Workshop

Table of Contents

Quarkus presentation

Slides available here.

Create a microservice in command line

```
$ mvn io.quarkus:quarkus-maven-plugin:1.5.1.Final:create
-DprojectGroupId=org.montrealjug -DprojectArtifactId=hello-quarkus
-DclassName="org.montrealjug.api.QuarkusWorkshopResource" -Dpath="/hello"
```

Modifying endpoint

Go inside the project just generated `hello-quarkus` and execute the following command :

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

Try the endpoint through a curl command :

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

To change the endpoint response, go to the class QuarkusWorkshopResource.

And change the return of the hello method to "hello Quarkus"

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

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

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

You can check the maven command, no hot reload occurred.

You have to call the endpoint through the curl command to trigger the hot reload.

Try to create a standard Jar with maven :

```
$ ./mvnw package
```

We get an error because Quarkus generated tests for the endpoint when we ran the command to create the project.

To correct it, we have to go to the test method `testHelloEndpoint` in the class `QuarkusWorkshopResourceTest`.

And change "is("hello")" to "is("hello Quarkus")".

```
@QuarkusTest
public class QuarkusWorkshopResourceTest {

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

}
```

Native compilation

To create a native executable, you have to way :

- with local GraalVM
- with GraalVM in Docker

With local GraalVM

Package

You can create a native executable using:

```
$ ./mvnw package -Pnative
```

Error there can be related to GRAALVM not properly configured.

~/.sdkman/candidates/java/20.1.0.r11-grl/bin/gu install native-image export GRAALVM_HOME=~/.sdkman/candidates/java/20.1.0.r11-grl/

Execute

```
$ ./target/hello-quarkus-1.0-SNAPSHOT-runner
```

Call the endpoint

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

With GraalVM in Docker

```
$ ./mvnw package -Pnative -Dquarkus.native.container-build=true
```

If the generation is long or don’t finish you have to increase your memory limit in Docker.

Execute

```
$ ./target/hello-quarkus-1.0-SNAPSHOT-runner
```

It could work or not depending on your OS. Because you should execute the native image in a Docker container it has been packaged for.

You can try to run the executable, but from a Linux machine, such as your Docker VM:

```
docker run -p 8080:8080 -v $PWD/target:/target -it ubuntu:latest /target/hello-
quarkus-1.0-SNAPSHOT-runner
```

Create a microservice and add an extension (Mongo)

```
$ mvn io.quarkus:quarkus-maven-plugin:1.5.1.Final:create
-DprojectGroupId=org.montrealjug -DprojectArtifactId=mongo-quarkus
-DclassName="org.montrealjug.api.TodoResource" -Dpath="/todos"
```

In order to list all the extension available in a Quarkus project, you could go to the Quarkus Website at the extensions page.

Or you could use the following command inside the project you just created :

```
$ ./mvnw quarkus:list-extensions
```

There are around 250 extensions at the moment.

Adding the mongo and opentracing extensions to our project

To add a quarkus extension to your project, you have 2 ways:

- through command line
- by modifying the pom

Through command line

```
./mvnw quarkus:add-extension -Dextensions="quarkus-mongodb-client,quarkus-smallrye
-opentracing"
```

The following dependency has been added to our pom file :

```
<dependencies>
    ....
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-mongodb-client</artifactId>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-smallrye-opentracing</artifactId>
    </dependency>
</dependencies>
```

By modifying the pom

You can directly add the dependencies to the dependencies part of the pom file.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-mongodb-client</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

Adding testcontainer

Test Container is going to allow us to have a mongo database directly in our tests suites:

```xml
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>testcontainers</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <scope>test</scope>
</dependency>
```

Coding our first test

In the class TodoResourceTest add the following annotations on top of the class declaration :

```java
@QuarkusTest
@QuarkusTestResource(DataResource.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class TodoResourceTest {
```

Delete the method testTodosEndpoint() and add the following methods :

```java
@Test
@Order(1)
public void testPutEndpoint() {
    Todo todo = new Todo("thisIsMyTodoTitle", true);
    JsonPath result = given()
                .body(todo)
                .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
                .header(HttpHeaders.ACCEPT, MediaType.APPLICATION_JSON)
                .when()
                .put("/todos")
                .then()
                .statusCode(HttpStatus.SC_OK)
                .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
                .extract()
                .response()
                .jsonPath();


    assertEquals("thisIsMyTodoTitle", result.getString("title"));
    assertEquals(true, result.getBoolean("completed"));
}

@Test
@Order(2)
public void testGetEndpoint() {
    JsonPath result = given()
                .when()
                .get("/todos")
                .then()
                .statusCode(HttpStatus.SC_OK)
                .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)
                .extract()
                .response()
                .jsonPath();

    System.out.println(result.prettyPrint());

    assertEquals("thisIsMyTodoTitle", result.getString("title[0]"));
    assertEquals(true, result.getBoolean("completed[0]"));
}
```

Create the Todo class in the main/java folder with the package "org.montrealjug.api" :

```java
package org.montrealjug.api;

import java.util.Objects;

public class Todo {
    private String title;
    private boolean completed;

    public Todo(String title, boolean completed) {
        this.title = title;
        this.completed = completed;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Todo todo = (Todo) o;
        return isCompleted() == todo.isCompleted() &&
                Objects.equals(getTitle(), todo.getTitle());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getTitle(), isCompleted());
    }
}
```

Create the Dataresource class in the test/java folder with the package "org.montrealjug.api" :

```java
package org.montrealjug.api;

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

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

public class DataResource implements QuarkusTestResourceLifecycleManager {

    private static final Integer MONGO_PORT = 27017;
    private static GenericContainer MONGO = null;

    @Override
    public Map<String, String> start() {
        MONGO = new GenericContainer("mongo:4.0.8").withExposedPorts(MONGO_PORT);
        MONGO.start();
        final String hosts = (MONGO.getContainerIpAddress() + ":" + MONGO
.getMappedPort(MONGO_PORT));

        return Collections.singletonMap("quarkus.mongodb.hosts", hosts);
    }

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

At this point, the project should compile in your IDE.

But we have to implement our endpoint and our service.

```java
package org.montrealjug.api;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import org.bson.Document;
import org.eclipse.microprofile.config.inject.ConfigProperty;

import javax.enterprise.context.ApplicationScoped;
import java.util.ArrayList;
import java.util.List;

@ApplicationScoped
public class TodoService {

    @ConfigProperty(name = "quarkus.mongodb.database")
    private String database;

    @ConfigProperty(name = "custom.quarkus.mongodb.collection")
    private String collection;


    private MongoClient mongoClient;

    public TodoService(MongoClient mongoClient) {
        this.mongoClient = mongoClient;
    }

    public Document add(Todo todo) {
        Document document = new Document()
                .append("title", todo.getTitle())
                .append("completed", todo.isCompleted());
        getCollection().insertOne(document);
        return document;
    }

    private <Document> MongoCollection<org.bson.Document> getCollection() {
        return mongoClient.getDatabase(database).getCollection(collection);
    }

    public List<Todo> list() {
        List<Todo> list = new ArrayList<>();
        MongoCursor<Document> cursor = getCollection().find().iterator();
        try {
            Document doc;
            while (cursor.hasNext()) {
                doc = cursor.next();
                list.add(new Todo(doc.getString("title"), doc.getBoolean("completed")
));


            }

        } finally {
            cursor.close();
        }
        return list;
    }
}
```

Add mongo info to your properties :

```
quarkus.mongodb.database=jug-quarkus-workshop
custom.quarkus.mongodb.collection=todos
```

Code the endpoint :

```java
package org.montrealjug.api;

import org.bson.Document;

import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import java.util.List;

@Path("/todos")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class TodosResource {

    private TodoService service;

    @Inject
    public TodosResource(TodoService service) {
        this.service = service;
    }

    @PUT
    public Document add(Todo todo) {
        return service.add(todo);
    }

    @GET
    public List<Todo> list() {
        return service.list();
    }
}
```

How to monitor a native app

We added earlier the "quarkus-smallrye-opentracing" extension :

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

To monitor all the transactions in our api, we are going to use Jaeger.

Building the image

https://quarkus.io/guides/container-image

/mvnw clean package -Pnative -Dquarkus.native.container-build=true

docker build -f src/main/docker/Dockerfile.native -t quarkus/mongo-quarkus:1.0 .

docker-compose up

curl -X PUT -H "Content-Type: application/json" -d '{"title":"Jam","completed":"false"}' http://localhost:8080/todos | jq curl -X PUT -H "Content-Type: application/json" -d '{"title":"Ham","completed":"false"}' http://localhost:8080/todos | jq

curl -X GET http://localhost:8080/todos | jq

You can do more request to have some data in jaeger.

When you consider having sent many requests you can go to Jaeger.

Auth0

TODO max

Reactive

TODO max