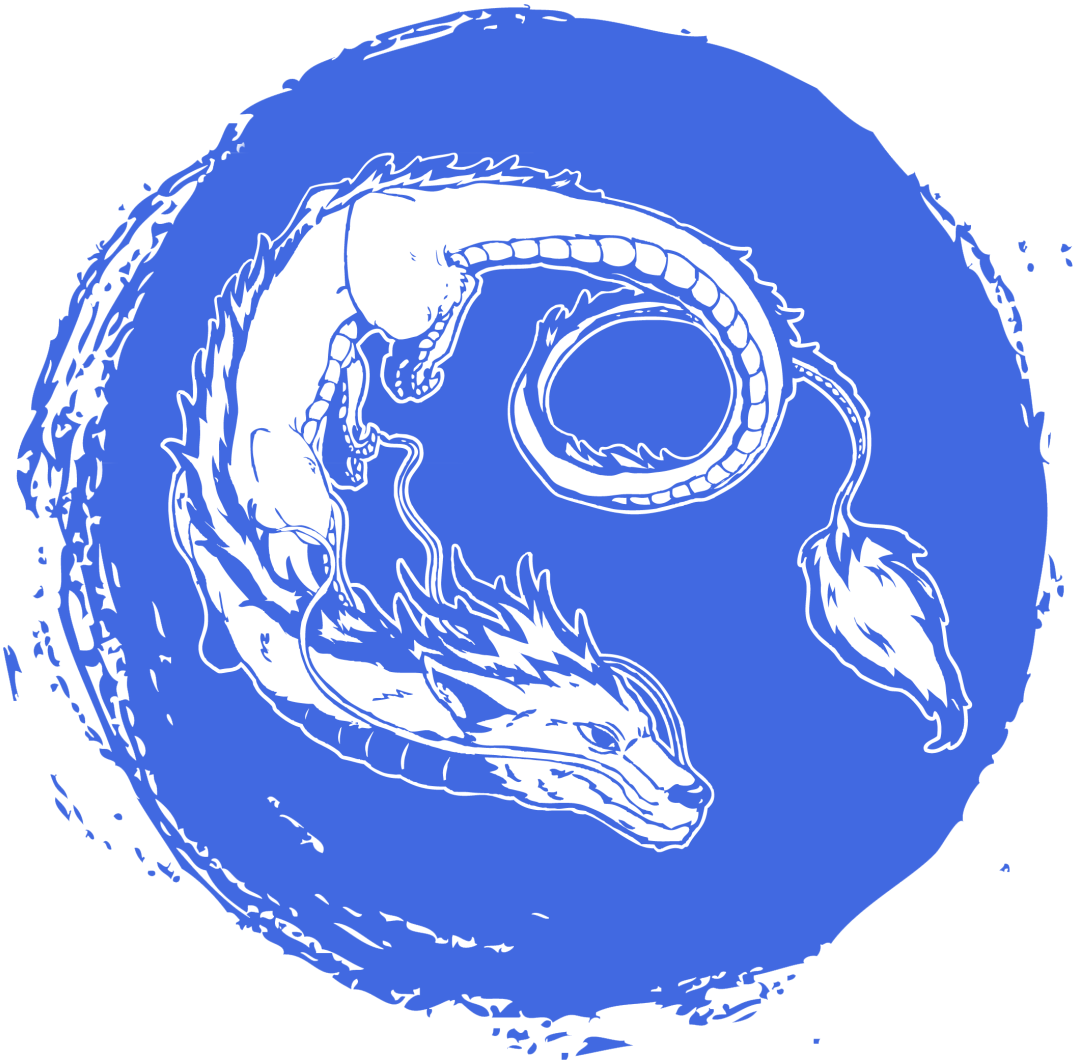
Webservice — Subject

version **#v2.4.3**



Yet Another Kind of Assistants 2022 <yaka@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: The coding-style needs to be respected at all times.

Obligation #6: **Global variables** are forbidden, unless they are **explicitly** authorized

Obligation #7: Anything that is not **explicitly** allowed is **disallowed**.

Advice

- Read the *whole* subject.
- If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **newsgroup** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- In examples, `42sh$` is our prompt: use it as a reference point.
- Do **not** wait for the last minute to start your project!

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

Files to submit:

- ./src/main/java/com/epita/assistants/yakamon/*

Provided files:

- ./pom.xml
- ./src/main/java/com/epita/assistants/yakamon/arch/Controller.java
- ./src/main/java/com/epita/assistants/yakamon/arch/DtoRequest.java
- ./src/main/java/com/epita/assistants/yakamon/arch/DtoResponse.java
- ./src/main/java/com/epita/assistants/yakamon/arch/Entity.java
- ./src/main/java/com/epita/assistants/yakamon/arch/Model.java
- ./src/main/java/com/epita/assistants/yakamon/arch/Repository.java
- ./src/main/java/com/epita/assistants/yakamon/arch/Service.java
- ./src/main/java/com/epita/assistants/yakamon/arch/TestSuite.java
- ./src/main/java/com/epita/assistants/yakamon/Yakamon.java
- ./src/test/java/com/epita/assistants/yakamon/arch/DtoTest.java
- ./src/test/java/com/epita/assistants/yakamon/arch/EntityTest.java
- ./src/test/java/com/epita/assistants/yakamon/arch/LayersTest.java
- ./src/test/java/com/epita/assistants/yakamon/arch/ModelTest.java
- ./src/main/resources/ddl.sql
- ./src/main/resources/logback.xml

1 Introduction

1.1 The Yakamon World

This wonderful world gathers numerous species of Yakamons, mystic creatures.

Each Yakamon has an area of expertise: some are able to create subjects, others to do perms. They can achieve specific actions, review a subject or help students. The description of each and every species and their evolutions can be found in the precious encyclopedia of this universe, the Yakadex.

Alongside Yakamons, Trainers also live in this world. Their goal is to capture as many Yakamons as possible. The more powerful, the better!

Yet, finding Yakamons is not that easy. Our world is divided into zones, and not every zone contains Yakamons. The Zones chapter of the Yakadex details where each Yakamon species lives.

A Yakamon can only be found in its habitat, as described before, or in the yakaball of a Trainer. When a Yakamon is created or freed by its Trainer, the zone where it appears must be a valid one.

1.2 Goal

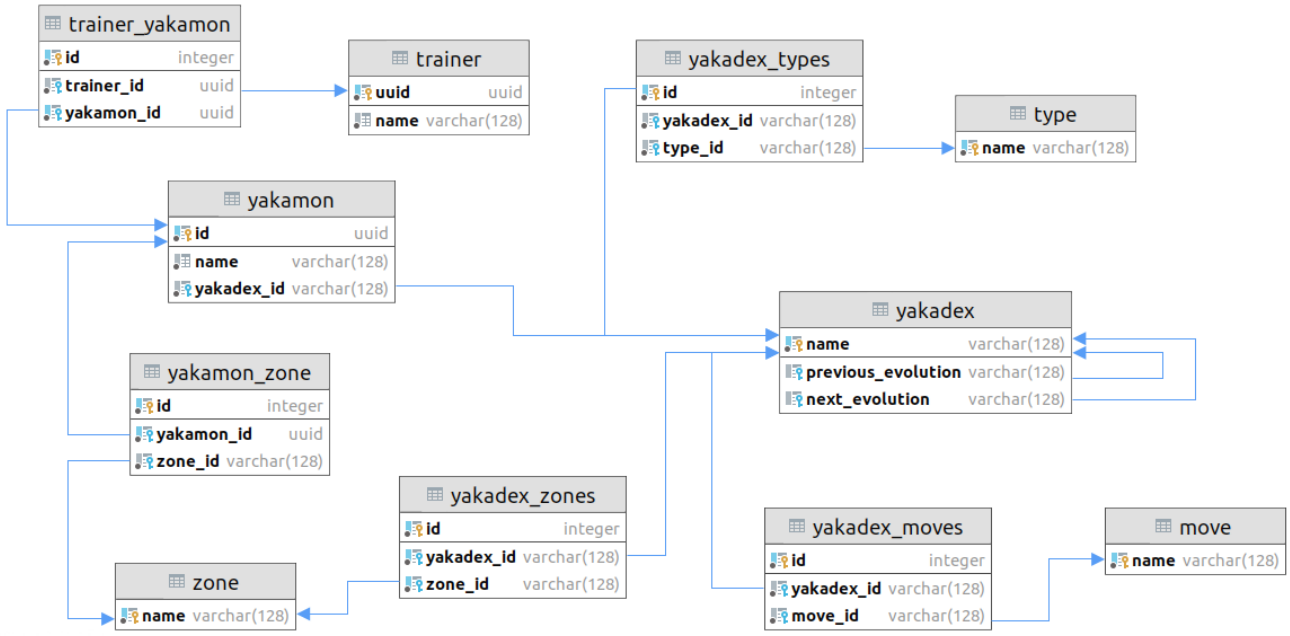
The goal of this project is to create the backend of a Yakamon-related app. You will need to implement multiple REST endpoints in order for external applications to communicate with your backend.

Those endpoints are all defined in the *JSON* file available on the intranet.

Your code will be tested in two different ways:

- Architectural tests: the architecture of your project will be tested. Therefore, it should comply with the *Architecture* section of the subject. Do not forget to use annotations!
- API tests: your backend should send proper responses to external requests. Those requests and responses are all defined in the *JSON* in the *REST-paths* section.

Here is a diagram that resumes the relationships between the different components of this universe, that you will find in the given database:



1.3 RESTful WebService / API

Application Programming Interfaces (API) provide a way to communicate and interact with softwares. An API could be a library or, in our case, a webservice.

REpresentational State Transfer (REST) is an architectural style introduced by Roy Fielding in this paper.

Nowadays, most APIs you will be able to interact with are REST APIs. A REST API (or RESTful WebService) is a kind of API with which we interact using REST calls. It follows the REST constraints and have the following features:

- It has a base URI. In our case, it is "http://127.0.0.1:<port>/" but it could have been "http://127.0.0.1:<port>/yakamon" if we had multiple webservices.
- It uses standard HTTP methods like `GET`, `PUT`, `DELETE`...

  HTTP defines a set of methods, also referred to as "verbs", to indicate the desired action to be performed on the identified resource. The most used request methods are[2]:

  - `GET`: to retrieve information;
  - `POST`: to create information;
  - `PUT`: to replace information;
  - `DELETE`: to delete information.
- It uses media types that is common to the vast majority of rest-points. This is used to have a cohesion between all rest-points and to ease the use of the API. In this project we will be using application/json on all of the rest-points.

[2] However, some APIs do not work that way. For example the Zabbix API does everything using POST: the Request body is the one telling the server what to do.

A **REST endpoint** is a path starting from the base URI where you can get a response from the server. It can return an object carrying data (DTO) or simply an HTTP response code.

In this project, the path below is a valid REST endpoint:

```
GET 127.0.0.1:4567/yakamon/e4d08489-68d5-4378-aecd-05e2681f9801
```

1.4 Layered Architecture

Here is a link to a course you **must** watch. It explains the architecture of the project in details followed by a live coding. We may add more video content to the channel if necessary.

2 Libraries

For this project you will be using two different libraries: jOOQ and Spark.

2.1 Jooq

You will use jOOQ (Java Object Oriented Query) to interact with your database. At compile time in the maven lifecycle, jOOQ will execute the given *SQL* file into a H2 database (an in-memory database used for this project) and generate classes related to the table.

Be careful!

Do not forget to run "`compile`" in the Maven Lifecycle if you want to be able to generate the files, to connect to the database and use it.

Tips

H2 is an in-memory database, that means that you do not have to run any server. The database will be loaded with *Java* at runtime.

Do not worry about finding out how to configure jOOQ or H2 in the `pom.xml`, as it is provided as usual. In order to connect to the database at runtime, a `setupConnection()` method is given to you in the file `Yakamon.java`.

You just have to use the **maven compile lifecycle**, it will create a folder named *ddl* containing classes representing your database tables. You will use these classes to communicate with your database in your repositories.

We strongly advise you to read the jOOQ documentation, especially the "Getting Started" part.

Tips

You should use static imports of your generated tables.

2.2 Spark

Spark is a library with which you will bind your endpoints to your Controllers' methods. Check out the Spark documentation for further information.

Be careful!

Be careful! Spark Java should be used, not Apache Spark!

3 Architecture

Following the right architectural pattern is the main goal of this project. It will be a useful knowledge if you have to work on the backend of an intranet or an application during your internship for example.

Your project should contain at least three packages inside the `yakamon` package:

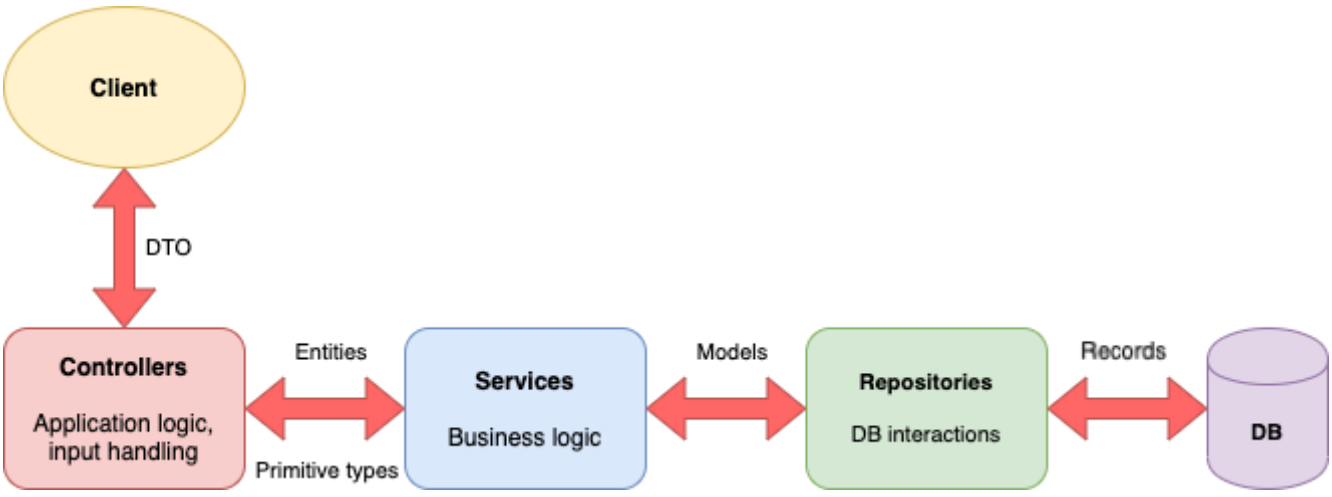
- the `controller` package;
- the `service` package;
- the `repository` package.

You are free to implement `Models`, `Entities` and `DTOs` in other packages.

Be careful!

You **MUST** implement all your classes in the `com.epita.assistants.yakamon` folder.

3.1 The Layered Architecture



This is commonly called the "layered architecture", and is a staple of backend development. You may find implementations across a wide variety of tools that follow the pattern with different terms for each layer and associated object type. This lack of standardization in naming should not be an obstacle to your comprehension of the pattern and your ability to recognize it.

3.1.1 Controllers

The **Controllers** are the doors for external applications to interact with your backend. The Controllers' goals are to:

- expose methods used by Spark in order to create endpoints;
- tell the client if something went wrong with the request;
- get the result from a service and send a response to the user.

These responses and requests are objects called **Data Transfer Objects**. For more information related to Request and Responses, please read the *API* section of this subject.

Be careful!

Your Controllers should **never** be able to interact with Models.

3.1.2 Services

The Services are the heart of your server, it is where all the logic happens. Although Services are usually used to manage a specific entity, it is not always bound to only one Repository since Entities may need information from different Models.

3.1.3 Repositories

The Repositories of your server are the part where you will implement all database-related methods, like retrieving data from your database for example.

Be careful!

You should not implement any logic in the Repositories. They only aim at interacting with the database and retrieving the needed data. Therefore you **must** have at most one Repository per table.

3.1.4 Data Transfer Objects

DTOs are the objects managed by the Controllers.

There are two types of DTOs:

- Response DTOs;
- Request DTOs.

They should be seen only in the Controllers. They are contracts the client must respect in their requests and that the API must respect for its responses.

One of the main reasons DTOs exist, is to separate presentation logic from business logic. You would not want to update the whole API for a simple service layer change. Decoupling allows for business update without the need to tell all your clients you mutated your API for internal reasons (of course, API changes for sound reasons should be made and communicated properly).

Therefore, your API should deserialize the request body into a Request DTO and return a serialization of a Response DTO.

Tips

To serialize/deserialize your DTOs, use the Jackson library provided in the `pom.xml`.

3.1.5 Entities

Entities are value objects managed by the Services. They are only seen in the Services and Controllers. They allow communication between the Controllers and the Services.

In many cases, especially in the early days of a project, Entities are similar to underlying Models. However, some Services are aggregation services, their responsibility will be the computation of aggregated data (statistics, processing, presentation...). Moreover, even "mapping" Services will diverge from their Model in time to accommodate business needs, justifying fully the layer separation.

Tips

Some Entities look a lot like their mapping Models, but not all services are mapping Services, and even mapping Services deviate from their Model in time.

3.1.6 Models

The Models are the objects managed by the Repositories. It should never be seen outside of Services and Repositories. They allow communication between the Services and the Repositories. A Model is the object representation of a database entry.

3.2 Given annotations

In order for us to test your architecture and still give you some freedom over your implementation, you should annotate your classes with the corresponding annotation:

- `Controller`
- `DtoRequest`
- `DtoResponse`
- `Entity`
- `Model`
- `Repository`
- `Service`

Be careful!

You **MUST** use those annotations, otherwise you will not get any points for your architecture.

Tips

You will find some tests in the given files. These tests are Architectural Tests, they allow you to see if your architecture is compliant with our requirements.

4 API

4.1 Response

Responses must have the `Content-type` header set to "application/json", regardless of the client's request.

A Response object must always be a *JSON* object containing a status: "SUCCESS" or "FAILURE".

```
{
  "status": "SUCCESS",
  "errorCode": null,
  "errorMessage": null,
}
```

In the above example, the status is SUCCESS, but the backend responded with empty errorCode and errorMessage. It is equivalent to the following *JSON*:

```
{
  "status": "SUCCESS"
}
```

4.2 ErrorHandling

Creating a client can be hard sometimes, and if something goes wrong, not knowing what happened in the backend can make it difficult to understand. This is the reason why your backend should send error codes if something went wrong.

Here are the error codes (with their respective messages) your backend should be able to send to the client:

- `errorCode: "OBJECT_NOT_FOUND", errorMessage: "Object not found"`

  It should be sent when the requested object was not found during a `GET` request.

- `errorCode: "DELETION_FAILED", errorMessage: "Deletion failed"`

  It should be sent when the deletion of an object did not happen during a `DELETE` request.

- `errorCode: "UPDATE_FAILED", errorMessage: "Update failed"`

  It should be sent when the update of an object failed during a `PATCH` request.

- `errorCode: "CREATION_FAILED", errorMessage: "Object creation failed"`

  It should be sent when the creation of an object failed during a `POST` or `PUT` request

- `errorCode: "BAD_PARAMETER", errorMessage: "Bad parameter"`

  It should be sent when the parameter of the path is not formatted correctly (e.g.: the id is a `String` instead of an `Integer`).

- `errorCode: "BAD_BODY", errorMessage: "Bad body"`

  It should be sent when the body of the request is not correct (e.g.: the request object is not the good one or compulsory fields are empty).

You do not have to send multiple error codes. Instead, you should send error codes regarding this priority:

1. BAD_PARAMETER
2. BAD_BODY
3. OBJECT_NOT_FOUND | CREATION_FAILED | DELETION_FAILED | UPDATE_FAILED

Any FAILURE response should contain at least the following fields:

- `status: "FAILURE"`
- `errorCode`: the error code ("BAD_PARAMETER", "BAD_BODY"...)
- `errorMessage`: the error message related to the error code.

4.3 Example of Use

In this example, we will try to get a list of Zones (those Zones are not the same as the ones used in the moulinette or the given file. Do not worry if you do not have the same return value).

Request:

```
GET http://127.0.0.1:4567/zone/
Accept: application/json
```

Response:

```
{
  "status": "SUCCESS",
  "zones": [
    {
      "name": "Cisco"
    },
    {
      "name": "Lab YAKA"
    },
    {
      "name": "LabSR"
    },
    {
      "name": "MidLab"
    },
    {
      "name": "Pasteur"
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "name": "SM14"
    }
  ]
}
```

Once we have the list of Zones, we would like to get the information regarding the Lab YAKA.

Request:

```
GET http://127.0.0.1:4567/zone/Labyaka
Accept: application/json
```

Response:

```
{
  "status": "FAILURE",
  "errorCode": "OBJECT_NOT_FOUND",
  "errorMessage": "Object not found",
  "name": null
}
```

As you can see, since Labyaka is not in an id contained in the Zone list, no object has been found and the backend responded with a FAILURE.

Request:

```
GET http://127.0.0.1:4567/zone/Lab%20YAKA
Accept: application/json
```

Response:

```
{
  "status": "SUCCESS",
  "name": "Lab YAKA"
}
```

Above is a SUCCESSfull response to the request we failed earlier.

4.4 REST paths

The JSON “rest-paths.json” on the intranet describes the REST paths you should implement in order for external applications (e.g., the moulinette) to interact with your backend.

Tips

Do not be intimidated by the number of endpoints, most of them are simple GETs, take them one by one.

In order to help you understand how to read the *JSON*, let’s take this endpoint:

```
{
  "basePath": "/trainer",
  "endpoints" : [
    {
      "method" : "GET",
      "path" : "/",
      "description" : "Get a list of the Trainers.",
      "parameters" : [ ],
      "request" : null,
      "response" : {
        "typeName" : "Response Object (DTO)",
        "fields" : [
          {
            "name" : "trainers",
            "type" : {
            "typeName" : "List<ELEMENT_TYPE>",
              "internalTypes" : {
                "ELEMENT_TYPE" : {
                  "typeName" : "Object",
                  "fields" : [
                    {
                      "name" : "uuid",
                      "type" : "UUID"
                    },
                    {
                      "name" : "name",
                      "type" : "String"
                    }
                  ]
                }
              }
            }
          }
        ]
      }
    },
    {
      "..."
    }
  ]
}
```

Those are the information we can take from this part of the *JSON* file:

The path is “/trainer”, it can be read from the “basePath” field.

It has at least one endpoint, beginning by “/trainer”.

This endpoint answers the “GET /trainer/” request (we can know that from the “method” and “path” fields). The “description” field tells us that this endpoint’s Response is a list of Trainers.

This endpoint takes no parameters. If it did, the “path” field would have contained “:id” for example in it and the “parameters” field would have been

```
{
  "name": ":id",
  "type": "STRING"
}
```

This endpoint takes no Request Object DTO since the “request” field is null. If it was not null, it would have the same format as the “response” field.

The “request” field can be null unlike the “response” field. The “response” field is a Response Object DTO (which makes sense since the Controllers should return DTOs). This DTO has only one field (found in the “field” field):

- a field named “trainers”

The “trainers” field’s type is a `List` of `Objects`. However, since saying `List<Object>` is not quite precise, the object is actually of type `ELEMENT_TYPE`, which is described in the “internalTypes” Map.

The `ELEMENT_TYPE` is actually an object with two fields:

- a “uuid” field of type: `UUID`
- a “name” field of type: `String`

If the `ELEMENT_TYPE` was a primitive type like `Integer` or `String`, `List<ELEMENT_TYPE>` would have been `List<Integer>` or `List<String>`.

If the “response” field was empty (only brackets: “{}”) it would say that nothing is returned except the “status”, “errorCode” and “errorMessage”.

After reading the *JSON*, we can now infer that the following *JSON*:

```
{
  "status": "SUCCESS",
  "trainers": [
    {
      "uuid": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
      "name": "Yakachu"
    }
  ]
}
```

is valid unlike this *JSON* object:

```
{
  "status": "SUCCESS",
  "trainers": {
    "Yakachu": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Tips

You can read the given *JSON* on your web browser, they usually allow to wrap the fields you do not want to see.

5 How to start

In order to start off on the right foot, we are going to explain how to implement an easy REST point: `/type`.

5.1 Model and Repository

We advise you to implement first the Model and the Repository. It will help you to get used to jOOQ.

In the case of the `/type` REST point, you will have to implement a `Type` Model and a `Type` Repository.

The `Type` Repository will implement methods to fetch data from the database.

The `Type` Model is just an object representing a Type.

Tips

When you ask jOOQ for data from your database, the return type of this data is a `Record`. Since a `Record` is not pleasant to use, you convert it to a Model (here a `Type` Model).

5.2 Service and Entity

Once it is done you have to implement the `Type` Service.

The `Type` Service methods will ask the `Type` Repository for data. The `Type` Service logic should be pretty straightforward since the two endpoints for `/type` are simple: get all types and get one particular type.

To communicate with the `Type` Controller you will have to implement a `Type` Entity. This `Type` Entity should be a conversion of the `Type` Model since the `Type` Service communicates with only one repository.

5.3 Controller and DTOs

Once it is done, you can implement the Controller of Type and its DTOs.

You will have to create a `Type` Controller with two methods, one when you want to get all the types and one when you want to get one particular type.

Tips

You will need to use Spark to register your methods in your `Type` Controller.

You will also need to implement the DTOs. Since there are no requests with body for `/type`, you should only have two DTOs representing responses, one when you get all the types and one when you get one particular type. These DTOs contains fields described in the *JSON* available on the intranet.

These DTOs are returned by the two methods of the `Type` Controller, you must serialize them into a *JSON*.

Tips

Use the Jackson library.

5.4 Test your endpoints

In order to help you test your endpoints, Postman is available on the PIE.

You can also use .http or .rest files in IntelliJ. The documentation about this functionality is here.

Be careful!

Do not forget to annotate your Entity, Controller, etc.

Don't be afraid, I just wanna help you.