Software Architectures
Assignment 2: Scala Play Framework

Arthur Chomé
arthur.vincent.chome@vub.be
0529279

1 Introduction

For the second assignment, we were given source code for a web application implemented with the Scala Play framework[1] and using the *iRail.be* API[2] to help users plan their train trips in Belgium. The application —however— was given to the students in an incomplete state. It had to be extended by individual students with the following functionalities: complete the route planner, make a liveboard feature, add links to both the route planner and liveboard and add a destination liveboard hyperlink. All these requirements were successfully implemented.

```
GET     /liveboard
GET     /liveboardparameter/:station
POST    /board
```

Figure 1: Methods implemented for the assignment in the "routes" file: "/liveboard" redirects the user to the liveboard page, "/liveboardparameter" does this while loading the liveboard of the given station parameter and "/board" allows to do liveboard requests to the *iRail.be* API.

2 Route Planner

Given two locations inserted at the front-end, this page would allow the users to see all trains going from the first location to the other location. The provided source code contained the view, the form, the controller and the JSON model to make it work. Though the request was made with all the parameters included and the response was given back in JSON format. It still had to be accessed from the model and dynamically updated on the front page. This required changing the front-end HTML code to display what was found with the API.
The situation where nothing was found by the REST API was also considered. We always check that the response's status is 200 to make sure our request was

[1] `https://www.playframework.com/`
[2] `https://irail.be/route`

a success. If we do not account for this, the whole application would crash, this happened quite a few times with me.

3 Liveboard

By filling in a specific station in the front-end, this feature should show all trains passing the given station. The page works in a similar way as the route planner by sending in requests to the *iRail* REST API.
The URL for this webservice[3] is composed of the station specified by the user and a boolean True for "fast". This means no specific train direction must be linked to trains, it was recommended to do so according to the API's documentation[4].
To get the data from the front-end to the back-end, we had to code a new data form. The form "LiveForm"[5] only has a "station"-field that we use to transport the specified station to the back-end.
The webservice has to give back its results in JSON-format. The issue is what to give back, we are not interested with everything the webservice knows but only the key information about the departures from the given station. This is why we introduce our own client-side JSON data model "LiveboardDeparture" that allows us to temporarily store the most important result information in small datasets. It contains a departure's vehicle, delay time, arrival time, final destination station and the platform it will depart from.
Before ultimately sending this information back to the front-end, it gets refined with the controller's "updateAndAddDeparture" function. At the front-end, all the JSON model fields get used to display the information.
This feature required a new view-page to be made ("liveboard.scala.html") with a controller ("LiveboardController.scala") and a "/liveboard" GET-method to access this view.

4 Links

When following the path "route-planner/conf/routes" in the source code files, you can find the routing file "routes" of the project. The "/liveboard" GET-method has been added to its methods. Just as the "/" GET-method puts the current view of the application to the index page, the "/liveboard" method puts the current view of the application to the liveboard page.
Two hyperlinks "planner" and "liveboard" were placed in both the index page and liveboard page respectively. The "liveboard"-hyperlink in the index page links to the liveboard page and the "planner"-hyperlink in the liveboard page links to the index page. This way, the user can navigate to both the index and liveboard pages.

[3] `https://api.irail.be/liveboard/?station=STATIONNAME&fast=true`
[4] `https://hello.irail.be/api/1-0/`
[5] route-planner/app/model/LiveForm

5 Destination Liveboard

When querying the route planner, it must be possible to click the found trains' destinations and get redirected to their liveboard. The destination station text in the table is a hyperlink that uses the "/liveboardparameter" GET-method that takes in a "station" URL-parameter when being called. This parameter gets passed to the "liveboardWithStation" function of the liveboard controller[6]. This function puts the application view on the liveboard page and loads it with the data of the given station parameter.

6 Conclusion

All 4 features described above were implemented with relative ease. I already implemented some websites before so the routing between the route-planner and liveboard pages was not difficult. I had also some experience with Wikipedia's MediaWiki[7] API so completing the route-planner and implementing the liveboard was just a question of reading the documentation of *iRail.be*.

[6] route-planner/app/controllers/LiveboardController.scala
[7] `https://www.mediawiki.org/wiki/API:Main_page`