Tasks and Assignments Course 02267

Hubert Baumeister

January 2024

1 Task (single/in pairs/using mob programming): A First REST project TASK_REST1

You can use any framework to write REST services and their clients. For this course though, I recommend to use the `JAX-RS` (Java Extensions for Rest Services) standard, which I am also using in the examples. Here[1] is documentation for implementing resources (the server side of REST applications). And here[2] is the documentation for writing the clients. We are also going to use Quarkus (`http://quarkus.io`) as our application server.

Important These tasks lead you in small steps to code templates that you can also use with later tasks in the examination project (with proper renaming of the projects in the the pom.xml files of course.)

1.1 Create your first REST Web service

Quarkus (`http://quarkus.io`) is a lightweight Web server based on the JBoss/WildFly application server. The idea of Quarkus is to create a jar file which already includes a Web server. One runs the jar file to create a stand-alone Web service which runs on port 8080. Its intended use is inside Docker containers, but can also be run outside a container.

% Update for 2024: % 1) New Quarkus 3.0 available % Important: Move to jakarta % I think I already used at some places 3.0.0.Alpha2 % 2) Packages to select `RestEasy Reactive` (Starter Code Yes). But % Classic will also work, but Starter Code has to be Yes to include % starter code.

The steps to create and run a simple REST application is as follows:

1. go to the Quarkus generator site[3]. Here, you can configure the generator by specifying the technologies that the Web application should support. We need `RESTEasy Classic` (basic REST Web service functionality) and `RESTEasy Classic JSON-B` (Serializer to and from JSON), and `RESTEasy Classic JAXB` (Serializer to and from XML) for REST Web services.

2. The result of the generation process is a zip file containing a Maven project which you can take as the bases for your own Web service.

3. To create a jar file that can be run using the `java -jar <jar-file>` command, you should run `mvn package`. This will produce the `quarkus-app/quarkus-run.jar` jar-file in the `target` directory that you can execute via `java -jar target/quarkus-app/quarkus-run.jar`.

4. Once you see `Quarkus` logo in the output, go to `http://localhost:8080/hello` in the browser. If working, this will return `Hello RESTEasy` in the browser.

5. Once, you have checked that the downloaded up work, you can start one-by-one modifying the application. For example, you should add additional dependencies, like the Cucumber dependencies to run local tests, to the `pom.xml` file. You can copy the dependencies from the Cucumber Example[4].

[1] `https://docs.jboss.org/resteasy/docs/5.0.1.Final/userguide/html/`

[2] `https://docs.jboss.org/resteasy/docs/5.0.1.Final/userguide/html/RESTEasy_Client_Framework.html`

[3] `http://code.quarkus.io`

[4] `http://www2.imm.dtu.dk/courses/02267/files/CucumberExample.zip`

For development purposes, one can call `mvn compile quarkus:dev`, which starts the server for testing without creating the jar-file first. However, the `../quarkus-run.jar` file is later used in connection with docker.

1.2 Create your first client to test your Web service

The Quarkus demo project already contains a way of testing the service. However, we are not going to use those tests in the future. Thus the classes `GreetingResourceTest` and `NativeGreetingResourceIT` should be removed from the Quarkus demo project. Later, there will be tests inside the project with the REST interface, but those wont use the REST interface. Instead they will test the business logic directly.

To test the REST interface, we are using a second, normal Java project where the tests within that project will call the deployed REST services. Running `mvn test` in the client project will automatically run all tests accessing the REST service. This requires that the REST service is deployed and running.

To develop the test client, you can start from the Cucumber example project[4].

There are several libraries you can use to access REST Web services, e.g. Jersey (jersey-client). This is the library used in the examples from the lecture. Here are the dependencies for the `pom.xml` file

```
<!-- General REST framework -->
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-client</artifactId>
    <version>6.2.6.Final</version>
</dependency>
<!-- Serializer to and from JSON -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jsonb</artifactId>
    <version>3.6.4</version>
</dependency>
<!-- Serializer to and from XML -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jaxb</artifactId>
    <version>3.6.4</version>
</dependency>
```

You should write a Cucumber scenario like the following

```
Feature: hello service
Scenario: hello service retuns correct answer
   When I call the hello service
   Then I get the answer "Hello RESTEasy"
```

with step definitions

```
public class HelloServiceSteps {
  String result;
  HelloService service = new HelloService();

  @When("I call the hello service")
  public void iCallTheHelloService() {
     result = service.hello();
  }
  @Then("I get the answer {string}")
  public void iGetTheAnswer(String string) {
     assertEquals(string,result);
  }
}
```

Here, the method `hello()` in class `HelloService` makes the REST call using the `JAX-RS` standard.

1.3 Test JSON serialization and deserialization

You have now a running Web service and tests testing your Web service. But how are complex objects, like a Person class etc. serialized using JSON and XML? Note that in REST, the client can request a resource in a specific representation, like JSON or XML by setting the `accept` field in the HTTP header.

1.3.1 Create a resource for a person and return JSON

Define a class `PersonResource` for URL `person` with a GET method that returns a person object (instance of a class Person) with name, and address (both strings). When you access `localhost:8080/person`, the JSON representation of an object should be shown in the browser.

Also, define a Cucumber test in the client, testing that the desired object is being returned.

1.3.2 Create a put method for the person resource to update the person

Create a `put` method for the resource with URL `person` such that the person object is updated with the new information. Next time a `get` on `/perso` is used, the update object should be returned.

1.4 Notes

1.4.1 Note 1. Changed package names from javax.ws to jakarta.ws in the client

This refers to the examples shown in the video and slides about REST. Recently, the Java Enterprise framework has changed its name to jakarta. Thus, in the slides and the video for this course, some classes that were in packages starting with `javax.ws` have now moved to `jakarta.ws`. Usually, you the IDE helps you with finding the right packages for a class.

The server code is not effected by this, but the client code is. This means, that some package names shown in the slides have to be changed.

1.4.2 Note 2. Empty values in objects transferred between REST service and client

It may happen that a JSON object has empty values when serialised from an object. Vice versa, an object may have empty values when deserialised from a JSON object. The root cause for this problem is that the de/serialiser does not have enough rights to read and set the attributes of the object, because the attributes are private or only visible in the current package.

The solution is to use the Java Beans convention, which requires a default public constructor and public getter and setter methods, which allows everybody to construct an object and to read and set its attributes.