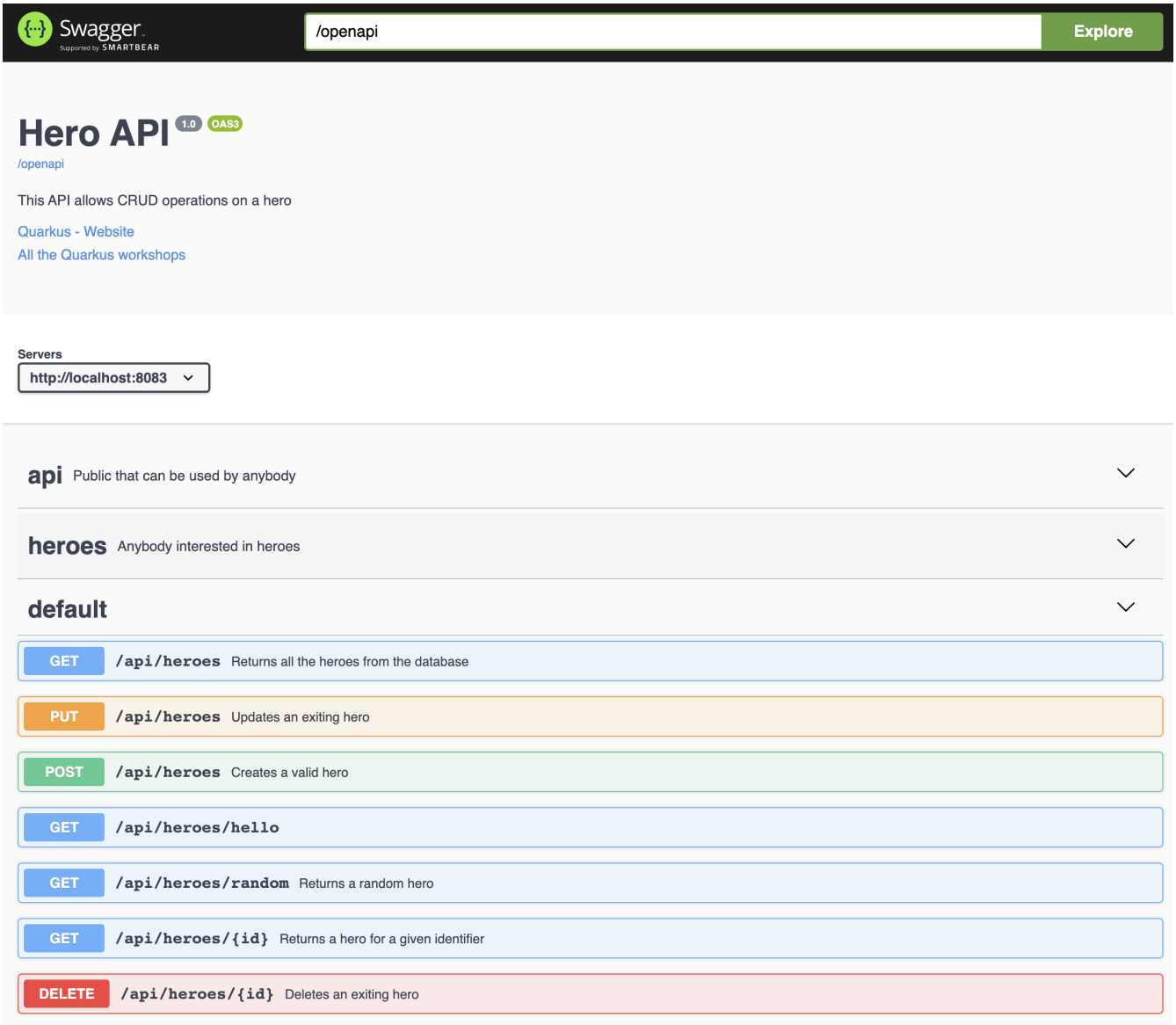
```
        name:
          maxLength: 50
          minLength: 3
          type: string
          nullable: false
        otherName:
          type: string
        picture:
          type: string
        powers:
          type: string
    URI:
      type: object
      properties:
        string:
          type: string
        rawAuthority:
          type: string
        rawFragment:
          type: string
        rawPath:
          type: string
        rawQuery:
          type: string
        rawSchemeSpecificPart:
          type: string
        rawUserInfo:
          type: string
        absolute:
          type: boolean
        opaque:
          type: boolean
    Long:
      format: int64
      type: integer
    String:
      type: string
```

Swagger UI

When building APIs, developers want to test them quickly. Swagger UI is a great tool permitting to visualize and interact with your APIs.[24] The UI is automatically generated from your OpenAPI specification. The Quarkus `smallrye-openapi` extension comes with a swagger-ui extension embedding a properly configured Swagger UI page. By default, Swagger UI is accessible at `/swagger-ui`. So, once your application is started, you can go to `http://localhost:8083/swagger-ui` and play with your API.

[hand o right] Call to action

You can visualize your API's operations and schemas.



For example, you can try the `/api/heroes/random` endpoint to retrieve a random hero.

OpenAPI Tests in HeroResourceTest

[hand o right] Call to action

Let's add a few extra test methods in `HeroResourceTest` that would make sure OpenAPI and Swagger UI are packaged in the application:

```
```

[hand o right] Call to action

Execute the test using `./mvnw test`.

If you have any problem with the code, don't understand or feel you are running, remember to ask for some help. Also, you can get the code of this entire workshop from https://github.com/quarkusio/quarkus-workshops/tree/master/quarkus-workshop-super-heroes.

[15] RESTEasy https://resteasy.github.io
[16] RestAssured http://rest-assured.io
[17] Panache https://github.com/quarkusio/quarkus/tree/master/extensions/panache
[18] Agroal https://agroal.github.io
[19] ArC https://github.com/quarkusio/quarkus/tree/master/independent-projects/arc
[20] Quarkus - Contexts and Dependency Injection https://quarkus.io/guides/cdi-reference.html
[21] TestContainers https://www.testcontainers.org
[22] Microprofile Config https://microprofile.io/project/eclipse/microprofile-config
[23] MicroProfile OpenAPI https://github.com/eclipse/microprofile-open-api
[24] Swagger UI https://swagger.io/tools/swagger-ui

Quarkus

In the previous chapter, you had a quick peek to Quarkus and how you can build HTTP / REST-based applications with it. But that was just the beginning, Quarkus can do a lot more, and this is the purpose of this chapter. In this chapter, you are going to see:

- What's Quarkus? and how does it change the Java landscape
- What are the main Quarkus idea and how it helps in the *cloud native world*
- The Quarkus build process, in other words, the *secret sauce*
- Some Quarkus features such as the application lifecycle support
- How you can use Quarkus to generate native executable

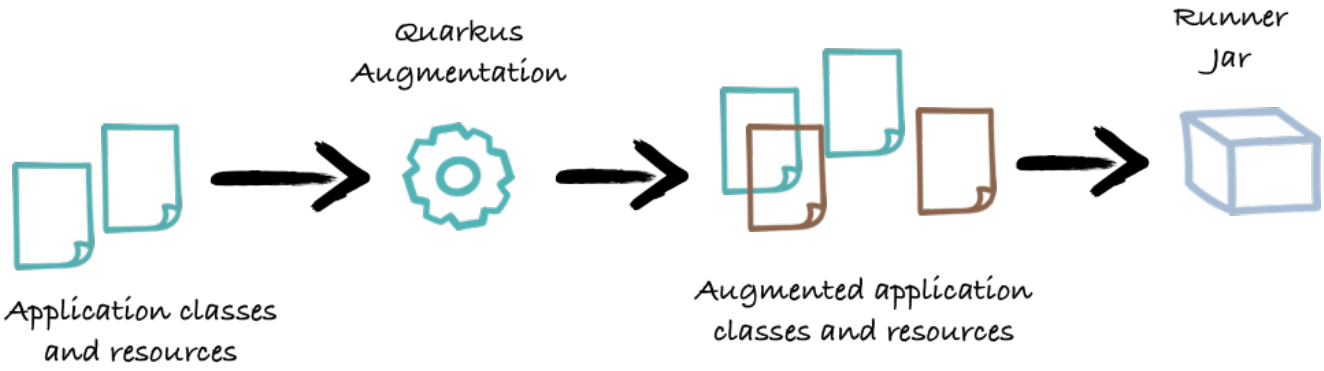
What's Quarkus?

Java was born more than 20 years ago. The world 20 years ago was quite different. The software industry has gone through several revolutions over these two decades. Java has always been able to reinvent itself to stay relevant.

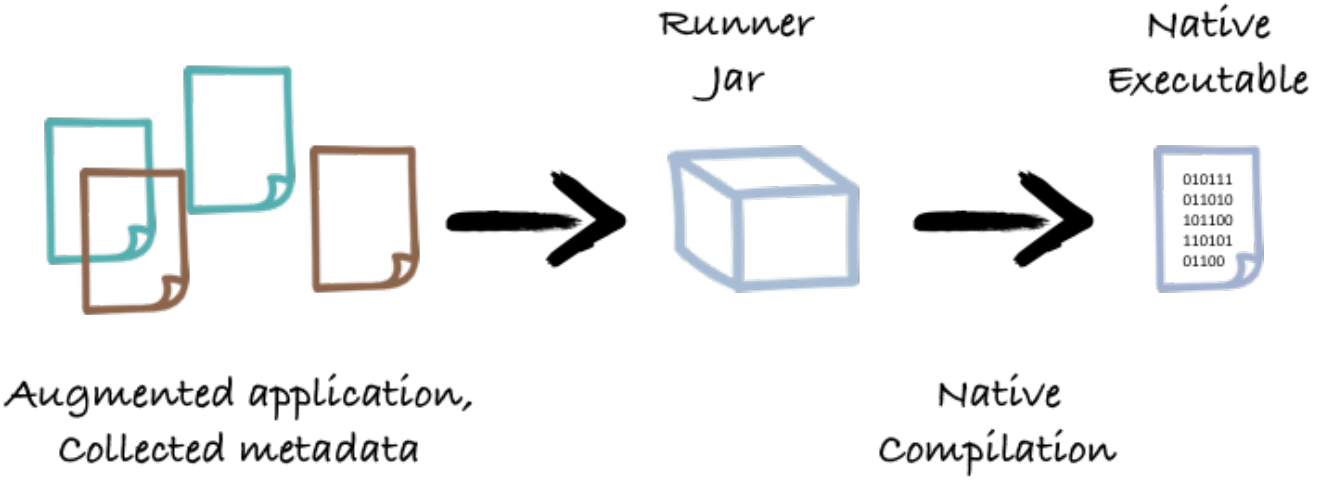
But a new revolution is happening right now. While for years, most applications were running on huge machines, with lots of CPU and memory, they are now running on the Cloud, in constrained environments, in containers, where the resources are shared. Density is the new optimization: crank as many mini-apps (or microservices) as possible per node. And scale by adding more instances of an app instead of a more powerful single instance.

The Java ergonomics, designed 20 years ago, do not fit well in this new environment. Java applications were designed to run 24/7 for months, even years. The JIT is optimizing the execution over time; the GC manages the memory efficiently… But all these features have a cost, and the memory required to run Java applications and startup times are showstoppers when instead of one application, you deploy 20 or 50 microservices. The issue is not the JVM itself; it's also the Java ecosystem that needs to be reinvented.

That's where Quarkus, and other projects, enter the game. Quarkus proposes to generalize "Ahead of Time" techniques.[25] When a Quarkus application is built, some work that usually happens at runtime is moved to the build time. Thus, when the application runs, everything has been pre-computed, and all the annotation scanning, XML parsing, and so on won't be executed anymore. It has two direct benefits: on the startup time (a lot faster) and on memory consumption (a lot lower).



So, as depicted on the figure above, Quarkus does bring an infrastructure for frameworks to embrace build time metadata discovery (like annotations), declare which classes need reflection at runtime, boot at build time, and generally offer a lot GraalVM optimization for free (or cheap at least). Indeed, thanks to all these metadata, Quarkus can configure native compilers such as the SubstrateVM compiler to generate a native executable for your Java application. Thanks to an aggressive dead-code elimination, the final executable is smaller, faster to start and use a ridiculously small amount of memory.



Quarkus does not stop there. As you have seen in the previous chapter, it proposes a stellar developer experience. It also unifies reactive and imperative so that you can mix regular JAX-RS and event-oriented code in the same application. Finally, Quarkus is based on many popular framework out there such as Eclipse Vert.x, Apache Camel, Undertow… You can already state that you have 5 years of experience with Quarkus.

Ok, but enough talking, time to see this in action.

Quarkus Augmentation

Let’s demystify all this.

So far, you have developed the superheroes microservice. This microservice is relatively simple, but it still has database access, ORM support, transaction, JSON serialization, and deserialization.

[hand o right] Call to action

Let’s now package this application using:

```
$ mvn package
```

In the log, you can see actions happening at build time during what Quarkus call the *augmentation* phase.

```
[INFO] --- quarkus-maven-plugin:1.9.2.Final:build (default) @ rest-hero ---
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Beginning quarkus augmentation
[INFO] [org.jboss.threads] JBoss Threads version 3.0.0.Final
[INFO] [org.hibernate.jpa.boot.internal.PersistenceXmlParser] HHH000318: Could not
find any META-INF/persistence.xml file in the classpath
[INFO] [org.hibernate.Version] HHH000412: Hibernate Core {5.4.5.Final}
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
2653ms
```

In this log, you can see that the Hibernate XML parser has been executed at build time. This saves from having to:

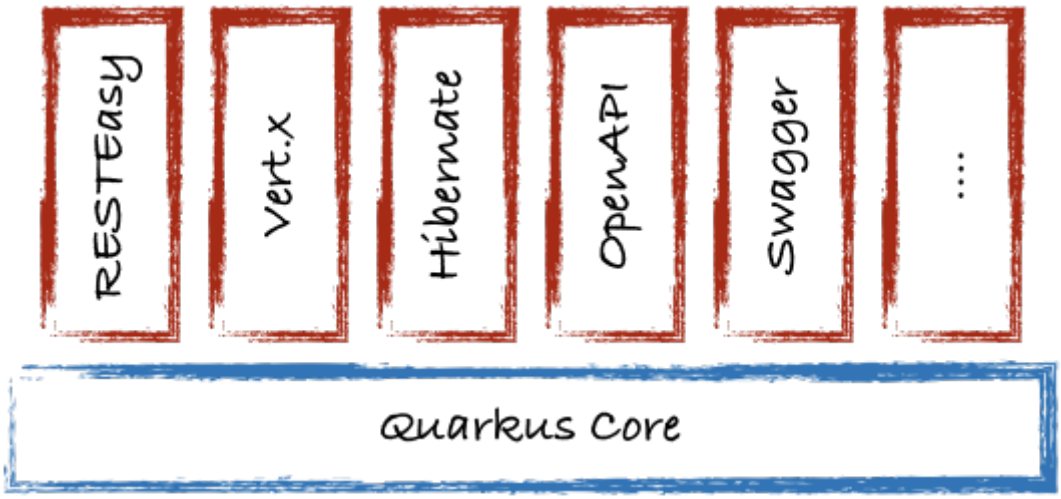
1. embed an XML parser at runtime,
2. Do the actual parsing,
3. Configure Hibernate based on the content of the file.

With Quarkus, at runtime, almost everything is already configured. Only runtime configuration properties are applied at startup (such as database URLs).

Also, during this augmentation, Java classes are generated or extended. Remember the `Hero` Panache entity. The class is extended during the *augmentation*. If you run `javap target/transformed-classes/io/quarkus/workshop/superheroes/hero/Hero.class`, you can see methods prefixed with `$$`, which have been added to the class.

If now you look at the `target/wiring-classes/io/quarkus/workshop/superheroes/hero`, you can see many generated classes.

All these metadata are computed and managed by *extensions*. The next figure present some of the extension you already used, but there are a lot more. We are going to learn more about extensions later in this workshop, and even build one. What's important to understand for now is that the magic is packaged into extension and every time you add a `quarkus-` dependency to your `pom.xml` file, you enable an extension.

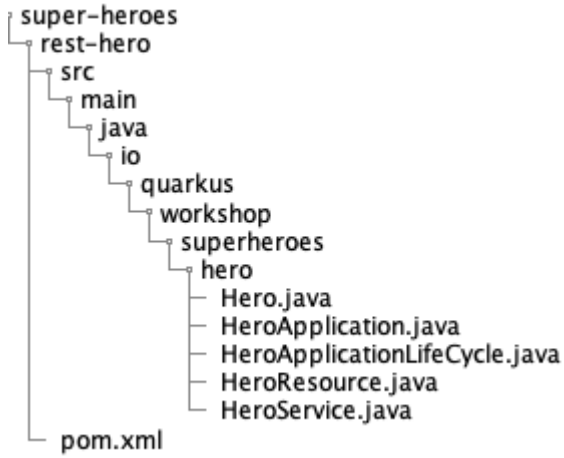

RESTEasy
Vert.x
Hibernate
OpenAPI
Swagger
....
Quarkus Core

Application Lifecycle

Now that you know how is structured Quarkus, let's continue using various extensions. You often need to execute custom actions when the application starts and clean up everything when the application stops. In this module we will display a banner in the logs once the Hero API has started.

Directory Structure

In this module we will add an extra class (`HeroApplicationLifeCycle`) to handle the Hero API lifecycle. You will end-up with the following directory structure:



Displaying a Banner

When our application starts, the logs are pretty boring... and lack of a banner (any decent application **must** have a banner nowadays). So the first thing that you need to do is to go to the following website and pick up your favourite "Hero API" text banner.

[hand o right] Call to action

Create a new class named `HeroApplicationLifeCycle` (or pick another name, the name does not matter) in the `io.quarkus.workshop.superheroes.hero` package, and copy your banner so you end up with a similar content:

```
package io.quarkus.workshop.superheroes.hero;

import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;
import io.quarkus.runtime.configuration.ProfileManager;
import org.jboss.logging.Logger;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

@ApplicationScoped
class HeroApplicationLifeCycle {

    private static final Logger LOGGER = Logger.getLogger(HeroApplicationLifeCycle
.class);

    void onStart(@Observes StartupEvent ev) {
        LOGGER.info("  _   _                          _     ____ ___ ");
        LOGGER.info(" | | | | ___ _ __ ___       / \\  |  _ \\_ _|");
        LOGGER.info(" | |_| |/ _ \\ '__/ _ \\     / _ \\ | |_) | | ");
        LOGGER.info(" |  _  |  __/ | | (_) |   / ___ \\|  __/| | ");
        LOGGER.info(" |_| |_|\\___|_|  \\___/   /_/   \\_\\_|  |___|");
        LOGGER.info("                          Powered by Quarkus");
    }

    void onStop(@Observes ShutdownEvent ev) {
        LOGGER.info("The application HERO is stopping...");
    }
}
```

Thanks to the CDI `@Observes`, the `HeroApplicationLifeCycle` is invoked:

- on startup with the `StartupEvent` so it can execute code (here, displaying the banner) when the application is starting
- on shutdown with the `ShutdownEvent` when the application is terminating

[hand o right] Call to action

Run the application with: `./mvnw quarkus:dev`, the banner is printed to the console. When the application is stopped, the second log message is printed.

If your application was still running, just send an HTTP request, like go to http://localhost:8083. As the application code changed, the application is restarted.

Configuration Profiles

Quarkus supports the notion of configuration profiles. These allow you to have multiple configuration in the same file and select between them via a profile name.

By default Quarkus has three profiles, although it is possible to use as many as you like. The default profiles are:

- `dev` - Activated when in development mode (i.e. `quarkus:dev`)
- `test` - Activated when running tests
- `prod` - The default profile when not running in development or test mode

Let's change the `HeroApplicationLifeCycle` so it displays the current profile.

[hand o right] Call to action

For that, just add a log invoking the `ProfileManager.getActiveProfile()` method:

```
void onStart(@Observes StartupEvent ev) {
    LOGGER.info("  _   _                          _    ____ ___ ");
    LOGGER.info(" | | | | ___ _ __ ___        / \\  |  _ \\_ _|");
    LOGGER.info(" | |_| |/ _ \\ '__/ _ \\     / _ \\ | |_) | | ");
    LOGGER.info(" |  _  |  __/ | | (_) |   / ___ \\|  __/| | ");
    LOGGER.info(" |_| |_|\\___|_|  \\___/   /_/   \\_\\_|  |___|");
    LOGGER.info("                          Powered by Quarkus");
    LOGGER.infof("The application HERO is starting with profile `%s`", ProfileManager
.getActiveProfile());
}
```

If not already done, you need to add the following import statement: `import io.quarkus.runtime.configuration.ProfileManager;`

In the `application.properties` file, you can prefix a property to be defined in the running profile. For example, we did add the `%test.level.multiplier=1` property in the previous chapter. This indicates that the property `level.multiplier` is set to 1 in the `test` profile.

Now, if you start your application in dev mode with `mvn compile quarkus:dev`, you will get the `dev` profile enabled. If you start the tests, the `test` profile is enabled (and so the `multiplier` is set to 1).

[hand o right] Call to action

Package your application with `mvn package`, and start it with `java -Dquarkus.profile=foo -jar target/rest-hero-1.0-SNAPSHOT-runner.jar`. You will see that the `foo` profile is enabled. As not overridden, the `level.multiplier` property has the value 3.

Profiles are very useful to customize the configuration per environment. We are going to see an example of such customization in the next section.

From Java to Native

Building a Native Executable

Let's now produce a native executable for our application. As explained in the introduction of this chapter, Quarkus is able to generate native executables. Just like Go, native executable don't need a VM to run, they contain the whole application, like an `.exe` file on Windows.

It improves the startup time of the application, and produces a minimal disk footprint. The executable would have everything to run the application including the "JVM" (shrunk to be just enough to run the application), and the application.

Choosing JVM execution vs native executable execution depends on your application needs and environment. Discuss with the lab organizers for some insights

To do so, you will find in the `pom.xml` the following profile:

```
<profile>
  <id>native</id>
  <activation>
    <property>
      <name>native</name>
    </property>
  </activation>
  <build>
    <plugins>
      <plugin>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-maven-plugin</artifactId>
        <version>${quarkus.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>native-image</goal>
            </goals>
            <configuration>
              <enableHttpUrlHandler>true</enableHttpUrlHandler>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>${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>
              </systemProperties>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <properties>
    <quarkus.package.type>native</quarkus.package.type>
  </properties>
</profile>
```

Make sure you have the `GRAALVM_HOME` environment variable defined and pointing to where you installed GraalVM.

[hand o right] Call to action

Then create a native executable using: `./mvnw package -Pnative`. In addition to the regular files (`rest-hero-1.0-SNAPSHOT.jar` and `rest-hero-1.0-SNAPSHOT-runner.jar`), the build also produces `target/rest-hero-1.0-SNAPSHOT-runner` (notice that there is no `.jar` file extension). You can run it using: `./target/rest-hero-1.0-SNAPSHOT-runner`.

Creating a native executable requires a lot of memory and CPU. It also takes a few minutes, even for simple application like the Hero microservice. Most of the time is spent during the dead code elimination, as it traverse the whole (closed) world.

Testing the Native Executable

Producing a native executable can lead to a few issues, and so it's also a good idea to run some tests against the application running in the native file. In the `pom.xml` file, the native profile contains:

```
<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>
        </systemProperties>
      </configuration>
    </execution>
  </executions>
</plugin>
```

This instructs the failsafe-maven-plugin to run integration-test and indicates the location of the produced native executable.

[hand o right] Call to action

Then, open the `src/test/java/io/quarkus/workshop/superheroes/hero/NativeHeroResourceIT.java` and update it with the following:

```
```

Instead of using `@QuarkusTest`, it uses the `@NativeImageTest` test runner that starts the application

from the native file before the tests. The executable is retrieved using the `native.image.path` system property configured in the Failsafe Maven Plugin.

Notice that `NativeHeroResourceIT` does not extend `HeroResourceTest`. It is a good practice to share the same tests for native and JVM mode that way but in our case the behavior is a bit different. For example, we do not want Swagger UI exposed in our native image.

Also notice that this class also use the `@QuarkusTestResource` to start and stop the database.

[hand o right] Call to action

To see the `NativeHeroResourceIT` run against the native executable, use `./mvnw verify -Pnative`:

```
[INFO] --- maven-failsafe-plugin:2.22.0:integration-test (default) @ rest-hero ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running io.quarkus.workshop.superheroes.hero.NativeHeroResourceIT
...
20:42:59 INFO  [io.qu.wo.su.he.HeroApplicationLifeCycle] (main)   _   _
_    ____ ___
20:42:59 INFO  [io.qu.wo.su.he.HeroApplicationLifeCycle] (main)  | | | | ___ _ __ ___
/ \  |  _ \_ _|
20:42:59 INFO  [io.qu.wo.su.he.HeroApplicationLifeCycle] (main)  | |_| |/ _ \ '__/ _ \
/ _ \ | |_) | |
20:42:59 INFO  [io.qu.wo.su.he.HeroApplicationLifeCycle] (main)  |  _  |  __/ | | (_)
|   / ___ \|  __/| |
20:42:59 INFO  [io.qu.wo.su.he.HeroApplicationLifeCycle] (main)  |_| |_|\___|_|  \___/
/_/   \_\_|  |___|
20:42:59 INFO  [io.qu.wo.su.he.HeroApplicationLifeCycle] (main)
Powered by Quarkus
...
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 9.068 s - in
io.quarkus.workshop.superheroes.hero.NativeHeroResourceIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.0:verify (default) @ rest-hero ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  05:29 min
[INFO] Finished at: 2019-10-18T20:43:03+02:00
[INFO] ------------------------------------------------------------------------
```

[25] Ahead-of-Time Compilation https://www.graalvm.org/docs/reference-manual/native-image

One Microservice is no Microservices

So far we've built one microservice. In the following sections you will develop two extra microservices: a *villain* microservice, a mad copycat of the *hero* microservice, and a *fight* microservice where heroes and villains fight. We will also add an Angular front-end so we can fight graphically.

```
Dot Executable: /opt/local/bin/dot
File does not exist
Cannot find Graphviz. You should try

@startuml
testdot
@enduml

 or

java -jar plantuml.jar -testdot
```

Each microservice is developed in it's own directory.

```
super-heroes
  infrastructure
  rest-fight
    src
      main
      test
  rest-hero
    src
      main
      test
  rest-villain
    src
      main
      test
  ui-super-heroes
    src
      app
      main
```

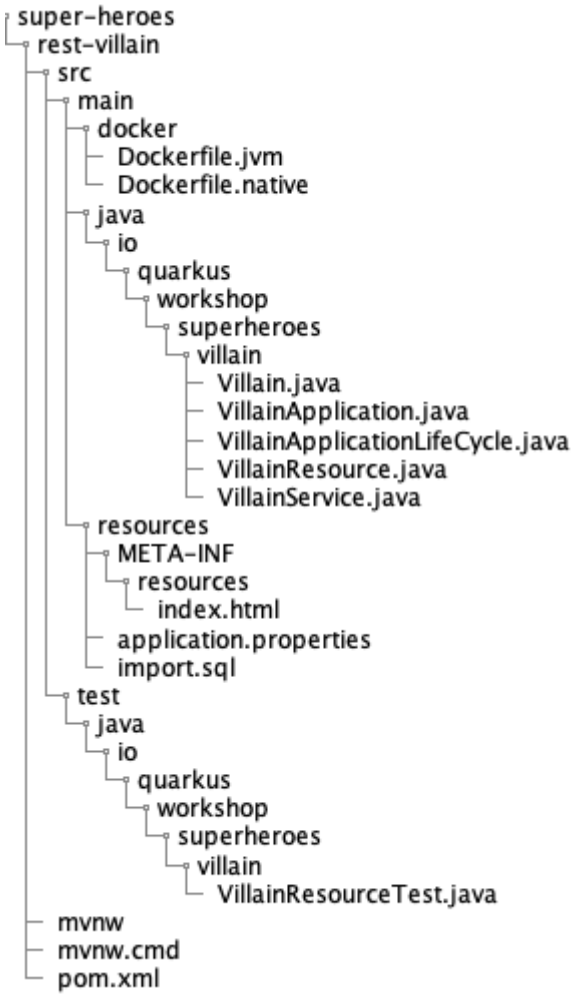
Villain Microservice

New microservice, new project! In this section we will see the counterpart of the Hero microservice: the Villain microservice! The Villain REST Endpoint is **really** similar to the Hero Endpoint.

The code has already been provided in the `/super-heroes/rest-villain/` directory. There is almost no differences with the hero microservice, just that it provides super villains instead and uses the port `8084`.

Directory Structure

As for the hero microservice, you have the following directory structure:



If you look at the code, it's very similar to the hero microservice.

Running, Testing and Packaging the Application

[hand o right] Call to action

First, make sure the tests pass by executing the command `./mvnw test` (or from your IDE).

Now that the tests are green, we are ready to run our application.

[hand o right] Call to action

Use `./mvnw quarkus:dev` to start it (notice the nice banner). Once the application is started, create a new villain with the following cUrl command:

```
$ curl -X POST -d  '{"level":2, "name":"Darth Vader", "powers":"Darkness, Longevity"}'
-H "Content-Type: application/json" http://localhost:8084/api/villains -v

< HTTP/1.1 201 Created
< Location: http://localhost:8084/api/villains/582
```

The cUrl command returns the location of the newly created villain. Take this URL and do an HTTP GET on it.

```
$ curl http://localhost:8084/api/villains/582 | jq

{
  "id": 582,
  "level": 4,
  "name": "Darth Vader",
  "powers": "Darkness, Longevity"
}
```

Remember that you can also check Swagger UI by going to http://localhost:8084/swagger-ui.

Fight Microservice

Ok, let's develop another microservice. We have a REST API that returns a random Hero. Another REST API that returns a random Villain... we need a new REST API that invokes those two, gets one random hero and one random villain and makes them fight. Let's call it the Fight API.

Bootstrapping the Fight REST Endpoint

Like for the Hero and Villain API, the easiest way to create this new Quarkus project is to use a Maven archetype. Under the `quarkus-workshop-super-heroes/super-heroes` root directory where you have all your code.

[hand o right] Call to action

Open a terminal and run the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.9.2.Final:create \
    -DprojectGroupId=io.quarkus.workshop.super-heroes \
    -DprojectArtifactId=rest-fight \
    -DclassName="io.quarkus.workshop.superheroes.fight.FightResource" \
    -Dpath="api/fights"
cd rest-fight
./mvnw quarkus:add-extension -Dextensions="jdbc-postgresql,hibernate-orm
-panache,hibernate-validator,quarkus-resteasy-jsonb,smallrye-openapi,kafka"
```

[hand o right] Call to action

Also add Testcontainers and other test-related dependencies to your `pom.xml`.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.12.2</version>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.12.2</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>kafka</artifactId>
    <version>1.12.2</version>
</dependency>
<dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <scope>test</scope>
</dependency>
```

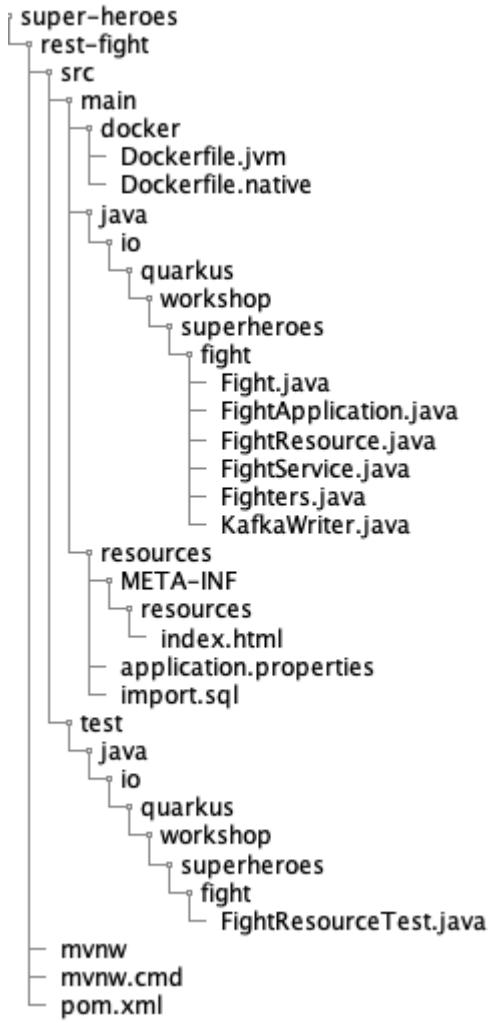
Prefering Web UI

Instead of the Maven command, you can use https://code.quarkus.io.

You can see that beyond the extensions we have used so far, we added the Kafka support which uses Eclipse MicroProfile Reactive Messaging. Stay tuned.

Directory Structure

At the end you should have the following directory structure:



Fight Entity

A fight is between a hero and a villain. Each time there is a fight, there is a winner and a loser. So the `Fight` entity is there to store all these fights.

```
package io.quarkus.workshop.superheroes.fight;

import io.quarkus.hibernate.orm.panache.PanacheEntity;
import org.eclipse.microprofile.openapi.annotations.media.Schema;

import javax.persistence.Entity;
import javax.validation.constraints.NotNull;
import java.time.Instant;

@Entity
@Schema(description="Each fight has a winner and a loser")
public class Fight extends PanacheEntity {

    @NotNull
    public Instant fightDate;
    @NotNull
    public String winnerName;
    @NotNull
    public int winnerLevel;
    @NotNull
    public String winnerPicture;
    @NotNull
    public String loserName;
    @NotNull
    public int loserLevel;
    @NotNull
    public String loserPicture;
    @NotNull
    public String winnerTeam;
    @NotNull
    public String loserTeam;

    // toString method
}
```

Fighters Bean

Now comes a trick. The Fight REST API will ultimatelly invoke the Hero and Villain APIs (next sections) to get two random fighters. The `Fighters` class has one `Hero` and one `Villain`. Notice that `Fighters` is not an entity, it is not persisted in the database, just marshalled and unmarshalled to JSon.

```
package io.quarkus.workshop.superheroes.fight;

import io.quarkus.workshop.superheroes.fight.client.Hero;
import io.quarkus.workshop.superheroes.fight.client.Villain;
import org.eclipse.microprofile.openapi.annotations.media.Schema;

import javax.validation.constraints.NotNull;

@Schema(description="A fight between one hero and one villain")
public class Fighters {

    @NotNull
    public Hero hero;
    @NotNull
    public Villain villain;

}
```

The Fight REST API is just interested in the hero's name, level, picture and powers (not the other name as described in the Hero API). So the `Hero` bean looks like this (notice the `client` subpackage):

```
package io.quarkus.workshop.superheroes.fight.client;

import org.eclipse.microprofile.openapi.annotations.media.Schema;

import javax.validation.constraints.NotNull;

@Schema(description="The hero fighting against the villain")
public class Hero {

    @NotNull
    public String name;
    @NotNull
    public int level;
    @NotNull
    public String picture;
    public String powers;

}
```

`Villain` is pretty similar (also in the `client` subpackage):

```
package io.quarkus.workshop.superheroes.fight.client;

import org.eclipse.microprofile.openapi.annotations.media.Schema;

import javax.validation.constraints.NotNull;

@Schema(description="The villain fighting against the hero")
public class Villain {

    @NotNull
    public String name;
    @NotNull
    public int level;
    @NotNull
    public String picture;
    public String powers;

}
```

So, these classes are just used to map the results from the Hero and Villain microservices.

FightService Transactional Service

To *transactionnally* manipulate the `Fight` entity we need a `FightService`. Notice the `persistFight` method. This method is the one creating a fight between a hero and a villain. As you can see the algorithm to determine the winner is a bit random (even though it uses the levels). If you are not happy about the way the fight operates, choose your own winning algorithm ;o)

```
package io.quarkus.workshop.superheroes.fight;

import io.quarkus.workshop.superheroes.fight.client.Hero;
import io.quarkus.workshop.superheroes.fight.client.Villain;
import org.jboss.logging.Logger;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.transaction.Transactional;
import java.time.Instant;
import java.util.List;
import java.util.Random;

import static javax.transaction.Transactional.TxType.REQUIRED;
import static javax.transaction.Transactional.TxType.SUPPORTS;

@ApplicationScoped
@Transactional(SUPPORTS)
public class FightService {

    private static final Logger LOGGER = Logger.getLogger(FightService.class);
```

```
    private final Random random = new Random();

    public List<Fight> findAllFights() {
        return Fight.listAll();
    }

    public Fight findFightById(Long id) {
        return Fight.findById(id);
    }

    @Transactional(REQUIRED)
    public Fight persistFight(Fighters fighters) {
        // Amazingly fancy logic to determine the winner...
        Fight fight;

        int heroAdjust = random.nextInt(20);
        int villainAdjust = random.nextInt(20);

        if ((fighters.hero.level + heroAdjust)
            > (fighters.villain.level + villainAdjust)) {
            fight = heroWon(fighters);
        } else if (fighters.hero.level < fighters.villain.level) {
            fight = villainWon(fighters);
        } else {
            fight = random.nextBoolean() ? heroWon(fighters) : villainWon(fighters);
        }

        fight.fightDate = Instant.now();
        fight.persist(fight);
        return fight;
    }

    private Fight heroWon(Fighters fighters) {
        LOGGER.info("Yes, Hero won :o)");
        Fight fight = new Fight();
        fight.winnerName = fighters.hero.name;
        fight.winnerPicture = fighters.hero.picture;
        fight.winnerLevel = fighters.hero.level;
        fight.loserName = fighters.villain.name;
        fight.loserPicture = fighters.villain.picture;
        fight.loserLevel = fighters.villain.level;
        fight.winnerTeam = "heroes";
        fight.loserTeam = "villains";
        return fight;
    }

    private Fight villainWon(Fighters fighters) {
        LOGGER.info("Gee, Villain won :o(");
        Fight fight = new Fight();
        fight.winnerName = fighters.villain.name;
```

```
        fight.winnerPicture = fighters.villain.picture;
        fight.winnerLevel = fighters.villain.level;
        fight.loserName = fighters.hero.name;
        fight.loserPicture = fighters.hero.picture;
        fight.loserLevel = fighters.hero.level;
        fight.winnerTeam = "villains";
        fight.loserTeam = "heroes";
        return fight;
    }


}
```

[hand o right] Call to action

For now, just implement an empty `Fighters findRandomFighters()` method which returns null. Later, this method will invoke the Hello and Villain API to get a random Hello and random Villain. So for now something like the following is enough:

```
public Fighters findRandomFighters() {
    // Will be implemented later
    return null;
}
```

FightResource Endpoint

To expose a REST API we also need a `FightResource` (with OpenAPI annotations of course).

```
package io.quarkus.workshop.superheroes.fight;

import org.eclipse.microprofile.openapi.annotations.Operation;
import org.eclipse.microprofile.openapi.annotations.enums.SchemaType;
import org.eclipse.microprofile.openapi.annotations.media.Content;
import org.eclipse.microprofile.openapi.annotations.media.Schema;
import org.eclipse.microprofile.openapi.annotations.parameters.Parameter;
import org.eclipse.microprofile.openapi.annotations.parameters.RequestBody;
import org.eclipse.microprofile.openapi.annotations.responses.APIResponse;
import org.jboss.logging.Logger;

import javax.inject.Inject;
import javax.validation.Valid;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
```

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import java.util.List;

import static javax.ws.rs.core.MediaType.APPLICATION_JSON;
import static javax.ws.rs.core.MediaType.TEXT_PLAIN;

@Path("/api/fights")
@Produces(APPLICATION_JSON)
public class FightResource {

    private static final Logger LOGGER = Logger.getLogger(FightResource.class);

    @Inject
    FightService service;

    @Operation(summary = "Returns two random fighters")
    @APIResponse(responseCode = "200", content = @Content(mediaType =
APPLICATION_JSON, schema = @Schema(implementation = Fighters.class, required = true)))
    @GET
    @Path("/randomfighters")
    public Response getRandomFighters() throws InterruptedException {
        Fighters fighters = service.findRandomFighters();
        LOGGER.debug("Get random fighters " + fighters);
        return Response.ok(fighters).build();
    }

    @Operation(summary = "Returns all the fights from the database")
    @APIResponse(responseCode = "200", content = @Content(mediaType =
APPLICATION_JSON, schema = @Schema(implementation = Fight.class, type = SchemaType
.ARRAY)))
    @APIResponse(responseCode = "204", description = "No fights")
    @GET
    public Response getAllFights() {
        List<Fight> fights = service.findAllFights();
        LOGGER.debug("Total number of fights " + fights);
        return Response.ok(fights).build();
    }

    @Operation(summary = "Returns a fight for a given identifier")
    @APIResponse(responseCode = "200", content = @Content(mediaType =
APPLICATION_JSON, schema = @Schema(implementation = Fight.class)))
    @APIResponse(responseCode = "204", description = "The fight is not found for a
given identifier")
    @GET
    @Path("/{id}")
    public Response getFight(@Parameter(description = "Fight identifier", required =
true) @PathParam("id") Long id) {
        Fight fight = service.findFightById(id);
        if (fight != null) {
            LOGGER.debug("Found fight " + fight);
```

```
            return Response.ok(fight).build();
        } else {
            LOGGER.debug("No fight found with id " + id);
            return Response.noContent().build();
        }
    }

    @Operation(summary = "Trigger a fight between two fighters")
    @APIResponse(responseCode = "200", description = "The result of the fight",
content = @Content(mediaType = APPLICATION_JSON, schema = @Schema(implementation =
Fight.class)))
    @POST
    public Fight fight(@RequestBody(description = "The two fighters fighting",
required = true, content = @Content(mediaType = APPLICATION_JSON, schema = @Schema
(implementation = Fighters.class))) @Valid Fighters fighters, @Context UriInfo
uriInfo) {
        return service.persistFight(fighters);
    }

    @GET
    @Produces(TEXT_PLAIN)
    @Path("/hello")
    public String hello() {
        return "hello";
    }
}
```

FightApplication for OpenAPI

The `FightApplication` class is just there to customize the OpenAPI contract.

```
package io.quarkus.workshop.superheroes.fight;

import org.eclipse.microprofile.openapi.annotations.ExternalDocumentation;
import org.eclipse.microprofile.openapi.annotations.OpenAPIDefinition;
import org.eclipse.microprofile.openapi.annotations.info.Contact;
import org.eclipse.microprofile.openapi.annotations.info.Info;
import org.eclipse.microprofile.openapi.annotations.servers.Server;
import org.eclipse.microprofile.openapi.annotations.tags.Tag;

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

@ApplicationPath("/")
@OpenAPIDefinition(
    info = @Info(title = "Fight API",
        description = "This API allows a hero and a villain to fight",
        version = "1.0",
        contact = @Contact(name = "Quarkus", url = "https://github.com/quarkusio")),
    servers = {
        @Server(url = "http://localhost:8082")
    },
    externalDocs = @ExternalDocumentation(url = "https://github.com/quarkusio/quarkus-
workshops", description = "All the Quarkus workshops"),
    tags = {
        @Tag(name = "api", description = "Public that can be used by anybody"),
        @Tag(name = "fight", description = "Anybody interested in fights"),
        @Tag(name = "superheroes", description = "Well, superhero fights")
    }
)
public class FightApplication extends Application {
}
```

Notice that there is no `FightApplicationLifeCycle` class. We will use a Quarkus extension later on to display a banner for Fight.

Adding Data

[hand o right] Call to action

To load some SQL statements when Hibernate ORM starts, download the SQL file import.sql and copy it under `src/main/resources`.

```
INSERT INTO fight(id, fightDate, winnerName, winnerLevel, winnerPicture, loserName,
loserLevel, loserPicture, winnerTeam, loserTeam)
VALUES (nextval('hibernate_sequence'), current_timestamp, 'Chewbacca', 5,
'https://www.superherodb.com/pictures2/portraits/10/050/10466.jpg', 'Buuccolo', 3,
'https://www.superherodb.com/pictures2/portraits/11/050/15355.jpg', 'heroes',
'villains');
INSERT INTO fight(id, fightDate, winnerName, winnerLevel, winnerPicture, loserName,
loserLevel, loserPicture, winnerTeam ,loserTeam)
VALUES (nextval('hibernate_sequence'), current_timestamp, 'Galadriel', 10,
'https://www.superherodb.com/pictures2/portraits/11/050/11796.jpg', 'Darth Vader', 8,
'https://www.superherodb.com/pictures2/portraits/10/050/10444.jpg', 'heroes',
'villains');
INSERT INTO fight(id, fightDate, winnerName, winnerLevel, winnerPicture, loserName,
loserLevel, loserPicture, winnerTeam ,loserTeam)
VALUES (nextval('hibernate_sequence'), current_timestamp, 'Annihilus', 23,
'https://www.superherodb.com/pictures2/portraits/10/050/1307.jpg', 'Shikamaru', 1,
'https://www.superherodb.com/pictures2/portraits/10/050/11742.jpg', 'villains',
'heroes');
...
```

Configuration

As usual, we need to configure the application.

[hand o right] Call to action

In the `application.properties` file add:

```
quarkus.http.port=8082

## Database configuration
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/fights_database
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=superfight
quarkus.datasource.password=superfight
quarkus.datasource.max-size=8
quarkus.datasource.min-size=2
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.sql=true

## Logging configuration
quarkus.log.console.enable=true
quarkus.log.console.format=%d{HH:mm:ss} %-5p [%c{2.}] (%t) %s%e%n
quarkus.log.console.level=DEBUG
quarkus.log.console.color=true

## Production configuration
%prod.quarkus.hibernate-orm.log.sql=false
%prod.quarkus.log.console.level=INFO
%prod.quarkus.hibernate-orm.database.generation=update

process.milliseconds=0
```

Note that the fight service uses the port 8082.

FightResourceTest Test Class

We need to test our REST API.

[hand o right] Call to action

For that, copy the following `FightResourceTest` class under the `src/test/java/io/quarkus/workshop/superheroes/fight` directory.

```
package io.quarkus.workshop.superheroes.fight;

import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;
import io.quarkus.workshop.superheroes.fight.client.Hero;
import io.quarkus.workshop.superheroes.fight.client.Villain;
import io.restassured.common.mapper.TypeRef;
import org.hamcrest.core.Is;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;
```

```
import java.util.List;

import java.util.Random;
import static io.restassured.RestAssured.get;
import static io.restassured.RestAssured.given;
import static javax.ws.rs.core.HttpHeaders.ACCEPT;
import static javax.ws.rs.core.HttpHeaders.CONTENT_TYPE;
import static javax.ws.rs.core.MediaType.APPLICATION_JSON;
import static javax.ws.rs.core.Response.Status.*;
import static org.hamcrest.CoreMatchers.*;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;




@QuarkusTest
@QuarkusTestResource(DatabaseResource.class)
@QuarkusTestResource(KafkaResource.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class FightResourceTest {

    private static final String DEFAULT_WINNER_NAME = "Super Baguette";
    private static final String DEFAULT_WINNER_PICTURE = "super_baguette.png";
    private static final int DEFAULT_WINNER_LEVEL = 42;
    private static final String DEFAULT_LOSER_NAME = "Super Chocolatine";
    private static final String DEFAULT_LOSER_PICTURE = "super_chocolatine.png";
    private static final int DEFAULT_LOSER_LEVEL = 6;

    private static final int NB_FIGHTS = 10;
    private static String fightId;

    @Test
    void shouldPingOpenAPI() {
        given()
            .header(ACCEPT, APPLICATION_JSON)
            .when().get("/openapi")
            .then()
            .statusCode(OK.getStatusCode());
    }

    @Test
    void shouldPingSwaggerUI() {
        given()
            .when().get("/swagger-ui")
            .then()
            .statusCode(OK.getStatusCode());
    }
```

```
    @Test
    public void testHelloEndpoint() {
        given()
            .when().get("/api/fights/hello")
            .then()
            .statusCode(200)
            .body(is("hello"));
    }

    @Test
    void shouldNotGetUnknownFight() {
        Long randomId = new Random().nextLong();
        given()
            .pathParam("id", randomId)
            .when().get("/api/fights/{id}")
            .then()
            .statusCode(NO_CONTENT.getStatusCode());
    }


    @Test
    void shouldNotAddInvalidItem() {
        Fighters fighters = new Fighters();
        fighters.hero = null;
        fighters.villain = null;

        given()
            .body(fighters)
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .header(ACCEPT, APPLICATION_JSON)
            .when()
            .post("/api/fights")
            .then()
            .statusCode(BAD_REQUEST.getStatusCode());
    }

    @Test
    @Order(1)
    void shouldGetInitialItems() {
        List<Fight> fights = get("/api/fights").then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .extract().body().as(getFightTypeRef());
        assertEquals(NB_FIGHTS, fights.size());
    }

    @Test
    @Order(2)
    void shouldAddAnItem() {
        Hero hero = new Hero();
        hero.name = DEFAULT_WINNER_NAME;
```

```
        hero.picture = DEFAULT_WINNER_PICTURE;
        hero.level = DEFAULT_WINNER_LEVEL;
        Villain villain = new Villain();
        villain.name = DEFAULT_LOSER_NAME;
        villain.picture = DEFAULT_LOSER_PICTURE;
        villain.level = DEFAULT_LOSER_LEVEL;
        Fighters fighters = new Fighters();
        fighters.hero = hero;
        fighters.villain = villain;

        fightId = given()
            .body(fighters)
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .header(ACCEPT, APPLICATION_JSON)
            .when()
            .post("/api/fights")
            .then()
            .statusCode(OK.getStatusCode())
            .body(containsString("winner"), containsString("loser"))
            .extract().body().jsonPath().getString("id");

        assertNotNull(fightId);

        given()
            .pathParam("id", fightId)
            .when().get("/api/fights/{id}")
            .then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .body("winnerName", Is.is(DEFAULT_WINNER_NAME))
            .body("winnerPicture", Is.is(DEFAULT_WINNER_PICTURE))
            .body("winnerLevel", Is.is(DEFAULT_WINNER_LEVEL))
            .body("loserName", Is.is(DEFAULT_LOSER_NAME))
            .body("loserPicture", Is.is(DEFAULT_LOSER_PICTURE))
            .body("loserLevel", Is.is(DEFAULT_LOSER_LEVEL))
            .body("fightDate", Is.is(notNullValue()));

        List<Fight> fights = get("/api/fights").then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .extract().body().as(getFightTypeRef());
        assertEquals(NB_FIGHTS + 1, fights.size());
    }

    private TypeRef<List<Fight>> getFightTypeRef() {
        return new TypeRef<List<Fight>>() {
            // Kept empty on purpose
        };
    }
}
```

You would also need, not only one but 2 `QuarkusTestResources`, one for the database and one for Kafka. We could have merged both in a single class, but we wanted to illustrate the possibility to have more than one resources.

[hand o right] Call to action

So, create the `DatabaseResource` class in the same package as the `FightResourceTest` class, and add the following content:

```
package io.quarkus.workshop.superheroes.fight;

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;
import org.testcontainers.containers.PostgreSQLContainer;

import java.util.Collections;
import java.util.Map;

public class DatabaseResource implements QuarkusTestResourceLifecycleManager {

    public static final PostgreSQLContainer DATABASE = new PostgreSQLContainer<>(
"postgres:10.5")
        .withDatabaseName("fights_database")
        .withUsername("superfight")
        .withPassword("superfight")
        .withExposedPorts(5432);

    @Override
    public Map<String, String> start() {
        DATABASE.start();
        return Collections.singletonMap("quarkus.datasource.jdbc.url", DATABASE
.getJdbcUrl());
    }

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

[hand o right] Call to action

Then, create the `KafkaResource` class with the following content:

```
package io.quarkus.workshop.superheroes.fight;

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;
import org.testcontainers.containers.KafkaContainer;

import java.util.Collections;
import java.util.Map;

public class KafkaResource implements QuarkusTestResourceLifecycleManager  {

    private static final KafkaContainer KAFKA = new KafkaContainer();


    @Override
    public Map<String, String> start() {
        KAFKA.start();
        return Collections.singletonMap("kafka.bootstrap.servers", KAFKA
.getBootstrapServers());
    }

    @Override
    public void stop() {
        KAFKA.stop();
    }
}
```

[hand o right] Call to action

Also, delete the generated `NativeFightResourceIT` class, as we won't run native test for this microservice.

Running, Testing and Packaging the Application

[hand o right] Call to action

First, make sure the tests pass by executing the command `./mvnw test` (or from your IDE).

Now that the tests are green, we are ready to run our application. Use `./mvnw quarkus:dev` to start it (notice that there is no banner yet, it will come later). Once the application is started, just check that it returns the fights from the database with the following cUrl command:

```
$ curl http://localhost:8082/api/fights
```

Remember that you can also check Swagger UI by going to http://localhost:8082/swagger-ui.

User Interface

Now that we have the three main microservices, time to have a decent user interface to start fighting. The purpose of this workshop is not to develop a web interface and learn *yet another web framework*. This time you will just download an Angular application, install it, and run it on another Quarkus instance.

The Web Application

Navigate to the `super-heroes/ui-super-heroes/ui-super-heroes` directory. It contains the code of the microservice. Being an Angular application, you will find a `package.json` file which defines all the needed dependencies. Notice that there is a `pom.xml` file. This is just a convenient way to install NodeJS and NPM so we can build the Angular application with Maven. The `pom.xml` also allows us to package the Angular application into Quarkus.

If you are not in a *frontend mood*, just scroll to Installing the Web Application on Quarkus

Looking at Some Code (optional)

You don't need to be an Angular expert, but there are some pieces of code that are worth looking at. If you look under the `src/app/shared` directory, you will find an `api` and a `model` sub-directory. Let's look at `fight.ts`.

```
export interface Fight {
    id?: number;
    fightDate: FightFightDate;
    winnerName: string;
    winnerLevel: number;
    winnerPicture: string;
    loserName: string;
    loserLevel: number;
    loserPicture: string;
}
```

As you can see, it matches our `Fight` Java class. Same for `fighters.ts`, `hero.ts` or `villain.ts`. Under `api` there is the `fight.service.ts` that defines all the methods to access to our Fight REST API through HTTP.

```
public apiFightsGet(observe?: 'body', reportProgress?: boolean):
Observable<Array<Fight>>;
public apiFightsGet(observe?: 'response', reportProgress?: boolean):
Observable<HttpResponse<Array<Fight>>>;
public apiFightsGet(observe?: 'events', reportProgress?: boolean):
Observable<HttpEvent<Array<Fight>>>;

public apiFightsRandomfightersGet(observe?: 'body', reportProgress?: boolean):
Observable<Fighters>;
public apiFightsRandomfightersGet(observe?: 'response', reportProgress?: boolean):
Observable<HttpResponse<Fighters>>;
public apiFightsRandomfightersGet(observe?: 'events', reportProgress?: boolean):
Observable<HttpEvent<Fighters>>;
```

Well, guess what? We didn't have to type this code either. It was generated thanks to a tool called `swagger-codegen`.[26] Because our Fight REST API exposes an OpenAPI contract, `swagger-codegen` just swallows it, and generates the TypeScript code to access it. It's just a matter of running:

```
$ swagger-codegen generate -i http://localhost:8082/openapi -l typescript-angular -o
src/app/shared
```

Here, you see another advantage of exposing an OpenAPI contract: it documents the API which can be read by a human, or processed by tools.

Installing the Web Application on Quarkus

Thanks to the `frontend-maven-plugin` plugin declared on the `pom.xml`, we can use a good old Maven command to install and build this Angular application.

[hand o right] Call to action

Execute `mvn install` and Maven will download and install Node JS and NPM and build the application. You should now have a `node_modules` directory with all the Angular dependencies. At this stage, make sure the following commands work:

```
ng version (or ./node_modules/.bin/ng version)
node -v    (or ./node/node -v)
```

To install the Angular application into a Quarkus instance, we just build the app and copy the bundles under the `resources/META-INF/resources` directory. Look at the `package.sh`, that's exactly what it does.

```
export DEST=src/main/resources/META-INF/resources
./node_modules/.bin/ng build --prod --base-href "."
rm -Rf ${DEST}
cp -R dist/* ${DEST}
```

[hand o right] Call to action

Execute the `package.sh` script. You will see all the Javascript files under `resources/META-INF/resources` directory. We are now ready to go.

If the `ng` command does not work because it can't find `node`, there is a little hack to solve it. Open the file `ui-super-heroes/node_modules/.bin/ng` and change the shebang line from `!/usr/bin/env node` to `!/usr/bin/env ./node/node`. This way `ng` knows it has to use NodeJS installed under the `ui-super-heroes/node` directory

Running the Web Application

[hand o right] Call to action

As usual, use `mvn quarkus:dev` to start the web application.

Be sure you have the hero and villain microservices running (dev mode is enough).

Once the application is started, go to http://localhost:8080 (8080 is the default Quarkus port as we didn't change it in the `application.properties` this time). It should display the main web page.

Welcome to Super Heroes Fight!

Astérix

9

Dexterity, Intelligence, Jump, Peak Human Condition, Reflexes, Stamina, Super Speed, Super Strength

NEW FIGHTERS

FIGHT !

Match

14

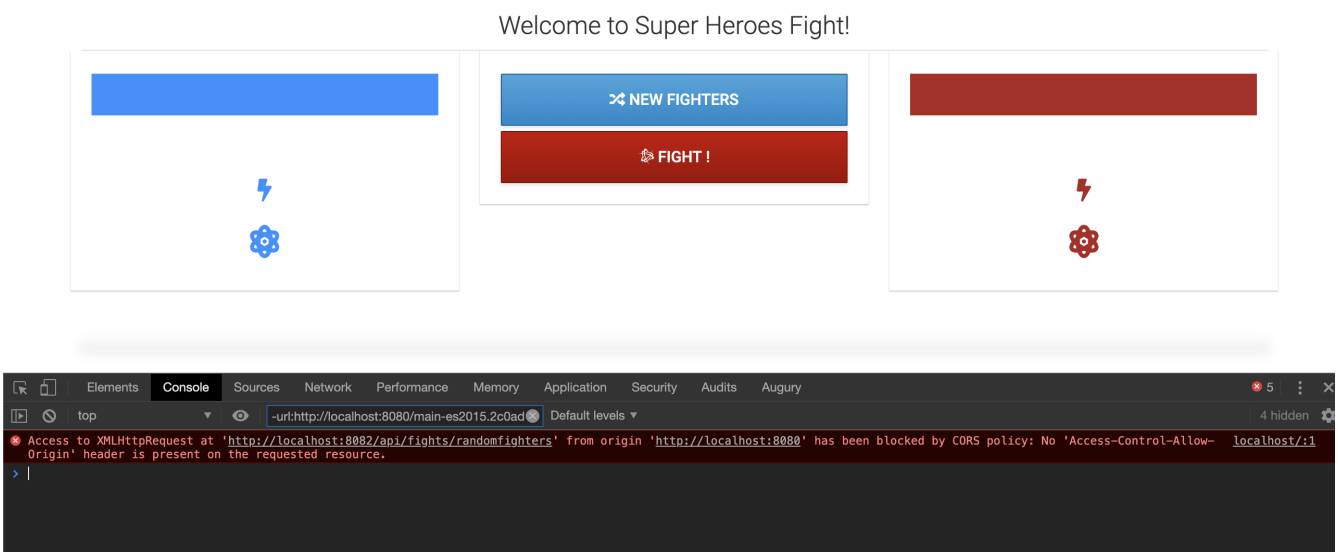
Accelerated Healing, Durability, Energy Absorption, Energy Blasts, Enhanced Hearing, Flight, Invulnerability, Jump, Super Breath, Super Speed, Super Strength, Telekinesis, Vision - Heat, Vision - Telescopic, Vision - X-Ray

Id	Fight Date	Winner	Loser
10	Oct 14, 2019, 11:04:22 AM	Black Canary	Superman
9	Oct 14, 2019, 11:04:22 AM	Tanker Bug	Shuri
8	Oct 14, 2019, 11:04:22 AM	Moondragon	Darth Plagueis
7	Oct 14, 2019, 11:04:22 AM	The Eraser	Gandalf The White
6	Oct 14, 2019, 11:04:22 AM	Anakin Skywalker	Janemba

Oups, not working yet! Not even the pictures, we must have been forgotten something! Let’s move on to the next section then and make the application work.

CORS

Cross-origin resource sharing (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. [27] So when we want our heroes and villains to fight, we actually cross several origins: we go from localhost:8080 (the UI) to localhost:8082 (Fight API) which invokes localhost:8083 (Hero) and localhost:8084 (Villain). If you look at the console of your Browser you should see something similar to this:



Quarkus comes with a CORS filter which intercepts all incoming HTTP requests. It can be enabled in the Quarkus configuration file:

If the filter is enabled and an HTTP request is identified as cross-origin, the CORS policy and headers defined using the following properties will be applied before passing the request on to its actual target (servlet, JAX-RS resource, etc.):

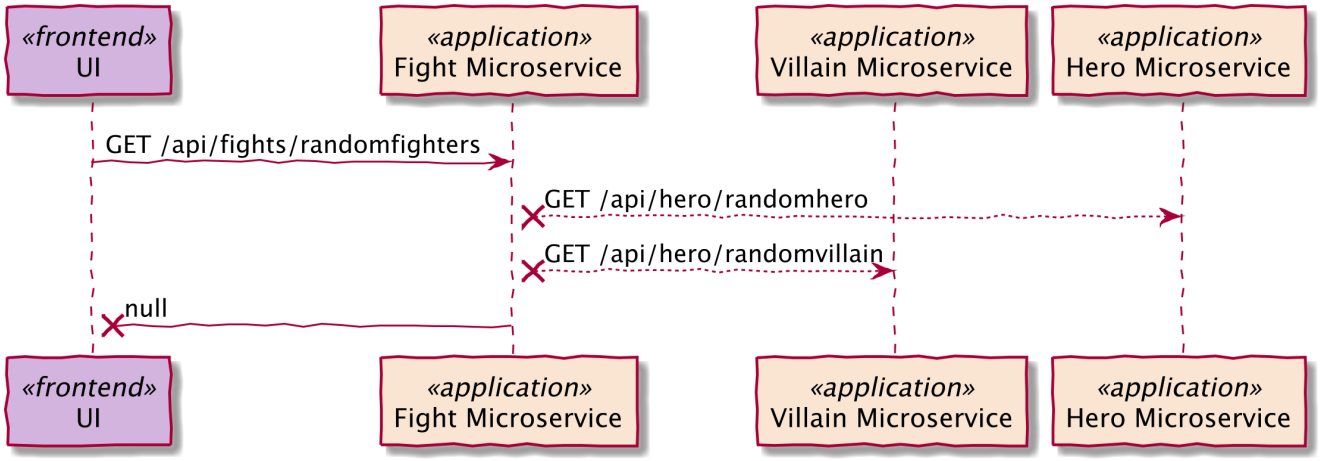
Property	Description
`quarkus.http.cors.origins`	The comma-separated list of origins allowed for CORS. The filter allows any origin if this is not set.
`quarkus.http.cors.methods`	The comma-separated list of HTTP methods allowed for CORS. The filter allows any method if this is not set.
`quarkus.http.cors.headers`	The comma-separated list of HTTP headers allowed for CORS. The filter allows any header if this is not set.
`quarkus.http.cors.exposed-headers`	The comma-separated list of HTTP headers exposed in CORS.
`quarkus.http.cors.access-control-max-age`	The duration indicating how long the results of a pre-flight request can be cached. This value will be returned in a Access-Control-Max-Age response header.

[hand o right] Call to action

So make sure you set the `quarkus.http.cors` property to `true` on the:

1. Fight microservice,
2. Hero microservice,
3. Villain microservice

But, even with this, the UI is still not working. The explanation is simple, we forgot another thing:



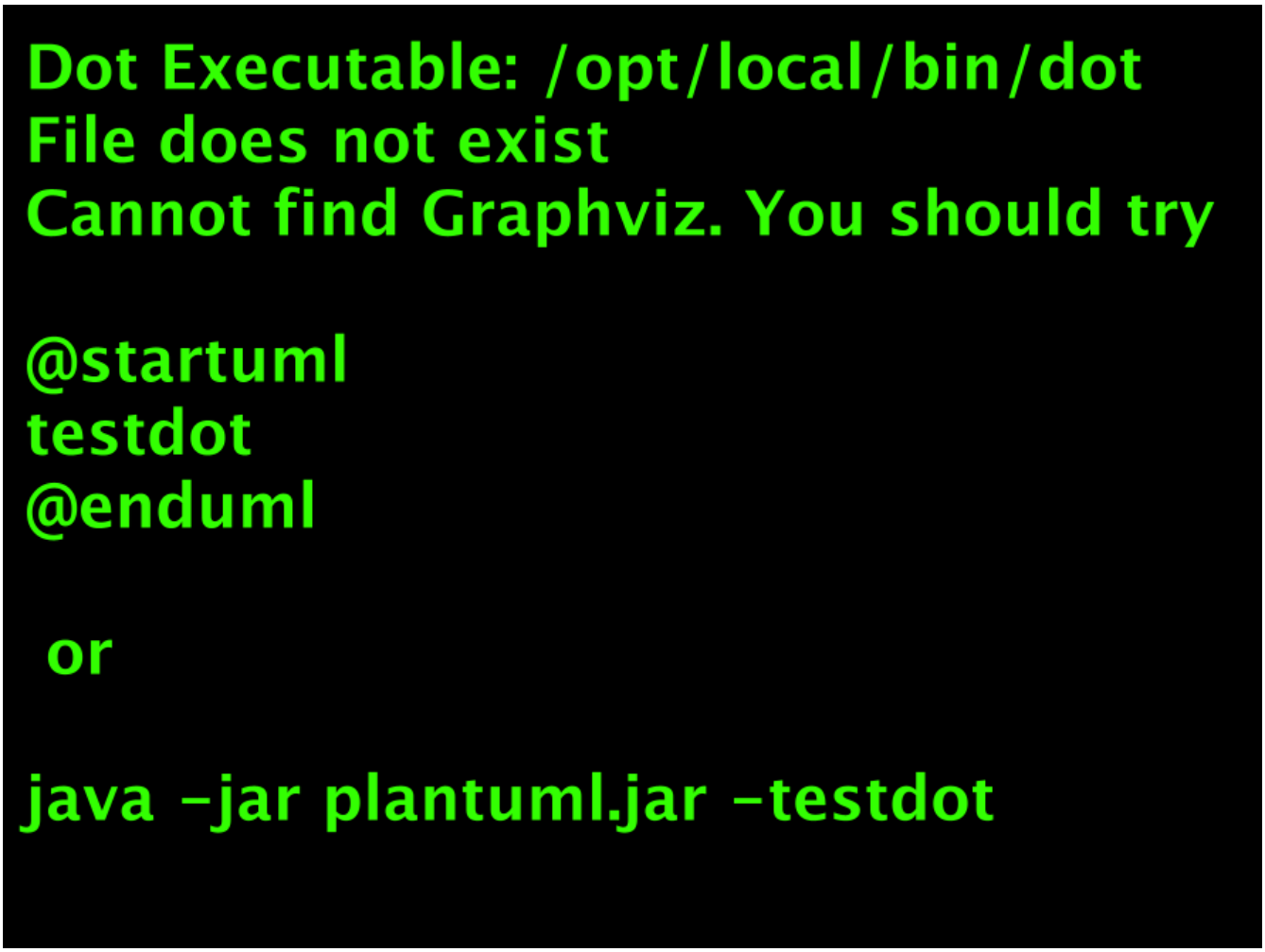
Remember the function to retrieve random fighters. We are currently returning `null`. Let's move to the next session to see how we can implement this method.

[26] Swagger Codegen https://github.com/swagger-api/swagger-codegen

[27] CORS https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

HTTP communication & Fault Tolerance

So far we've built one Fight microservice which need to invoke the Hero and Villain microservices. In the following sections you will develop this invocation thanks to the MicroProfile REST Client. We will also deal with fault tolerance thanks to timeouts and circuit breaker.

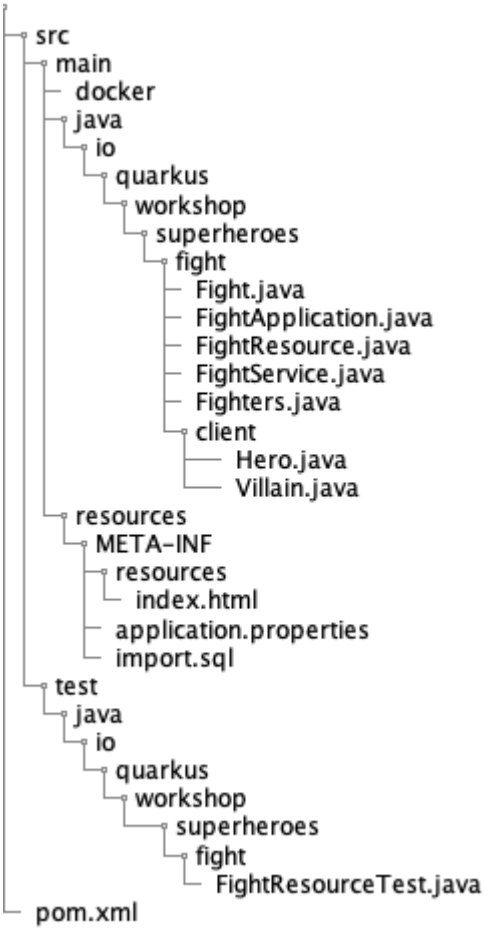


REST Client

This chapter explains how to use the MicroProfile REST Client in order to interact with REST APIs with very little effort.[28]

Directory Structure

Remember the structure of the Fight microservice:



We are going to rework the:

- `FightService` class
- `FightResourceTest` class
- `application.properties`

Installing the REST Client Dependency

[hand o right] Call to action

To install the MicroProfile REST Client dependency, just run the following command:

```
$ ./mvnw quarkus:add-extension -Dextensions="rest-client"
```

This will add the following dependency in the `pom.xml` file:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-rest-client</artifactId>
</dependency>
```

FightService Invoking External Microservices

Remember that in the previous sections we left the `FightService.findRandomFighters()` method returns `null`. We have to fix this. What we actually want is to invoke both the Hero and Villain APIs, asking for a random hero and a random villain.

[hand o right] Call to action

For that, replace the `findRandomFighters` method with the following code to the `FightService` class:

```
@Inject
@RestClient
HeroService heroService;

@Inject
@RestClient
VillainService villainService;

Fighters findRandomFighters() {
    Hero hero = findRandomHero();
    Villain villain = findRandomVillain();
    Fighters fighters = new Fighters();
    fighters.hero = hero;
    fighters.villain = villain;
    return fighters;
}

Hero findRandomHero() {
    return heroService.findRandomHero();
}

Villain findRandomVillain() {
    return villainService.findRandomVillain();
}
```

Note that in addition to the standard CDI `@Inject` annotation, we also need to use the MicroProfile `@RestClient` annotation to inject `HeroService` and `VillainService`.

If not done automatically by your IDE, add the following import statement: `import org.eclipse.microprofile.rest.client.inject.RestClient;`

Creating the Interfaces

Using the MicroProfile REST Client is as simple as creating an interface using the proper JAX-RS and MicroProfile annotations.

[hand o right] Call to action

In our case both interfaces should be created under the `client` subpackage and have the following content:

```
package io.quarkus.workshop.superheroes.fight.client;

import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

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

@Path("/api/heroes")
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient
public interface HeroService {

    @GET
    @Path("/random")
    Hero findRandomHero();
}
```

The `findRandomHero` method gives our code the ability to query a random hero from the Hero REST API. The client will handle all the networking and marshalling leaving our code clean of such technical details.

The purpose of the annotations in the code above is the following:

- `@RegisterRestClient` allows Quarkus to know that this interface is meant to be available for CDI injection as a REST Client
- `@Path` and `@GET` are the standard JAX-RS annotations used to define how to access the service
- `@Produces` defines the expected content-type

The `VillainService` is very similar and looks like this:

```
package io.quarkus.workshop.superheroes.fight.client;

import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

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

@Path("/api/villains")
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient
public interface VillainService {

    @GET
    @Path("/random")
    Villain findRandomVillain();
}
```

[hand o right] Call to action

Once created, go back to the `FightService` class and add the following import statements:

```
import io.quarkus.workshop.superheroes.fight.client.HeroService;
import io.quarkus.workshop.superheroes.fight.client.VillainService;
```

Configuring REST Client Invocation

[hand o right] Call to action

In order to determine the base URL to which REST calls will be made, the REST Client uses configuration from `application.properties`. The name of the property needs to follow a certain convention which is best displayed in the following code:

```
io.quarkus.workshop.superheroes.fight.client.HeroService/mp-
rest/url=http://localhost:8083
io.quarkus.workshop.superheroes.fight.client.HeroService/mp-
rest/scope=javax.inject.Singleton
io.quarkus.workshop.superheroes.fight.client.VillainService/mp-
rest/url=http://localhost:8084
io.quarkus.workshop.superheroes.fight.client.VillainService/mp-
rest/scope=javax.inject.Singleton
```

Having this configuration means that all requests performed using `HeroService` will use http://localhost:8083 as the base URL. Using this configuration, calling the `findRandomHero` method of `HeroService` would result in an HTTP GET request being made to http://localhost:8083/api/heroes/random.

Having this configuration means that the default scope of `HeroService` will be `@Singleton`. Supported scope values are `@Singleton`, `@Dependent`, `@ApplicationScoped` and `@RequestScoped`. The default scope is `@Dependent`. The default scope can also be defined on the interface.

Now, go back in the UI and refresh, you should see some pictures!

Updating the Test with Mock Support

But, now we have another problem. To run the tests of the Fight API we need the Hero and Villain REST APIs to be up and running. To avoid this, we need to Mock the `HeroService` and `VillainService` interfaces.

Quarkus supports the use of mock objects using the CDI `@Alternative` mechanism.[29]

[hand o right] Call to action

To use this simply override the bean you wish to mock with a class in the `src/test/java` directory, and put the `@Alternative` and `@Priority(1)` annotations on the bean. Alternatively, a convenient `io.quarkus.test.Mock` stereotype annotation could be used. This built-in stereotype declares `@Alternative`, `@Priority(1)` and `@Dependent`. So, to mock the `HeroService` interface we just need to implement the following `MockHeroService` class:

```
package io.quarkus.workshop.superheroes.fight.client;

import io.quarkus.test.Mock;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import javax.enterprise.context.ApplicationScoped;

@Mock
@ApplicationScoped
@RestClient
public class MockHeroService implements HeroService {

    public static final String DEFAULT_HERO_NAME = "Super Baguette";
    public static final String DEFAULT_HERO_PICTURE = "super_baguette.png";
    public static final String DEFAULT_HERO_POWERS = "eats baguette really quickly";
    public static final int DEFAULT_HERO_LEVEL = 42;

    @Override
    public Hero findRandomHero() {
        Hero hero = new Hero();
        hero.name = DEFAULT_HERO_NAME;
        hero.picture = DEFAULT_HERO_PICTURE;
        hero.powers = DEFAULT_HERO_POWERS;
        hero.level = DEFAULT_HERO_LEVEL;
        return hero;
    }
}
```

[hand o right] Call to action

Do the same for the `MockVillainService`:

```
package io.quarkus.workshop.superheroes.fight.client;

import io.quarkus.test.Mock;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import javax.enterprise.context.ApplicationScoped;

@Mock
@ApplicationScoped
@RestClient
public class MockVillainService implements VillainService {

    public static final String DEFAULT_VILLAIN_NAME = "Super Chocolatine";
    public static final String DEFAULT_VILLAIN_PICTURE = "super_chocolatine.png";
    public static final String DEFAULT_VILLAIN_POWERS = "does not eat pain au
chocolat";
    public static final int DEFAULT_VILLAIN_LEVEL = 42;

    @Override
    public Villain findRandomVillain() {
        Villain villain = new Villain();
        villain.name = DEFAULT_VILLAIN_NAME;
        villain.picture = DEFAULT_VILLAIN_PICTURE;
        villain.powers = DEFAULT_VILLAIN_POWERS;
        villain.level = DEFAULT_VILLAIN_LEVEL;
        return villain;
    }
}
```

[hand o right] Call to action

Finally, edit the `FightResourceTest` and add the following method:

```
import io.quarkus.workshop.superheroes.fight.client.MockHeroService;
import io.quarkus.workshop.superheroes.fight.client.MockVillainService;

    //....
    @Test
    void shouldGetRandomFighters() {
        given()
            .when().get("/api/fights/randomfighters")
            .then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .body("hero.name", Is.is(MockHeroService.DEFAULT_HERO_NAME))
            .body("hero.picture", Is.is(MockHeroService.DEFAULT_HERO_PICTURE))
            .body("hero.level", Is.is(MockHeroService.DEFAULT_HERO_LEVEL))
            .body("villain.name", Is.is(MockVillainService.DEFAULT_VILLAIN_NAME))
            .body("villain.picture", Is.is(MockVillainService.DEFAULT_VILLAIN_PICTURE
))
            .body("villain.level", Is.is(MockVillainService.DEFAULT_VILLAIN_LEVEL));
    }
```

You would need the following import statements:

```
import io.quarkus.workshop.superheroes.fight.client.MockHeroService;
import io.quarkus.workshop.superheroes.fight.client.MockVillainService;
```

Running and Testing the Application

[hand o right] Call to action

First, make sure the tests pass by executing the command `./mvnw test` (or from your IDE).

Now that the tests are green, we are ready to run our application. Use `./mvnw compile quarkus:dev` to start it. Once the application is started, go to http://localhost:8080 and start fighting (finally !).

Fallbacks (optional)

This chapter is optional, you can jump to the Timeout (optional) chapter, Observability (optional) chapter or to the Event-driven and Reactive microservices chapter if you already know about fault-tolerance and fallback.

So now you've been playing this great Super Heroes Fight for a few hours... and you kill the Hero REST API. What happens? Well, the Fight REST API cannot invoke the Hero API anymore and breaks with the following exception:

```
ERROR [io.qu.ve.ht.ru.QuarkusErrorHandler] HTTP Request to /api/fights/randomfighters
failed:
org.jboss.resteasy.spi.UnhandledException: javax.ws.rs.ProcessingException:
RESTEASY004655: Unable to invoke request: java.net.ConnectException: Connection
refused (Connection refused)
at
org.jboss.resteasy.core.ExceptionHandler.handleApplicationException(ExceptionHandler.j
ava:106)
at org.jboss.resteasy.core.ExceptionHandler.handleException(ExceptionHandler.java:372)
at
org.jboss.resteasy.core.SynchronousDispatcher.writeException(SynchronousDispatcher.jav
a:209)
at
org.jboss.resteasy.core.SynchronousDispatcher.invoke(SynchronousDispatcher.java:496)
```

One of the challenges brought by the distributed nature of microservices is that communication with external systems is inherently unreliable. This increases demand on resiliency of applications. To simplify making more resilient applications, Quarkus contains an implementation of the MicroProfile Fault Tolerance specification.[30]

Installing the Fault Tolerance Dependency

To install the MicroProfile Fault Tolerance dependency, just run the following command:

[hand o right] Call to action

```
$ ./mvnw quarkus:add-extension -Dextensions="smallrye-fault-tolerance"
```

This will add the following dependency in the `pom.xml` file:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-fault-tolerance</artifactId>
</dependency>
```

Adding Fallbacks

Let's make our find random fighters feature better by providing a fallback way of getting a dummy hero or villain in case of failure.

[hand o right] Call to action

For that, add two fallback methods to the `FightService` and a `@Fallback` annotation to both `findRandomHero` and `findRandomVillain` methods as follows:

```
@Inject
@RestClient
HeroService heroService;

@Inject
@RestClient
VillainService villainService;

Fighters findRandomFighters() {
    Hero hero = findRandomHero();
    Villain villain = findRandomVillain();
    Fighters fighters = new Fighters();
    fighters.hero = hero;
    fighters.villain = villain;
    return fighters;
}

@Fallback(fallbackMethod = "fallbackRandomHero")
Hero findRandomHero() {
    return heroService.findRandomHero();
}

@Fallback(fallbackMethod = "fallbackRandomVillain")
Villain findRandomVillain() {
    return villainService.findRandomVillain();
}

public Hero fallbackRandomHero() {
    LOGGER.warn("Falling back on Hero");
    Hero hero = new Hero();
    hero.name = "Fallback hero";
    hero.picture = "https://dummyimage.com/280x380/1e8fff/ffffff&text=Fallback+Hero";
    hero.powers = "Fallback hero powers";
    hero.level = 1;
    return hero;
}

public Villain fallbackRandomVillain() {
    LOGGER.warn("Falling back on Villain");
    Villain villain = new Villain();
    villain.name = "Fallback villain";
    villain.picture =
"https://dummyimage.com/280x380/b22222/ffffff&text=Fallback+Villain";
    villain.powers = "Fallback villain powers";
    villain.level = 42;
    return villain;
}
```

[hand o right] Call to action

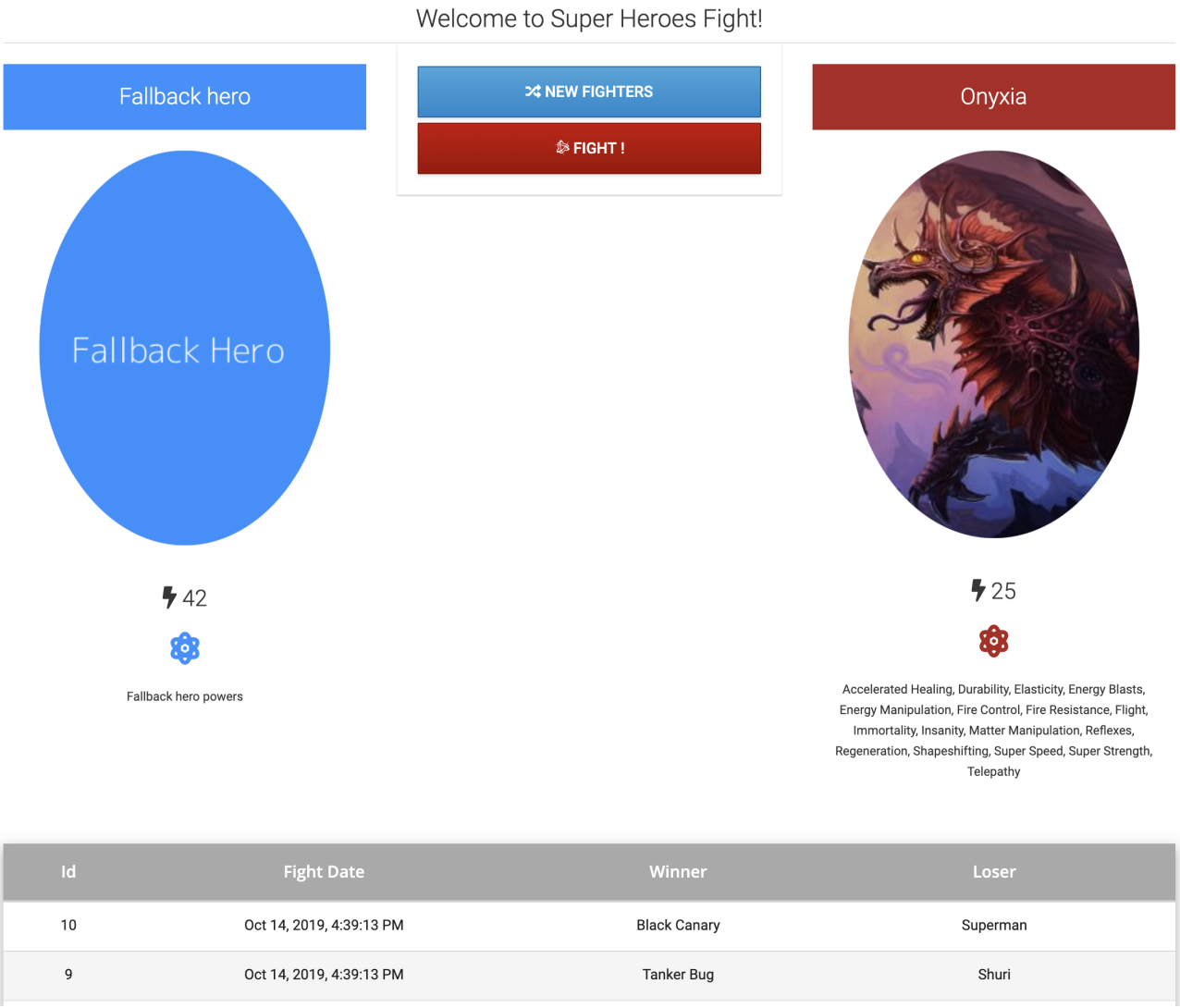
Also add the `import org.eclipse.microprofile.faulttolerance.Fallback;` statement.

Running the Application

Now we are ready to run our application and test the fallbacks.

[hand o right] Call to action

For that, kill the Hero (and/or the Villain API) and start playing again. You should see the following:

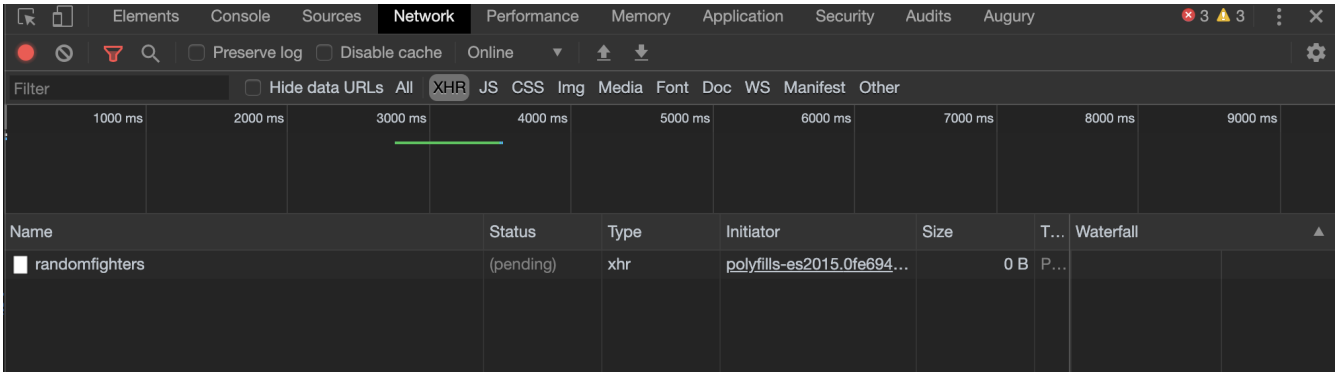


Restart the Hero REST API... and keep on playing. Super heroes are back to the fight!

Timeout (optional)

This chapter is optional, you can jump to the Observability (optional) chapter or to the Event-driven and Reactive microservices chapter if you already know about fault-tolerance and timeout.

Sometimes invoking a REST API can take a long time. In fact, the more microservices invoke other microservices, the more network latency you can have. And what happens when a HTTP request takes long? Well, if hangs. On your browser you can see the request pending if you turn on the dev tools and look at what's the network is doing.



Adding Timeouts

Getting random fighters can take longer than expected.

[hand o right] Call to action

To simulate a long running process, update the `FightResource` with the following code:

```
@ConfigProperty(name = "process.milliseconds", defaultValue="0")
long tooManyMilliseconds;

private void veryLongProcess() throws InterruptedException {
    Thread.sleep(tooManyMilliseconds);
}

@Operation(summary = "Returns two random fighters")
@APIResponse(responseCode = "200", content = @Content(mediaType = APPLICATION_JSON,
schema = @Schema(implementation = Fighters.class, required = true)))
@Timeout(250)
@GET
@Path("/randomfighters")
public Response getRandomFighters() throws InterruptedException {
    veryLongProcess();
    Fighters fighters = service.findRandomFighters();
    LOGGER.debug("Get random fighters " + fighters);
    return Response.ok(fighters).build();
}
```

Don't forget to add the following import: `import org.eclipse.microprofile.config.inject.ConfigProperty;`

Let's say we've added some new functionality in the `veryLongProcess` method. When the process is really too long and the system is overloaded, we would rather time out.

Here we throw the `InterruptedException` in the method signature. But what you would have to do in real life is to catch it, and propose a fallback algorithm when the process times out.

[hand o right] Call to action

Note that the timeout was configured to 250 ms, and a `Thread.sleep` was introduced and can be configured in the `application.properties`. If we set the property to a higher value than the timeout, let's say 10.000, then the request should be interrupted.

```
process.milliseconds=10000
```

Running the Application

Now that you have set the number of waiting milliseconds to 10.000, run the application and start fighting again. You should see the following:

```
ERROR [io.qu.ve.ht.ru.QuarkusErrorHandler] HTTP Request to /api/fights/randomfighters
failed:
org.jboss.resteasy.spi.UnhandledException:
org.eclipse.microprofile.faulttolerance.exceptions.TimeoutException:
com.netflix.hystrix.exception.HystrixRuntimeException:
io_quarkus_workshop_superheroes_fight_FightResource#getRandomFighters() timed-out and
no fallback available.
at
org.jboss.resteasy.core.ExceptionHandler.handleApplicationException(ExceptionHandler.j
ava:106)
```

[hand o right] Call to action

Before going further, set the `process.milliseconds` to 0.

If you have any problem with the code, don't understand or feel you are running, remember to ask for some help. Also, you can get the code of this entire workshop from https://github.com/quarkusio/quarkus-workshops/tree/master/quarkus-workshop-super-heroes.

[28] MicroProfile REST Client https://github.com/eclipse/microprofile-rest-client

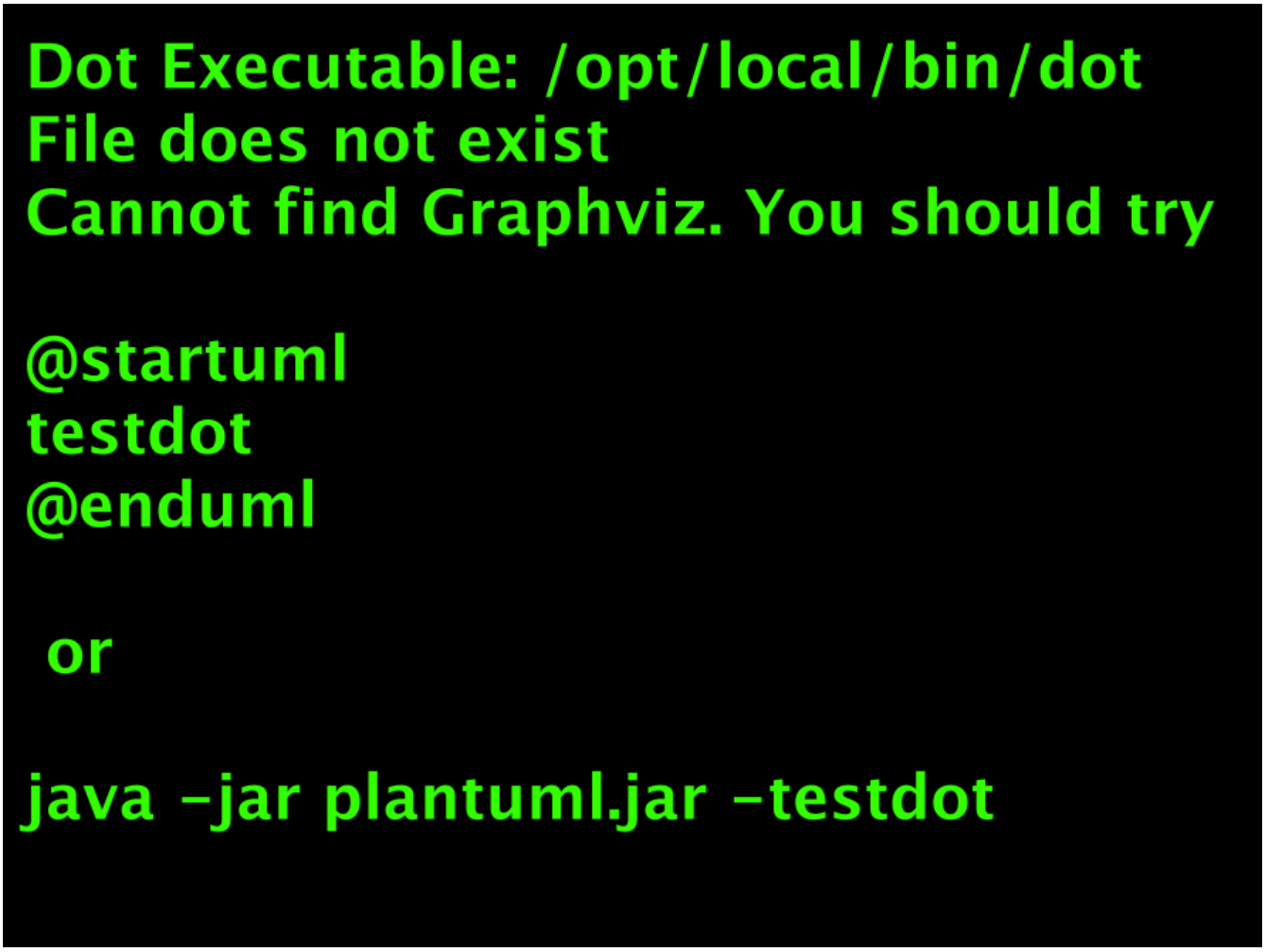
[29] Alternatives https://docs.jboss.org/weld/reference/latest/en-US/html/beanscdi.html#_alternatives

[30] MicroProfile Fault Tolerance https://github.com/eclipse/microprofile-fault-tolerance

Observability (optional)

This chapter is optional. It covers health checks, metrics and prometheus. You can jump to the Event-driven and Reactive microservices chapter if you want.

Now that we have several microservices, observing them starts to be a bit tricky: we can't just look at the logs of all the microservices to see if they are up and running or behaving correctly. In the following sections you will add health checks and several metrics to the Fight, Hero and Villain APIs and gather them within Promotheus.

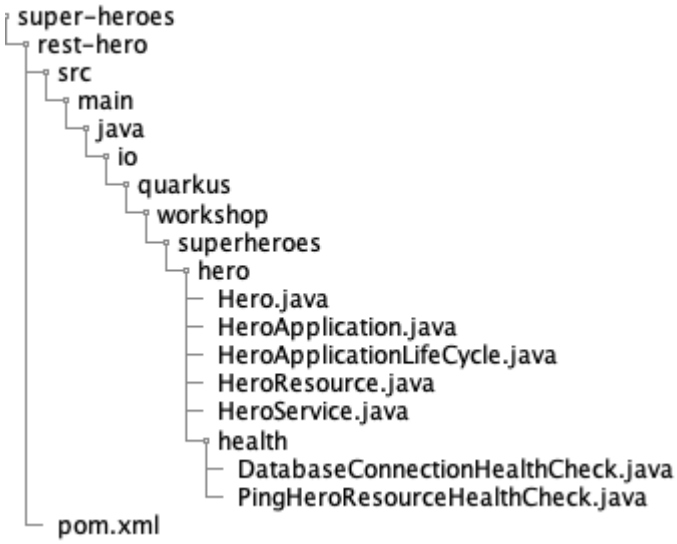


Health Check

Quarkus applications can utilize the MicroProfile Health specification through the SmallRye Health extension. The MicroProfile Health allows applications to provide information about their state to external viewers which is typically useful in cloud environments where automated processes must be able to determine whether the application should be discarded or restarted.[31]

Directory Structure

In this module we will add an extra subdirectory with two classes to handle the Health Check. You will end-up with the following directory structure:



While you could add health checks to all our microservices, we are focusing on the **Hero microservice**. So don't forget to switch to the Hero microservice code. You can apply the same to the other microservices.

Installing the Health Dependency

[hand o right] Call to action

To install the MicroProfile Health dependency, just run the following command:

```
$ ./mvnw quarkus:add-extension -Dextensions="health"
```

This will add the following dependency in the `pom.xml` file:

```
```

Running the Default Health Check

Importing the `smallrye-health` extension directly exposes three REST endpoints:

- `/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.

To check that the `smallrye-health` extension is working as expected, access using your browser or cURL:

- http://localhost:8083/health/live
- http://localhost:8083/health/ready
- http://localhost:8083/health

All of 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

The general `status` of the health check is computed as a logical AND of all the declared health check procedures. The checks array is empty as we have not specified any health check procedure yet so let's define some.

Adding Liveness

To check that our Hero API application is live, we can check that the `HeroResource.hello()` method works.

[hand o right] Call to action

For that, this is the `PingHeroResourceHealthCheck` class that we can write under the `io.quarkus.workshop.superheroes.hero.health` sub-package:

```
package io.quarkus.workshop.superheroes.hero.health;

import io.quarkus.workshop.superheroes.hero.HeroResource;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Liveness;

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

@Liveness
@ApplicationScoped
public class PingHeroResourceHealthCheck implements HealthCheck {

    @Inject
    HeroResource heroResource;

    @Override
    public HealthCheckResponse call() {
        heroResource.hello();
        return HealthCheckResponse.named("Ping Hero REST Endpoint").up().build();
    }
}
```

As you can see health check procedures are defined as implementations of the `HealthCheck` interface which are defined as CDI beans with the CDI qualifier `@Liveness`. The liveness check accessible at `/health/live`. `HealthCheck` is a functional interface whose single method `call` returns a `HealthCheckResponse` object which can be easily constructed by the fluent builder API shown in the example.

[hand o right] Call to action

As we have started our Quarkus application in dev mode simply repeat the request to http://localhost:8083/health/live by refreshing your browser window or by using curl http://localhost:8083/health/live. Because we defined our health check to be a liveness procedure (with `@Liveness` qualifier) the new health check procedure is now present in the checks array.

```
{
    "status": "UP",
    "checks": [
        {
            "name": "Ping Hero REST Endpoint",
            "status": "UP"
        }
    ]
}
```

Adding Readiness

We've just created a simple liveness health check procedure which states whether our application is running or not. Here, we will create a readiness health check which will be able to state whether our application is able to process requests.

We will create another health check procedure that accesses our database. If the database can be accessed, then we will always return the response indicating the application is ready.

[hand o right] Call to action

Create the `io.quarkus.workshop.superheroes.hero.health.DatabaseConnectionHealthCheck` class as follow:

```
package io.quarkus.workshop.superheroes.hero.health;

import io.quarkus.workshop.superheroes.hero.Hero;
import io.quarkus.workshop.superheroes.hero.HeroService;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.HealthCheckResponseBuilder;
import org.eclipse.microprofile.health.Readiness;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import java.util.List;

@Readiness
@ApplicationScoped
public class DatabaseConnectionHealthCheck implements HealthCheck {

    @Inject
    HeroService heroService;

    @Override
    public HealthCheckResponse call() {
        HealthCheckResponseBuilder responseBuilder = HealthCheckResponse
            .named("Hero Datasource connection health check");

        try {
            List<Hero> heroes = heroService.findAllHeroes();
            responseBuilder.withData("Number of heroes in the database", heroes.size(
)).up();
        } catch (IllegalStateException e) {
            responseBuilder.down();
        }

        return responseBuilder.build();
    }
}
```

If you now rerun the health check at http://localhost:8083/health/live the checks array will contain only the previously defined `PingHeroResourceHealthCheck` as it is the only check defined with the `@Liveness` qualifier. However, if you access http://localhost:8083/health/ready (in the browser or with curl http://localhost:8083/health/ready) you will see only the Database connection health check as it is the only health check defined with the `@Readiness` qualifier as the readiness health check procedure. If you access http://localhost:8083/health you will get back both checks.

```
{
    "status": "UP",
    "checks": [
        {
            "name": "Hero health check",
            "status": "UP",
            "data": {
                "rows": 951
            }
        },
        {
            "name": "Database connection(s) health check",
            "status": "UP"
        }
    ]
}
```

Health Check Tests in HeroResourceTest

[hand o right] Call to action

Let's add a few extra test methods that would make sure Health Check are available in the application:

```
```

Here we've just shown you the health check for the Hero API, but you should do the same for Fight and Villain.

Metrics

MicroProfile Metrics allows applications to gather various metrics and statistics that provide insights into what is happening inside the application.[32] The metrics can be read remotely using JSON format or the OpenMetrics format, so that they can be processed by additional tools such as Prometheus, and stored for analysis and visualisation.[33]

While you could add add metrics to all our microservices, we are focusing on the Hero microservice. You can apply the same to the other microservices.

Installing the Metrics Dependency

[hand o right] Call to action

To install the MicroProfile Metrics dependency, just run the following command:

```
$ ./mvnw quarkus:add-extension -Dextensions="metrics"
```

This will add the following dependency in the `pom.xml` file:

```
```

Adding Metrics to HeroResource

We want now to measure all methods of all our REST resources.

[hand o right] Call to action

For that, we need a few annotations to make sure that our desired metrics are calculated over time and can be exported for manual analysis or processing by additional tooling. The metrics that we will gather are these:

- `countGetRandomHero`: A counter which is increased by one each time the user gets a random hero.
- `timeGetRandomHero`: This is a timer, therefore a compound metric that benchmarks how much time the request take.

Below the source code of the `getRandomHero()` method, but make sure to add metrics on all methods:

```
```

[hand o right] Call to action

You will need to add the following import statements:

```
import org.eclipse.microprofile.metrics.MetricUnits;
import org.eclipse.microprofile.metrics.annotation.Counted;
import org.eclipse.microprofile.metrics.annotation.Timed;
```

Metrics Tests in HeroResourceTest

[hand o right] Call to action

Let's add an extra test method that would make sure Metrics are available in the application:

```
```

Reviewing the Generated Metrics

To view the metrics, execute `curl -H "Accept: application/json" http://localhost:8083/metrics/application`. You will receive a response such as:

```
{
  "io.quarkus.workshop.superheroes.hero.HeroResource.countGetRandomHero": 44,
  "io.quarkus.workshop.superheroes.hero.HeroResource.timeGetRandomHero": {
    "p99": 16.227182,
    "min": 2.525987,
    "max": 16.227182,
    "mean": 3.202769680486923,
    "p50": 2.967352,
    "p999": 16.227182,
    "stddev": 1.55809730109504,
    "p95": 3.565725,
    "p98": 4.157616,
    "p75": 3.104259,
    "fiveMinRate": 0.12382047800943181,
    "fifteenMinRate": 0.04406239601227314,
    "meanRate": 0.08741866976313094,
    "count": 44,
    "oneMinRate": 0.4332024919472982
  }
}
```

Let's explain the meaning of each metric:

- `countGetRandomHero`: A counter which is increased by one each time the user asks for a random hero.
- `timeGetRandomHero`: This is a timer, therefore a compound metric that benchmarks how much time the request takes. All durations are measured in milliseconds. It consists of these values:

- `min`: The shortest duration it took to perform a request.
- `max`: The longest duration.
- `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 `countGetRandomHero`).
- `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.

If you prefer an OpenMetrics export rather than the JSON format, remove the `-H "Accept: application/json"` argument from your command line.

Again, in this chapter, we've just shown you how to add metrics for the Hero API, but you should do the same for Fight and Villain.

Loading the Microservices

Now that we have the three main microservices exposing health checks and metrics, time to have a decent user interface to monitor how the system behaves. The purpose of this workshop is to add some load to our application. You will download the load application, install it and run it.

Give me some load!

In the `super-heroes/load-super-heroes` directory, there is an application that is **NOT** a Quarkus application. It's a simple Java app that simulates users interacting with the system so it generates some load.

Looking at Some Code

The `SuperHeroesLoad` class is just a `main` that executes the `FightScenario`, `HeroScenario` and `VillainScenario` in different threads. For example, if you look at the `HeroScenario`, you will see that it's just a suit of HTTP calls on the Hero API:

```
private static final String targetUrl = "http://localhost:8083";

private static final String contextRoot = "/api/heroes";

@Override
protected List<Endpoint> getEndpoints() {
    return Stream.of(
         endpoint(contextRoot, "GET"),
         endpoint(contextRoot + "/hello", "GET"),
         endpoint(contextRoot + "/random", "GET"),
         endpointWithTemplates(contextRoot + "/{id}", "GET", this::idParam),
         endpointWithTemplates(contextRoot + "/{id}", "DELETE", this::idParam),
         endpointWithEntity(contextRoot, "POST", this::createHero)
    ).collect(collectingAndThen(toList(), Collections::unmodifiableList));
}
```

Running the Load Application

[hand o right] Call to action

You are all set! Time to compile and start the load application using:

```
$ mvn compile
$ mvn exec:java
```

You will see the following logs:

```
INFO: GET - http://localhost:8082/api/fights/1 - 200
INFO: DELETE - http://localhost:8084/api/villains/440 - 204
INFO: GET - http://localhost:8083/api/heroes - 200
INFO: GET - http://localhost:8084/api/villains/hello - 200
INFO: GET - http://localhost:8082/api/fights - 200
INFO: GET - http://localhost:8083/api/heroes/581 - 200
INFO: GET - http://localhost:8084/api/villains/126 - 200
INFO: GET - http://localhost:8082/api/fights/hello - 200
INFO: DELETE - http://localhost:8083/api/heroes/491 - 204
```

Displaying Metrics on Prometheus

Now that we've added some load to our application, let's measure it with Prometheus.[34] Prometheus is an open-source systems monitoring and alerting toolkit that integrates well with Quarkus.

Configuring Prometheus

Prometheus needs to be configured to poll data from our microservices. This is made under our `infrastructure` directory, in the `prometheus.yml` file:

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'fights'
    static_configs:
      - targets: ['host.docker.internal:8082']
  - job_name: 'heroes'
    static_configs:
      - targets: ['host.docker.internal:8083']
  - job_name: 'villains'
    static_configs:
      - targets: ['host.docker.internal:8084']
```

This file contains basic Prometheus configuration, plus a specific `scrape_config` which instructs Prometheus to look for application metrics from both Prometheus itself, and our Quarkus microservices at the `/metrics` endpoint.

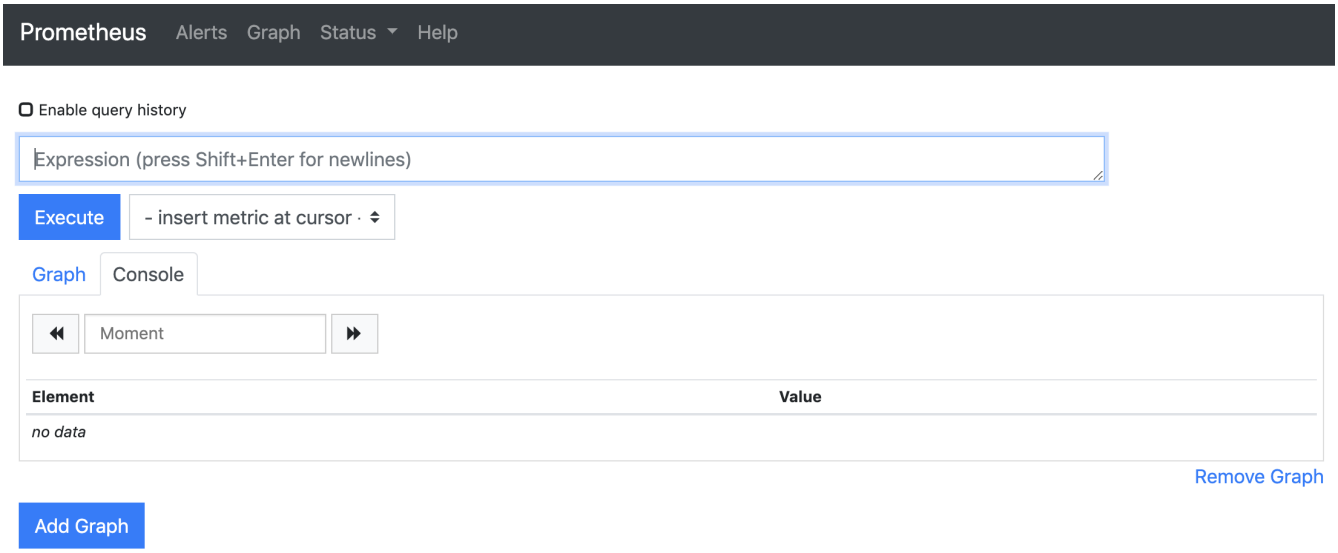
To execute the application we now need Prometheus. Make sure the infrastructure is up and running. This means that you've executed `docker-compose -f docker-compose.yaml up -d`.

On Linux / Ubuntu

On Linux, you may need to update the `localhost` from the `prometheus-linux.yml` file with the IP associated with the `docker0` interface. Find the address, update the `prometheus-linux.yml` and restart the infrastructure. Don't forget to restart the microservices to re-populate the databases.

Adding Graphs to Prometheus

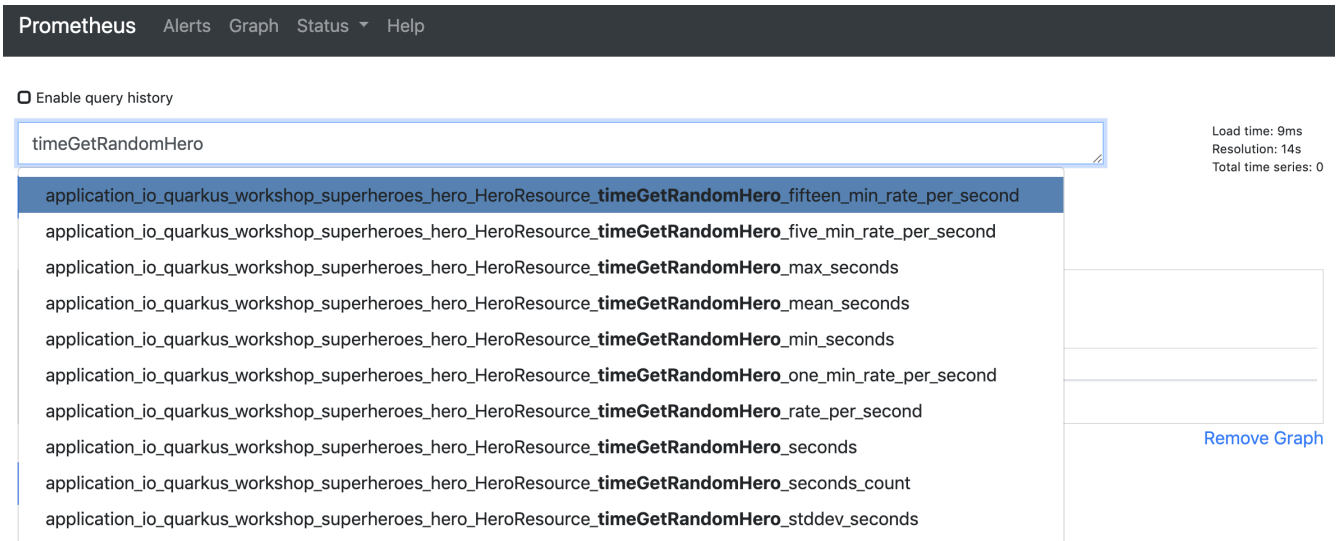
The Prometheus console is accessible at http://localhost:9090.



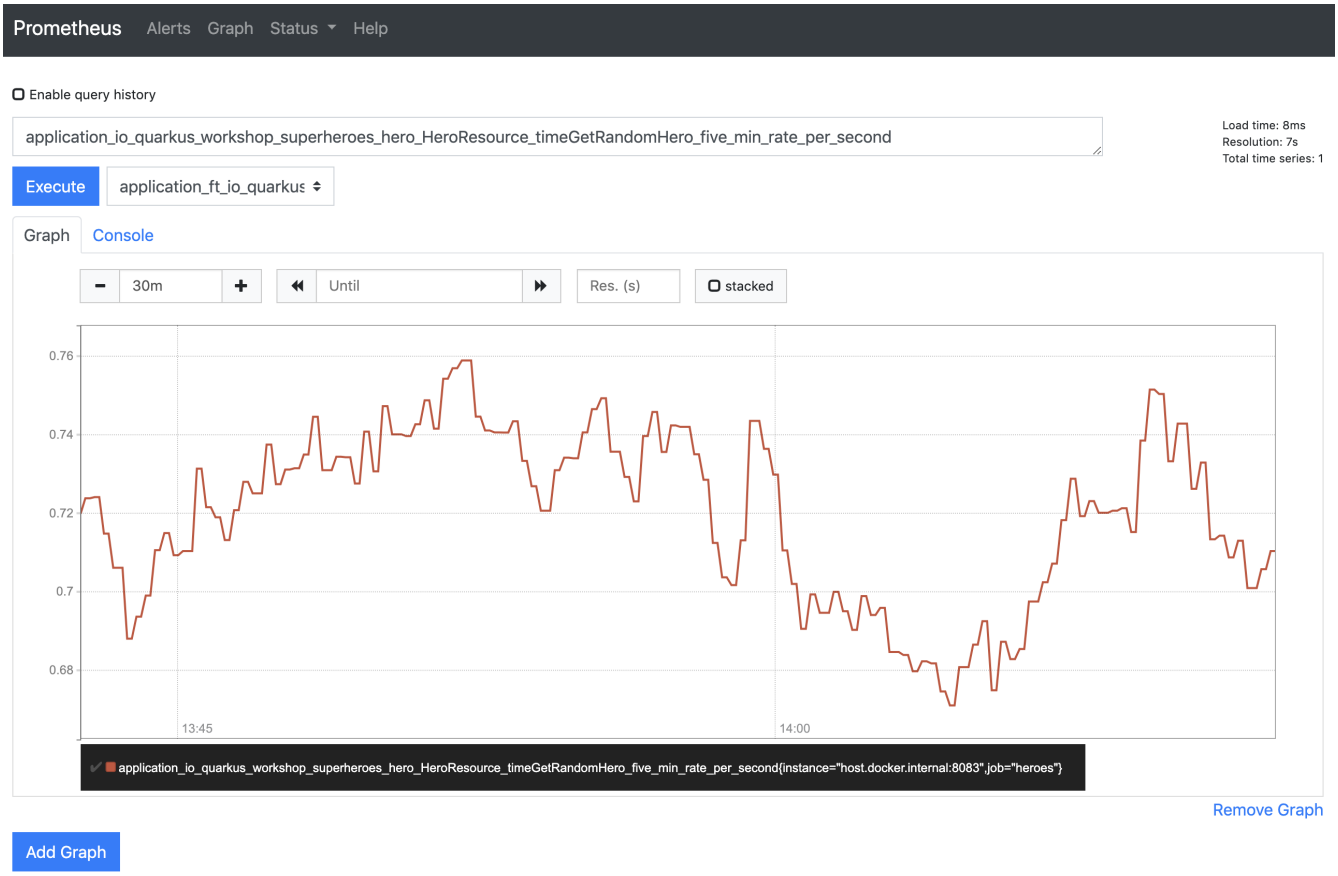
Out of the box, you get a lot of basic JVM metrics or even metrics of Prometheus itself, which are useful. But let's create new graphs with the metrics of our microservices.

[hand o right] Call to action

Tape `timeGetRandomHero` in the query, and select `application_io_quarkus_workshop_superheroes_hero_HeroResource_timeGetRandomHero_five_min_rate_per_second` in the box, and click *Execute*.



This will fetch the values from our metric showing the number of checks performed:



If you have any problem with the code, don't understand or feel you are running, remember to ask for some help. Also, you can get the code of this entire workshop from https://github.com/quarkusio/quarkus-workshops/tree/master/quarkus-workshop-super-heroes.

Stop the load application before going further.

[31] MicroProfile Health https://microprofile.io/project/eclipse/microprofile-health

[32] MicroProfile Metrics https://microprofile.io/project/eclipse/microprofile-metrics

[33] OpenMetrics https://openmetrics.io

[34] Prometheus https://prometheus.io

Event-driven and Reactive microservices

So far, we have build 3 microservices, all using HTTP to interact. However, HTTP has significant flaws, such as temporal coupling between the actors. If the service is not there or is slow, the caller is directly impacted. Also, it's hard to guess the capacity of the service you call; maybe you should not call it right now because this service is under heavy load.

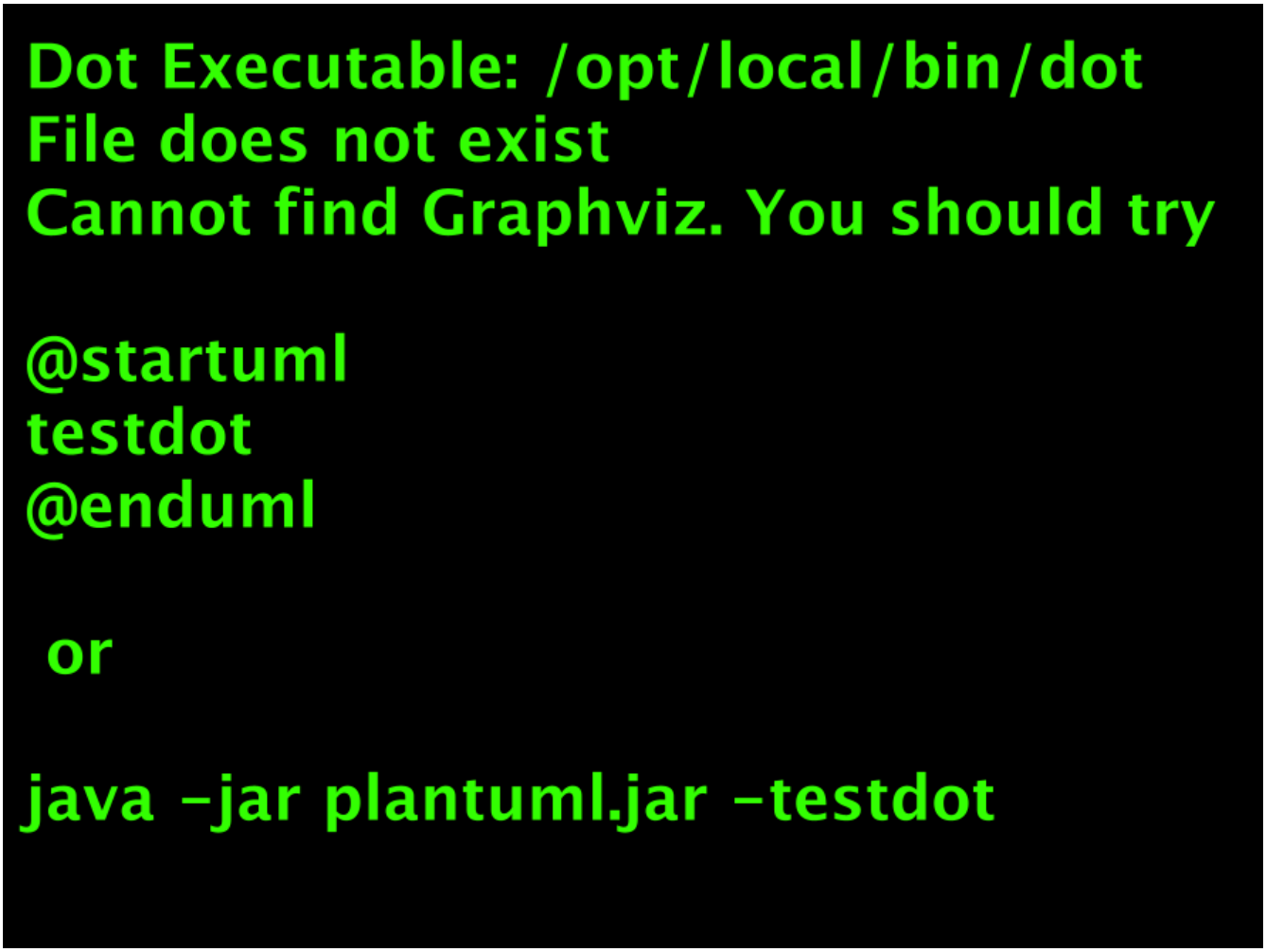
Fortunately, event-driven microservices are rising and avoid most of these issues. By using events (wrapped in messages), the different microservices enforce a looser coupling. Depending on the messaging protocol you use, it may handle durability (avoiding the temporal coupling) and back-pressure (avoiding the overload).

In this section, we are going to see how Quarkus let you build event-driven microservices. More specially, you are going to see how to:

- send messages and process them
- connect a Quarkus application to Apache Kafka
- write Kafka records and read them
- use reactive programming to compute statistics on the fly
- how to send messages to the browser using web sockets

Quarkus uses MicroProfile Reactive Messaging to interact with Kafka, and other messaging middleware (such as AMQP).[35]

In this chapter, we are going to use events as a way for microservices to interact. You are going to extend the current system with the `stats` group depicted on the next figure:



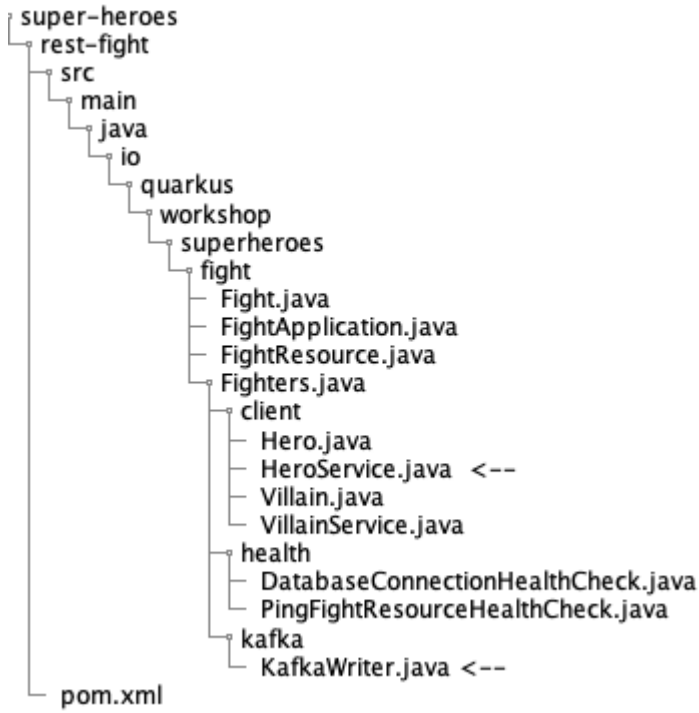
When the application persists a new fight, in the *fight* microservice, you are going to send it to a Kafka topic. These messages are read in the *statistics* microservice, processed, and the result is sent to a UI using web sockets.

Sending Messages to Kafka

In this section, you are going to see how you can send messages to a Kafka topic.[36]

Directory Structure

In this section we are going to extend the **Fight microservice**. In the following tree, we are going to edit the marked classes



Adding the Reactive Messaging Dependency

[hand o right] Call to action

To install the Kafka support, just run the following command:

```
$ ./mvnw quarkus:add-extension -Dextensions="kafka"
```

The previous command adds the following dependency:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-reactive-messaging-kafka</artifactId>
</dependency>
```

The extension may already have been added when the fight microservice has been created.

Then, you can restart the microservice, using `mvn compile quarkus:dev`.

To execute the application we now need Kafka. Make sure the infrastructure is up and running. This means that you've executed `docker-compose -f docker-compose.yaml up -d` (or the Linux variant).

Connecting Imperative and Reactive Using an Emitter

[hand o right] Call to action

Now edit the `FightService` class. First, add the following field:

```
@Inject
@Channel("fights") Emitter<Fight> emitter;
```

You will also need to add the following imports:

```
import io.smallrye.reactive.messaging.annotations.Channel;
import io.smallrye.reactive.messaging.annotations.Emitter;
```

This field is an *emitter*, and lets you send events or messages (here fights) to the *channel* specified with the `@Channel` annotation. A *channel* is a *virtual* destination.

In the `persistFight` method, add the following line just before the `return` statement:

```
emitter.send(fight);
```

With this in place, every time the application persists a `fight`, it also sends the `fight` to the `fights` *channel*.

Connecting to Kafka

At this point, the serialized fights are sent to the `fights` channel. You need to connect this `channel` to a Kafka topic.

[hand o right] Call to action

For this, edit the `application.properties` file and add the following properties:

```
## Kafka configuration
mp.messaging.outgoing.fights.connector=smallrye-kafka
mp.messaging.outgoing.fights.value.serializer=io.quarkus.kafka.client.serialization.JsonbSerializer
```

These properties are structured as follows:

```
mp.messaging.[incoming|outgoing].channel.attribute=value
```

For example, `mp.messaging.outgoing.fights.connector` configures the connector used for the `outgoing` channel `fights`.

The `mp.messaging.outgoing.fights.value.serializer` configures the serializer used to write the message in Kafka. When omitted, the Kafka topic reuses the channel name (`fights`). Also, it connects by default to `localhost:9092`. You can override this using the `kafka.bootstrap.servers` property.

If you go back to the UI and play some fights, you would see, in the fight microservice log, something like:

```
Message
org.eclipse.microprofile.reactive.messaging.Message$$Lambda$1021/763397690@1f6e0af3
sent successfully to Kafka topic 'fights'
```

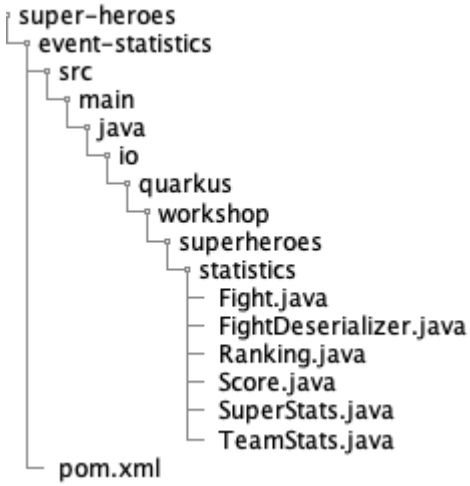
Now, you have connected the fight microservice to Kafka, and you are sending new fights to the Kafka topic. Let's see how you can read these messages in the `stats` microservice.

Receiving Messages from Kafka

In this section, you are going to see how you can receive messages from a Kafka topic. For this, you are going to create a new microservice, named `stats`. This microservice computes statistics based on the results of the fights. For example, it determines if villains win more battle than heroes, and who is the superhero or super-villain having won the most fights.

Directory Structure

In this section, we are going to develop the following structure:



Bootstrapping the Statistics REST Endpoint

Like for the other microservice, the easiest way to create this new Quarkus project is to use a Maven command. Under the `quarkus-workshop-super-heroes/super-heroes` root directory where you have all your code.

[hand o right] Call to action

Open a terminal and run the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.9.2.Final:create \
    -DprojectGroupId=io.quarkus.workshop.super-heroes \
    -DprojectArtifactId=event-statistics \
    -DclassName="io.quarkus.workshop.superheroes.statistics.StatisticResource" \
    -Dpath="api/stats" \
    -Dextensions="kafka, vertx, resteasy-jsonb, undertow-websockets"
cd event-statistics
```

As you can see in the command, you can configure the extensions you want using the `-Dextensions` parameter. Open the `pom.xml` file to see the result.

Computing Statistics

[hand o right] Call to action

Now, create the `io.quarkus.workshop.superheroes.statistics.SuperStats` class with the following content. This class contains 2 methods annotated with `@Incoming` and `@Outgoing`, both consuming the `Fight` coming from Kafka.

The `computeTeamStats` method transform the fight stream into a stream of ratio indicating the amount of victories for heroes and villains. It calls `onItem().transform` method for each received fight. It sends the computed ratios on the channel `team-stats`.

The `computeTopWinners` method uses more advanced reactive programming constructs such as `group` and `scan`:

```
package io.quarkus.workshop.superheroes.statistics;

import io.smallrye.mutiny.Multi;
import io.smallrye.reactive.messaging.annotations.Broadcast;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class SuperStats {

    private final Ranking topWinners = new Ranking(10);
    private final TeamStats stats = new TeamStats();

    @Incoming("fights")
    @Outgoing("team-stats")
    public Multi<Double> computeTeamStats(Multi<Fight> results) {
        return results
            .onItem().transform(stats::add);
    }

    @Incoming("fights")
    @Outgoing("winner-stats")
    public Multi<Iterable<Score>> computeTopWinners(Multi<Fight> results) {
        return results
            .groupItems().by(fight -> fight.winnerName)
            .onItem().transformToMultiAndMerge(group ->
                group
                    .onItem().scan(Score::new, this::incrementScore))
            .onItem().transform(topWinners::onNewScore);
    }

    private Score incrementScore(Score score, Fight fight) {
        score.name = fight.winnerName;
        score.score = score.score + 1;
        return score;

    }

}
```

[hand o right] Call to action

In addition, create the `io.quarkus.workshop.superheroes.statistics.Ranking`, `io.quarkus.workshop.superheroes.statistics.Score` and `io.quarkus.workshop.superheroes.statistics.TeamStats` classes with the following contents:

Then, create the `Ranking` class, used to compute a floating top 10, with the following content:

```
package io.quarkus.workshop.superheroes.statistics;

import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

public class Ranking {

    private final int max;

    private final Comparator<Score> comparator = Comparator.comparingInt(s -> -1 * s
.score);

    private final LinkedList<Score> top = new LinkedList<>();

    public Ranking(int size) {
        max = size;
    }

    public Iterable<Score> onNewScore(Score score) {
        // Remove score if already present,
        top.removeIf(s -> s.name.equalsIgnoreCase(score.name));
        // Add the score
        top.add(score);
        // Sort
        top.sort(comparator);

        // Drop on overflow
        if (top.size() > max) {
            top.remove(top.getLast());
        }

        return Collections.unmodifiableList(top);
    }
}
```

The `Score` class is a simple structure storing the name of a hero or villain and its actual score, *i.e.* the number of won battles.

```
package io.quarkus.workshop.superheroes.statistics;

import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection
public class Score {
    public String name;
    public int score;

    public Score() {
        this.score = 0;
    }
}
```

The `TeamStats` class is an object keeping track of the number of battles won by heroes and villains.

```
package io.quarkus.workshop.superheroes.statistics;

class TeamStats {

    private int villains = 0;
    private int heroes = 0;

    double add(Fight result) {
        if (result.winnerTeam.equalsIgnoreCase("heroes")) {
            heroes = heroes + 1;
        } else {
            villains = villains + 1;
        }
        return ((double) heroes / (heroes + villains));
    }

}
```

The `@RegisterForReflection` annotation instructs the native compilation to allow reflection access to the class. Without, the serialization/deserialization would not work when running the native executable.

Reading Messages from Kafka

It's now time to connect the `fights` channel with the Kafka topic.

[hand o right] Call to action

Edit the `application.properties` file and add the following content:

```
quarkus.http.port=8085

## Kafka configuration
mp.messaging.incoming.fights.connector=smallrye-kafka
mp.messaging.incoming.fights.value.deserializer=io.quarkus.workshop.superheroes.statis
tics.FightDeserializer
mp.messaging.incoming.fights.auto.offset.reset=earliest
mp.messaging.incoming.fights.broadcast=true
```

As for the writing side, it configures the Kafka connector. The `mp.messaging.incoming.fights.auto.offset.reset=earliest` property indicates that the topic is read from the earliest available record. Check the Kafka configuration to see all the available settings.

Sending Events on WebSockets

At this point, you read the *fights* from Kafka and computes statistics. Actually, even if you start the application, nothing will happen as nobody consumes these statistics.

In this section, we are going to consume these statistics and send them to two WebSockets. For this, we are going to add two classes and a simple presentation page:

- `TeamStatsWebSocket`
- `TopWinnerWebSocket`
- `index.html`

Quarkus uses the JSR 356 providing an annotation-driven approach to implement WebSockets.

The TeamStats WebSocket

[hand o right] Call to action

Create the `io.quarkus.workshop.superheroes.statistics.TeamStatsWebSocket` class with the following content:

```
package io.quarkus.workshop.superheroes.statistics;

import io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.subscription.Cancellable;
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.jboss.logging.Logger;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.websocket.OnClose;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

@ServerEndpoint("/stats/team")
@ApplicationScoped
public class TeamStatsWebSocket {

    @Inject
    @Channel("team-stats")
    Multi<Double> stream;

    private static final Logger LOGGER = Logger.getLogger(TeamStatsWebSocket.class);

    private final List<Session> sessions = new CopyOnWriteArrayList<>();
```

```
    private Cancellable cancellable;

    @OnOpen
    public void onOpen(Session session) {
        sessions.add(session);
    }

    @OnClose
    public void onClose(Session session) {
        sessions.remove(session);
    }

    @PostConstruct
    public void subscribe() {
        cancellable = stream.subscribe().with(ratio -> sessions.forEach(session ->
write(session, ratio)));
    }

    @PreDestroy
    public void cleanup() {
        cancellable.cancel();
    }

    private void write(Session session, double ratio) {
        session.getAsyncRemote().sendText(Double.toString(ratio), result -> {
            if (result.getException() != null) {
                LOGGER.error("Unable to write message to web socket", result
.getException());
            }
        });
    }
}
```

This component is a WebSocket as specified by the `@ServerEndpoint("/stats/team")` annotation. It handles the `/stats/team` WebSocket.

When a client (like a browser) connects to the WebSocket, it keeps track of the session. This session is released when the client disconnects.

The `TeamStatsWebSocket` also injects a `Multi` attached to the `team-stats` channel. After creation, the component subscribes to this stream and broadcasts the fights to the different clients connected to the web socket.

The *subscription* is an essential part of the stream lifecycle. It indicates that someone is interested in the items transiting on the stream, and it triggers the emission. In this case, it triggers the connection to Kafka and starts receiving the messages from Kafka. Without it, items would not be emitted.

The TopWinner WebSocket

The `io.quarkus.workshop.superheroes.statistics.TopWinnerWebSocket` follows the same pattern but subscribes to the `winner-stats` channel.

[hand o right] Call to action

Creates the `io.quarkus.workshop.superheroes.statistics.TopWinnerWebSocket` with the following content:

```
package io.quarkus.workshop.superheroes.statistics;

import io.smallrye.mutiny.Multi;
import io.smallrye.mutiny.subscription.Cancellable;
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.jboss.logging.Logger;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.json.bind.Jsonb;
import javax.json.bind.JsonbBuilder;
import javax.websocket.OnClose;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

@ServerEndpoint("/stats/winners")
@ApplicationScoped
public class TopWinnerWebSocket {

    private static final Logger LOGGER = Logger.getLogger(TopWinnerWebSocket.class);
    private Jsonb jsonb;

    @Inject @Channel("winner-stats")
    Multi<Iterable<Score>> winners;

    private final List<Session> sessions = new CopyOnWriteArrayList<>();
    private Cancellable cancellable;

    @OnOpen
    public void onOpen(Session session) {
        sessions.add(session);
    }

    @OnClose
    public void onClose(Session session) {
        sessions.remove(session);
```

```
    }

    @PostConstruct
    public void subscribe() {
        jsonb = JsonbBuilder.create();
        cancellable = winners
            .map(scores -> jsonb.toJson(scores))
            .subscribe().with(serialized -> sessions.forEach(session -> write(session,
serialized)));
    }

    @PreDestroy
    public void cleanup() throws Exception {
        cancellable.cancel();
        jsonb.close();
    }

    private void write(Session session, String serialized) {
        session.getAsyncRemote().sendText(serialized, result -> {
            if (result.getException() != null) {
                LOGGER.error("Unable to write message to web socket", result
.getException());
            }
        });
    }
}
```

Because the items (top 10) need to be serialized, the `TopWinnerWebSocket` also use JSONB to transform the object into a serialized form.

The UI

Finally, you need a UI to watch these live statistics.

[hand o right] Call to action

Replace the `META-INF/resources/index.html` file with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Super Battle Stats</title>

    <link rel="stylesheet" type="text/css" href=
"https://cdnjs.cloudflare.com/ajax/libs/patternfly/3.24.0/css/patternfly.min.css">
    <link rel="stylesheet" type="text/css" href=
"https://cdnjs.cloudflare.com/ajax/libs/patternfly/3.24.0/css/patternfly-
additions.min.css">
```

```
    <style>
        .page-title {
            font-size: xx-large;
        }

        .progress {
            background-color: firebrick;
        }

        .progress-bar {
            background: dodgerblue;
        }

    </style>
</head>
<body>
<div class="container">
<div class="row">
    <div class="col"><h1 class="page-title">Super Stats</h1></div>
</div>
</div>
<div class="container container-cards-pf">
    <div class="row row-cards-pf">
        <div class="col-xs-12 col-sm-6 col-md-4 col-lg-3">
            <!-- Top winners -->
            <div class="card-pf card-pf-view card-pf-view-select">
                <h2 class="card-pf-title">
                    <i class="fa fa-trophy"></i> Top Winner
                </h2>
                <div class="card-pf-body">
                    <div id="top-winner">

                    </div>
                </div>
            </div>
        </div>

        <div class="col-xs-12 col-sm-8 col-md-6 col-lg-6">
            <!-- Top losers -->
            <div class="card-pf card-pf-view card-pf-view-select">
                <h2 class="card-pf-title">
                    <i class="fa pficon-rebalance"></i> Heroes vs. Villains
                </h2>
                <div class="card-pf-body">
                    <div class="progress-container progress-description-left progress-
label-right">
                        <div class="progress-description">
                            Heroes
                        </div>
                        <div class="progress">
                            <div id="balance" class="progress-bar" role="progressbar"
```

```
aria-valuenow="50" aria-valuemin="0" aria-valuemax="100" style="width: 50%;">
                                <span>Villains</span>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js"
></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.3.7/js/bootstrap.min.js"></script>
<script src=
"https://cdnjs.cloudflare.com/ajax/libs/patternfly/3.24.0/js/patternfly.min.js"></scri
pt>
<script src="https://cdnjs.cloudflare.com/ajax/libs/c3/0.7.11/c3.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/5.12.0/d3.min.js"></script>

<script>
    $(document).ready(function() {
        var host = window.location.host;

        var top = new WebSocket("ws://" + host + "/stats/winners");
        top.onmessage = function (event) {
            updateTop(event.data);
        };
        var team = new WebSocket("ws://" + host + "/stats/team");
        team.onmessage = function(event) {
            console.log(event.data);
            updateRatio(event.data);
        };
    });


    function updateTop(scores) {
        $("#top-winner").children("p").remove();
        JSON.parse(scores).forEach(function(score) {
            $("#top-winner").append($("<p>" + score.name + " [" + score.score + "
]</p>"))
        });
    }

    function updateRatio(ratio) {
        var percent = ratio * 100;
        $("#balance").attr("aria-valuenow", ratio * 100).attr("style", "width: " +
percent + "%;");
    }
</script>
```

```
</body>
</html>
```

Running the Application

You are all set!

[hand o right] Call to action

Time to start the application using:

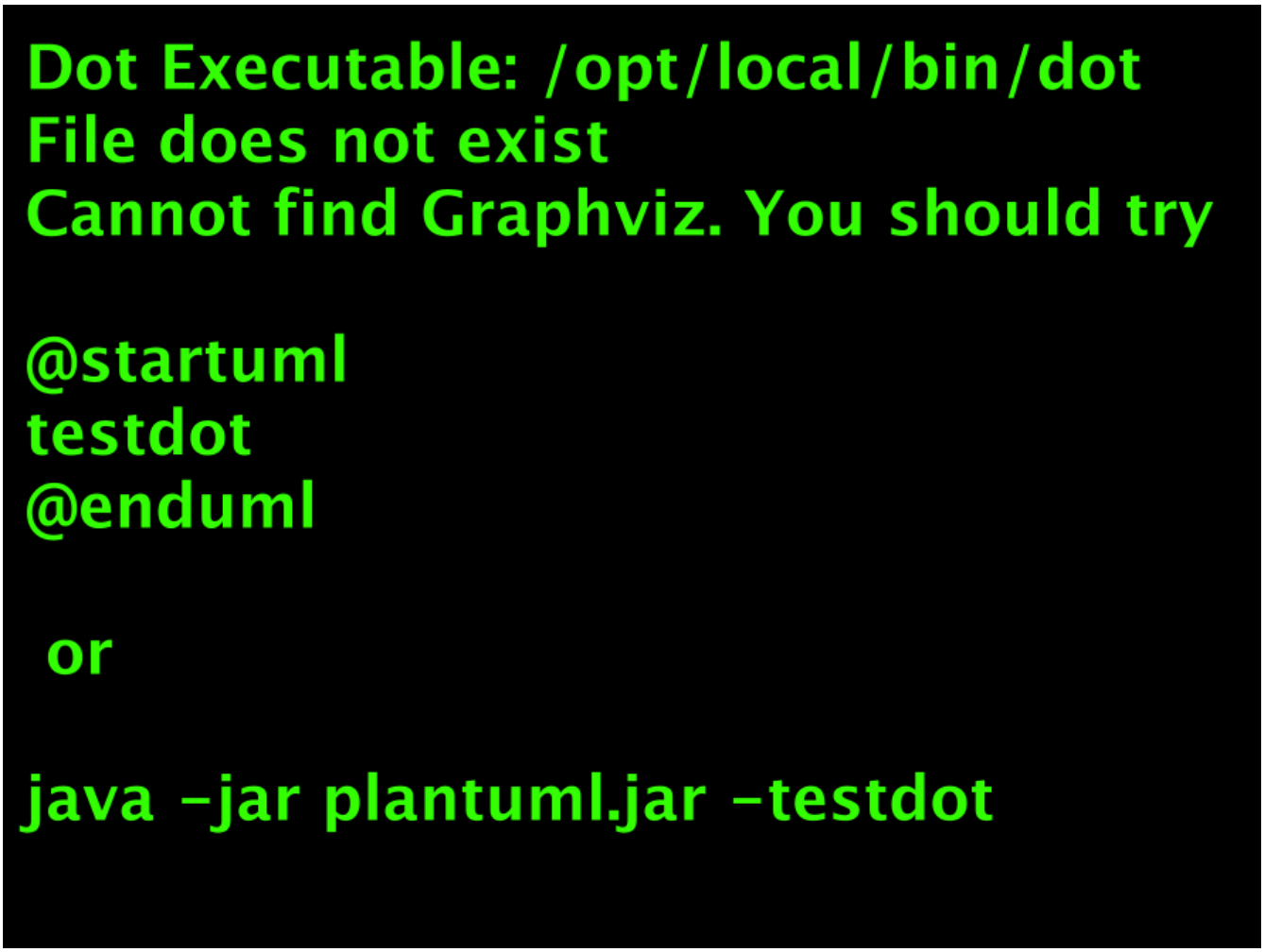
```
$ mvn compile quarkus:dev
```

Then, open `http://localhost:8085` in a new browser window. Trigger some fights, and you should see the live statistics moving.

If you have any problem with the code, don't understand or feel you are running, remember to ask for some help. Also, you can get the code of this entire workshop from https://github.com/quarkusio/quarkus-workshops/tree/master/quarkus-workshop-super-heroes.

Unifying Imperative and Reactive Programming

So, as seen in this chapter, Quarkus is not limited to HTTP microservices, but fits perfectly in an event-driven architecture. The secret behind this is to use a single reactive engine for both imperative and reactive code:



This unique architecture allows to mix imperative and reactive, but also use the right *model* for the job. To go further on this, we recommend:

- https://quarkus.io/guides/reactive-routes-guide to use reactive routes
- https://quarkus.io/guides/reactive-sql-clients to access SQL database in a non-blocking fashion
- https://quarkus.io/guides/kafka-guide to integrate with Kafka
- https://quarkus.io/guides/amqp-guide to integrate with AMQP 1.0
- https://quarkus.io/guides/using-vertx to understand how you can use Vert.x directly

[35] MicroProfile Reactive Messaging https://github.com/eclipse/microprofile-reactive-messaging

[36] Kafka Topic https://kafka.apache.org/intro#intro_topics

Writing a Quarkus Extension

Most of the Quarkus magic happens inside extensions. The goal of an extension is to compute *just enough bytecode* to start the services that the application requires, and drop everything else.

So, when writing an extension, you need to distinguish the action that:

- Can be done at build time
- Must be done at runtime

Because of this distinction, extensions are divided into 2 parts: a build time augmentation and a runtime. The augmentation part is responsible for all the metadata processing, annotation scanning, XML parsing… The output of this augmentation is **recorded bytecode**, which, then, is executed at runtime to instantiate the relevant services.

In this chapter, you are going to implement a *banner* extension. Instead of having to include the bean invoked during the application startup in the application code, you are going to write an extension that does this.

The extension framework

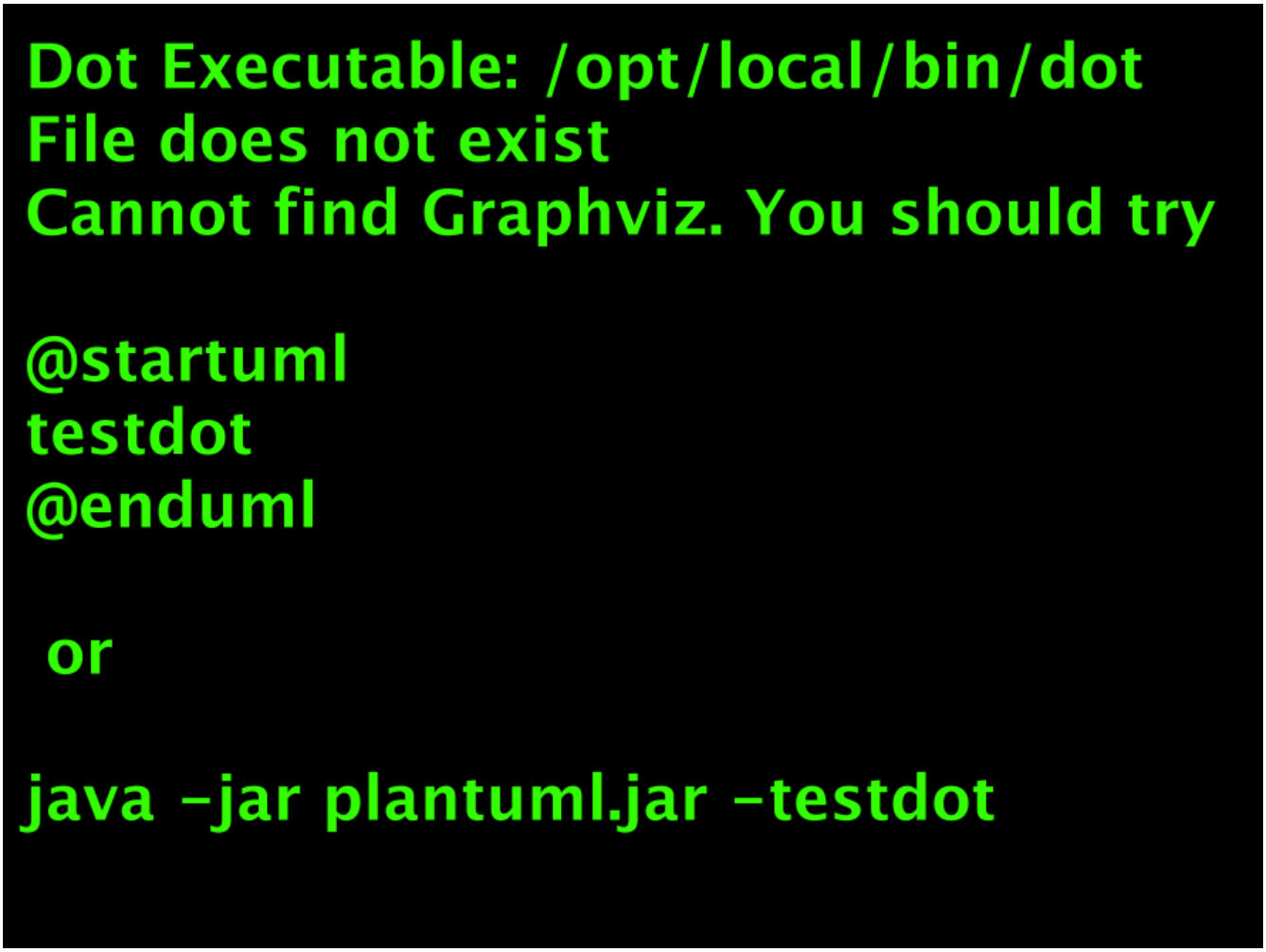
Quarkus's mission is to transform your entire application, including the libraries it uses, into an artifact that uses significantly fewer resources than traditional approaches. These can then be used to build native executables using GraalVM. To do this, you need to analyze and understand the full "closed world" of the application. Without the full context, the best that can be achieved is partial and limited generic support.

To build an extension, Quarkus provides a framework to:

- read configuration from the `application.properties` file and map it to objects,
- read metadata from classes without having to load them, this includes classpath and annotation scanning,
- generate bytecode if needed (for proxies for instance),
- pass sensible defaults to the application,
- make the application compatible with GraalVM (resources, reflection, substitutions),
- implement hot-reload

Structure of an extension

As stated above, an extension is divided into 2 parts, called `deployment` (augmentation) and `runtime`.



[hand o right] Call to action

From the directory `extensions/extension-banner` execute the following commands:

```
mkdir -p deployment/src/main/java
mkdir -p deployment/src/main/resources
mkdir -p runtime/src/main/java
mkdir -p runtime/src/main/resources

echo "<project xmlns='http://maven.apache.org/POM/4.0.0'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
         xsi:schemaLocation='http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd'>
    <modelVersion>4.0.0</modelVersion>

    <groupId>io.quarkus.workshop.super-heroes</groupId>
    <artifactId>extension-banner-parent</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
    <name>Quarkus Workshop :: Extensions :: Banner Extension</name>

    <modules>
        <module>runtime</module>
        <module>deployment</module>
```

```
    </modules>

     <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-bom</artifactId>
                <version>\${quarkus.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <properties>
        <quarkus.version>1.9.2.Final</quarkus.version>
        <surefire-plugin.version>2.22.0</surefire-plugin.version>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.build.timestamp.format>yyyy-MM-dd</maven.build.timestamp.format>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
" > pom.xml

echo "<project xmlns='http://maven.apache.org/POM/4.0.0'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
         xsi:schemaLocation='http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd'>
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>io.quarkus.workshop.super-heroes</groupId>
        <artifactId>extension-banner-parent</artifactId>
        <version>1.0</version>
    </parent>

    <artifactId>extension-banner-deployment</artifactId>
    <name>Quarkus Workshop :: Extensions :: Banner Extension :: Deployment</name>

    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-core-deployment</artifactId>
            <version>\${quarkus.version}</version>
        </dependency>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-arc-deployment</artifactId>
            <version>\${quarkus.version}</version>
```

```
        </dependency>
        <dependency>
            <groupId>io.quarkus.workshop.super-heroes</groupId>
            <artifactId>extension-banner</artifactId>
            <version>\${project.version}</version>
        </dependency>

        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-junit5-internal</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>io.quarkus</groupId>
                            <artifactId>quarkus-extension-processor</artifactId>
                            <version>\${quarkus.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M4</version>
                <configuration>
                    <systemProperties>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
                    </systemProperties>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>" > deployment/pom.xml

echo "<project xmlns='http://maven.apache.org/POM/4.0.0'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
         xsi:schemaLocation='http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd'>
    <modelVersion>4.0.0</modelVersion>

    <parent>
```

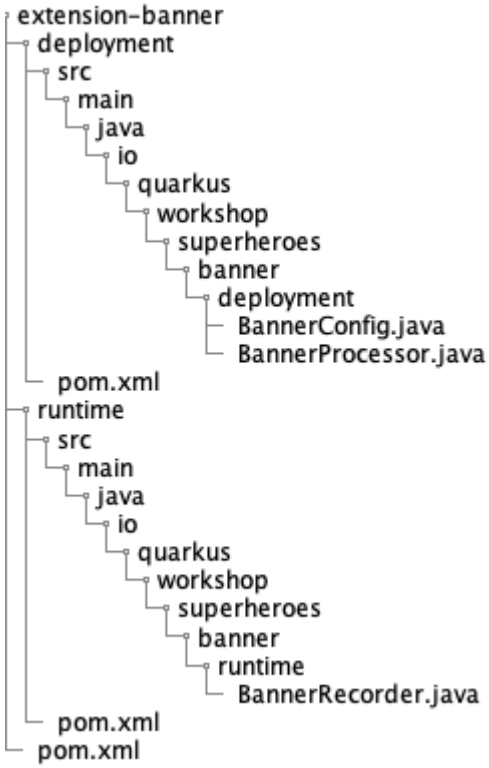
```
        <groupId>io.quarkus.workshop.super-heroes</groupId>
        <artifactId>extension-banner-parent</artifactId>
        <version>1.0</version>
    </parent>

    <artifactId>extension-banner</artifactId>
    <name>Quarkus Workshop :: Extensions :: Banner Extension :: Runtime</name>

    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-core</artifactId>
        </dependency>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-arc</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-bootstrap-maven-plugin</artifactId>
                <version>\${quarkus.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>extension-descriptor</goal>
                        </goals>
                        <phase>compile</phase>
                        <configuration>
                            <deployment>\${project.groupId}:\${project.artifactId}-
deployment:\${project.version}</deployment>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>io.quarkus</groupId>
                            <artifactId>quarkus-extension-processor</artifactId>
                            <version>\${quarkus.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
```

```
        </plugins>
    </build>
</project>" > runtime/pom.xml
```

This script creates the structure for the banner extension:

- one parent `pom.xml` importing the `quarkus-bom`
- a module for the runtime
- a module for the deployment, with a dependency on the runtime artifact

The final structure of the extension developed in this section is the following:



The banner extension

The goal of this chapter is to implement an extension that displays a textual *banner* when the application starts. For this, what do we need:

1. The banner itself, in a file - so some configuration,
2. Some code that would print the banner when the application starts; so some runtime code,
3. Some augmentation code (build steps) that receives the configuration reads; the content of the banner file and record the runtime invocations,
4. A way to monitor the content of the file and trigger a hot-reload in dev mode

The Runtime module

The *runtime* part of an extension contains only the classes and resources required at runtime. For

the banner extension, it would be a single class that prints the banner.

[hand o right] Call to action

In the runtime module, creates the `io.quarkus.workshop.superheroes.banner.runtime.BannerRecorder` class with the following content:

```
package io.quarkus.workshop.superheroes.banner.runtime;

import io.quarkus.runtime.annotations.Recorder;

@Recorder
public class BannerRecorder {

    public void print(String banner) {
        System.err.println(banner);
    }
}
```

Simple right? But how does it work? Look at the `@Recorder` annotation. It indicates that this class is a *recorder* that is used to record actions executed, later, at runtime. Indeed, these actions are replayed at runtime. We will see how this recorder is used from the *deployment* module.

The deployment module

This module contains *build steps*, i.e., methods called during the augmentation phase and computing just enough bytecode to serve the services the application requires. For the banner extension, it consists of two build steps:

1. The first build step is going to read the banner file and use the `BannerRecorder`
2. The second build step is related to the dev mode and triggers a hot-reload when the content of the banner file changes.

[hand o right] Call to action

In the deployment module, create the `io.quarkus.workshop.superheroes.banner.deployment.BannerConfig` with the following content:

```
package io.quarkus.workshop.superheroes.banner.deployment;

import io.quarkus.runtime.annotations.ConfigItem;
import io.quarkus.runtime.annotations.ConfigPhase;
import io.quarkus.runtime.annotations.ConfigRoot;

@ConfigRoot(name = "banner", phase = ConfigPhase.BUILD_TIME)
public class BannerConfig {

    /**
     * The path of the banner.
     */
    @ConfigItem public String file;
}
```

This class maps entries from the `application.properties` file to an object. It's a convenient mechanism to avoid having to use the low-level configuration API directly. The `ConfigRoot` annotation indicates that this class maps properties prefixed with `quarkus.banner`. The class declares a single property, `file`, which is the `quarkus.banner.file` user property. Instances of this class are created by Quarkus and are used in the second part of the deployment module: the processor.

[hand o right] Call to action

Create the `io.quarkus.workshop.superheroes.banner.deployment.BannerProcessor` class with the following content:

```
package io.quarkus.workshop.superheroes.banner.deployment;

import io.quarkus.deployment.annotations.BuildStep;
import io.quarkus.deployment.annotations.ExecutionTime;
import io.quarkus.deployment.annotations.Record;
import io.quarkus.deployment.builditem.HotDeploymentWatchedFileBuildItem;
import io.quarkus.deployment.util.FileUtil;
import io.quarkus.workshop.superheroes.banner.runtime.BannerRecorder;

import java.io.IOException;
import java.io.InputStream;
import java.io.UncheckedIOException;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

public class BannerProcessor {

    @BuildStep
    @Record(ExecutionTime.RUNTIME_INIT)
    public void recordBanner(BannerRecorder recorder, BannerConfig config) {
        String content = readBannerFile(config.file);
        recorder.print(content);
    }

    @BuildStep
    List<HotDeploymentWatchedFileBuildItem> watchBannerChanges(BannerConfig config) {
        List<HotDeploymentWatchedFileBuildItem> watchedFiles = new ArrayList<>();
        watchedFiles.add(new HotDeploymentWatchedFileBuildItem((config.file)));
        return watchedFiles;
    }

    private String readBannerFile(String path) {
        URL resource = Thread.currentThread().getContextClassLoader().getResource(
path);
        if (resource != null) {
            try (InputStream is = resource.openStream()) {
                byte[] content = FileUtil.readFileContents(is);
                return new String(content, StandardCharsets.UTF_8);
            } catch (IOException e) {
                throw new UncheckedIOException(e);
            }
        } else {
            throw new IllegalArgumentException("Cannot find the banner file: " + path
);
        }
    }

}
```

This class is the core of the extension. It contains a set of methods annotated with `@BuildStep`.

The `recordBanner` method is responsible for recording the actions that happen at runtime. In addition to the `@BuildStep` annotation, it also has the `@Record` annotation allowing to receive a *recorder* object (`BannerRecorder`) and indicating when the recorded bytecode is replayed. Here we are replaying during the runtime initialization, *i.e.* equivalent to the `public static void main(String… args)` method.

The method reads the content of the banner file. This file is located using the `path` property from the `BannerConfig` object. Once the content is retrieved, it calls the recorder with the content. This invocation is going to be replayed at runtime.

Recorder at deployment time

At deployment time, proxies of recorders are injected into `@BuildStep` methods that have been annotated with `@Record`. Any invocations made on these proxies will be recorded, and bytecode will be written out to be executed at runtime to make the same sequence of invocations with the same parameters on the actual recorder objects.

At this point, the extension is functional, but don't forget one of the pillars of Quarkus: the developer joy. The extension is also responsible for implementing hot reload logic. It is the role of the `watchBannerChanges` method, which indicates that the banner file must be watched, and the application restarted when this file changes.

Packaging the extension

[hand o right] Call to action

From the root directory of the extension, run:

```
mvn clean install
```

Using the extension

[hand o right] Call to action

Go back to the *fight* microservice, and add the following dependency to the `pom.xml` file:

```
<dependency>
    <groupId>io.quarkus.workshop.super-heroes</groupId>
    <artifactId>extension-banner</artifactId>
    <version>1.0</version>
</dependency>
```

Go to http://patorjk.com/software/taag/ to generate a banner for the fight microservice.

[hand o right] Call to action

Once you have the content, write it into `src/main/resources/banner.txt`. For instance:

```
          .--.            .
     _.._ |__|  .--./)  .'|
   .' .._|.--. /.''\\  <  |            .|
   | '    |  || |  | |  | |          .' |_
 __| |__  |  | \`-' /   | | .'''-.  .'     |
|__   __| |  | /("'`    | |/.'''. \'--.  .-'
   | |    |  | \ '---.  |  /    | |   |  |
   | |    |__|  /'""'.\ | |     | |   |  |
   | |         ||     ||| |     | |   |  '.'
   | |         \'. __// | '.    | '.  |   /
   |_|          `'---'  '---'   '---' `'-'
```

[hand o right] Call to action

Then, edit the `src/main/resources/application.properties` and add:

```
quarkus.banner.file=banner.txt
```

[hand o right] Call to action

Now, restart the microservice with:

```
mvn quarkus:dev
```

And the banner will be displayed. While keeping the dev mode running, edit the file, save and wait a few seconds. Once the change is detected, the application is restarted, and the banner updated.

[hand o right] Call to action

Let's now check the behavior in native mode. Compile the microservice with:

```
mvn package -Pnative
```

And then start the service with:

```
./target/rest-fight-01-runner
```

Conclusion

In this section you have seen how to develop a simple extension for Quarkus. Quarkus offers a complete toolbox to implement extensions, from configuration support, tests, bytecode generation…

The mindset to implement an extension is crucial. The distinction between build time and runtime is what makes Quarkus so efficient. To go further, check https://quarkus.io/guides/extension-authors-guide.

Containers & Cloud (optional)

This chapter explores how you can deploy Quarkus applications in containers and Cloud platforms. There are many different approaches to achieve these deployments. In this chapter, we are focusing on the creation of containers using Quarkus native executables and the deployment of our system in Kubernetes/OpenShift.

From bare metal to containers

In this section, we are going to package our microservices into containers. In particular, we are going to produce Linux 64 bits native executables and runs them in a container. The native compilation uses the OS and architecture of the host system.

And… Linux Containers are … *Linux*. So before being able to build a container with our native executable, we need to produce compatible native executables. If you are using a Linux 64 bits machine, you are good to go. If not, Quarkus comes with a trick to produce these executable:

```
$ mvn clean package -Pnative -Dnative-image.docker-build=true -DskipTests
```

The `-Dnative-image.docker-build=true` allows running the native compilation inside a container (provided by Quarkus). The result is a Linux 64 bits executable.

Building a native executable takes time, CPU, and memory. It's even more accurate in the container. So, first, be sure that your container system has enough memory to build the executable. It requires at least 6Gb of memory, 8Gb is recommended.

[hand o right] Call to action

Execute the above command for all our microservices. We also copy the UI into the fight service, to simplify the process:

```
cd rest-hero
mvn clean package -Pnative -Dnative-image.docker-build=true -DskipTests
cd ..
cd rest-villain
mvn clean package -Pnative -Dnative-image.docker-build=true -DskipTests
cd ..
cd rest-fight
cp -R ../ui-super-heroes/dist/* src/main/resources/META-INF/resources
mvn clean package -Pnative -Dnative-image.docker-build=true -DskipTests
cd ..
cd event-statistics
mvn clean package -Pnative -Dnative-image.docker-build=true -DskipTests
cd ..
```

Building containers

Now that we have the native executables, we can build containers. When you create projects, Quarkus generates two `Dockerfiles`:

1. `Dockerfile.jvm` - A `Dockerfile` for running the application in JVM mode
2. `Dockerfile.native` - A `Dockerfile` for running the application in native mode

We are interested in this second file. Open one of these `Dockerfile.native` files:

```
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"]
```

It's a pretty straightforward `Dockerfile` taking a minimal base image and copying the generated native executable. It also exposes the port 8080. Wait, our microservices are not configured to run on the port 8080. We need to override this property as well as a few other such as the HTTP client endpoints, and database locations.

To build the containers, use the following scripts:

```
export ORG=xxxx
cd rest-hero
docker build -f src/main/docker/Dockerfile.native -t $ORG/quarkus-workshop-hero .
cd ..
cd rest-villain
docker build -f src/main/docker/Dockerfile.native -t $ORG/quarkus-workshop-villain .
cd ..
cd rest-fight
docker build -f src/main/docker/Dockerfile.native -t $ORG/quarkus-workshop-fight .
cd ..
cd event-statistics
docker build -f src/main/docker/Dockerfile.native -t $ORG/quarkus-workshop-stats .
cd ..
```

Replace `ORG` with your DockerHub / Quay.io username.

Deploying on Kubernetes

This section is going to deploy our microservices on Kubernetes. It is required to have access to a Kubernetes or OpenShift cluster.

To deploy your microservices, push the built container images to an image registry accessible by your cluster, such as Quay.io or DockerHub.

We recommend using a specific namespace to deploy your system. In the following sections, we use the `quarkus-workshop` namespace.

Deploying the infrastructure

The first thing to deploy is the required infrastructure:

- 3 PostgreSQL instances
- Kafka brokers (3 brokers with 3 Zookeeper to follow the recommended approach)

There are many ways to deploy this infrastructure. Here, we are going to use two operators:

- PostgreSQL Operator by Dev4Ddevs.com
- Strimzi Apache Kafka Operator by Red Hat

[hand o right] Call to action

With these operators installed, you can create the required infrastructure with the following custom resource definition (CRD):

```
apiVersion: postgresql.dev4devs.com/v1alpha1
kind: Database
metadata:
    name: heroes-database
    namespace: quarkus-workshop
spec:
    databaseCpu: 30m
    databaseCpuLimit: 60m
    databaseMemoryLimit: 512Mi
    databaseMemoryRequest: 128Mi
    databaseName: heroes-database
    databaseNameKeyEnvVar: POSTGRESQL_DATABASE
    databasePassword: superman
    databasePasswordKeyEnvVar: POSTGRESQL_PASSWORD
    databaseStorageRequest: 1Gi
    databaseUser: superman
    databaseUserKeyEnvVar: POSTGRESQL_USER
    image: centos/postgresql-96-centos7
    size: 1
```

[hand o right] Call to action

This CRD creates the database for the Hero microservice. Duplicate this CRD for the fight and villain databases.

For the Kafka broker, create the following CRD:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-kafka
  namespace: quarkus-workshop
spec:
  kafka:
    version: 2.3.0
    replicas: 3
    listeners:
      plain: {}
      tls: {}
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: '2.3'
    storage:
      type: ephemeral
  zookeeper:
    replicas: 3
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

This CRD creates the brokers and the Zookeeper instances.

It's also recommended to create the topic.

[hand o right] Call to action

For this, create the following CRD:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: fights
  labels:
    strimzi.io/cluster: my-kafka
  namespace: quarkus-workshop
spec:
  partitions: 1
  replicas: 3
  config:
    retention.ms: 604800000
    segment.bytes: 1073741824
```

Once everything is created, you should have the following resources:

```
$ kubectl get database
NAME                AGE
fights-database     16h
heroes-database     16h
villains-database   16h

$ kubectl get kafka
NAME       DESIRED KAFKA REPLICAS   DESIRED ZK REPLICAS
my-kafka   3
```

Deploying the Hero & Villain microservices

Now that the infrastructure is in place, we can deploy our microservices. Let's start with the hero and villain microservices.

For each, we need to override the port and data source URL.

[hand o right] Call to action

Create a config map with the following content:

Listing 2. config-hero.yaml

```
apiVersion: v1
data:
    port: "8080"
    database: "jdbc:postgresql://heroes-database:5432/heroes-database"
kind: ConfigMap
metadata:
    name: hero-config
```

[hand o right] Call to action

Do the same for the villain microservice. Then, apply these resources:

```
$ kubectl apply -f config-hero.yaml
$ kubectl apply -f config-villain.yaml
```

Once the config maps are created, we can deploy the microservices.

[hand o right] Call to action

Create a `deployment-hero.yaml` file with the following content:

```
---
apiVersion: "v1"
```

```
kind: "List"
items:
    - apiVersion: "v1"
      kind: "Service"
      metadata:
          labels:
              app: "quarkus-workshop-hero"
              version: "01"
              group: "$ORG"
          name: "quarkus-workshop-hero"
      spec:
          ports:
              - name: "http"
                port: 8080
                targetPort: 8080
          selector:
              app: "quarkus-workshop-hero"
              version: "01"
              group: "$ORG"
          type: "ClusterIP"
    - apiVersion: "apps/v1"
      kind: "Deployment"
      metadata:
          labels:
              app: "quarkus-workshop-hero"
              version: "01"
              group: "$ORG"
          name: "quarkus-workshop-hero"
      spec:
          replicas: 1
          selector:
              matchLabels:
                  app: "quarkus-workshop-hero"
                  version: "01"
                  group: "$ORG"
          template:
              metadata:
                  labels:
                      app: "quarkus-workshop-hero"
                      version: "01"
                      group: "$ORG"
              spec:
                  containers:
                      - image: "$ORG/quarkus-workshop-hero:latest"
                        imagePullPolicy: "IfNotPresent"
                        name: "quarkus-workshop-hero"
                        ports:
                            - containerPort: 8080
                              name: "http"
                              protocol: "TCP"
                        env:
```

```
                        - name: "KUBERNETES_NAMESPACE"
                          valueFrom:
                              fieldRef:
                                  fieldPath: "metadata.namespace"

                        - name: QUARKUS_DATASOURCE_URL
                          valueFrom:
                              configMapKeyRef:
                                  name: hero-config
                                  key: database

                        - name: QUARKUS_HTTP_PORT
                          valueFrom:
                              configMapKeyRef:
                                  name: hero-config
                                  key: port
```

This descriptor declares:

1. A service to expose the HTTP endpoint
2. A deployment that instantiates the application

The deployment declares one container using the container image we built earlier. It also overrides the configuration for the HTTP port and database URL.

[hand o right] Call to action

Don't forget to create the equivalent files for the villain microservice.

Then, deploy the microservice with:

```
$ kubectl apply -f deployment-hero.yaml
$ kubectl apply -f deployment-villain.yaml
```

Deploying the Fight microservice

Follow the same approach for the fight microservice. Note that there are more properties to configure from the config map:

- the location of the hero and villain microservice
- the location of the Kafka broker.

Once everything is configured and deployed, your system is now running on Kubernetes.

Conclusion

References

- https://github.com/cescoffier/quarkus-todo-app
- https://github.com/agoncal/baking-microservice-pie
- https://forge.jboss.org/document/hands-on-lab
- https://bit.ly/forge-hol
- https://code.quarkus.io
- https://quarkus.io/guides/all-config