1.2 | Remote communication: REST/OpenAPI

Service-oriented architectures for remote services using REST and OpenAPI

---

1 Introduction

In a service-oriented architecture (SOA), applications are constructed based on reusable services that focus on making machine-to-machine interactions interoperable and independent from a particular programming model or language. In contrast to distributed objects, such as in the case of Java RMI, services are often stateless and rely on remote process communication (RPC) rather than remote method invocations (RMI) to objects. In an online setting, SOAs are typically built as web services and using web services standards. For example, interactions and information are exchanged using a standardized machine-processable format such as XML or JSON. As a transport layer, often HTTP is used. Higher-level service protocols specify which standards to use at each layer of the stack. Older, more formal specifications include SOAP and WSDL, while more recently the *Representational State Transfer* (REST) architecture has become dominant. During this session, we will focus on constructing such RESTful services for an online service. We will follow two approaches: *code-first* and *API-first*.

Goal: Your goal is to learn and practice the development and testing of remote communication through online RESTful services.

Loading the project. The code from which you start can be found on Toledo, under 'Assignments'. Make all your changes based on this application code: extend or modify this code when necessary, even if not explicitly described in the assignment. Use your preferred IDE or the command line.

In the **computer labs**, you can use the preinstalled IntelliJ IDE:

```
/localhost/packages/ds/intellij.sh
```

Unzip the file on Toledo, and (`File >`) `Open` the top-level folder to generate a project. For this lab session, there are two zip files: one for REST, and one for OpenAPI.

If you want to develop your code **on your own machine**, check the 'Bring Your Own Device Guide' on Toledo – we require you to use Java SE Development Kit (JDK) version **17**.

If necessary, tell IntelliJ to use the Java 17 SDK on your machine, rather than another version (e.g. 1.8 or 16). In the IDE, go to: `File -> Project Structure -> Project Settings -> Project`. Set the Project SDK to 17 and Project language level to the SDK default.

2 Used tools and middleware

Before we get to the key parts of this assignment, we first briefly explain the tools and middleware that need to be used for this assignment.

Spring Boot. The Spring Boot[1] platform contains a Java-based application server that can host HTTP-based services, such as traditional web apps, but also RESTful remote services. These services need to be developed in Java using the Spring API. The task of the application server is to listen for incoming HTTP requests, unmarshal and parse them, and call the right application-level operation in your RESTful service.

The service that you develop will be compiled and packaged together with the Spring Boot middleware, and will be deployed and running as a single JAR file on the Java VM.

Curl. Curl[2] is a well-known command-line tool to make HTTP calls to a given HTTP address. We will use this tool to interact remotely with our HTTP-based remote services from a client machine. This will allow you to inspect and control the actual communication with the remote service.

On Windows, you can use `curl.exe` instead of `curl` in the example commands in this assignment.

Maven. Maven[3] is a software building and packaging tool that we will use to manage the libraries and dependencies of our software project, compile our code, and package the online services with the Spring Boot middleware into a single JAR. It can also be used to run the applications.

You can use Maven on the command line in the root folder of your project to package and run your application using `mvn spring-boot:run`. Alternatively, you can also just build and package the JAR file with `mvn clean package`. In IntelliJ, you can use the plugins section of the Maven tab window to run the `spring-boot:run` command.

3 The food delivery application

You will develop an online service of a food delivery company. The online service offers remote operations that are accessible over the internet to inspect the menu and order one or more meals.

There are two key concepts:

- A `Meal` consists of a unique id, a short name, a longer description, a price, an energetic value in kcal, and a meal type (meat, fish, vegan, or veggie).
- An `Order` consists of an address to deliver the order and a list of meal names that one wants to order.

The online service is envisioned to offer the following operations via HTTP-based remote communication over the internet, to inspect the menu and put in an order:

- Retrieve a meal by its id.
- Retrieve the cheapest meal.
- Look up the most energetic meal.
- Put in an order for one or more meals that need to be delivered at the given address.

4 Part 1: Code-first remote services using REST

In this section, your goal is to implement, *code-first*, a food delivery application using a RESTful service architecture. To do this swiftly, we provide a ready-to-use and ready-to-run software project. The project is not fully implemented; only the REST operations to fetch all meals (`/rest/meals`) and fetch a single meal (`/rest/meals/{id}`) are implemented in the provided source code.

[1] `https://spring.io/projects/spring-boot`
[2] `https://curl.se/`
[3] `https://maven.apache.org/`

4.1 The *Richardson maturity model* for REST

Leonard Richardson presented a model,[4] known as the *Richardson maturity model*, that breaks down RESTful APIs into levels:

- *Level 0:* Remote APIs that only use HTTP as a transport system and do not rely on properties of the Web belong to this category, effectively using HTTP to tunnel any RPC protocol. For example, SOAP typically uses HTTP to contact a single predefined endpoint, and the requests are passed through XML message bodies.
- *Level 1 - Resources:* In this category of APIs, there is more than one API endpoint, and the API is based on the concept of resources. This means that such APIs employ URIs to address particular resources. For example, in the food delivery application, we have a meal resource with the address `/meals/id`. This category of APIs typically uses a single and fixed HTTP verb, typically POST, for all sorts of operations on the resources.
- *Level 2 - HTTP Verbs:* This category of services rely on the concept of resources (URIs) and more importantly use HTTP verbs to signify the intended client operation on resources. For example, `GET /restrpc/meals` would fetch the meals, `POST /restrpc/meals (body:meal)` would create a new meal, and `DELETE /restrpc/meals/123` would delete the resource `meal(id=123)`. In this level, it is important to follow the semantics of the HTTP protocol and produce correct response codes for each operation.
- *Level 3 - Hypermedia controls:* This level of remote APIs is hypermedia-driven like the Web. This means that resources should be discoverable by reflecting the hyperlinks related to the concerned resource. And, it should be normally based on the application state. For example, when you fetch the list of meals, each meal reflects 2 links: (1) a link to fetch the meal, and (2) a link to list all the meals. In a more complex application, it should reflect for each meal a link for ordering as well. If the application state indicates that a meal is sold out, the order link for that meal should not be reflected.

In this assignment, we call the *Level 2* type of remote APIs *RPC-Style REST API*; however, this form of REST APIs is not fully following the principles of the Web. Next to this, we call the Level 3 of purely HTTP-based APIs the *RESTful architecture*. The majority of the industry implements Level 2 REST APIs.

The assignment goals are twofold:

1. *REST, the RPC style*: our goal is to implement the food delivery application by only relying on the HTTP verbs (`GET/POST/PUT/DELETE`).
2. *True RESTful service architecture*: Next, our goal is to implement the same operations but in a true RESTful service architecture. These types of services are hypermedia-driven.

In the source code, we call the RPC-style REST *RestRpc* and the true REST just *Rest*. We briefly discuss the contents of this provided software project.

4.2 The provided software project

Conceptual and functional content. The provided project includes:

- The definition of the `Meal` concept with its key properties.
- The definition and implementation of two remote operations:
  1. `/restrpc/meals` and `/rest/meals`: to look up all existing meals, and
  2. `/restrpc/meals/{id}` and `/rest/meals/{id}`: to look up the meal with a given `id`

The implementation of these functional concepts. The implementations of the two RESTful services are defined in the following software artifacts residing in `src/main/java`.

In the "`be.kuleuven.foodrestservice.domain`" package:

- **Meal.java** and **MealType.java** are regular Java classes (POJOs).
- **MealsRepository.java** is a Spring component that contains the core application logic. It initializes a set of meals and implements two functions to list all available foods and to find a food item by its identifier.

In the "`be.kuleuven.foodrestservice.controllers`" package:

[4] `https://martinfowler.com/articles/richardsonMaturityModel.html`

- **MealsRestRpcStyleController.java** is a Spring REST controller that enables you to define your RPC-style REST API. There are currently two operations implemented: *getMealById(String id)* and *getMeals()*.
- **MealsRestController.java** is also a Spring REST controller. In this controller, we aim to implement a hypermedia-driven REST API. The same functions for looking up the meals are implemented. However, in this implementation, the operations return an object that encapsulates an *EntityModel*. This is a powerful concept since you are now able to compose related hypermedias to your resources, and define a correct HTTP *status* for each invocation (by using *ResponseEntity*; we will explain this later).

In the "`be.kuleuven.foodrestservice.exceptions`" package:

- **MealNotFoundException.java** and **MealNotFoundAdvice.java** define an exception that can be used by the REST server. This exception triggers the server to use HTTP Status 404 (not found) when a requested meal does not exist.

Other project artifacts. Furthermore, the software project contains the following artifacts:

- **pom.xml** is the Maven file that defines all the used dependencies and required plugins of the project. You do not need to edit this or change this during this assignment.
- **/resources/requests** contains some example scripts that interact with your APIs to *delete*, *add*, or *update* a meal.

4.3 Running the RESTful server

In the Maven window of IntelliJ, you can find the `spring-boot:run` command in the Plugins section. Using this command you will recompile, package, and run the Spring Boot application with your REST services.

Alternatively, you can also run the application from the command line using Java 17 and Maven. You can test your current Java version for your machine on the command line using `java -version`. You can then call the Maven command `mvn spring-boot:run` to build and run your application server.

In the terminal window of IntelliJ, you can now execute some client-side requests to test the implemented (RPC-style) REST API:

```
$ curl -X GET localhost:8080/restrpc/meals
$ curl -X GET localhost:8080/restrpc/meals/4237681a-441f-47fc-a747-8e0169bacea1
```

You can also execute these client requests in any console window from the project directory.

The first `curl` command lists the meals, and the second command only looks up the Portobello mushroom burger. As you can see, we use the `-X GET` option to indicate that our service call is a GET HTTP operation. If you add the `-v` option, you will get more verbose responses. For example, you can see that the HTTP response comes with the status code 200. This response code means that the resource has been fetched and it is transmitted in the message body. These are standard HTTP response codes.[5] If you want to look up an imaginary meal that does not exist in the system or was perhaps deleted from the menu (e.g. `id=12345`), your response comes with the HTTP status code 404, which means that the server cannot find the requested resource. You can try it with this `curl` command (make sure to copy/paste the spaces between each segment of the commands):

```
$ curl -v -X GET localhost:8080/restrpc/meals/12345 -H 'Content-type:application/json'
```

The project also provides an implementation for the same operations but using a Hypermedia-driven RESTful service architecture. To call those operations remotely, run:

```
$ curl -v -X GET localhost:8080/rest/meals
$ curl -v -X GET localhost:8080/rest/meals/4237681a-441f-47fc-a747-8e0169bacea1
```

[5] You can find more information about HTTP status codes at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`

Since we rely on the HTTP protocol, the alternative option to try out the GET operations is to simply use a browser. If you open up your browser and browse for `http://localhost:8080/rest/meals`, you will see the list of the meals. The core idea of the RESTful architectural style for remote APIs is to mimic how the Web works. Normally when you fetch a web page, it is likely that it contains links to other pages or server-side functionalities. This property makes the Web discoverable, and most importantly your browser (the client) is not coupled to the evolution of the web page.

As you can see, a new field called *_links* has been added to the response. Upon each remote function invocation over HTTP, true RESTful APIs should reflect hyperlinks related to the concerned resource or operation, considering the application state.

4.4 Requested extensions

In this assignment, you will need to implement the following extensions to your application.

Preliminary remarks: It is important to take the following points into account:

1. *Two REST controllers*: you need to implement the below functionalities for both the Level 2 REST controller (in *MealsRestRpcStyleController.java*) and the Level 3 REST controller (in *MealsRestController.java*).
2. *HTTP response status codes*: your REST API should return an appropriate response status code for each API call.
3. *HTTP verbs*: your REST API should make use of an appropriate HTTP verb for each API call on resources.
4. *Hypermedia-driven API*: your REST API should reflect the appropriate set of URIs for each resource.

Searching meals that match a constraint. You should add two functions, for getting the *cheapest* and the *largest* meal, to both the *MealsRestRpcStyleController* and *MealsRestController* controllers.

Adding/updating/deleting a meal. You need to add 3 remote operations to your REST controllers to support (1) meal addition, (2) meal update, and (3) meal deletion. For the addition and update, the meal object should be transmitted in the HTTP body. Make sure you use the right HTTP verbs and status codes. In the *MealsRestRpcStyleController* it is sufficient to return HTTP Status code 200 in case of success and HTTP Status code 404 in case the meal is not found. In *MealsRestController* you can also return HTTP Status code 201 when a new meal is created by using `ResponseEntity.created`, which takes a URI pointing to the newly created meal as a parameter, after which you can add the body to the response.

Measuring the delay of remote operations. We will now run the server and test client on two different machines in the PC labs.

1. Run the application server on your local machine that you are working on and leave it running.
2. Open a new terminal window and ssh to a different machine in the PC labs (see the list at `https://mysql.student.cs.kuleuven.be/`) and go to your project directory.
3. Execute the client requests again on the remote client machine, but use your own machine's name instead of localhost for the server. For example, assuming you are working on Aalst and are using Gent as remote client machine:

   ```
   gent$ curl -v -X GET http://aalst.student.cs.kuleuven.be:8080/rest/meals
   ```

4. You can get additional information about the timing of your request using the `-w` option of curl. What is the response time of the remote REST operations? What is the response time when testing locally?

   ```
   gent$ curl -v -X GET http://aalst.student.cs.kuleuven.be:8080/rest/meals
              -w "\n time:  %{time_total}\n"
   ```

4.5 OpenAPI Specification

As mentioned in the introduction, we will look at two different approaches to implement a RESTful service. Until now, you used a *code-first* approach, where you just wrote some code, but have no documentation and specification of your newly created interface yet. Now, we will build a standardized specification and the accompanying documentation, starting from the code. We will use OpenAPI[6] as the specification standard, as it is widely used and supported by most tools.

The Maven configuration contains a dependency to `springdoc-openapi`[7], a library that examines the application at runtime and automatically generates an OpenAPI specification based on the Spring configuration, class structure, and various code annotations.

You can consult the specification by executing the following commands. The first returns the specification in JSON format, the latter in YAML.

```
$ curl http://localhost:8080/v3/api-docs
$ curl http://localhost:8080/v3/api-docs.yaml
```

SpringDoc also supports Swagger UI, which generates a user interface based on the OpenAPI specification that allows you to explore the API interface in a user-friendly way and even interact with it. To access the Swagger UI, open a browser and browse to `http://localhost:8080/swagger-ui/index.html`. At the top of the page, you will see the two controllers with all their implemented service endpoints. At the bottom, you will get more information on the domain model of the RESTful services.

Additional questions. The following additional questions will ask you to test the remote service a bit further with the help of the Swagger UI.

1. Did you notice one of the methods of the *MealsRestController* has extra information? Have a look at the implementation and annotations of this method to understand how this was done.
2. Try getting a meal with id "12345" for both controllers via the Swagger UI interface. Notice the difference? Swagger UI used the OpenAPI specification to validate the format before sending the request.
3. Does the same request as in the previous step work if you execute it via curl?
4. When you open a PUT request in Swagger UI, you will notice that you have to enter an `id` as path parameter, but the example request body also contains an id. If you change one but not the other, which one will be used as the id to fetch the meal?

Some remarks. As you will have noticed, OpenAPI can be used to document your REST services, and your API consumers can get a lot of information on how to use your services. What we saw here was just the tip of the iceberg. You can include example requests, security requirements and much more. Fully documenting the meal service is not the goal of this exercise, but is well worth the effort when you are in a real environment.

While testing the service, you will have noticed that Swagger UI can do some extra validation based on the specification, but if you send an invalid request directly to the backend, it will still get accepted and processed. The server should never expect that the client (such as Swagger UI) has done the validation, and should always do additional validation checks!

The final question of the additional questions made clear that you should think about your data model to avoid inconsistencies. Normally, you want to either include the id in the body or in the path, but not both. We will have a look at this in the next section.

[6] `https://swagger.io/specification/`
[7] `https://springdoc.org/`

5 Part 2: API-first remote services using OpenAPI

In this second part, we will go in the opposite direction. We will start from an OpenAPI specification and then use a code generator to generate scaffolding code for the server that provides an API according to that specification – hence the name *API-first*. As in the previous part, a ready-to-use and ready-to-run software project is provided to get started. This allows you to experience how we can implement a similar RESTful service (Level 2). We will also extend the OpenAPI specification with the `Order` concept mentioned in the introduction.

5.1 The provided software project

Additional Tools: OpenAPI Generator. We will use OpenAPI Generator[8], which can be used to generate a client, a server, and the API documentation based on an OpenAPI specification.

The necessary dependencies and configurations have been added to the Maven `pom.xml`. Note that there are many configuration options possible for OpenAPI Generator. We opted to limit the generation to the API interface. The advantage of this is that you can extend the OpenAPI specification and then easily regenerate (only) the code specific to the API interface. If we want to add the Java project to a version control system like GitHub, we only have to check in the OpenAPI specification and our own files, not the generated files. The alternative would be to generate all server classes (including the controllers) one time and modify these classes. However, if ever we would want to add new endpoints to the OpenAPI specification, we would not be able to easily regenerate these classes as this would erase any changes that were made in the meantime.

Conceptual and functional content. The provided project includes:

- The definition of the `Meal` concept with its key properties, but here they are specified as an OpenAPI specification.
- The definition (OpenAPI specification) and implementation of remote operations related to the `Meal` concept under the `/meals` endpoint.

The implementation of these functional concepts. The implementation of the RESTful service is defined in the following software artifacts, residing in `src/main/java`. You will notice that some classes are missing (and IntelliJ will complain about it for now), for example, the `Meal` class. These will be generated later.

In the `be.kuleuven.foodrestservice.domain` package:

- **MealsRepository.java** is a Spring component that contains the core application logic. It initializes a set of meals and implements the functions for the meals-related operations.

In the `be.kuleuven.foodrestservice.controller` package:

- **MealsController.java** is the actual REST controller that will implement the API interface that will be generated. We will restrict ourselves to a Level 2 implementation of the services, but will use *ResponseEntity* to have more control over the HTTP status codes.

In the `be.kuleuven.foodrestservice.exceptions` package:

- **MealNotFoundException.java** and **MealNotFoundAdvice.java** define an exception that can be used by the REST server. This exception triggers the server to use HTTP Status 404 (not found) when a requested meal does not exist.

Other project artifacts. In addition to the **pom.xml** as in the previous project, here we start with **/resources/static/openapi.yaml**, which is the OpenAPI specification we will implement. The file is placed in the `static` subfolder, as this allows us to use the file in Swagger UI directly, instead of generating the specification on the fly.

[8] https://openapi-generator.tech/

5.2 Generating code and running the server

Similar to the first part, we can use the `spring-boot:run` command in the Maven Plugins section to run the server. This will read the OpenAPI specification file(s), generate the classes based upon them, compile everything, and finally start the Spring application.

For IntelliJ to detect this generated code, right-click on the project and choose `maven > Generate Sources and Update Folders`. You can use the Maven `clean` command between runs if you want to remove all previously generated files, so you can start from a clean slate.

Inspect the generated classes. First, have a look at the newly generated files, which can all be found under the `/target/generated-sources` folder. Of special interest to us are the following packages:

- `be.kuleuven.foodrestservice.api` contains the **MealsAPI** interface. When you open it, you can see that all methods specified in the `openapi.yaml` specification are defined here, but will return a `NOT_IMPLEMENTED` status code.
- `be.kuleuven.foodrestservice.model` contains the class definition of the `Meal` concept. Note that in this implementation, we defined two different schemas for this concept in `openapi.yaml`. **Meal.java** differs from **MealUpdateRequest.java** in the `id` property. As suggested in the first part, when we create a new `Meal` or update an existing one, we do not want to provide an `id` as this will either be generated or is already provided as path variable in the endpoint.

You will notice that the generated files are not necessarily identical to the ones we created ourselves. Some things might be a bit more complex than we actually need, but this is worth it given the ease of modifying a YAML file and regenerating the classes, which we will experience soon ourselves when we extend the interface. On the other hand, a generator might already apply some best practices for you. Did you notice the `Meal` now has an `id` of type `UUID`? Our previous specification did specify that the `id` should have the format of a UUID but did not actually enforce it.

Test the server with Swagger UI. Open a browser and go to `http://localhost:8080/swagger-ui/index.html`. Test the service by adding a new meal or deleting a meal.

- What happens when you update a meal that does not exist?
- What happens when you send invalid data?
- Did you notice a `Meal` has some required fields in the specification?

5.3 Requested extensions

In this extension, you will need to extend the RESTful service with the `Order` concept.

- Extend **openapi.yaml** with a schema for the `Order`. As specified in the introduction, an order contains an address and a list of meals (names or id's, it is your choice).
- Extend **openapi.yaml** with an operation to add a new `Order` under a different endpoint `/orders`. This operation should return your order with its `id` as confirmation. Also, add an operation to retrieve all orders.
- Generate the code by running the application again: `spring-boot:run`.
- Notice that as this is a different endpoint, a new API class was generated. Create a new controller to implement this interface and expand the `MealRepository` with the new logic.
- Run the server again and test the new endpoints.