UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

UCLCampus: a moblie application for UCL students

Supervisor: Yves Deville

Readers: Kim Mens
Hildeberto Mendonça
Mathieu Zen
Jorge Perez Medina

Thesis submitted for the Master's degree in computer science and engineering options:

by : Arnold Moyaux
Baptiste Lacasse

Louvain-la-Neuve

Academic year 2015-2016

Contents

Chapter 1

Introduction

Brief introduction of the project, the goals and the contents of the rest of the thesis.

Chapter 2

Background

In this section, we will look at the different existing technologies relating to the different aspects of our project and will explain the choices we made. // TODO

2.1 Cross-platform mobile development tools

In each of these sections, we will detail the different approaches one could choose to develop a cross-platform moblie applications. We will also present several frameworks using these approaches. We will then compare them and choose one of those approaches for the rest of the project. //TODO

2.1.1 The native approach

The first approach we considered for our project was what we call a native approach. The native approach consists in using the native technology and language for each platform, for instance Java for Android and Objective-C for iOS.

Pros	Cons
Best achievable performance	Low maintainability
Always up-to-date with the latest API	Harder to find contributors fluent in all technologies
Can use any platform	Can lead to different versions of the application

Table 2.1: Pros and cons of the native approach

2.1.2 The web approach

A second approach we considered was the web approach. This approach consists in using HTML5 to develop an application that will be usable on any platform.

Pros	Cons
Can be used on any mobile platform	Doesn't have access to native platform features
Easy to find contributors fluent in HTML5	Harder to implement local storage/security (//TODO NEED BETTER SOURCE)
Easy to maintain	Not as performant as native

Table 2.2: Pros and cons of the web approach

2.1.3 The hybrid approach

The last approach to develop a mobile application is called the hybrid approach. An hybrid app is mostly built using HTML5 and JavaScript and is then wrapped inside a thin native container, giving it access to native features.

Pros	Cons
Can be used on any mobile platform	Not as performant as native
Easy to find contributors fluent in HTML5 and JavaScript	
Easy to maintain	

Table 2.3: Pros and cons of the hybrid approach

2.1.4 Our choice

2.2 Open-source project and code sharing

In this section, we will explain the choices we made concerning the code sharing platforms we used as well as the licenses we used to protect our work.

2.3 Project Management Methodologies

Here we detail the choices we made as to how we were going to manage the different parts of the project.

Chapter 3

Functionalities of UCLCampus

In this chapter, we will show how we defined the relevant functionalities of our application as well as the user interface.

3.1 Choice of functionalities and sections

In order to define what kind of functionalities we wanted to be part of our application, we needed to know what the students needed. The first step was to define a number of user stories that we would then translate into functionalities.
We split our user stories into several categories:

- Studies: anything directly related to a user's studies, for instance his classes, the lecture halls or the libraries.
- Campus: anything related to student life in the campus but not related to the user's studies. For instance "Kot à Projets" or "Cercles".
- City: anything related to the city the user is in but not related to the university. For instance a cinema or restaurants.
- Tools: the tools offered by the application that might relate to several other categories. For example the map.
- Settings: the settings of the application. For example the language or the currently selected campus.

We also define two types of users:

- Students: students can access all the functionalities of the application using their UCL login information. Indeed, some functionalities are student specific. For instance, it wouldn't make sense for a person who isn't a student to try to access his or her schedule.

- Users: users are people who aren't students but might still be interested in some functionalities the application has to offer.

In our user stories, any story starting by 'as a student' cannot be used by users while any story starting by 'as a user' can be used by both users and students.

We will now give a list of the different user stories we thought of for each of our categories.

Studies

- Schedule
  - As a student, I can access my schedule in order to know when my courses are given.
  - As a student, I want to know where a course is given.
  - As a student, I want to know the name of a teacher giving a certain course of my schedule.
  - As a student, I can export my schedule to my phone's agenda so that I don't need Internet access to see it.
- Libraries
  - As a user, I can see whether a library is open or closed.
  - As a user, I can display the address of any library.
  - As a user, I can have a GPS guide to access libraries from my location.
- Lecture halls
  - As a user, I can check the address of any lecture hall.
  - As a user, I can have a GPS guide to access lecture halls from my location.
- Websites
  - As a user, I can quickly access the moodle website through the application.
  - As a user, I can quickly access the UCL website through the application.

Campus

- Events
  - As a user, I can see a list of events taking place in my campus.
  - As a user, I can sort the events by category.
- Kots à Projet
  - As a user, I can check Kots à Projet to know their address and projects.

  - As a user, I can have a GPS guide to access Kots à Projet from my location.

- Cercles
  - As a user, I can check "Cercles" to know their address.
  - As a user, I can have a GPS guide to access "Cercles" from my location.

- Restaurants Universitaires
  - As a user, I can see the different "Restaurants Universitaires" in my campus.
  - As a user, I can check the menu of the "Restaurants Universitaires".
  - As a user, I can have a GPS guide to access "Restaurant Universitaires" from my location.

- Sports
  - As a user, I can see a list of sports organized in my campus.
  - As a user, I can sort the sports by day or by sport.

City

- Tourism
  - As a user, I can see the address of the city's information center.
  - As a user, I can see a list of the museums of the city I'm in.
  - As a user, I can see whether a museum is opened or closed.

- Activities
  - As a user, I can see the address of the city's cinema in order to access it with the help of a GPS guide.
  - As a user, I can see several activities I can do in the city I'm in.

- Restaurants and bars
  - As a user, I can see a list of the restaurants of the city I'm in.
  - As a user, I can see a list of the bars of the city I'm in.

Tools

- Maps
  - As a user, I can access a map of the city I'm in in order to check points of interests.
  - As a user, I can receive help from a GPS guide in order to access a location of my choice on the map.

- Mail
  - As a student, I can check my emails on my uclouvain account.
  - As a student, I can send emails from my uclouvain account.
- Help
  - As a user, I can see where the UCL parkings are located.
  - As a user, I can receive help about how to configure the UCL wifi.
  - As a user, I can receive help about common transportation.

Settings

- As a user, I can change the application's language to French, English or Dutch.
- As a user, I can select my campus.
- As a student, I can login on my UCLouvain.be account.
- As a student, I can log out from my UCLouvain.be account.

3.2 User interface

Once we determined the different features we wanted in our application, we needed to organize them in a way that makes sense for users. In order to do so, we made sketches of what the application might look like using InVision. InVision is a website that lets people design and style mobile applications prototypes. It allows us to get an idea of what the finished product might look like without having to dive into any code. The sketches we made are available in the annex.

In these sketches, we can see that we decided to have one menu per category we defined in the previous section. Each menu has an associated color, allowing the user to always have visual clues to help them know where they are.

Once the sketches were done, we shared the link to our prototype to over 1000 students, mostly student in their first year, as they represent the future users of this application. They were able to browse through the application using the buttons, as if it was already working, and leave comments and feedback wherever they wanted to.

While we didn't receive as much feedback as we would have hoped, the one we received was constructive and helpful. Most people were satisfied with our 3 first categories, Studies, Campus and City. The fourth category, however, was more criticized. Here are some comments we received concerning the Tools category.

After reading these comments, we decided to rework the Tools section. We agreed that the mail part was superfluous and we decided to drop it entirely. We also decided to drop the "Help" section as we didn't think it was important enough. That left us with the map. We decided that it was important that the map was not grouped with the city, the studies or the campus as it was important to all three sections. We thus decided to leave it in the Tools section. We also decided against renaming the section "Maps" as future contributors may very well add functionalities we didn't think of in this section.

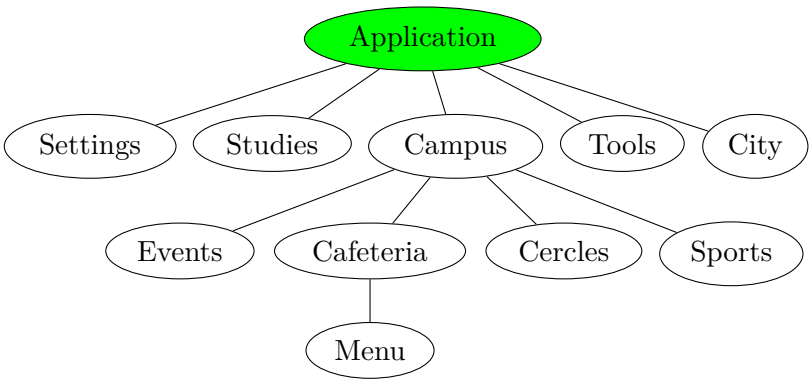
Chapter 4

Implementation

Here we will explain the overall architecture of the application. We will also explain some aspects we considered when implementing the application.

4.1 Architecture

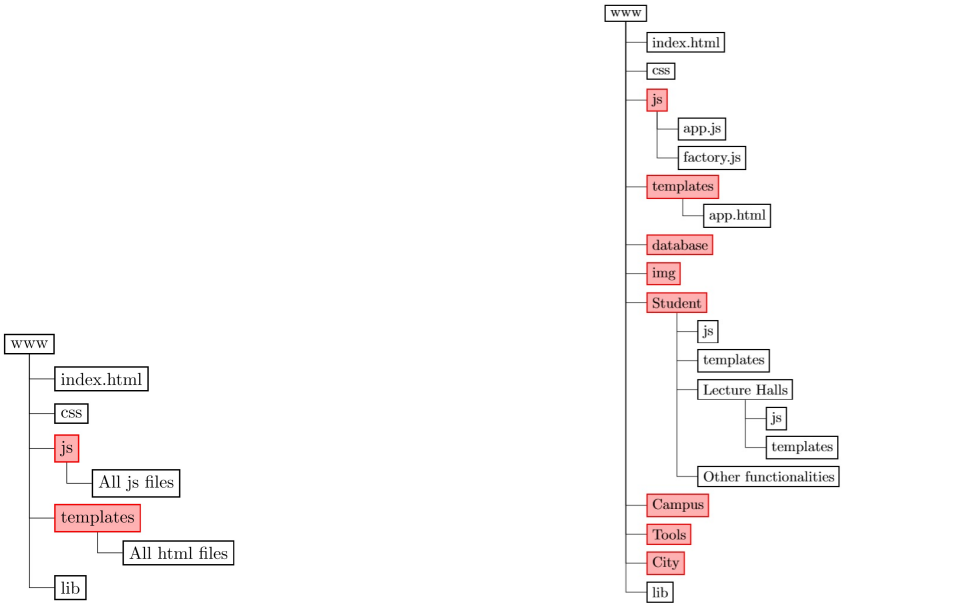
The purpose of the application is to be extensible and easily maintainable. We wanted a programmer can add its functionalities at each level to the application. For this we thought our implementation as a tree. At the top level we have the application with the settings menu and base configuration after it we have four branches pointing to global sections that we decided to create, themselves pointing to their functionalities and so on. Here is a part of the tree in order to give you the idea.



Each node having his own JavaScript,html and css (if needed) code.

4.1.1 Folder organisation

We wanted to keep the same state of mind for the file organisation. Ionic base architecture is to put all html file in a folder named templates, all js in a JS folder, ... The problem is that become messy once we have a lot of functionalities(maybe having more than one js and html). We modify it to respect the tree architecture we want. With our folder system, a programmer can add his own subtree to the main tree. And if you want to modify a specific functionality,



(a) Before (b) After

Figure 4.1: Folder evolution

you have directly access to the related files. Figure 4 1 is a summary of the change, red folder are those we modify.

4.1.2 Information processing

Here I just explain how we deal with the information processing from an architecture's design point of view. The section 4.4 explains in detail how we did it for each specific part. There is a lot of external information to relay into the application (libraries schedule, libraries addresses, daily events in the campus,...). We have two possible ways to import them.

Database

Pros	Cons
Always available	Need someone to update
Easy information retrieval(query)	Takes memory
Easy to modify	
Fast	

Table 4.1: Pros and cons of a database

Web parsing

Pros	Cons
Automatic update	Need an Internet connexion and an operational server side
Easy information retrieval with web services(query)	Horrible information retrieval without web services
No hardware memory consumption	If the web server change, maybe you will need to recode all the parsing method
	Slow

Table 4.2: Pros and cons of the web parsing

An considerable limitation is the need for someone in order to update the database or creating new parsing system. We have no workforce for it, so if we have the choice between the two methods, we will select the one needing less modification in the future.

4.1.3 Front-end and back-end

- Front-end: Part of the user interface that can be separated in two fields. First is design and the second is html, css and JavaScript development.
- Back-end: Is the hidden part of the iceberg, what you can't see. For example: the database, the parsing function, ...

We create a front-end and a back-end system in our application. It helps a lot for the maintain because you can modify part without involving the other. For example, we store lecture halls in the database but the UCL create a new website with web services providing all lecture halls and their information. It's better than the database because it automatically updates and so you want to use them instead. You can do it in a specific part that is totally isolated from the code for the user interface.

4.1.4 Factory

Factory is a functionality from angularjs which we used as a back-end service. The factory will take the data from the database or web parsing, modify the data format to be easier to handle and then transmit it to the front-end. Figure 4.2 illustrates this.

For example, we open the page for the schedule. (1)the system notifies the schedule factory in order to get his data. (2) the factory create a custom HTTP request and send it to the ADE

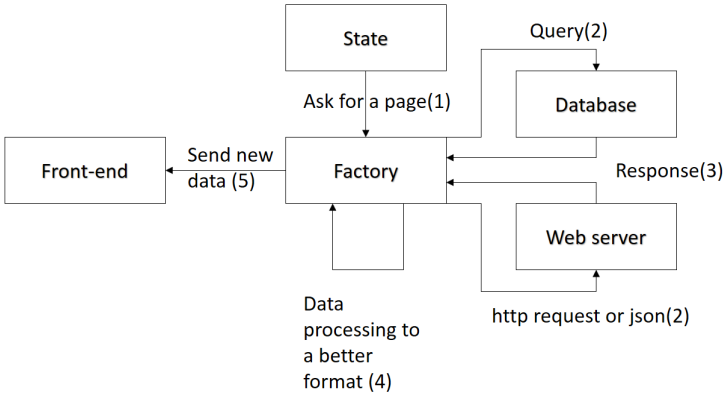


Figure 4.2: Factory operation

domain. (3) Server sends a response. (4) Data in the response are not easy to manipulate (long string with html tags inside), so the factory pick the important element in the response(with a parsing algorithm) and put them in a JavaScript object where data are easy to handle. (5) Send new data to front-end.

4.1.5 State provider

The state provider is a functionality of angularjs. It allow you to define the page configuration (cached or not for example). Furthermore it create an edge between the back end and the front end, figure 4.3 illustrate the operation of the state provider.

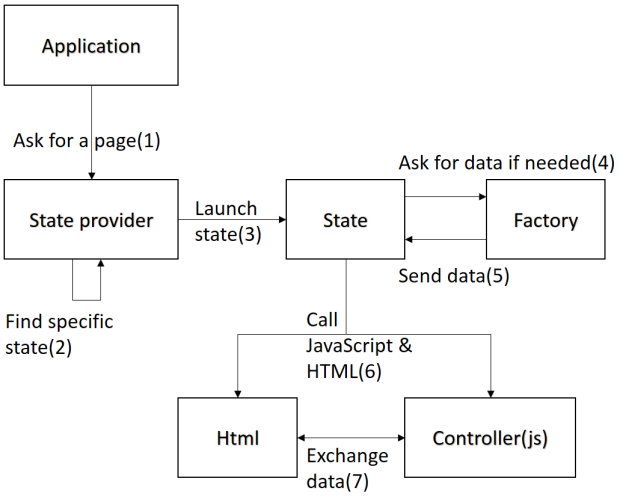


Figure 4.3: State provider operation

First the application ask for a page. The state provider contain a list of specific state(one state by page), it will return the state relative to the asked page. The state will contain information about the page:

- The location of the html template
- The associate controller(js)
- A flag for cache memory(true or false)
- The tab membership(for example student if we are in the subtree student)

. For back end data(optional), the state will call the needed factory and wait its response before launching the template and the controller for the user interface. JavaScript(controller) and Html can exchange informations and sent it to back end in some cases.

4.2 Coding standards

4.2.1 Header and footer

We managed our code to create automatically a default new footer and header when you start new html template. The footer and header have a specific color for each tab-section (blue for student, yellow for campus, purple for tools and green for town). We allow developers to overwrite the default header but under two condition

- The title have to keep the same font and have to be center.
- The header color must respect the section color

Footer must not be overwrite or be hidden. If developers need to insert their specific header or footer, Ionic provide a css in order to add subfooter and subheader.

4.2.2 Tree hierarchy

We want developers to keep the tree hierarchy we created for the state and for the folder. For css and extra JavaScript, html, we don't enforce any rules as long as it stay in the relative module.

4.2.3 Colors

We used the UCL colors code[1]. For this we modified the sass theme of Ionic. This allow module developed with basic ionic css component to update with the application color instantly and automatically. The main color of UCLCampus is the blue "UCL".

Color	css name	code
Blue	positive	#00214e
Yellow	energized	#f29400
Purple	royal	#88005d
Green	balanced	#76ad1c

[1]https://www.uclouvain.be/459461.html

4.3 Security

4.4 Information retrieval

In this section we will detail the different sources of information we used in our application as well as the technologies we had to use in order to retrieve said information.

4.4.1 Libraries and lecture halls

In order to retrieve a list of all libraries and lecture halls in the university, we contacted the UCL to ask them if such a database existed. Sadly, it appeared that no database of this kind exists. We then wondered how LLNCampus //TODO parler de LLNCampus avant// managed to obtain all the information they had on lecture halls and libraries. Since their application is also open-source, we were able to find the database they used. Since it was very complete and was the only database available, we decided to re-use this database. However, since their database was built with only one campus in mind, we had to change it a bit.

Database structure

//TODO include er diagram ? useful ? The database is an SQLite database. The table we use for the most part is called "poi" (point of interest) and has the following fields:

- ID (integer): used to identify the item
- Name (text): name of the point of interest
- Latitude (real): latitude of the point of interest
- Longitude (real): longitude of the point of interest
- Type (text): type of the point of interest, can be "auditoire", "kap", "bibliotheque", ...
- Address (text): address of the point of interest
- Img (text): path to the image linked to the poi

To these fields we added the following two:

- Campus (text): campus where the poi is located
- Abbr (text): abbreviated version of the poi. For instance "BARB" for the Ste-Barbe lecture hall

The first field is useful if we want to have different campuses (//TODO check orthographe) while the second is used to find the poi on the map.

Retrieving lecture halls

Once we had the completed the database, we had to connect it to our application. In order to do so, we used the cordovaSQLite plugin. Once connected, we were able to retrieve the lecture halls using the following SQL query:

```
SELECT * FROM poi WHERE TYPE = 'auditoire' AND CAMPUS = ?
```

This query will retrieve all the fields from the poi table we discussed earlier which have the type "auditoire" and are located in the desired campus. The "?" character is used as a placeholder for an argument we can pass to the function. We will then store the result of the query in a JavaScript array and used that array to display the information to the user.

Retrieving libraries

In order to retrieve all the information concerning libraries, the poi table was not enough. Indeed, we wanted the users to be able to know whether a library was opened or closed and the schedule was not available in the poi table. Thankfully this information was available in another table, called "bibliotheque_horaire". This table only has 4 fields:

- Building_ID (integer): identifies the library. Same id as the one used in the poi table.
- Day (integer): Day of the week (0 for monday, 1 for tuesday, ...)
- Begin_time (integer): Time of the day when the library opens. Given in minutes elapsed since midnight
- End_time (integer): Time of the day when the library closes. Given in minutes elapsed since midnight

Given this new table we were able to retrieve all the information we needed using the following SQL query:

```
SELECT * FROM poi, bibliotheque_horaire WHERE poi.TYPE = 'bibliotheque'
AND poi.ID == bibliotheque_horaire.BUILDING_ID AND DAY = ? AND CAMPUS = ?
```

This query retrieves all poi with type "bibliotheque" and finds their respective schedule for a given day, all that for a given campus. The result in then stored in a JavaScript array.

4.4.2 Events

In order to find events taking place in Louvain-la-Neuve, we used the www.louvainfo.be website. This website provides a schedule of all upcoming events in the city. It is however specific to Louvain-la-Neuve and we would need to find equivalent websites for the other campuses.
The website provides an Atom RSS feed of the events. We decided to use Yahoo Query Language to parse the RSS feed and retrieve the information. The reason we used the Yahoo Query

language was that it was the only way we found to perform such an operation. Indeed, the more widely used Google Feed API has been deprecated. We therefore had no other choice but thankfully we haven't encountered any issue with YQL.

We used the following query to retrieve the feed:

```
select * from feednormalizer where url='http://louvainfo.be/evenements/feed/calendar/'
```

The result is then parsed and stored in a JavaScript array.

4.4.3 Schedule

For the schedule we had no choice between database or web parsing. The ADE dump database for all courses(of all the students) has a size of 70mb which is way too much memory for a mobile application. Therefore ADE has a custom web site where you are able to send customize url in order to extract specific informations. It's a bit worse than web services because you need a parsing technique.

ADE custom URL

The request always start with `http://horairev6.uclouvain.be/jsp/custom/modules/plannings/direct_planning.jsp?`. After it we can add some query parameters.

- code : The course code we want to see on the schedule. We picked the logged student courses codes for it.
- weeks : The weeks we want to see. It's a number per weeks. We wanted to get all the year so we used a suite from 1 to 52 (1,2,3,...).
- projectID : A number defining the academic year we want to see(16 for 2015-2016). This number need to be manually updated each year. We couldn't update it automatically because it seems to be a random number chosen each year(7 and 16 for the last two years).
- password and username : Require for ADE connexion.
- page configuration : There are multiple parameters that I won't explain. Their purpose is to custom the ADE web page we access. We used it to get a tabular summarizing the schedule that is easier to parse than the original web site.

Parsing

Http response contain an html code. The tabular is really useful here in the parsing because it provides a block of informations easy to extract. It contain information about each lecture of the year. Here is the syntax of the tabular in the response.
<tbody>

```
<tr>
    <td>Date</td>
    <td>Course Code</td>
    <td>Hours:Minutes</td>
    <td>Date</td>
    <td></td>
    <td>Useless information</td>
    <td>Eleves</td>
    <td>Professors</td>
    <td>Place</td>
    <td>Course Code again</td>
</tr>
<tr> an other lecture </tr>
...
</tbody>
```

We used a string parsing based on regular expression in order to extract information from tags and place it into a JavaScript object with field easily accessible. At the end the back end sent a list of lecture object to the front end.

Local storage

We thought it could be good to keep the schedule relative to the logged student in memory. In this case he could have an access to it every time without the need of an internet connexion. For this, once the parsing done, we keep the new schedule object in memory and the next time the student access the page we load it from local storage instead of parsing it again. That allow us a better performance because it's faster and has a better availability. The problem is that the schedule can change during the year so we allow the student to force a new web request with a new parsing in order to update it (with a refresh button on the page).

4.4.4 University canteen

We choose to store the main informations(name, place, image, opening time) about restaurants in the database because it's something that will not change every year and there is not a lot of field to update if some modification are needed. We encode the 6 restaurants present on the ucl website[2]. About the menu we couldn't store it in the database because of the existence of the daily menu (we should omit them or update them manually each week). We wanted to do some web parsing but the html code for each restaurant had his own syntax(even if they have the same rendering) ant thus we would need to create one parsing technique by university canteen which was a too heavy workload for the time we had. Instead we chose to create a button linking to the related menu page.

[2]https://www.uclouvain.be/restaurants-universitaires.html

4.4.5 Sports

We couldn't store the sports in a database because the planning undergoes changes every year and we don't have manpower to update it. The sports department don't have web services either so the only solution we had was to parse the website. On a positive note, the Louvain-la-Neuve and Woluwe sites share the same website and thus the parsing is effective for both.

URL

The base url for the two sites is the same [3]. There exist two main query parameters possible

- The campus: A number (1 for all, 2 for Louvain-la-Neuve, 4 for Woluwe). The factory create a specific request with 2(resp.4) when the selected campus is Louvain-la-Neuve(Woluwe)
- The skip: The sport web site response contains only 50 sports time slots. The skip argument permit to access the other sports instances after the first fifty.

It is a bit more difficult and slower than the other request because in this case we need to create a request by 50 sports time slots(for example if we have 132 sports we need three request. The first taking 0-50 instances, the second 51-100 and 101-132 for the last). Moreover the website is not 100% available some times, we couldn't access it because it was off line so we added a time-out method in order to prevent the user that the website is probably off-line.

Parsing

For the parsing the sport has quite close structure than with the schedule except different tag name so we used an other string parsing method based on regular expression. The sport website create a page that contain the sport closer than the day and the time we are. For example if we Friday 14 hours, it will show the Friday sports time slots after 14 hours first. Thus we iterate over all the time slots until we are sure that we capture all the sport for a week. We stocked the result in an other list of JavaScript object that we store and send to the front end.

Local storage

Exactly the same reasoning than for the schedule. Here we saved the result for both Louvain-la-Neuve and Woluwe in different table so we have them both in memory once they has been accessed at least once. Moreover each time the page is requested the software will reorganize the list in order to have the closer time slots first which are in general more suitable for the user.

[3] `http://ucl-fms01.sipr.ucl.ac.be:82/ucl_sport/recordlist.php?`

Chapter 5

The application

In this section we will present the application as we implemented it.

5.1 The application UCLCampus

5.1.1 Studies

5.1.2 Campus

5.1.3 City

5.1.4 Tools

5.1.5 Others

5.2 Modularity and how to add a new functionality

5.3 Future functionalities and possible improvements

Chapter 6

Analysis

Here we will reflect about the many choices we made and try to decide wether they were the right ones or not.

6.1 Ionic framework

6.2 GitHub

6.3 Project Management

Chapter 7

Conclusion

Chapter 8

Bibliography