Remote communication: SOAP/REST

Service-oriented architectures for remote services using SOAP and REST

---

1 Introduction

In a service oriented architecture, applications are constructed based on reusable online services that support interoperable machine-to-machine interactions over a network. Service oriented architectures focus on interoperability and are not dependent on a particular programming model or language. Interactions and information are exchanged using a machine-processable format such as XML or JSON. As a transport layer, often HTTP is used. 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.

During this session we will focus on two approaches to construct online services: web services using SOAP, and RESTful services.

Goal: Your goal is to learn and practice the development and remote communication of online services that are either SOAP-based or RESTful. Your tasks in this assignment are the development and testing of two remote services for both SOAP-based as well as RESTful communication.

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.

Opening the start code as IntelliJ project. All tools that you need should already be installed in your Ubuntu Desktop VM, assuming you did your homework. We will use IntelliJ as well as some command line tools such as *java*, *curl* and *maven*.

For this lab session, there are two zip files: one for SOAP, and one for REST. Unzip a file you have downloaded from Toledo, and open the top-level folder to create an IntelliJ project (via `File >`). 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 two 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 SOAP-based and RESTful remote services. These

[1] `https://spring.io/projects/spring-boot`

services need to be developed in Java using the Spring API. The task of the application server is to listen for incoming HTTP requests, unmarshall and parse them, and call the right application-level operation in your SOAP-based service or 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. Install curl on your Ubuntu Desktop VM if you didn't do so yet.

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. The most important commands that you will need are:

- `mvn clean package`
- `mvn spring-boot:run`

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 short unique 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:

- `Meal getMeal(String name)` returning a meal with all its properties.
- `Meal getCheapestMeal()` returning the currently cheapest meal on offer.
- `Meal getLargestMeal()` returning the currently most energetic meal on offer.
- `OrderConfirmation addOrder(Order order)` to put in an order for one or more meals that need to be delivered at the given address in the order. You are free to define what properties are included in an `OrderConfirmation`.

4 Part 1: A HTTP-based remote service using SOAP

To get you started quickly and swiftly we provide a ready-to-use and ready-to-run software project you can start from, including the minimal code base of a SOAP-based online service, with implementations for *getMeal* and *getCheapestMeal*. We briefly discuss the contents of this provided software project.

4.1 The provided software project

Conceptual and functional content. The provided project only includes:

- The definition of the `Meal` concept with its key properties: name, description, kcal, and meal type.

[2] `https://curl.se/`
[3] `https://maven.apache.org/`

- The definition and implementation of two remote operations:
  1. `Meal getMeal(String name)`
  2. `Meal getLargestMeal()`

The implementation of these functional concepts. The implementation of the SOAP-based service is defined in the following software artifacts residing in `src/main/resources/` and `src/main/java/`:

- **meals.xsd** is an XML-schema that defines the types of the different concepts that are present in the online interface of the SOAP service:
  - The *Meal* concept and its different elements
  - The *types of meals* (Fish, Meat, Veggie and Vegan).
  - *GetMealRequest* and *GetMealResponse* for the incoming request and outgoing reply of the getMeal operation.
  - *GetLargestMealRequest* and *GetLargestMealResponse* for the getLargestMeal operation.

  The maven plugin `jaxb` will generate and compile the appropriate Java classes based on this XML schema file. You can then use these classes in your code. If you change this XML schema file, you will have to run `mvn package` again to recompile it. The source code of these classes is in `target/generated-sources/jaxb/io/foodmenu/gt/webservice`, but you don't need to access or edit this generated code.
- **MealRepository.java** is a Spring component that contains the core application logic of the Meal service. It initialises a set of meals and implements the key application operations to retrieve a meal by name or to find the largest meal.
- **MenuEndpoint.java** is *the main implementation of the SOAP service itself*, and is responsible for mapping the incoming SOAP requests to the right operation. The operation types for the incoming requests as created in the XML-schema above, e.g. GetMealRequest, are mapped to a specific operation using the RequestPayload annotation. The operation must return the appropriate type as ResponsePayload (e.g. GetMealResponse). In this class, both getMeal and getLargestMeal are implemented using this approach.
- **SpringSoapApplication.java** is the entry point of your Spring Boot application when it is started as a server process.
- **WebServiceConfig.java** contains configuration code for your SOAP-based webservice, for example:
  - to generate the WSDL from the meals.xsd schema.
  - to expose the SOAP-based service on a certain sub-URL (e.g. /ws/meals)
  - to configure the right message dispatcher servlet to handle incoming requests for SOAP-based services.

  You don't need to change this file during the assignment.

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

- **request.xml** and **requestbiggest.xml** which contain two example SOAP-based requests with their headers and envelop. These will be sent from the client to the remote SOAP service using *curl* to test our SOAP service.
- **pom.xml** which is the Maven file that defines all the used dependencies and required plugins of the project. You don't need to edit this or change this during this assignment.

4.2 Running the server and client

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 SOAP service.

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

```
$ mvn spring-boot:run
```

In the terminal window of IntelliJ you can now execute some client-side requests to get information about a meal, or to get the biggest meal in terms of energetic value.

```
$ curl --header "content-type: text/xml" -d @request.xml http://localhost:8080/ws
$ curl --header "content-type: text/xml" -d @requestbiggest.xml http://localhost:8080/ws
```

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

In some rare cases, IntelliJ is not able to find the source code with the classes generated from the XML schema. If you are noticing this problem, where the generated classes are not found:

1. Right click project folder
2. Select Maven
3. Select Generate Sources And Update Folders

Then, IntelliJ automatically imports generated sources to the project.

4.3 Requested extensions

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

Getting the cheapest meal. First you will need to add a price to meals. Edit the meal concept in the meals.xsd and also add the request and reply type for the getCheapestMeal operation, analogous to the getLargestMeal operation. Provide implementations for this operation in both the MealRepository component and MenuEndpoint, again analogous to the getLargestMeal operation.

Orders. You need to extend the SOAP service with the AddOrder operation as described earlier in the introduction. This will require extensions to the meals.xsd to define the Order concept. You will also need to add the request and response schema for the AddOrder operation. Furthermore you will need to implement the core application logic of this operation in the MealRepository component, and you will need to handle the incoming operation in the MenuEndpoint.

Running and testing your SOAP service. To test your application locally on your machine, you can use the same commands as above to run the application server and testing it with curl. Create a custom request XML-file for the different operations you are testing (e.g. requestcheapest.xml and requestorder.xml).

Additional questions. The following additional questions will ask you to test the remote service a bit further regarding the HTTP-based SOAP communication.

1. Which HTTP verb (method) is used by SOAP for the request to get the cheapest meal? Which HTTP verb is used to add the order? You can use the `-v` option of curl to get more verbose information about the request you make.
2. When making an order, what is the content-length and content-type of the HTTP request and response?
3. If you ask for a non-existing meal, or a non-existing SOAP operation in the request, which HTTP response code is returned for each of those erroneous requests?
4. Using the command below, you can ask the service for its WSDL, which specifies the interface of the SOAP service and how to communicate with the service. Which concepts do you find in this WSDL in addition to what you specified in the meals.xsd? What is the SOAP binding style and transport configuration?

   ```
   $ curl http://localhost:8080/ws/meals.wsdl
   ```

   You can also open this URL in your browser, to see the XML nicely formatted.

5 Part 2: Remote services using REST

In this section, your goal is to re-implement the food delivery application using a RESTful service architecture. To do this swiftly, we provide a ready-to-use and ready-to-run software project, similar to the SOAP project from Section 4. The project is not fully implemented; however, the REST operations to fetch all meals (`/rest/meals`) and fetch a single meal (`/rest/meals/{id}`) are implemented in the provided source code. 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`). In industry, most software services are implemented using this approach.
2. *True RESTful service architecture*: Next, our goal is to implement the same operations but in true RESTful service architecture. These types of services are hypermedia-driven. We will discuss what we mean by this in the rest of the assignment.

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.

5.1 The provided software project

Conceptual and functional content. Conceptually, the majority of the project content is similar to the SOAP project. The provided project only includes:

- The definition of the `Meal` concept with its key properties similar to the previous project. In this part, each meal has a unique identifier *id*.
- 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 implementation of the RESTful service is 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) with the same properties as listed in the SOAP section.
- **MealsRepository.java** is a Spring component that contains the core application logic, almost the same as the previous assignment. It initialises 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:

- **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 return types of the operations are of *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 (if you use *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 return HTTP Status 404 (not found) when a requested meal does not exist.

Other project artifacts are similar to the previous project. However, there is no XML schema definition for the REST API.

5.2 Running the RESTful server

You should run almost the same steps as in the SOAP project because this project is also a Maven-based Spring project. First, make sure that you have a correct Java version (17) installed and configured. To run the server, execute the `spring-boot:run` Maven goal.

To test the currently implemented REST API, we can again use the `curl` command-line tool.

```
$ curl -X GET localhost:8080/restrpc/meals  -H 'Content-type:application/json'
$ curl -X GET localhost:8080/restrpc/meals/4237681a-441f-47fc-a747-8e0169bacea1
       -H 'Content-type:application/json'
```

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 use the `-v` option, you will get more verbose responses. For example, you can see that the HTTP response comes with the 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 [4]. If you want to look up an imaginary food that does not exist in the system or 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 if you copy/paste the following commands on your terminal, there is a space between each segments 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  -H 'Content-type:application/json'
$ curl -v -X GET localhost:8080/rest/meals/4237681a-441f-47fc-a747-8e0169bacea1
       -H 'Content-type:application/json'
```

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 `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 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. We will explain this feature in the next section.

5.3 The *Richardson maturity model* for REST

Leonard Richardson presented a model [5], known as the *Richardson maturity model*, that breaks down the RESTful APIs into 3 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. For example, SOAP typically uses HTTP to contact a single predefined endpoint, and the requests are passed through the message bodies.
- *Level 1 - Resources:* In this category of APIs, an API is not a single endpoint with a single URI, but it is based on the concept of a collection of resources. This means that such APIs provide multiple URIs to address multiple 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.

[4] You can find more information about the codes at https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
[5] https://martinfowler.com/articles/richardsonMaturityModel.html

- *Level 2 - HTTP Verbs:* This category of services rely on the concept of resources (URIs) and most 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 currently 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.

5.4 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.

Getting the cheapest and the largest meal. You should add these functions 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 update and addition, the meal object should be transmitted in the HTTP body. Make sure you use the right HTTP verbs and status codes.

Ordering meals. Analogously to the SOAP web service, you should be able to order a meal, by providing an address and a list of meal identifiers. Design and implement this in both REST controllers, and provide test scripts for both.

6 Packaging, deploying and testing your services

The Maven build process creates a jar file that packages your Spring Boot application as a stand-alone application. In one of the next sessions we will deploy and run these jar files on a remote server in the cloud via a commandline interface.

To prepare for this next session, copy the jar files to a new, separate, empty deployment folder, and try running these jar files from the commandline. Also try to run the tests (curl and shell scripts) from a different commandline to test everything.