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 using Quarkus.

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
- 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.

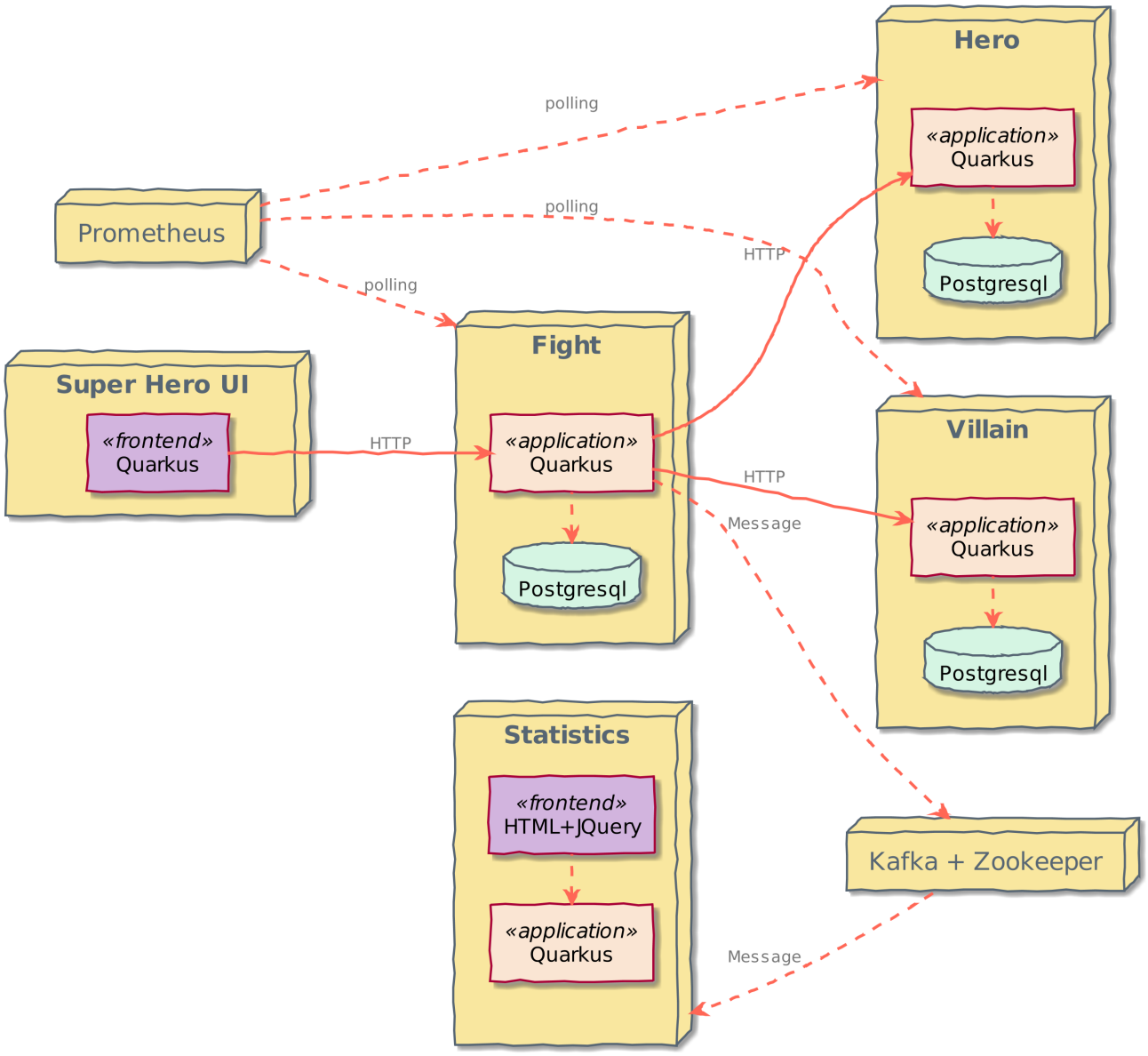
The idea is that you leave this workshop with a first experience using Quarkus, what it is not, and how it can help you in your projects. Then, you'll be prepared to investigate a bit more and, hopefully, contribute.

NOTE	Get this workshop from https://github.com/Red-Hat-Developer-Games/quarkus-workshop/tree/main/quarkus-workshop-super-heroes

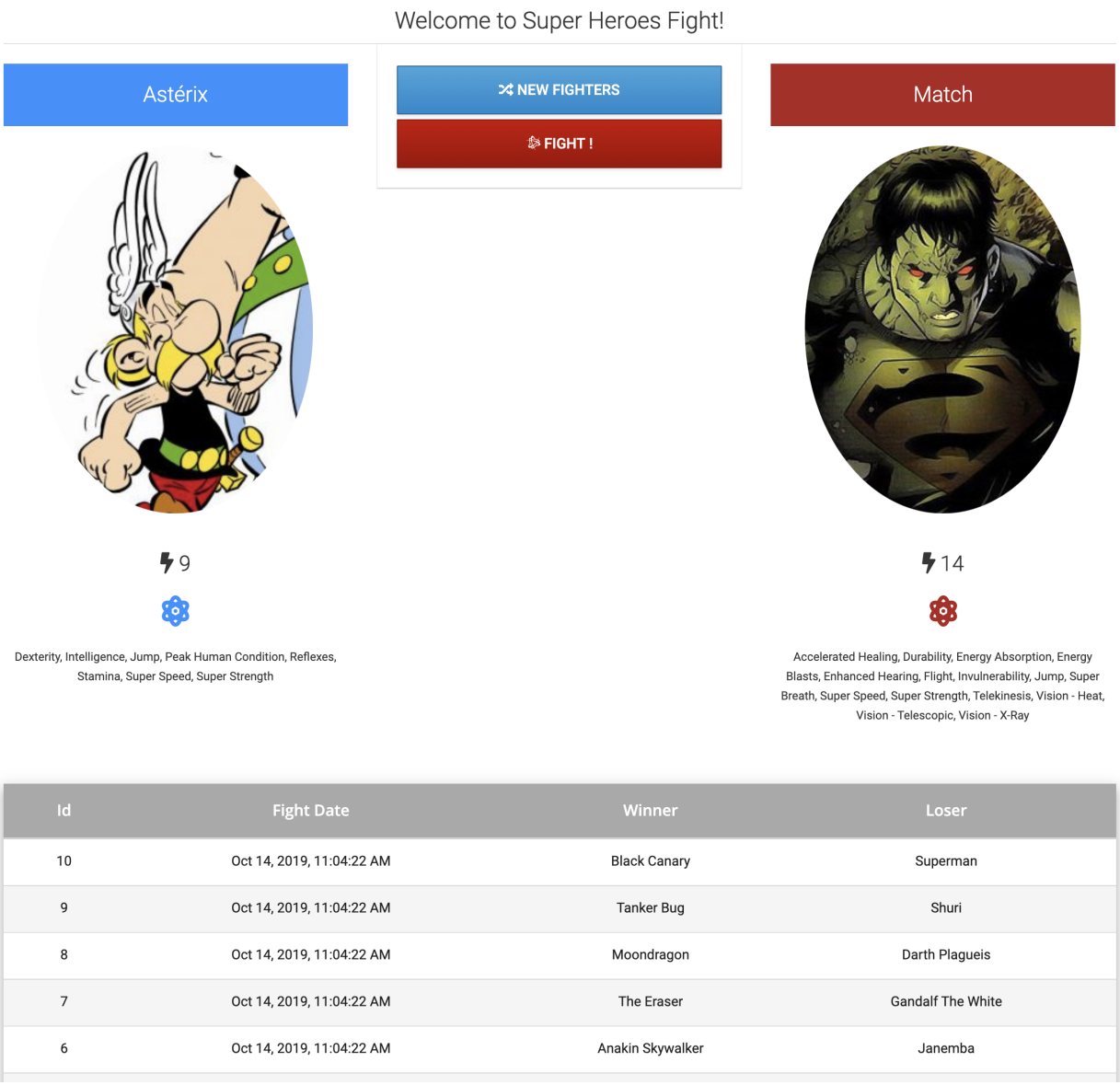
What Will You Be Developing?

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 :

- *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



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.



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

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

If you already have the tools installed, skip the *Installing all the needed tools* section and jump to the sections *Developing with 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 Web browser installed on your laptop and internet connectivity. You will also need a GitHub account.

Your environment is remote and can be accessed via CodeReady Workspaces (CRW) through your local browser, you just need to sign up and configure some elements. Your environment includes also Red Hat's OpenShift Container Platform (OCP).

The next section focuses on how to install and setup the needed software.

Installing Software

CodeReady Workspace

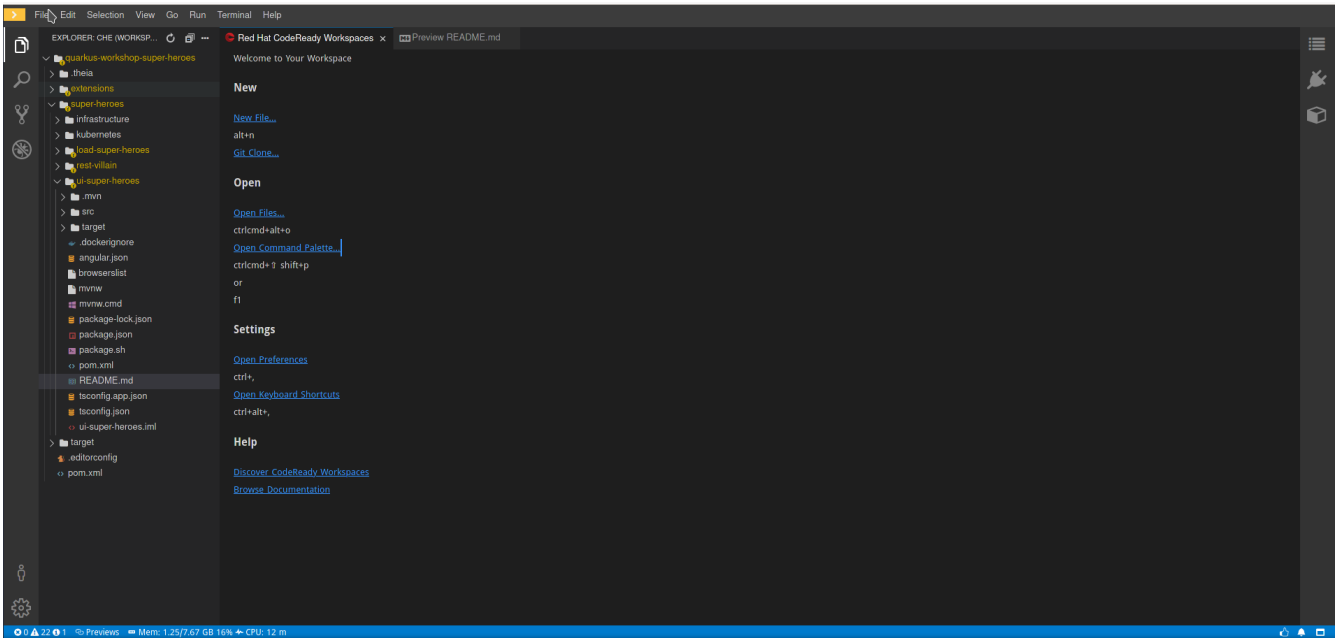
CodeReady Workspaces is a collaborative Kubernetes-native development solution that delivers OpenShift workspaces and in-browser IDE for rapid cloud application development. You are going to create your own environment.

CRW creation

- Go to the Etherpad site and choose an user. This user will be used to access the CRW and the OpenShift Web Console and for naming some components that you are going to create during the workshop.
- Launch the CRW creation by clicking the link mentioned in the Etherpad site.
- Once the CRW creation done, access to your CRW and sign up with your own user (selected previously in Etherpad) and full fill the form:

```
user: USERNAME
pwd: openshift
email: USERNAME@ocp.com
first name: Yago
last name: Sanchez
```

If everything goes well, you should have a CodeReady Workspace with a `quarkus-workshop` folder ready to start to code:



- Finally, open a terminal from the Terminal menu → Open Terminal in specific container → maven.

Command Line Utilities

Just make sure the following commands work on your CRW terminal

```
$ java -version
$ mvn -version
$ curl --version
```

OpenShift Container Platform

Your lab environment includes Red Hat's OpenShift Container Platform (OCP).

Access to your OCP resources can be gained via both the `oc` CLI utility and the OCP web console.

OpenShift Namespaces

The project we are going to develop will contain 3 microservices accessing to a PostgreSQL database. We will make the databases to run in a OpenShift dedicated namespace.

1. In the terminal of your CRW, authenticate into OpenShift as a non cluster admin user (USERNAME) using the `oc` utility.

NOTE	You can get the command for authenticating from the OpenShift Web Console.

```
$ oc login
```

There are 2 namespaces (OpenShift projects) in your OpenShift cluster: The namespace for hosting your CRW environment is USERNAME-codeready where `USERNAME` correspond to your specific username. The namespace for hosting databases and microservices is USERNAME-heroes.

NOTE	change the USERNAME with your own.

Operators

Your lab environment comes pre-installed with an PostgreSQL operator.

PostgreSQL operator

The PostgreSQL operator allows you to package, install, configure and manage a PostgreSQL database within an OpenShift cluster.

Congratulations! Your lab environment is now ready to use.

Preparing for the Workshop

Workshop scaffolding

[hand o right] Call to action

The workshop folder containing the code is already in your CRW, at the root of the filesystem.

In this workshop you will be developing an application dealing with Super Heroes (and Super Villains). The code will be in the super-heroes directory:

```
quarkus-workshop-super-heroes
└─ 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...)
   ├─ 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
```

For next steps, you need to be authenticated on OpenShift with your own user

```
$ oc login ....
$ oc project USERNAME-heroes
```

change USERNAME with your own.

Databases

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. The infrastructure folder contains the OpenShift manifests to

create the OpenShift resources in order to have this technical service available. We need to create a database for each microservice: heroes, villains and fights. The OpenShift manifests describing them can be found under `$CHE_PROJECTS_ROOT/quarkus-workshop-super-heroes/super-heroes/infrastructure/` directory.

We can deploy the DBs easily using `oc` utility and the manifests from the CRW terminal. Make sure that you are authenticated with your USERNAME and you are using the `USERNAME-heroes` namespace, then `apply` the databases yaml file:

```
oc apply -f $CHE_PROJECTS_ROOT/quarkus-workshop-super-heroes/super-
heroes/infrastructure/databases.yaml
```

You can verify the availability of databases with the following command:

```
curl http://heroes-database.USERNAME-heroes:5432
```

You should have the following response:

```
curl: (52) Empty reply from server
```

Warming up Maven

Now that you have the initial structure in place, we just need configure the correct url for the database.

On your CRW, edit the `$CHE_PROJECTS_ROOT/quarkus-workshop-super-heroes/super-heroes/rest-villain/src/main/resources/application.properties` file and change database configuration. You need to change DB's URL:

```
quarkus.datasource.jdbc.url=jdbc:postgresql://villains-database.USERNAME-
heroes:5432/villains_database
```

Change USERNAME with your own.

Now, in the Terminal, you can navigate to the root directory `quarkus-workshop-super-heroes/` and run:

[hand o right] Call to action

```
mvn clean install
```

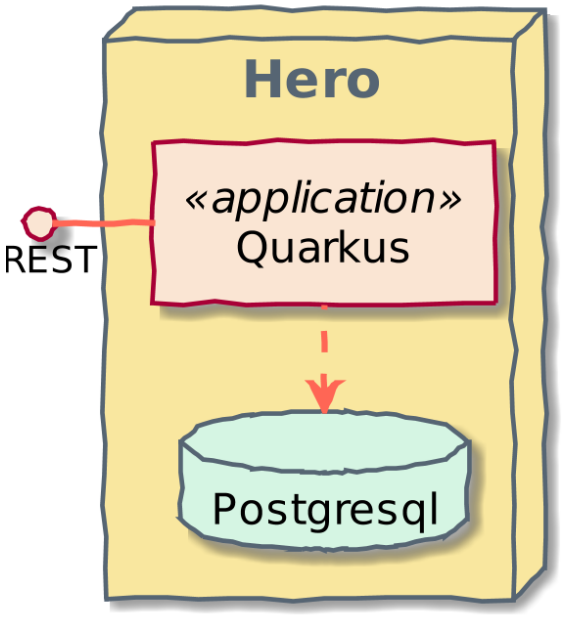
By running this command, it downloads all the required dependencies.

Ready?

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

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

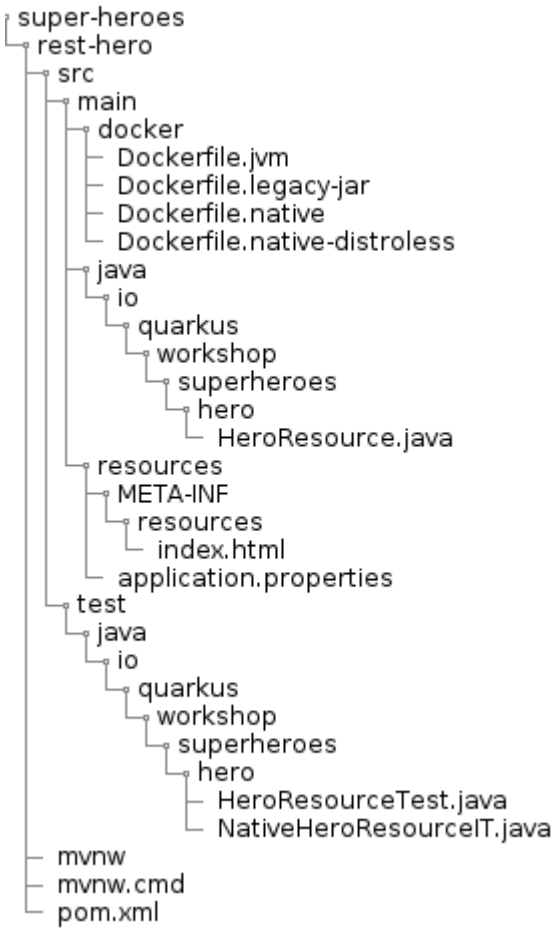
The easiest way to create a new Quarkus project is to use the Quarkus Maven plugin. We have created the project structure earlier, so we will move to the `super-heroes` directory and run the project creation command. Open a terminal and run the following command:

[hand o right] Call to action

```
cd $CHE_PROJECTS_ROOT/quarkus-workshop-super-heroes/super-heroes
mvn io.quarkus:quarkus-maven-plugin:2.2.2.Final:create \
    -DprojectGroupId=io.quarkus.workshop.super-heroes \
    -DprojectArtifactId=rest-hero \
    -DclassName="io.quarkus.workshop.superheroes.hero.HeroResource" \
    -Dpath="api/heroes"
```

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 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.

```
<properties>
    <compiler-plugin.version>3.8.1</compiler-plugin.version>
    <maven.compiler.parameters>true</maven.compiler.parameters>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <quarkus.platform.artifact-id>quarkus-bom</quarkus.platform.artifact-id>
    <quarkus.platform.group-id>io.quarkus.platform</quarkus.platform.group-id>
    <quarkus.platform.version>2.2.2.Final</quarkus.platform.version>
    <surefire-plugin.version>3.0.0-M5</surefire-plugin.version>
</properties>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>${quarkus.platform.group-id}</groupId>
            <artifactId>${quarkus.platform.artifact-id}</artifactId>
            <version>${quarkus.platform.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>${quarkus.platform.group-id}</groupId>
            <artifactId>quarkus-maven-plugin</artifactId>
            <version>${quarkus.platform.version}</version>
            <extensions>true</extensions>
            <executions>
                <execution>
                    <goals>
                        <goal>build</goal>
                        <goal>generate-code</goal>
```

```
                        <goal>generate-code-tests</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${compiler-plugin.version}</version>
            <configuration>
                <parameters>${maven.compiler.parameters}</parameters>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${surefire-plugin.version}</version>
            <configuration>
                <systemPropertyVariables>
                    <java.util.logging.manager>
org.jboss.logmanager.LogManager</java.util.logging.manager>
                    <maven.home>${maven.home}</maven.home>
                </systemPropertyVariables>
            </configuration>
        </plugin>
    </plugins>
</build>
```

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

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

RESTEasy

You may not be familiar with RESTEasy.[1] It's an implementation of JAX-RS and it uses to implement RestFul services in Quarkus.

What's this Quarkus bom 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`:

IMPORTANT copy the url mentioned in the CodeReady Workspace pop-up. The url should contain the port 8080 as we can see in the previous log. This url corresponds to the rest-hero microservice.

```
$ ./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:2.2.2.Final:dev (default-cli) @ rest-hero ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO] Nothing to compile - all classes are up to date
Listening for transport dt_socket at address: 5005
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2020-11-16 10:01:51,331 INFO  [io.quarkus] (Quarkus Main Thread) rest-hero 1.0-
SNAPSHOT on JVM (powered by Quarkus 2.2.2.Final) started in 3.797s. Listening on:
http://0.0.0.0:8080
2020-11-16 10:01:51,343 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev
activated. Live Coding activated.
2020-11-16 10:01:51,343 INFO  [io.quarkus] (Quarkus Main Thread) Installed features:
[cdi, resteasy]
```

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

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

```
$ curl $URL/api/heroes
hello RESTEasy
```

Alternatively, you can open $URL/api/heroes in your browser.

Change the url by the one you should have got from CRW

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://$(oc get route -n USERNAME-codeready | grep 8080 | awk '{ print $2
}')/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:

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

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:

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${surefire-plugin.version}</version>
    <configuration>
        <systemPropertyVariables>
            <java.util.logging.manager>
org.jboss.logmanager.LogManager</java.util.logging.manager>
            <maven.home>${maven.home}</maven.home>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

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.[2]

[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 the `quarkus-run.jar` file in the `target/quarkus-app/` directory. Be aware that it's not an *über-jar* as the dependencies are copied into the `target/quarkus-app/lib/` directory.

You can run the application using: `java -jar target/quarkus-app/quarkus-run.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

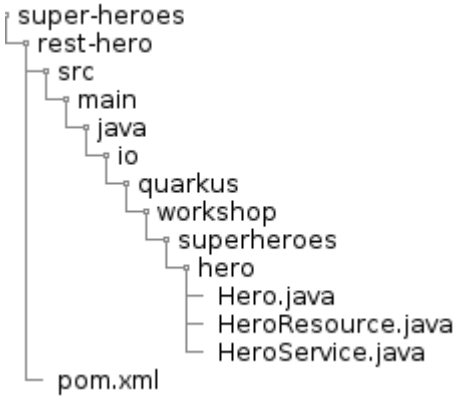
```
127.0.0.1 localhost my-node.local
```

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.[3]

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:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-orm-panache</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jdbc-postgresql</artifactId>
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-validator</artifactId>
</dependency>
```

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

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:

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

import io.quarkus.hibernate.orm.panache.PanacheEntity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import java.util.Random;
@Entity
public class Hero extends PanacheEntity {

    @NotNull
    @Size(min = 3, max = 50)
    public String name;
    public String otherName;
    @NotNull
    @Min(1)
    public int level;
    public String picture;

    @Column(columnDefinition = "TEXT")
    public String powers;


    @Override
    public String toString() {
        return "Hero{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", otherName='" + otherName + '\'' +
            ", level=" + level +
            ", picture='" + picture + '\'' +
            ", powers='" + powers + '\'' +
            '}';
    }
}
```

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:

```
public static Hero findRandom() {
    long countHeroes = Hero.count();
    Random random = new Random();
    int randomHero = random.nextInt((int) countHeroes);
    return Hero.findAll().page(randomHero, 1).firstResult();
}
```

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`):

```
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.sql=true
```

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:

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

import javax.enterprise.context.ApplicationScoped;
import javax.transaction.Transactional;
import javax.validation.Valid;
import java.util.List;

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

@ApplicationScoped
@Transactional(REQUIRED)
public class HeroService {


    @Transactional(SUPPORTS)
```

```
    public List<Hero> findAllHeroes() {
        return Hero.listAll();
    }

    @Transactional(SUPPORTS)
    public Hero findHeroById(Long id) {
        return Hero.findById(id);
    }

    @Transactional(SUPPORTS)
    public Hero findRandomHero() {
        Hero randomHero = null;
        while (randomHero == null) {
            randomHero = Hero.findRandom();
        }
        return randomHero;
    }

    public Hero persistHero(@Valid Hero hero) {
        Hero.persist(hero);
        return hero;
    }

    public Hero updateHero(@Valid Hero hero) {
        Hero entity = Hero.findById(hero.id);
        entity.name = hero.name;
        entity.otherName = hero.otherName;
        entity.level = hero.level;
        entity.picture = hero.picture;
        entity.powers = hero.powers;
        return entity;
    }

    public void deleteHero(Long id) {
        Hero hero = Hero.findById(id);
        hero.delete();
    }
}
```

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.[4]

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

[hand o right] Call to action

Just add the following datasource configuration:

```
quarkus.datasource.jdbc.url=jdbc:postgresql://heroes-database.USERNAME-
heroes:5432/heroes_database
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=superman
quarkus.datasource.password=superman
quarkus.datasource.max-size=8
quarkus.datasource.min-size=2
```

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:

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

import org.jboss.logging.Logger;

import javax.inject.Inject;
import javax.validation.Valid;
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import java.net.URI;
import java.util.List;

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

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

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

    @Inject
    HeroService service;

    @GET
    @Path("/random")
    public Response getRandomHero() {
        Hero hero = service.findRandomHero();
        LOGGER.debug("Found random hero " + hero);
```

```
        return Response.ok(hero).build();
    }

    @GET
    public Response getAllHeroes() {
        List<Hero> heroes = service.findAllHeroes();
        LOGGER.debug("Total number of heroes " + heroes);
        return Response.ok(heroes).build();
    }

    @GET
    @Path("/{id}")
    public Response getHero(
        @PathParam("id") Long id) {
        Hero hero = service.findHeroById(id);
        if (hero != null) {
            LOGGER.debug("Found hero " + hero);
            return Response.ok(hero).build();
        } else {
            LOGGER.debug("No hero found with id " + id);
            return Response.noContent().build();
        }
    }

    @POST
    public Response createHero(
        @Valid Hero hero, @Context UriInfo uriInfo) {
        hero = service.persistHero(hero);
        UriBuilder builder = uriInfo.getAbsolutePathBuilder().path(Long.toString(hero
.id));
        LOGGER.debug("New hero created with URI " + builder.build().toString());
        return Response.created(builder.build()).build();
    }

    @PUT
    public Response updateHero(
        @Valid Hero hero) {
        hero = service.updateHero(hero);
        LOGGER.debug("Hero updated with new valued " + hero);
        return Response.ok(hero).build();
    }

    @DELETE
    @Path("/{id}")
    public Response deleteHero(
        @PathParam("id") Long id) {
        service.deleteHero(id);
        LOGGER.debug("Hero deleted with " + id);
        return Response.noContent().build();
    }
```

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

Dependency Injection

Dependency injection in Quarkus is based on ArC which is a CDI-based dependency injection solution tailored for Quarkus' architecture.[5] You can learn more about it in the Contexts and Dependency Injection guide.[6]

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
```

Consider copying the url mentioned by CRW

Then, open your browser to $URL/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 but in our context, because of CRW, we will be using the database configured and running on OpenShift.

[hand o right] Call to action

We need to install in our `pom.xml` an extra dependency for data-binding functionality:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <scope>test</scope>
</dependency>
```

[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:

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

import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;
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 javax.ws.rs.core.HttpHeaders;
import javax.ws.rs.core.MediaType;
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.is;
import static org.junit.jupiter.api.Assertions.*;

@QuarkusTest
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class HeroResourceTest {

    private static final String DEFAULT_NAME = "Super Baguette";
    private static final String UPDATED_NAME = "Super Baguette (updated)";
    private static final String DEFAULT_OTHER_NAME = "Super Baguette Tradition";
    private static final String UPDATED_OTHER_NAME = "Super Baguette Tradition
(updated)";
    private static final String DEFAULT_PICTURE = "super_baguette.png";
    private static final String UPDATED_PICTURE = "super_baguette_updated.png";
    private static final String DEFAULT_POWERS = "eats baguette really quickly";
    private static final String UPDATED_POWERS = "eats baguette really quickly
(updated)";
    private static final int DEFAULT_LEVEL = 42;
    private static final int UPDATED_LEVEL = 43;

    private static final int NB_HEROES = 951;
    private static String heroId;




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

    @Test
    void shouldNotGetUnknownHero() {
        Long randomId = new Random().nextLong();
        given()
```

```
            .pathParam("id", randomId)
            .when().get("/api/heroes/{id}")
            .then()
            .statusCode(NO_CONTENT.getStatusCode());
    }

    @Test
    void shouldGetRandomHero() {
        given()
            .when().get("/api/heroes/random")
            .then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON);
    }

    @Test
    void shouldNotAddInvalidItem() {
        Hero hero = new Hero();
        hero.name = null;
        hero.otherName = DEFAULT_OTHER_NAME;
        hero.picture = DEFAULT_PICTURE;
        hero.powers = DEFAULT_POWERS;
        hero.level = 0;

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

    @Test
    @Order(1)
    void shouldGetInitialItems() {
        List<Hero> heroes = get("/api/heroes").then()
            .statusCode(OK.getStatusCode())
            .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
            .extract().body().as(getHeroTypeRef());
        assertEquals(NB_HEROES, heroes.size());
    }

    @Test
    @Order(2)
    void shouldAddAnItem() {
        Hero hero = new Hero();
        hero.name = DEFAULT_NAME;
        hero.otherName = DEFAULT_OTHER_NAME;
        hero.picture = DEFAULT_PICTURE;
```

```
        hero.powers = DEFAULT_POWERS;
        hero.level = DEFAULT_LEVEL;

        String location = given()
            .body(hero)
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .header(ACCEPT, APPLICATION_JSON)
            .when()
            .post("/api/heroes")
            .then()
            .statusCode(CREATED.getStatusCode())
            .extract().header("Location");
        assertTrue(location.contains("/api/heroes"));

        // Stores the id
        String[] segments = location.split("/");
        heroId = segments[segments.length - 1];
        assertNotNull(heroId);

        given()
            .pathParam("id", heroId)
            .when().get("/api/heroes/{id}")
            .then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .body("name", Is.is(DEFAULT_NAME))
            .body("otherName", Is.is(DEFAULT_OTHER_NAME))
            .body("level", Is.is(DEFAULT_LEVEL))
            .body("picture", Is.is(DEFAULT_PICTURE))
            .body("powers", Is.is(DEFAULT_POWERS));

        List<Hero> heroes = get("/api/heroes").then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .extract().body().as(getHeroTypeRef());
        assertEquals(NB_HEROES + 1, heroes.size());
    }

    @Test
    @Order(3)
    void shouldUpdateAnItem() {
        Hero hero = new Hero();
        hero.id = Long.valueOf(heroId);
        hero.name = UPDATED_NAME;
        hero.otherName = UPDATED_OTHER_NAME;
        hero.picture = UPDATED_PICTURE;
        hero.powers = UPDATED_POWERS;
        hero.level = UPDATED_LEVEL;

        given()
            .body(hero)
```

```
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .header(ACCEPT, APPLICATION_JSON)
            .when()
            .put("/api/heroes")
            .then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .body("name", Is.is(UPDATED_NAME))
            .body("otherName", Is.is(UPDATED_OTHER_NAME))
            .body("level", Is.is(UPDATED_LEVEL))
            .body("picture", Is.is(UPDATED_PICTURE))
            .body("powers", Is.is(UPDATED_POWERS));

        List<Hero> heroes = get("/api/heroes").then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .extract().body().as(getHeroTypeRef());
        assertEquals(NB_HEROES + 1, heroes.size());
    }

    @Test
    @Order(4)
    void shouldRemoveAnItem() {
        given()
            .pathParam("id", heroId)
            .when().delete("/api/heroes/{id}")
            .then()
            .statusCode(NO_CONTENT.getStatusCode());

        List<Hero> heroes = get("/api/heroes").then()
            .statusCode(OK.getStatusCode())
            .header(CONTENT_TYPE, APPLICATION_JSON)
            .extract().body().as(getHeroTypeRef());
        assertEquals(NB_HEROES, heroes.size());
    }

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

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.

[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.

```
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
```

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

```
quarkus.http.port=8083
```

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.[7] 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:

```
@ConfigProperty(name = "level.multiplier", defaultValue="1")
int levelMultiplier;
```

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:

```
public Hero persistHero(@Valid Hero hero) {
    hero.level = hero.level * levelMultiplier;
    Hero.persist(hero);
    return hero;
}
```

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:

```
# Business configuration
level.multiplier = 3
```

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" $(oc get route -n USERNAME-codeready | grep 8083 |
awk '{ print $2 }')/api/heroes -v

< HTTP/1.1 201 Created
```

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 $(oc get route -n USERNAME-codeready | grep 8083 | awk '{ print $2
}')/api/heroes/952 | jq

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

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

[hand o right] Call to action

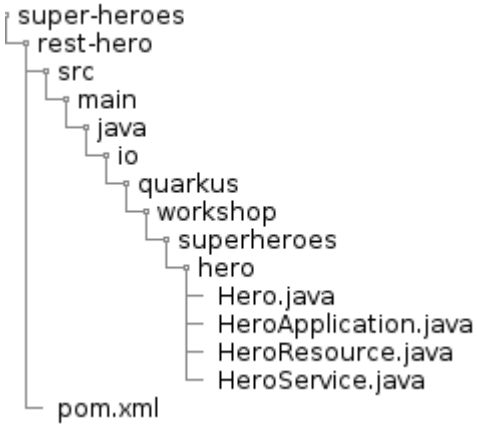
In the `application.properties` file, add: `%test.level.multiplier=1` which set the multiplier to 1 when running the tests. We will cover the `%test` syntax soon.

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.[8]

[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:

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

Open API

Now, you *curl* the hero endpoint:

```
$ curl $(oc get route -n USERNAME-codeready | grep 8083 | awk '{ print $2
}')/q/openapi
---
```

```
openapi: 3.0.3
info:
    title: rest-hero API
    version: 1.0.0-SNAPSHOT
paths:
    /api/heroes:
        get:
            responses:
                "200":
                    description: OK
        put:
            requestBody:
                content:
                    application/json:
                        schema:
                            $ref: '#/components/schemas/Hero'
            responses:
                "200":
                    description: OK
        post:
            requestBody:
                content:
                    application/json:
                        schema:
                            $ref: '#/components/schemas/Hero'
            responses:
                "200":
                    description: OK
    /api/heroes/hello:
        get:
            responses:
                "200":
                    description: OK
                    content:
                        text/plain:
                            schema:
                                type: string
    /api/heroes/random:
        get:
            responses:
                "200":
                    description: OK
    /api/heroes/{id}:
        get:
            parameters:
                - name: id
                  in: path
                  required: true
                  schema:
                      format: int64
                      type: integer
```

```
            responses:
                "200":
                    description: OK
        delete:
            parameters:
                - name: id
                  in: path
                  required: true
                  schema:
                      format: int64
                      type: integer
            responses:
                "200":
                    description: OK
components:
    schemas:
        Hero:
            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
```

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

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

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.*;
import javax.ws.rs.core.*;
import java.net.URI;
import java.util.List;

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

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

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

    @Inject
    HeroService service;

    @Operation(summary = "Returns a random hero")
    @APIResponse(responseCode = "200", content = @Content(mediaType =
APPLICATION_JSON, schema = @Schema(implementation = Hero.class, required = true)))
    @GET
    @Path("/random")
    public Response getRandomHero() {
        Hero hero = service.findRandomHero();
        LOGGER.debug("Found random hero " + hero);
        return Response.ok(hero).build();
    }

    @Operation(summary = "Returns all the heroes from the database")
```

```
    @APIResponse(responseCode = "200", content = @Content(mediaType =
APPLICATION_JSON, schema = @Schema(implementation = Hero.class, type = SchemaType
.ARRAY)))
    @APIResponse(responseCode = "204", description = "No heroes")
    @GET
    public Response getAllHeroes() {
        List<Hero> heroes = service.findAllHeroes();
        LOGGER.debug("Total number of heroes " + heroes);
        return Response.ok(heroes).build();
    }

    @Operation(summary = "Returns a hero for a given identifier")
    @APIResponse(responseCode = "200", content = @Content(mediaType =
APPLICATION_JSON, schema = @Schema(implementation = Hero.class)))
    @APIResponse(responseCode = "204", description = "The hero is not found for a
given identifier")
    @GET
    @Path("/{id}")
    public Response getHero(
        @Parameter(description = "Hero identifier", required = true)
        @PathParam("id") Long id) {
        Hero hero = service.findHeroById(id);
        if (hero != null) {
            LOGGER.debug("Found hero " + hero);
            return Response.ok(hero).build();
        } else {
            LOGGER.debug("No hero found with id " + id);
            return Response.noContent().build();
        }
    }

    @Operation(summary = "Creates a valid hero")
    @APIResponse(responseCode = "201", description = "The URI of the created hero",
content = @Content(mediaType = APPLICATION_JSON, schema = @Schema(implementation =
URI.class)))
    @POST
    public Response createHero(
        @RequestBody(required = true, content = @Content(mediaType = APPLICATION_JSON,
schema = @Schema(implementation = Hero.class)))
        @Valid Hero hero, @Context UriInfo uriInfo) {
        hero = service.persistHero(hero);
        UriBuilder builder = uriInfo.getAbsolutePathBuilder().path(Long.toString(hero
.id));
        LOGGER.debug("New hero created with URI " + builder.build().toString());
        return Response.created(builder.build()).build();
    }

    @Operation(summary = "Updates an exiting  hero")
    @APIResponse(responseCode = "200", description = "The updated hero", content =
@Content(mediaType = APPLICATION_JSON, schema = @Schema(implementation = Hero.class)))
    @PUT
```

```
    public Response updateHero(
        @RequestBody(required = true, content = @Content(mediaType = APPLICATION_JSON,
schema = @Schema(implementation = Hero.class)))
        @Valid Hero hero) {
        hero = service.updateHero(hero);
        LOGGER.debug("Hero updated with new valued " + hero);
        return Response.ok(hero).build();
    }

    @Operation(summary = "Deletes an exiting hero")
    @APIResponse(responseCode = "204")
    @DELETE
    @Path("/{id}")
    public Response deleteHero(
        @Parameter(description = "Hero identifier", required = true)
        @PathParam("id") Long id) {
        service.deleteHero(id);
        LOGGER.debug("Hero deleted with " + id);
        return Response.noContent().build();
    }

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

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:

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

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 = "Hero API",
        description = "This API allows CRUD operations on a hero",
        version = "1.0",
        contact = @Contact(name = "Quarkus", url = "https://github.com/quarkusio")),

    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 = "heroes", description = "Anybody interested in heroes")
    }
)
public class HeroApplication extends Application {
}
```

Customized Contract

If you go back to the `$URL/q/openapi` endpoint you will see the following OpenAPI contract:

```
---
openapi: 3.0.3
info:
    title: rest-hero API
    version: 1.0.0-SNAPSHOT
paths:
    /api/heroes:
        get:
            summary: Returns all the heroes from the database
            responses:
                "200":
                    description: OK
                    content:
                        application/json:
                            schema:
                                type: array
```

```
                            items:
                                $ref: '#/components/schemas/Hero'
            "204":
                description: No heroes
    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:
                            format: uri
                            type: string
/api/heroes/hello:
    get:
        responses:
            "200":
                description: OK
                content:
                    text/plain:
                        schema:
                            type: string
/api/heroes/random:
    get:
        summary: Returns a random hero
        responses:
            "200":
                description: OK
                content:
```

```
                        application/json:
                            schema:
                                $ref: '#/components/schemas/Hero'
    /api/heroes/{id}:
        get:
            summary: Returns a hero for a given identifier
            parameters:
                - name: id
                  in: path
                  description: Hero identifier
                  required: true
                  schema:
                      format: int64
                      type: integer
            responses:
                "200":
                    description: OK
                    content:
                        application/json:
                            schema:
                                $ref: '#/components/schemas/Hero'
                "204":
                    description: The hero is not found for a given identifier
        delete:
            summary: Deletes an exiting hero
            parameters:
                - name: id
                  in: path
                  description: Hero identifier
                  required: true
                  schema:
                      format: int64
                      type: integer
            responses:
                "204": {}
components:
    schemas:
        Hero:
            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
```

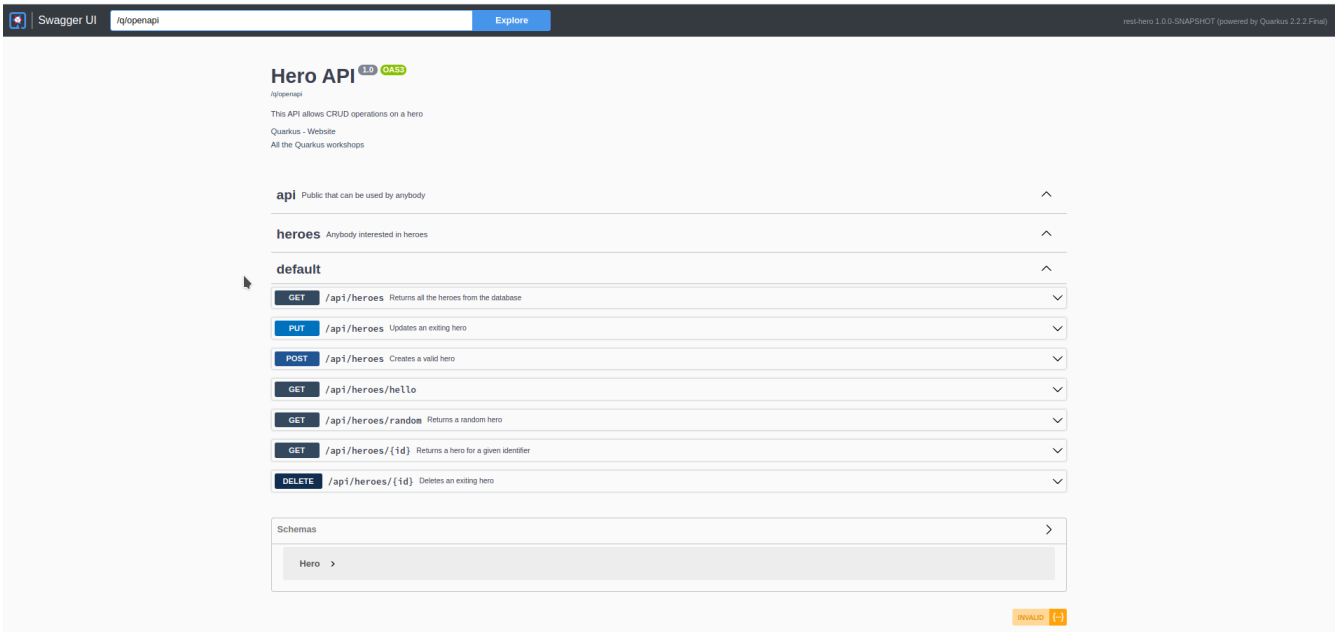
Swagger UI

When building APIs, developers want to test them quickly. Swagger UI is a great tool permitting to visualize and interact with your APIs.[9] 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 `/q/swagger-ui`. So, once your application is started, you can go to `$URL/q/swagger-ui` and play with your API.

[hand o right] Call to action

You can visualize your API's operations and schemas.

Notice that in the Swagger UI you have your specific URL exposing the rest-hero service



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:

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

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

[hand o right] Call to action

Execute the test using `./mvnw test`.

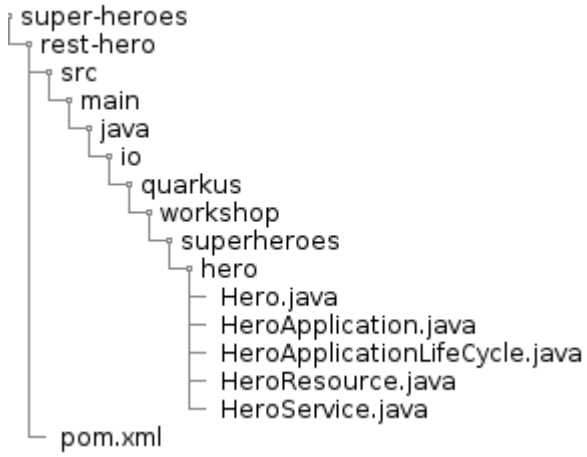
If you have any problem with the code, don't understand or feel you are running, remember to ask for some help. Also, you can get the code of this entire workshop from https://github.com/Red-Hat-Developer-Games/quarkus-workshop/tree/main/quarkus-workshop-super-heroes.

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 the hero endpoint url. 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/quarkus-app/quarkus-run.jar`. You will see that the `foo` profile is enabled. As not overridden, the `level.multiplier` property has the value 3.

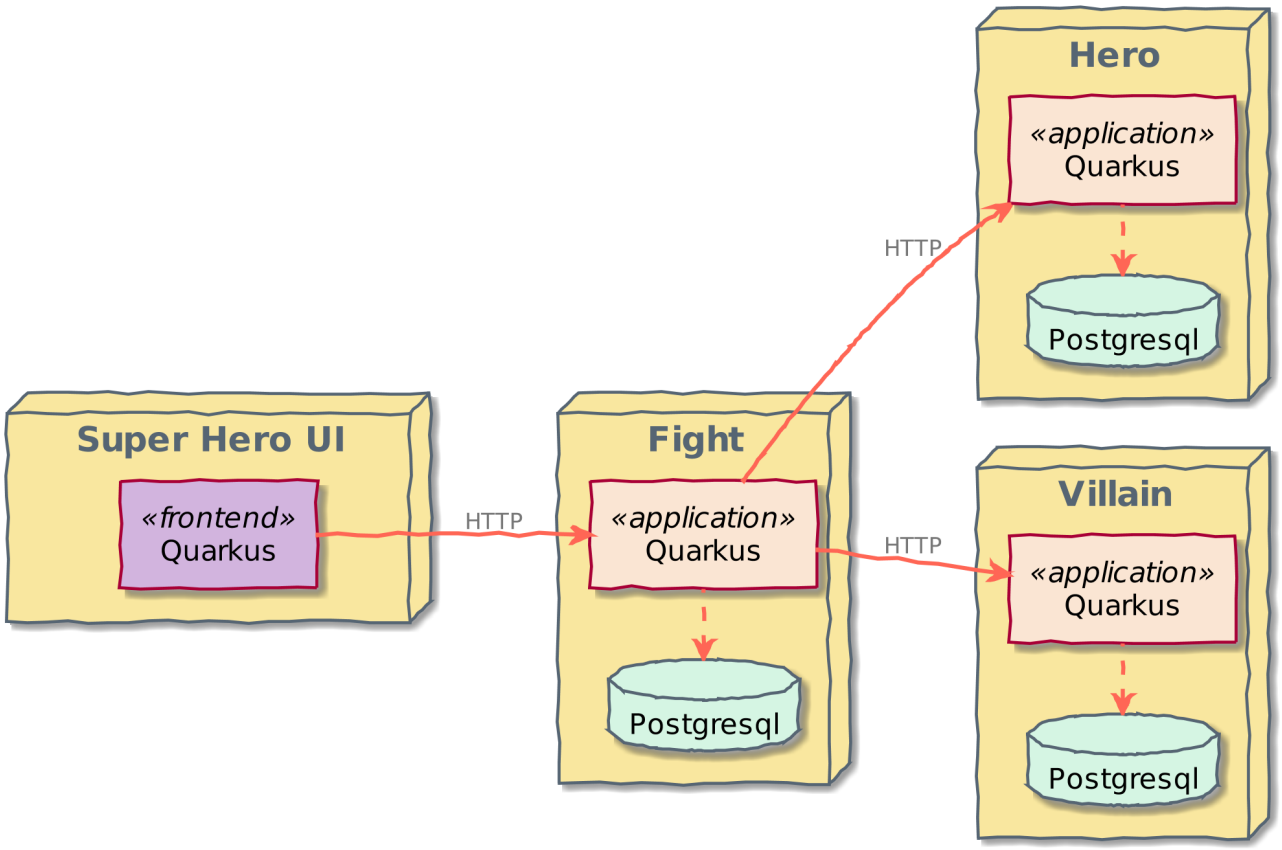
Profiles are very useful to customize the configuration per environment. We are going to see an example of such customization in the next section.

[1] RESTEasy https://resteasy.github.io

[2] RestAssured http://rest-assured.io
[3] Panache https://github.com/quarkusio/quarkus/tree/master/extensions/panache
[4] Agroal https://agroal.github.io
[5] ArC https://github.com/quarkusio/quarkus/tree/master/independent-projects/arc
[6] Quarkus - Contexts and Dependency Injection https://quarkus.io/guides/cdi-reference.html
[7] Microprofile Config https://microprofile.io/project/eclipse/microprofile-config
[8] MicroProfile OpenAPI https://github.com/eclipse/microprofile-open-api
[9] Swagger UI https://swagger.io/tools/swagger-ui

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.



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
```

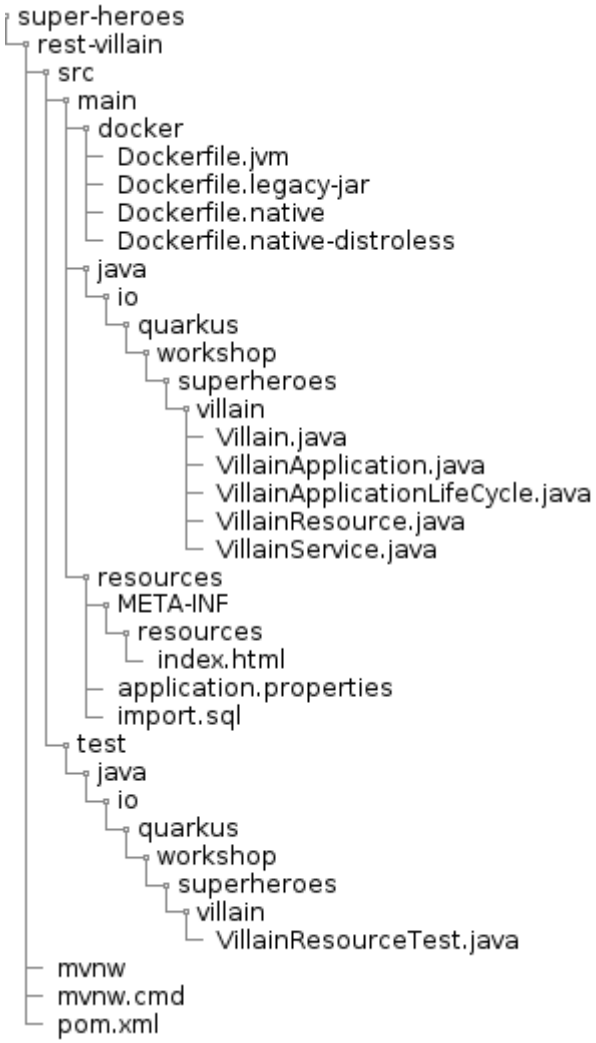
Villain Microservice

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" $(oc get route -n USERNAME-codeready | grep 8084 |
awk '{ print $2 }')/api/villains -v

< HTTP/1.1 201 Created
```

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

```
$ curl $URL/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 $URL/q/swagger-ui.

Change the URL with yours.

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 `$CHE_PROJECTS_ROOT/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:2.2.2.Final:create \
    -DprojectGroupId=io.quarkus.workshop.super-heroes \
    -DprojectArtifactId=rest-fight \
    -DclassName="io.quarkus.workshop.superheroes.fight.FightResource" \
    -Dpath="api/fights" \
    -Dextensions="jdbc-postgresql,hibernate-orm-panache,hibernate-validator,quarkus
-resteasy-jsonb,smallrye-openapi"
cd rest-fight
```

[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>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <scope>test</scope>
</dependency>
```

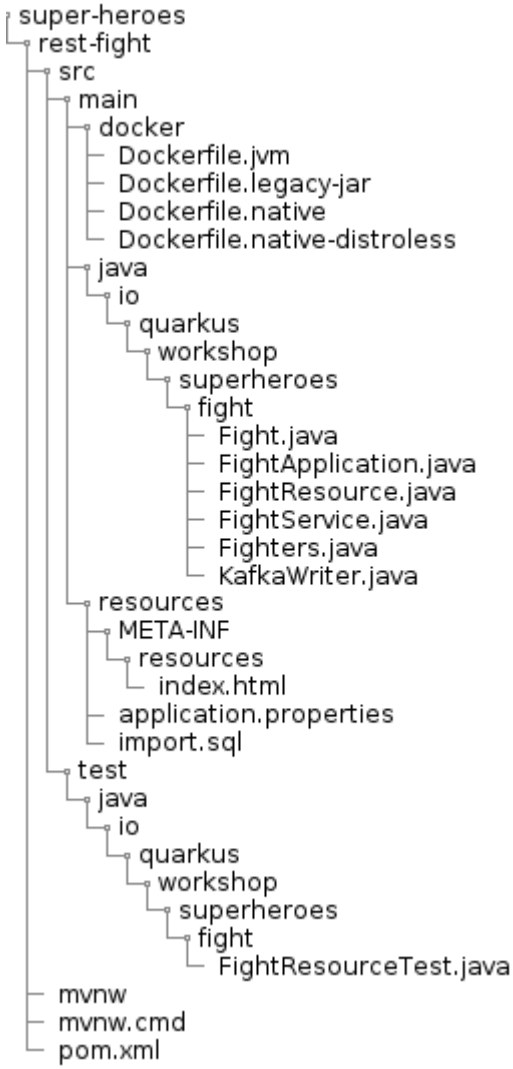
Prefering Web UI

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 ultimately 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 RESTEasy";
    }
}
```

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")),
    externalDocs = @ExternalDocumentation(url = "https://github.com/Red-Hat-Developer-
Games/quarkus-workshop", 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
quarkus.http.host=0.0.0.0

## Database configuration
quarkus.datasource.jdbc.url=jdbc:postgresql://fights-database.USERNAME-
heroes: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



quarkus.kubernetes-client.trust-certs=true
quarkus.kubernetes-client.namespace=USERNAME-heroes
quarkus.openshift.expose=true
```

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
@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("/q/openapi")
            .then()
            .statusCode(OK.getStatusCode());
    }

    @Test
    void shouldPingSwaggerUI() {
        given()
```

```
            .when().get("/q/swagger-ui")
            .then()
            .statusCode(OK.getStatusCode());
    }



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

    @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
        };
    }
```

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 $URL/api/fights
```

Change the URL with yours.

Remember that you can also check Swagger UI by going to $URL/q/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`.[10] 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 $(oc get route -n USERNAME-codeready | grep 8082 | awk
'{ print $2 }')/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

[hand o right] Call to action

Execute `npm install` in the `ui-super-heroes` directory for building 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

Set the following env var to allow the ui-super-heroes service contact the rest-fight.

```
$ export BASE_PATH=https://$(oc get route -n USERNAME-codeready | grep 8082 | awk '{
print $2 }')
```

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 (`mvn quarkus:dev`), open a browser on the ui-super-heroes url it should display the main web page. You can get the `ui-super-heroes` url with this command:

```
$ oc get route -n USERNAME-codeready | grep 8080 | awk '{ print $2 }'
```

Notice that the 8080 is the default Quarkus port as we didn't change it in the `application.properties` this time.

Welcome to Super Heroes Fight!

Astérix

9

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

NEW FIGHTERS

FIGHT !

Match

14

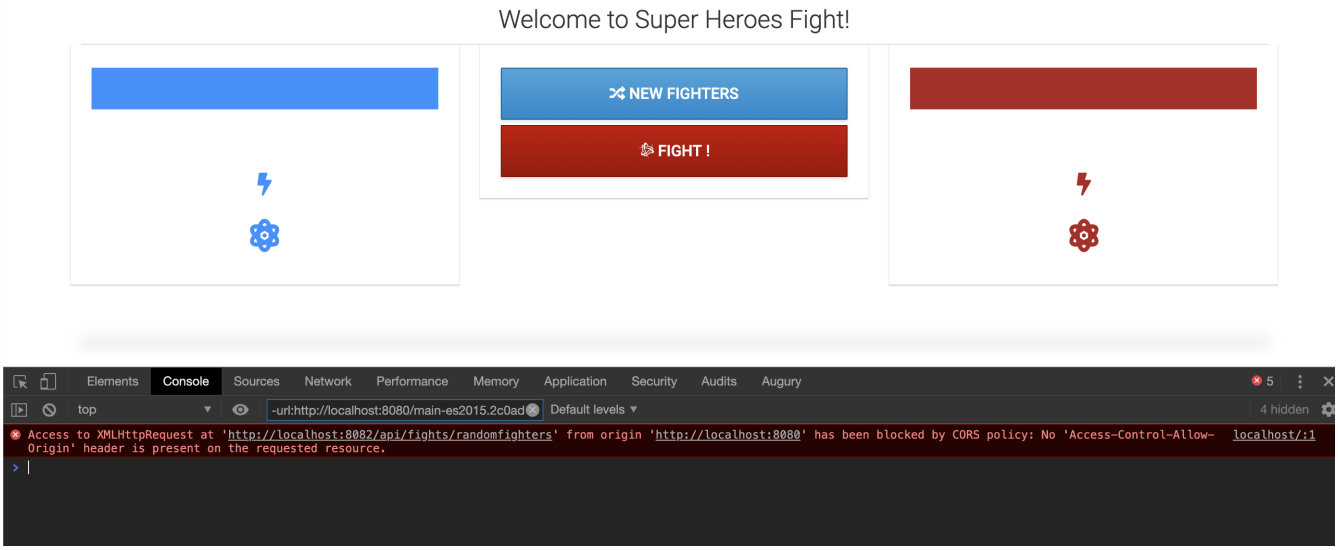
Accelerated Healing, Durability, Energy Absorption, Energy Blasts, Enhanced Hearing, Flight, Invulnerability, Jump, Super Breath, Super Speed, Super Strength, Telekinesis, Vision - Heat, Vision - Telescopic, Vision - X-Ray

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. [11] So when we want our heroes and villains to fight, we actually cross several origins: we go from the UI to Fight API which invokes Hero API and Villain API. 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:

```
quarkus.http.cors=true
```

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.):

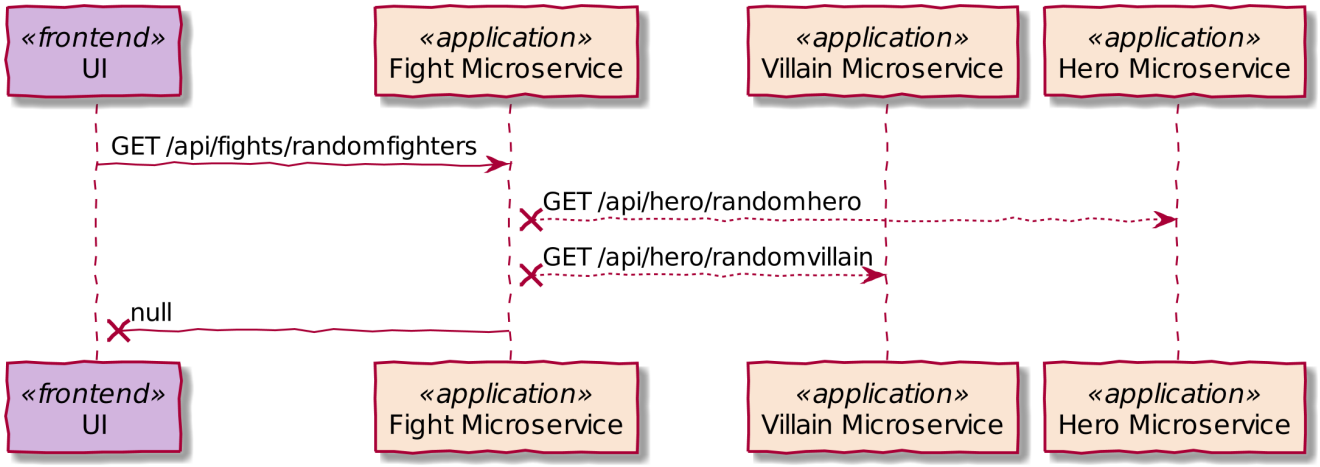
Property	Description
`quarkus.http.cors.origins`	The comma-separated list of origins allowed for CORS. The filter allows any origin if this is not set.
`quarkus.http.cors.methods`	The comma-separated list of HTTP methods allowed for CORS. The filter allows any method if this is not set.
`quarkus.http.cors.headers`	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:



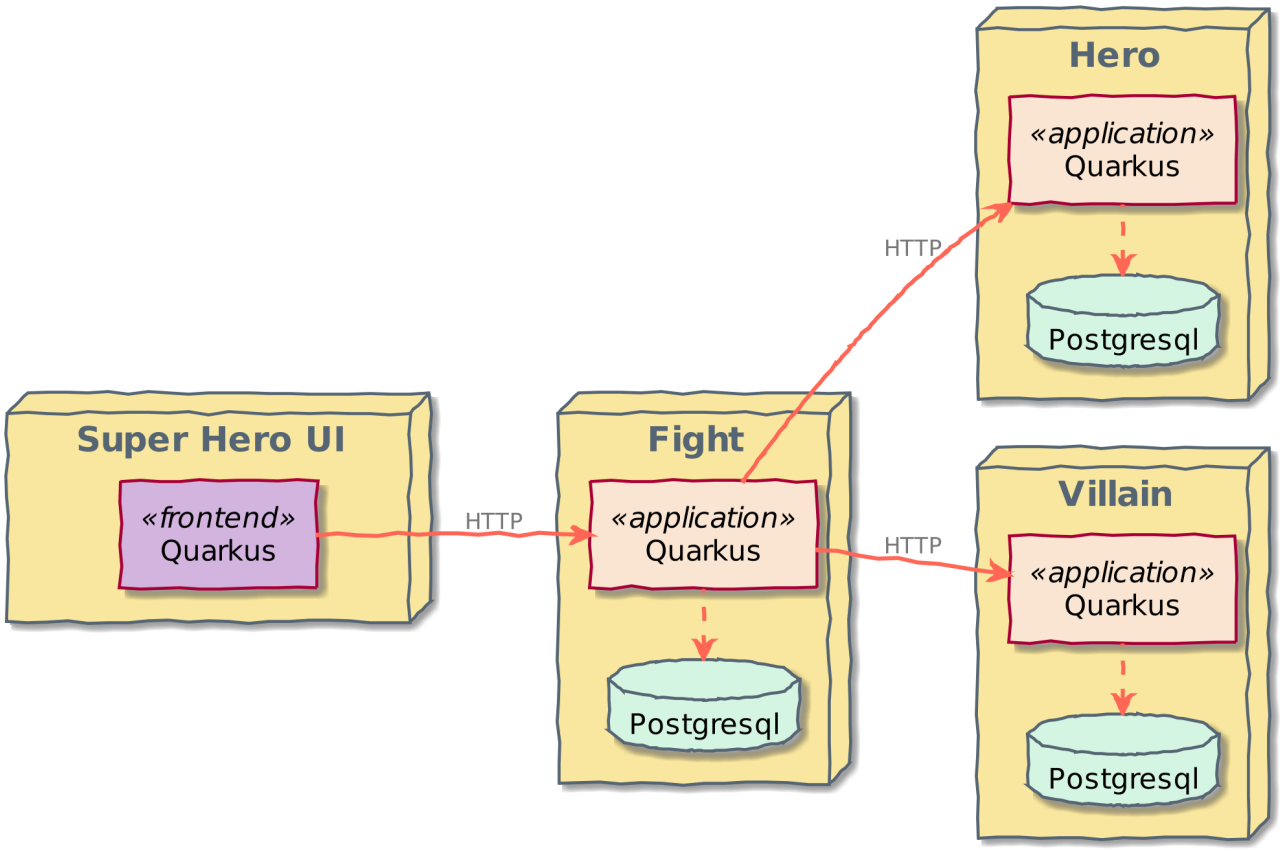
Remember the function to retrieve random fighters. We are currently returning `null`. Let's move to the next session to see how we can implement this method.

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

[11] 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.

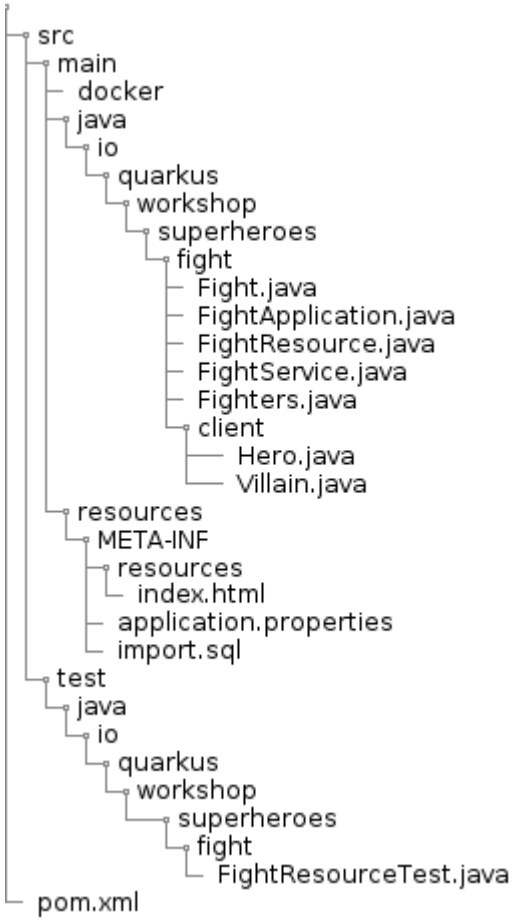


REST Client

This chapter explains how to use the MicroProfile REST Client in order to interact with REST APIs with very little effort.[12]

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:

```
%dev.io.quarkus.workshop.superheroes.fight.client.HeroService/mp-
rest/url=http://${WKSPC_SERVICE:localhost}:8083
io.quarkus.workshop.superheroes.fight.client.HeroService/mp-
rest/scope=javax.inject.Singleton
%dev.io.quarkus.workshop.superheroes.fight.client.VillainService/mp-
rest/url=http://${WKSPC_SERVICE: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 your specific rest-fight service exposed by the `USERNAME-codeready` project, http://localhost:8083 by default, as the base URL. Using this configuration, calling the `findRandomHero` method of `HeroService` would result in an HTTP GET request being made to ${rest-hero-url}/api/heroes/random.

This configuration works only for the `dev` profile. In order to make it work, you need to set up the `WKSPC_SERVICE` env var. You can do it running the following command:

```
$ export WKSPC_SERVICE="$(oc get svc -n USERNAME-codeready | grep 8083 |  awk '{ print
$1 }')"
```

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.[13]

[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 the corresponding url (`$(oc get route -n USERNAME-codeready | grep 8080 | awk '{ print $2 }')`) and start fighting (finally !).

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

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

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 java executables and the deployment of our system in OpenShift using the Quarkus OpenShift extension.

Build and Deploy (in a single step)

Add the OpenShift extension running the following command in the rest-heroes directory:

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

Some properties need to be added to the `application.properties` file of each microservice. Edit the file and add:

```
quarkus.kubernetes-client.trust-certs=true
quarkus.kubernetes-client.namespace=${PROJECT_NAME:USERNAME-heroes}

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

The expose parameter's purpose is to expose the service to clients outside the cluster via `Route`.

Make sure you change USERNAME with your own

To trigger the build and deployment in a single step, run the following command. Make sure that you are logged and using the `USERNAME-heroes` project.

```
$ oc login
$ oc project USERNAME-heroes
$ ./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

The aforementioned command will build a jar file locally, trigger a container image build and then apply the generated OpenShift resources. To confirm the above command has created an image stream, a service resource and has deployed the application (has a pod running), you can run these commands:

```
oc get is
oc get pods
oc get svc
```

If everything went well the URL of the microservice should be logged in the console where you ran

the build&deploy command:

```
[INFO] [io.quarkus.container.image.openshift.deployment.OpenshiftProcessor] Push
successful
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Deploying to openshift
server: https://api.cluster-2599.2599.example.opentlc.com:6443/ in namespace: user2-
heroes.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Service rest-
hero.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: ImageStream
openjdk-11.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: ImageStream rest
-hero.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: BuildConfig rest
-hero.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: DeploymentConfig
rest-hero.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] Applied: Route rest-hero.
[INFO] [io.quarkus.kubernetes.deployment.KubernetesDeployer] The deployed application
can be accessed at: http://rest-hero-user2-heroes.apps.cluster-
2599.2599.example.opentlc.com
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
86106ms
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  02:39 min
[INFO] Finished at: 2020-11-11T12:57:24Z
[INFO] ------------------------------------------------------------------------
```

Deploying the others microservices

For villain microservice follow the same approach.

Fight microservice

- Add the OpenShift extension running the following command in the rest-fights directory:

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

- Add the following properties in the rest-fights `application.properties` file:

```
quarkus.kubernetes-client.trust-certs=true
quarkus.kubernetes-client.namespace=${PROJECT_NAME:USERNAME-heroes}

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

- Also, we need to configure the new locations of the hero and villain microservice. Edit the `application.properties` file and modify the following properties:

```
io.quarkus.workshop.superheroes.fight.client.HeroService/mp-rest/url=http://rest-
hero:8083
io.quarkus.workshop.superheroes.fight.client.VillainService/mp-rest/url=http://rest-
villain:8084
```

Trigger the build and deployment, run the following command. Make sure that you are logged and using the `USERNAME-heroes` project.

```
$ oc login
$ oc project USERNAME-heroes
$ ./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

UI microservice

- Add the OpenShift extension running the following command in the rest-fights directory:

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

- Add the following properties in the ui-super-heroes `application.properties` file:

```
quarkus.kubernetes-client.trust-certs=true
quarkus.kubernetes-client.namespace=${PROJECT_NAME:USERNAME-heroes}

quarkus.openshift.route.expose=true
quarkus.openshift.env.vars.BASE_PATH=${FIGHT_ROUTE}
```

The `` property make the UI able to contact the fight microservice. As it depends on an env var, consider to define it using the following command:

```
$ export FIGHT_ROUTE=http://$(oc get route -n USERNAME-heroes | grep fight | awk '{
print $2 }')
```

Don’t forget use your own user

Trigger the build and deployment, run the following command. Make sure that you are logged and using the `USERNAME-heroes` project.

```
$ oc login
$ oc project USERNAME-heroes
$ ./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

Once everything is configured and deployed, your system is now running on OpenShift.

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