Rodrigo García Carmona

Assignment 2: Development of a Web Application

Due date: -
Weight: *20 % of practical assignments' score*

This assignment is *mandatory*. Note that:

- Section ?? explains what items should be delivered to the professor.
- No assignments will be accepted past the due date.

This assignment *is mandatory* and therefore *must be delivered to the professor*. This assignment will contribute to the final score of the course.

1. Introduction

The aim of this assignment is to serve as a brief introduction to the setup and development of three-tier web applications using the Java programming language and the Play framework. Play has been chosen because it's easy to use and configure, follows modern web development paradigms and can be programmed with Java, a language most students are already familiar with.

This assignment is split in two parts. During the first one, you will download and configure an already existing web application, so you can test it using your own computer. In the second part, you'll modify the aforementioned application, to add features not already present.

1.1. Required Software Installation

Before tackling this assignment you must install the following software items:

- **Java SDK:** Can be installed from the official Java website[1]. You need to download version 8 or newer, and we strongly recommend that you choose the 64 bit version (ask the professor if your computer supports it if you don't know). It's possible that you'll also need to add the "javac" command to the system path. If you don't know how to do this, again, ask the professor.
- **Node.js:** Can be installed from the official Node.js website[2]. Although you are not going to use Node.js as a web application framework, Play can use the Node.js JavaScript engine, which is faster than the stock one.
- **IntelliJ IDEA Community:** Can be installed from the official IntelliJ website[3]. This is the Java IDE (Integrated Development Environment) that you will use to program you web application.

The Play framework itself will be installed with the provided sample web application.

2. Setup of a Play Framework Web Application

In this section you'll setup and launch an already existing web application.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`
[2] `https://nodejs.org/en/download/`
[3] `https://www.jetbrains.com/idea/download/`

2.1. Download the Application's Source Code

Go to `https://github.com/rgarciacarmona/fisio-repo`, click on the green *Clone or download* button at the right, and then select *Download ZIP*. You'll end with a ZIP file that you should extract to the folder you want to work with. Any folder is OK. We'll refer to this location in this document as the "project folder". Also, all relative routes to folders and files in this document are assumed to be inside it.

Alternatively, if you have Git installed in your system and know how to use it, you can clone the web application's Git project to your local machine.

2.2. Configure the Web Application

Some features of the web application must be configured before it can be properly launched. Since you are going to run it in your local machine, you should make the application use an in-memory database. This is not appropriate for a production environment, but for testing its fine.

Using a text editor (not a word processor) the file "conf/application.conf" and search for the following lines:

```
# Amazon RDS parameters
default.url = "jdbc:mysql://DBURL/fisiorepo?characterEncoding=UTF-8"
default.driver = com.mysql.jdbc.Driver
default.username = fisiorepo
default.password = PASSWORD

## H2 Database parameters
# default.driver = org.h2.Driver
## Choose one of the following two
## In memory database
# default.url = "jdbc:h2:mem:play"
## Local database mimicking MySQL
# default.url = "jdbc:h2:file:~/db/db;MODE=MYSQL"
```

The "#" character at the beginning of a line is a comment symbol. You have to configure the web application so it uses a local in-memory database that mimicks MySQL, so you need to comment the Amazon RDS lines and uncomment the appropriate H2 Database lines. The result will be this:

```
# Amazon RDS parameters
# default.url = "jdbc:mysql://DBURL/fisiorepo?characterEncoding=UTF-8"
# default.driver = com.mysql.jdbc.Driver
# default.username = fisiorepo
# default.password = PASSWORD

## H2 Database parameters
 default.driver = org.h2.Driver
## Choose one of the following two
## In memory database
# default.url = "jdbc:h2:mem:play"
## Local database mimicking MySQL
 default.url = "jdbc:h2:file:~/db/db;MODE=MYSQL"
```

2.3. Install and Run the Web Application

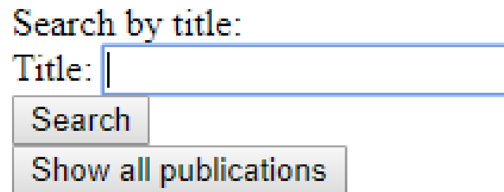
Open a terminal (*PowerShell* in Windows, *Terminal* in Mac OS) and navigate to the project folder. Inside that folder, type the following commands:

```
sbt clean
sbt compile
sbt run
```

Be patient, since the second command in particular could take a long time. The first line cleans your development environment (usually not needed at this point), the next one compiles the web application files, and the last line runs a web server with the application.

As explained by the output of the last command, now the server should be listening at port 9000, and will continue to run until you press enter in the terminal window.

Open a web browser (any will do) and go to `http://localhost:9000/`. Wait a little, since the first time the server answer a request it takes some time. You can see the progress in the terminal window. If everything has worked properly you should see something like this:



Congratulations! You've successfully run your first Play application.

3. Modify a Web Application

In this section you'll understand how the web application is made up and will make a small modification to it.

3.1. Anatomy of a Play Web Application

The Play framework applications follow the MVC (Model View Controller) model. These web applications have a fixed folder structure; each of them serving a different purpose. If you understand this structure it will be easy to find what you're looking for and add different features to the application.

The downloaded application has many folders, but the more important ones are the following:

```
app                      - Application sources
 | controllers           - Controllers
 | models                - Models
 | views                 - Views
conf                     - Configurations files
 | application.conf      - Main configuration file
 | routes                - Routes definition
 | META-INF              - Application-specific configuration files
   | persistence.xml     - Database configuration file
public                   - Public assets
 | stylesheets           - CSS files
 | javascripts           - Javascript files
 | images                - Image files
```

As you can see, the “app” folder contains the three types of elements that comprise an MVC application: models, views and controllers. Review the unit 1 slides for an explanation of the MVC model.

But applications also contain the so-called “public” resources: images, JavaScript files and other assets that are served directly by the web server and aren't part of the MVC model. These assets are usually (but not always) concerned with the look and feel of the application, so you'll not need to work with them in this assignment.

The “conf” folder contains the configuration files for the setup and tuning of the application. You already know about the “application.conf” file, since you change some of its lines to make it work with an in-memory database, although this file configures many other aspects of the web application.

The “routes” file is specially important, since it defines the web application's endpoints. Play applications follow the REST model, so all dynamically generated resources will be defined by a verb (like *POST*), a location (like */author*) and the controller who will implement the

action (like *controllers.AuthorController.addAuthor()*). Therefore, to develop a Play application we must create a set of models, views and controllers, and then define the endpoints that will be used to access the provided features. You should be familiar with the REST model from the first assignment and the slides of units 2 and 3.

Finally, the "persistence.xml" file contains the configuration parameters specific to the database solution selected (in this case, the JPA[4] ORM[5]). The only lines of this document that you need to be concerned with for the scope of this project are those with the *class* tag, since all the elements of your model should be reflected here.

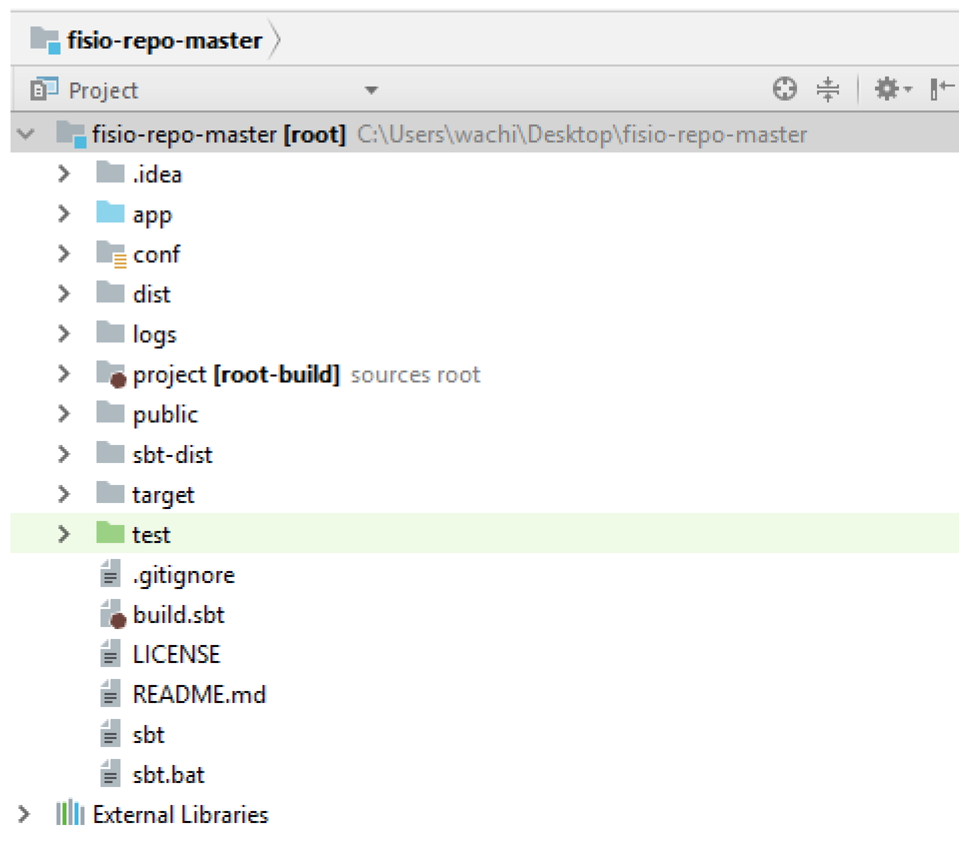
3.2. Import the Project into IntelliJ

You are going to use IntelliJ to have an easier time coding in Java. Of course, you could use a simple text editor, but the syntax highlighting and other features will make your life happier. You're going to use IntelliJ instead of Eclipse or Netbeans since this IDE (Integrated Development Environment) is the easiest one to set up with Play.

To import the web application project, follow these steps:

- Open IntelliJ
- Click in *File* and select *Open...*
- Select your project folder and click *OK*.
- Click *OK* again.

At the left part of the screen you should see something like this:



You can now explore all the project files, open them and edit their contents if you wish. You'll still continue using the terminal to compile and run the project, though.

[4] Java Persistence API
[5] Object Relational Model

3.3. Analysis of an Existing Feature: Authors

To add your own feature, you must first understand how one of the already existing features work. Let's study the inner workings of the way this application stores authors of scientific physiotherapy papers.

3.3.1. Routes Definition

First, look again at the "routes" file, paying special attention to these lines:

```
GET     /newAuthor                     controllers.AuthorController.index()
POST    /author                        controllers.AuthorController.addAuthor()
GET     /authors                       controllers.AuthorController.getAuthors()
```

As you can see, there are three endpoints defined:

- **newAuthor:** For showing a form that allows the user to insert a new author.
- **author:** That inserts a new author into the database.
- **authors:** That shows all the existing authors to the user.

Each of these endpoints is linked to a controller method of the *AuthorController.java* class. You'll take a look at this class later.

3.3.2. Model Definition

Open the "app/models/Author.java" class:

```
@Entity
@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class,
property = "id")
public class Author {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long id;

    public String shortName;
    public String fullName;
    public String affiliation;
    @ManyToMany(mappedBy="authors")
    @JsonIgnore
    public List<Publication> publications;
}
```

This piece of code defines a data type that will be managed by the application. In this example: an author. As you can see, authors have an ID, a short name, a full name and an affiliation, and are linked to several publications. Defining a data type is as easy as creating a class like this one.

You must also take a look at two other files in the model folder: "app/models/AuthorRepository.java" and "app/models/JPAAuthorRepository.java". These are, respectively, an interface that defines the actions that can be performed over an author, and its implementations. As you can see by looking at the interface:

```
@ImplementedBy(JPAAuthorRepository.class)
public interface AuthorRepository {

    CompletionStage<Author> add(Author author);

    CompletionStage<Stream<Author>> list();
}
```

There are two actions defined: one for adding a new author, and another for retrieving all authors from the database. The implementing class ("app/models/JPAAuthorRepository.java") details are not specially important at this point, so don't worry about them for now.

3.3.3. View Definition

First, open the "app/views/main.scala.html" file. This file defines the basic shape of all the views of this web application. Therefore, its details aren't really important for the analysis of the author feature, but note that the main tags of an HTML document, like *head* and *body*, are present in this file and used by all other views.

Now, open the "app/views/author.scala.html" file. This is the part you need to focus for now:

```
<form method="POST" action="@routes.AuthorController.addAuthor()">
    @helper.CSRF.formField
    Short Name: <input type="text" name="shortName"/><br />
    Full Name: <input type="text" name="fullName"/><br />
    Affiliation: <input type="text" name="affiliation"/><br />
    <button>Add Author</button>
</form>
```

This view presents a form with all the fields needed to create a new author. Note that there's no field for ID, since this attribute is automatically assigned by the database when the author is created. Also note that this form sends a POST request to the URL specified in the "routes" file for the *addAuthor()* method. This form doesn't offer the possibility of linking an author to a existing or new publication. That feature is out of the scope of this assignment.

While reading the code, pay attention to how the model and view are linked; this view allows the input of the information needed to create a new element. Finally, notice that the view calls the controller through the form.

3.3.4. Controller Definition

Open the "app/controllers/AuthorController.java" class. This class contains all the controller logic for the authors. The "routes" file defined three endpoints for the authors, and each of them is realized by a method of this class. Let's look at them in order. First, the *index()* method:

```
public Result index() {
    return ok(views.html.author.render());
}
```

This method returns an OK HTTP response (review unit 1 slides) and renders the view that you studied in the previous section. That is, it calls for a form that allows the user to create a new author, hence the endpoint address.

Now, look at the *addAuthor()* method:

```
public CompletionStage<Result> addAuthor() {
    Author author = formFactory.form(Author.class).bindFromRequest().get();
    return authorRepository.add(author).thenApplyAsync(p -> {
        return redirect(routes.AuthorController.index());
    }, ec.current());
}
```

This method performs the following tasks, in order:

1. Creates a new author from the data received in the HTTP request.
2. Inserts the author into the database.
3. Redirects to the *index()* method. So, after the author is inserted, the new author form is shown again (as explained before).

This method is invoked by a POST request, so it makes sense that is used to add information to the database.

Finally, look at the *getAuthors()* method:

```
public CompletionStage<Result> getAuthors() {
    return authorRepository.list().thenApplyAsync(authorStream -> {
        return ok(toJson(authorStream.collect(Collectors.toList())));
    }, ec.current());
}
```

This method performs the following tasks, in order:

1. Retrieves all authors from the database.
2. Marshalls the list of authors into a JSON document.
3. Returns an OK HTTP response with the JSON document.

This method does the most basic read operation over a database: just retrieve all items of a specific type. However, it is easy to expand it to allow for searches (if you feel curious, look at the *searchPublications(String title)* method of the *PublicationController* class).

This controller method returns a JSON document so you can see that HTML is not the only document type that can be returned in an HTTP response. It's as easy as working with HTML.

And, with that, you understand how everything fits together! You can now get the full picture of how models, views, controllers and routes fit together in a Play framework application.

3.4. Add a New Feature to the Application

You have already understood how an existing Play application is coded, configured and built, so now is your turn to add a new feature to the provided web application. You must extend it by:

- Adding a new element to the model, similar to the author used as an example in this text. To do this, you must:
  1. Create a new class that represents the model (like *Author.java*) in the appropriate folder. For this assignment, you don't need to link that element to any other (like author and publication are). It's enough to create an element with an ID and two or three other attributes.
  2. Create a repository interface (like *AuthorRepository.java*) and a class that implements it (like *JPAAuthorRepository.java*). This repository must have two methods: one for adding a new element and the other for retrieving all existing elements. You can reuse a lot of code from the examples provided.
  3. Add the element to the classes listed in the "persistence.xml" configuration file.
- Creating a new view for that model. This view should have a form with the fields needed to create a new element of the model. Take a look at "app/views/author.scala.html" for inspiration and put your view in the appropriate folder.
- Coding a controller that links the view and the model you've just created. Look carefully at *AuthorController.java*, since most of your code will be very similar to that class. Keep in mind that you will need to create methods for:
  - Rendering the view (with the form) that you have created before.
  - Adding a new element of the model from a POST HTTP request.
  - List all existing elements in the database as a JSON document.
- Adding three new entries to the "routes" file, one for each of the controller behaviors programmed. Again, look at the existing entries of the "routes" file for inspiration.

When you've finished doing all this, run your web application by typing:

```
sbt compile
sbt run
```

This is the web application that you should deliver to the professor.

4. Delivery

To pass this assignment you must make a ZIP file of all the folders of your web application and send it to the professor's email (r.garcia.carmona@gmail.com). Alternatively, if you know what you're doing, you can fork the provided GitHub project and send the link to your version of the project to the professor.

5. If you are bored...

If you're bored and have already finished the assignment, you can try any of the following advanced tasks:

1. Use a MySQL database in your computer instead of the in-memory database. For that you will need to install MySQL Workbench and MySQL Server Community Edition. Both can be installed from the official MySQL website[6]. This is the DBMS (DataBase Management System) that the web application is going to use to store the data in the production environment (in the next assignment).

2. Try to understand the complete structure of the provided web application. It has several elements that, linked together, model a scientific paper.

3. Use Postman to manually send HTTP requests to the application. If you've done the previous optional task, it could be interesting to check the "/publicationJSON" endpoint, since it uses JSON to add publications to the database, instead of the already explained HTML forms. Postman: Can be installed from the official Postman website[7]. You will use this software to send HTTP requests to your web application.

[6] `https://dev.mysql.com/downloads/`
[7] `https://www.getpostman.com/`