Universidade de Santiago de Compostela

ESCOLA TÉCNICA SUPERIOR DE ENXEÑARÍA

App4Refs

Mobile application to enhance refugees integration in Greece

Autor:
David Campos Rodríguez

Titor:
José Varela Pet

Grao en Enxeñaría Informática

July 27, 2018

Traballo de Fin de Grao presentado na Escola Técnica Superior de Enxeñaría da Universidade de Santiago de Compostela para a obtención do Grao en Enxeñaría Informática

This work is licensed under the **Creative Commons Attribution-ShareAlike 4.0 International License**. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

The attached source code is licensed under the **Apache License 2.0**. To view a copy of this license, visit: https://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Special thanks

To MY PARENTS, as this is their achievement rather than mine. Without their efforts I would not be making this project today.

To JOSÉ VARELA PET, for the support received during the elaboration of the project, and in a more general sense to all the teachers who, along my life, helped me to find my way.

Finally, to all those friends who are always there, holding me up when I fail to find my ground.

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 The project and my participation . . . 2
1.3 This document . . . 3

2 Technologies **5**
2.1 The web technologies . . . 5
2.2 About the back-end . . . 7
2.3 Building the front-end: Gulp . . . 8
2.4 Integrated development environments and documentation . . . 9

3 Project management **11**
3.1 Project charter . . . 12
3.2 Scope management . . . 13
3.2.1 Work Breakdown Structure . . . 13
3.2.2 Description of scope . . . 14
3.2.3 Out of the scope . . . 14
3.2.4 Project restrictions . . . 14
3.3 Project life cycle . . . 15
3.4 Configuration management . . . 16
3.5 A short note on the requirements . . . 17
3.6 Use case point analysis . . . 18
3.7 Cost management . . . 21
3.8 Schedule management . . . 23
3.9 Risk management . . . 25
3.9.1 Risk measures . . . 25
3.9.2 Risk palliation strategies . . . 29
3.9.3 Risk specification . . . 29

4 Requirements capture **39**
4.1 Context study . . . 39
4.1.1 Current situation . . . 39
4.1.2 Project vision and opportunities . . . 40

4.1.3 Interested parties . . . 41
4.1.4 Systems to interact with . . . 42
4.2 Requirements specification . . . 42
4.2.1 Stakeholders of the system . . . 42
4.2.2 Goals . . . 43
4.2.3 General vision of the proposed solution . . . 44
4.2.4 Use cases . . . 49
4.2.5 Functional requirements . . . 58
4.2.6 Non-functional requirements . . . 63
4.2.7 Conceptual data scheme . . . 77

5 Design 81
5.1 General structure of the system . . . 81
5.2 Back-end: The database . . . 83
5.2.1 Entity-relationship model . . . 83
5.2.2 Relational model . . . 87
5.3 Back-end: The API . . . 88
5.3.1 Interface design . . . 88
5.3.2 The architecture . . . 90
5.3.3 Domain layer and view Layer . . . 91
5.3.4 Managing errors . . . 93
5.3.5 Data layer . . . 96
5.3.6 Transactions execution . . . 98
5.4 Front-end: The PWA . . . 104
5.4.1 The pages . . . 105
5.4.2 Application, routing, resources and navigation bar . . . 107
5.4.3 Grid pages . . . 111
5.4.4 Items, periods and the list page . . . 112
5.4.5 API connection . . . 114
5.4.6 Map page . . . 117
5.4.7 Geo-locating the user . . . 118

6 Implementation and testing 121
6.1 The file structure of the project . . . 121
6.2 Back-end . . . 122
6.2.1 The database . . . 122
6.2.2 Initial data . . . 123
6.2.3 The API . . . 124
6.2.4 Auto-loading the classes . . . 125
6.2.5 Redirection . . . 126
6.2.6 Other implementation details . . . 126
6.2.7 Unitary tests . . . 127
6.3 Front-end . . . 128

6.3.1 File structure . . . 128
6.3.2 Precache and dynamic caching . . . 129
6.3.3 Grouping periods . . . 134
6.3.4 Saving the application to the home screen . . . 137
6.3.5 Deployment . . . 138
6.4 General testing . . . 139

7 Conclusions and future work 141
7.1 Satisfied requirements . . . 141
7.2 Improvements to be made . . . 141
7.3 Future work . . . 143

A Technical manuals 145
A.1 Database deployment . . . 145
A.2 Importing the initial data . . . 145
A.3 API deployment . . . 146
A.4 Adding new URLs to the API . . . 147
A.5 Building and deploying the PWA . . . 149
A.5.1 Gulp tasks . . . 151

B User manual 153
B.1 What can be done with App4Refs? . . . 153
B.2 Home screen . . . 153
B.2.1 Navigation bar . . . 154
B.2.2 List pages . . . 155
B.2.3 The maps page . . . 156

Bibliography 159

List of Figures

2.1 Classical web and AJAX comparison 6
2.2 Example of Gulp execution 8

3.1 Work breakdown structure 13
3.2 Gantt diagram (I) 26
3.3 Gantt diagram (II) 27
3.4 Gantt diagram (III) 28

4.1 Home screen and help area mock-ups 45
4.2 Information, leisure and services area mock-ups 46
4.3 Example of list area 47
4.4 Maps mock-ups 48
4.5 General use case diagram of the system 49
4.6 Conceptual data scheme of the system 78

5.1 General structure of the project 82
5.2 Entity-relationship model of the database 83
5.3 Data flow over the three tiers of the API 90
5.4 Class diagram of the domain and view layers of the API 92
5.5 Sequence diagram of the URL matcher 94
5.6 Sequence diagram attending a request 95
5.7 Class diagram of the API exceptions hierarchy 96
5.8 Class diagram of the data layer of the API 97
5.9 Sequence diagram of the transaction to get categories 98
5.10 Sequence diagram of the transaction to get items 99
5.11 Sequence diagram of mapping an item 100
5.12 Sequence diagram of the transaction to delete an item 101
5.13 Sequence diagram of the transaction to create a new item 101
5.14 Sequence diagram of the transaction to update an item 102
5.15 Page and navigation bar 105
5.16 Class diagram of the pages hierarchy 106
5.17 Class diagram of the main classes of the PWA 108
5.18 Sequence diagram navigating to a new page 110
5.19 Class diagram of the grid pages 111

5.20 Class diagram of the list page 113
5.21 Class diagram of the API service from the PWA 115
5.22 Sequence diagram getting categories through AJAX 116
5.23 Class diagram of the map page 117
5.24 Class diagram of the geolocator 119

6.1 File structure of the API source code 124
6.2 File structure of the PWA folder 128
6.3 Example of hashed files in the service worker 130
6.4 Sequence diagram of the network-first handler 131
6.5 Sequence diagram of the cache-first handler 132
6.6 Sequence diagram of the “fastest” handler 133
6.7 Pseudo-code for the period-grouping algorithm 135

List of Tables

3.1 Project charter 12
3.2 Actors complexity evaluation 19
3.3 Use case complexity evaluation criteria 19
3.4 Use case complexity decided values 19
3.5 Technical factors score 20
3.6 Environmental factors value 21
3.7 Estimated effort per role 22
3.8 Estimated costs per role 22
3.9 Final estimated cost 22
3.10 Project schedule 24
3.11 Risk exposition measurement 29

5.1 Entities dictionary 84
5.2 Relations dictionary 85
5.3 Attributes dictionary 87
5.4 Resources of the API 89

7.1 Status of the non-functional requirements 142

Chapter 1

Introduction

Since I wrote my first line of code when I was eight years old in a real-time, strategy game, I realised there was something very powerful behind that bunch of chips. Since that, I have always seen technology, and specifically computer science, in two main ways. On the one hand, I see them as an unstoppable, incredibly complete access to creation, since computers allow us to create unimaginable things which go far beyond what any other previous form of creation did. Inside a computer, we can create highly-complex universes with its own internal rules, the limits are only in the imagination. On the other hand, I perceive computer science as a medium to take those universes out of the screens and change the real world around us, to solve the problems we face day by day, helping humanity step by step to get a bit closer to a higher level in our existence.

The reason for which I studied this degree is because I think there is some kind of power, maybe even *magic*, behind what we, engineers, do and I believe we can use our knowledge to make life better for everyone around. Said this, the project I have been working on aligns perfectly with my vision of information technologies, as it has an important social implication and can improve the lives of many people.

In this section the context of the project, its general objectives and some related information will be explained to give a rough, general idea about the system, its implications and its motivations. Also, the contents of this document and structure are clarified.

1.1 Context

In the past years, Europe has been facing a massive increase in the immigration affluence, caused partially (but not exclusively) by the outbreak of the war in Syria [5]. More than three hundred sixty thousand migrants risked their life to get to Europe in 2016 and more than one thousand had drowned in the Mediterranean at July 3, 2018, according to UNHCR [24]. One of the most common ways for people to arrive is through the eastern Mediterranean, refugees coming mostly from Syria make their way along Turkey and cross the sea to Greece. Although applying for asylum can be a lengthy procedure, many

applications are been received, many more than the countries are accepting so far. In 2015, EU countries offered asylum to 292,540 refugees, in contrast with the more than a million ones who applied for it on the same year [19].

As indicated in the presentation of the UNINTEGRA project (further explanation later), more than fifty thousand refugees and asylum appliers are blocked in Greece and live in precarious conditions. In September 2017, the European relocation program of two years ended. As UNHCR reports, only a few refugees are able to leave Greece and move to a third country. The long time living in refugee camps or in irregular places have affected negatively these refugees in social and psychological terms. Despite of the efforts of the Greek and international authorities, the projects of the NGOs and the volunteers, refugees do not have complete access to the basic social and health services yet. There is a clear need of innovative projects to help refugees to get back into an active life, looking for a social integration and inclusion and putting special attention to special necessities like LGTB people, children in risk or mothers which are alone with kids.

It is because of this that the University of Santiago de Compostela (Spain) [26] leads, since 2017, the project UNINTEGRA [25], in collaboration with the National and Kapodistrian University of Athens (Greece) [20], Universidade do Minho (Portugal) [27], Fundació Acsar (Spain) [9], Fundació Universitària Balmes (Spain) [10] and Concello de Santiago de Compostela. The UNINTEGRA project is also co-funded by the Asylum, Migration and Integration Fund (AMIF) [6] of the European Commission. This project has the main aim to intervene in the migration process by providing people in refugee sites and centres with the necessary resources to empower them, encouraging their participation in the host community. The duration of the project is 24 months and its period of implementation runs from 1st December 2017 to 30th November 2019, and it is divided in six work packages which affect different areas inside the general objectives.

The project this document treats about corresponds to the fourth work package: the App4Refs application. This working package is led by the Fundació Acsar and counts with the collaboration of the Department of Electronics and Computer Science in the University of Santiago de Compostela, who offered it as a final project.

1.2 The project and my participation

The objective of the project is to develop an application to support social participation, inclusion and integration of refugees, and the way to achieve this is by handing them free and useful tools for field-detected needs through a cell phone app. There are three main areas the app will turn around: legal information, useful local resources location and community interaction. Due to technical and financial reasons, the scope of the project will be reduced to the metropolitan area of Athens in first place, but never discarding to progressively extend its use to a wider scope, eventually reaching the whole Europe.

The smart phones have a crucial importance for refugees and they are, in fact, one of their most precious possessions, not only because they allow them to keep in contact

with their acquaintances in their origin countries, but also because they are the most powerful tool to move around in an unknown, unfamiliar country. App4Refs has the intention to exploit this technology they already use in a daily basis to provide them with the information they need to manage in their situation, allowing them to feel more empowered and increasing their security and the security of those around them.

My participation in the project consists in the complete development of the application following the requirements specification which Javier Ideami [15], working for the Fundació ACSAR, provided me and always under his supervision. The process requires designing the code and implementing it, as well as helping in its deployment. Although I am obligated to work in the project till October, this final degree project has to be deposited in July. Given this, it will be elaborated, in a first version, treating only about the alpha version of the application, the general planning of the whole project and how the development is taken in these first months (since this alpha version should be released in July, coinciding this with the final data to deposit my final project of the degree). This documentation is intended to be extended at the end of my labour in order to be also a useful reference to future developers working on the project.

The developed application consists of a progressive web application (commonly named by their acronym, *PWA* [13]) programmed in the HTML5 environment and supported by a server back-end compound of a relational database and a JSON API [7] which provides access to it. The project includes working with languages like HTML, CSS, JavaScript, PHP, MySQL or Python and making use of a wide variety of modern libraries and technologies like Bootstrap [3], AJAX [29], npm [21], gulp, MariaDB [18], PHPUnit or the Google Maps API [12]. It requires also of some knowledge of object-oriented design and programming, how modern web pages work, how to design responsive designs and connect them with an online API or how to design and access databases in web environments. Although a little bit of knowledge about web security is involved in the project (as it will count on an administration panel which will allow to modify the contents of the application through internet, requiring then the implementation till some level of OAuth2.0 [14] in the API and some care for the dangers that user input into an internet-exposed application entails), this knowledge is not completely reflected here since the implementation of this panel is beyond the scope of the alpha version, so the exposure to attacks is somehow reduced by now.

The decisions about the required functionality, the design of the interface or related processes have been done by the Fundació ACSAR together with the responsible engineer (previously mentioned, Javier Ideami) and it is out of my hands to change anything of them. Still, I have actively participated, suggesting changes and improvements and providing some feedback based on the studies I am about to complete.

1.3 This document

The purpose of this document, as previously stated, is not only to serve as a memory of my final degree project, but also as a reference for future implicates in the project so they

can check it to understand some questions of the design and development. It is, then, a complete documentation of the work I developed in the project and the decisions that have been made (only until the release of the alpha version, in this first elaboration of the document). Together with this document, the complete code is delivered with extensive comments which explain the low-level details of the implementation, including standard ways of code documentation as, for example, phpDocumentor [22]. Diagrams and other figures are present to clarify the key aspects of the text, always as a supporting content but never replacing the textual description. Each chapter is preceded by a very short paragraph explaining what the chapter contents will be, intended to facilitate the search of concrete information on the whole document.

Chapter 2

Technologies

In this chapter the technologies involved in the development of the project will be specified and explained to provide a base for the rest of the document.

2.1 The web technologies

The Internet was a revolutionary idea, the concept of a global communicated world with access to any information we could need. We owe a lot to internet, half of our lives pass searching and learning information in it, but old/classic web pages were quite limited. When I started programming web pages, before my degree, very few standards existed. The web was a slow, quite-ugly thing which nobody trusted to do serious, really interactive stuff.

Times change, nowadays we all have an smart phone with very small capabilities in our pockets, doing almost all the stuff we need through the Internet. The web has became faster, more reliable and much more standardised. It could not be other way, this continuous evolution of the web applications getting closer and closer to the native ones would end, some way or another, leading to a complete blurring of the limits between ones and the others. With the birth of the Progressive Web Apps, we are finally living the end of the separation between web and native applications.

The progressive web applications are just web applications, developed in the classical technologies of the web (HTML, CSS and JavaScript) but with the capability to run in your device *camouflaged* like native applications. They are able to store information, access some device features that used to be considered only available to native apps, keep running on the background when they are not visible and much more. All of this is possible thanks to a big effort to provide the web with modern technologies able to dissipate those differences and make the web more powerful. The main advantage of this technology is clear: they have a really high portability, as they can run wherever a compatible navigator can run. Moreover, they allow our web page, our mobile application and our desktop application to share a unique code, giving the user the chance to choose

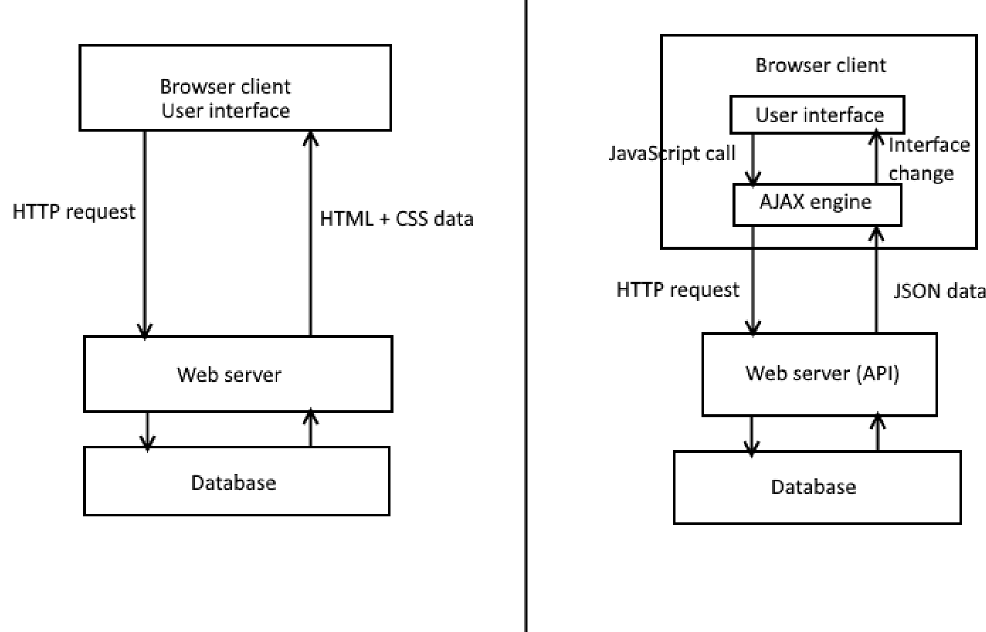


Figure 2.1: At the left, the classical way to make web applications. At the right, the architecture of the webs designed with AJAX.

how they would like to be able to use them, installed on the device or accessing them through the web browser.

The traditional architecture of a web application consists in a client and a server exchange. The client connects to the server through an internet connection, using the HTTP protocol, and receiving back HTML documents which the browser can interpret to show the web page. This HTML documents can include, embedded or referenced, several other kinds of code. The most common ones are CSS and JavaScript. CSS allows us to specify the style of the display for the web page, while JavaScript allows us to script the page (since HTML is just an structural language), allowing us to achieve a more complete interaction with the user. The classical flow of request for complete pages and answers has a small flaw: we have to receive a new complete document each time we want to change the content of just a part. This breaks the user experience and makes the movement through the web slow. To solve this, we use AJAX.

AJAX, acronym for *Asynchronous JavaScript And XML*, is a technology designed to allow the creation of web pages with asynchronous connections to the server which run in a back layer, allowing the user to keep a consistent, fluid experience. Over the years, despite of preserving the name, the use of XML has been drastically reduced and the most common format for the data exchange is now JSON (this is why our API in the back-end will provide, in principle, a JSON interface). The figure 2.1 shows the difference between the classical mode to make web pages and the AJAX method.

To get to the progressive web applications using only AJAX is not enough, there are

still a couple of barriers to break. Not only a fluid experience is necessary, we also need the option to keep the application running in the background or installing it to launch it as a native application. This can be achieved thanks to the Service Workers introduced in the recent versions of JavaScript and the *manifest* of the web page, a JSON file which allow us to control how the application will behave when installed as a native one (where will it appear, which images use as the icons, etc). Also, to imitate the behaviour of native apps certain UI design decisions should be taken: the web should be responsive, adapting to the user's behaviour and environment based on screen size, platform and orientation.

All of this technologies are used in our system.

2.2 About the back-end

The back-end of the system consists of a server which will provide the data for the app to work with through the API. The reasons to choose this structure are explained in the section 5.1. The PWA can be served by the same server which serves the API or by a completely different one, since both codes have been developed on separated structures. In our case, to simplify the deployment the server which provides the PWA will be the same which provides the access to the data.

The server used in the back-end will be Apache. Apache is currently the most common software to serve web applications. This position in the market makes it a really good choice as we can expect any future engineer which would work with the project to have knowledge (at least a basic one) into the technology. Our application, anyways, would be easy to port to any other server able to interpret and run PHP with only a couple of adjustments in some specific files.

As mentioned above, we will use PHP to develop our application. PHP is a very known language which was conceived as a *template language* for web pages in C, but which ended being a programming language powerful enough to develop complete systems (without programming C at all). Despite of its bad fame, PHP is a very powerful language to create functional web pages with a low-medium complexity in short times of development. Some years ago, PHP started to accept its condition of complete language incorporating object-oriented capacities and some other sophisticated features. Nowadays, it is a multiparadigm language in continuous improvement which offers an easy initiation for beginner programmers at the same time it gives tools for professional programmers to work with. This object-oriented characteristics were crucial to be chosen as a technology of development, since there was interest in making an object-oriented design of the system to develop.

The data will be provided by a relational database and the language under use will be MySQL. There are several database servers able to work with MySQL or a subset of the MySQL instructions nowadays and all of them will be able to run our system (with maybe a few adaptations, but without big compatibility problems). In the case of our

deployment, we will use MariaDB, as it is a very famous server developed by the original developers of MySQL and guaranteed to stay open source.

2.3 Building the front-end: Gulp

As previously explained, the web applications turn around three main, very closely related technologies: HTML, CSS and JavaScript. To get a professional, comfortable way to work with our application it is desirable, once we get to a certain complexity, to have the ability to separate the final result served to the client from the development. For example, we would like to have different JavaScript files for each class we manage, but we better serve only one minified (without any unnecessary characters or any comments) JavaScript file to the client in order to save connection resources and reduce the loading time of the page. For this one and many other similar *building tasks* we use Gulp. This software is built in JavaScript, it runs over Node.js and allows us to define a series of rules to build our project, deciding how to process the files to generate a final distribution folder. We use this technology to minify the HTML, CSS and JavaScript of our page, as well as some other minor tasks. One important, very special task, is the one performed by babel, a plugin designed to translate *modern JavaScript* (ECMAScript 6) into older JavaScript code, making our application tons of times more compatible to different users while allowing us to program with a modern, more optimal tool. The figure 2.2 shows an example of execution of Gulp in a terminal to build a project, it is possible also to find a detailed tutorial on how to build the PWA in the section A.5.

```
PS C:\USC\4 Cuarto curso\TFG\app4refs\PWA> gulp
[22:06:18] Requiring external module
[22:06:24] Using gulpfile
[22:06:24] Starting 'build'...
[22:06:24] Starting 'generate-service-worker-dist'...
Caching static resource "dist/css/style.min.css" (29.6 kB)
Caching static resource "dist/index.html" (2.02 kB)
Caching static resource "dist/js/javascript.min.js" (10.9 kB)
Total precache size is about 42.5 kB for 3 resources.
[22:06:24] Finished 'generate-service-worker-dist' after
[22:06:24] Starting 'dist-javascript'...
[22:06:26] Finished 'dist-javascript' after
[22:06:26] Starting 'dist-css'...
[22:06:26] Finished 'dist-css' after
[22:06:26] Starting 'dist-html'...
[22:06:26] Finished 'dist-html' after
[22:06:26] Finished 'build' after
[22:06:26] Starting 'default'...
[22:06:26] Finished 'default' after
```

Figure 2.2: Example of execution of Gulp to build a project.

2.4 Integrated development environments and documentation

To develop the application the main-used development environments were PhpStorm and PyCharm from IntelliJ and Notepad++ developed by GNU. To try the web application in a local server XAMPP was used, and to transfer the files for deployment FileZilla was the FTP client. To create this documentation, StarUML and LaTeX (with the web page *Sharelatex*) were used.

Chapter 3

Project management

In this chapter all the relevant subjects relative to the management of our project will be explained. The requirements specification made as part of this project management is shown in the next chapter. First we will display the project charter which represents the act of constitution of the project, then we will show the scope management, defining the scope of the project in general lines (as it will be precised in the requirements specification) and including a diagram of the Work Breakdown Structure (WBS). We will define the project life cycle and schedule all the necessary tasks, providing a Gantt chart to help to visualise them. We will also make an evaluation of the expected cost of the project and finally elaborate a risk analysis to define strategies to confront the different possible risks.

3.1 Project charter

PROJECT CHARTER			
PROJECT NAME		**DATE**	**AREA OF FOCUS**
App4Refs		2018-May	Mobile application development
BUSINESS CASE		**SCOPE**	
There is need for a mobile application to enhance refugees integration and provide them with useful information. The application will (1) show locations with general services, (2) show locations which provide help, (3) provide useful internet resources and (4) help to find recreative areas to enhance integration.		**IN SCOPE**	**OUT OF SCOPE**
		Mobile / Web application development	Data gathering
		Server-side development	
		KEY DELIVERABLES	
		Project requirements	Project memory
MEASURABLE TARGET/GOAL		**TIMELINE**	
Decide between PWA or native		**ACTIONS/MILESTONES**	**TARGET DATE/STATUS**
Server-side API		Alpha version	01/08/18
Client-side		Beta version	01/09/18
Server-side panel		Final release	01/10/18
TEAM MEMBERS			
NAME	**FUNCTION**		
Jordi Tolrà	Direction		
Javier Ideami	Design & supervision		
David Campos Rodríguez	Development		
ASSUMPTIONS/CONSTRAINTS		**RISK PLANNING**	
The developer will be attending classes during the first months		Time might be very tight for the developer until first release because of the high load of work in his current studies, miscommunication between the team and the focus groups or too superfluous tests might cause a bad reception of the application by the users.	

Table 3.1: Project charter.

3.2 Scope management

3.2.1 Work Breakdown Structure

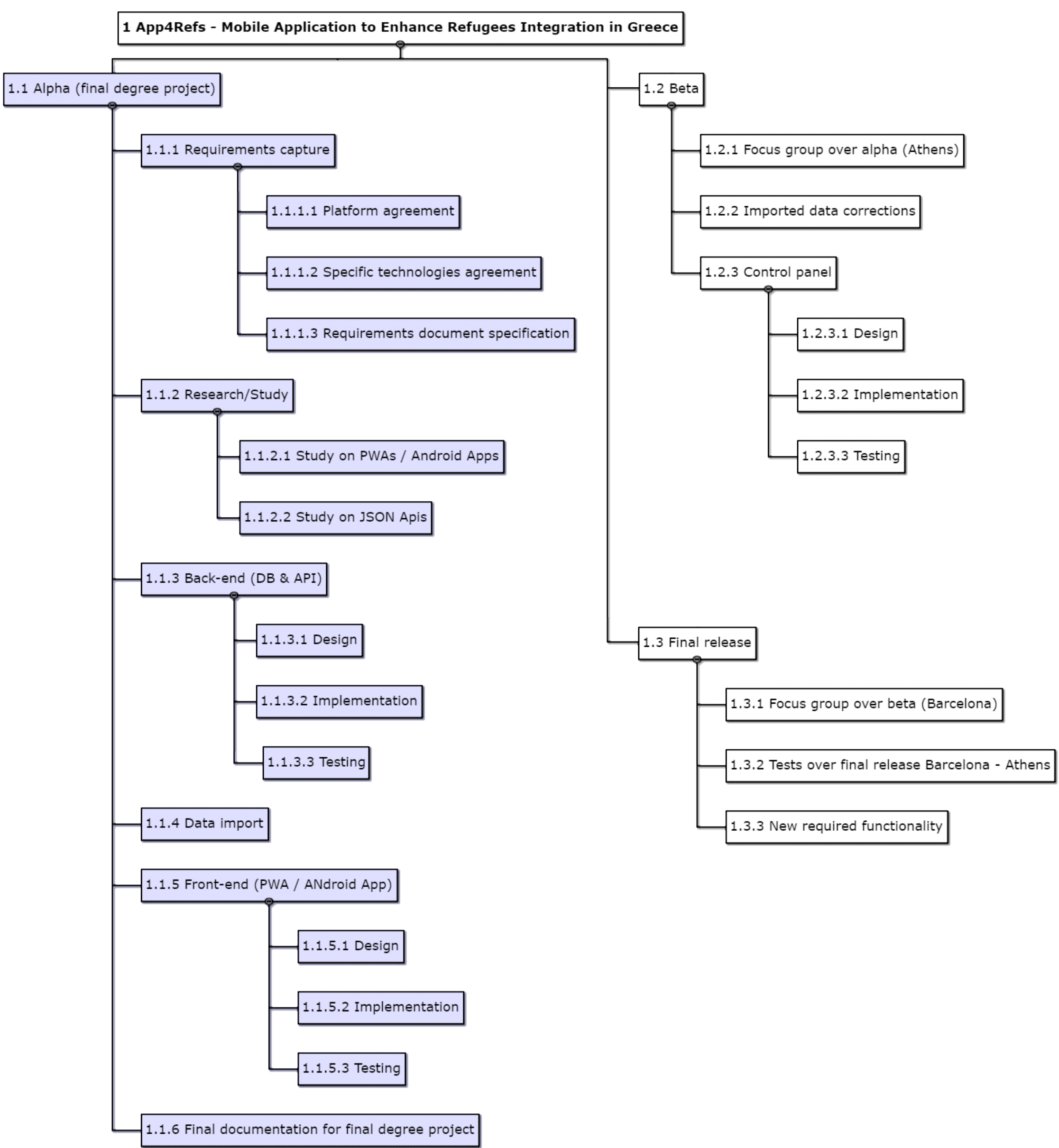


Figure 3.1: Work breakdown structure of the project.

To give an initial idea of the scope of the project we use a work breakdown structure which can be seen in the figure 3.1 and defines clearly the processes the project has to go through from my entrance to the project until the end of it. As indicated in [23],

the WBS gives a global vision from the most general to the most specific of the project scope and shows the necessary work to complete the project. In this sense, one thing has to be clarified: although I have only until the 28th of July to present my final degree project, I will work for the project until October. For this reason, my final degree project refers to the alpha version of the software exclusively, whose elaboration corresponds to this period of time (check section 3.8 about schedule management). Despite of this, the project management and requirements capture, which are elaborated previously, cover the whole project (it is not the case of the design and implementation chapters, which refers only to the alpha version). In the WBS, the alpha, beta and final release parts and the work to do on each of them are separated clearly.

3.2.2 Description of scope

The system to develop is going to be a mobile application with the capacity to provide useful information for the refugees in Athens, Greece, about their situation and how to move around and integrate into the community. The project is limited to the mobile application (which might be a web page or a native application) and the server to provide the information for it. The system will also provide a control panel where it will be possible (for authorised users) to upload new data or update the existent one in the application.

3.2.3 Out of the scope

It is out of the scope of this project to gather the information to display, while we decide the information which will appear in big groups, it is a Greek team of people the one which will search the concrete information about the places and services displayed. It is, however, part of the project to load this initial information into the system to be displayed.

3.2.4 Project restrictions

There are a few restrictions the project, in general, has to accomplish:

- The application will work on mobile phones, both Android and iOs. It might as well work on browsers if we finally decide to make a progressive web application.

- All the project should be oriented by the focus groups in Athens.

- The project will be finished by the end of October 2018.

About the part which I, as developer, will make on the whole project there is an extra constraint: I have to adapt to the interface designed by Javier Ideami and stick to it. I may, however, propose alternatives and suggestions whenever I consider it necessary.

3.3 Project life cycle

We will follow an adapted rational unified process (RUP) to accomplish the project. The rational unified process is a software development process compound by 4 phases: the inception, the elaboration, the construction and the transition; and defines the following *six best practices* for modern engineering:

- Develop iteratively, with risk as the primary iteration driver.
- Manage requirements.
- Employ a component-based architecture.
- Model software visually (UML).
- Continuously verify quality.
- Control changes.

It is an adaptable process which suggest an incremental approach at the lower level. In consequence, the project will be developed in an incremental approach. In each iteration we design, develop and test a series of functionality. There are three main releases: the alpha, the beta and the final release. After each release, we will work with a focus group testing the application and providing feedback from the release, which we will use to correct mistakes from the previous version in the following one. It also will allow us to decide whether to add or not new functionality to the system. More information on RUP can be found in [17]. We chose it because it is adaptable to our necessities and very commonly used in big projects (that is why it is studied during the degree). It also fits better with a final degree project by the documentation it generates.

Although the project is initially presented in a "classic" project management philosophy and methodology and because of this we will stick to the basic principles in project management described in [23], we take some ideas from the Agile philosophy to keep a fluid, change-receptive environment. The main characteristics that we want to take are:

- Develop close to the "business people": we will work very close on each version with the director of the Fundació ACSAR, Jordi Tolrà, to make sure we are going in the right direction. Several meetings will be planned (depending on his agenda) and we will make use of video-conference and chat technologies to keep in contact with him the whole time of the development. For this reason, Jordi is included as a member of the team in the project charter.
- Self-documented code: despite of elaborating this complete documentation because it is a final degree project, we advocate for a self-documented code. The code will be intensively commented and as clear as possible to ensure future developers which might need to work on the project will be able to do so producing effective code in a very short period of time.

- Face-to-face conversation as the most efficient and effective way to communicate: as previously stated, meetings and the use of video-conference and chat technologies will be fundamental for a good communication.
- Working software is the primary measure of progress.
- Continuous attention to technical excellence and good design.

Still, and even though we would like to have means of doing it, we do not have a common place of work, available schedules and other requirements to follow a complete Agile philosophy (even more, this extensive documentation that is, with reason, required to evaluate a final degree project goes, in essence, against the Agile principles somehow), so we want to leave clear we are not following an Agile method (like they could be Scrum, XP, CI...), and in consequence there will not be in this document user stories, planning poker or other Agile-specific methodology. More information on the Agile philosophy can be found in [1]. Notice the sixth edition of [23] includes now an "Agile Practice Guide", too.

The inception phase, the stage we are currently dealing with, will allow us to plan project, elaborate the requirements, have a clear vision of what we want to do in a general sense and preview times, costs and risk. The elaboration and the construction phases, dedicated to the conceptual design of the whole system and the corresponding implementation respectively, will be split into the two main components of our system (the DB with the API and the front-end application). Since the API can be developed without the front end and the front end still needs some discussion, we prefer to break both phases into two halves and design and implement the back end first and then design and implement the front end. The transition phase will consist on the beta testers work and the final corrections to get the product to the final users, as well as taking the system through an evaluation period after the final release (which will be in October).

3.4 Configuration management

The configuration management allows us to ensure the integrity of the products of our work. It is, in consequence, an essential part to elaborate any project. We have to keep control of the documentation of the final degree project (which is, at the same time, the documentation of the development of the project) and the code itself.

To control the code itself we use one version control mechanism of the several options available in the market. In our case, we have chosen Git since it is the one which all the members of the team are used to and it is really comfortable. To avoid loosing the data we will keep our local git repository synchronised with GitHub[1]. The three marked releases (alpha, beta and final) will be tagged in the repository so they can be accessed

[1] The url of the GitHub repository is `https://github.com/david-campos/app4refs` and it will be publicly available at some point in time, probably before the final release.

easily at any point in time. Although the project will be developed by only one person, Git is a reliable tool to develop in team.

To keep track of the documentation you are reading, and since it is created with LaTeX, we elaborate it in *ShareLaTeX*, which is an open-source online service and will keep our files even if we suffer from some computer breakdown. Other files related to the project are preserved by the Fundació ACSAR, and also in the personal Google Drive of the developer. Each important communication is performed through email to keep a register of them.

3.5 A short note on the requirements

Although the requirements capture was elaborated at this point in the project inception phase, it is long enough to require of its own chapter. For this reason, we decide to move the requirements capture document to the chapter 4. In the following analysis we will make use of the use cases specified during the analysis, but by now we will not explain them in detail. We will introduce here only their brief-style description for it to serve as a previous introduction. For a detailed explanation on the use cases, refer to the section 4.2.4.

This section serves as a quick-reference to check the purpose of each of the use cases of the system. There are 10 main use cases and 3 other ones which are different versions of the same main one (3.1, 3.2, 3.3). All the use cases displayed are based on the meetings established with the Fundació ACSAR director and the engineer directing the project.

UC1. Get category items: Actors: `User`, `Greek Team Member`. The system back-end receives a request for the items of a specified category and it returns the list of the solicited items.

UC2. Get subcategories: Actors: `User`, `Greek Team Member`. The system back-end receives a request for the subcategories of a specified kind and it returns the list of the categories of that kind.

UC3. Check locations: Actors: `User`, `Maps Provider`. The users choose the type of categories they want to check, so the categories are shown to them. After this, they pick a category to see the items inside it. They can pick an item to check its position on the map and how to get there or check the position of all the items in the category. This use case is a generalisation of 3.1, 3.2 and 3.3.

UC3.1. Get general information: Actors: `User`, `Maps Provider`. This is a child use case for `UC3. Check locations`, in this case the user chooses to visualise locations which can provide him/her with general information.

UC3.2. Get services: Actors: `User`, `Maps Provider`. This is a child use case for `UC3. Check locations`, in this case the user chooses to visualise locations which can provide him/her with services (like WiFi places, showers,...).

UC3.3. Find leisure places: Actors: `User`, `Maps Provider`. This is a child use case for `UC3. Check locations`, in this case the user chooses to visualise locations

where he/she can spend his/her spare time and integrate with the cultural activity of the city.

UC4. Check links: Actors: `User`. The user is offered some categories to pick one, when one is picked the items of that category are shown. No map available, only web links and the other information about the item.

UC5. Check online help: Actors: `User`. The user is offered some categories of different topics he/she may need help with. When picking one he/she is redirected to a web page where he/she can find that kind of help.

UC6. Make emergency call: Actors: `User`. The user clicks the emergency call button and a call to the emergency services is started.

UC7. Update item: Actors: `Greek Team Member`. The member of the Greek team, logged into the control panel, selects the items he/she wants to update. He/she can change the values of the items and the associated opening hours as he/she pleases (except the item identification, which remains invariable) and finally the system validates and stores the data.

UC8. Delete item: Actors: `Greek Team Member`. The member of the Greek team, logged into the control panel, selects the items he/she wants to delete. After a confirmation dialog the items are deleted.

UC9. Add item: Actors: `Greek Team Member`. The member of the Greek team, logged into the control panel, selects to add a new item. He/she fulfils a form with all the required data and add all the opening hours periods they wish. He/she also provides an image for the item. The system validates the data, re-sizes and crops the image into the right format and stores everything.

UC10. Log in: Actors: `Greek Team Member`. A member of the Greek team, introduces a user name and a password to access the control panel. The system registers a session for this user and certifies them they are logged in.

3.6 Use case point analysis

In this section we will make the UCPA to estimate the effort required by the project. This is a usual process to forecast the software size when developing systems in UML in the context of a rational unified process. We start this by assigning to each actor a weight as defined in the table 3.2 and add them all together for each use case in accordance with the following formula[2]:

$$\text{UAW} = \sum_{a \in \text{actors}} \text{weight}(a)$$

Our system has three actors, two of them humans (`User` and `Greek Team Member`), so we assign them a weight of 3. We assign 1 to `Maps Provider` cause map providing systems have clear specifications. So our final UAW is **7**.

[2] UAW stands for *unadjusted actor weight*.

Complexity	Weight	Criterion
Simple	1	Another system which offers an API
Medium	2	Another system with another kind of interface
Complex	3	Human interaction

Table 3.2: Actors complexity evaluation and criterion.

Complexity	Weight	Criterion		
		Work	GUI	Entities
Simple	5	Simple	Simple	1
Medium	10	Medium	Medium	2
Complex	15	Complex	Complex	>2

Table 3.3: Use case complexity evaluation and criteria.

Now we need to evaluate the complexity of our use cases, to do this, we use as reference the table 3.3. With the help of said table, we assign a punctuation of 5, 10 or 15 to each use case based on its complexity. This complexity comes determined by the complexity of the work, the GUI and the number of entities implicated. When we have all the values we add them up to get the **UUCW**[3]. The table 3.4 shows the values considered for each of the factors in the criteria and the final use case complexity decisions, as well as the final UUCW.

Use Case	Work	GUI	Entities	Final	Weight
UC1	Medium	Medium	>2	Medium	10
UC2	Medium	Simple	>2	Medium	10
UC3	Medium	Complex	>2	Complex	15
UC4	Medium	Medium	>2	Medium	10
UC5	Simple	Medium	>2	Medium	10
UC6	Simple	Simple	0	Simple	5
UC7	Simple	Simple	2	Simple	5
UC8	Simple	Simple	2	Simple	5
UC9	Simple	Simple	2	Simple	5
UC10	Complex	Medium	?	Medium	10
				UUCW:	**85**

Table 3.4: Complexity and values decided for each use case.

[3]UUCW stands for *unadjusted use case weight*

Then, we proceed with the calculation of the technical complexity factor (TCF). To make this calculation we need to assign a score between 0 (the factor is irrelevant) and 5 (the factor is essential) to each of the elements of the list of 13 factors described in the table 3.5. The final TCF factor is calculated with the following formula:

$$\text{TCF} = 0.6 + \frac{\sum_{f \in \text{factors}} \text{weight}(f) \times \text{score}(f)}{100}$$

The table 3.5 shows the assigned scores and the resulting sum, too.

Factor	Description	Weight	Score	weight × score
T1	Distributed system	2	0	0
T2	Performance	1	2	2
T3	End user efficiency	1	3	3
T4	Complex internal processing	1	2	2
T5	Reusability	1	3	3
T6	Easy to install	0.5	4	2
T7	Easy to use	0.5	5	2.5
T8	Portability	2	5	10
T9	Easy to change	1	4	4
T10	Concurrency	1	0	0
T11	Special security features	1	1	1
T12	Direct access 3rd parties	1	2	2
T13	User-training facilities	1	0	0
				Total: 31.5

Table 3.5: Technical factors to estimate the TCF.

As the table shows, the sum of all the weights multiplied by the scores is 31.5, so our final TCF is $0.6 + 31.5/100 = 0.915$.

Last, but not least, we have to obtain the environmental complexity factor. This value takes into account all the factors relative to the project, context, etc. that are not manageable and can influence the project. There are eight factors which we must evaluate from 0 to 5. These values are multiplied by the given weights and added up to obtain the environment factor (EF). The final ECF is calculated with the following formula:

$$\text{ECF} = 1.4 + (-0.03 \times \text{EF})$$

The table 3.6 shows our calculation for the environmental complexity factor, with the 8 factors defined by the method. We assign a value of 2 to the familiarity with the development process because it is something the developer has studied deeply along his degree, but it is not something he has experience with. We assign 2 also to part-time

Factor	Description	Weight	Value	weight × value
E1	Familiarity with development process	1.5	2	3
E2	Part-time workers	-1	2	-2
E3	Analyst capability	0.5	4	2
E4	Application experience	0.5	3	1.5
E5	Object-oriented experience	1	4	4
E6	Motivation	1	5	5
E7	Difficult programming language	-1	2	-2
E8	Stable requirements	2	3	6
				Total: 17.5

Table 3.6: Weights and values for the environmental factors to calculate the ECF.

workers because the developer will be a part-time worker for a bit more than the first third of the project. We assign 4 to the analyst capability, 3 to application experience (since the developer has been working on the domain, the web applications, for more than 10 years already, with some pauses to attend other preferences), 4 to object-oriented experience because object-orientation is a topic which is very long treated during the degree, a 5 in motivation because it is a really motivating project given its social implication, 2 to the E7 because the used programming languages are very wide-known and common, but still not 0 or 1 because JavaScript is a little bit tricky and finally 3 to stable requirements because although there is a very documented project specification, the meetings with the director suggest there is a wide margin for changes. With all of this, we obtain a final ECF of $1.4 + (-0.03 \times 17.5) = 0.875$.

To obtain the final UCP for the project we apply the formula: UCP $=$ (UUCW$+$UAW)$\times$ TCF $\times$ ECF, which in our case corresponds to a final UCP of $(85 + 7) \times 0.915 \times 0.875 \approx 73.66$.

We can estimate now the effort for the application. If we take a value of 22 man hours per use case point (this value should be adjusted in base of previous projects where this estimation was applied), we obtain a final effort of 1620.52 man hours.

3.7 Cost management

To estimate the costs of the project we will take our estimation of the effort as the basis. First of all, we will estimate an effort per role of the team taking a percentage of the total effort, as shown in the table 3.7. We estimate most of the work in this case will be for the developer, while designer and supervision will be a 30% and direction a 15%; thus, we estimate for the developer an effort of 55%. This makes sense because most of the tasks are assigned to him.

Role	% Effort	Hours
Project director	15%	243.078 h
Designer & supervision	30%	486.156 h
Developer	55%	891.286 h
	Total:	**1,620.52 h**

Table 3.7: Estimated effort for each of the roles.

Then, we define a cost per hour for each of the roles in our team, as shown in the table 3.8. We estimate the social security and other associated taxes as a 40% of the net salary, and workplace as a fixed cost of 200 €. We include in this fixed cost all the immobilised material relative to the workplace, such as the wear of the table, computer, cleaning, etc.

Role	Net salary	×Effort	SS & others	Wrkpl.	Cost
Project director	12.6 €/h	3,062 €	1,224.80 €	200 €	4,486 €
Designer & supervision	10 €/h	4,862 €	1,944.80 €	200 €	7,006 €
Developer	8 €/h	7,130 €	2,852.00 €	200 €	10,182 €
				Total:	**21,674 €**

Table 3.8: Estimated costs for each role.

We multiply add to the result a 15% more of structural expenses (like water, electricity and similar indirect costs) and a 10% for contingencies. Resulting in a total of **27,000 €**, as shown in the table 3.9.

Cost	Structural	Contingencies	Final
21,674 €	3,251.10 €	2,167.40 €	27,092.50 €

Table 3.9: Final estimated cost of the project.

Time after the estimation shown above, we decided to contract to the maps service an extra limit of 100 € / month over the 200 $ that Google[4] gives you for free, until May 2019. Since the release of the application, in October 2018, it makes a total period of 8 months. We estimate that this quantity of connections is hardly reachable and that we will have more than enough this way[5]. In the worst case, 8 months with 100 € / month is still totally inside the 2,000 € estimated for contingencies, so the cost of the project remains unchanged.

[4]Later in the project, we decided to use Google Maps to display the maps in the application.

[5]Google Maps API has now a pay-as-you-go billing model, so you pay in function of the number of requests attended. The price for each request to the Directions API is 0.005 USD. Google allows for an initial margin of 200 USD per month, which is equivalent to 40,000 requests.

3.8 Schedule management

When planning the schedule of our system we have to keep in mind we are strictly restricted by some external impositions on the required dates and the expected functionality for them, which in the end cause the project to be a bit unbalanced over time. The first release, the alpha counts with too much functionality and overlaps with the developer finalisation of studies (which have a very huge workload) so it gets really heavy, while the final release counts with a very relaxed month dedicated to minor corrections and (maybe) some extra required functionality lately discovered when testing the first two releases with the focus groups. Some of the releases already have a group of functional requirements associated. The imposed dates and associated requirements are as following:

- The alpha release, on August 1, 2018. It should include the complete implementation of FR-1, FR-2, FR-7, FR-8, FR-9, FR-10, FR-11, FR-12, FR-13, FR-14.
- The beta release, on September 1, 2018. It should include the complete implementation of FR-3, FR-4, FR-5, FR-6. Posterior modifications amplified this with the functional requirement FR-15 (check the risk RS-13 in the section 3.9.3 for more information).
- The final release, on October 1, 2018.
- The project end data, on October 31, 2018.

As described in the section 3.3, our project is divided into the common four phases of RUP, inception, elaboration, construction and transition, but the two central parts (elaboration and construction) will be made for each component (the API and the front-end) at each time and divided along the different releases. We organised the periods as represented in the table 3.10[6]. We assign around a 10% of the whole available time to the inception, around an 80% for elaboration and construction and a bit more than the 10% to the transition at the end. We start the table on May instead of April because it is when the developer really enters the project (although it technically starts in April, previous time was spent trying to arrange the initial meetings).

For the inception phase we have three tasks: the requirements capture, project management, and the study of the technologies. We assigned most of the time to requirements capture and project management as they are considerably larger. We are, at the moment of writing these lines, in this stage of the project.

The alpha version will contain only the API and the front-end. The API has less functionality and it requires, in consequence, a bit less dedicated time. Also, we estimate the Greek team gathering the data should have the initial data ready around the time to implement the front-end, so we need certain time to load this data into the system. Testing has very few time dedicated, since after each release there is a long period of

[6] Although the project runs, in fact, from much sooner, we start the schedule in the dates when I, the developer, start to work on the project.

Stage	Component	Task	≈Duration	Dates
Inception		Requirements / proj. mgm.	14 days	01/05 - 18/05
		Study on the technologies	3 days	21/05 - 23/05
Alpha	API	Design	10 days	24/05 - 06/06
		Implementation	10 days	07/06 - 20/06
	Front-end	Design	10 days	21/06 - 04/07
		API data loading	2 days	∈(09/07 - 15/07)
		Implementation	12 days	04/07 - 22/07
	API+FE	Testing	7 days	23/07 - 31/07
		Finish documentation	5 days	23/07 - 27/07
Beta	API+FE	Testing & correcting alpha	23 days	01/08 - 31/08
	API	Design log in	3 days	01/08 - 03/08
		Implementation	5 days	06/08 - 10/08
	Panel	Design	5 days	13/08 - 17/08
		Implementation	7 days	20/08 - 28/08
	API+Panel	Testing	3 days	29/08-31/08
Final	All	Testing & correcting beta	20 days	03/09 - 28/09
		Design	10 days	03/09 - 14/09
		Implementation	10 days	17/09 - 28/09
Transition		Transition	23 days	01/10 - 31/10

Table 3.10: Project planned schedule. Some of the tasks will be executed in parallel.

testing with the focus group in parallel with the development of the next one, also unitary testing development time is counted as part of the implementation. In consequence, the testing indicated at the end of each of the parts is a prerelease testing which does not need a lot of time but just some checks. Towards the beta release, the control panel has to be added to the system. We give most of the available time to this because the other parts should only require of a few adaptations and to complete a bit. We put *testing and correcting bugs* with a period of four weeks, meaning this will be done in parallel with the other tasks along the whole month and should not require too much time. We also assign a bit more of time for the API as adding the control panel will require to implement the log in with OAuth. In the final release we give around a 50% of the time to elaboration and 50% of the time to construction, but this should be a relaxed stage cause it depends mostly in whether the previous tests show new functionality needs to be added or changed.

Finally, we reserve the last 4 weeks, the last month, for the transition. We will make use of all the period that comes after the final release to make a general diagnosis of the

project and follow its results in the first month, maybe applying some extra bug fixes.

The figures 3.2, 3.3, 3.4 show the Gantt diagram of the planned schedule, it has been split into three parts (because of the paper limitations) and rotated so it can be read better, the numbers of the rows are preserved in each fragment to facilitate reading, allowing to keep track of the task we are on.

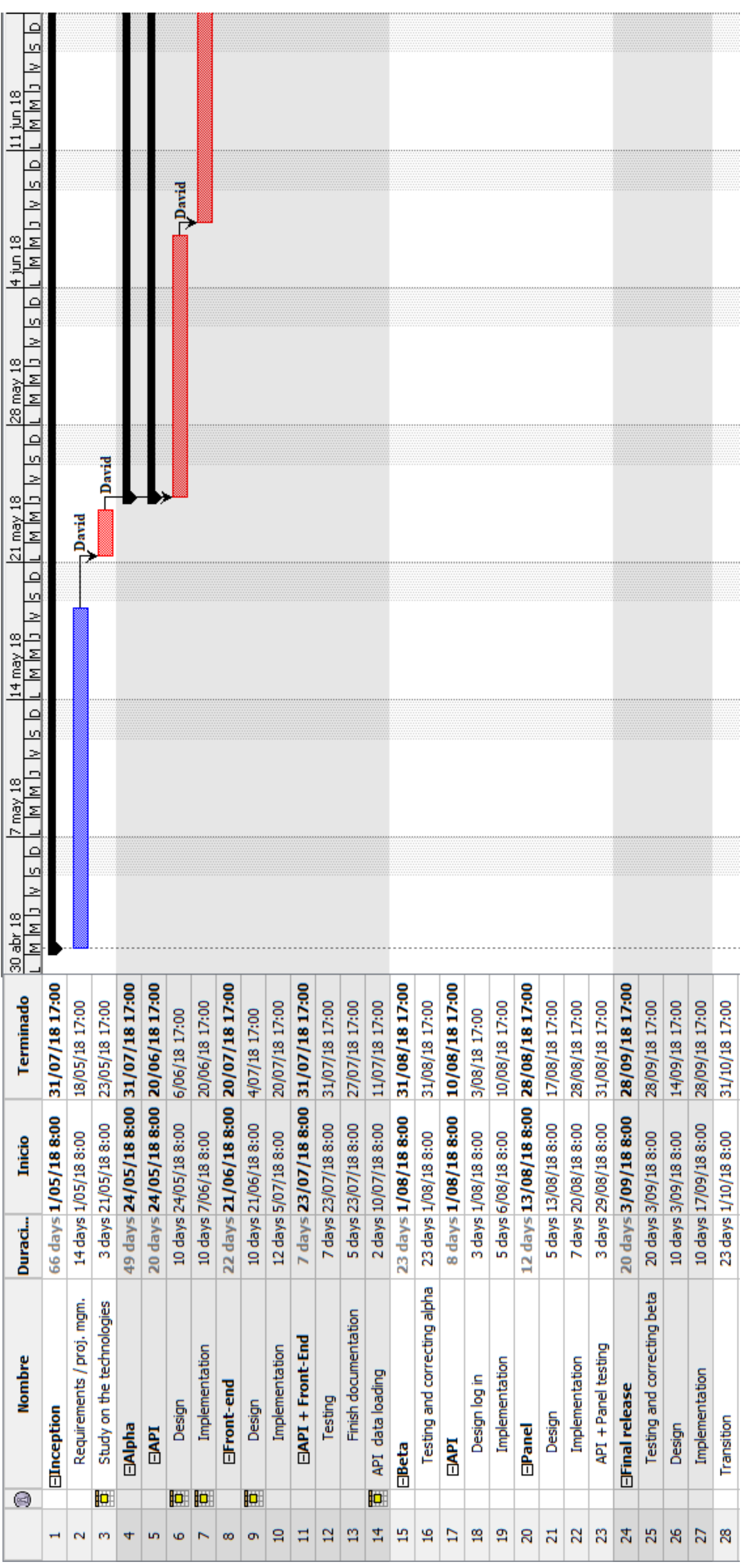
3.9 Risk management

To finish this chapter of project management, we perform a risk analysis. This is of vital importance for any project, and specially for such a long and complex one. A bad management of unexpected risks can cause the project to be cancelled so it is important not only to identify the possible risks, but also to define effective methods to prevent, eliminate or minimise their impact. This point is crucial to avoid getting out of the budget or missing the deadlines.

3.9.1 Risk measures

To evaluate the different identified risks, we will use the following measures during their analysis and cataloguing:

- **Likelihood**: whether and how much the risk is likely to happen or not. We will assign the following values:
  - **Unlikely**: the risk is very unlikely to ever happen.
  - **Possible**: the risk probability is between a 30% and a 70%.
  - **Likely**: the risk is very likely to happen.
- **Impact**: an evaluation of the extra time, effort or costs which the risk would suppose to the project development. We will assign the following values to this measure:
  - **Tolerable**: the repercussion of the risk would be almost unnoticeable. Only small variations on the planning would be needed. The final product shall be unaffected.
  - **Severe**: the repercussion of the risk is important for the project. It can cause big variations on the planing or the resulting product.
  - **Critical**: the repercussion of the risk can cause the cancellation or complete failure of the project.
- **Exposition**: a combination of the previous factors to try to give an associated priority to each one of them. We will evaluate it with a number from 1 to 5. The table 3.11 shows the combinations of said measures and the resulting value for each of them.



	Nombre	Duraci...	Inicio	Terminado
1	**Inception**	66 days	**1/05/18 8:00**	**31/07/18 17:00**
2	Requirements / proj. mgm.	14 days	1/05/18 8:00	18/05/18 17:00
3	Study on the technologies	3 days	21/05/18 8:00	23/05/18 17:00
4	**Alpha**	49 days	**24/05/18 8:00**	**31/07/18 17:00**
5	**API**	20 days	**24/05/18 8:00**	**20/06/18 17:00**
6	Design	10 days	24/05/18 8:00	6/06/18 17:00
7	Implementation	10 days	7/06/18 8:00	20/06/18 17:00
8	**Front-end**	22 days	**21/06/18 8:00**	**20/07/18 17:00**
9	Design	10 days	21/06/18 8:00	4/07/18 17:00
10	Implementation	12 days	5/07/18 8:00	20/07/18 17:00
11	**API + Front-End**	7 days	**23/07/18 8:00**	**31/07/18 17:00**
12	Testing	7 days	23/07/18 8:00	31/07/18 17:00
13	Finish documentation	5 days	23/07/18 8:00	27/07/18 17:00
14	API data loading	2 days	10/07/18 8:00	11/07/18 17:00
15	**Beta**	23 days	**1/08/18 8:00**	**31/08/18 17:00**
16	Testing and correcting alpha	23 days	1/08/18 8:00	31/08/18 17:00
17	**API**	8 days	**1/08/18 8:00**	**10/08/18 17:00**
18	Design log in	3 days	1/08/18 8:00	3/08/18 17:00
19	Implementation	5 days	6/08/18 8:00	10/08/18 17:00
20	**Panel**	12 days	**13/08/18 8:00**	**28/08/18 17:00**
21	Design	5 days	13/08/18 8:00	17/08/18 17:00
22	Implementation	7 days	20/08/18 8:00	28/08/18 17:00
23	API + Panel testing	3 days	29/08/18 8:00	31/08/18 17:00
24	**Final release**	20 days	**3/09/18 8:00**	**28/09/18 17:00**
25	Testing and correcting beta	20 days	3/09/18 8:00	28/09/18 17:00
26	Design	10 days	3/09/18 8:00	14/09/18 17:00
27	Implementation	10 days	17/09/18 8:00	28/09/18 17:00
28	Transition	23 days	1/10/18 8:00	31/10/18 17:00

Figure 3.2: Gantt diagram, part I.

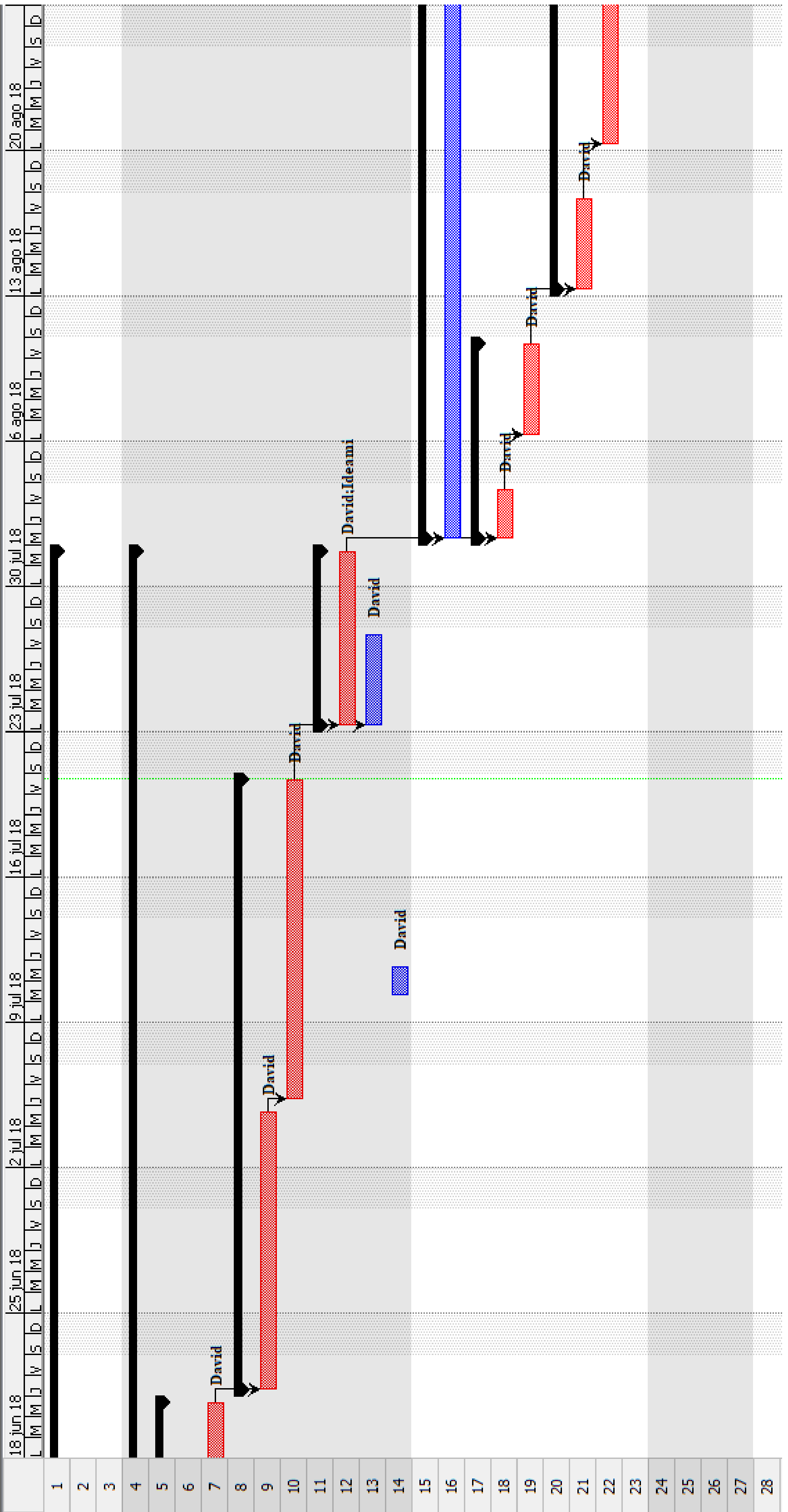


Figure 3.3: Gantt diagram, part II.

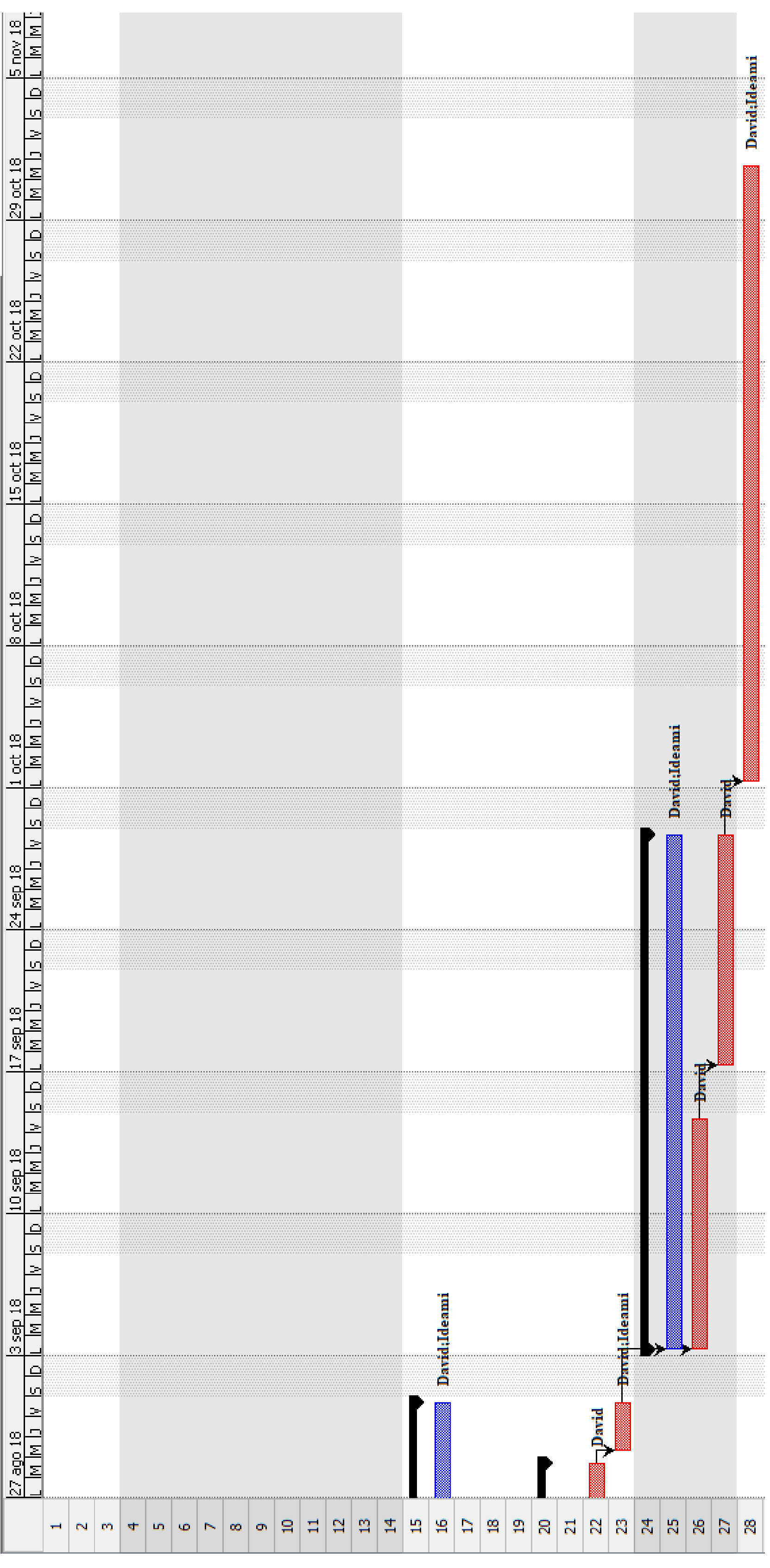


Figure 3.4: Gantt diagram, part III.

		Likelihood		
		Unlikely	**Possible**	**Likely**
Impact	**Tolerable**	1	2	3
	Severe	2	3	4
	Critical	3	4	5

Table 3.11: Risk exposition measurement given the probability and the impact of a risk.

3.9.2 Risk palliation strategies

Four kinds of strategies will be applied to each risk, depending on the risk itself and the situation:

- **Prevention**: these strategies try to preview the apparition of the risk before it happens.

- **Minimisation**: these strategies are intended to minimise the impact of the risk when it appears.

- **Contingency plan**: these strategies shall enter in action once the risk has already happened with the aim to nullify or reduce it.

- **Acceptation**: the risk is assumed with its consequences, this applies to any of the risks when no other strategy has the expected effect.

3.9.3 Risk specification

Finally, we present the risk specification, sorted by exposition and likelihood. We include some indications on the risks which have appeared and a brief description of what has been done about them.

RS-1	**Impossible functionality**	Exposition: **5**
Description:	The current web technologies do not allow to implement some of the required functionality.	
Indicators:	• A functionality cannot be implemented because it is not possible in web applications.	
Contingency plan:	Try to reformulate the functionality to provide a similar one, in other case discard it.	
HAPPENED ON 17/07/18	On FR-12, it is not possible to start a call directly from a web page in many phones, the system will only show the call window with the telephone of emergencies, but another click will be required to make the call to the emergency services. NFR-13 is still accomplished.	
	Likelihood: **Likely**	Impact.: **Critical**

RS-2	**Excessively optimist planning**	Exposition: **4**
Description:	The planning assigns too short time to the tasks and, in consequence, a delay in the development appears. It is especially likely to happen during the development of the alpha release.	
Indicators:	• Tasks extending further than the planned time. • Non-implemented requirements on the release deadlines.	
Minimisation:	Check planning at the end of each week and readjust.	
Contingency plan:	Postpone requirements to next release, cancel them or compensate with another parts of the planning.	
HAPPENED ON 22/07/18	Front-end implementation has gone a bit further than planned and documentation for the final project needs still a lot of work. We did not move any functionality to the beta release but the testing will be performed only between the 30 and 31 of July.	
	Likelihood: **Likely**	Impact.: **Severe**

RS-3	**Complications on deployment**	Exposition: **4**
Description:	The access to the deployment server is not complete or there are problems with the URLs to serve the application or related issues.	
Indicators:	• The deployment generates lots of problems and a big loss of time.	
Prevention:	Design both the API and the PWA in a way that it is easy to deploy, add configuration elements to allow the system to adapt to the deploy environment.	
HAPPENED ON 02/07/18	Complications with the deployment of the database because of incompatibilities due to an old version of MariaDB in the server. The deployment of the database delayed for some days more than expected.	
HAPPENED ON 18/07/18	Problems with the Service Worker which provides the pre-cache and dynamic cache because of a misconfiguration in the redirection at the root of the domain. Solved given access to the root of the domain to the developer.	
	Likelihood: **Likely**	Impact.: **Severe**

RS-4	**Undetected risks**	Exposition: **4**
Description:	Something bad happens and the risks specification did not have it in account.	
Indicators:	• New risks appear during the evolution of the project • A negative situation which affects the development appears and it is not contemplated in the risks management.	
Prevention:	Perform a rigorous revision of the risks identification.	
Contingency plan:	Id it has not happened yet, edit the risks management adding the new detected risk as soon as possible.	
HAPPENED ON 12/7/18	New risk RS-8 added.	
	Likelihood: **Possible**	Impact.: **Critical**

RS-5	**Excessive schedule pressure**	Exposition: **4**
Description:	The schedule makes excessive pressure over the developers getting to affect the quality of the product. This requirement is suggested by Capers Jones as one of the most serious in [16].	
Indicators:	• The developer suffers from serious stress problems caused by the project. • The designs or the code loose their quality. • Sticking so hard to the planning that the topic of the time bounds interrupts the realisation of the other processes.	
Prevention:	Do not stick too hard to the planned schedule.	
Contingency plan:	Stop the process. Remake the schedule. Do extra hours. Take some time away from the project (maybe leave some task to do for the weekend).	
	Likelihood: **Possible**	Impact.: **Critical**

RS-6	**Compatibility problems**	Exposition: **3**
Description:	The functionality does not work on some of the required systems in the way it should.	
Indicators:	• The focus groups show certain functionality does not work on certain systems.	
Prevention:	Make use of highly-standardised technologies for every vital functionality. Use tools to enhance CSS and JavaScript compatibility.	
Contingency plan:	Correct the compatibility problems.	
	Likelihood: **Likely**	Impact.: **Tolerable**

RS-7	**Incorrect data format for importing**	Exposition: **3**
Description:	The Greek team gathering the initial data did not respect correctly the indicated format.	
Indicators:	• The automation on loading the data fails.	
Prevention:	Give clear and concise indications about the format of every field in the document and the image files names.	
Contingency plan:	Correct the data manually.	
HAPPENED ON 12/07/18	The data provided by the Greek team did not follow the indications they were given correctly. There is a lack of images in some elements and the naming convention has not been strictly followed. They have been corrected manually.	
	Likelihood: **Likely**	Impact.: **Tolerable**

RS-8	**Unavailable information**	Exposition: **3**
Description:	Some of the information required for the application might not be found.	
Indicators:	• There is one or more categories with no content. • There is one or more items with no address, coordinates, schedule... • There is one or more items with no image.	
Contingency plan:	Discard the categories or the items, try to look for them manually in Athens, look for alternatives or, in the case of the images, make placeholder images.	
HAPPENED ON 12/7/18	There are no items for the *Transport* category, we changed this category to be a link pointing to Google Maps.	
HAPPENED ON 12/7/18	There are no icon images for some of the items gathered, placeholders have been made.	
	Likelihood: **Likely**	Impact.: **Tolerable**

RS-9	**Illness**	Exposition: **3**
Description:	The developer is ill and he cannot continue developing the project for some time.	
Indicators:	• The developer suffers a sick leave.	
Minimisation:	Keep the hygiene and security standards on the working place.	
Contingency plan:	Re-plan the project as possible. Pact with the team a new distribution of the functionality over the releases, new release dates or/and extra hours.	
HAPPENED ON 05/07/18	Stomach problems. Two days away from work, compensated working harder on the weekend. The stomach continued hurting for two weeks more, but much less so it was possible to work anyways.	
	Likelihood: **Possible**	Impact.: **Severe**

RS-10	**Release out of time**	Exposition: **3**
Description:	One of the planned versions is released later than planned. This would be totally unacceptable since public meetings are already arranged.	
Indicators:	• A version of the application should be released 3 days ago and it is still unreleased.	
Prevention:	Adjust requirements for the releases as the project advances.	
	Likelihood: **Unlikely**	Impact.: **Critical**

RS-11	**Failures in data leading to dangerous places**	Exposition: **3**
Description:	The data stored in the system includes physical locations and the project will be working with people in risk of social exclusion. The failures in the locations or the routes could lead the users to dangerous places for them.	
Indicators:	• A user is led to a place which suppose danger to his integrity by he application.	
Prevention:	Check the data of the application before the release, make sure no dangerous places are given by mistake.	
Minimisation:	Use a legal disclaimer rejecting the responsibility over the data and the consequences of its use.	
	Likelihood: **Unlikely**	Impact.: **Critical**

RS-12	**Change in the deadlines**	Exposition: **3**
Description:	The imposed deadlines are changed to a sooner date, reducing the time available and invalidating the project plan.	
Indicators:	• The director indicates the deadlines for the releases have changed.	
Minimisation:	Develop tasks in the minimum (reasonable) time possible. If we go quicker than planned readjust the plan to leave space later for changes.	
Contingency plan:	Re-elaborate the planning, change the functionality assigned to each release or make extra-hours if necessary.	
	Likelihood: **Unlikely**	Impact.: **Critical**

RS-13	**Requirements amplification**	Exposition: **2**
Description:	The director demands new requirements or the focus groups show some kind of unexpected functionality is needed.	
Indicators:	• There are new functionality required by the director or focus groups.	
Minimisation:	Try to keep close contact with the director to assimilate changes as soon as possible. Construct a good design with high tolerance to changes.	
Contingency plan:	Design and implementation. There is time for new functionality development contemplated in the initial planning before the final release.	
HAPPENED ON 08/06/18	During database implementation it was decided not to check one of the constraints of the relational model in the database, and let it to the control panel interface, adding a new requirement to ensure this is achieved. More information in the section 6.2.1.	
	Likelihood: **Possible**	Impact.: **Tolerable**

RS-14	**Hardware failure**	Exposition: **2**
Description:	The hardware used for the development suffers some breakdown and it is not available.	
Indicators:	• The system used for the development does not work.	
Minimisation:	All files of the project should be backed-up externally.	
Contingency plan:	Call to the technical service, use public-accessible computers.	
	Likelihood: **Possible**	Impact.: **Tolerable**

RS-15	**Bad requirements specification**	Exposition: **2**
Description:	The requirements are ambiguous or too far from the director's perception of the application expected functionality.	
Indicators:	• Difficulties or misunderstands appearing when talking about the requirements. • Undesired/unexpected functionality presented to the director.	
Prevention:	Elaborate requirements in a detailed, exhaustive way. Keep close contact with the director about the product development. Talk explicitly about all the requirements with him.	
Contingency plan:	Correct the requirements making sure they are adequate and clear now.	
	Likelihood: **Unlikely**	Impact.: **Severe**

RS-16	**Incorrect design**	Exposition: **2**
Description:	The design created for some part of the system contains mistakes which difficult the implementation.	
Indicators:	• Design smells. • Ambiguous/confusing diagrams. • Something results impossible to implement in the way it is designed.	
Prevention:	In the inception phase, ensure a complete and clear analysis of the domain and give a clear general idea of the solution. Check several times all the diagrams of the design in detail. Make usage of design patterns.	
Minimisation:	Keep the diagrams clear, simple and readable. Implement the application with people with deep knowledge on the environment to be able to find alternative solutions to mitigate the problem.	
Contingency plan:	Correct the design during implementation.	
	Likelihood: **Unlikely**	Impact.: **Severe**

RS-17	**Silver bullet syndrome**	Exposition: **2**
Description:	There is the belief that the next big change in the procedures or resources will miraculously solve all the current problems of the project. This requirement is suggested by Capers Jones as one of the most serious in [16].	
Indicators:	• The management insists that some given functionality will change completely the success of the product between the focus groups.	
Prevention:	Pay attention to the results with the focus groups and establish direct and realistic solutions to each of the detected problems.	
Minimisation:	Keep a realistic vision of the project and its limitations.	
	Likelihood: **Unlikely**	Impact.: **Severe**

RS-18	**Problems to understand the technologies**	Exposition: **1**
Description:	The developer has problems to understand some of the employed technologies, delaying the development process.	
Indicators:	• Some technology required to implement the system is not well understood.	
Prevention:	Take advantage of the time indicated in the planning to get deeper in the knowledge about the technologies with which the team feels less comfortable.	
Contingency plan:	Check reference books or online help on the topic. If the technology affects a small section of the project, look for alternatives.	
	Likelihood: **Unlikely**	Impact.: **Tolerable**

Chapter 4

Requirements capture

In this chapter we will analyse the context of the application and the requirements it should accomplish. It will allow us to have a general vision of the solution, also described here, and provide us with the necessary elements to correctly evaluate our progress through the development as well as orientating us towards a more adequate solution to satisfy all the parties involved.

4.1 Context study

The study of the context of our system is a previous step before starting defining all the requirements for it. We need to make sure we have a clear understanding of the circumstances which surround the project and how they can influence the success or failure of our final system.

4.1.1 Current situation

As the director of the ACSAR foundation indicated, and based on the enquiries made in-place in Athens, the refugees are currently using their smart phones for their everyday life. The main application they use is Google Maps, which allows them to move around in the city (and, as some of them told us, even helped them to cross some borders). The main problem they suggest is the lack of an application which gathers together all the information they need to face the problems which being a refugee in a foreign country, and more specifically in the city of Athens, entails. Searching on Google can be a very hard task, as information can be too few, making it very difficult to find meaningful results, or too much, making it very difficult to filter it and pick the reliable, better one.

To this, we have to add that the refugees suffer from a deep isolation from the rest of the society. There are some serious problems of racism and their conditions living in irregular places aggravate the situation making it even harder for them to integrate. They need not only an app to provide them with access to the general information about

their situation and useful resources, but also with some kind of point of access to start integrating with the community, participate in recreational stuff and be closer to the other citizens.

Finally, the refugees need some tool to make them feel more empowered, as their conditions in the host country have had a negative psychological influence on many of them. Having the right information at their fingertips would allow them to feel more confident, which at the same time would increase their approach to the rest of the society around them and take them back to a more active, participatory life in the host country.

4.1.2 Project vision and opportunities

Our project is intended to be that app that will provide the refugees with the necessary information to feel safer, empowered and integrated in their new environment, getting them closer to common activities. To this purpose we need the application to be as simple to use as possible, since we want the users to quickly adapt to it. We should not forget we are dealing, in most of the cases, with people living in very hard conditions which need an application to help them rather than something to learn how to use and waste time on. In this same sense, it would be nice for the application to be understandable by any person without the need of a specific language, since we worked in place with people from very different countries with completely different languages and cultures. We have to be really careful in this sense, as some symbols might be misunderstood by some people depending on their culture and traditions. The application is also expected to have the ability to run correctly in many different devices.

There are a couple of points where we have the opportunity to improve the experience of our target users:

- The information provided, since right now they have no trustworthy, official font of information available. Our application will be recommended in the refugee centres and NGOs, we count with the collaboration of some institutions in the city and the support of the European Union. In this sense, our application offers an initial point of confidence which we can enhance keeping very up-to-date, well filtered and reliable information.

- A close work with the refugees. Other existing systems failed mainly because they did not count with their target users during the development of the system. Our system development will be completely oriented by the enquiries and tests made in-place with the real target users. We want them to feel comfortable with the application, achieve an easily understandable interface and to know what their real needs are.

This two main points are crucial to the success of our project.

4.1.3 Interested parties

In this section we will try to identify all the parts interested on our system and what is their role and interests in the project.

- The refugees: they are clearly interested parts, as they are the target users of our application. Their role in the system is to be the final users. Their main interest is to improve their situation and life, getting access to more specific, reliable and useful information.
- The other citizens: they are indirectly interested, since the use of the app will improve the approach between the refugees and the rest of the society, improving the peace and quality of life for both of them.
- The NGOs: they are needed to provide the information and the real help our application gives access to. Their role in our system will be to provide information and to attend the refugees which will get to them through our system. Their main interest is to make their labour more effective and accessible.
- The Greek Forum of Migrants: their role will be to recollect the necessary information to the first release of the application and to keep the information updated through an administration panel that we have to design in the latest versions of the system. They are interested because helping refugees is a main part of their job.
- The Athens/Greek government: they are both politically and economically interested in improving the situation of the refugees. Both because they need to give a good, solidarity image and because they are interested in the increase of the level and quality of life of their citizens. Their role will be also to provide information for the system and answer the requests of the refugees who get to them using our application. Their interest is to achieve a better coexistence between citizens and refugees, and get these last ones to participate more in the social activity, creating economical and social wealth for everyone.
- The European Union: the project can have big implication, and its scope can be gradually increased to reach the whole Europe. Because of this, the European Union is also an interested party. They are the main font of financing (through the AMIF) of the project.
- The Fundació ACSAR: they are the ones who led the working package designed to develop the app. Their role is to take all the major decisions and provide the team with the general vision, the context and all the necessary information. They are interested because helping refugees is the main intention for the creation of the foundation.
- The Universidade de Santiago de Compostela: the university of Santiago de Compostela is the leader of the whole UNINTEGRA project, in which this project is

embedded. In this application, in concrete, its role is to provide a team to perform the implementation of the application. It is interested because it is an educational institution and so it watches over the advance and improvement in the quality of life for everyone.

- The development team: our role is to perform the implementation of the application following the guidelines stated by the Fundació ACSAR to turn the application into a real thing. We are interested because we want to help people with my job and, in my case, because it is my final project which (hopefully) will allow me to get my degree too.

4.1.4 Systems to interact with

Our system will dispose of all the required information, only needing to interact with Google Maps to display the location information in a way the users are already used to. It will need also to redirect to the pages of the NGOs or the public services in some concrete cases.

Google Maps API provides a really simple interface for several platforms which will allow us to show high-quality, interactive maps in our project with a very low programming effort.

4.2 Requirements specification

Now that we understand the context of our system, we can define the requirements for it. We will start by the stakeholders, then define the objectives of the system, the general vision and how it fits with our proposed solution, the functional and non-functional requirements and finally a conceptual data scheme to work with in the design phase.

4.2.1 Stakeholders of the system

We understand here stakeholders as those interested parties directly related to the use of the system. We want to know who they are and what they expect from our system to develop the right solution for them. To get the cooperation of the stakeholder we need to keep in mind that a person will only use a software if the reward for using it is bigger than the cost, and the identification of this two elements is the main aim of this section. The following ones have been identified based on the interested parties of the context analysis:

- The refugees: they are the final user of the application. They expect the system to be easy to use and to provide reliable, useful information they can understand. The reward for them is really big, since they get to move around and integrate in

the unknown city they are now faster, while the cost will be as low as we get the interface to be simple and easy to understand.

- The greek forum of migrants: they will be the people who will use the administration panel to keep the information in the system up to date. This is crucial to provide the final users with useful information. They expect the system to be easy to use an to allow them to modify the categories, links and other information of the system. The reward for them is to expand their work and reach more people in a more effective way. The cost to pay will be learning how to use the system. In this sense, the system will succeed if they find it easy to use and flexible enough to offer the information they want to provide in a clear way.
- The Fundació ACSAR: they are the leaders of this project. They expect it to be cheap and to be developed fast and effectively. The main reward for them is the project itself, while the cost depends on the time of development.

For the sake of simplicity, we decide to group all the refugees under a single group, although there are lots of different people with subtle differences in their interests. This shouldn't suppose a problem since they all share the main interest and cost.

4.2.2 Goals

We describe now the objectives of the system. It is important for this goals to be SMART[1]: specific, measurable, achievable, relevant and time-bounded.

1. **Providing the refugees with useful, updated information**: all the information for the system should be valid in the moment it is accessed, all outdated information should be out of the system. The users should declare the information to be useful in the enquiries. The information should be available from the same moment the application is released and cover, at least, the following aspects:
    - Relevant places location: public office for social service, unemployment, insurance, etc. Also other relevant places decided with the help of the Greek Forum of Migrants.
    - Information after arrival: how to get a tax number, insurance, how to open a bank account, how to get a health insurance, etc.
    - Geo-located services: food, clothes, showers, health, LGTB+ information, children services, women, disability...
    - Leisure information: free activities where users can enjoy their time with their family and participate in the life of the city they are in. Divided in different areas (like cultural activities, educational activities, open places, sports...).

[1]There are several definitions of SMART, depending on the author, we will stick to the indicated ones. The differences between them are minor and they share a common vision of the task.

- Links to useful resources on the web about education, refugee information, public and emergency phones and sites, established communities, etc.
- Quick button to access the emergency number in case of needing it.

2. **Offering the information in the most international way possible**: using written languages should be avoided whenever possible. Iconography and imagery should replace the text. When writing is unavoidable several languages should be available, at least English, Greek and Arab. In the enquiries after the release all the users should declare to have understood all the sections and items of the system without any problem.

3. **General use by part of the refugees**: we expect, at least, a 30% of the refugees in Athens to make use of the application in the two months following the release of the application.

4. **Refugees integration and inclusion**: we expect, in the six months after the release of the system, the statistics of refugees inclusion and integration to get a substantial increase. We expect also the numbers of refugees in precarious conditions to be reduced at least a 10% in the same time.

4.2.3 General vision of the proposed solution

Based on the data and guidelines provided by the Fundació ACSAR and the previous stated information, the following system structure is proposed: a Progressive Web Application implemented in JavaScript and HTML5 which uses AJAX to connect with the back-end, a server with all the information of the application, stored in a relational database and provided through a JSON interface. Using a PWA will have a number of key advantages:

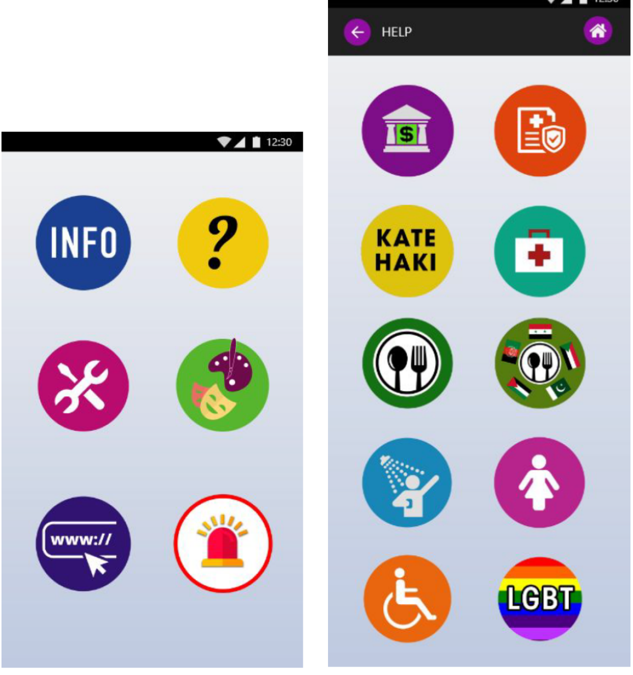
- The app will work in all devices: mobile phones (iPhone, Android, etc), laptops, desktops, etc.
- The app can be accessed from any web browser without having to install anything.
- The app can be added to the home screen of the device to interact with it as a native app would do.
- Using the cache features, the app can be used offline when there is no internet connection.

During the focus group, it was found that most refugees have phones that are less than two years old. This means that their mobile phone web browsers are mostly updated to recent versions and will work well with this technology.

The app front-end will have the following main areas. We include some mock-ups provided by Javier Ideami, designer of the interface and supervisor of the project, to indicate how the resulting product should look like at first[2].

- **Home page**: The home page will be composed of 6 icons that link to the main 6 areas: Information, services, help, leisure, links and the emergency call page. Responsive design will be needed to ensure the icons fit nicely distributed in one single screen on any device, both in portrait and landscape configuration. The figure 4.1a shows the intended appearance of this screen.

- **Emergency call page**: This will be a confirmation screen with a large button. Clicking that button should trigger a phone call to the 112 emergency phone number.



(a) Example of home screen.

(b) Example of help area.

Figure 4.1: Home screen and help area mock-ups.

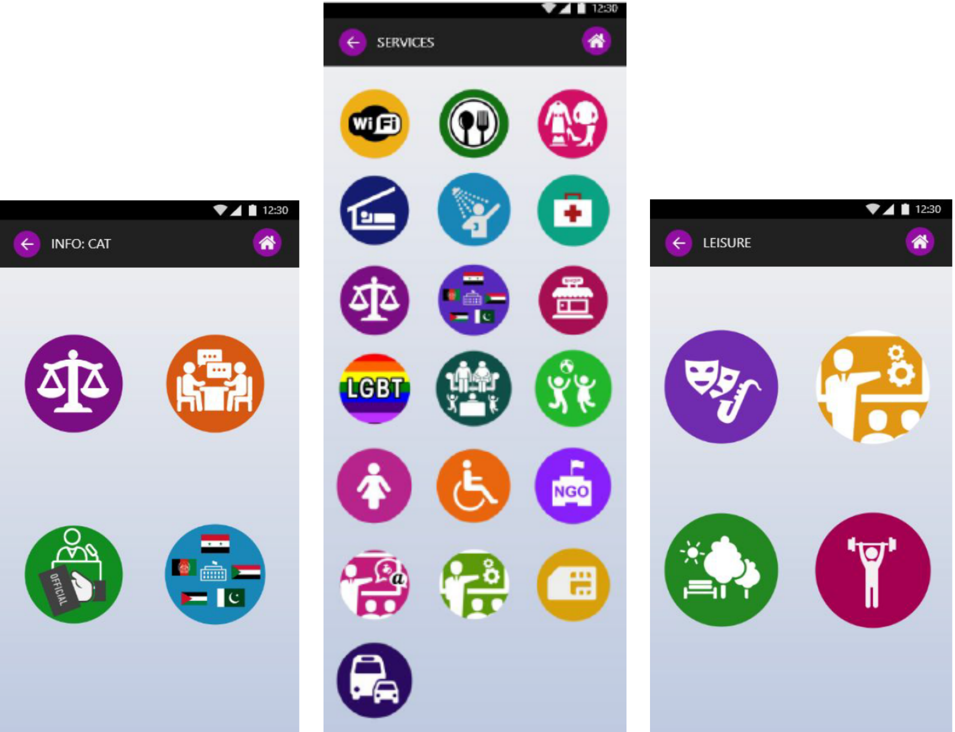
- **Top bar**: On the top of all the categories and location areas there is a bar that always contain the same options:
  - On the left, a left arrow button to go to the previous screen.

[2]The icons and some other minor details have changed a lot along the development, these are the first projected screens to give an indicative idea

  - On the right, a home icon which takes the user back to the home screen.
  - On the right of the left arrow button, a text which contains the title of the current section. It is important for this text not to extend beyond the available space, using ellipsis when necessary.

- **Help area**: It will contain a grid of icons linking directly with the web pages the Greek team will provide. When clicking on these icons, external web pages will open on separate tabs of the browser. The figure 4.1b exemplifies the desired look and feel for this screen.

- **Information, services and leisure areas**: These pages will display a grid of icons which represent different categories. When clicking one of these categories, you access a page which displays each location that fits with the category. The figure 4.2 shows the expected look of these three screens.



(a) Example of information area.

(b) Example of services area, if the screen is not able to fit all in an scroll bar should appear.

(c) Example of leisure area.

Figure 4.2: Information, leisure and services area mock-ups.

The locations will be displayed with the following format:

- **An icon**, an image in square format and small size. The Greek team will provide this images. When developing the control panel for the Greek team to keep the information updated back-end code will be needed to automatically re-size and format the uploaded images to be used as icons for the locations.
- **The name of the location**, displayed on the side of the icon.
- **The address of the location**, displayed just below the name.
- Some icons indicating whether the service provided in the location is **free or not** and the **supported languages** in the link (see below).
- **A web link** under the icons, clicking on them should open an external tab on the web browser with that link.
- **The opening hours** of the place, displayed in a human-readable format. When it is not possible to get opening hours (because it is purely virtual, for example) this line will be substituted by a phone number to call to require an appointment.
- **A map button**, clicking on it will open the embedded Google Maps screen showing the location and how to get there.

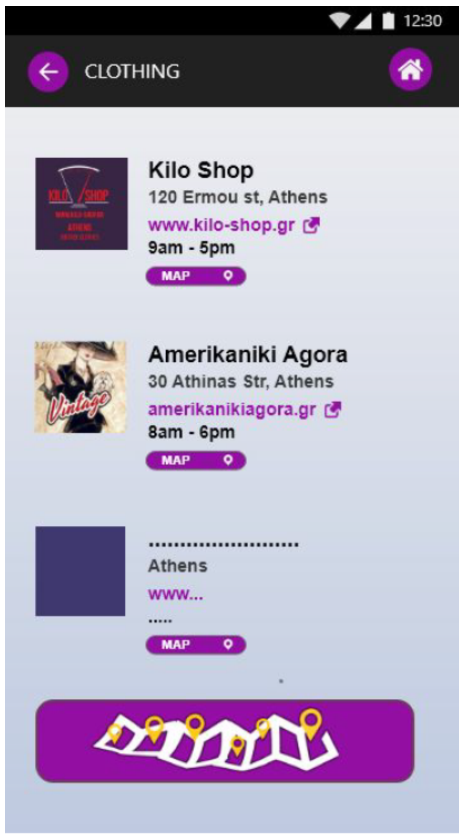
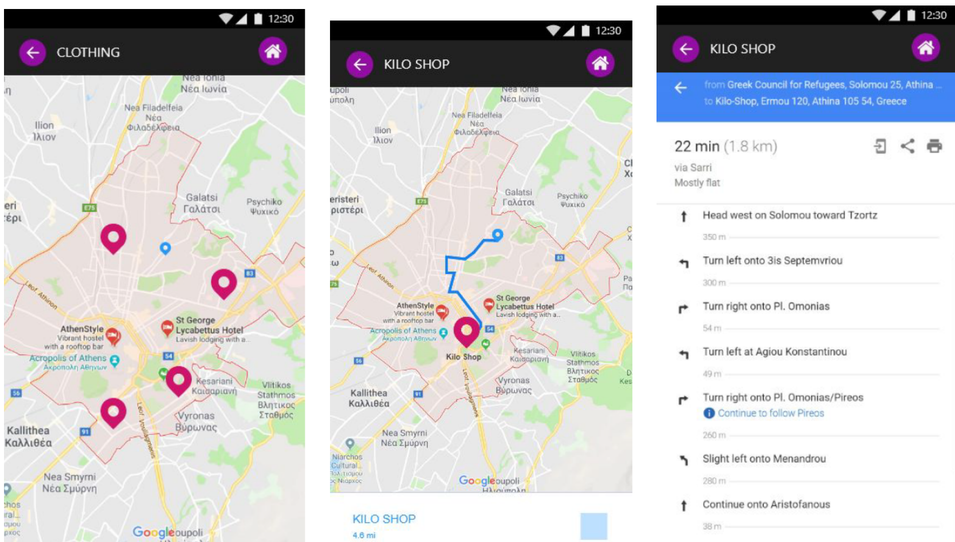


Figure 4.3: Example of list area, please notice the icons to indicate payment and language were not added yet.

At the end of the list of locations, there will be **large map button**. Clicking that button will open the Google Maps screen displaying in a single map all the locations of the category. The expected appearance of this screen can be seen in the figure 4.3.

- **Google Maps screen**: When clicking on the individual map buttons, a map of Athens with the route to reach the location will be displayed, as shown in the figure 4.4b. Clicking it again, it will immediately display step by step instructions about how to get to the location. The figure 4.4c exemplifies this behaviour. When clicking on the large button below the list of locations all the locations should be displayed on it, as shown in the figure 4.4a.



(a) Example of embedded map displaying several locations.

(b) Example of embedded map displaying the route to a given location.

(c) Example of indications screen inside maps.

Figure 4.4: Maps mock-ups.

During the focus group test in Athens, all the participants agreed that they would prefer to see Google Maps embedded inside the app rather than leaving the app to see the maps. So this screen should embed the map into the application. In all of the maps, the user should be able to zoom, move around and use the main Google Maps features.

- **Links page**: The links page works exactly like the previous areas but with no maps involved. When clicking on the web link of the location, an external web page will open.

In the final version, the system will also count with a very simple administration panel which will allow to update, remove and add locations with all their relative information.

4.2.4 Use cases

Now that we have a general idea of the proposed solution, we will specify all the requirements for our system to be considered complete and correct. We start defining our use cases to make sure we are taking into account all the possible uses when defining the requirements, defining this way the scope of our project as indicated in the Volere requirements template [2].

General use case diagram

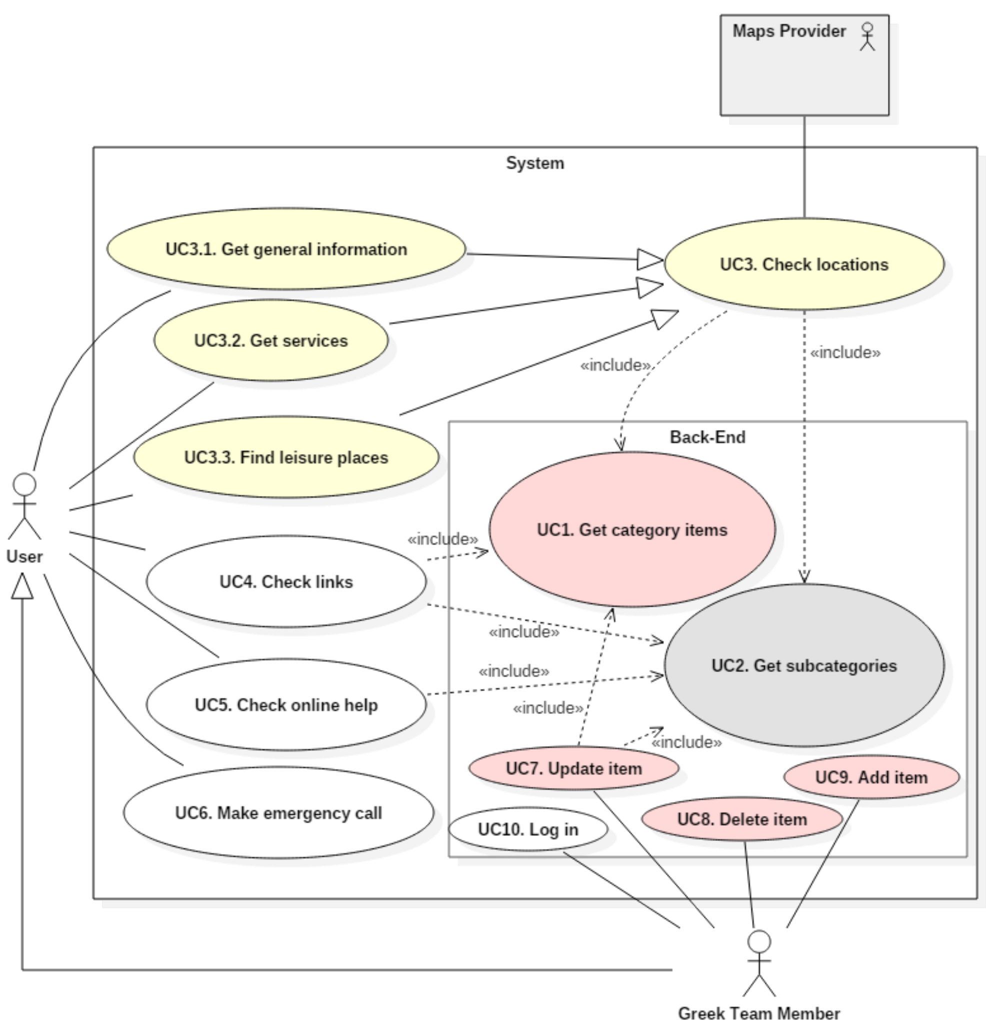


Figure 4.5: General use case diagram of the system.

The figure 4.5 shows the elaborated use case diagram of the system. Colours have been used to help to visually group related use cases. The `Maps Provider` actor

is specified with a slightly alternative notation, only to indicate that it is not a **human** actor. The use cases 3.1, 3.2 and 3.3 make reference to the described different location accesses, and they include getting the corresponding subcategories and, later, getting the different items[3] for one of those categories. The use case 4 is related to the links section, where we will not use maps. The use case 5 includes getting subcategories but not getting the category items, since they will be the own categories themselves the ones which will point to the different web locations. The use cases 1, 2, 7, 8, 9 and 10 have been grouped under *back-end* to try to give some more meaning to the diagram, indicating that these use cases are the ones that are strongly related with the back-end of our system.

Actors

We identified three main actors which will interact with the system:

- **User**: this actor includes all the habitual users of the application. They will be mainly refugees in Athens looking for some kind of concrete help. They have access to all the information the application but they cannot modify it.
- **Greek Team Member**: this actor corresponds to an arbitrary member of the Greek team, responsible for updating, deleting and adding the items. The members of this team are allowed to modify the items in the application to keep the data up to date.
- **Maps Provider**: the maps provider is an external actor we will have to interact with to provide our users with maps displaying the locations they are checking.

Complete specification

Now we define in a detailed, more complete way all the use cases of the system, with their main scenario and the possible alternatives of their execution. The use cases are also assigned a priority, which can be `low`, `medium` or `high`, and a list of preconditions which should be met before the use case main scenario.

[3]From now on, we will use the term *item* to refer to any of the basic elements of information that our application manages. For more detail on what is an item check the section 4.2.7.

Id: **UC1**	Name: **Get category items**	Priority: **High**
Actors:	None	
Preconditions:	None	
Main scenario:	1. The system receives a request for a list of items given the identification of a category 2. The system returns the items of the category 3. END.	
Alternative scenario 1: (1) **The category does not exist**	2. The system indicates through an error there is no such category 3. END.	
Alternative scenario 2: (1) **There are no items in the given category**	2. The system returns an empty answer 3. END.	

Id: **UC2**	Name: **Get subcategories**	Priority: **High**
Actors:	None	
Preconditions:	None	
Main scenario:	1. The system receives a request for a list of categories of a specified kind 2. The system returns the categories 3. END.	
Alternative scenario 1: (1) **There are no categories of the given kind**	2. The system returns an empty answer 3. END.	

Id: **UC3**	Name: **Check locations**	Priority: **High**
Actors:	User, Maps Provider	
Preconditions:	None	
Main scenario:	1. The `User` chooses to check some kind of locations. 2. The system displays the available categories for this kind of locations (UC2). 3. The `User` picks a category of the ones given by the system. 4. The system displays the items for that category (UC1). 5. The `User` can check all of them. He chooses to see one item on the map. 6. The system connects with the `Maps Provider` 7. The system displays the item location on the map, the user location and a route to get to the item location. 8. The user chooses to check the instructions of the route. 9. The system shows the instructions step by step to follow the given route. 10. END.	
Alternative scenario 1: (2) **There are no categories of the given kind**	3. The system shows that there are no categories. 4. END.	
Alternative scenario 2: (4) **There are no items in the picked category**	5. The system shows that there are no items in the category. 6. END.	
Alternative scenario 3: (4) **The picked category is a direct link**	5. Instead of displaying the locations, the system opens the link in a new tab. 6. END.	
Alternative scenario 4: (6) **The user chose to see all the items on the map**	7. The system displays all the items locations on the map, as well as the user location. 8. The user may pick one of the items. 9. BACK TO STEP 7.	
Alternative scenario 5: (1) **In any moment, the user decides to finish the interaction**	2. END.	

Id: **UC3.1**	Name: **Get general information**	Priority: **High**
Actors:	User, Maps Provider	Generalisation: UC3
Preconditions:	None	
Description:	The locations to display provide general information. By now, it behaves exactly like UC3.	

Id: **UC3.2**	Name: **Get services**	Priority: **High**
Actors:	User, Maps Provider	Generalisation: UC3
Preconditions:	None	
Description:	The locations to display provide services. By now, it behaves exactly like UC3.	

Id: **UC3.3**	Name: **Find leisure places**	Priority: **High**
Actors:	User, Maps Provider	Generalisation: UC3
Preconditions:	None	
Description:	The locations to display are places to spend leisure time. By now, it behaves exactly like UC3.	

Id: **UC4**	Name: **Check links**	Priority: **High**
Actors:	User	
Preconditions:	None	
Main scenario:	1. The `User` chooses to check some web links. 2. The system displays the available categories for links (UC2). 3. The `User` picks a category of the ones given by the system. 4. The system displays the items for that category (UC1). 5. The `User` can check all of them. He clicks on the link of one item. 6. The system opens the link in a new tab. 7. END.	
Alternative scenario 1: (1) **In any moment, the user decides to finish the interaction**	2. END.	

Id: **UC5**	Name: **Check online help**	Priority: **High**
Actors:	User	
Preconditions:	None	
Main scenario:	1. The `User` chooses to check some online help. 2. The system displays the available categories for help (UC2). 3. The `User` picks a category of the ones given by the system. 4. The system opens a web page with help in a new tab. 5. END.	
Alternative scenario 1: (1) **In any moment, the user decides to finish the interaction**	2. END.	

Id: **UC6**	Name: **Make emergency call**	Priority: **High**
Actors:	User	
Preconditions:	None	
Main scenario:	1. The `User` chooses the emergency call. 2. A call to the emergency services is started. 3. END	

Id: **UC7**	Name: **Update item**	Priority: **Medium**
Actors:	Greek Team Member	
Preconditions:	The `Greek Team Member` is logged in the system.	
Main scenario:	1. The `Greek Team Member` wants to update an item and picks the type of categories in which the category of the item he wants to change is. 2. The system shows the subcategories for the type he chose (UC2). 3. The `Greek Team Member` chooses one category of the displayed ones. 4. The system shows the items of the category (UC1). 5. The `Greek Team Member` chooses an item and edits any data in the item, included the opening hours and except the identification of the item, as they please. 6. When the item is correct, the `Greek Team Member` indicates this to the system. 7. The system stores the information. 8. END.	
Alternative scenario 1: (1) **In any moment, the user decides to finish the interaction**	2. END.	

Id: **UC8**	Name: **Delete item**	Priority: **Medium**
Actors:	Greek Team member	
Preconditions:	The `Greek Team Member` is logged in the system.	
Main scenario:	1. The `Greek Team Member` wants to delete an item and picks the type of categories in which the category of the item he wants to delete is. 2. The system shows the subcategories for the type he chose (UC2). 3. The `Greek Team Member` chooses one category of the displayed ones. 4. The system shows the items of the category (UC1). 5. The `Greek Team Member` chooses an item. 6. The system asks the `Greek Team Member` if he/she is sure. 7. The `Greek Team Member` agrees. 8. The system deletes the item from data store. 9. END.	
Alternative scenario 1: (7) **The Greek Team Member denies.**	8. BACK TO STEP 4.	
Alternative scenario 2: (1) **In any moment, the user decides to finish the interaction**	2. END.	

Id: **UC9**	Name: **Add item**	Priority: **Medium**
Actors:	Greek Team member	
Preconditions:	The `Greek Team Member` is logged in the system.	
Main scenario:	1. The `Greek Team Member` wants to add a new item. 2. The system shows a form where they can fill all the attributes of the item, included the opening hours or the category it belongs to. 3. The `Greek Team Member` fills the form and chooses to save. 4. The system stores the new item. 5. END.	
Alternative scenario 1: (1) **In any moment, the user decides to finish the interaction**	2. END.	

Id: **UC10**	Name: **Log in**	Priority: **Medium**
Actors:	Greek Team member	
Preconditions:	The `Greek Team Member` is not logged in the system.	
Main scenario:	1. The `Greek Team Member` wants to log in. 2. The system shows a form where he/she can fill his/her name and password to access. 3. The `Greek Team Member` fills the form and chooses to submit. 4. The system checks the password and name match. 5. The user is logged in. 6. END.	
Alternative scenario 1: (4) **Name and password do not match**	5. The system displays an error which contains no information about the real password or even if the user exists. 6. BACK TO STEP 2	
Alternative scenario 2: (1) **In any moment, the user decides to finish the interaction**	2. END.	

4.2.5 Functional requirements

We proceed now with the functional requirements. Both functional and non-functional requirements in this document follow an adapted version of the Volere *requirements shell* described in [2]. This shell includes the following parameters:

- Requirement: it indicates the requirement identification.
- Type: it indicates the type (section) of the requirement in the Volere template.
- Use cases: use cases that need the requirement.
- Description: a one sentence statement of the intention of the requirement.
- Rationale: a justification for the requirement.
- Fit criterion: a measurement of the requirement such that it is possible to test if the solution matches the original requirement.
- Customer satisfaction: degree of stakeholder happiness if the requirement is successfully implemented. It is scaled from 1 (uninterested) to 5 (extremely pleased).
- Priority: a rating of the customer value. In this document we will rate priority from 1 to 5, to keep the scale of the customer satisfaction and dissatisfaction described in the Volere template, meaning 1 the lowest priority and 5 the highest one.
- Customer dissatisfaction: degree of stakeholder unhappiness if the requirement is not part of the final product. It is scaled from 1 (hardly matters) to 5 (extremely displeased).

We divide them between back end and front end because we want to specify concrete functionality for each of them.

Back-end

Requirement: **FR-1**	Type: **9a. Functional requirements**	Use cases: **UC1**
Description:	**Users can obtain items for a given category**	
Rationale:	**We want to show and manipulate those items in the front-end.**	
Fit criterion:	**The system should output all the items for the required category with its opening hours, available languages, etc.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-2**	Type: **9a. Functional requirements**	Use cases: **UC2**
Description:	**Users can obtain categories of a given type of item**	
Rationale:	**We need the subcategories to choose one to check the items in it.**	
Fit criterion:	**The system should output all the stored categories for the given item type.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-3**	Type: **9a. Functional requirements**	Use cases: **UC10**
Description:	**Log into the system**	
Rationale:	**Only authorised people should be able to edit the data, we need to check them.**	
Fit criterion:	**Authorised users shall be able to make log in and receive access to the control panel.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **5**

Requirement: **FR-4**	Type: **9a. Functional requirements**	Use cases: **UC7**
Description:	**Authorised users can update items**	
Rationale:	**Authorised people needs to update the outdated data or correct the incorrect ones.**	
Fit criterion:	**The system shall allow the authorised, logged-in users to choose any item of the system and modify its data (except the data that identifies the item).**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **5**

Requirement: **FR-5**	Type: **9a. Functional requirements**	Use cases: **UC8**
Description:	**Authorised users can delete items**	
Rationale:	**Authorised people needs to delete the outdated or incorrect data sometimes.**	
Fit criterion:	**The system shall allow the authorised, logged-in users to choose any items of the system and delete them.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **5**

Requirement: **FR-6**	Type: **9a. Functional requirements**	Use cases: **UC9**
Description:	**Authorised users can add items**	
Rationale:	**Authorised people needs to add new data to the system.**	
Fit criterion:	**The system shall allow the authorised, logged-in users to add new items to the storage.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **5**

Front-end

Requirement: **FR-7**	Type: **9a. Functional requirements**	Use cases: **UC3.1**
Description:	**The application allows the users to find places to get general information about their situation.**	
Rationale:	**This is one of the main motivations of creating the application.**	
Fit criterion:	**The system should display these places in a clear, intercultural way.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-8**	Type: **9a. Functional requirements**	Use cases: **UC3.2**
Description:	**The application allows the users to find places to get different kinds of services.**	
Rationale:	**This is one of the main motivations of creating the application.**	
Fit criterion:	**The system should display these places in a clear, intercultural way.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-9**	Type: **9a. Functional requirements**	Use cases: **UC3.3**
Description:	**The application allows the users to find leisure places.**	
Rationale:	**This is one of the main motivations of creating the application.**	
Fit criterion:	**The system should display these places in a clear, intercultural way.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-10**	Type: **9a. Functional requirements**	Use cases: **UC4**
Description:	**The application allows the users to find useful web resources.**	
Rationale:	**This is one of the main motivations of creating the application.**	
Fit criterion:	**The system should display links to interesting online information of associations, NGOs, etc.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-11**	Type: **9a. Functional requirements**	Use cases: **UC5**
Description:	**The application allows the users to find help to their problems online in a quick way.**	
Rationale:	**There are general tasks that refugees need to do when arriving to the country which should be very quickly and easily accessible in the application.**	
Fit criterion:	**There should be direct links for tasks like opening a bank account, getting health service...**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-12**	Type: **9a. Functional requirements**	Use cases: **UC6**
Description:	**Users should have available some button to make an emergency call.**	
Rationale:	**Refugees can be trapped in dangerous situations or need quick attendance, many of them do not know which are the emergency numbers in Athens.**	
Fit criterion:	**There should be a way to make a call to the emergency services in the application. This should be also easy to find.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **FR-13**	Type: **9a. Functional requirements**	Use cases: **UC1, UC2, UC3, UC4, UC5**
Description:	**The system should be able to cache the data received from the API.**	
Rationale:	**We need the system to be able to work without internet connection.**	
Fit criterion:	**The system shall be able to cache the requests to the API or save the data in some way to use it when accessing without connection.**	
C. satisfaction: **4**	Priority: **4**	C. dissatisfaction: **3**

Requirement: **FR-14**	Type: **9a. Functional requirements**	Use cases: **UC1, UC2, UC3, UC4, UC5**
Description:	**The application should give the users the possibility to install it to the home screen.**	
Rationale:	**Some users might want the application installed in their devices.**	
Fit criterion:	**The application should give, whenever it is possible (depends on the system), the chance to install it as a native application to the home screen.**	
C. satisfaction: **3**	Priority: **3**	C. dissatisfaction: **2**

Requirement: **FR-15**	Type: **9a. Functional requirements**	Use cases: **UC7, UC9**
Description:	**The control panel shall check the items opening hours are not overlapped for the item the Greek team member is trying to update or insert**	
Rationale:	**This requirement was added cause the constraint was not implemented in the database, more information on this in the section 6.2.1**	
Fit criterion:	**It shall not be possible to create or update any item giving overlapped periods. The control panel itself should check this before sending it to the database.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **4**

4.2.6 Non-functional requirements

The non-functional requirements have been classified and defined following the Volere template described in [2]. They also follow an adapted version of the Volere *requirements shell*, just like the functional requirements written in the previous section. For a description of the shell parameters check the introductory paragraph of the section 4.2.5. These requirements are not separated between back-end and front-end since they affect both in most of the cases.

Look and Feel Requirements

Requirement: **NFR-1**	Type: **10a. Appearance**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The product shall be a modern-looking system**	
Rationale:	**It is important for the system to look modern since it will enhance the trust on the information it provides to the refugees.**	
Fit criterion:	**Anonymous enquiries should declare the system modern-looking by at least a 70% of the enquired users.**	
C. satisfaction: **2**	Priority: **2**	C. dissatisfaction: **2**

Requirement: **NFR-2**	Type: **10b. Style**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The product shall have a colourful palette of icons and a light, calm background.**	
Rationale:	**This will make the system have a relaxing, cheerful style which will be positive to improve the users feel and experience.**	
Fit criterion:	**The colour palette of the icons should have reds, greens, blues, yellows, violets... The background should be a relaxed white or blueish colour.**	
C. satisfaction: **2**	Priority: **2**	C. dissatisfaction: **2**

Usability and Humanity Requirements

Requirement: **NFR-3**	Type: **11a. Ease of use**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The product shall be used by people with no training, and possibly no understanding of English nor Greek.**	
Rationale:	**Refugees come from a wide variety of origins, with different cultures and languages, and have different levels of formation.**	
Fit criterion:	**The usage shall be tested in the tests after the alpha release, it also has been tested on the prototype with the focus group.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **NFR-4**	Type: **11a. Ease of use**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**All the information except the textual indications for a route should be at a maximum of three clicks from the home screen.**	
Rationale:	**The director of the Fundació ACSAR imposed this requirement, it relates to the simplicity in the use of the application.**	
Fit criterion:	**Starting from the home screen, we should check we can get to any item in the application in three or less clicks. The textual indications of how to get to a place are an exception to this rule and they will be at 4 clicks.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **3**

Requirement: **NFR-5**	Type: **11b. Pers. and Intern.**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**Internationalisation shall not be needed, users should be able to access it directly without choosing any specific language.**	
Rationale:	**As stated by the director of the Fundació ACSAR, the application *should have no language*, everyone should be able to understand the classification of the data independently of their origin language.**	
Fit criterion:	**There should be no option to choose language, or if there is one it should change only very small, almost-irrelevant texts.**	
C. satisfaction: **1**	Priority: **3**	C. dissatisfaction: **3**

Requirement: **NFR-6**	Type: **11c. Learning**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The product shall be able to be used by a public who will receive no training before using it**	
Rationale:	**The application, recommended and advertised in refugee centres, will not have any previous training for the users.**	
Fit criterion:	**Tests with the focus groups should certify all the users can understand the interface and the flow without the help of a previous training.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **5**

Requirement: **NFR-7**	Type: **11c. Learning**	Use cases: **UC7, UC8, UC9, UC10**
Description:	**The control panel might require a little previous explanation to the team which will update the information**	
Rationale:	**The information update might need a bit of understanding on how the application works which users do not need to know.**	
Fit criterion:	**The Greek team should confirm they understand the use of the control panel and what they are supposed to do with each option on it.**	
C. satisfaction: **3**	Priority: **3**	C. dissatisfaction: **4**

Requirement: **NFR-8**	Type: **11d. Underst. and Pol.**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The application should use symbols and icons that are naturally understandable by users from any community or culture.**	
Rationale:	**Refugees come from a wide variety of origins, with different cultures and languages, and have different levels of formation.**	
Fit criterion:	**Tests with the focus groups should certify all the users can understand the interface and the icons independently of their origin country.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **4**

Requirement: **NFR-9**	Type: **11d. Underst. and Pol.**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The product shall hide the details of its construction from the user**	
Rationale:	**This is a general requirement for any web application.**	
Fit criterion:	**The testing groups include mostly people who are inexpert in information technologies, so it should be enough to test this requirement.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **4**

Requirement: **NFR-10**	Type: **11e. Accessibility**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The application should be accessible to partially sighted or blind users, as well as deaf ones.**	
Rationale:	**Between the refugees we find many people with different kind of disabilities, blind and deaf users are ones of the most common.**	
Fit criterion:	**The application shall not have any kind of sound, or if it has one, it should be completely irrelevant for the use. The icons and all the data should use the specific HTML tags to make them accessible for blind people as well.**	
C. satisfaction: **4**	Priority: **3**	C. dissatisfaction: **3**

Performance Requirements

Requirement: **NFR-11**	Type: **12a. Speed & Latency**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**Every page content of the application should load in less than 350ms in any case.**	
Rationale:	**Long periods waiting for answer breaks the user's experience.**	
Fit criterion:	**When communicating with the API from Athens, every request to the API should receive the answer with a TTFB (time till first byte) of less than 350ms**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **4**

Requirement: **NFR-12**	Type: **12a. Speed & Latency**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**Every user interaction should receive an answer in 100ms as maximum.**	
Rationale:	**More than 100ms of reaction from the UI makes the application feel slow.**	
Fit criterion:	**Every button clicked by the user will trigger some kind of change in the UI in less than 100ms.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **4**

Requirement: **NFR-13**	Type: **12b. Safety-critical**	Use cases: **UC6**
Description:	**It should be possible to perform the emergency call in less than 2 seconds after clicking the emergency button.**	
Rationale:	**Time might be very limited when the user is in an emergency, so they need to be able to start the call quite quick.**	
Fit criterion:	**From the home screen, it should be tested we can be making the emergency call in less than 2 seconds.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **NFR-14**	Type: **12c. Precision / Accuracy**	Use cases: **All**
Description:	**All the times in the application should be given with a minutes precision.**	
Rationale:	**The schedules information is accurate enough managing days, hours and minutes. Seconds precision is unnecessary.**	
Fit criterion:	**The opening hours of the different items shall be presented with a precision of minutes.**	
C. satisfaction: **4**	Priority: **3**	C. dissatisfaction: **2**

Requirement: **NFR-15**	Type: **12d. Reliability and Availability**	Use cases: **All**
Description:	**The system should be available 24 hours per day, every day of the year.**	
Rationale:	**The system might be needed at any time by the refugees.**	
Fit criterion:	**The system should be available 24 hours per day, every day of the year.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **NFR-16**	Type: **12e. Robustness and Fault-Tolerance**	Use cases: **All**
Description:	**The application should continue working when internet connection is not available.**	
Rationale:	**Many refugees do not enjoy a continuous internet connection, they might depend on the public WiFi to have internet.**	
Fit criterion:	**After the first loading of a page, loading it without internet connection shall display the last displayed information. The maps page is an exception to this, it will be the only one requiring internet connection to load every time.**	
C. satisfaction: **5**	Priority: **5**	C. dissatisfaction: **5**

Requirement: **NFR-17**	Type: **12f. Capacity**	Use cases: **UC1, UC2**
Description:	**The system should be able to attend 10,000 users per day.**	
Rationale:	**This requirement comes from Javier Ideami, engineer director of the project in the Fundació ACSAR.**	
Fit criterion:	**The system should be able to attend 10,000 users per day.**	
C. satisfaction: **2**	Priority: **3**	C. dissatisfaction: **4**

Requirement: **NFR-18**	Type: **12f. Capacity**	Use cases: **UC1, UC2**
Description:	**The application should be able to serve 50 users simultaneously.**	
Rationale:	**Although we do not expect a high amount of users (over 10000 per day), these users might be distributed in certain peaks along the day. Sometimes there might be a lot of users connected at the same time.**	
Fit criterion:	**Stress tests should be performed before the beta release to check if the page is able to keep the answer time limit when serving 50 simultaneous petitions.**	
C. satisfaction: **2**	Priority: **4**	C. dissatisfaction: **5**

Requirement: **NFR-19**	Type: **12g. Scalability**	Use cases: **UC1, UC2**
Description:	**The system should be able to duplicate its maximum number of users per day each three months during the first year.**	
Rationale:	**This requirement comes imposed by Javier Ideami, engineer director of the project inside the Fundació ACSAR.**	
Fit criterion:	**Scalability should be taken into account while designing the back-end for the data access.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **5**

Operational and Environmental Requirements

Requirement: **NFR-20**	Type: **13b. Interf. with Adj. Syst.**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The system shall work on the last five releases of the five most popular browsers for mobile phones.**	
Rationale:	**During the focus group previous to the elaboration of this document, it was checked that refugees generally have quite-recent, updated mobile phones.**	
Fit criterion:	**The application shall run correctly on (the mobile versions of) Chrome, Safari, Opera, Samsung Internet and AOSP (the Android browser)[30]. UC has been excluded because it is hardly ever used outside China.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **4**

Requirement: **NFR-21**	Type: **13c. Productisation**	Use cases: **All**
Description:	**The application should be able to be installed to the home screen to work as a native application would do.**	
Rationale:	**This is a requirement of PWAs, the user has the option to *install* it and launch it from the home screen of their phones.**	
Fit criterion:	**The manifest of the web page shall be correctly configured to allow this functionality on the phones and browsers which accept the interface described in [4].**	
C. satisfaction: **3**	Priority: **3**	C. dissatisfaction: **1**

Requirement: **NFR-22**	Type: **13d. Release**	Use cases: **UC1, UC2**
Description:	**Each change in the API interface should keep the compatibility with previous versions.**	
Rationale:	**At the moment when these lines are written, there is an ongoing discussion about the Web App Manifest specification related to whether applications should or should not be able to force updating on the users with installed versions of the applications. Since at the moment it is not possible to force the update, and it might never be, we need to make sure new versions of the API will not break previous versions of the PWA.**	
Fit criterion:	**The fields of the interface offered by the API shall never be deleted, since it would cause the previous versions using those fields to fail. When existing fields change format, it will be needed to make sure that the format change will not cause the previous versions to fail (adding a new, alternative field when this is not possible). The PWA should be designed to ignore all the fields in the API answers it does not know, in order to keep working with future versions.**	
C. satisfaction: **2**	Priority: **4**	C. dissatisfaction: **5**

Maintainability and Support Requirements

Requirement: **NFR-23**	Type: **14a. Maintenance**	Use cases: **All**
Description:	**The application code should be highly maintainable, clean and easily understandable.**	
Rationale:	**The application might be edited by different developers than the original ones.**	
Fit criterion:	**The PHP code should adapt to PSR-2 standard, the whole code both for the PWA and the back-end should be approved by the supervisor engineer in the Fundació ACSAR, Javier Ideami.**	
C. satisfaction: **3**	Priority: **4**	C. dissatisfaction: **5**

Requirement: **NFR-24**	Type: **14c. Adaptability**	Use cases: **UC3, UC4, UC5, UC6**
Description:	**The application is expected to run under Android or iOs in mobile phones.**	
Rationale:	**Android and iOs are the two most common phone operating systems.**	
Fit criterion:	**The testing groups for the alpha and beta versions should include users of both of these operating systems.**	
C. satisfaction: **4**	Priority: **5**	C. dissatisfaction: **4**

Security Requirements

Requirement: **NFR-25**	Type: **15a. Access**	Use cases: **UC1, UC2, UC3**
Description:	**Only the Greek team and authorised members should be able to modify the locations of the application.**	
Rationale:	**The application information should be protected.**	
Fit criterion:	**The control panel should have an access control by password or a similar security feature.**	
C. satisfaction: **1**	Priority: **5**	C. dissatisfaction: **4**

Requirement: **NFR-26**	Type: **15b. Integrity**	Use cases: **UC1, UC2, UC3**
Description:	**The system should ensure items are associated to existing categories.**	
Rationale:	**Manual insertion of data usually leads to integrity problems.**	
Fit criterion:	**The system should ensure no item is associated to an nonexistent category.**	
C. satisfaction: **3**	Priority: **5**	C. dissatisfaction: **4**

Requirement: **NFR-27**	Type: **15b. Integrity**	Use cases: **UC1, UC2, UC3**
Description:	**The system should ensure hours and other kinds of data are held between their possible maximum and minimum values.**	
Rationale:	**Manual insertion of data usually leads to integrity problems.**	
Fit criterion:	**When introduced, the system shall check the days for the opening hours of the items are correct, that hours are set between 0 and 23, minutes between 0 and 59 and check other logical ranges/options for all the data whenever it is possible.**	
C. satisfaction: **3**	Priority: **5**	C. dissatisfaction: **4**

Requirement: **NFR-28**	Type: **15c. Privacy**	Use cases: **All**
Description:	**The application should be served over HTTPS.**	
Rationale:	**Secure HTTP is the base for the protection of the user's privacy as well as the integrity of the transferred data.**	
Fit criterion:	**The server should serve the page over HTTPS.**	
C. satisfaction: **1**	Priority: **5**	C. dissatisfaction: **3**

Requirement: **NFR-29**	Type: **15c. Privacy**	Use cases: **All**
Description:	**If saving cookies to the browser of the users were needed, a dialog should inform the users about it.**	
Rationale:	**Cookies can be used to track users along the way, being this an attack to their privacy that should be accepted by them to be legitim.**	
Fit criterion:	**If saving cookies to the browser of the users were needed, a dialog should inform the users about it.**	
C. satisfaction: **2**	Priority: **4**	C. dissatisfaction: **1**

Requirement: **NFR-30**	Type: **15e. Immunity**	Use cases: **UC7, UC8, UC9, UC10**
Description:	**The site should be protected from cross-site request forgery (CSRF).**	
Rationale:	**CSRF is a well-known, common kind of attack for web pages which consists in fooling the users to perform actions on your site from another site created by the attacker.**	
Fit criterion:	**The system should implement a system of random tokens generation to try to make sure any request of data modification comes from the own site, the control panel, and not from an external site.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **1**

Requirement: **NFR-31**	Type: **15e. Immunity**	Use cases: **UC7, UC8, UC9, UC10**
Description:	**The site should be protected from SQL injections.**	
Rationale:	**SQL injection is still the most common attack to web pages which accept user input.**	
Fit criterion:	**The develop should be extremely careful when processing data in SQL statements. Pen-testing tools shall be used to check there are no possible weaknesses in this sense.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **1**

Requirement: **NFR-32**	Type: **15e. Immunity**	Use cases: **UC7, UC8, UC9, UC10**
Description:	**The site should be protected from cross-site scripting XSS.**	
Rationale:	**XSS consists in introducing malicious code through the user inputs to change the behaviour of the application with the users.**	
Fit criterion:	**The develop should be extremely careful when processing user input. Pen-testing tools shall be used to check there are no possible weaknesses in this sense.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **1**

Requirement: **NFR-33**	Type: **15e. Immunity**	Use cases: **UC7, UC8, UC9, UC10**
Description:	**The system should be protected against other common kinds of attacks.**	
Rationale:	**More than the literally specified in this requirements, there are lots of well-known, common attacks performed to web pages (as DDoS) which may affect our application.**	
Fit criterion:	**A well-known, trustful pen-testing tool should be ran against the final version of the system and prove it to be non-vulnerable to the attacks tested by the tool.**	
C. satisfaction: **2**	Priority: **5**	C. dissatisfaction: **1**

Cultural and Political Requirements

Requirement: **NFR-34**	Type: **16a. Cultural**	Use cases: **All**
Description:	**The data provided by the system should be respectful to any religious or ethnic groups.**	
Rationale:	**Refugees profess many different religions and are part of different ethnic groups.**	
Fit criterion:	**The data provided by the Greek team will be approved by the supervisor of the project together with the director of the Fundació ACSAR.**	
C. satisfaction: **1**	Priority: **5**	C. dissatisfaction: **5**

Legal Requirements

Requirement: **NFR-35**	Type: **17a. Compliance**	Use cases: **All**
Description:	**The product must comply with the attribution requirements of all the libraries and frameworks that are used.**	
Rationale:	**Libraries, frameworks and related products usually require attribution, license references or similar kinds of information to be displayed on the derived products.**	
Fit criterion:	**The product shall provide a place to display the previous information in its final release.**	
C. satisfaction: **1**	Priority: **5**	C. dissatisfaction: **5**

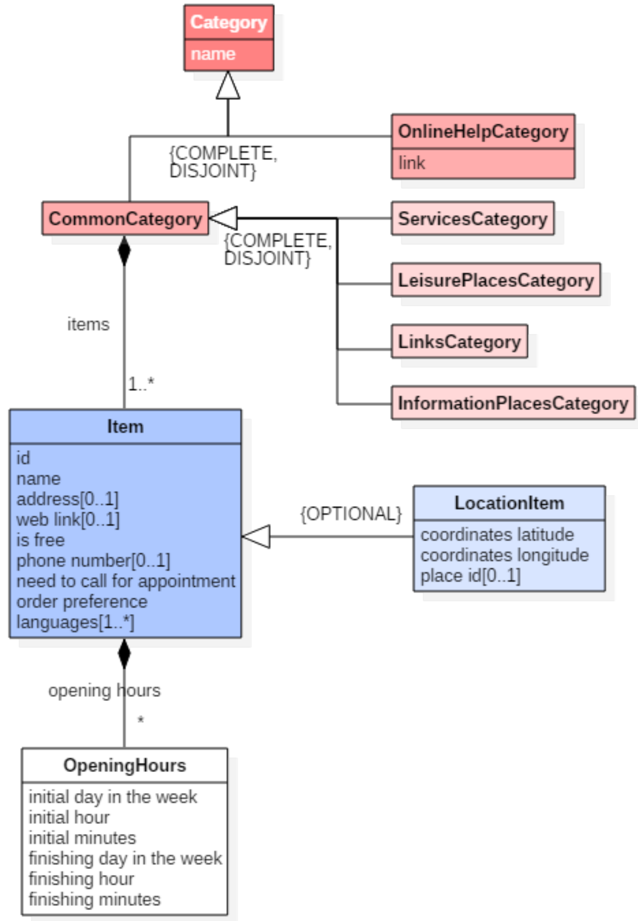
4.2.7 Conceptual data scheme

Finally, the general conceptual data scheme (data model) is presented, since it will be the base for the design process. This scheme together with the description explained in this section work also as a dictionary of the terms used in the project, what they mean and which attributes and relations they have. A UML class diagram will be used, accompanied by textual explanations of the concepts and their relations. The figure 4.6 shows the class diagram in UML displaying the concepts to be managed by our system.

In first place we have the `Categories`, there are two main kind of categories based on the use cases of our application: the online help categories, which link directly to some external web page, and all the other categories, which are composed of `Items`, that may also link to web pages, among other information. These other categories are the services, the leisure places, the links and the information places. A category should be of one, and only one of all this different types. All the categories can be identified by a name.

Then we have the `Items`, which have a name to describe what organism/association/place in the real world the item is related to, an optional address of the physical place where people can find that related organism/association/place, an optional link to the web page, an attribute indicating whether it is free or not, an optional phone number to call, an attribute which indicates if it is needed to make a call requesting an appointment, an order preference to display in the application and one or more languages in which people is attended there. Optionally, items can (and most of them will) be locations, having an optional *place id* used to associate it with the maps engine (like Google Maps Places) and latitude and longitude to locate them on the map.

Finally, `Items` have a series of `OpeningHours` associated to them. These opening hours are periods of time when the `Item` is available (open) to people. As many periods as necessary can be added to define precisely the desired schedule.



1. Primary keys: (Category, name); (Item, id); (OpeningHours, Item:id, initial day in the week, initial hour, initial minutes, finishing day in the week, finishing hour, finishing minutes)
2. If an item needs a call to get an appointment, it should have a valid phone number.
3. If an item does not have address, then it needs a call to get an appointment.
4. Two OpeningHours associated to the same item should not overlap.
5. OpeningHours:'initial hour' and 'finishing hour' should be between 0 and 23.
6. OpeningHours:'initial minutes' and 'finishing minutes' should be between 0 and 59.
7. OpeningHours:'initial day of the week' and 'finishing day of the week' should be a valid day of the week.

Figure 4.6: Conceptual data scheme of the system in UML, colours have been used to facilitate reading.

These three kinds of objects are joined by composition relationships, since none of the lower-level ones can exists without a higher-level one associated. There are a couple of restrictions which cannot be expressed through UML syntax, which are the following:

1. The primary keys of the items. The categories can be identified by name. The items, since there is no reason to think it is not possible to have two items with the same name in the same category, are identified by an internally generated id. The opening hours are

identified by all of their elements together with the item they correspond to.

2. If an item needs a call to get an appointment, it necessarily should have a valid phone number to call to.

3. If an item does not have any address to reach them, then it needs a call to arrange an appointment and this way decide a place of meeting.

4. Two periods associated with the same item should not overlap, since doing this would mean some kind of mistake while defining the schedule.

5. The initial and finishing hours of the periods should be in a range from 0 to 23, since other values are not valid hours.

6. The initial and finishing minutes of the periods should be in a range from 0 to 59, since greater values would correspond to one hour more.

7. The initial and last day of the week of each period should be valid days of the week (from Monday to Sunday).

Chapter 5

Design

In this chapter we will explain how the system is created, which components it is comprised of and how they communicate. The main design questions and decisions will be explained in detail. We assume the reader has all the necessary knowledge about the employed pattern designs, so it is possible that their concrete interaction (e.g. the singleton instantiating) were not detailed in this document. However, most of them can be found in [11] or [8].

The design is made taking into account all the non-functional requirements and prioritising the interchangeability and maintenance of the code. In this sense, we will try in every moment to generate very reusable classes with as low coupling between them as possible. In order to prevent this document from getting too long, we will not provide a complete specification for the interaction between the objects of our system through sequence diagrams. We will limit ourselves to provide diagrams for the most interesting and representative interactions and verbal descriptions of the other functionality. Some more sequence diagrams can be found in the GitHub of the project[1] in the *diagrams.mdj* file.

5.1 General structure of the system

As explained in the section 2, our system is divided into three independent components, as it is usual in modern PWA development. Each one can be served from an independent machine, and for scalability reasons some of them may be even split in the future into several machines also. First, we have the relational database, it will provide the system with all the necessary information, taking control of some of the main integrity constraints for the data. Second, to have a layer between the database and our PWA, making our front-end independent from the underlying database system, we have an API with a JSON interface and a Rest-like style. Finally, our information is displayed to the users through a PWA application running on *client side* (on the

[1] `https://github.com/david-campos/app4refs`, it is not public at the moment of writing these lines but it will be, eventually, in the coming months.

browser of the client). The figure 5.1 shows this general structure of the system.

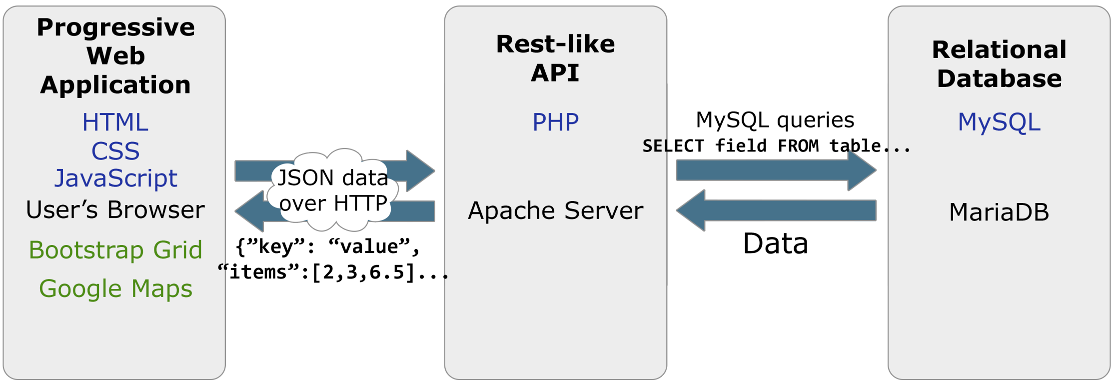


Figure 5.1: Diagram showing the three main parts of the project and how they communicate to each other.

There is a fourth component, the control panel, whose design and implementation is not presented in this document. As described in the 3.8 section, this panel will only be present since the beta release and it will be designed and implemented after finishing this document (as this covers, because of the deadline to deposit the final project, only the construction of the alpha version).

The use of the API has several advantages: on the one hand, it is a standard solution which will provide an stable interface for our front-end, making it independent from the underlying system. This way, if we ever need to change the DBMS we can do it without having to modify anything in the code of our PWA. On the other hand, having an API makes it easier for developers to understand the communication, which is performed in a human-readable format, and allows us to create several *client* applications communicating in the same way with the same data source (for example, we could decide in the future to make an Android application; this application would be able to communicate with the same API in pretty much the same way our PWA does and display the information as it were more convenient). In general, this API gives us more control over the data giving us a central point to access it without having to deal directly with the underlying storing technology. We make use of a Rest-like API, as explained in 5.3, because Rest approach offers a very intuitive interface which makes extensive use of the HTTP technology alone.

Advantages of using PWA have been explained in the section 4.2.3. Based on how recent the smartphones were, in general, in the focus groups previous to the development, PWA was decided as the best option because of its cross-platform capabilities. With a single code, we are programming at the same time a web page and an Android, iOs and desktop application. Modern web technologies allow PWA to perform all our requested functionality.

One of the principles while doing the whole design of the system was always to rely as few as possible in frameworks and libraries. This was suggested by the engineer in the Fundació

ACSAR, Javier Ideami, given the small scale of the project. It comes to special relevance in the part of the front-end, because PWAs and modern JavaScript in general have a big tendency to abuse the use of libraries and frameworks. The only framework employed in run-time in the PWA is Bootstrap, and not even the whole library but only its grid module.

5.2 Back-end: The database

To design the database we start from the conceptual data scheme described in the section 4.2.7 and we make, based on it, an entity-relationship model from which we obtain the final relational model.

5.2.1 Entity-relationship model

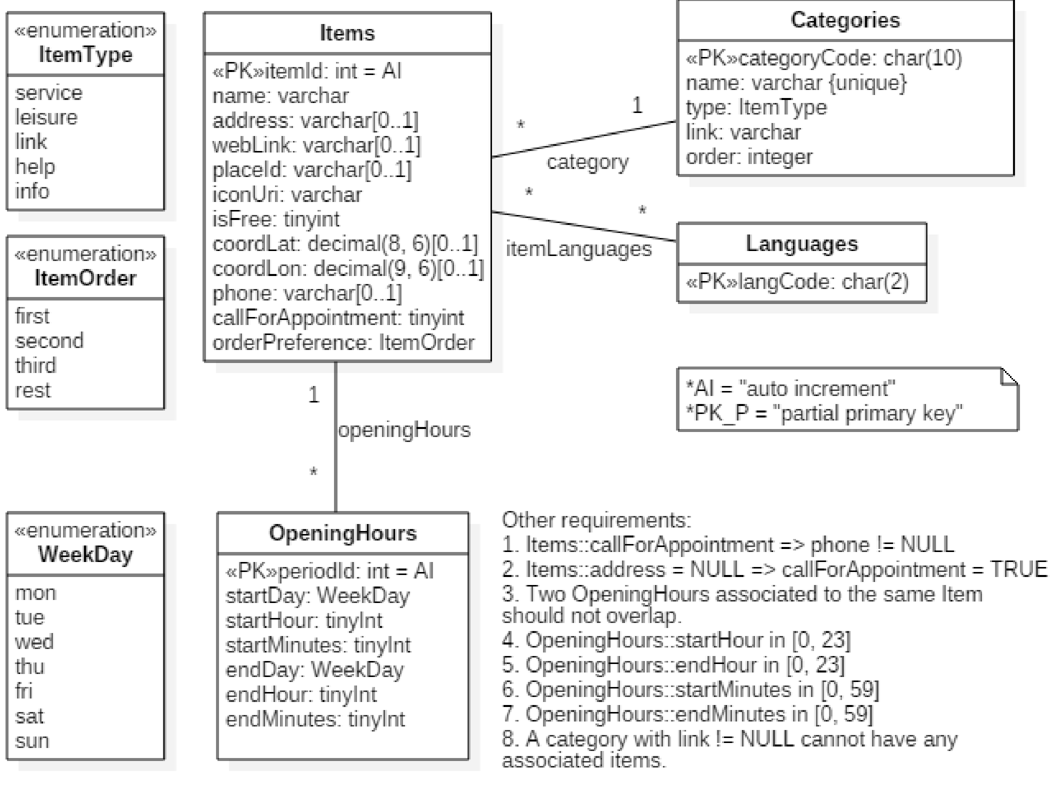


Figure 5.2: Diagram of the entity-relationship model.

The complete hierarchy of `Category` is collapsed into a single entity, with a *type* attribute which indicates the type of category it is, we add also an *order* attribute which will serve interface

purposes. We make the same with `Item`, since most of them will be locations, and move languages to a new entity to give it more relevance. This will help to prevent mistakes in the data. Allowing languages to take any value would allow for typos to be accepted, making it some kind of *enumeration* field would be a bad idea because we are not really sure about how many languages we will have (it is not a static set of values). Opening hours remains as it is, but given the high number of fields of its PK, we decide to add an auto generated *periodId* field for efficiency reasons and to make the code simpler and readable when working with them. The diagram 5.2 shows an UML description of the entity-relation model of the database. The names of the entities are turned into plural, the attributes converted to camel case and some more database-specific details added (like the types of each element or the primary keys).

Entity	**Description**	**Number of instances**
Categories	All the categories of items for the application. They correspond to a simple classification of the items directly connected to the navigation through the page (Check 4.2.3).	We preview around 50.
Items	All the items of the application, they represent real-life places or entities like shops, NGOs...	As minimum five per category, as maximum we can estimate 20 per category. This is, between 250 and 1,000.
Languages	All the relevant, available languages for the items	We plan to have only English and Greek. But they might be more in the future. Probably never more than 10.
OpeningHours	Periods of time in which the items are opened	Probably five or ten per item (one or two for each working day of the week). This means between 1,250 and 10,000 records.

Table 5.1: Entities data dictionary.

As we can read in the figure, there are a couple of requirements which the diagram itself cannot express:

1. If an item has `callForAppointment` set to 1 (*TRUE*), then `phone` cannot be a *NULL* value.

2. If an item has `address` set to a *NULL* value, then `callForAppointment` should be set to 1 (*TRUE*).

3. Two records for `OpeningHours` should never overlap in time.

4. The `startHour` of the opening hours should be in the range [0, 23].

5. The `endHour` of the opening hours should be in the range [0, 23].

6. The `startMinutes` of the opening hours should be in the range [0, 59].

7. The `endMinutes` of the opening hours should be in the range [0, 59].

8. A category with a link that is not NULL cannot have any associated items.

The model is completely described by the data dictionaries shown in the tables 5.1, 5.2 and 5.3.

Entity	Mult.	Relationship	Mult.	Entity
Items	*	category	1	Categories
	*	itemLanguages	*	Languages
	1	openingHours	*	OpeningHours

Table 5.2: Relationships data dictionary.

Entity / Relationship	Attribute	Description	Type	N	M	D	Def
Items	itemId {PK}	Auto generated id of the item	Positive integer	-	-	-	A.I.
	name	The name of the represented entity	Variable character, 100	-	-	-	-
	address	The physical address of the represented entity	Variable character, 255	✓	-	-	-
	webLink	Link to the web page	Variable character, 255	✓	-	-	-
	placeId	Id to identify the place in the maps system	Variable character, 255	✓	-	-	-
	iconUri	The URI of the icon to display for the item	Variable character, 255	-	-	-	-

	isFree	Whether the provided service, the provided information or the access to the place are free or not	Boolean	-	-	-	-
	coordLat	The global latitude to place the item in the map	Decimal number (8, 6)	✓	-	-	-
	coordLon	The global longitude to place the item in the map	Decimal number (9, 6)	✓	-	-	-
	phone	A phone number to get in contact with the entity represented by the item	Variable character, 100	✓	-	-	-
	callForAppointment	Indicates whether a call to agree an appointment is needed	Boolean	-	-	-	-
	orderPreference	The preference of order for the item in the list of items of its category	`first`, `second`, `third` or `rest`	-	-	-	`rest`
	categoryCode {PK}	The code to identify the category	Character 10	-	-	-	-
Categories	name	A name for the category	Variable character, 100	-	-	-	-
	type	The type of items in the category	`service`, `leisure`, `help`, `link` or `info`	-	-	-	-
	link	A link to directly open in a new tab instead of showing the list of items.	Variable character, 100	✓	-	-	-
	order	A number to set the order the categories follow in the interface	Positive integer	-	-	-	-
Languages	langCode {PK}	Language code	Character, 2	-	-	-	-

OpeningHours	periodId {PK}	Internal id to identify the period	Positive integer	-	-	-	A.I.
	startDay	The day of the week in which the period starts	`mon`, `tue`, `wed`, `thu`, `fri`, `sat` or `sun`	-	-	-	-
	endDay	The day of the week in which the period ends	`mon`, `tue`, `wed`, `thu`, `fri`, `sat` or `sun`	-	-	-	-
	startHour	The hour of `startDay` in which the period starts	Positive integer	-	-	-	-
	startMinutes	The minutes inside the `startHour` in which the period starts	Positive integer	-	-	-	-
	endHour	The hour of `endDay` in which the period ends	Positive integer	-	-	-	-
	endMinutes	The minutes inside the `endHour` in which the period ends	Positive integer	-	-	-	-
category							
itemLanguages							
openingHours							

Table 5.3: Attributes data dictionary.

5.2.2 Relational model

Converting now this diagram into the corresponding relational graph is quite easy. We converted it as follows. Notice the underlined attributes specify the primary key of the relations, REFER indicates a foreign key constraint (also called *reference*) with another relation, CHECK indicates a simple check the database should perform on the rows (directly implementable in many relational databases) and *index* indicates the creation of an index over the column.

categories (category_code, name, link, item_type, position) INDEX position

languages (lang_code)

items (<u>item_id</u>, name, address, web_link, place_id, icon_uri, is_free, coord_lat, coord_lon, phone, call_for_appointment, order_preference, *category_code*)

```
REFER category_code => categories::category_code
    UPDATE cascade, DELETE restrict
INDEX order_preference
CHECK (¬(call_for_appointment ∧ phone = null))
CHECK (¬(address = null ∧ ¬call_for_appointment))
```

item_languages (*<u>item_id</u>*, *<u>lang_code</u>*)

```
REFER item_id => items::item_id
    UPDATE cascade, DELETE cascade
REFER lang_code => languages::lang_code
    UPDATE cascade, DELETE restrict
```

opening_hours (<u>period_id</u>, start_day, end_day, start_hour, start_minutes, end_hour, end_minutes, *item_id*)

```
REFER item_id => items::item_id
    UPDATE cascade, DELETE cascade
CHECK (0 ≤ start_hour < 24)
CHECK (0 ≤ end_hour < 24)
CHECK (0 ≤ start_minutes < 60)
CHECK (0 ≤ end_minutes < 60)
```

Still, some triggers should be implemented to check the values for the opening hours assigned to the same item do not overlap between them and to check a category with a link value does not have any associated items. We have decided to insert indexes in `items` and `categories` to speed up the ordering of these elements.

5.3 Back-end: The API

Now that we have the storage of the data designed, we need to design the REST-like API that will give access to the data in a human-readable format making use of the HTTP protocol. We say this is a REST-like implementation because it does not have some of the fundamental parts of the REST applications (like the hypermedia), but it really does fit to another REST principles like the concept of resources, the use of HTTP methods to perform operations over these resources, the stateless communication or the uniform interface with self-descriptive messages.

5.3.1 Interface design

To design our API the first we needed was to identify all the resources of our system and how they would be accessed. A resource in a REST API is anything that can be named, these resources

can be accessed through resource representations and over them we can perform certain API operations. Each resource has an specific resource identifier (URI) which allows us to access the resource. Based on the conceptual data scheme described in the section 4.2.7 and the storage we designed on the section 5.2.1, we decided we only need to have two resources in our API:

- **Items**: The items will contain the `itemId`, `name`, `address`, `webLink`, `placeId`, `iconUri`, `isFree`, `coordLat`, `coordLon`, `phone`, `callForAppointment` and `orderPreference` attributes described in the entity-relation model in 5.2.1. It also will include, invisible to the user that this is another entity, all the opening hours of the item with their `startDay`, `startHour`, `startMinutes`, `endDay`, `endHour` and `endMinutes` attributes, as well as the codes of the languages the item is related to. The reason to group these entities under the same resource comes from the use cases and their associated functional requirements, specially **UC1**, where we can check the items and this "nested" entities will always be obtained together as a single one. To accomplish the functional requirements FR-1, FR-4, FR-5 and FR-6 we need to dispose of the four CRUD operations over this resource. We will need a resource representation to access the items for a given category.
- **Categories**: The categories will contain the `categoryCode`, `name`, `link`, `type` and `order` attributes. The categories only need to be obtained but not modified, since we do not have required functionality of editing categories. They need a representation based on their type cause that is how they will be selected (**UC2**).

Resource	URI	Method	Description
Item	/categories/**:categoryCode**/items/	GET	Gets the items for the category with code *categoryCode*.
	/items/**:id**	PUT	Updates the item with id *id*.
	/items/**:id**	DELETE	Deletes the item with id *id*.
	/items/	POST	Creates a new item.
Category	/item-types/**:type**/categories/	GET	Gets the categories for the given *type*.

Table 5.4: Resources of the API.

While updating and deleting the items, notice the `itemId` **should not be present** in the

body of the item. The periods associated to the item shall contain the `periodId`, and it should be set to a reserved keyword (e.g. *new*) for all the newly created periods. While updating the item, all the periods with specified id shall be preserved and modified, all the ones with id *new* should be created and all the absent ones should be deleted.

There is a requirement missing, which is the FR-3. Due to a debate about the kind of access we are going to give to the Greek Team members to the control panel, and given that the control panel development corresponds to the beta version of the system, we decided not to design the API log-in yet, and left it to be discussed in later meetings once we had the alpha to test in Athens. For this reason, this first version does not have log-in and, although the functionality to modify items will be designed and implemented already, it will not be available in this first release. The table 5.4 shows the selection of resources, their representations and the http methods associated to each operation.

There are two main parts of the API where we would like future changes to be easy: the input and output formats and the database connection. For this first one we will add two optional GET parameters[2] that the API will recognise: `in` and `out`. This values will allow to specify the input and output formats, respectively, being JSON the default one. The second one is explained in 5.3.5, but first we need to enter into more detail about the decided architecture for the API.

5.3.2 The architecture

The API will be designed in a three-tier architecture. There will be a first layer which we will refer as the view, where we will have the interface-specific details, another layer called the domain, where the items of our application reside, and a final layer denominated the data layer, where we will define the persistent-data accesses and how they are managed. Dividing the system into this three tiers allows us to have a better visualisation of the whole system... The view layer is the one in charge of receiving the input and printing the output, it will be compound by the input parser and the output printer (which we called, a bit as a joke, *outputter*), together with the factory responsible to generate both of them. The domain layer will contain the transaction-managing and other stuff relative to the domain manipulation, for example the URL matching. Finally, the data layer will contain all the necessary classes to communicate with our database. The data flow during a request to the API is visually

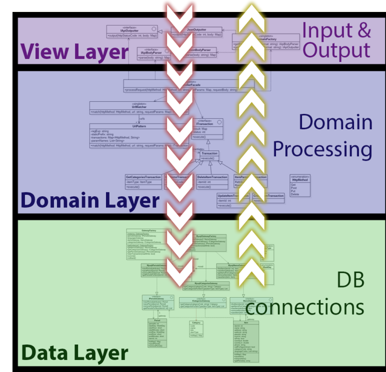


Figure 5.3: Data flow over the three tiers of the API.

[2]The GET parameters are a list of key-value pairs which we can add to the URL we are accessing through HTTP. They are indicated by a *?* symbol and a succession of the pairs in the format `key=value` separated by `&` each one from the other.

described by the figure 5.3.

Each request we receive will enter to the same file, which we will call the *index* in this section. For this reason, server redirection has been needed for the implementation, more information on this can be found in the section 6.2.3. It is interesting to note, also, that each request will run in a completely independent process or thread (depends on the server), so they will not be able to interfere with each other. The execution of each of these request will be performed in a completely sequential way. Whether including multithreading would improve or not the performance has not been tested, but literature online suggests it would not. Our request processing is quite short and straightforward so even though some minor tasks could be paralleled, probably tiny performance improvements would be obtained. Also, the general parallel processing the server is performing in a superior layer than our code would make these improvements completely negligible.

5.3.3 Domain layer and view Layer

Since the view layer is very simple in this case, we present the domain and the view layers together in a unique diagram to ease reading. The following explanation can be read following the UML class diagram in the figure 5.4 for a better understanding.

The view layer is compound by the *body parsers* and the *outputters* (the name we gave to the objects which *output* the data generating the HTTP answers). The body parsers receive the body of a request[3] as string and parse into an associative array (also called map), which can have several nested levels. The outputter objects perform pretty the opposite operation, taking an associative array and parsing it into a string, but this time they do not return anything. Instead, they directly output the string to the body of the HTTP answer and set the right headers for it. They allow to choose the HTTP status in the response, which will be used for error managing, among other things (more detail on this in 5.3.4). To facilitate the creation of different parser and outputter objects for the different formats of input and output data we used the factory pattern, creating a `FormatFactory` class which provides two methods to instantiate the tight body parser or outputter, returning two interfaces `IApiBodyParser` and `IApiOutputter`, which allows us to isolate from the exterior the classes for the different formats. We will implement, by now, only the JSON versions of both the body parser and outputter.

To manage the different operations which the API can perform we use a *transaction pattern*, encapsulating each transaction in an object of the class of the transaction. This is a common organisation for the domain layer. The five transactions we need are getting the categories of a given type (`GetCategoriesTransaction`), getting the items for a given category (`GetItemsTransaction`), deleting the items (`DeleteItemTransaction`) and, finally, creating and updating items (`CreateItemTransaction` and `UpdateItemTransaction`). These last two have in common that they need to extract from the parsed body of the request the right values for the created/updated item. To simplify the implementation of this functionality

[3]Some HTTP requests, like POST or PUT, include a body which can contain any type of data.

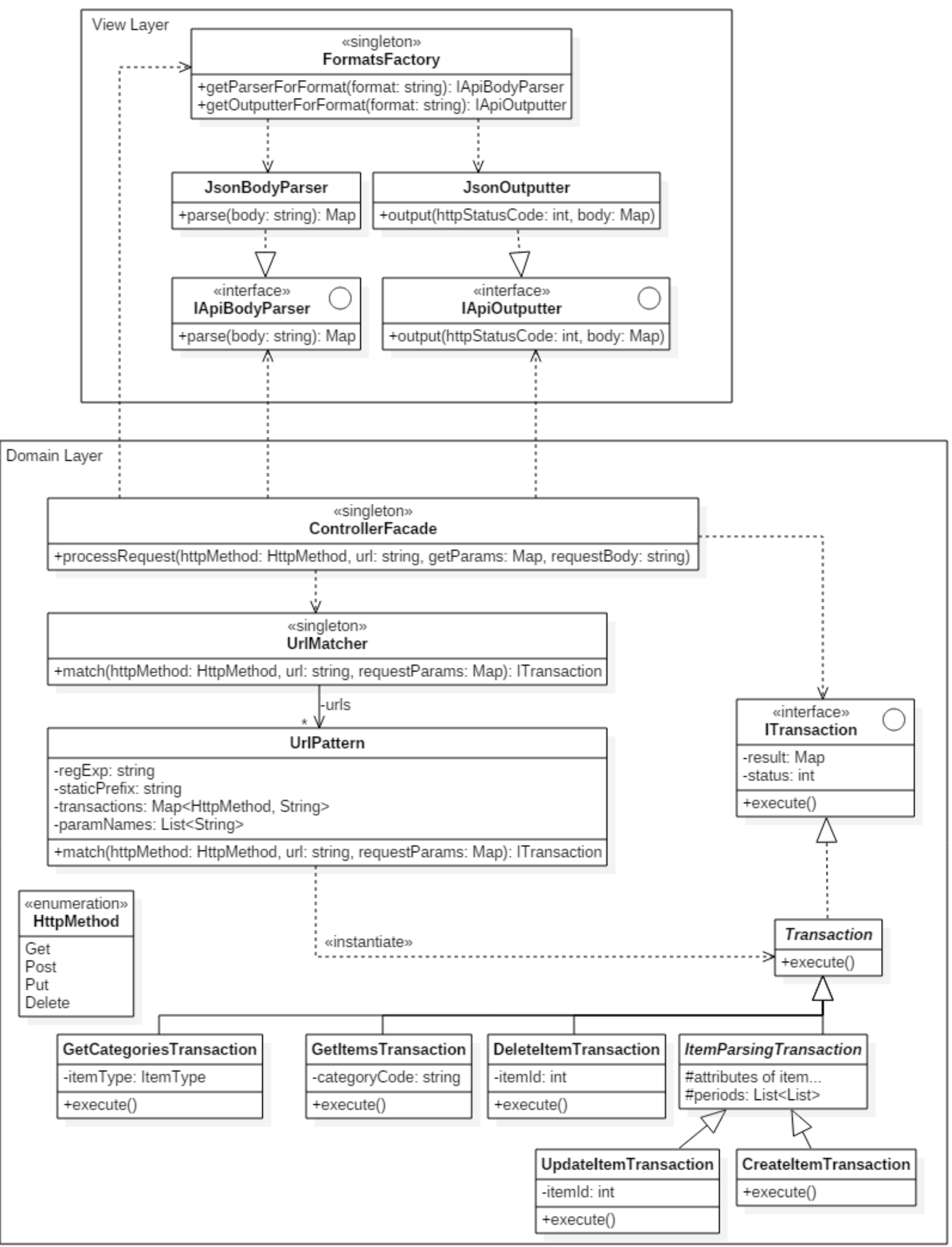


Figure 5.4: Class diagram of the domain and the view layers. Setters, getters and some other less relevant functions have been omitted to simplify reading.

we generalise them to an abstract `ItemParsingTransaction`. All of them will implement the interface `ITransaction`, which has a method to execute the transaction and two attributes employed to save the results. All of them will be children of a general, abstract `Transaction` class which implements said interface. More detail on the behaviour of the transactions can be found in 5.3.6.

To instantiate the right transaction for the received request the system depends on the `UrlMatcher`. This is a singleton class referencing a set of `UrlPattern` instances, with a method to iterate over them searching for a matching pattern to create the transaction. This behaviour is shown in the interaction diagram in the figure 5.5. The `UrlPattern` class is the responsible to match in an URL each HTTP method to a given class implementing `ITransaction`. To be able to perform this, it receives on the constructor an URL pattern[4] similar to the ones given in the table 5.4 and generates from it a regular expression to check against the URL reception of each request. Since computing regular expressions is a bit more expensive than string comparison, to improve the efficiency we also save a *static prefix*, this comes in imitation of the matcher classes some very known PHP frameworks use. This static prefix is the prefix with all the characters of the URL which do not change (all the characters before the first parameter). The parameter names attribute will be used to generate an associative array to pass to the constructor of the transaction including the parameters extracted from the request URL. Notice we can instantiate the right transaction object from the class name thanks to the metaprogramming paradigm supported by PHP.

When the API receives a new request, the *index* will automatically call the `processRequest` method in a singleton class we called `ControllerFacade`. This class has the aim to process the requests received, extracting their parameters and coordinating some of the other classes instances to create the right transaction, executing it and producing the right HTTP answer. This is achieved making use of the `FormatFactory` to obtain the right parser and *outputter*, and the `UrlMatcher` to obtain the adequate transaction. The UML interaction diagram in the figure 5.6 illustrates this behaviour.

5.3.4 Managing errors

During the execution of the request processing many things can fail. Some required parameter might be missed, some value can be incorrect, the data might not be valid... To manage all this situation we take advantage of the PHP exception handling. We define a hierarchy at the top of which we have a class called `PrintableException`. This abstract class possesses a message and an status and represents an exception which can be safely printed to the user (because we are sure it does not contain sensitive information). All the possible errors which may appear processing the request which are not debt to internal malfunctioning should throw an exception extending this class. It is the responsibility of `ControllerFacade` to catch all

[4]More information on the `UrlPattern` patterns in the section A.4.

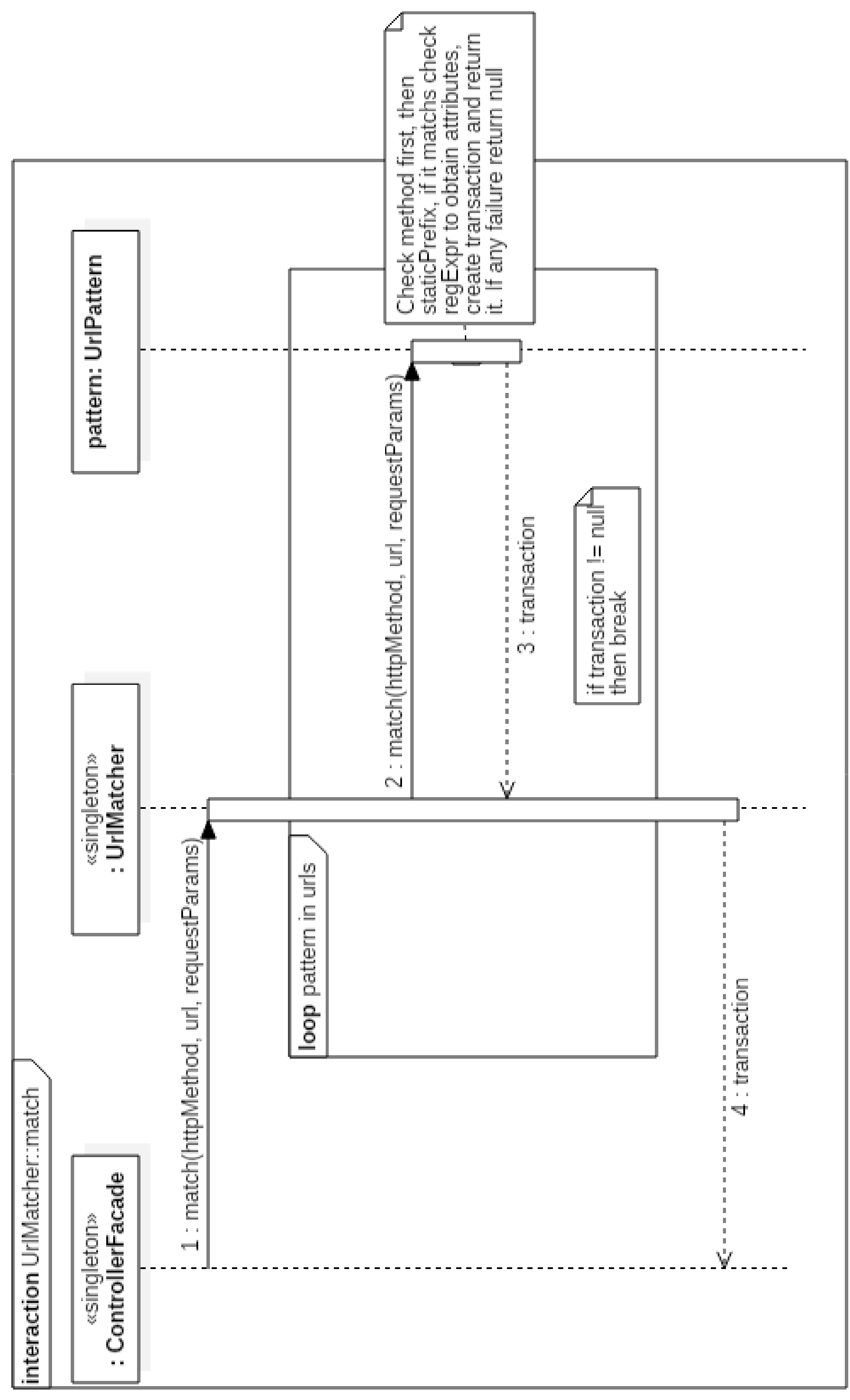


Figure 5.5: Sequence diagram of the URL matcher matching a request.

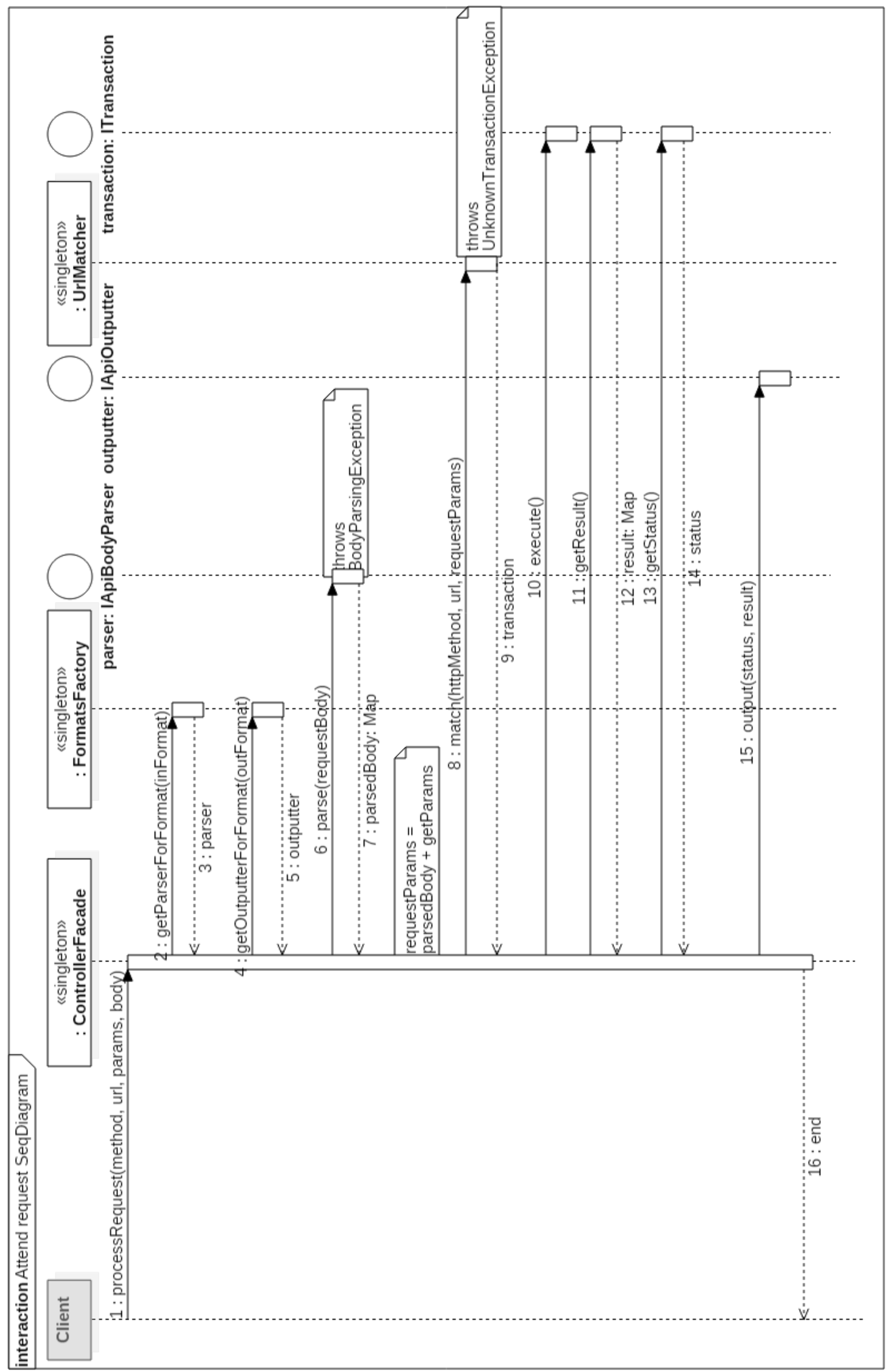


Figure 5.6: Sequence diagram attending a request.

the printable exceptions and send them to the printer to be printed, indicating the status of the Exception as the HTTP status of the answer. It is responsability of the *index* (the *main* of our code, where all request executions will start) to catch all the other exceptions, log them to the error log and send the user an answer with status code 500 (internal server error). The figure 5.7 illustrates the exception management of the system.

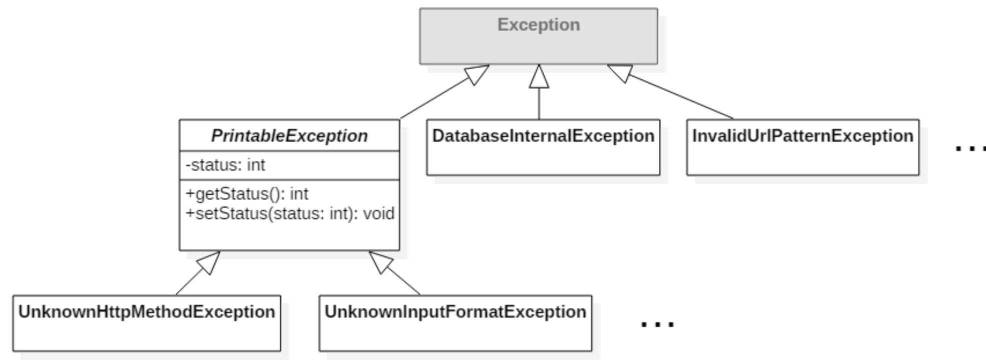


Figure 5.7: Class diagram of the API hierarchy of exceptions.

5.3.5 Data layer

The data layer of our system is the responsible for isolating all the database peculiarities from the external system, we designed it in a way that can be easily modified in case of needing to change the database manager of the system. We should keep in mind that, not like other environments, PHP does not have, in principle, a unified interface to access database systems. Each system has a different library with a different interface to access it.

We have three data objects we need to access of the database: `Period`, `Category` and `Item`. These classes are the central part of our system, and possess a method `toMap` which allows to convert them to associative arrays **with only basic-type values**, ready for the transactions to save them into their `result` attribute to be printed. Each one of these objects can be created making use of their *gateway*. The gateway makes a bit the function of the *finder* of the *row data gateway* pattern, but also allows to perform the operations of the gateway in that pattern (saving and deleting the row). Despite of this, it is not exactly an *active record* pattern, since the gateway does not store any information on its own. It simply encapsulates the database code to isolate it from our data objects. In the case of `Category` we are only able to get them but not to update or delete them, since this is the only required functionality for the API in this aspect.

To obtain the right gateway for the database system we are managing, giving the code an easy way to swap to between storage systems to achieve the desired compatibility and interchangeability, we make use of the *abstract factory* design pattern. To do this, first we create

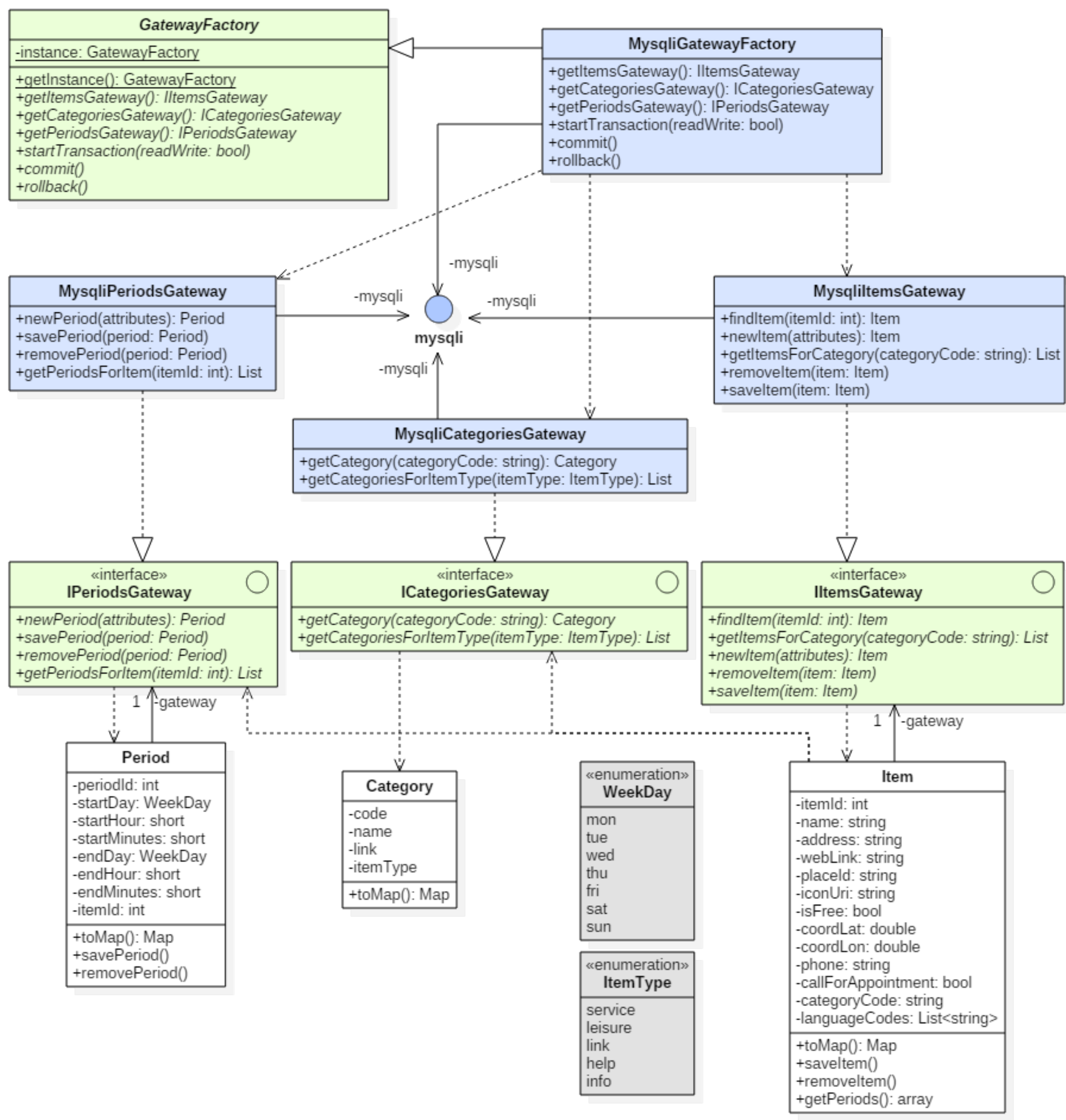


Figure 5.8: Class diagram of the data layer of the API. The interfaces which wrap the abstract factory are represented in a light-green colour, while the family of the gateways for *mysqli* are presented in blue.

an interface for each gateway which allows us to isolate the data objects from the concrete *family* of gateways in use. We create then an abstract class which will be the root of a hierarchy of factory-implementations for each supported family. This class will behave like a *singleton* pattern, but instead of instantiating itself it will instantiate the right factory-implementation which will give us the right gateway. Apart from the different methods to obtain each of the gateways, this class provides also methods to start and finish the transactions to help keeping the integrity of the database.

The figure 5.8 shows the UML class diagram for all the described data-layer with an example of the gateway family for *mysqli*[5].

5.3.6 Transactions execution

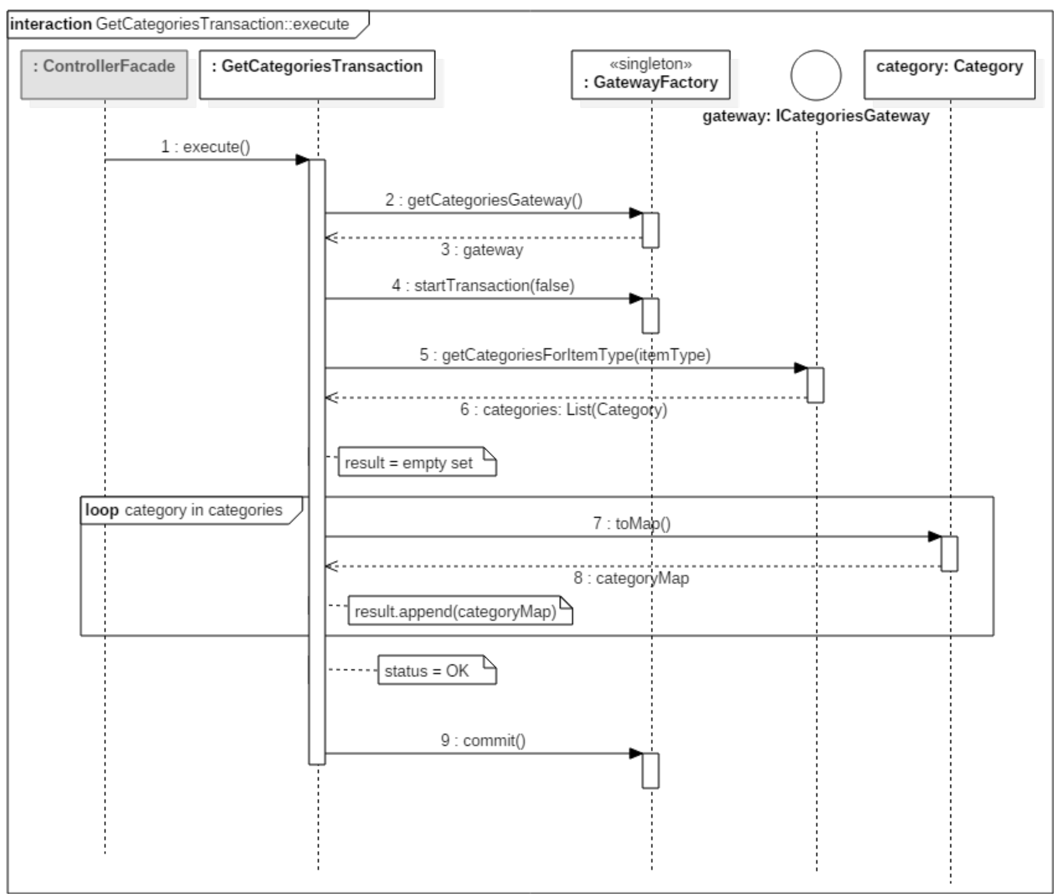


Figure 5.9: Sequence diagram of `GetCategoriesTransaction::execute`.

[5]The PHP extension *mysqli* is a very common extension which allows PHP to connect with MySQL databases.

To illustrate a little bit better the way the transactions are processed and how the domain layer interacts with the data layer we will explain in more detail the interaction of the transactions of the system described in the section 5.3.3. Each one of them is accompanied by an interaction diagram which helps to follow the explanation.

The first one, `GetCategoriesTransaction`, is the transaction which allows us to obtain the list of all the categories of a given type. It starts, as all of them, when the `ControllerFacade` calls its `execute` method. It obtains the gateway for the categories and indicates to the factory it wants to start a reading transaction. Then it calls the `ICategoriesGateway` obtained from the factory to get the list of categories (notice that the transaction does not know whether it is working with *mysqli* or another family). It configures the `result` as an empty set of objects and iterates over all the categories to map them to associative arrays that it saves into the same attribute. Finally, it sets the status to the HTTP OK (200) and commits the transaction with the factory. All of this can be visualised in the figure 5.9.

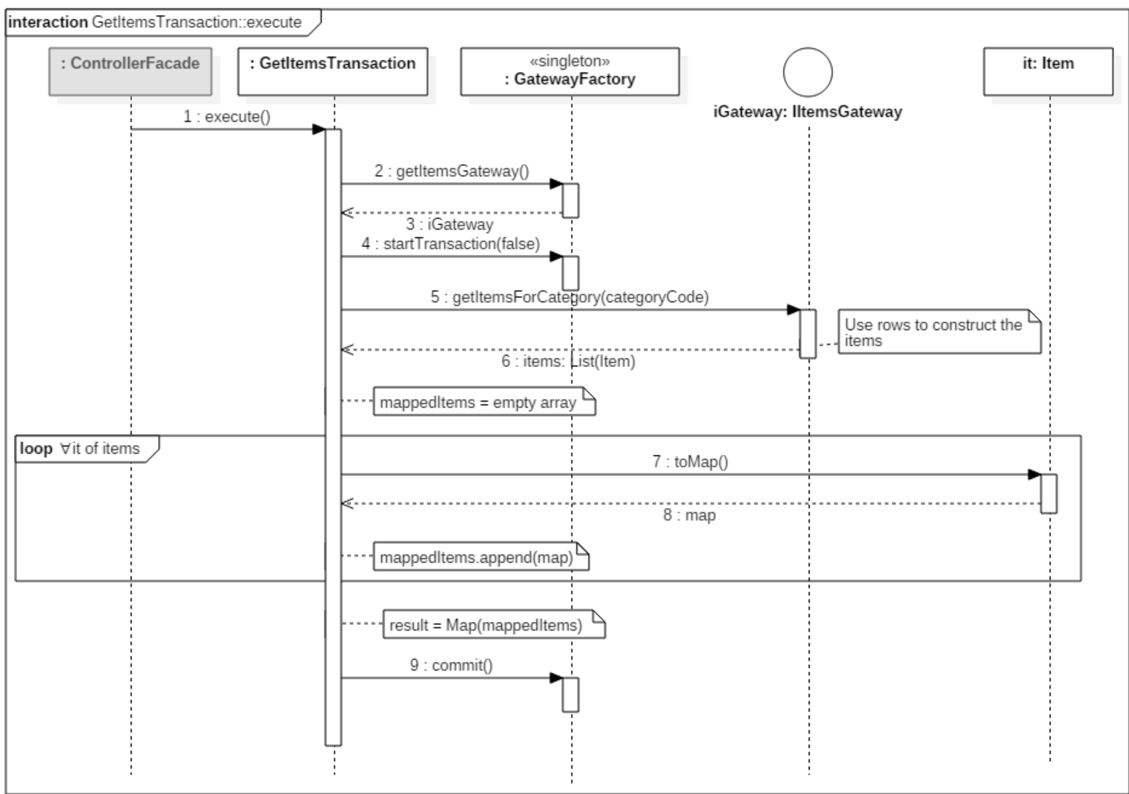


Figure 5.10: Sequence diagram of `GetItemsTransaction::execute`.

The case of getting the items is quite similar, as shown in the figure 5.10, but it has a peculiarity which needs to be further explained. When mapping the items into associative arrays to save them into the result we will find something that can be a little bit confusing: when mapping the

items, we need to have a field `openingHours` with all the periods associated to it. We decide to implement this functionality right in the method `toMap` of the item, which will perform a lazy loading of the periods associated to it (using its item id), as shown in the interaction diagram in the figure 5.11. The item needs to make use of the periods gateway, generating a little bit more of coupling between `Item` and those classes. Whether it is better to generate this coupling with the transaction class or with the item is not clear and depends on the functionality which might be added later to the API, in our case we chose this option.

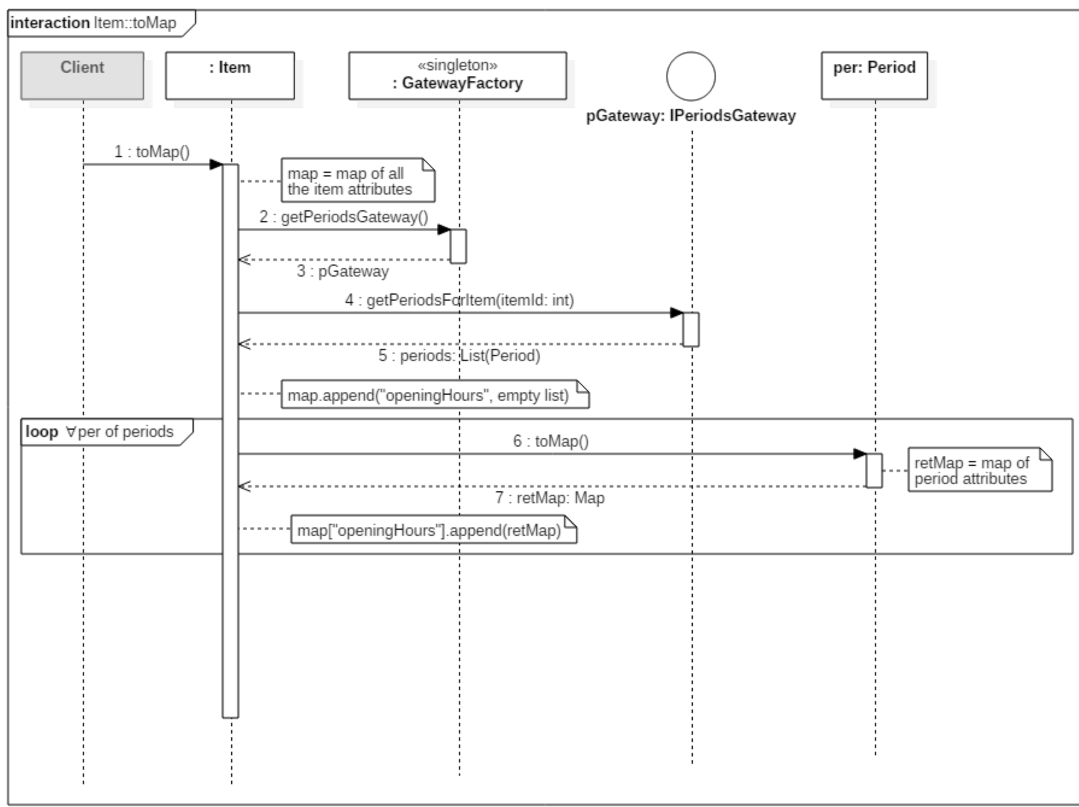


Figure 5.11: Sequence diagram of `Item::toMap`.

Deleting an item is a very simple operation, in fact, since it only requires to obtain the item and call to `removeItem` inside it. The item itself will delegate this method to its gateway, which will delete it from the permanent storage. Notice that thanks to our design of the database (section 5.2.2) deleting the items is enough to get all the associated periods to be removed from storage. The figure 5.12 illustrates the described behaviour.

The creation of an item, for the same reason as the two before, requires the creation of the associated periods. This will be performed by the transaction, as we can see in the figure 5.13. As stated in the section 5.3.1 and since we are creating a new item all the periods should have

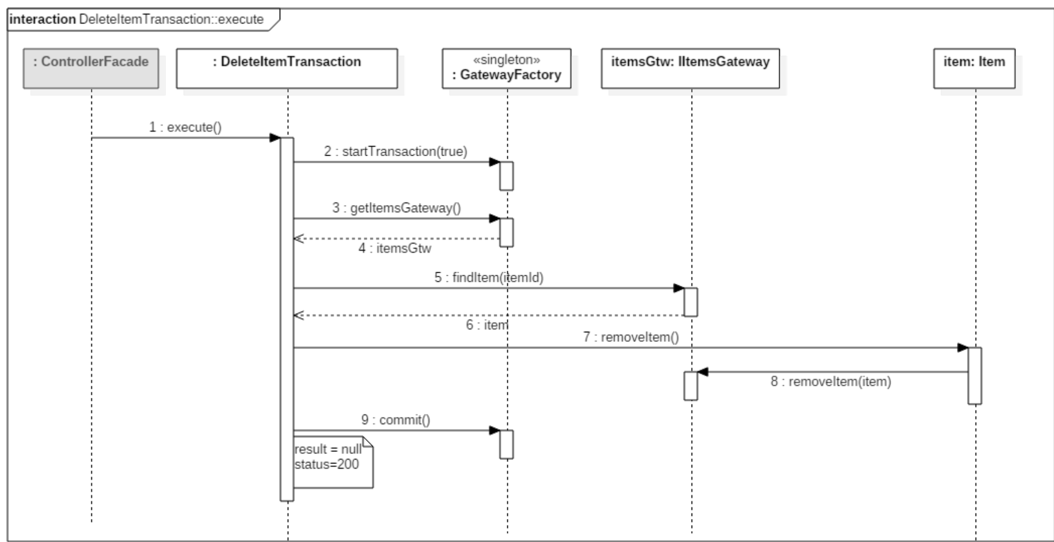


Figure 5.12: Sequence diagram of `DeleteItemTransaction::execute`.

the *new* keyword as `periodId`. The transaction should check it and trigger an error if this is not the case.

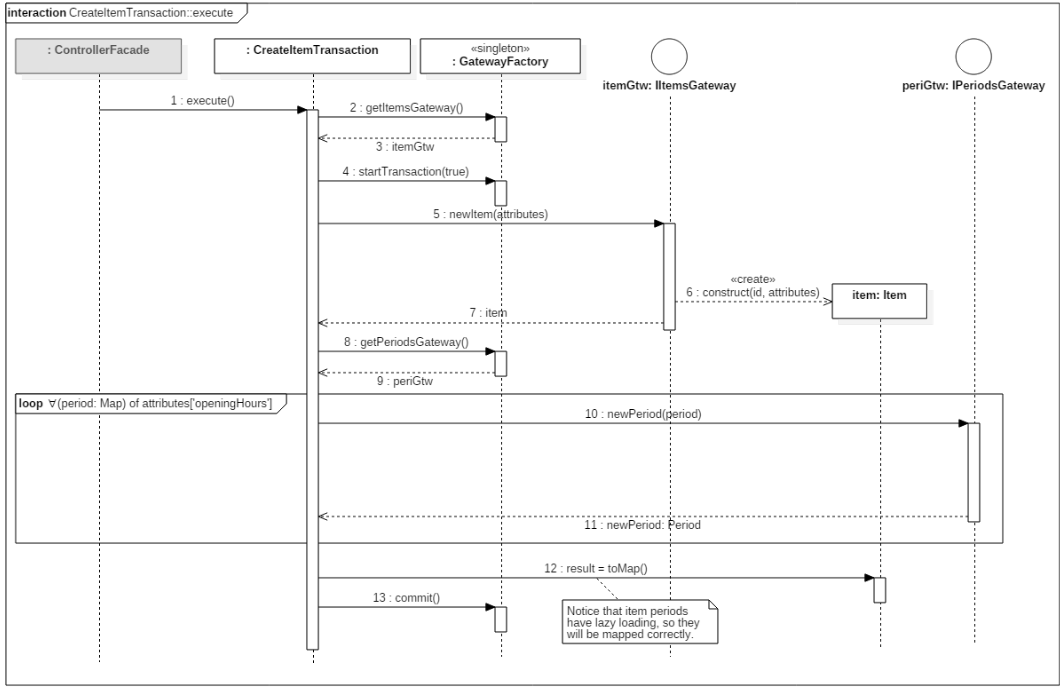


Figure 5.13: Sequence diagram of `CreateItemTransaction::execute`.

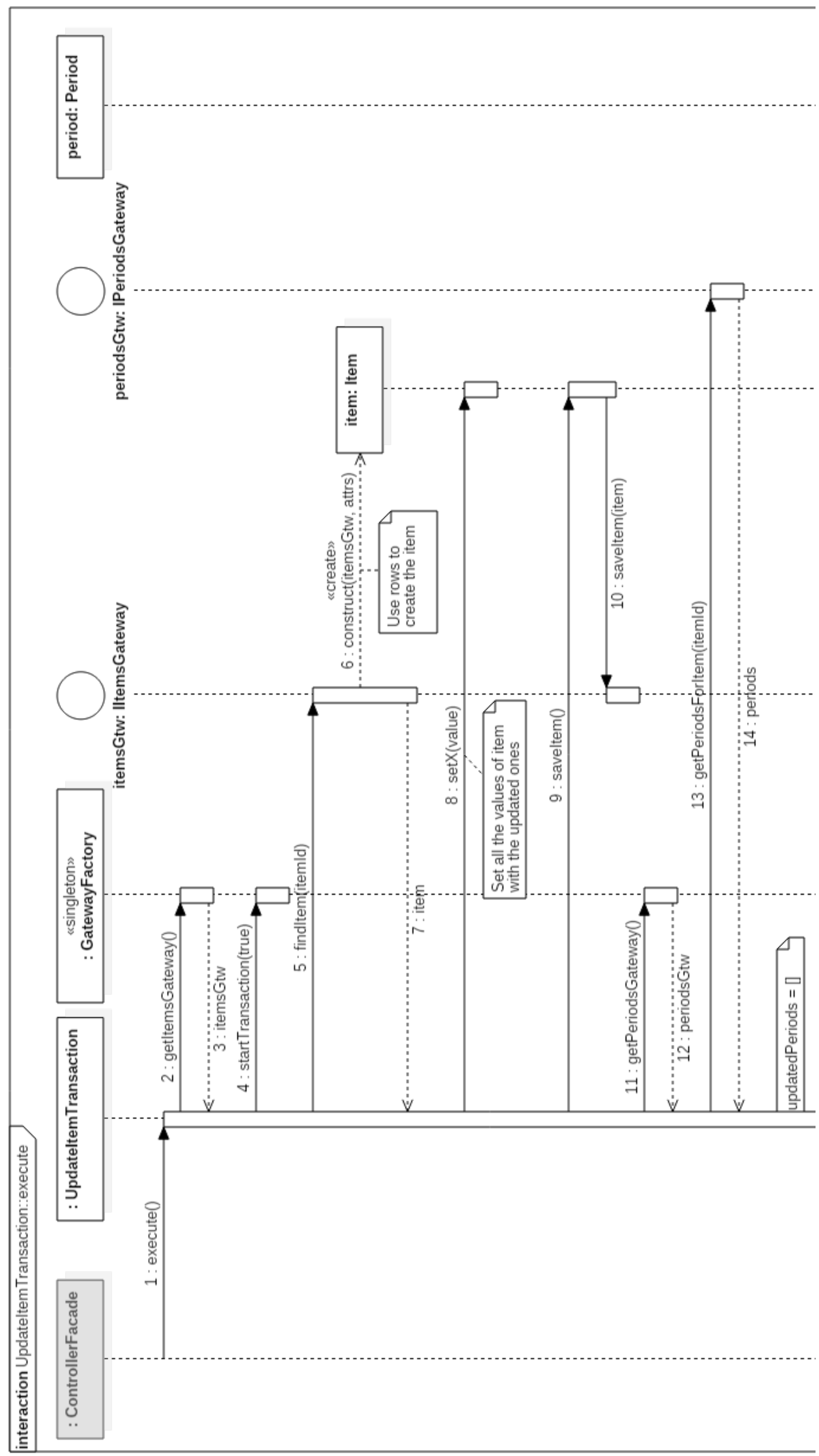
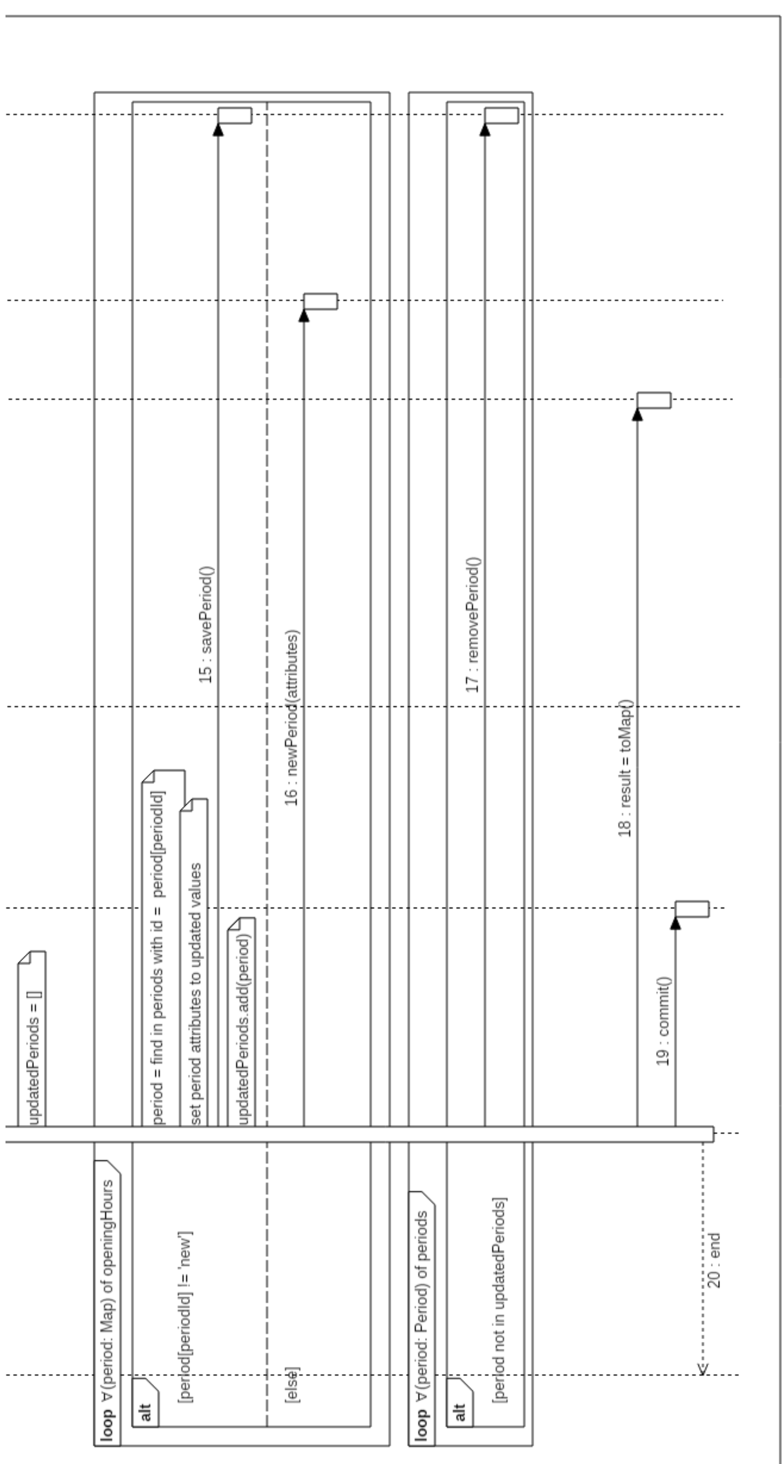


Figure 5.14: Sequence diagram of `UpdateItemTransaction::execute`.


updatedPeriods = []
loop ∀(period: Map) of openingHours
alt
[period[periodId] != 'new']
period = find in periods with id = period[periodId]
set period attributes to updated values
updatedPeriods.add(period)
15 : savePeriod()
[else]
16 : newPeriod(attributes)
loop ∀(period: Period) of periods
alt
[period not in updatedPeriods]
17 : removePeriod()
18 : result = toMap()
19 : commit()
20 : end

Finally, updating an item is a bit complicated because we have to manage three different cases for the periods: the ones which were already present, the new ones and the deleted ones. After getting the gateway and retrieving the item from the database, we set all the values of the item to the ones received in the body of the request and call to `saveItem`. Then, we start an empty list for all the updated periods. As we loop through the periods received in the request (saved in the field `openingHours` of the transaction), we check if the received period is *new*. If it is, we simply need to create it with the given values and associated to the given item. In the opposite case we search for the given `periodId` in the periods of the item. If it cannot be found, we simply throw an exception. If we do, we update it and save the period in updated periods. Finally, we iterate all the old periods of the item deleting from storage all those one which have not been updated, save the item mapped in the result and commit the transaction. Notice the item will be printed in its updated version of the periods because the periods are loaded from storage when calling `Item::toMap`.

5.4 Front-end: The PWA

Finally, the Progressive Web Application design will be explained here. The design was made keeping in mind it would be implemented in JavaScript, which has a different work flow from the usual object-oriented designs (it is prototype-based). Keeping this in mind, we decide to take advantage of its special characteristics in our design. Although this decision makes the system design a little bit less adaptive to other technologies, this is not a problem since JavaScript is going to be the client-side web technology for long time (although some new technologies are appearing, like Web Assembly[28], but it surely will not cause JavaScript to disappear) and it will allow us to assume in the design some characteristics of modern languages. This characteristics, anyways, use to be possible to emulate in the classic object-oriented paradigm so it would not be too hard managing to implement them. Despite of this, we wanted to express our intention to make use of these features and, in consequence, there are things which UML does not allow us to express as clearly as we would like. The main example of this (which is our main interest in this case) can be the callbacks: passing functions as if they were objects so they can be called by the receiver. We express the format of these callbacks in our UML diagrams making use of a «`callback`» stereotype, where the attributes are the expected parameters of the callback. They are marked in a light green in all the diagrams so they can be easily identified and they all inherit from `Callback`. We will also use `...` before a final argument name to express a list of arguments of variable length. We will consider any class or callback to be a descendent of `Object`, and if we indicate a parameter is an *Object* we are meaning that parameter may have any attributes, being those simple key-value pairs. We can understand an *Object* in this case as the generally known as associative array, map or dictionary in another languages. In JavaScript this is the root for everything in the code (even the functions are *Objects*).

In other languages with a more classic approach, like Java, we can emulate the callbacks

feature creating interfaces with a single method, which describes the expected "callback", and making the class which implements the callback to implement said interface, passing the object itself as the argument to the desired method (or implementing a complete observer pattern, in case that many “callbacks” need to be managed).

5.4.1 The pages

The main class of the PWA is the `Page`. Each `Page` represents a different screen in our system. It is an abstract class from which all the pages extend. It possesses a title string to display in the navigation bar, an indicator of whether that bar should be visible or not when we are in the page and a `visible` attribute to indicate if the page is currently rendered to screen or not. In the figure 5.15 we can see an example of a rendered page and the navigation bar visible.

The most important method of every page is `render`, it receives as a parameter an element of the DOM that is the application container to render the pages on, and it “draws” all the elements of the page on it. All the children classes of Page should call always the parent’s render method, since it is there where the visible value gets updated to be true. The `load` function is called when the page is loaded by the application but it has not been rendered yet. Although right now the page is rendered immediately after it is loaded, we insert this method because it might be possible that in future versions the page could be rendered several times but loaded only once. The `resize` method is called when the page is the current page of the application and the window gets re-sized, this happens when we are visualising the web on a navigator and we scale the window, or for example when we rotate the screen on a mobile device. To control when the page is hidden we add the method `onHide`, which gets called when the page is about to be exchanged by another page. The methods `getPageHeight` and `getPageWidth` allow to obtain the available space (in pixels) for the page to render and `isVisible` returns whether the page is visible or not. The class diagram in the figure 5.16 shows the hierarchy of page classes in our application.

Figure 5.15: A page with navigation bar displayed on the system.

Each of the page classes has an associated *state* class (in blue in the diagram). This is used by the `Router` to be able to recreate the web we were in before, more information about this class can be found in 5.4.2. By now, all we need to know is that the state is a basic JavaScript object, with no functions or class involved, which will allow the page to be restored from the browser history. The `PageState` class is the root of all the page states hierarchy. The information about

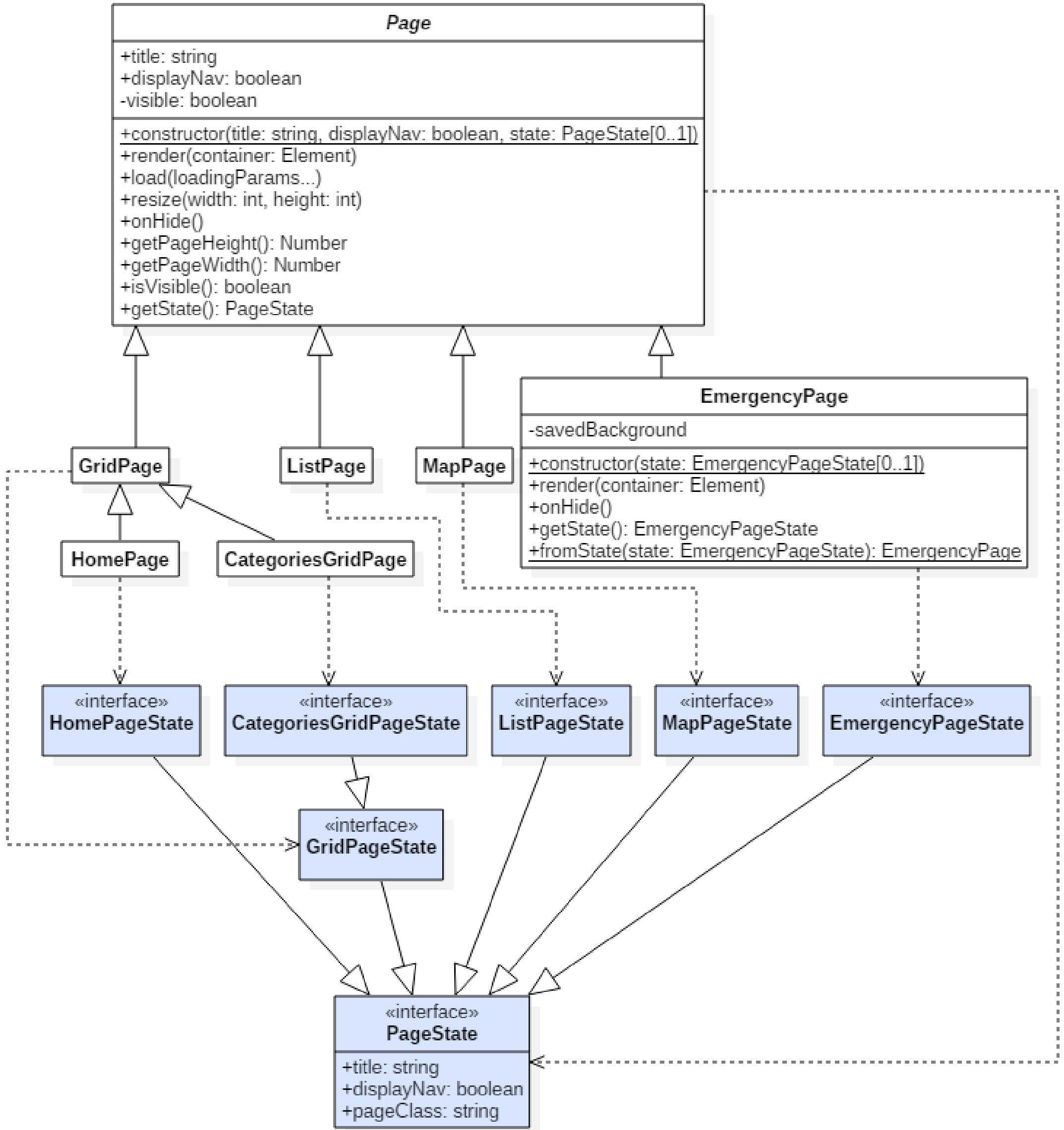


Figure 5.16: Class diagram of the hierarchy of pages.

the visibility of the page is not saved to the state because it would not make sense, when the page is recreated its visibility will be again `false` until a call to `Page::render` is made. To obtain the current state of a page we can call `getState`, this method should be implemented in all the page classes to include their own state information to the one of the parent. When a page is created, an optional attribute allows to pass in an state to restore the values from it. Notice we save a string attribute called `pageClass`, this attribute will allow us to know which class we should instantiate when restoring the state in the `Router`, each of the children classes shall

override this value on their implementations of `getState` to set their own class names.

As an illustrative example, we can see the class `EmergencyPage`, which will override the rendering method to introduce its own rendering. Since it will change the background (it has an special red background), we add an attribute called `savedBackground` to save the previous one. When calling `render`, the page will change the background of the application to a reddish colour, it will also change the style of the navigation bar and add the content of the page (a text which says 112 and a button to perform the call). On the event `onHide`, the page will restore the original background. The method `getState` only needs, in this case, to call the parent `getState` method, overwrite the `pageClass` value with *EmergencyPage* and return the state. Notice the static method `fromState`, this method will appear in every instantiable page class and is used by the `Router`.

The `GridPage`, `HomePage` and `CategoriesGridPage` classes are explained in more detail in the section 5.4.3.

5.4.2 Application, routing, resources and navigation bar

At the top level on the structure of our application we have the class `App`. This class, designed with a singleton pattern to make sure it is accessible from any part of the code, is the responsible for managing all the other classes of the application. It is instantiated when the document loads and it is responsible, during its construction, of instantiating the router and navigation bar classes. The `App` class works as a point of connection between the elements of the application. It will be called when a navigation to a new `Page` is required, when the maps API needs to be required or to clear the container before rendering again. It contains the following methods:

- `navigateToPage`: it receives a page and a variable number of loading parameters. The method will perform the transition to the given page making sure the state gets saved to the browser history. It performs this transition calling to `App::loadAndRender`. The sequence diagram of the figure 5.18 displays this behaviour and illustrates some other methods of this class.

- `fakeNavigation`: it allows the system to "simulate" a navigation to a new page without making an actual change of page (but saving a new state to the browser history). This is used by the `MapPage` when displaying several items, more information about this can be read in the section 5.4.6.

- `updateCurrentSavedState`: updates the state of the current page saved to the browser history. Notice this will not create a new entry in the browser history but, instead, it will replace the previous one with a more updated version.

- `loadAndRender`: It performs the change from the current page to a new one, but this method does not save the change into the browser history. The method starts by calling `onHide` in the current page to allow the page to execute any code it considers necessary

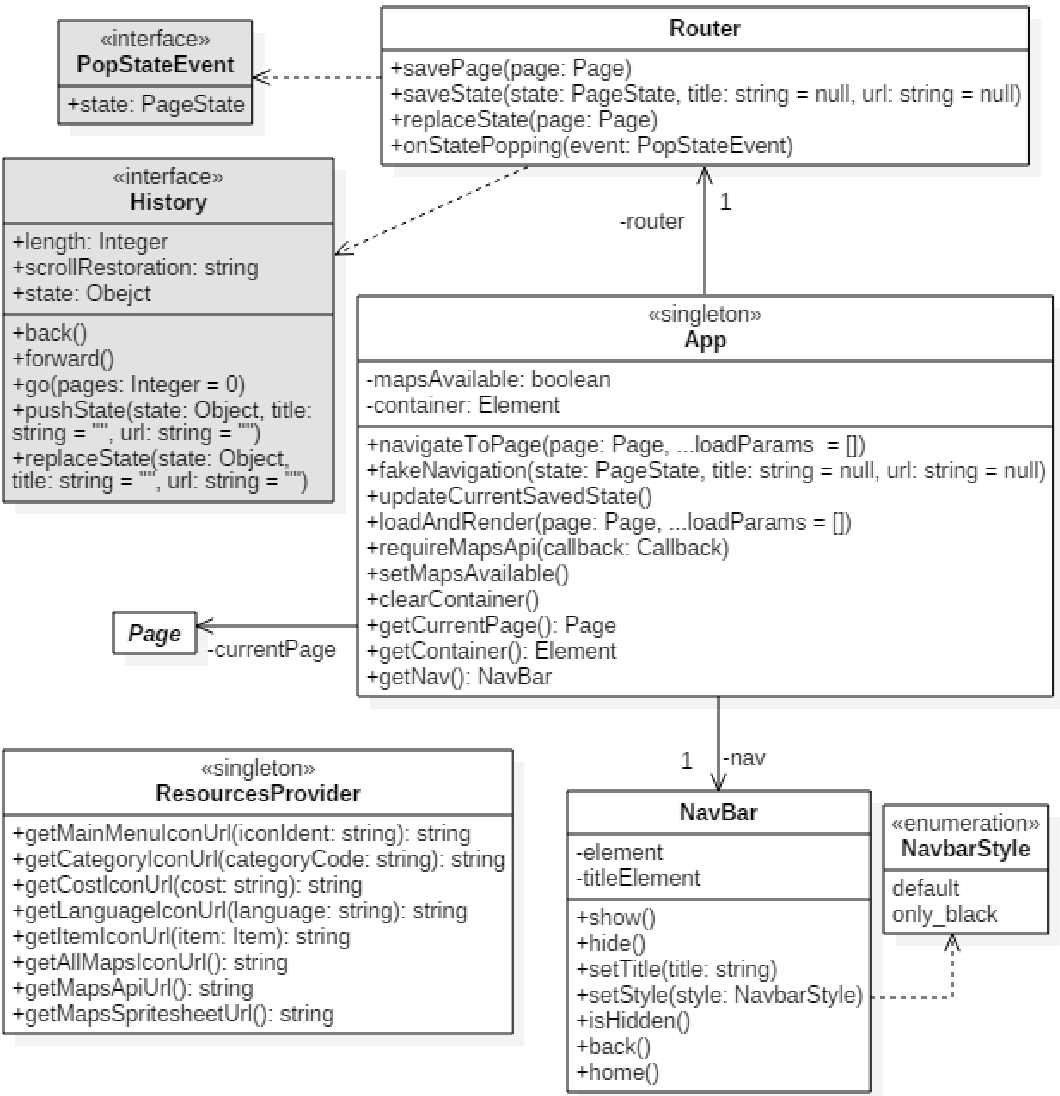


Figure 5.17: Class diagram of the `App`, `Router`, `ResourcesProvider` and `NavBar` classes. These classes are the ones at the top level in the structure of the PWA. Notice the interfaces in a darker colour are interfaces provider by the browser.

before getting replaced. After this, it will replace the associated current page and call `load` with the given load parameters. If the new page does not display the navigation bar, it will tell the navigation bar to hide, in the opposite case it will show it, restore it to the default style and change the title to the one indicated by the page. Finally, it will clear the page container and call the page to render itself on it.

- `requireMapsApi`: the method requires the Maps API (which needs the insertion of an script in the DOM to be included) and stores the passed callback to be called when the API loads completely. If the API is already loaded the callback is called immediately.
- `setMapsAvailable`: called when the maps API finishes loading. It will change `mapsAvailable` to true, check if there is a callback registered and, if there is one, it will call it.
- `clearContainer`: simply deletes all the content of the page container to allow a new page to be rendered.
- `getCurrentPage`: gets the currently displayed page.
- `getContainer`: gets the container for the pages to be rendered in.
- `getNav`: gets the navigation bar object.

The class `Router` is in charge of loading the right pages having in account the browser history. The browser history allows us to move inside our single page application as we would do on a multi-page web and works as a stack of *states*. The history object available from JavaScript allows us to save *states* associated to a title and a URL in the browser history[6]. When the user presses the back button in his/her smart phone or the back arrow on the browser, the navigator offers us an event to listen to this interaction and perform any desired action. It also offers us a method to emulate this behaviour from code, so we can cause it to be triggered from the navigation bar *back* button.

When the `App` class is instantiated (after finishing loading the document) and it instantiates the `Router`, this one checks automatically the browser current state. It is necessary to make this check because when we are coming from external pages back to our page, the listener to state popping events (the events triggered when the user presses back) will not be triggered. After checking the browser state to know in which page we were in case we are coming back, it registers the listener for the state popping. Notice that our `NavBar` class will use this mechanism of the browser to go back, and in consequence it will also trigger the state-popping event. The listener it registers for this event will be the method `onStatePopping` of the `Router` object.

The methods `savePage` and `saveState` allow us to save a `PageState` in the browser history. The first one requires automatically the necessary information to the given `Page`, while the second needs to receive the state and optionally a title and a URL. This last method is employed by `App::fakeNavigation` to simulate a navigation that has not occurred. We can use `replaceState` to replace the last saved state by a new one of our choice, as the method `App::updateCurrentSavedState` does. The method `onStatePopping`, triggered when the user presses the back button well in the phone, well in the navigator or well in our navigation bar, checks for the popped state `pageClass` attribute and instantiates the right

[6]The living standard for the `History` interface can be found in the following web address: *https://html.spec.whatwg.org/multipage/browsers.html#the-history-interface*.

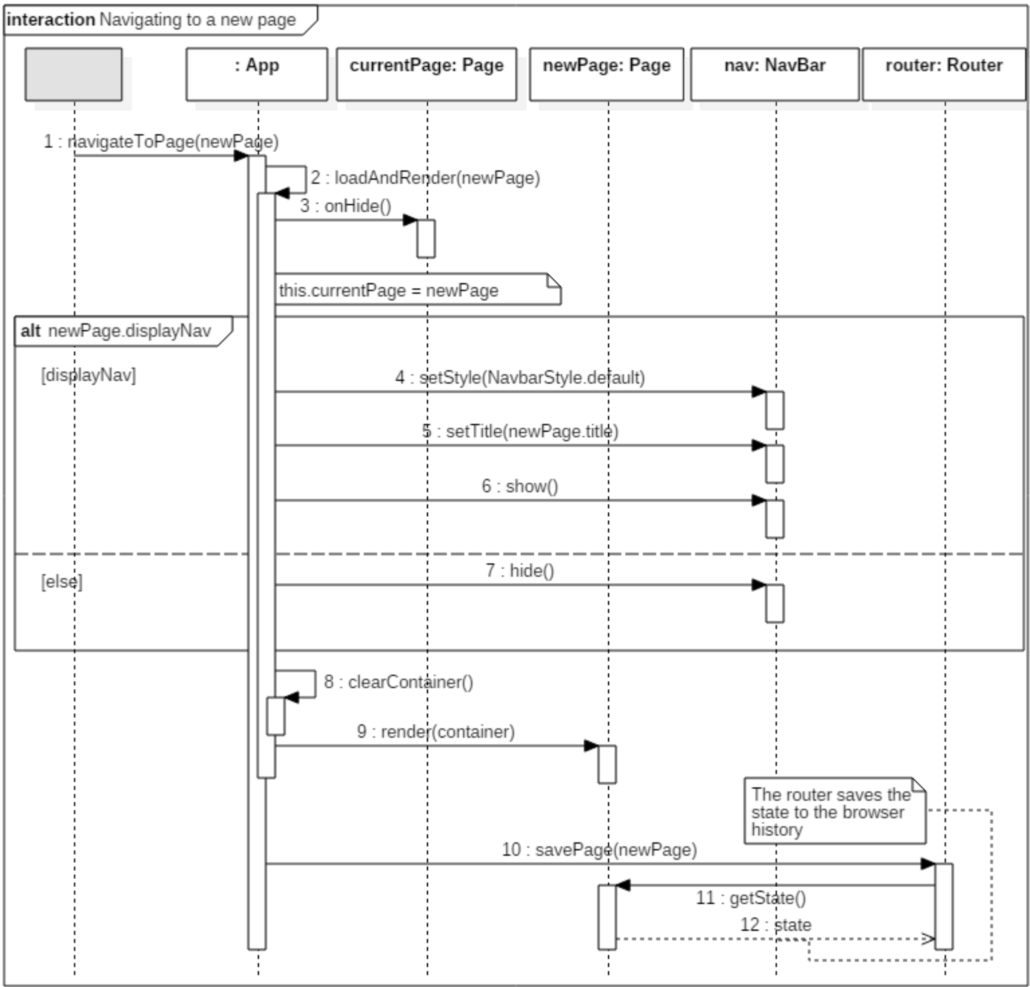


Figure 5.18: Sequence diagram for `App:navigateToPage`.

page class, not by calling directly to the constructor but, instead, by calling the static method `fromState`. This becomes specially important when popping a `MapPageState` (for more information check the section 5.4.6). Once the router has obtained the restored page it simply calls to `App::loadAndRender` to display the page without saving a new navigation entry to the history. If the `fromState` method returns a `null` value, the method finishes without calling `App::loadAndRender`.

The `NavBar` class is intended to manage all the stuff relative to the navigation bar of the application. It provides methods to show the navigation bar, hide it away, change the title, the style (there is a `default` style as the one shown in the mock-ups in the services page, for example, and an `only_back` style which corresponds to the style of the navigation bar in the emergency

page: transparent, showing only the back button in a black colour), check if it is hidden, go back (as it would when clicking on the back-arrow button displayed in the bar) or go to the home screen (as it would do when clicking on the home button displayed in the bar). The `back` method makes use of the navigator `history` object to make sure it will trigger the state-popping event and will be listened by the router.

Finally, the `ResourceProvider` is a class which encapsulates all the access to the resources of the application. Since this is a web page, generally the resources are accessed by URLs, this class is responsible for giving the right URLs to obtain all the different resources the web page makes use of, like the icons for the categories or items, the buttons or the URL for the maps API.

5.4.3 Grid pages

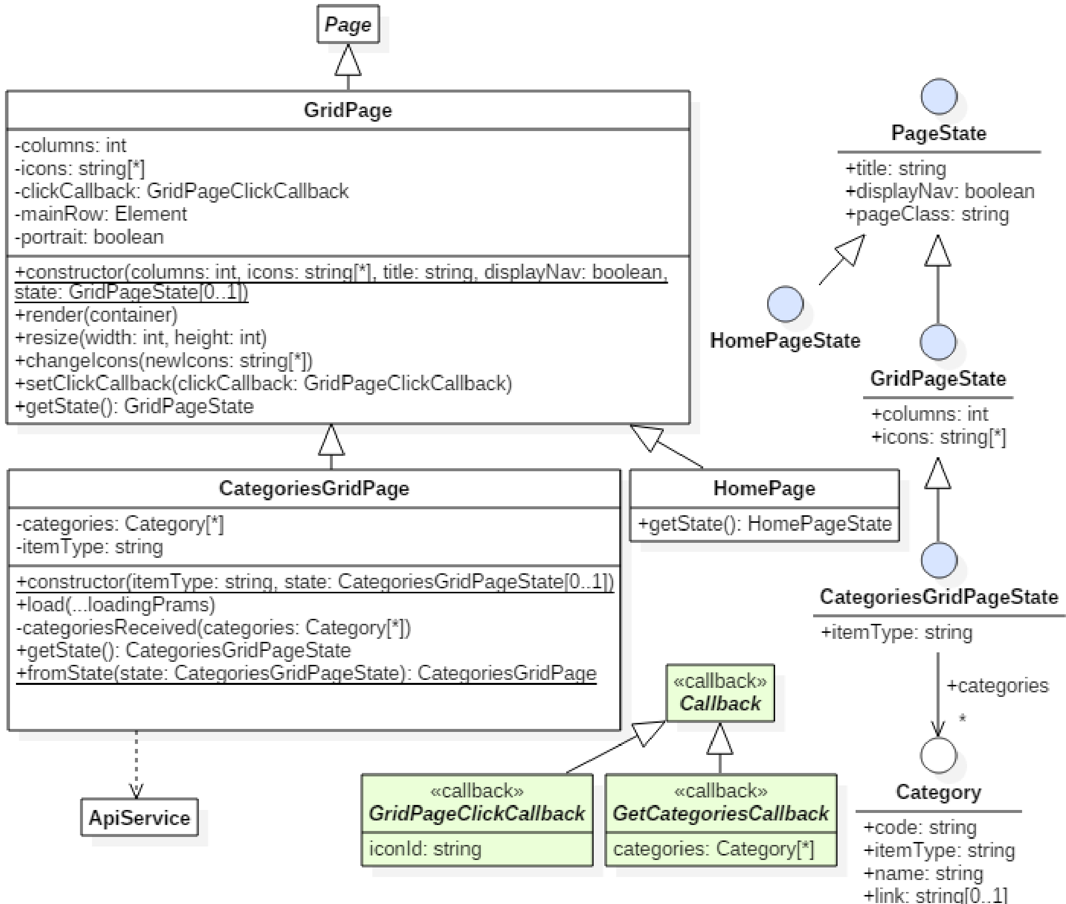


Figure 5.19: Class diagram of the grid pages of the application.

Both the home screen and all the screens displaying categories of our application are organised in a grid-style layout. The rendering of this grid is managed by the class `GridPage`. This page receives, apart from the `title` and `displayNav` attributes necessary to create the parent `Page` class, the number of columns we desire to have in the portrait orientation of the screen (when the height is bigger than the width) and the icons we want to display. The icons are received as an associative array of strings, where each key is the id for the icon to be used when calling the `GridPageClickCallback` set through `setClickCallback` and each value is the URL of the icon we want to display. The `GridPage` will automatically manage the reorientation of the screen, adapting the number of columns adequately.

The home screen of the application is rendered by the `HomePage` class, a very simple class which provides the six icons for the main screen and responds to the clicks navigating to a `CategoriesGridPage` constructed with the right `itemType` attribute. The only exception to this is the emergency button, which will cause a navigation to the `EmergencyPage`, described in previous sections of this document.

The `CategoriesGridPage`, while still relatively simple, overrides the `load` method of `Page` to start a call to the `ApiService`. This call works asynchronously and will perform an AJAX request to obtain fresh categories for the given item type from the API. Once this call is finished, if it returns a success, `CategoriesGridPage::categoriesReceived` will be called to update the interface with the new categories and update the saved state in the browser history. This behaviour is illustrated in the sequence diagram of the figure 5.22, in the section 5.4.5. Please notice that this method, `CategoriesGridPage::categoriesReceived`, matches the interface definition of `GetCategoriesCallback`, required by the `ApiService` class when trying to obtain the categories.

The `Category` interface which can be seen in the diagram represents the interface we expect the categories received from the API to match. JavaScript can parse the received JSON automatically and convert it into a JavaScript object or array. The interface `Category` contains the attributes we expect to find in each item of the array obtained when parsing the answer for the API request. More information on the communication with the API can be found in the section 5.4.5.

The icons passed to the parent (`GridPage`) by the `CategoriesGridPage` shall have as keys the ids of the categories, in order to be able to pick the right category when one icon is clicked to make the app navigate to a new `ListPage` constructed for the selected category.

5.4.4 Items, periods and the list page

Since the items we receive from the domain and their associated periods are more complex, it might be necessary, not like the categories, to have a dedicated class inside our application for them. For this reason, we create the `Item` and `Period` classes, which are the representations inside of our PWA of the items and periods received from the API. This classes appear at the center-bottom in the class diagram of the figure 5.20. They are constructed directly from the

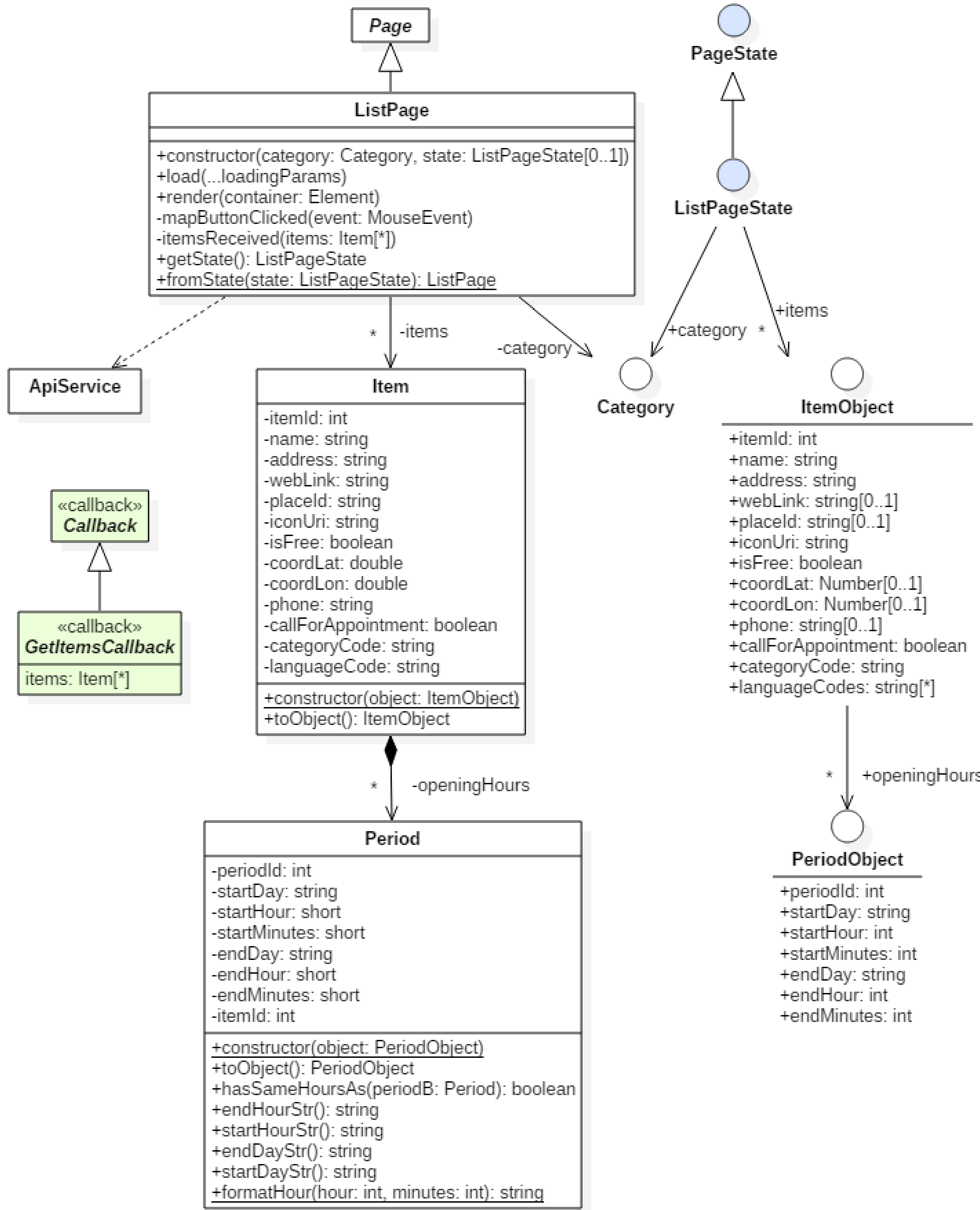


Figure 5.20: Class diagram of the `ListPage` class, `Item` and `Period`.

received objects, which are represented in the diagram by the interfaces `ItemObject` and `PeriodObject`. Both classes present a method `toObject` which allows to obtain the object of

creation, this is handy when we want to store them in a state of the browser history, because saving `Item`s or `Period`s will cause some browsers to silently store a null value instead of the passed state. The `Period` class offers also some methods to facilitate displaying the data they contain or to compare whether they correspond to the same period of time that another given `Period`. This is used by the `ListPage` to group the events, to obtain more details on how periods are grouped check the section 6.3.3.

The `ListPage` class is the page that displays the items in the application. They are disposed in a list-style layout with the icons at the left and the information at the right[7]. This page behaves almost completely in the same way `CategoriesGridPage` does but managing items, so for the sake of briefness it will not be explained here (this explanation can be read in the section 5.4.3). We will just point out that the callback used to receive the items from the API is, in this case, `ListPage::itemsReceived`.

5.4.5 API connection

Both the `ListPage` and the `CategoriesGridPage` make use of the `ApiService` to communicate with the API. This class works as an abstraction for every API communication and if new resources or actions were added to the API in the future, methods to perform those requests should be added to the `ApiService`. Since in our application pages make only one request each, the `ApiService` manages only one request at a time, cancelling the last one if a new one is performed. This functionality could be easily extended to manage several queries at a time. Notice that, since `ApiService` is not a singleton, several pages can manage several instances of this class, so no problems of cancellation of request between pages will ever happen.

To communicate with the API asynchronously in the background the `XMLHttpRequest` object provided by the browser is needed, but the way this interface works is a little bit odd so we introduce an intermediate `ApiAjaxAdapter` specialised in creating the AJAX queries for the API. Another advantage of this is the possibility to modify in future versions the behaviour of the queries. If we decide, for example, to use the `localStorage` of the `window` object to save some information about each received query, we can make this is a clean way from the `ApiService` while keeping our AJAX logic isolated. This is not, however, the mechanism employed for the offline mode. To understand that functionality check the section 6.3.2.

The methods provided by the `ApiAjaxAdapter` allow to make generic requests to a JSON API, and they could be used to communicate with any other API through AJAX if it were necessary. Since we only make `GET` requests from our application, only the `get` method will be implemented by now. Each `ApiAjaxAdapter` manages a new `XMLHttpRequest` object, trying to start a new request when another one is in process will throw an error. The `ApiAjaxAdapter` ignores completely which kind of objects it is managing, it is the responsibility of the `ApiService` to

[7]Check the section 4.2.3 to take a look at the mock-ups of the application and how the layout of the `ListPage` shall look like.

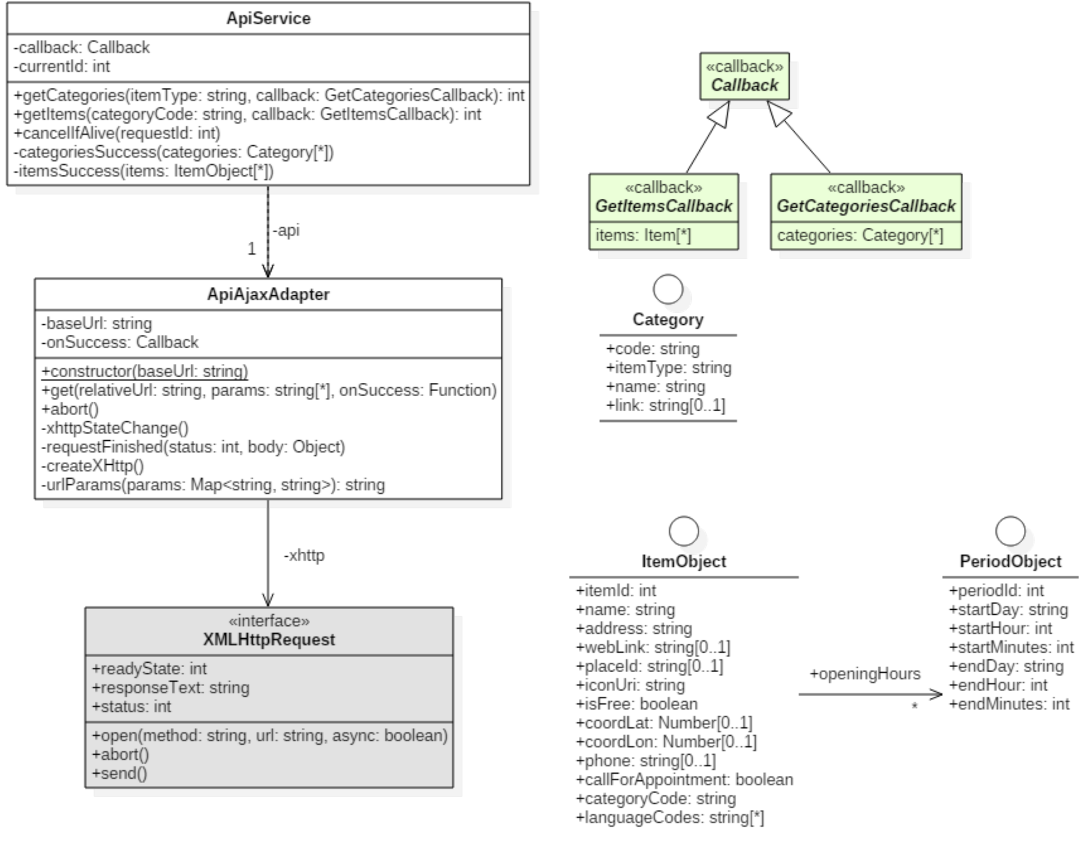


Figure 5.21: Class diagram of the `ApiService` and `ApiAjaxAdapter` classes. The darker interface `XMLHttpRequest` is provided by the browser.

receive the objects and keep track of whether their interface corresponds to `ItemObject`s or `PeriodObject`s, as well as creating the `Item` or `Period` objects before calling the corresponding callback.

The class diagram in the figure 5.21 shows the previously described structure. Only the used methods and attributes of `XMLHttpRequest` are displayed[8].

Finally, in the figure 5.22 we show an example of a `CategoriesGridPage` loading and how it receives the asynchronous answer from the AJAX with the categories. The diagram does not show the continuation of the receiving method because it would become too small to be readable, but the method would continue calling the parent method to replace the icons with the new ones (making use of the `ResourcesProvider`), calling the `App` to clear the container and rendering the page again. Of course, if the page has been already exchanged by another one the

[8] The complete living standard specification for this interface can be found in *https://xhr.spec.whatwg.org/*.

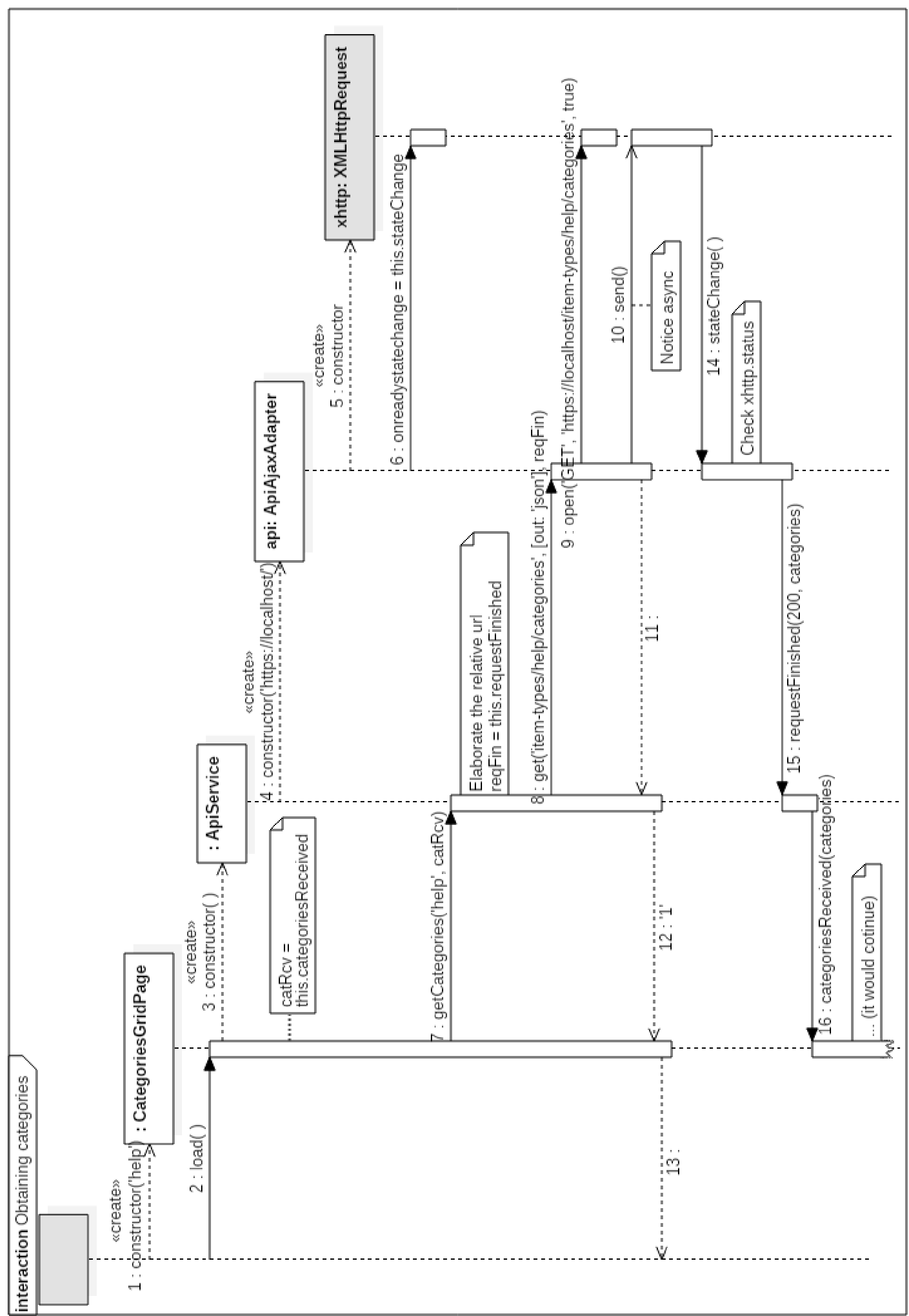


Figure 5.22: Sequence diagram showing how the `CategoriesGridPage`, `ApiService` and `ApiAjaxAdapter` classes interact to obtain the categories for a given item type from the API when the `CategoriesGridPage` load.

method `CategoriesGridPage::categoriesReceived` simply finishes without performing any action.

5.4.6 Map page

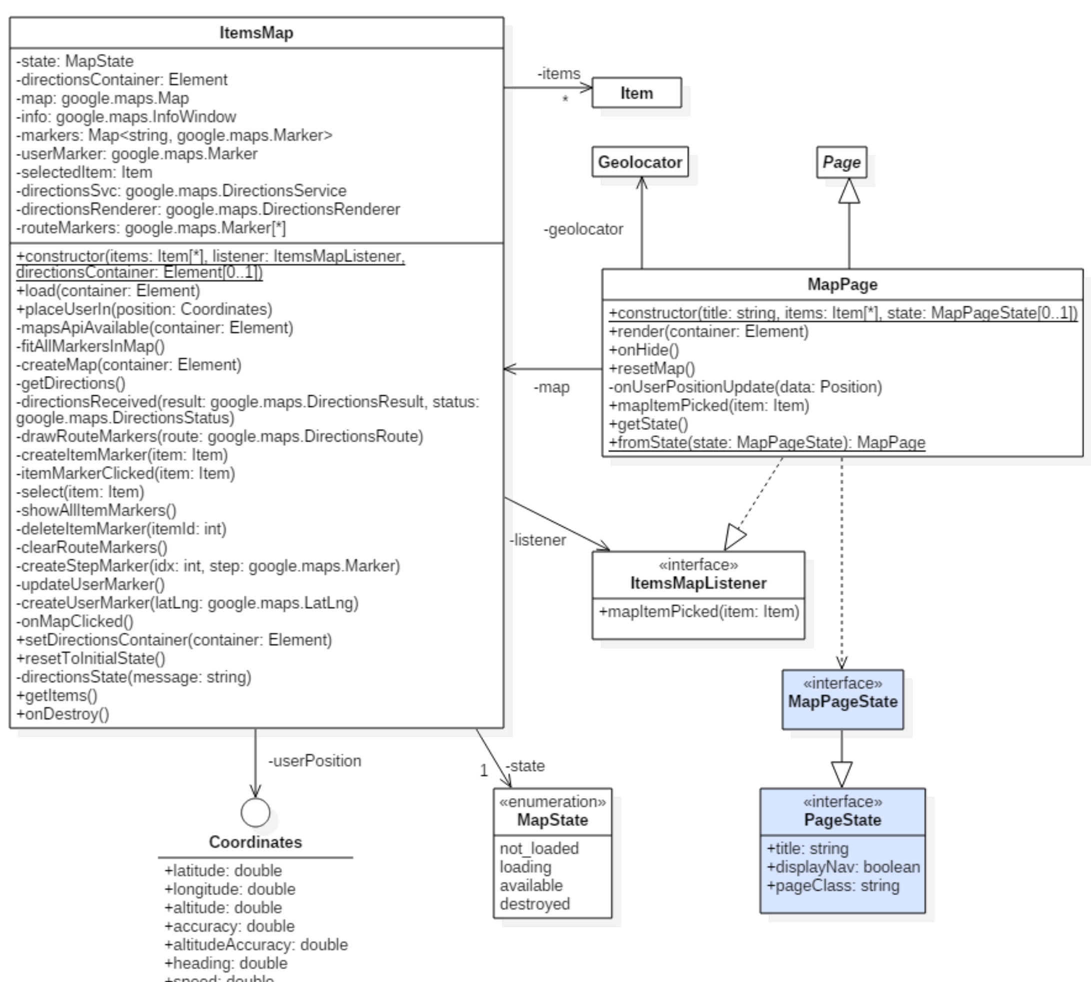


Figure 5.23: Class diagram of the `MapPage` and `ItemsMap` classes.

The `MapPage` is the responsible for creating and displaying the map with the items over it. We will make use of the Google Maps API, and to draw over it we create the class `ItemsMap`. This class encapsulates all the logic behind the map behaviour, it manages a list of items and creates the markers for them, allows to choose one, requests and draws the route to the selected one, manages the helper panel to print there the directions, notifies information, etc. This way,

the MapPage only needs to load the ItemsMap on rendering the page, to call the method to destroy it in onHide and to listen for picked items to update the state of the page (when we are seeing a group of items and we pick one, on pressing *back* we want the application to show again all the items, not to go back to the ListPage).

There are, still, a couple of things more that MapPage does and we would like to point. On the one hand, it creates and listens to a Geolocator, which provides it with information about the user position. The MapPage passes this information to the ItemsMap::placeUserIn method so this one is able to update the marker position to the right location. The internals of the Geolocator are explained in the section 5.4.7. On the other hand, the MapPage also controls when we have came back into the same page and calls to ItemsMap::resetToInitialState. This behaviour avoids the map to be reloaded when pressing back to see all the items again. To achieve this, in the ItemsMap::fromState static method we check whether the state to restore contains exactly the same items as the currently displayed MapPage (if one). If this is the case, it calls to resetToInitialState on the ItemsMap of the current page and returns a null value. As mentioned in the section 5.4.2, when receiving a null value back from fromState, the Router simply does nothing.

The ItemsMap class, as previously stated, encapsulates all the logic for the connection with the Google Maps map. The use of the classes from Google Maps is complex and this document would become too big, but more information on the subject can be found in [12]. When calling load on an ItemsMap, it will automatically call App::requireMapsApi to cause the Google Maps API to be loaded (if it was not loaded already). Once it has ended loading, the application will call the callback provided by the ItemsMap, which shall be MapPage::mapsApiAvailable. Once this happens a google.maps.Map element is created on the container indicated by the MapPage when calling ItemsMap::load and all the google.maps.Marker for the items are created. The object will keep track of these items, hiding all the non-selected ones when one item is selected or showing them when resetting it to the initial state. It will also keep track of the user location with the periodic notifications from the MapPage and update the user marker accordingly. If the ItemsMap is created for a single item, the item is automatically selected. When an item is selected and if the user position is available, a call to the Directions API of Google Maps is initialised. On receiving the route, it is displayed over the map and the directions for it are rendered on the directionsContainer element, if it is not null. The route is not updated until we go back and select an item again, even if we deviate from it, but this was a design decision because our number of available requests to the Directions API is limited.

5.4.7 Geo-locating the user

Thanks to modern web standards, geo-locating the user is not as hard as it would seem. We can make use of the interface Geolocation, through the object provided by the browser to this purpose in navigator.geolocation. This object possesses a method watchPosition which receives a callback and notifies periodically the position of the user in an object implementing

the `Position` interface. It also allows to provide a callback for errors (since this method requires authorisation by the user of the web browser). When creating a `Geolocator` object, this will automatically check if the browser supports the `geolocation` features. If it does, `geolocationAvailable` will be turned to true[9]. When calling `start`, the `Geolocator` will call the `watchPosition` method previously described and from then on, it will notify all the registered listeners each time a position update is received. Calling `stop` on the `Geolocator` will cause it to cancel the watch for the user position.

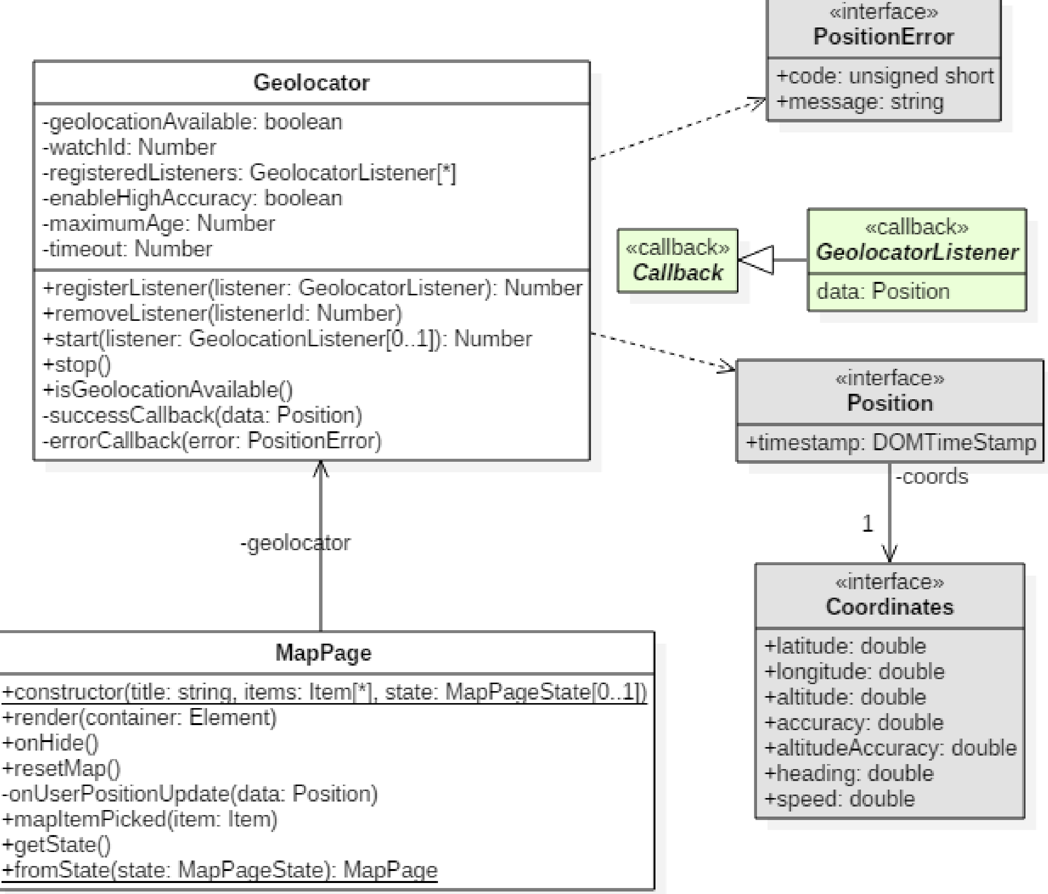


Figure 5.24: Class diagram of the `Geolocator`-related structure. Please, notice that the interfaces with a grey background are provided by the browser.

[9]Geolocation requires user authorisation and it is only allowed on web pages served over HTTPS.

Chapter 6

Implementation and testing

This chapter will describe the implementation process, avoiding excessive low-level details and explaining the aspects which were not obvious or maybe problematic. We recommend reading the chapter 5 before this one in order to have a general idea of the application design at a higher level of abstraction. We will describe also the initial data import very briefly and talk about the testing of the system.

We would like to point that all the code has been developed paying special attention to obtain clean, high-quality code. Moreover, we made wide use of commentaries and documented every class using the automatic documenters for the languages employed (PHPDoc and JsDoc).

6.1 The file structure of the project

The project structure fits with the one of the attached source code and consists of three main folders, each one of them dedicated to a different and independent part of the project. The `DB` folder gives us access to the MySQL scripts which allow to replicate the database of the project and make some initial insertions. It also contains the script which imports the initial data into the database. The `API` folder contains the source code of the back-end API, the structure inside this folder is explained in the section 6.2. The last one, `PWA`, contains the source code for the front-end, single-page, progressive-web application. We decided to organise the structure this way because these are three very isolated parts corresponding to different stages of the project. Some thoughts on making a third folder containing the static resources for the web were taken in consideration, since serving static resources from a different server is a common practice to keep the system performance and escalate adequately, but it was discarded by now because it would hinder the static pre-cache for the PWA, further explanation on this can be found in the section 6.3.

6.2 Back-end

The back-end API elaboration required the implementation of the database and the posterior construction of the PHP code. While this was a quite-smooth process in general, there are some details about the employed technologies and the decisions we made that are worth to mention in this section, given that they would not be represented in other parts of the document.

6.2.1 The database

Respect to the database, the implemented model corresponds directly to the described in the section 5.2.2. Two files were elaborated, a creation script (called `CREATION_SCRIPT.sql`), which allows to create the database structure completely and a insertion script (`INSERTION_SCRIPT.sql`) which inserts into the database the languages we are gonna use (English and Greek) and the categories for the application with the adequate order configuration to match the prototype developed by J. Ideami. This insertion script contains already the links for the help section indicated by the Greek team, they were added manually because it was more comfortable given the small number of *help* categories.

The only note on the database to be made is about checks. The first implementations on the database system made use of the MySQL `CHECK` constraint to make all the checks indicated in the relational model. To our surprise, none of the checks was being performed. After lots of small tweaks to the code we found the following in the MySQL official web page: "The CHECK clause is parsed but ignored by all storage engines".

This is a common source of errors in MySQL, not only the `CHECK` but some other clauses are silently ignored by the engines to maintain compatibility with SQL without implementing the functionality. Luckily, this has a simple workaround: we implemented all the checks with triggers on update and insert, and it does the trick perfectly.

About the constraints implemented, please notice we implemented all the ones the relational model had (checks, indexes, foreign keys...) and also one of those which the model could not represent: that a category with a `link` value cannot have any associated items. Still, we decided not to implement the constraint that requires the item opening hours to not overlap between them, because we consider this could be a too intensive trigger which could be easily replaced by a check on the control panel to input the data. For this reason we added the new requirement FR-15 for the control panel that indicates this. The initial data, given the way it is loaded, is ensured to have non-overlapping opening hours for the items.

The database creation script contains `IF NOT EXISTS` clauses, not supported by old versions of MariaDB (this was noticed while doing the deployment). Just remove them if this is the case when deploying. More information about deploying the database in the appendix A.1.

6.2.2 Initial data

To import the initial data in a comfortable way into the database, a Python script was created in order to read it from a *CSV* file directly into the database. To this purpose, the Greek team gathering the data was indicated a strict format they should follow while filling the spreadsheet. To use the script we need to save each page of the spreadsheet as a different CSV so we can have each item type in one single file. In the attached source code it is given as an example the initial data we received from the Greek team (which contained several mistakes) and the CSV files already prepared to import (and with the necessary corrections made). The script simply reads the files and makes use of the MySQL connector library to insert all the items with their languages and opening hours associations directly into the database. If it detects any failure in the data or if some MySQL clause fails, the script stops and the transaction is rolled back so no data at all is saved.

One tricky part of this was the opening hours. We decided to establish the following context-free grammar for this field on the document:

```
OpeningHours  →  Schedule (";" Schedule)*
Schedule      →  DayList " " HourRange ("&" HourRange)*
DayList       →  WeekDay ("," WeekDay)* | WeekDay "-" WeekDay
HourRange     →  Hour (":" Minutes)?  ("am"|"pm") "-"
                 Hour (":" Minutes)?  ("am"|"pm")
```

Where `Hour` and `Minutes` are successions of numeric digits, `WeekDay` is "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" or "Sunday"; "|" represents the union, "*" represents the Kleene closure and `A?` is equivalent to $(A|\lambda)$.

This grammar was, of course, explained to the data-gathering team in a less technical/formal way so they could understand it correctly. The kind of values we expect to find in this field are of the type “*Monday-Friday 9am-1:30pm & 4pm-8pm; Saturday 4pm-10pm*” (meaning the item is available from Monday to Friday, from 9am until 1:30pm and from 4pm until 8pm, and on Saturdays from 4pm until 10pm) or “*Monday,Wednesday,Friday 10:30am-2:30pm*” (meaning the item is available on Monday, Wednesday and Friday from 10:30am until 2:30pm). The script processes these schedules and inserts the right periods into the database. Alternatively, the field can contain `appointment(number)` to indicate that the item requires to be called to ask for an appointment and the phone number is `number`.

Respect to the images naming convention, we indicated the images names should correspond directly to the names of the items in lowercase, without spaces and with extension *jpg*. All the special characters not admitted in Windows files should be omitted, as well as the hyphens and the parenthesis. This way, the name of the icon image for an item with name “Citizens’ Service Centres (KEP).” should be “citizens’servicecentreskep.jpg”.

Unfortunately, the initial data received contained several mistakes: format errors, impossible hours, images with incorrect names so they could not be found by the system... All of this had to be solved manually and several hours of work (days, in fact) were lost in the process.

6.2.3 The API

To develop the API, we decided to use the folder structure shown in the figure 6.1. As we can see, we have a `public_html` folder, which should be set as the *document root* of our API server. This way, we get all the files in `class` or `config` to be unreachable through direct URLs, in order to avoid giving public access to them by mistake.

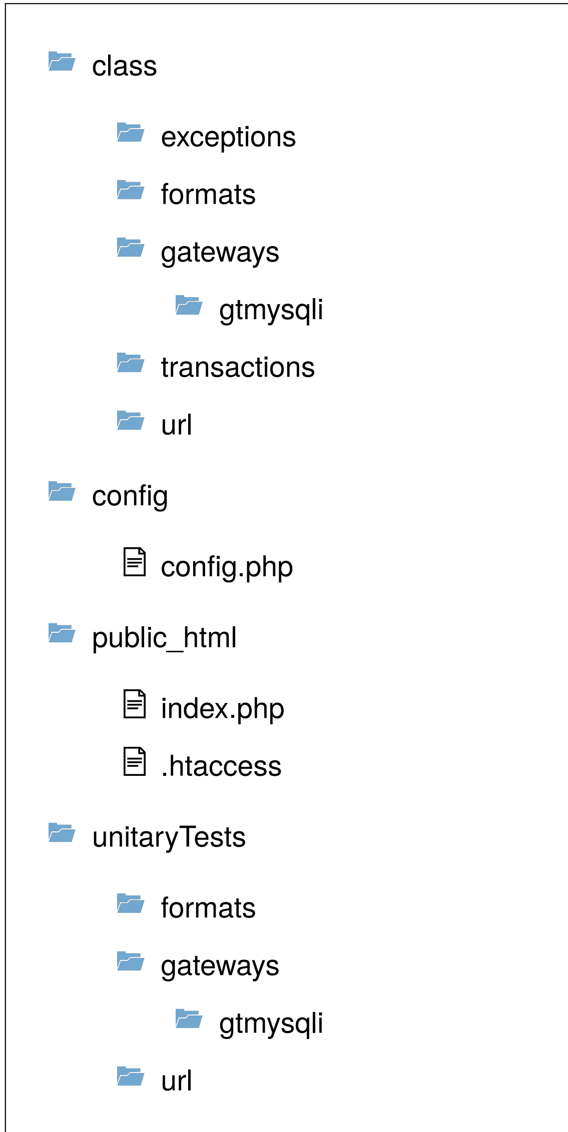


Figure 6.1: File structure of the API source code. Only relevant files are shown.

6.2.4 Auto-loading the classes

The PHP namespaces have been configured following the folder structure to facilitate class loading. When using object-oriented PHP, we like to have each class in a separated file, named with the name of the class, as it is usual in object-oriented environments. But PHP used to require to manually include the files with `include`[1], `include_once`[2], `require`[3] or `require_once`[4] sentences, so one of the most annoying problems when writing the files is having to write long lists of these sentences to make sure of including all the related classes. To work around this PHP 5.0 introduced the class auto-loading.

With auto-loading, when a used class is not defined, PHP will automatically try to find and include the file for it. Unfortunately, the default auto-load has a weird behaviour which is not even stable between systems (over XAMPP default configuration in Windows it searched for files preserving the original casing of the word, while when deploying it to our Linux server it searched for names in lower case). However, to solve this, PHP provides the option to change the function which automatically handles the auto-loading. Our version of the auto-loading is written in the file `config.php`, visible in the figure 6.1. The code which registers our auto-loader in particular goes like the following:

```
1  spl_autoload_register(function ($class) {
2      $paths = explode(PATH_SEPARATOR, get_include_path());
3      foreach($paths as $includePath) {
4          $file = $includePath . DIRECTORY_SEPARATOR
5              . str_replace('\\', DIRECTORY_SEPARATOR, $class)
6              . '.php';
7          if (file_exists($file)) {
8              require $file;
9              return true;
10         }
11     }
12     return false;
13 });
```

As we can see, our register simply loops over the registered include paths and searches for a file with the same name as the class (preserving the upper case letters) in a route of folders determined by the *namespaces* (notice the replacement of \ by the directory separator on line 5). Notice also that we make `require`, and not `require_once` in the line 8 for efficiency reasons, since this function will only be called whenever the class does not exist. Since all our classes will be in the `class` directory, we must make sure to include that folder in the include path (it is done previously, in the same file). Once this is done a class whose name is `gateways\gtmysqli\MysqliItemsGateway` would cause the inclusion of the file

[1] *http://php.net/manual/en/function.include.php*
[2] *http://php.net/manual/en/function.include-once.php*
[3] *http://php.net/manual/en/function.require.php*
[4] *http://php.net/manual/en/function.require-once.php*

class/gateways/gtmysqli/MysqliItemsGateway.php when it is used and it was not already defined.

Finally, we simply need to indicate in index.php to require the config.php file and the system starts working.

6.2.5 Redirection

To make the API work we need all the requests made to any folder under the API folder to be redirected to our index.php. Although this should be configured in the server in use, to provide a quick alternative we add to the source code an .htaccess file, which is an Apache configuration file, configuring the RewriteEngine to rewrite all the URLs under it and direct them to our *index* file.

We have to be careful when redirecting other links into the API, making sure the server will continue rewriting the links recursively, in another case the API will stop working. Also keep in mind this redirection when trying to add new php files to the folder where the API is published. In general, we do not recommend to do this, we suggest to deploy the API into its own folder, with no other files to be accessed there. If sharing the folder were mandatory, just introduce a new RewriteCond rule to the .htaccess to avoid rewriting the file you want to allow the access to. For example, the rule RewriteCond %{REQUEST_URI} !/my_awesome_file.php$ would prevent the rewrite engine from rewriting the URLs pointing to my_awesome_file.php.

6.2.6 Other implementation details

Since PHP does not provide enumerations, we made a workaround implementing a FakeEnum class which simply stores a value and can be extended, adding static methods for each of the enumeration values. As a bonus, we can now create the enumeration values from strings without long switch-case constructions, which results very useful when parsing data from the database, as an example take a look at the ItemType implementation:

```
1  class ItemType extends FakeEnum {
2      protected function __construct($value) {
3          parent::__construct($value);
4      }
5
6      public static function SERVICE() {
7          return new self('service');
8      }
9      public static function LEISURE() {
10         return new self('leisure');
11     }
12     /* [More similar methods...] */
13
14     /**
15      * Creates the right item type for the given string,
```

```
16      * the strings in this class should match the ones
17      * in the database, so we can pass them directly into here.
18      * @param $string string The string to check.
19      * @return ItemType the item type with the given string as value.
20      * @throws UnknownTypeStrException if the given type does
21      *     not match any of the types known by the current
22      *     implementation of this class.
23      */
24     public static function FOR_STR($string) {
25         if(in_array($string, array('service', 'leisure',
26                 'link', 'help', 'info'))) {
27             return new self($string);
28         } else {
29             throw new UnknownTypeStrException(
30                 "Tried to create unknown item type '$string'.");
31         }
32     }
33 }
```

We also added the `IApiInterface` class which contains constants relative to the interface of the API, like the strings used for each field, the base URI where the API is placed, the default input and output formats, etc.

6.2.7 Unitary tests

To test the functionality of the API several unitary tests were developed along with the code. These tests are also useful to check everything is fine when we add new functionality to the code. All of them are placed in the `unitaryTests` folder[5] and their structure is similar to the one of the source code files. To develop the tests PHPUnit was used, concretely the version 7.1.5. PHPUnit provides a `TestCase` class which we can extend to perform all the necessary tests.

One little problem was to make the unitary tests for the `mysqli` gateways, which we considered very fundamental. To make them, we needed to make an stub replacement for the real `mysqli` object, but the functionality provided by PHPUnit to *mock* classes was not enough to make the stub work. The main problem with it was that PHPUnit-generated *mocks* are not able to manage references in their methods, so we could not emulate correctly the behaviour of the `mysqli_stmt` class[6].

When we desire to save the results of a `mysqli_stmt`, we execute the statement and make a call to `mysqli_stmt::bind_result`, passing by reference all the variables we want to make the bind with. After this is done, we call to `mysqli_stmt::fetch` to save the values of the next row returned by the query in the binded variables. Unfortunately, PHPUnit-generated *mocks*

[5]When executing all the unitary tests, make sure your working directory is set to this folder, or some tests may not be able to import the required mocks.

[6]The `mysqli_stmt` class is the class which encapsulates all the functionality for the prepared statements in `mysqli`.

cannot receive parameters by reference, since they make a copy of the parameters in the process of mocking up the class, loosing the original addresses. To work around this we finally made our own extension of `mysqli_stmt` overriding the methods we needed.

However, the replacement is still a bit uncomfortable to work with. Our `mysqli` mock needs to know which exact queries it expects to receive to know which *prepared_stmt* mock generate, and this cause all the tests to break when minor changes are made in the queries (e.g. when changing the order of the fields).

6.3 Front-end

The front end of the system, the PWA, had a more complicated implementation than the API conditioned by the development environment. This development environment was set up using *npm* and *gulp*.

6.3.1 File structure

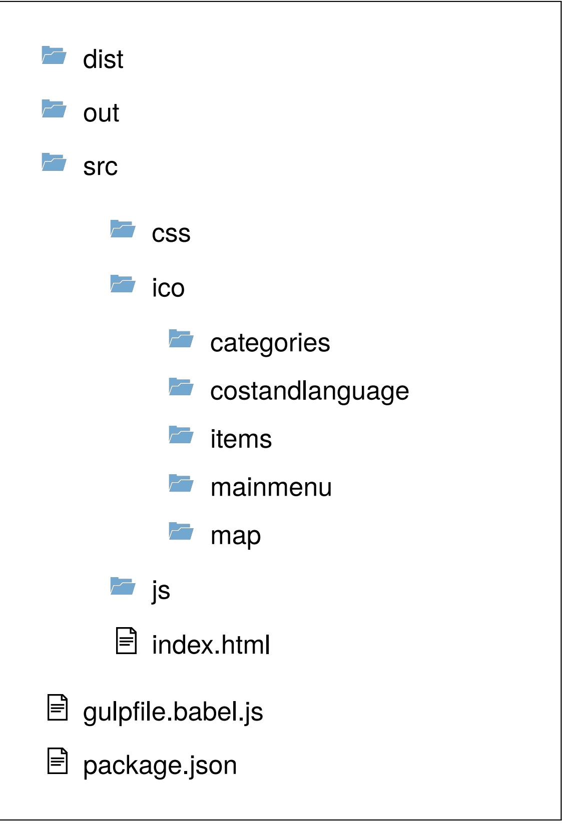


Figure 6.2: File structure of the PWA source code. Only relevant files are shown.

The main folder of the PWA contains the configuration files for the project. The `src` folder contains all the code and resources to build the application, while the `dist` and `out` folders are generated when building the application and when generating the JsDoc, respectively. For information about how to build the application check the section A.5.

The main files to notice of this structure are `package.json`, `gulfile.babel.js` and the `index.html`. The first one is our "configuration" file for `npm`, the more important function in our case is to store the development libraries we use to build the application so it is really easy to prepare the environment in any computer. The `gulpfile.babel.js` is the configuration file for `gulp`. In this file we write all the tasks we want gulp to configure and how they behave, C developers can think about this like some kind of *Makefile* for web pages and written in JavaScript. Finally, we say our `index.html` is one of the fundamental files of the structure because there is where everything starts. A PWA is, by definition, a single page application (SPA), that means all the web is accessible through a single "page", there is no navigation between screens and all the data transferring happens in the background (check section 2.1 for a brief introduction to modern SPAs behaviour accompanied by explanatory figures).

It is worth to point also that once our web is built, all the JavaScript is concatenated and minified into a single script, and the same happens with the CSS. This speeds up the page loading when accessing the page for the first time.

6.3.2 Precache and dynamic caching

To accomplish the functional requirement FR-13 and the non-functional requirement NFR-21, we need the application to keep at least the fundamental data available even when the phone has no connection. To achieve this we need to use the cache.

Modern JavaScript includes the concept of service workers. A service worker is just a JavaScript code which runs in the background, asynchronous from the main code of the page. This service worker is totally independent and can communicate with the main code through messages. An interesting characteristic of the service workers is that they are able to listen to http requests to files under their scope, and intercept them. The scope of a service worker includes all the files under the folder the service worker is placed, and it can never be placed in a more general folder than the folder where the service worker is placed. To give an example, if we want to request a service worker from the file whose URL is *`https://mydomain.com/my/folder/index.html`*, our service worker needs to be placed inside the same folder or in a deeper one, it could be for example in *`https://mydomain.com/my/folder/foo/bar/service-worker.js`* but not in *`https://mydomain.com/service-worker.js`*. If we place the service worker in *`https://mydomain.com/my/folder/service-worker.js`*, its scope would be every request pointing to files inside *`https://mydomain.com/my/folder/`* or a deeper one. This is very important when creating re-directions for the application, as we will see in the section 6.3.5. Apart from intercepting the HTTP request under their scope, modern JavaScript offers the possibility to the service workers to manipulate the cache to save or load requests directly. This

is a powerful tool which gives us the key to keep our application working offline and, at the same time, speed up the page loading[7].

Of course, we could implement ourselves the cache functionality and, in fact, it is not really complicated, but, good for us, Google has created a completely-free, Apache 2.0-licensed library to help us with this and make our service worker to cache the application really quick to configure and more powerful. The library we use is called `sw-precache` and allows us to create an static pre-cache of the application shell and a dynamic cache for the API requests in question of minutes (once we have certain domain over its configuration).

The pre-cache allows our site to directly cache the shell, the "skeleton" of our application in the very moment we access it the first time. We can think about this like a usual application installation. When building our PWA (check the section A.5), `sw-precache` will check all the files we indicated and create for them a hash that it will store in the service worker it generates. If some of the hashes has changed, the service worker will change in consequence and the next time users enter our web page it will update in the background. Once it is updated, it will cache again the files which have changed (and only those ones). This way we reduce the data consumption drastically. Moreover, we speed up the page loading since every time we load the page after the first loading, the data will be loaded from cache and not from the network. The files we decided to pre-cache can be checked in the `gulpfile.babel.js` file[8], and they are the `index.html`, the minified JavaScript, the minified CSS, the icons for the home screen and the categories and the favicon. The figure 6.3 shows an example of hashed files in a service worker generated by `sw-precache`.

```
var precacheConfig = [[
    "/alpha/public_html/css/style.min.css", "
       dc31e8dc0a2e1ff9600f3a04b64e8e40",
    ["/alpha/public_html/favicon.ico","324865
       c614d2845fa261c27ece9a1a46"],
    ["/alpha/public_html/ico/categories/help_banka.png","821
       e1a79a29f76ca5fec148e88c09f8c"],
    ["/alpha/public_html/ico/categories/help_disab.png","1
       dc50ab348ea89ffbd55d99d86b8e813"],
    /*...*/]];
```

Figure 6.3: Example of hashed files in a generated service worker.

But the shell of our application is nothing without the dynamic data. With dynamic data

[7]Both the service workers and the cache functionality require HTTPS to work. The browser will not allow service workers on webs served over plain HTTP. However, they do allow them on `localhost` to facilitate the development process. Our application, of course, has been deployed to a server with HTTPS.

[8]The configuration details for `sw-precache` are too long and out of the scope of this document. However, the information is available in the GitHub page of the project *https://github.com/GoogleChromeLabs/sw-precache*.

we refer here to all the information we obtain from the API of our system. The same library `sw-precache`, making implicit use of another library developed by Google called `sw-toolbox`, allows us to create another cache for the dynamic content. There are several configurations that can be done to this dynamic cache[9], but the most important and the one we will discuss here is the *handler*. The handler defines the politics the cache follows to decide how to respond to the HTTP requests it intercepts. Notice that the pre-cache is always executed the first time we enter the page and every time the content is updated, but this dynamic caching intercepts the request to our API to decide whether to cache them or not, and whether to serve the cached version or wait for the network answer.

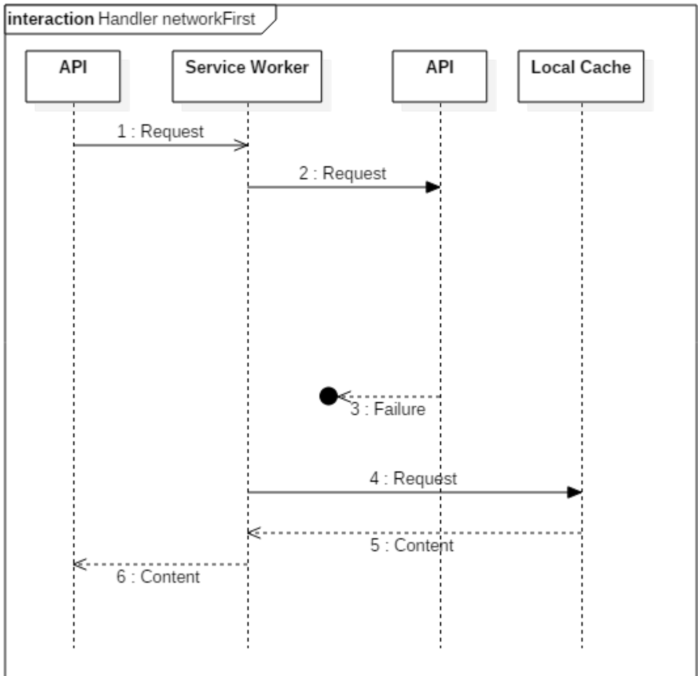


Figure 6.4: Sequence diagram illustrating the interaction between the PWA, the service worker, the API and the cache when the `networkFirst` handler is used. In this example the API is not available for some reason and the service worker sends the request to the cache.

There are five built-in handlers in the library, corresponding to the most common network strategies: `networkFirst`, `cacheFirst`, `fastest`, `cacheOnly` and `networkOnly`. The

[9]For more information on configuring the dynamic cache check *https://googlechromelabs.github.io/sw-toolbox/*, check also *https://github.com/GoogleChromeLabs/sw-precache* for more information about how the `sw-toolbox` library configuration is integrated into the `pre-cache` library.

last two of them correspond to extreme cases with very specific uses (one never uses the network and the other never uses the cache), so we will centre ourselves in the three first ones.

The `networkFirst` handler tries to answer the requests fetching them from the network, if the request succeeds the received data is stored into the cache. When the request fails the fulfils the request serving from the cache. This way, the application shows always the freshest data while keeping cached data to supply when the network is not available. A downside of this comes with very slow networks, since the service worker will wait for the network connection to fail and, in consequence, loosing time waiting for the unavoidable to happen. This, however, can be palliated with a timeout configuration for `sw-toolbox` which will cause the service worker to give the request by lost when the timeout is exceeded. The diagram in the figure 6.4 illustrates the case where the network fails to answer the request.

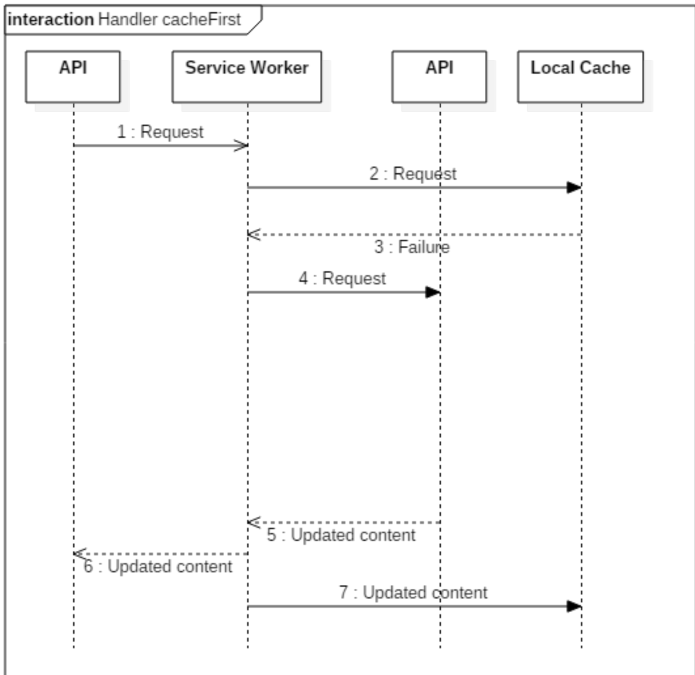


Figure 6.5: Sequence diagram illustrating the interaction between the PWA, the service worker, the API and the cache when the `cacheFirst` handler is used. In this example the cache does not have the requested file, so the service worker sends the request to the API.

The `cacheFirst` handler performs exactly the opposite behaviour. It request always the information to the cache first, if the request matches a cache entry, it responds directly with that. Otherwise, the request is tried to fetch from network. In case the network succeed, the cache will be updated with the new information. With this handler, the application will keep serving the same

data until it gets lost for some reason (the browser can delete the cache always that it considers it needs the space). This could be useful for very static resources that will never change, and it allows to save both network usage and battery avoiding unnecessary networks requests at the same time it reduces drastically the loading times. It could be fine for our categories requests, since we do not expect them to change, but it is definitely not a good strategy for our items. The diagram in the figure 6.5 illustrates the case where the cache fails to answer the request.

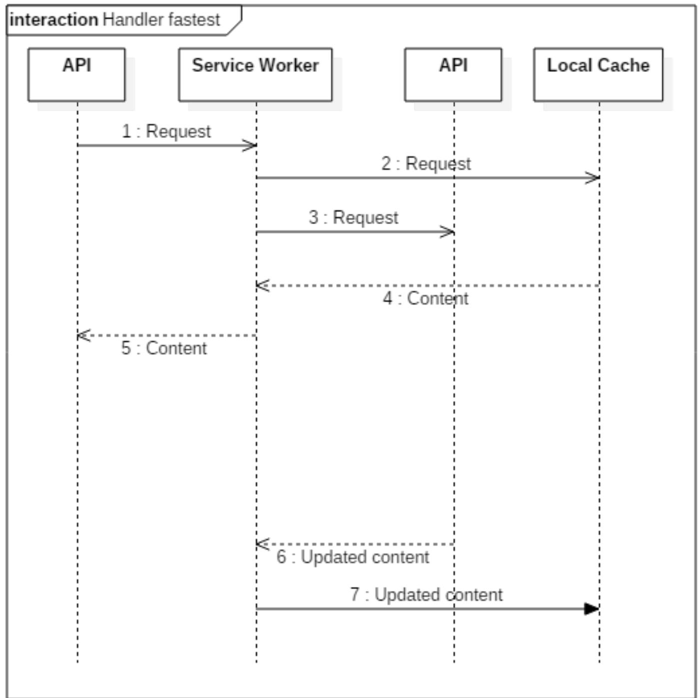


Figure 6.6: Sequence diagram illustrating the interaction between the PWA, the service worker, the API and the cache when the `fastest` handler is used. In this example the cache has the requested file and answers faster than the API.

The `fastest` handler is a mix between the previous two, giving us the flexibility to keep an updated cache while fulfilling the requests really fast. When a request is received, the handler request both the cache and the network in parallel. Whichever of them is the first to answer, it will be one used to respond the query. Of course, when the entry is already in the cache the usual answer will come from there but, still, when the network answer is received the cache will be updated, so next time we enter the page the information will be already updated. This way, we get always a network request, even when the resource is cached. The diagram in the figure 6.6 illustrates the case where the cache answers the request first.

Since we would like to keep our items updated, but the changes to them will be usually small and not so common, we decide to use the `fastest` handler. Of course, the library allows us to

configure different handlers for different URL patterns, so we could use a `cacheFirst` strategy for the categories requests and a `fastest` or even `nerworkFirst` strategy for the items, but due to the small amount of time available we decide to keep it simple and use only one cache handler for the whole API by now. In the beta or the final versions this may change or maybe we will even use a customised strategy.

6.3.3 Grouping periods

While in the database we save the opening hours of the items in a period-by-period format, we would like to display this to the user in a more human-readable format. For example, imagine we have the following opening hours for a given item:

Start day	End day	Start hour	End hour
Monday	Monday	10:00	14:00
Tuesday	Tuesday	10:00	14:00
Wednesday	Wednesday	10:00	14:00
Friday	Friday	10:00	14:00
Saturday	Saturday	10:00	14:00
Sunday	Sunday	14:00	19:00

Being this the case, we would like to group all the periods of successive days with the same starting and finishing hour and display them in a way similar to this one:
Monday-Wednesday 10:00-14:00
Friday-Saturday 10:00-14:00
Sunday 14:00-19:00

To achieve this, we need to scan the periods grouping the ones which are successive and have the same starting and finishing hour. The algorithm we developed behaves as shown in the pseudo-code in the figure 6.7.

The precondition for the algorithm is that the periods are sorted ascendant by their ending and they do not overlap between them. This is not a problem because we simply configure the API `IPeriodsGateway` to return them ordered by their ending and they are ensured to not overlap by the FR-15 because, apart from the initial data, all the other data will be introduced through the control panel.

We start with an empty list of schedules (this is the name we will give to those “groups” of periods in the algorithm). While we have periods left to classify we repeat the loop from line 5 until line 36. Inside this loop, we pop the first element of the remaining periods (notice it will be the one which ends first, and since they do not overlap, also the one which starts first). We create with this element a new schedule which starts and ends in the same period, and save also the information about what is the day we expect the next period to start (*next_day*. The day we expect the new period to start depends on the hours of the period. If the the period ending

```
1  # PRECONDITION: periods is sorted ascendant by end_day, end_hour and
       ↪ end_minutes and they do not overlap
2  function groupPeriods(periods):
3      schedules := []
4      # continue while there are periods left to group
5      while count(periods) > 0:
6          next = pop_first(periods)
7          new_schedule = {start_period: next, end_period: next, next_day:
               ↪ next_day(next)}
8          found_next := false
9          repeat:
10             found_next = false
11             # expand while some new is found
12             for each period in periods:
13                 if hours(period) = hours(next) and start_day(period) =
                       ↪ new_schedule.next_day then:
14                     new_schedule.end_period = period
15                     new_schedule.next_day = next_day(period)
16                     found_next = true
17                     break
18                 end if
19             end for
20         while found_next
21
22         # check if this is the previous to another one and join them
23         found_next = false
24         for each schedule in schedules:
25             if hours(schedule.start_period) = hours(new_schedule.
                   ↪ start_period) and start_day(schedule) = new_schedule.
                   ↪ next_day
26                 schedule.start_period = new_schedule.start_period
27                 found_next = true
28                 break
29             end if
30         end for
31
32         # it was not, create a new one
33         if not found_next then:
34             schedules.append(new_schedule)
35         end if
36     end while
37     return schedules
38 end function
```

Figure 6.7: Pseudo-code for the period-grouping algorithm.

hour (and minutes) is later than the starting hour, the day we expect the following period to start in order for both to be a continuous schedule is the day just after the last day of the period. So, for example, if the period is `Monday 9:00 - Tuesday 22:00`, our continuation day would be *Wednesday*, since we cannot start at 9:00 on the same Tuesday if we end at 22:00. If the last day is Sunday, the day after would be, of course, Monday. If the period ending hour is sooner than the starting hour, then the next day will be the same day the period ends. This way, if hour

period is `Monday 17:00 - Tuesday 3:00`, we expect the following period to start on the same Tuesday (at 17:00). When the period has the shape `Monday 17:00 - Monday 3:00`, it might seem the algorithm behaviour is mistaken and that the next day should be the next Tuesday and not the same Monday. However, notice that this period does not mean from Monday at 17:00 until Tuesday at 3:00, this period means from Monday at 17:00 all the week around until the next Monday at 3:00, so our algorithm expects the right day.

The loop from line 9 to line 20 simply expands the created schedule each time it finds the next period to continue the schedule. Notice that, since we take always the first and they are chronologically ordered and non overlapping, we will never obtain broken schedules inside the same week. Imagine the case where the periods were not ordered, as following:

Period id	Start day	End day	Start hour	End hour
1	Wednesday	Wednesday	15:00	20:00
2	Monday	Monday	15:00	20:00
3	Tuesday	Tuesday	15:00	20:00
4	Friday	Friday	15:00	20:00
5	Thursday	Thursday	15:00	20:00
6	Saturday	Saturday	15:00	20:00

In this case, we would take first the period 1 ending in Wednesday and expand the schedule adding the periods 5, 4, 6 to be the schedule `Wednesday - Saturday 15:00 - 20:00`. Then we would take the new first, which would be the period 2 and make a new schedule expanding to the period 3, having at the end two periods `Monday - Tuesday 15:00 - 20:00` and `Wednesday - Saturday 15:00 - 20:00`. Starting with them ordered chronologically, we obtain a single schedule `Monday - Saturday 15:00 - 20:00`, that is the correct result.

Finally, the last loop from 24 to 30 iterates over the schedules previously saved to check if some of them was the continuation for this new schedule. This is necessary because week days are modular, after the Sunday it comes the Monday again. When we order we set the Monday as the first element and the Sunday as the last one. So imagine a case like the following:

Period id	Start day	End day	Start hour	End hour
3	Monday	Monday	15:00	20:00
4	Tuesday	Tuesday	15:00	20:00
1	Saturday	Saturday	15:00	20:00
2	Sunday	Sunday	15:00	20:00

The periods are ordered, but if we execute the first part of the algorithm what we obtain is two schedules like the following: `Monday - Tuesday 15:00 - 20:00, Saturday - Sunday 15:00 - 20:00`. This is solved by the last loop since it would find the first schedule was the continuation of the one to add when going to add `Saturday - Sunday 15:00 - 20:00` and it would merge them in consequence.

Finally, if the schedule has not been merged, we simply insert it into the list and continue with the remaining periods.

The complexity of the algorithm, being *N* the number of periods to sort is, in the worst case:

$$\mathcal{O}(\text{loop}_{5-36}) \times (\mathcal{O}(\text{loop}_{9-20}) \times \mathcal{O}(\text{loop}_{12-19}) + \mathcal{O}(\text{loop}_{24-30})) =$$

$$O(N) \times \left(O\left(\frac{N}{2}\right) \times O\left(\frac{N}{2}\right) + O\left(\frac{N}{2}\right)\right) = O(N) \times \left(O\left(\frac{N^2}{4}\right) + O\left(\frac{N}{2}\right)\right) =$$

$$O(N) \times O\left(\frac{N^2}{4}\right) = O\left(\frac{N^3}{4}\right) = O(N^3)$$

The internal loops are calculatedd as $O\left(\frac{N}{2}\right)$ because each time they execute there is a period less (so, in average in the worst case they loop over $\frac{N}{2}$ periods).

6.3.4 Saving the application to the home screen

The non-functional requirement 21 requires the application to be installable to work like a native application would do in the systems which allow this functionality. To achieve this, we need to create a manifest file.

The manifest of our PWA is simply a JSON file which contains some information for the browser about what we expect it to do with our web page. There we define a name for our application, a theme color, the launch icon, the initial URL, the scope and some other details. For the web browser to offer the user the addition of our PWA to the home screen, Google Developers[10] indicates the following criteria[11]:

- The web app is not already installed.
- The page meets a user engagement heuristic (at the moment of writing these lines and in Google Chrome this heuristic is that the user has interacted with the domain for at least 30 seconds).
- It is loaded valid web app manifest which includes:
  - The short name or the name of the PWA.
  - Icons for the launcher, at least the 192 px and the 512 px sized ones.
  - An start URL which indicates the initial page to load when launching the app.
  - A display value of `fullscreen`, `standalone` or `minimal-ui` (the display value indicates how we want our application to be displayed to the user when launched from the home screen).

[10] *https://developers.google.com/web/fundamentals/app-install-banners/*

[11] The criteria varies a bit from one browser to another, but at the moment of writing these lines and after checking the Edge, Firefox, Opera and Samsung Internet sites we confirmed that the criteria imposed by Chrome is the most restrictive.

- The page is served over HTTPS.
- The web page has registered a service worker with a `fetch` even handler (i.e., the page is caching the requests to the network to provide offline functionality).

When this criteria is met and the browser is disposed to allow the addition of our PWA to the home screen of the user, it triggers a `beforeinstallprompt` event, giving us the chance to cancel the prompt of the banner to the user or to save the promise to display it later.

6.3.5 Deployment

In our deployment of the application, since this was an alpha version, we wanted to public the application in the `/alpha/public_html/` folder of our domain. But to give a quick access through a nice URL to the focus group in Athens J. Ideami wanted to redirect the requests to `/a` (and only the requests to `/a`) to this location. This came with a couple of problems.

The obvious solution would be to create a redirection that causes all the URLs of the format `/a/`*`something`* to be redirected to `/alpha/public_html/`*`something`* and explicitly tell the browser to redirect `/a` to `/a/`, but at the start of the development J. Ideami preferred not to give me access to the root of the server, and since I do not have administration permissions (all I had was the FTP to the `/alpha/` folder) I could not change the redirection he had made. This was a problem because when loading `/a`, all the relative URLs to scripts, style sheets and images were understood by the browser to be meant for `/`. For example, requesting from the HTML something like `<script src="js/script.js">` led the browser to request `/js/script.js` and the server to answer with a 404 status (not found).

So the workaround was to include a base URL for the web page, which can be configured before building and is added to every request in the application (it was not hard to do, given that we had already the `ResourcesProvider` described in the section 5.4.2). This worked fine because it allowed us to establish an absolute base URL like `/alpha/public_html/` and it would work without any problem.

Unluckily, a new problem came when trying to implement the cache and work with the service worker. As explained in section 6.3.2, the service worker has a defined scope for the requests it affects and these are all the requests under the folder the service worker is placed. In consequence, since the service worker was requested to `/alpha/public_html/cache-service-worker.js` it was not possible for it to cache the request to `/a`, i.e., the request of the HTML of our page, and the offline mode did not work at all from the redirection point. To solve this, J. Ideami finally ended accepting to give me access to the root of the server, where I configured another redirection. I configured Apache to redirect every request to `/cache-service-worker.js` to `/alpha/public_html/cache-service-worker.js`, and simply programmed the page to load the service worker from a relative URL, instead of using our application base URL. However, this did not work at all and the service worker request was continuously answered with a 404.

It took some time to realise what was the problem: the NGINX static-resources server. The server of deployment uses a very common configuration to improve the efficiency of the system: it serves the dynamic files with Apache but introduces NGINX as an intermediary proxy which automatically serves static files (such as `.js`). The consequence of this was that our request to `/cache-service-worker.js` would never get to Apache for them to be redirected. NGINX processed the request first and found there was no such file in that directory, returning a 404 answer.

The final solution was to define the same URL rewrite in both the root folder of the server and the `/alpha/public_html/` folder. Both of them redirecting the requests to `worker` to `/alpha/public_html/cache-service-worker.js`. After this, I programmed the page to request the service worker to the URL `worker`. Since this URL does not contain any of the extensions managed by NGINX (in fact, it does not contain any extension at all), the request gets intact to Apache, which in both cases serves `/alpha/public_html/cache-service-worker.js`, inducing the browser which requested `/worker` to think the service worker is, in fact, in the same folder as the HTML of the page and, in consequence, being able to cache correctly our application shell.

6.4 General testing

Although some general testing was planned in the initial planning, due to some complications and delays debt, mainly, to the several mistakes in the initial data document and the very tight schedules, this general testing has not been possible to automate yet. Some automatic testing will be developed in the following days after presenting this document.

However, the API counts with several unitary tests and J. Ideami is checking the functionality of the PWA screen by screen to make sure everything is displayed and works as he designed and expects. Moreover, we count with a focus group which will test the alpha version of the application for the whole next month in Greece, so this does not suppose a problem for the evolution of the project.

Chapter 7

Conclusions and future work

In this chapter we will analyse which requirements have been satisfied already, what requirements are left, which improvements could be made to the already implemented system and what is our plan of future work over the project for the coming months.

7.1 Satisfied requirements

The application functional requirements specified for the alpha version of the project, as stated in the section 3.8 are the requirements FR-1, FR-2, FR-7, FR-8, FR-9, FR-10, FR-11, FR-12, FR-13, FR-14. All of these requirements have been, in principle, satisfied, but we have to wait still for the results of the tests the next month to be completely sure. With this requirements, the use cases UC1, UC2, UC3, UC4, UC5 and UC6 shall be completed.

The functional requirements FR-3, FR-4, FR-5, FR-6 and FR-15 are still missing and should be accomplished with the beta release of the application at the end of the coming month. Completing the implementation for the use cases UC7, UC8, UC9 and UC10.

The current status of all the non-functional requirements is shown in the table 7.1.

7.2 Improvements to be made

The already implemented functionality has some details which can and should be improved. Here we list some of the improvements we noticed and have in mind.

- The `ItemsMap` class should be redesigned, dividing its functionality among several more-specialised classes. It is clearly a too-complex class which performs too many different actions. Its functionality will be divided between different components collaborating together so the code is more changeable and maintainable.

Requi.	Impl.	Satisf.	Req.	Impl.	Satisf.
NFR-1	✓	Not tested	NFR-2	✓	✓
NFR-3	✓	Not tested	NFR-4	✓	✓
NFR-5	✓	✓	NFR-6	✓	Not tested
NFR-7	X	Not tested	NFR-8	✓	Not tested
NFR-9	✓	✓	NFR-10	X	X
NFR-11	✓	Not tested	NFR-12	✓	✓
NFR-13	✓	✓	NFR-14	✓	✓
NFR-15	✓	✓	NFR-16	✓	✓
NFR-17	✓	Not tested	NFR-18	✓	Not tested
NFR-19	✓	✓	NFR-20	✓	✓
NFR-21	✓	✓	NFR-22	✓	✓
NFR-23	✓	✓	NFR-24	✓	Not tested
NFR-25	X	X	NFR-26	✓	✓
NFR-27	✓	✓	NFR-28	✓	✓
NFR-29	✓	✓	NFR-30	X	Not tested
NFR-31	X	Not tested	NFR-32	X	Not tested
NFR-33	X	Not tested	NFR-34	✓	✓
NFR-35	X	X			

Table 7.1: Current status of the non-functional requirements of the application. The three columns mean: *requirement*, *implemented* and *satisfied*.

- The maps still need to show the direction the user is looking at. This is something really needed because it is very useful to get the right orientations in the map. We need to set up a `google.maps.Symbol` with some kind of arrow as the user marker icon so we are able to rotate it over time. Then, we need to implement an algorithm to combine the information about the speed given by the `Geolocation` interface, with the difference between the previous and the new position and the information from the magnetometer and the gyro of the phone. This way we would be able to keep the right orientation for the marker.

- It is still needed more control over the side-cases of the geo-location and other features. We still need to display to the user certain error notifications. At this time, the system silently ignores cases like the geo-location being denied by the user (it simply does not display the position, but it does not show any message) or the Google Maps API returning a request limit error.

- The JavaScript features to check whether the page is visible (on top of the other tabs) or not could be used to stop the geo-locator and this way save battery.

- The URLs should be correctly updated so a user can share the URL with another user and they both will see the same page inside the application. Right now the URLs always add the hash string `#not_implemented_yet`, no matter the page we are visualising, so when we share the URL with another user, this other user is taken to the home page.
- The dynamic catcher could use a hybrid, personalised handler which sends a message to update the GUI when the answer from the network is received, without waiting for the next access to the page. This should not be too complicated to implement and it would allow for fresher information on the page.
- Grouping periods can be improved so it does not search further than the next day. Right now it continues searching until the end of the periods, but since we know the periods are sorted chronologically, we do not need to keep searching when we overpass the day we want to find. This would improve the efficiency of the algorithm in the average case.

7.3 Future work

The development of the project will continue following the planning established. The beta release will be published the first day of September and the final release will be the first day of October. The coming month the development team will personally go to Greece to interact with the focus group in person, improving the feedback.

In a more long term, we expect the application to scale and reach to cover many cities from maybe other countries in Europe. However, this depends on the external financing and the results of this attempt. From a more local point of view, the application maybe could be amplified in scope to cover more real-time data like events in the city or other interesting kinds of information which it does not manage at the moment.

Appendix A

Technical manuals

In this appendix we include some manuals for the technical tasks that could be performed over the system.

A.1 Database deployment

To deploy the database it is needed some DBMS able to manage MySQL databases. The code has been checked during development using MariaDB 10.1.19. To get the MySQL database structure ready in your system simply create an empty database and, with the database selected, run the commands contained in the attached source code in the file *DB/SQL scripts/CREATION_SCRIPT.sql*. It is possible, if you are using older versions of MariaDB or another DBMS, that you receive an error message caused by the `IF NOT EXISTS` of the `CREATE TABLE`s. If this is the case, simply remove all the `IF NOT EXISTS` and run the commands again. Everything should be fine. Make sure your DBMS has the `TRIGGER` functionality available and running to keep the database integrity.

If you want to insert the initial data we supply, you will need to run also the commands contained in *DB/SQL scripts/INSERTION_SCRIPT.sql*. This will introduce the English and Greek languages and the categories of the application into the database. This step is required before executing the data-import script.

A.2 Importing the initial data

To import the initial data supplied by the Greek team (and already corrected), we supply a simple Python script which can be run from any terminal with Python installed. The script is designed for Python 3.0 or later and makes use of the `mysql.connector` library for Python. You can find this script in `DB/Python scripts/data_loading.py`. If you are not sure of the version of Python you have installed or if you have the required library, simply run the script and it will

detect it and notify you if this is not the case. It will also provide you with a link to download the `mysql.connector` if it is not able to find it.

Before running the script, make sure your database is created (see section A.1), running and accessible from your system. Run the script as any other python script (typing `python ./data_loading.py`). The script will ask you for the information necessary to perform the connection to your database: the host, the user, the password and the name of the database. Once you provided this information you should see a message saying “Connected”. The program will ask, one by one, for the names of the files for each type of item. The files we provide have the following names, in the order they are asked by the script: `info.csv`, `leisure.csv`, `links.csv` and `services.csv`.

A.3 API deployment

To deploy the API you will only need to make a couple of configurations. First, copy all the files inside the `class`, `config` and `public_html` directories into your deployment server. Make sure the public_html folder of the API is the only publicly accessible folder of the API through an URL. One easy way to achieve this is by setting the document root of your server directly to the public_html folder, but you can do it the way you prefer. Once you have all the files in place, go to the file `class/gateways/gtmysqli/MysqliGatewayFactory.php`. In the constructor of this file you can configure the values for the `mysqli` connection to your database.

Let’s suppose now that you need to serve your API from a sub-folder of your server, for example because you have your web in the root folder and so you need it to be served from the `api` sub-folder. To do this, you only need to move the contents of `public_html` into the `api` folder and, once there, open the `index.php` file. At the start of this file you can see two lines like the following:

```
// The config file configures the autoloaders for the classes
require('../config/config.php');
```

Edit the `require` to make sure it includes correctly the `config.php` file. In our case we need to set it to be like this: `require('../../config/config.php');`. Great!

If we try to query one of the URLs of the API now, for example to get the categories for the item type *leisure* (http://yourhost.com/api/item-types/leisure/categories), we obtain the following answer:

```
{
"error": "The␣URL␣('/api/item-types/leisure/categories')␣and␣method␣
    provided␣do␣not␣lead␣to␣any␣valid␣resource."
}
```

This is because we have not configured the base URL of our API yet, so the `UrlMatcher` is trying to directly match our URL (including the ’/api’) with the set of registered patterns, and since

none of them contains `api` at the start, none of them is giving a positive result. To configure this we need to edit the API interface, stored in the `class/IApiInterface.php` file.

The first constant of this file, `API_BASE_URI`, defines the base URI for the whole API. Set the value to `'api'` and perform the HTTP request again. Everything should be working fine now.

A.4 Adding new URLs to the API

Adding new URLs to the API is a very simple process. Once you have implemented and tested the `ITransaction` which you want to be ran when accessing the new URL. Go to the file `class/url/UrlMatcher.php`. In this file, look for the private class constructor at the end of the file. If you are adding new functionality for other HTTP methods in an already existent URL rather than adding a new URL, add the method and the transaction you want to associate to one of the existent `TransactionMap`. If you really want to introduce a new URL, instantiate a new `TransactionMap`, add the method and the transaction you want to configure and create a new `UrlPattern` in the `$this->urls` array, indicating the desired pattern for the URL.

The patterns for the `UrlPattern` are specified as strings and relative to the base URL of the API (configurable in the interface `IApiInterface`). The patterns may include parameters to be taken directly from the URL. To include a parameter, you only need to start it with ':', indicate a name for the parameter (only alphanumerical characters allowed) and a type between '<' and '>'. The accepted types of parameters are the following:

- `str`: an alphanumerical string with '_' and '-' accepted.
- `int`: an unsigned integer, it only accepts numerical digits.
- `flt`: a floating number, with an optional sign first, one or more digits and an optional '.' with one or more digits again.
- `hex`: a hexadecimal number, with digits or characters from A to F in uppercase or lowercase.

When the transaction for the URL is instantiated, it receives as the unique argument an associative array which contains all the parameters of the request. The parameters parsed from the body are under the `body` key, the parameters parsed from the URL (and the get parameters) are placed under the `get` key. If a parameter of the URL shares name with some of the get parameters, the value of the URL parameter will be the obtained one. Please, notice that the types of the parameters are just 'masks' for the pattern matching. No matter what type the parameter you declared is, the value will always be passed to the transaction as a string. It is up to you to parse it to the desired value.

Let's illustrate all of this with an example. Suppose we want to add a new URL to execute our new, awesome `GetHugsTransaction`. We want to execute it under a URL which will

allow us to specify the number of hugs we want to get. The first we do is to configure our `GetHugsTransaction` to receive this parameter in the URL, as follows:

```
<?php

namespace transactions;

use formats\IApiOutputter;

/**
 * GetHugsTransaction, it gives you as many hugs as you need!
 */
class GetHugsTransaction extends Transaction {
    /** @var int */
    private $hugs;

    /**
     * GetHugsTransaction constructor. The request params shall
     * have the following structure:
     * ['get' => ['hugs'=>number of hugs], 'body'=>[]]
     * @param array $requestParams the request params after
     *      all the processing
     */
    public function __construct($requestParams) {
        $this->hugs = intval($requestParams['get']['hugs']);
    }

    /**
     * Executes the transaction, has no return
     */
    public function execute() {
        $this->result = [];
        for( $i=0 ; $i < $this->hugs ; $i++ ) {
            $this->result[] = [
                    'name' => 'hug',
                    'affectionLevel' => rand(50, 100)
                ];
        }
        $this->status = IApiOutputter::HTTP_OK;
    }
}
```

Now that we have our transaction, we simply need to make the `UrlMatcher` to know about it. To do this we go to the `UrlMatcher` file and we edit the code of the constructor to be as follows:

```
/**
 * UrlMatcher constructor, private because of singleton pattern.
 * Here we register all the url patterns we want to use.
 */
private function __construct() {
```

```
6       $tmCategories = new TransactionMap();
7       $tmCategories->put(HttpMethod::GET(), GetCategoriesTransaction::
            ↪ class);
8
9       $tmCategoryItems = new TransactionMap();
10      $tmCategoryItems->put(HttpMethod::GET(), GetItemsTransaction::
            ↪ class);
11
12      $tmItems = new TransactionMap();
13      $tmItems->put(HttpMethod::POST(), CreateItemTransaction::class);
14
15      $tmItem = new TransactionMap();
16      $tmItem->put(HttpMethod::DELETE(), DeleteItemTransaction::class);
17      $tmItem->put(HttpMethod::PUT(), UpdateItemTransaction::class);
18
19      $tmHugs = new TransactionMap();
20      $tmHugs->put(HttpMethod::GET(), GetHugsTransaction::class);
21
22      $this->urls = [
23          new UrlPattern('/item-types/:itemType<str>/categories/',
                ↪ $tmCategories),
24          new UrlPattern('/categories/:categoryCode<str>/items/',
                ↪ $tmCategoryItems),
25          new UrlPattern('/items/:itemId<int>', $tmItem),
26          new UrlPattern('/items/', $tmItems),
27          new UrlPattern('/get/:hugs<int>/hugs/', $tmHugs)
28      ];
29  }
```

And that is all! We can now go to the address `http://yourhost.com/get/50/hugs/` and get fifty sweet hugs.

A.5 Building and deploying the PWA

To build the PWA the first we need is to install *npm*. It is the Node.js package manager and it runs over it, we can download and install both from the official page of npm: `https://www.npmjs.com/get-npm`. Once we have Node.js installed and npm ready to go we can go to our PWA folder through a command window. Then, we just need to run `npm install` and let npm do the magic, it will automatically read the `package.json` file and look for and install all the development dependencies. Between them, the most important one: Gulp. Gulp is a building software for JavaScript, it allows us to automate the tasks for building our website so we do not have to do them manually and it allows us to do it in a very comfortable way: using the same language we are using to develop our application.

There are four configuration files in our PWA main folder:

- `package.json`: As previously explained, this is the configuration file for `npm`, it saves infor-

mation about our "package". The most relevant for us of all the fields is the `devDependencies` array, which contains all the dependencies for the development of our project. To add a new dependency to the file simply add the argument `-D` or `-save-dev` when installing it with npm. The current development dependencies of our project are the following ones:

- **gulp**: the building software.
- **babel-core**, **babel-preset-env** and **gulp-babel**: the babel library with all the necessary dependencies to make it work with gulp. This library allows us to write code in ES6 (a modern JavaScript specification) and it compiles it automatically to "classic" JavaScript, making our code more compatible. This packacge is the reason for our Gulp configuration file to be called `gulpfile.babel.js`. The `env` preset configures babel to the latest presets so we do not have to specify them manually.
- **gulp-concat**: it allows us to automate the concatenation of several files into one. It is useful to reduce our JavaScript and our CSS to single files.
- **gulp-minify-css**: it minifies CSS deleting commentaries, line jumps, reducing rules which can be simplified, etc. We use it to reduce the size of our final CSS file.
- **gulp-uglify**: it minifies our JavaScript code in the same way `gulp-minify-css` does with out CSS. Thanks to this plugin we get to reduce the size of our resulting JavaScript file considerably.
- **sw-precache**: this is the library from Google which automatically generates the service worker for our cache managing. It is configured in the Gulp configuration file.
- **gulp-htmlmin**: minifies the HTML removing all the innecessary stuff.
- **gulp-sourcemaps**: generates source maps for our concatenated and minified CSS and JavaScript. The source maps allow the browser to map the sentences in the resulting files to their corresponding sentences in the original ones. This way we can debug our application easily without loosing the advantages of the minification.
- **gulp-replace**: it allows us to replace strings inside the files. It allows to do it based on regular expressions. We use it to set the base URL of the deployment into the `index.html` file.
- **gulp-clean**: it is used to remove files. Used to clean our distribution directory before certain tasks.
- **run-sequence**: it simply provides a quick method to run sequences of Gulp tasks. More information about Gulp task bellow in this document.
- **path**: a simple but useful tool that allows us to work with paths in a comfortable way without having to worry about whether a string ends with '/' or the other starts with it or not.

  - **fancy-log**: a library needed to configure the log of the `sw-precache` to the terminal.
  - **gulp-jsdoc3**: the library to generate the jsDoc files for our JavaScript code.

- `.babelrc`: it contains the configuration for babel. It does nothing more than configuring it to use the `env` preset.
- `jsdoc.json`: configuration file for jsDoc, currently only indicating it where to save the output.
- `gulpfile.babel.js`: the configuration file for Gulp with babel. It contains all the tasks we have programmed to build our site.

A.5.1 Gulp tasks

Our Gulp configuration file has been programmed with fourteen tasks which we can use to comfortably build our site, and they are the following ones:

- **clean-all**: it deletes the whole distribution folder `dist` with all its content.
- **dist-javascript**: it takes all the JavaScript files listed in a constant defined inside the gulpfile, called `DEV_JS_SRC`, concatenates all the files, *babelifies* them into old JavaScript and *uglifies* the result to reduce the size of the final file, which is saved in the JavaScript folder of the distribution folder (`dist/js/`). Note: it is important to keep in mind that only the files listed in `DEV_JS_SRC` will be concatenated and distributed. When a new JavaScript file is added, it needs to be added to this list to be distributed. Because of the way babel transforms the class hierarchies, the order the files appear in this list matters, and changing the order can cause the resulting distribution file to stop working.
- **dist-css**: this task will concatenate and minify all the CSS files in the development CSS directory (`src/css`) and save the to the distribution CSS folder (`dist/css`).
- **dist-html**: it will take our `index.html` file, replace the application base URL where it is needed, minify it and save it into the distribution folder.
- **dist-manifest**: it will simply replace the application base URL inside the manifest of the application and save the file into the distribution folder.
- **dist-other**: it distributes some other files associated to the manifest. These are the favicon and launcher icon files for the different common browsers.
- **dist-ico**: it copies into the corresponding folder in the distribution directory all the images and icons of the application (except the icons distributed by `dist-other`). Basically, these are all the images in the sub-folders of the `src/ico` folder.

- **clean-ico**: it cleans the `ico` folder in the distribution directory (`dist/ico/`).
- **generate-service-worker-dist**: generates the service worker which does both the pre-caching and the dynamic caching for our application. Notice that if the service worker was previously distributed and a browser has already installed it and downloaded the precached files, it will not update those files (between them, the JavaScript and the CSS minified files) until we make another distribution of the service worker (or they loose the cache for some reason). When you modify some of the precached files, distribute the service worker again so the browsers of the users can now they should update those files. Notice this task precaches the files in the dist folder so it needs the dist folder to be already generated when executing it.
- **generate-service-worker-dev**: in development environments, it is interesting to have the service worker caching the requests to check which resources are being cached and when they are updated, but it is annoying having the service worker serving the cache files cause it might prevent sometimes our code from being updated. With this task we generate a service worker which will cache everything just as the distribution version would do but will always answer with petitions to the network, never from cache. Notice this task precaches the files in the dist folder so it needs the dist folder to be already generated when executing it.
- **build**: it is the default task (the one which will execute if you introduce simply `gulp` in your terminal). It performs `dist-javascript`, `dist-css`, `dist-html`, `dist-ico`, `dist-other` and `dist-manifest`. It basically builds the web without the service worker (which needs to be generated after the distribution folder has been created to be able to precache the files correctly).
- **clean-build**: it executes `clean-all` (deleting the distribution folder) and, after it, `build`.
- **jsdoc**: generates the jsDoc for our JavaScript code. The output will be saved to the `out` folder. It is useful to get a quick introduction to the classes of our PWA and how they work.
- **watch**: it starts "watching" the CSS, JavaScript, HTML, manifest and "other" files so it will automatically rebuild the corresponding part whenever a change to the files is made.

To run a Gulp task, simply open a terminal in the PWA folder and type `gulp` *`task`*. For example, to clean and build the application type `gulp clean-build`. To completely build the PWA the first time, simply run the command `gulp` and, when it ends, run `generate-service-worker-dist`. In the `dist` folder your distribution files will be available.

Appendix B

User manual

We introduce here a preliminary version of the user manual. The current version of the application (the alpha) can be accessed in `https://app4refs.org/a`, please keep in mind this version is still under test (and that it might be already down by the time this document is being read). Check the root of the domain `https://app4refs.org/` for more information about the current status of development of the project.

B.1 What can be done with App4Refs?

App4Refs is the application which will offer the refugees all the necessary information to move around in the city of Athens. With App4Refs, any person can find information about locations to perform the most common and needed tasks: where to make a bank account, where to receive legal assistance, where to find LGBT+ information or things as simple as where to get a haircut.

B.2 Home screen

The home screen possesses six buttons, giving you access to six different kind of information. At the right, you can see the screen that is displayed when you enter the application. The icons (numerated in the image from 1 to 6) correspond to the following contents:

1. **Information**: this section provides you with places to obtain information about legal problems, assistance, general official places or embassies.

2. **Help**: this section provides you with quick, direct links to get help with common stuff like banks, social security, asylum, health and some more.

3. **Services**: lots of places where to get basic services, like WiFi connection, food, clothes, shelters, language training, simcard...

4. **Leisure**: information about leisure places where to spend the free time.

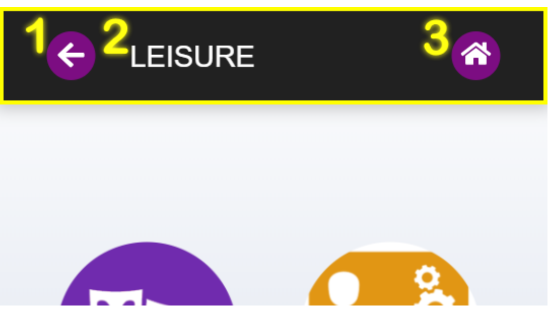
5. **Links**: useful links to web pages with information about education, health, hairdresser, information for refugees, etc.

6. **Emergency**: gives access to the emergency button, which gives you the number to call the emergency services in case you are in danger.

Please, notice the display can be slightly different from the one in the image if you are watching it in a computer or tablet.

Whenever you choose one of these kinds of information a list of the categories available for that kind will be displayed. Choose the one you need help about, we included very orientating icons to suggest what each category is.

B.2.1 Navigation bar



Whenever you are not in the home page the navigation bar will appear at the top of the screen and it will stand fixed there. In the image above this text you can see the navigation bar marked in yellow with the three main items it will always contain:

1. **Back button**: it allows you to go back to the previous page you were seeing. You can, alternatively, use the back button displayed on your phone controls or the back button of the navigator if you are opening the application through the browser.

2. **Title**: the title of the page you are seeing. In the example image we are in the *leisure* categories screen.

3. **Home screen button**: allows to go directly back to the home screen from any page inside the application.

B.2.2 List pages

Once you have chosen the category you want, the application will show you a screen like the one you see at the right of this text. This is the page that lists the information the application has for the requested category. Notice that depending on the category or the concrete place you are checking information about, some of the information may be not shown. For example, if you chose `links` in the home screen, the maps buttons will not appear.

We have marked and numerated each of the elements which could possibly appear in the list from 1 to 10. These are their meanings:

1. **Name of the place**: here you can see the name of the place being listed. In the example image there are three places with the names *Velos*, *National Park* and *Watoto Africa*.

2. **Address**: the address of the city where you can find the place. If it is an online place or it is not possible to show the address for some other reason, this field will not appear.

3. **Link**: a link to the web page of the displayed place. It might not appear if it is not available.

4. **Opening hours**: the schedules of the place so you can now when it is possible to contact / visit them. Several schedules (for different days or starting and ending hours) might be displayed, on per line, they all apply to the location. Some places do not have a predefined schedule so a call to get an appointment might be needed.

5. **Telephone number**: if the place does not have a schedule, then a telephone number is displayed to call the place. If you are on a mobile phone you can click the number to make the call.

6. **Map button**: clicking this button will take you to the map page where you can see the place on the map and, if you give the page access to your location, it will show you the route on how to get to the selected place (service provided by Google Maps).

7. **Item icon**: an icon describing the displayed item, in the sample screen shot there are three places shown with their respective icons.

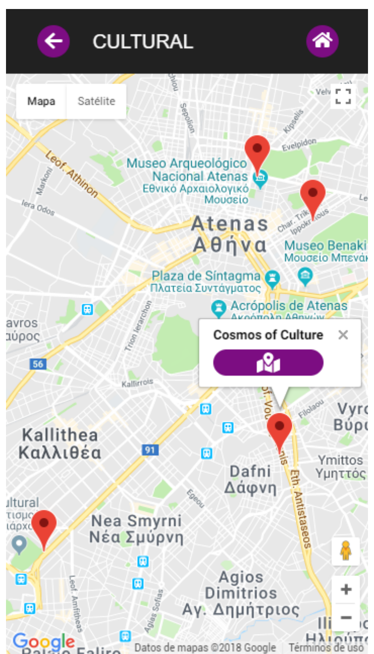
8. **Free icon**: an empty, green circle indicates the services offered in the place (or the visit to the place) are for free.

9. **Pay icon**: a green circle with the symbol of the euro inside indicates the place requires payment for some or all of the services it provides.

10. **Language icons**: the blue circles indicate the languages the item offers attention in. A circle with the text "*EN*" means it offers attention in English while "*GR*" means information is available in Greek.

At the end of the list you can find a big button like the one over this text which allows you to open the map page to show all the places of the list on it. From there you will be able to pick one place to check the route to it.

B.2.3 The maps page



When you click the button to show all the items in the map page you will see a screen like the this one[1].

The service is provided by Google Maps, so the interface is the same as their interface. Each marker (those red "balloons" over the map) indicates the position of one place of the list on the map. You can click them and a dialog like the one which can be seen in the screen shot will pop up just over the marker you clicked, indicating you the name of the item. In our case it was "Cosmos of Culture". When clicking on the violet button just underneath the name, the system will provide you with the route to the selected place.

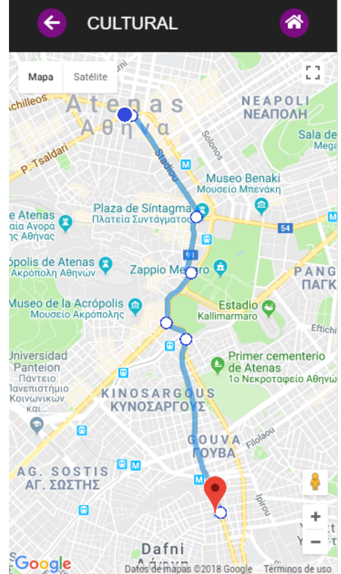
To display the route to the selected place the location service of your phone should be activated and the web page should have granted the access to it. Notice that if these two conditions are met, the map will also show a blue circle to mark your current location.

When clicking to get the route to an item (or if you open the item location through the *map* button placed under each of the items), if you have the geo-location service activated in your phone and our page has granted the access to it, the route will be displayed in a light blue over the map, marking the start of each of the steps of the route with a white circle.

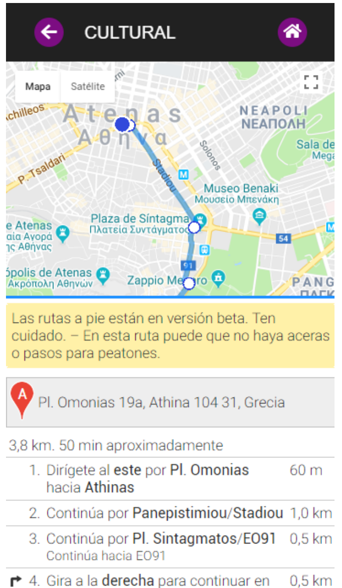
[1]The maps page is not available in the offline mode of the application.

Each of the steps of the route can be clicked and it will display a floating message with the indications for that point. The icon displaying your position gets updated in real time as you move through the route.

If you wish to see all the indications to get there in a textual form click any empty place on the map and the *indications panel* will appear from the bottom of the screen. The indications given by the indications panel appear in the language selected by Google based on the phone and browser configuration and the origin of your request.



The maps page displaying a route.



The indications panel shown over the maps page.

Bibliography

[1] “AGILE ALIANCE”: Beck, K. et al. *The Agile Manifesto* [online]. 2001 [visited in July 20, 2018]. Available in: https://www.agilealliance.org/

[2] ATLANTIC SYSTEMS GUILD LTD. *Volere Requirements Specification Template* [online]. Volere requirement resources [visited on July 10, 2018]. Available in: http://www.volere.co.uk/template.htm

[3] BOOSTRAP CONTRIBUTORS. *Bootstrap project home page* [web page] [visited on July 10, 2018]. Available in: https://getbootstrap.com/

[4] CACERES M. (MOZILLA CORPORATION). Web App Manifest. In: *W3C* [online]. K. R. Christiansen (Intel Corporation), M. Lamouri (Google Inc.), A. Kostiainen (Intel Corporation), R. Dolin (Microsoft Corporation), M. Giuca (Google Inc.). W3C Working Draft [visited on July 15, 2018]. Available in: https://www.w3.org/TR/appmanifest/

[5] ERICKSON, A. 6 basic questions about the war in Syria. *The Washington Post* [online]. April 15, 2018 [visited on July 7, 2018]. Available in: https://www.washingtonpost.com/news/worldviews/wp/2018/04/12/syria-explained/

[6] EUROPEAN COMISSION. *Migration and Integration Fund (AMIF)* [online][visited on July 9, 2018]. Available in: https://ec.europa.eu/home-affairs/financing/fundings/migration-asylum-borders/asylum-migration-integration-fund_en

[7] FIELDING, T. *Dissertation about representational state transfer (REST)* [online]. University of California, Irvine, 2000 [visited on July 10, 2018]. Available in: https://www.ics.uci.edu/

[8] FOWLER, M. *Patterns of Enterprise Application Architecture*. USA: Addison-Wesley, 2002. ISBN 0-321-12742-0

[9] FUNDACIÓ ACSAR [web page] [visited on July 9, 2018]. Available in: http://www.fundacioacsar.org/es/

[10] FUNDACIÓ UNIVERSITARIA BALMES [web page][visited on July 9, 2018]. Available in: https://www.uvic.cat/fundacio-universitaria-balmes

[11] “GANG OF FOUR”: Gamma, E., Helm, R., Johnson, R., Vlissiddes, J. *Dessign Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1994.

[12] GOOGLE. *Google Maps Javascript API Overview* [wen page]. Google Maps Platform [visited on July 10, 2018]. Available in: https://developers.google.com/maps/documentation/javascript/tutorial

[13] GOOGLE. *Progressive Web Apps* [web page]. Google Developers [visited on July 10, 2018]. Available in: https://developers.google.com/web/progressive-web-apps/

[14] HARDT, D. *The OAuth 2.0 Authorization Framework* [online]. Internet Engineering Task Force (IETF), RFC 6749 [visited on July 10, 2018]. Available in: https://tools.ietf.org/html/rfc6749

[15] JAVIER IDEAMI. *About Ideami* [web page] [visited on July 10, 2018]. Available in: http://ideami.com/ideami/

[16] JONES, C. *Assessment and Control of Software Risks*. USA: Prentice Hall, 1994. ISBN 978-0137414062

[17] KRUCHTEN, P. *The Rational Unified Process: An Introduction*. USA: Addison-Wesley, 1999. ISBN 978-0321197702

[18] MARIADB FOUNDATION. *About MariaDB* [web page]. MariaDB.org [visited on July 10, 2018]. Available in: https://mariadb.org/about/

[19] Migrant crisis: Migration to Europe explained in seven charts. *BBC* [online]. March 4, 2016 [visited on July 7, 2018]. Available in: https://www.bbc.com/news/world-europe-34131911

[20] NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS [web page][visited on July 9, 2018]. Available in: https://en.uoa.gr/

[21] NPM, INC. *npm project* [web page] [visited on July 10, 2018]. Available in: https://www.npmjs.com/

[22] *PHP Documentor* [web page] [visited on July 9, 2018]. Available in: https://www.phpdoc.org/

[23] PROJECT MANAGEMENT INSITUTE. *Project Management Body of Knowledge*. 6th edition. USA: PMI, 2017. ISBN 978-1-62825-184-5

[24] UNITED NATIONS HIGH COMMISSIONER FOR REFUGEES. *Europe situation* [web page] [visited on July 6, 2018]. Available in: `http://www.unhcr.org/europe-emergency.html`

[25] UNINTEGRA. *UNINTEGRA project*[web page]. Available in: `https://unintegra.usc.es/`

[26] UNIVERSIDADE DE SANTIAGO DE COMPOSTELA [web page] [visited on July 9, 2018]. Available in: `http://www.usc.es/en/index.html`

[27] UNIVERSIDADE DO MINHO [web page] [visited on July 9, 2018]. Available in: `https://www.uminho.pt/PT`

[28] *Web Assembly Home Page* [web page] [visited on July 23, 2018]. Available in: `https://webassembly.org/`

[29] WIKIPEDIA CONTRIBUTORS. Ajax (programming). In: *Wikipedia* [online] [visited on July 10, 2018]. Available in: `https://en.wikipedia.org/wiki/Ajax_(programming)`

[30] WIKIPEDIA CONTRIBUTORS. Usage share of web browsers. In: *Wikipedia* [online] [visited on July 15, 2018]. Available in: `https://en.wikipedia.org/wiki/Usage_share_of_web_browsers#Summary_tables`