Integrated Mobility Management System

Nuno Miguel Silvestre Cameira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Jorge Brás Monteiro Guerra e Silva
Filipe Manuel Mercier Vilaça e Moura

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. Luís Jorge Brás Monteiro Guerra e Silva
Members of the Committee: Prof. João António De Abreu e Silva

May 2018

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible.

I would also like to acknowledge my dissertation supervisors Prof. Luís Guerra e Silva and Prof. Filipe Moura for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

Several of transportation options exist nowadays when someone wants to travel from one place to another, but there is no way for someone to search specific options only available within that specific organization where one studies or works, while at the same time searching for general transportation options (e.g. public transports). To make this possible, we devised a configurable and extendable mobility portal, to be deployed within an organization, which enables its users to find the best transportation options to/from/between several sites, most often different facilities of the organization. We implemented a system that provides this functionality by querying external services that provide information on routes and respective schedule, with visual representation of the results and map interaction. We also implemented our own carpooling service and respective plugin that is connected to our mobility portal. An evaluation showed people could use and understand the whole system, thus certifying we had achieved our initial objectives.

Keywords

carpooling, mobility portal, trips, routes, driver, passenger

Resumo

Hoje em dia existem várias opções de transporte quando alguém quer viajar de um local para outro, mas não há como pesquisar opções específicas apenas disponíveis dentro de uma organização específica onde alguém estuda ou trabalha. Para tornar isso possível, criámos um portal de mobilidade extensível e configurável, a ser implantado numa organização, que permite que os utilizadores encontrem as melhores opções de transporte para/de/entre vários locais, na maioria das vezes, instalações diferentes da organização. Implementámos um sistema que fornece essa funcionalidade consultando serviços externos que fornecem informações sobre rotas e respectivo horário, com representação visual dos resultados e interação num mapa. Também implementamos o nosso próprio serviço de carpooling e o respectivo plug-in conectado ao nosso portal de mobilidade. Uma avaliação mostrou que as pessoas conseguem usar e entender todo o sistema, certificando que alcançámos os nossos objetivos iniciais.

Palavras Chave

carpooling, portal de mobilidade, viagens, trajetos, condutor, passageiro

Contents

List of Figures

Listings

1

Introduction

Nowadays there are many transportation options available when someone wants to travel from one place to another. With all these possibilities it becomes difficult to search for the most convenient option (either in financial terms, travel time or even the distance covered) for a particular situation. People can search for generic transportation options in several systems, the most well-known being Google Maps, but what if we look, for example, at the college students, how can they search for transportation options that only their college provides and only they can use? Or in companies with several facilities where employees must commute from their homes to a given facility or between facilities? There is not a particular system that allows these specific transportation options while at the same time searching for general transportation options (e.g. public transports), thus we propose to develop one.

In this work we developed a system able to query different services for routes and their schedules in order to find the one that best suits the user's needs. This system which we called Mobility Portal, should be able to be deployed on any organization that provides their own transportation options by adding custom plugins that can query those specific transportation services, as well as other external services that may be of interest to the user community.

Since this system aims to be deployed on a given organization, this means that all its users are part of the same organization. This is a very interesting situation and has a lot of potential, because not only it will be easier for users to trust each other, but also, because most of the times their trips share the same starting or ending point, which is the location (or one of the campi) of the organization to which they both belong to, making it much more effective. Therefore we thought that it would be useful to develop a Carpooling system and use it as a transportation provider in the Mobility Portal. In order to do this we developed a plugin for the Mobility Portal that can communicate with this Carpooling system.

Carpooling is basically the sharing of car journeys among several people. It is inexpensive, since in carpooling usually the travel expenses are divided between all the occupants of the vehicle, it is environmentally friendly and contributes to the reduction of traffic congestion since fewer vehicles will be needed, which means less emissions per passenger thus less pollution. There is also the case of a social benefit for both driver and passengers. With all these advantages why should anyone drive alone? With this question in mind we want to encourage users to do carpooling. For this Carpooling service we analyzed different existing solutions and compared them in order to take advantage of their best advantages and avoid their drawbacks.

Thus our two main objectives are:

- Create a modular system (Mobility Portal) that allows users to define an origin, destination and departure date and search for different ways to travel from one point to the other on that departure date;
- Build our own Carpooling service and add the respective plugin to the Mobility Portal, such that it may query that service.

A requirement underlying all our developments is that both the Mobility Portal and the Carpooling system should be able to be deployed on any organization with minimal effort.

In the next Chapter we will discuss several existing systems related both to Carpooling and Mobility Portals. Additionally, we will also review several technologies that can be used to implement our systems. Then, in Chapter 3, we will describe the proposed approach for the design of both systems, detailing the types of users, the actions that they can execute and the general requirements that the systems should conform to. In Chapter 4 we will describe the implementation, details concerning both the back-end and front-end. Chapter 5 presents experimental evidence for validating and evaluating the performance of the system. Finally, we conclude in Chapter 6 with a brief summary of our work and proposals for future work.

2

Related Work

Contents

In this section we will review background information including existing approaches to carpooling and mobility portals that relate to our work and the different technologies that can be used to implement our systems.

2.1 Existing approaches

In this section we present the existing approaches to carpooling and mobility portals that relate to our work.

2.1.1 Arena Mobility portal

Our work requires the development of a mobility portal that allows users to search for options to travel from one place to another. The Arena Mobility Portal [1] provides part of that functionality. On this online platform the users can find transportation options from and to the Amsterdam Arena. The key information that is requested on this system interface is: origin, destination and date. The user must fill all of those fields except one, either the origin or the destination must be the Amsterdam Arena, the other must be filled with either a location or a ZIP-code, which is depicted on Figure 2.1.

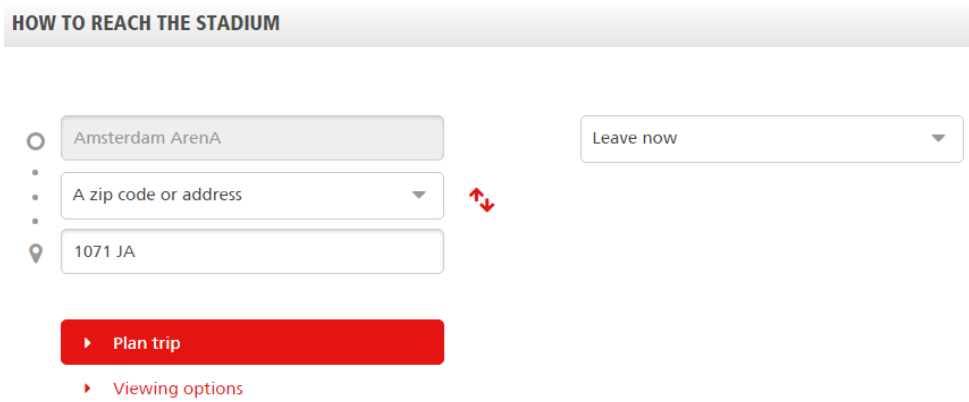

Figure 2.1: System interface with an example of zip code

Then the system generates a set of routes and schedules that consists of different traveling options that are displayed in a map for better visual understanding, depicted on Figure 2.2. Some of these options are public transportation, parking information, taxi information, bicycle, motorbike and parking spaces for disabled persons, as depicted on Figure 2.3. The user can even book the transportation services presented on one of those options. For example, if an option is by train the system provides a link that will redirect the user to a website where the user can buy a ticket.

This system has one major drawback that we do not want to have in our Mobility Portal, which is locking one point, either the origin or destination point needs to be the Amsterdam Arena.

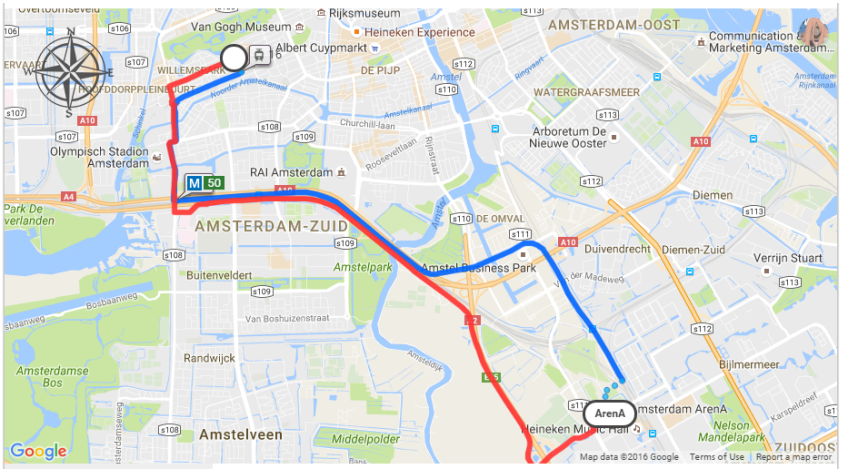

Figure 2.2: Query response for the example on Figure 2.1

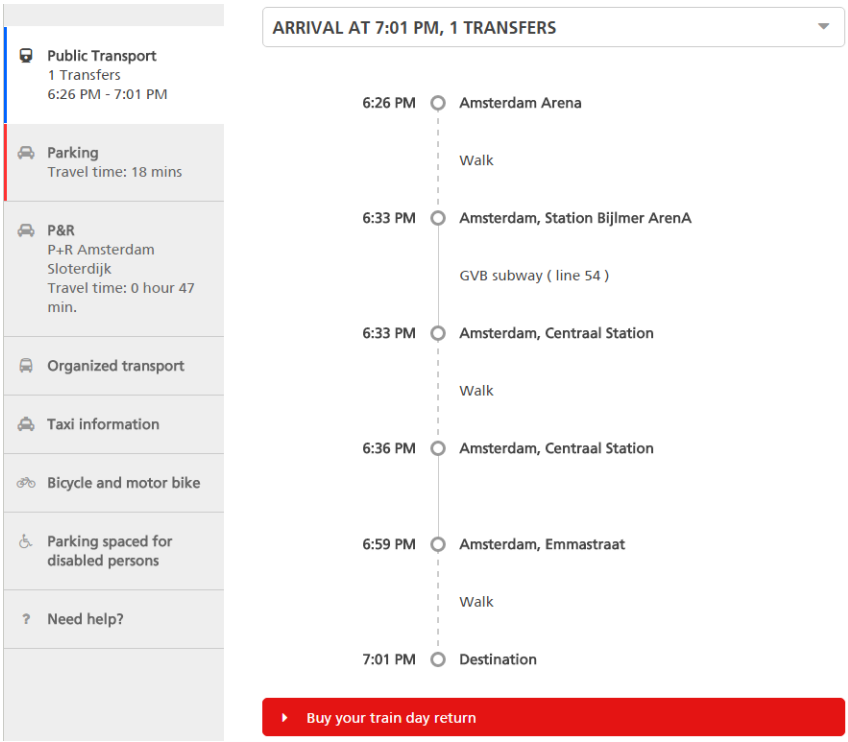

Figure 2.3: Different transportation options for the example on Figure 2.2

2.1.2 511 SF Bay

The 511 SF Bay [2] is a free web source for up-to-the-minute Bay Area traffic, transit, rideshare, and bicycling information. It is available 24 hours a day, 7 days a week, from anywhere in the nine-county Bay Area. The search interface of this system is a box which is depicted on Figure 2.4, where we can specify an origin and destination point for the trip, whether we want to go by car, public transportation, bicycle or on foot and a depart date. This box is on top of a map where the results for the given inputs will appear, an example of this is depicted on Figure 2.5. On top of this map there is a yellow horizontal bar where reports of accidents are shown, this same accidents are also displayed on the map with a yellow sign, which can be seen also on Figure 2.5. This particular information on accidents is very helpful for a user to choose between different options.

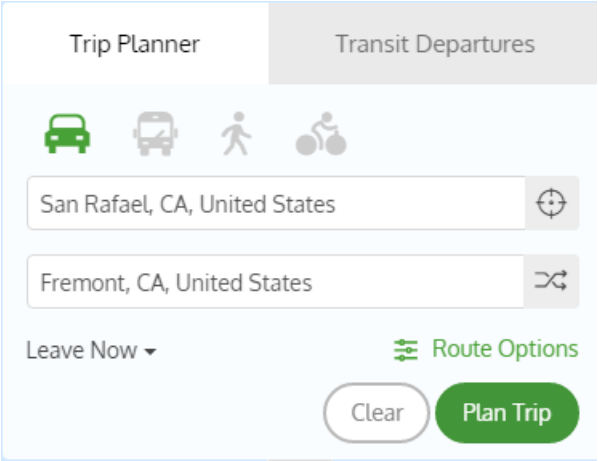

Figure 2.4: System interface with an example

2.1.3 Carpooling

Carpooling is basically the sharing of car journeys whereby a driver agrees to share its car with additional passengers during a given trip. There are many carpooling services across the world. The most well known in Portugal are **boleia.net** [3] and **blablacar.pt** [4]. They both share the core functionality of any carpooling service, for example they both require:

- An origin and destination
- A depart and arrive date
- A price
- The number of seats
- Additional comments about the trip

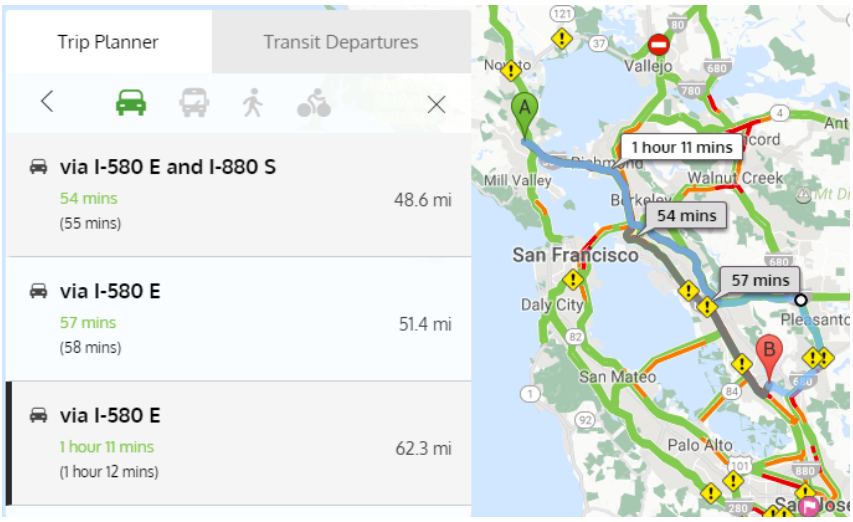

Figure 2.5: Query response for the example in Figure 2.4

- The possibility to see the trip on the map
- The possibility to inspect the driver's profile

Both boleia.net and blablacar.pt interfaces are depicted on Figure 2.6 and 2.7 respectively.

Figure 2.6: Boleia.net interface

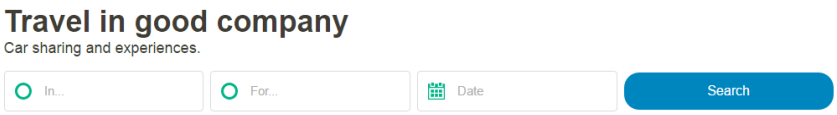

Figure 2.7: Blablacar.pt interface

After providing the origin and destination (the date is not mandatory on blablacar.pt and does not even exist as input on boleia.net) both systems will provide a list of options that meet the given goal, depicted on Figures 2.8 and 2.9 (in the same order). These options can be inspected for even more details.

In addition to this the also allow a user to login with the facebook account. Although they both say how many seats are available, blablacar.pt also says which seats are available (e.g. only back seats are available). We have seen what they have in common but they have some differences as well, for instance in boleia.net a user can define a recurring trip (e.g. Sunday, every week) and define possible

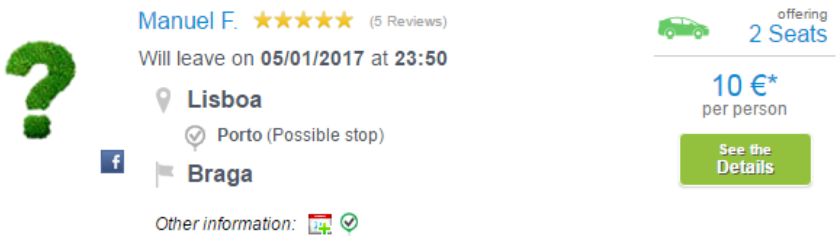

Figure 2.8: Boleia.net query response example

Figure 2.9: Blablacar.pt query response example

stops. Blablacar.pt does not enable the user to define possible stops but it has a similar functionality which is allowing the user to specify a detour time (e.g. 15min). Additionally it also enables the user to define how much space the car has for luggage (e.g. small, big) and how flexible is the departure time (e.g. leaves right on time). In the end it provides a Q&A section where users can ask questions to the driver. They both implement a user profile with most information in common, for instance they both have:

- Name
- Age
- Other people's evaluation of past trips with that user
- Rating
- Car model
- Past trips (although they show it in different ways)
- E-mail and phone number (both hidden but validated)

One information that only boleia.net has in the user profile is the languages that the user speaks, which is quite relevant in this context. Blablacar.pt allows the user to specify its preferences (e.g. likes to talk, does not transport animals) which can also be useful information to carpooling.

We also found an interesting carpooling service in North America, which is **Pop Rideshare** [5]. This service has an appealing interface, where a map with points representing trips that begin or end in those points is displayed (the bigger the point size the more trip options available), as depicted in Figure 2.10.

This interface allows the users to have an idea of what trips are available, even without having done any search.

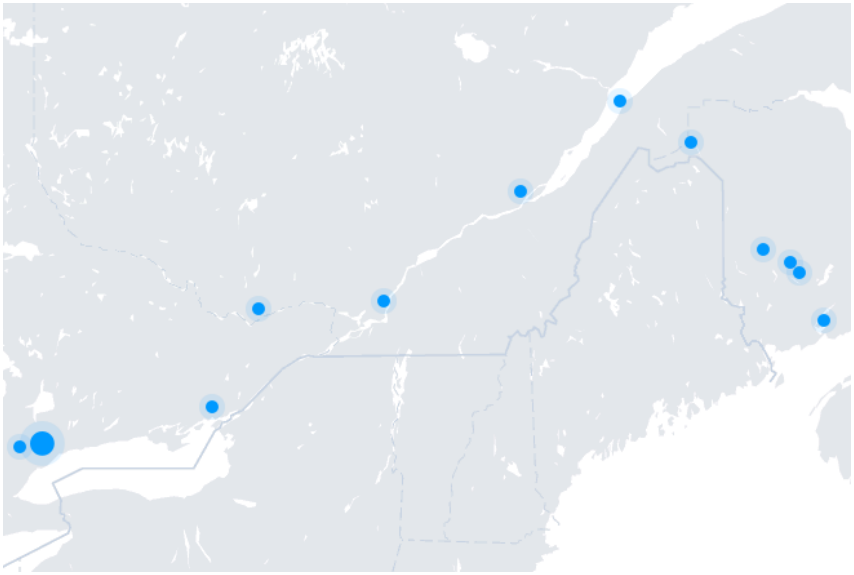

Figure 2.10: Pop Rideshare map

The usual interface with a starting and ending point, and an optional departure date is also available, as depicted on Figure 2.11. When a user does a query, either with that interface or by selecting a point in the map, a list of results is presented, as depicted in Figure 2.12.

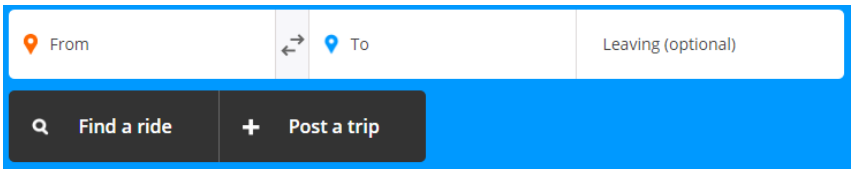

Figure 2.11: Pop Rideshare search interface

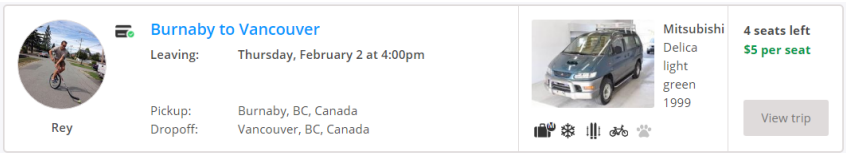

Figure 2.12: Pop Rideshare query result

Some interesting aspects are that the users can upload a picture of their own car, choose a set of preferences and allow passengers to pay for the trips they booked and the driver to receive that money.

The last carpooling service we have examined was the **SINGU Car Pooling** [6] from Instituto Superior Técnico. The first thing that comes up after logging in is a menu asking if I am a passenger or a driver

as depicted in Figure 2.13.

Figure 2.13: SINGU interface after login

This is a very helpful feature for users that are new or still not very comfortable with the interface:

- I'm a passenger - directs the user to the find trip menu;
- I'm a driver - directs the user to the create trip menu.

If we move to the create trip interface depicted on Figure 2.14, we can see that the "Departure" allows any location to be inserted but the "Destination" only allows three options:

- Tecnológico e Nuclear (CTN);
- Taguspark;
- Alameda.

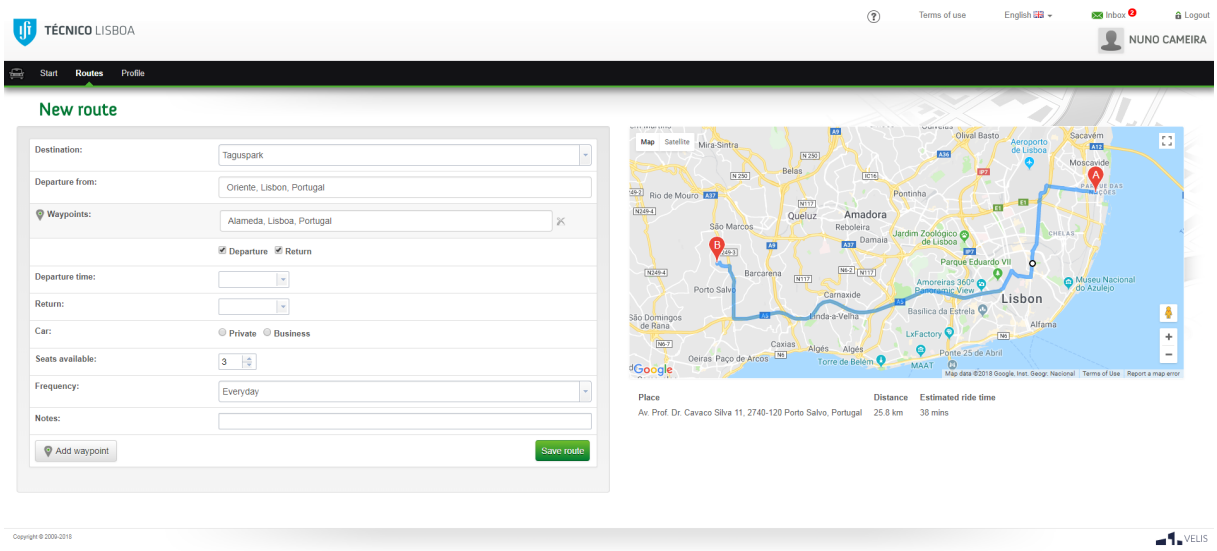

Figure 2.14: SINGU interface to create a trip

Which we consider a negative aspect, since the users of this system are all part of the same organization it should explore that aspect and remove that three location restriction. A positive aspect is the "Add waypoint" functionality which adds a location between the origin and the destination, thus allowing it to be found if a user searches that waypoint as destination. After the departure place is filled the route is displayed on a map with the distance and estimated time, which helps the user to have a better visual

understanding of the trip. Then we have the option to select a departure and return time which is great for passengers to see a more detailed trip. There is also a frequency field which has to be select with one of the following:

- Selected days (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
- Everyday;
- Business days & Saturdays;
- Business days.

The departure and return times in combination with the frequency will generate the trips, therefore the system only allows the creation of recurring trips, which we also find limiting, what if a user only wanted to create just one trip?

If we move to the find trip interface depicted in Figure 2.15 the first thing we noticed was that it only requires one place to find trips, which is the origin, since the destination can only be one of the three options previously described.

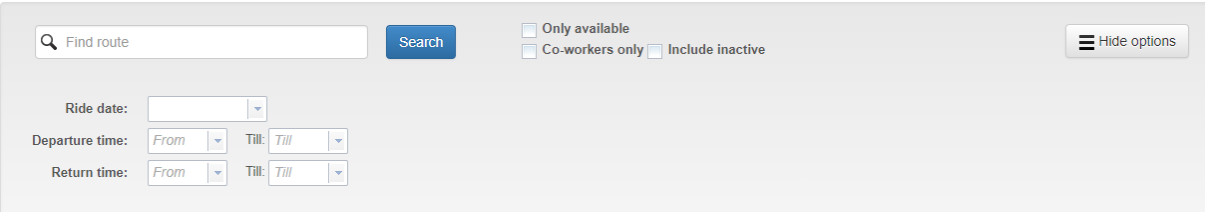

Figure 2.15: SINGU interface to find a trip

The system allows for a more customized search by specifying the ride date, departure and return times. After searching for a trip it displays a list of trips with the basic information (destination, frequency, departure, return). If we inspect one of such trips, the information depicted on Figure 2.16 is shown. Here we can join the passengers of that trip or check other future trips from this driver with the same frequency and join those. This join request needs to be accepted by the driver.

2.2 Technologies

In this section we present the different technologies that can be used to implement our system.

2.2.1 Representational State Transfer (REST)

The Representational State Transfer (REST) architectural style provides important qualities like: performance, scalability, simplicity, modifiability, visibility, portability, and reliability [7]. Nowadays almost

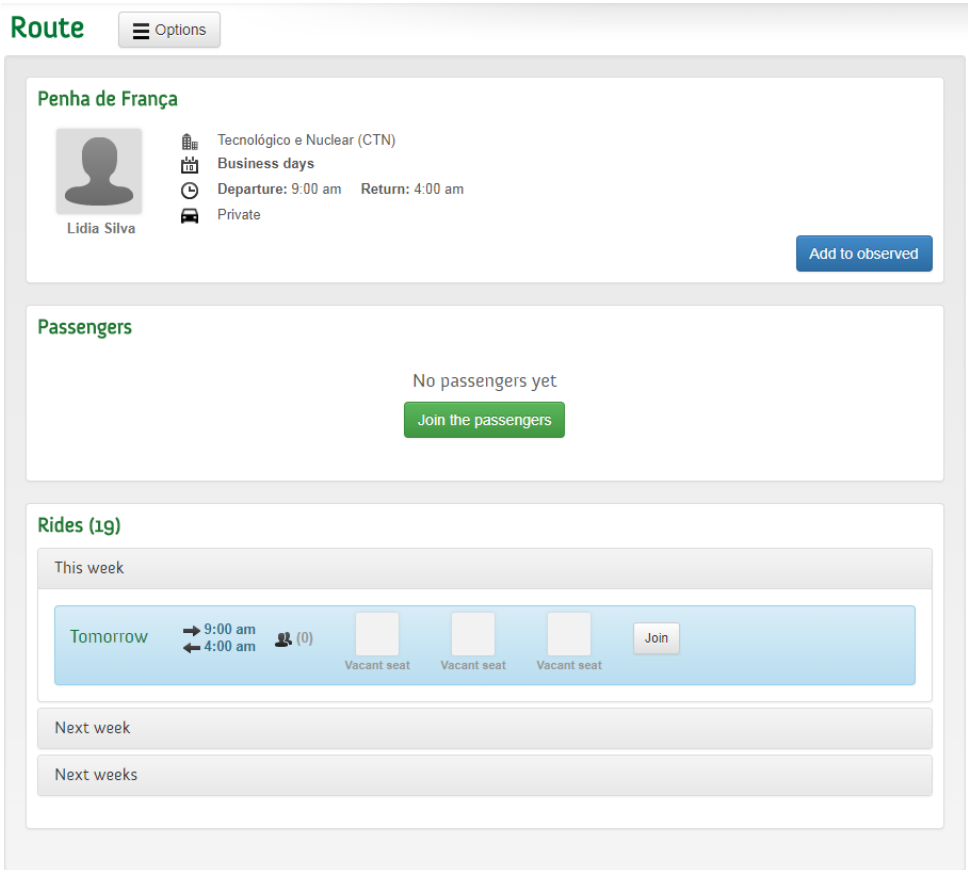

Figure 2.16: SINGU interface to view a trip

every project or application provides a REST API, such as Twitter, YouTube or Facebook. It is the most logical, efficient and widespread standard in the creation of APIs for Internet services. It relies on a stateless, client-server, cacheable communication protocol and in most of the cases, the HTTP protocol. The main idea is that, rather than using complex protocols like RPC or SOAP to connect between machines, it simply uses HTTP to make calls between machines. REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components to reduce interaction latency and enforce security. So this is a great approach for our Mobility Portal and Carpooling design.

A simple example of a RESTful Web Service would be:

- A service that accepts HTTP GET requests at **http://localhost:8080/greeting**;
- And respond with a JSON representation of a greeting {**"content":"Hello, World!"**}.

Or even customize the greeting with an optional name parameter in the query string:

- HTTP GET request **http://localhost:8080/greeting?name=User**;
- JSON response {**"content":"Hello, User!"**}.

2.2.2 Spring

The Spring Framework is an open source application framework and inversion of control container for the Java platform. The Spring Web model-view-controller (MVC) [8] framework is designed around a DispatcherServlet that dispatches requests to handlers. These handlers are based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods. With the introduction of Spring 3.0, the @Controller mechanism also allows the creation of RESTful Web sites and applications, through the @PathVariable annotation and other features.

Like Struts [9], Spring MVC is a request-based framework. The framework defines strategy interfaces for all of the responsibilities that must be handled by a modern request-based framework. The goal of each interface is to be simple and clear so that it is easy for Spring MVC users to write their own implementations, if they choose to do so. MVC paves the way for cleaner front end code. A request processing workflow in Spring Web MVC is depicted on Figure 2.17 which illustrates most of the important interfaces defined by Spring MVC, and their responsibilities.

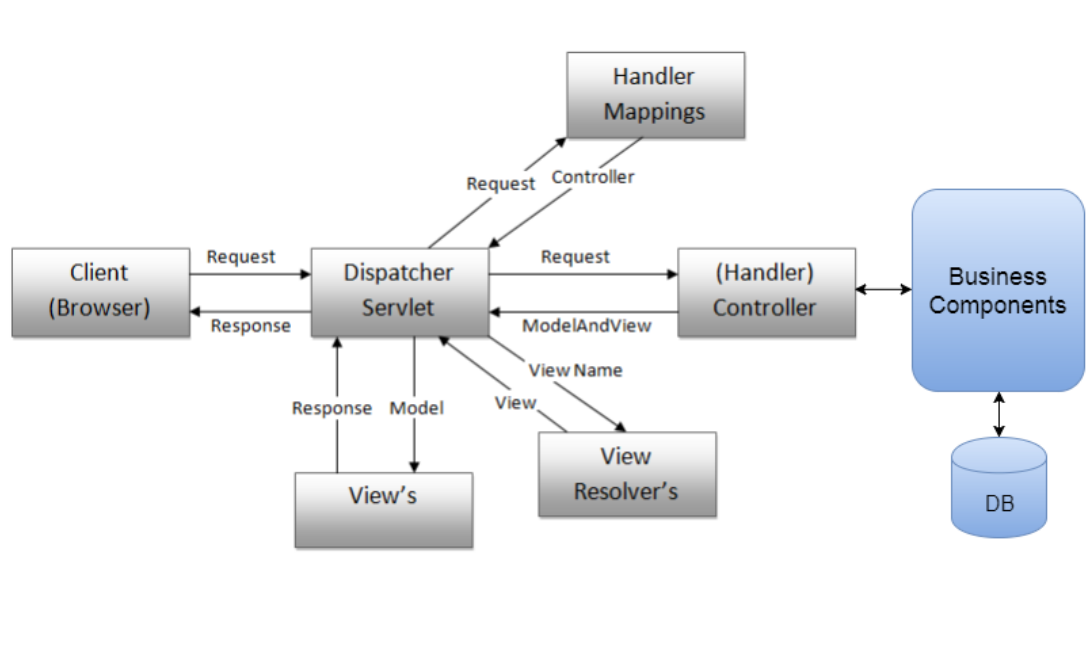

Figure 2.17: Request processing workflow in Spring Web MVC

DispatcherServlet is the front controller of the framework and is responsible for delegating control to the various interfaces during the execution phases of an HTTP request.

HandlerMapping selects objects that handle incoming requests (handlers) based on any attribute or condition internal or external to those requests.

Controller comes between Model and View to manage incoming requests and redirect to proper re-

sponse. It acts as a gate that directs the incoming information. It switches between going into model or view.

ViewResolver selects a View based on a logical name for the view (use is not strictly required)

View is responsible for returning a response to the client. Some requests may go straight to view without going to the model part; others may go through all three.

Each strategy interface above has an important responsibility in the overall framework. The abstractions offered by these interfaces are powerful, so as to allow for a set of variations in their implementations. The ease of testing of the implementations of these interfaces is one important advantage of the high level of abstraction offered by Spring MVC.

Within an application, the view layer may use one or more different technologies to render the view. Spring web-based applications support a variety of view options, often referred to as view templates. These technologies are described as ”templates” because they provide a markup language to expose model attributes within the view during server-side rendering.

2.2.3 Thymeleaf

Thymeleaf [10] is an modern open-source Java XML/XHTML/HTML5 template engine that can work both in web (Servlet-based) and non-web environments. It is better suited for serving XHTML/HTML5 at the view layer of MVC-based web applications, but it can process any XML file even in offline environments. It provides full Spring Framework integration.

In web applications Thymeleaf aims to be a complete substitute for JSP, and implements the concept of elegant Natural Templates: HTML that can be directly displayed in browsers and that still work correctly as static prototypes.

2.2.4 Google Maps

The Google Maps API [11] offers a wide range of services, among those we will state the ones that are more relevant for our work, which are satellite imagery, street maps and route planning, for traveling by foot, car, bicycle, or public transportation.

In order for our Mobility Portal to reach a set of routes that fulfill the user needs it will need to use different plugins to query external services for information to aid in that goal. This information will contain routes, departing dates, methods of transportation and other information, we can view it like “transportation schedules”. There is already a specification that is meant to transmit this kind of information to Google Maps.

The General Transit Feed Specification (GTFS) [12] which defines a common format for public transportation schedules and associated geographic information. This enables public transit agencies to publish all their schedule information so that the developers can then consume it in an interoperable way. We can take this specification and use something similar or even follow the exact specification to transmit information between the plugins and our system.

If after asking all of our plugins for information the system can not reach a valid route, it will delegate what is left of that task (if not all) to a Google Maps Plugin. This Plugin will serve as a last resort to complete what is left of our current task.

The Google Maps Places service, which is used to search for places (defined in the API as establishments, geographic locations, or prominent points of interest) contained within a defined area, such as the bounds of a map, or around a fixed point. It also offers an autocomplete feature which is used for the type-ahead-search behavior of the Google Maps search field. When a user starts typing an address, autocomplete will fill in the rest.

The Google Maps Directions service, which is used to calculate directions (using a variety of methods of transportation) has some limitations in case we use the standard plan (free), which we will quote from [13] :

> *"2,500 free directions requests per day, calculated as the sum of client-side and server-side queries. Up to 23 waypoints allowed in each request, whether client-side or server-side queries. 50 requests per second, calculated as the sum of client-side and server-side queries."*

In short we can use Google Maps for the following purposes: to find places, to use the auto-complete functionality, to calculate routes and to present the final result to the user.

The algorithm used by Google Maps to find the fastest/shortest path between two points is based on a search over a giant graph where sites are modeled as nodes and roads are modeled as edges. It then uses some variation of the Dijkstra's algorithm to find the shortest path. This is a very complex task since shortest path problems can get insanely deep very fast, which would result in a combinatorial explosion, thus making the problem practically incomputable and taking an infinite amount of time to solve. To overcome this problem, Google Maps uses some sort of hierarchical superstructure on top of the graph. For example, they might add edges that connect major traffic hubs within cities to the highways close to that city in one step, or in addition to splitting a highway into one node/edge per exit they may also add 'longer' edges that connect exits at major cities. The algorithm then decides on which level of the hierarchy it wants to operate for different parts of the route. In that way, the algorithm can explore longer paths without having to explore every back alley of each city. There is even an interesting article that compares the Dijkstra's algorithm and Google Maps to find the shortest path on a predefined map. [14] In order to compare them a map was created based on a small section of Williamsburg in Brooklyn, NY,

with circles representing graph vertices and black lines which are directed edges with assigned weights. Then populated the graph with edges and vertices as depicted on Figure 2.18.

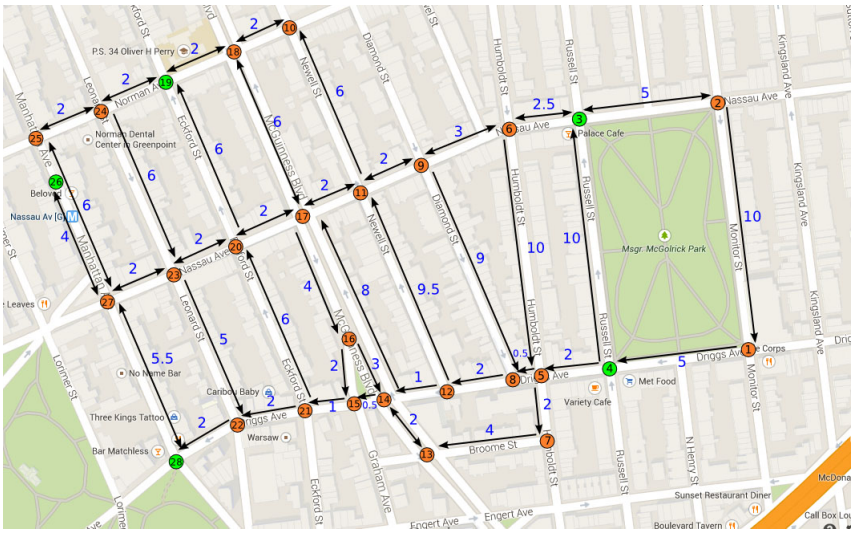

Figure 2.18: populated the graph

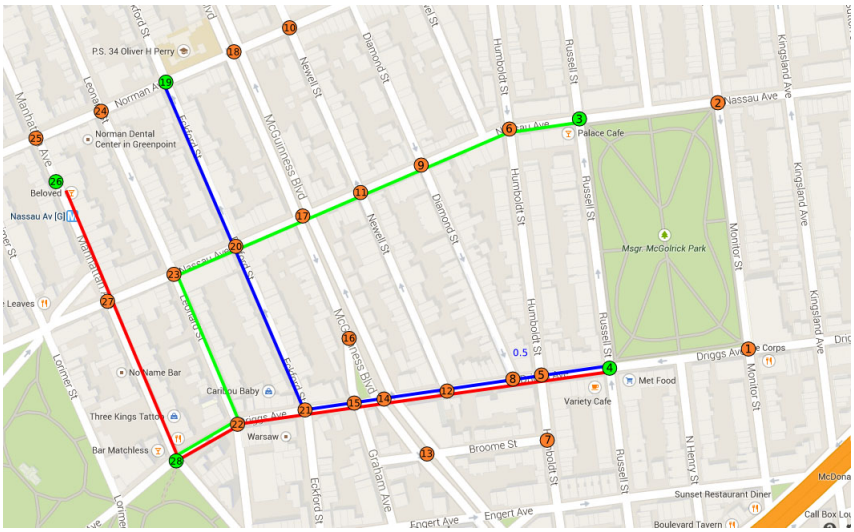

Figure 2.19: Dijkstra's algorithm output

Starting and destination points are on green. Then the Dijkstra's algorithm was run and gave the output depicted on Figure 2.19. After that they chose the exact starting and ending points and used Google Maps to calculate the shortest path, depicted on Figure 2.20.

With this they concluded that the red and blue routes are exactly the same for both algorithms, the green route however is a bit different. Google decided to take a left turn earlier for some reason and consequently adding slightly more distance to the route. This was due to the fact that they calculated their edge weights by using only distances between two points. In real life, edge weights should be

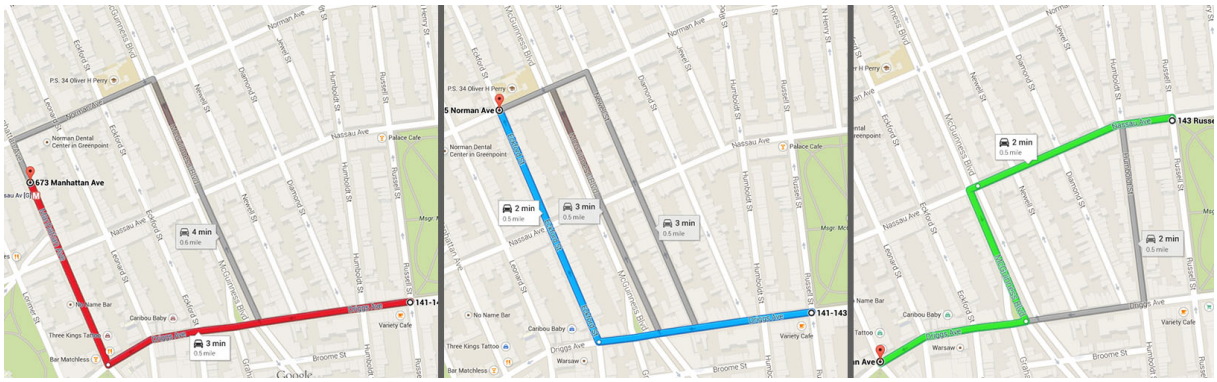

Figure 2.20: Google Maps output for the same starting and ending points

a function of distance, speed limits, traffic conditions, etc. The Google Maps result may also change depending on the time of the day because traffic patterns can change throughout the day.

2.2.5 Authentication

Our work requires that the users should be authenticated in order to be able to use it. An authenticated user is defined as someone that is enrolled in the centralized authentication system of the organization, in this case Instituto Superior Técnico (IST).

We want to avoid asking for the IST credentials, so that our system doesn't have to manage passwords, instead we want to delegate the authentication to another service, we can even ask IST for access to only specific user data that our system needs, this way if our system becomes compromised the IST credentials are still safe. Nowadays this delegation of authentication is the most used method to authenticate a user.

There are several frameworks related to this feature, the most relevant are **OAuth 2.0** [15], **OpenID**, **OpenID Connect** [16] and **CAS** [17].

The **OAuth** (Open Authorization) is an open standard for authorization and token-based authentication on the Internet. This standard enables Internet users to authorize third-party services, such as Facebook, Google, and others, to access their information without exposing the user's credentials. [18] Basically OAuth acts as an intermediary on behalf of the end user, providing third party applications with an access token that authorizes specific account information to be shared.

OAuth 2.0 is the next evolution of the OAuth protocol which replaces and obsoletes the OAuth protocol [19]. This version focuses on simplifying client development while providing specific authorization flows for web applications, desktop applications, mobile phones and living room devices. The key aspect here is that OAuth 2.0 is not an authentication protocol [20]. Its focus is authorization, it grants access to data, functionality or other things without having to deal with the authentication.

OpenID is a framework which handles authentication (proving that a user is who is said he is). The OpenID protocol enables users to sign in to multiple websites using login credentials from an OpenID provider (e.g. Google) without creating a separate identity and password for each [21]. A user can associate information with his OpenID and control how much of that information is shared with the websites he visits.

If we compare OpenID with OAuth we see that the main differences are that, with OpenID we get an assertion of identity, while in the OAuth we get an access token that may grant the application access to some APIs, on the user's behalf. This token doesn't provide any information about the user identity.

There is a framework that combines the two previous solutions, **OpenID Connect**, which is an interoperable authentication protocol based on the OAuth 2.0. OpenID Connect allows third-party services to launch sign-in flows and receive verifiable assertions about the identity of signed-in users. It is said that OpenID Connect is the sum of Identity and Authentication plus OAuth 2.0 [22].

Central Authentication Service (CAS), is a single sign-on (SSO) protocol which offers some key features like support for standards (OAuth Protocol, OpenID & OpenID Connect Protocol), RESTful API, support for clustering, services management and long term SSO. The highlight is that it allows web applications to authenticate users without gaining access to a user's security credentials, such as a password, which is exactly what we are looking for. Although we referred to CAS as a protocol [23] we can also refer to the software package that implements this protocol.

We will need to analyze deep down if just using OAuth 2.0 is enough for our work, or if we need to use it as part of an authentication protocol.

2.2.6 Databases

In this section we will analyze two very well known technologies for databases, **MySQL** and **MongoDB** by briefly describing each one and comparing them.

MySQL is an open-source relational database management system (RDBMS) and is mostly used to store data for web applications. Basically it stores data in tables and uses a structured query language (SQL) for database access. With MySQL we can predefine the database schema and set up rules to govern the relationships between fields in the tables.

MongoDB is an open-source document-oriented database that stores data in JSON-like documents (called collections) that can vary in structure. It was designed with high availability and scalability in mind, and includes out-of-the-box replication and auto-sharding. It supports horizontal scaling through sharding which is a method for distributing data across multiple machines. This sharding method is used to support deployments with very large data sets and high throughput operations. MongoDB uses

dynamic schemas, which means that we can create records without first defining the structure, such as the fields or the types of their values. Related information is stored together for fast query access through the MongoDB query language. An important feature is that documents do not need to have an identical set of fields and it is common to use denormalization of data. One major limitation of MongoDB is that unlike the relational MySQL, it does not offer an easy way to join tables.

SQL allows more complex and accurate queries and although they scale well vertically (more memory, faster disks, better cpu, etc) due to the necessary table and database structure in relational databases, they do not scale well horizontally (adding more servers). MongoDB can be scaled within and across multiple distributed data centers, providing new levels of availability and scalability thus reaping the advantages of distributed computing.

We can conclude that MongoDB is a more flexible, scalable and available system, with a quicker query response. While MySQL allows more complex and multi-row transactions protected by the security and data integrity offered by relational databases. Each one excels in certain scenarios and is unfit for others. Due to their very different structures each one contains features not found in the other.

2.2.7 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language like HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

CSS is designed to enable the separation of presentation and content, including layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file, and reduce complexity and repetition in the structural content.

2.2.8 JavaScript

JavaScript is a high-level, interpreted programming language (executes instructions directly and freely, without previously compiling the program into machine-language instructions). It enables interactive web pages and thus is an essential part of web applications. The vast majority of websites use it, and all major web browsers have a dedicated JavaScript engine to execute it.

3

Proposed Approach

Contents

In this chapter we detail the requirements and functional aspects of both our systems, the Carpooling and the Mobility Portal.

3.1 Carpooling

The main goal of our Carpooling service is to allow driver users to offer trips with a specific departure date, duration, starting and ending locations, with a given number of seats and price per seat. Passenger users can perform queries on such trips (e.g. retrieve all trips that go from point A to B and depart at 01/11/2018 at 15:00) and book seats by issuing a request that can be accepted or rejected by the corresponding driver. In order to maintain the system we created an administrator role.

Our system has three different roles for users:

- Passenger;
- Driver;
- Administrator.

We list below all the possible operations that our system allows:

- As a Passenger
  - Find a trip;
  - Request to book seats on a trip;
  - View requests;
  - Cancel requests;
  - View current and past trips;
  - Rate trips.
- As a Driver
  - Create a trip;
  - View book requests;
  - Accept/reject requests;
  - View current and past trips;
  - Delete a future trip;
  - Delete recurring trips.
- As an Administrator

  - Add/remove preferences;
  - Add/remove roles;
  - Add administrator permissions to a user.

- As any role (Administrator/Driver/Passenger)
  - View notifications;
  - View public profile;
  - Edit profile.

A very important aspect of any carpooling service is trust between users. Our system targets organizations where, usually, there is a good starting level of trust between users. Most often such organizations rely on a central authentication system. Therefore our system was designed to easily interface with such a system. In our test case we are using Fénix for authentication as well as to extract some basic information about the user.

In order for the users to have a clear understanding of our system and how to interact with it as easily as possible we decided to divide it into three main menus, which are displayed on a navigation bar on the top of the page, each of these menus have their own set of tabs:

- Passenger
  - Find Trip;
  - Requests;
  - My Trips.
- Driver
  - Create Trip;
  - Requests;
  - My Trips.
- Profile
  - Notifications;
  - Edit Profile;
  - Public Profile.

We will start by presenting the typical scenario in which our system will be used. Currently the users of our Carpooling system must be enrolled at Instituto Superior Técnico to be able to authenticate themselves with Fénix.

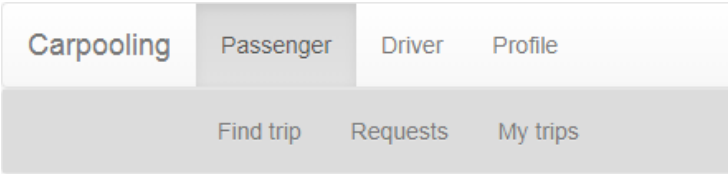

Figure 3.1: Navigation bar Passenger menu

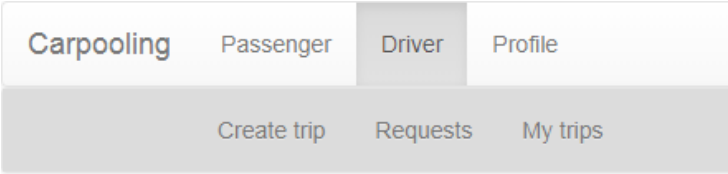

Figure 3.2: Navigation bar Driver menu

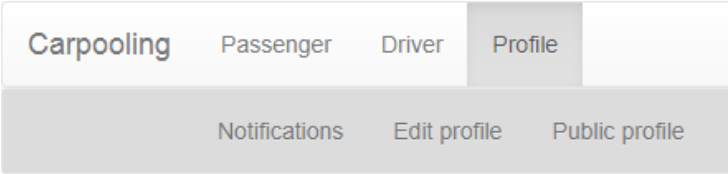

Figure 3.3: Navigation bar Profile menu

The roles that a user can have fluctuate as he navigates the interface of our system, more precisely the navigation bar with the menus of a passenger, driver and profile (which is generic), depicted in Figures 3.1, 3.2 and 3.3. The typical user will only jump between passenger and driver. The administrator role is protected with a special login (username and password) that is required in order for a user to assume that role, which is in the same interface that enables the login with Fénix, depicted on Figure 3.5. If a user navigates to any of the sub-menus of the passenger then he is currently assuming the role of a passenger, or if a user navigates to any of the sub-menus of the driver then he is currently assuming the role of a driver. All the sub-menus of each menu are grouped in a drop-down list on the top right of the interface, depicted on Figure 3.4.

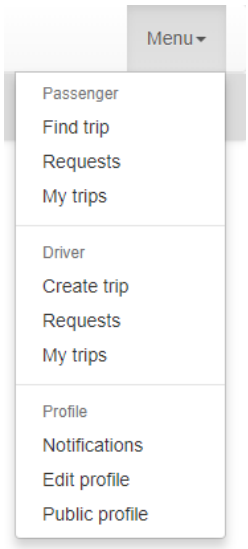

Figure 3.4: Drop-down Menu

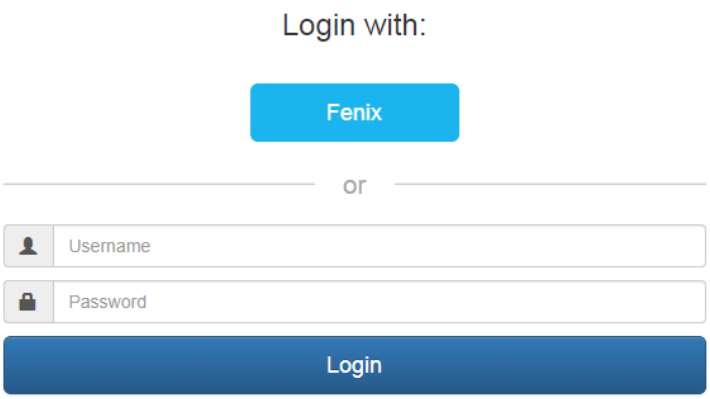

Figure 3.5: Login page

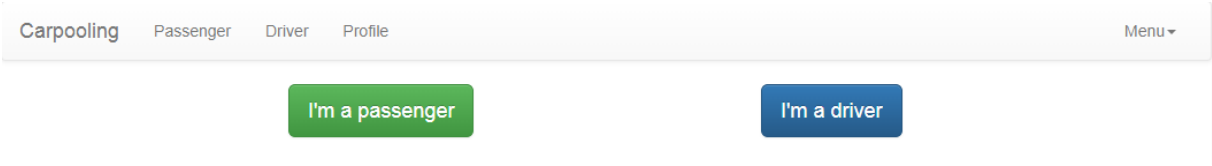

Figure 3.6: Home page

The first time a user enters the system, he is presented with the interface depicted in Figure 3.6, which is designed to help guiding new users through the system. In this interface the two main roles of our system are represented by two buttons:

- I'm a passenger - redirects the user to the "Find trip" page
- I'm a driver - redirects the user to the "Create trip" page

We chose to adopt this feature from the SINGU Car Pooling which we did not have at an early stage, but after some initial acceptance testing we found out that our system needed this feature.

In the following sections we detail the functionality provided by all menus in the application.

3.1.1 Profile

In the Profile menu the user can choose between three options: Notifications, Edit profile and Public profile.

3.1.1.1 Edit profile

This interface enables the user to update information about himself. The first time a user enters the system it creates a notification for him to choose if most often he is going to be a driver or a passenger, which takes him to this "Edit Profile" page, in an attempt for him to also fill the remaining information that

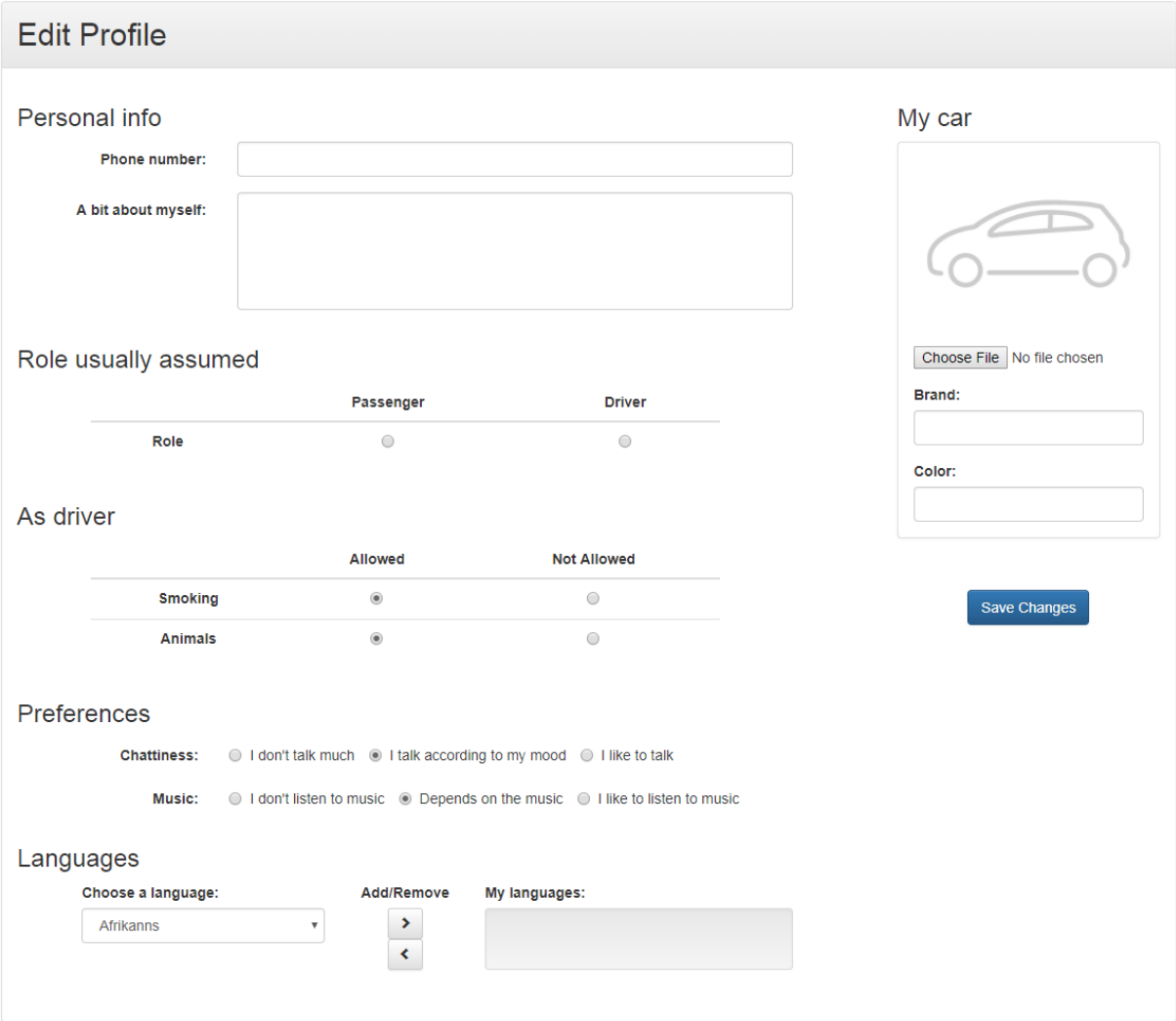

Figure 3.7: Edit profile interface

is not imported from Fénix, to achieve a more complete profile (which is optional). This page is depicted in Figure 3.7 and is organized in various sections.

Personal information

In this field the user can fill in his phone number and write a short text about himself.

Role usually assumed

In this field the user can specify his usual role, which will determine the first page that will be presented to that user in subsequent logins. Three cases can be considered, depending on what the user selects:

- Passenger: means the user will probably login to find trips where he can go as a passenger more

than any other thing, so this option sets the first page to the ”Find trip” page of the Passenger menu of our system

- Driver: means the user will probably login to create trips and check for passengers for those trips more than any other thing, so this option sets the first page to the ”Create trip” page of the Driver menu of our system
- Choose on login: this is the option by default in case the user does not select any other, and in this case the first page stays the same, which is the ”Home” page

Preferences as driver

In this field the user has to select if as a driver he will allow or not the preferences listed there, these choices will only serve as a default when creating a trip, to help drivers that will stick with the same preferences for almost all trips they create. The drivers can still chose a different option when creating a trip if they want.

Generic preferences

These are generic preferences for both passenger and driver, they are not used for any matching purposes, just to help giving someone that is inspecting that profile a bit more information about the user.

Languages

In this field the user can select which languages they speak and add them to their list of spoken languages. Each added language can also be removed.

My car

In this field the user can upload an image of their car as well as write its brand and color.

3.1.1.2 Public Profile

The purpose of this interface, depicted in Figure 3.8, is to show the user what their profile looks like when viewed by other users. In this profile is displayed the name, age, gender and email which can not be edited. It is also displayed everything that can be edited in the ”Edit profile” page with the addition of the average rating of that user and the comments, which are both given by other users (passengers) upon completing a trip where the user was a driver.

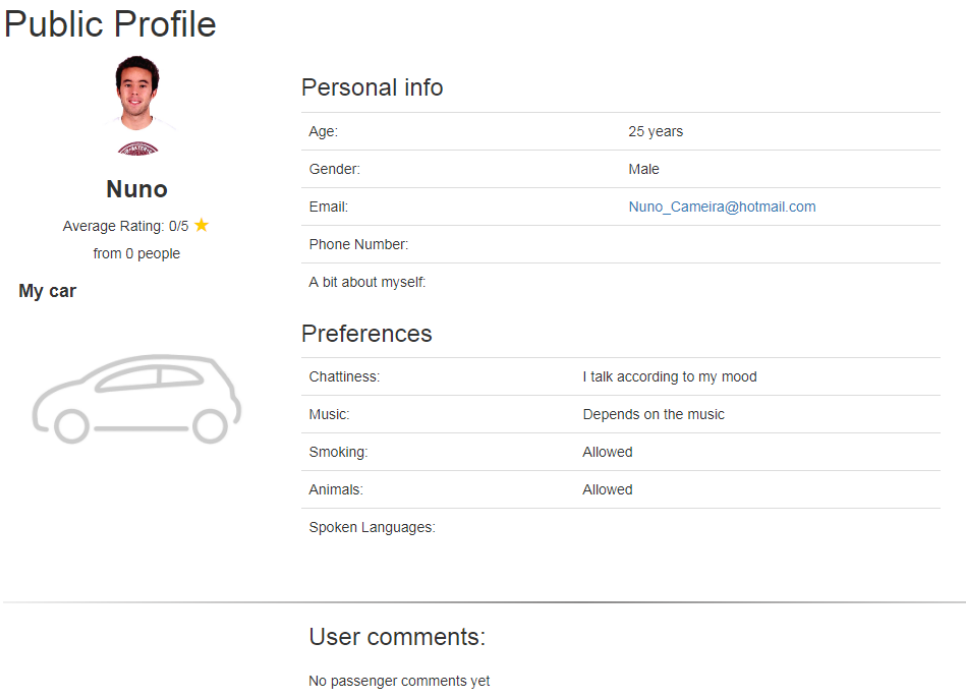

Figure 3.8: Public profile interface

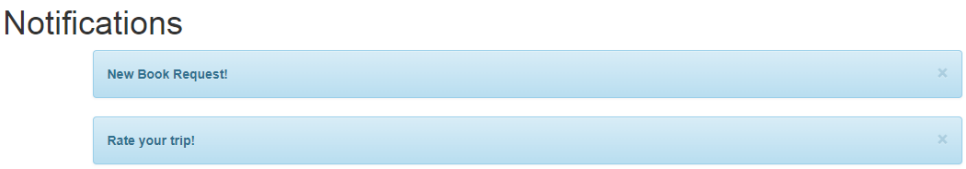

Figure 3.9: Notifications interface

3.1.1.3 Notifications

The purpose of this interface is to inform the user of new events that happened, for example, about a new booking request which is available or a trip that needs to be rated. An example of this is illustrated on Figure 3.9. The user just needs to click on the notification and he is redirected to that event, in this case to accept or reject the booking request, and to the trip interface to be rated.

3.1.2 Passenger

In the Passenger menu the user can choose between three options: ”Find trip”, ”Requests” and ”My trips”. We have talked about how the user could change the first page that is presented after the login, one of the options was the “Find trip” page which will we will explain bellow.

3.1.2.1 Find Trip

This interface allows the user to search for trips in our system and is depicted in Figure 3.10. The user can select origin (pick-up) and destination (drop-off) points. The Google maps auto-complete functionality is provided to help the user in choosing real locations for this. The departure date that can be either written in the appropriate format or chosen from a widget to help that process. There is also some optional preferences the user can specify for a more rigorous search (e.g. trips that allow smoking). We provide a “Preview” button that displays the trip path on the map after having selected both origin and destination.

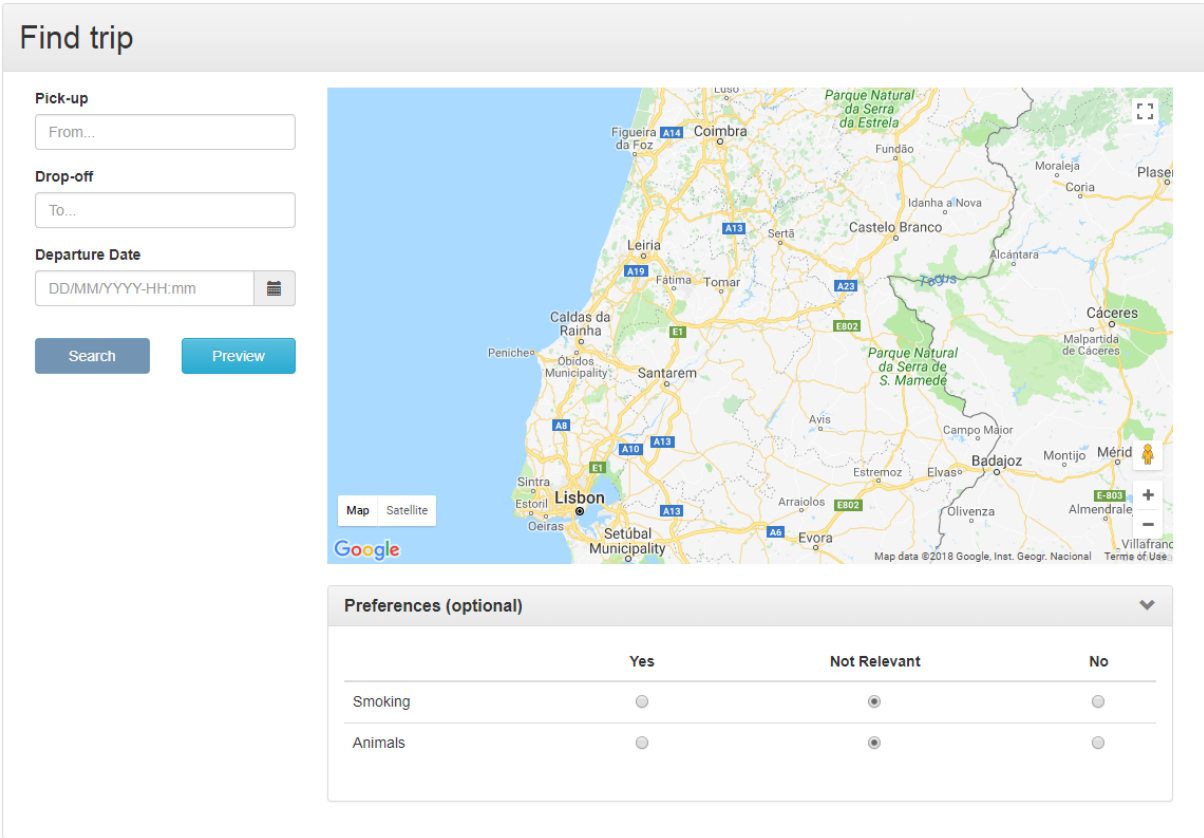

Figure 3.10: Find trip interface

When the user has filled all the necessary fields he can press the “Search” button to query the system for all matching trips and the result will be a list of trips organized by relevance (starting with the trips with departure date closer to the one asked, followed by all other on that same day), illustrated in Figure 3.11. Each trip has information about the driver (name and age), the origin, the destination the duration, the vehicle preferences, how many seats are left and the price per seat. The user can then select on one of the listed trips to view more details, either by clicking on the “book seats” button or by clicking

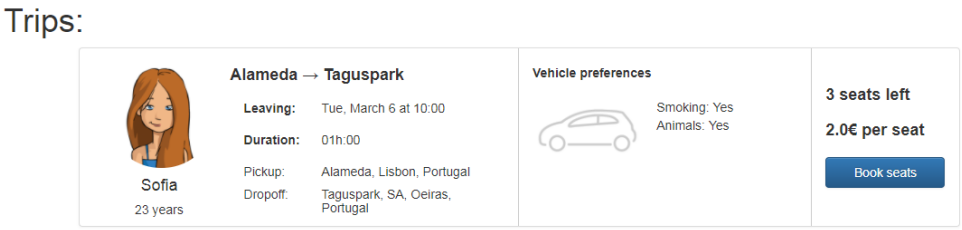

Figure 3.11: Find trip interface after searching

anywhere inside the trip box. When viewing the details of a trip, in addition to the information already presented in the previous interface it is also displayed the passengers that will go on this trip as well as a button to make a request to book a number of seats in the trip. This detailed trip view is depicted in Figure 3.12.

Figure 3.12: Find trip interface after clicking on a trip

3.1.2.2 Requests

If a user chooses to issue a booking request, as explained before he will get a visual confirmation that the request was created (or a warning in case the system cannot create the booking request) and he can then go to the tab "Requests" and see it. Here the user can choose to withdraw his booking request as depicted in Figure 3.13.

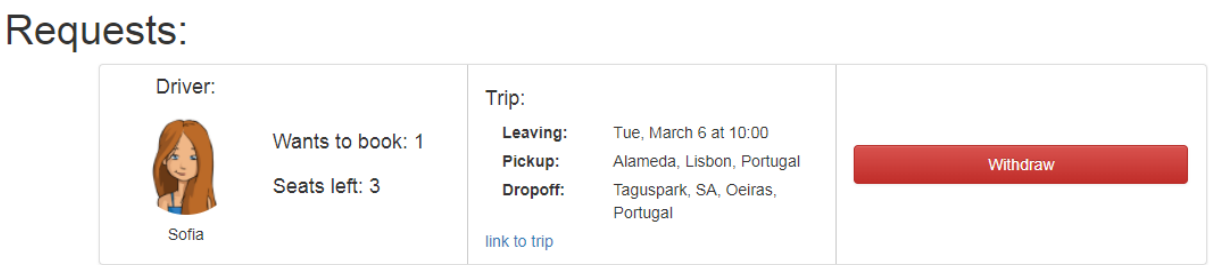

Figure 3.13: Passenger requests interface

3.1.2.3 My trips

After having issued a booking request, as a passenger, the user can then check if it was accepted by going to the “My trips” tab, which shows trips where the booking request was accepted by the driver but which did not take place yet, and the old trips which are trips that already happened. The user can click on any trip to view their details. For old trips, a new interface will appear for the user to rate them. To rate the user can select a number of stars (from 0 to 5) and write a comment before submitting the rate, as depicted in Figure 3.14.

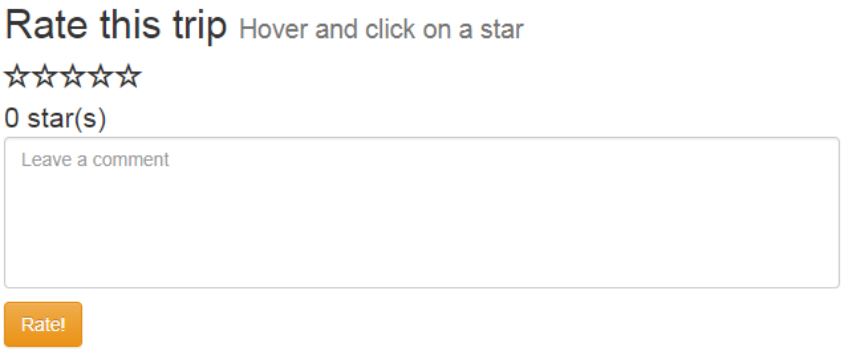

Figure 3.14: Rating menu

3.1.3 Driver

In the Driver menu the user can choose between three options: Create trip, Requests and My trips. We have already discussed about how the user could change the first interface that is presented after the login, one of those options was the “Create trip” interface which will we will explain bellow.

3.1.3.1 Create Trip

This interface allows the user to create new trips, as depicted in Figure 3.15, after filling the following fields:

- Pick-up (origin);
- Drop-off (destination);
- Departure date (DD/MM/YYYY-HH:mm);
- Duration (HH:mm);
- Price (€);
- Number of seats;
- Passengers allowed (the type of passengers allowed in this trip, e.g. Student);

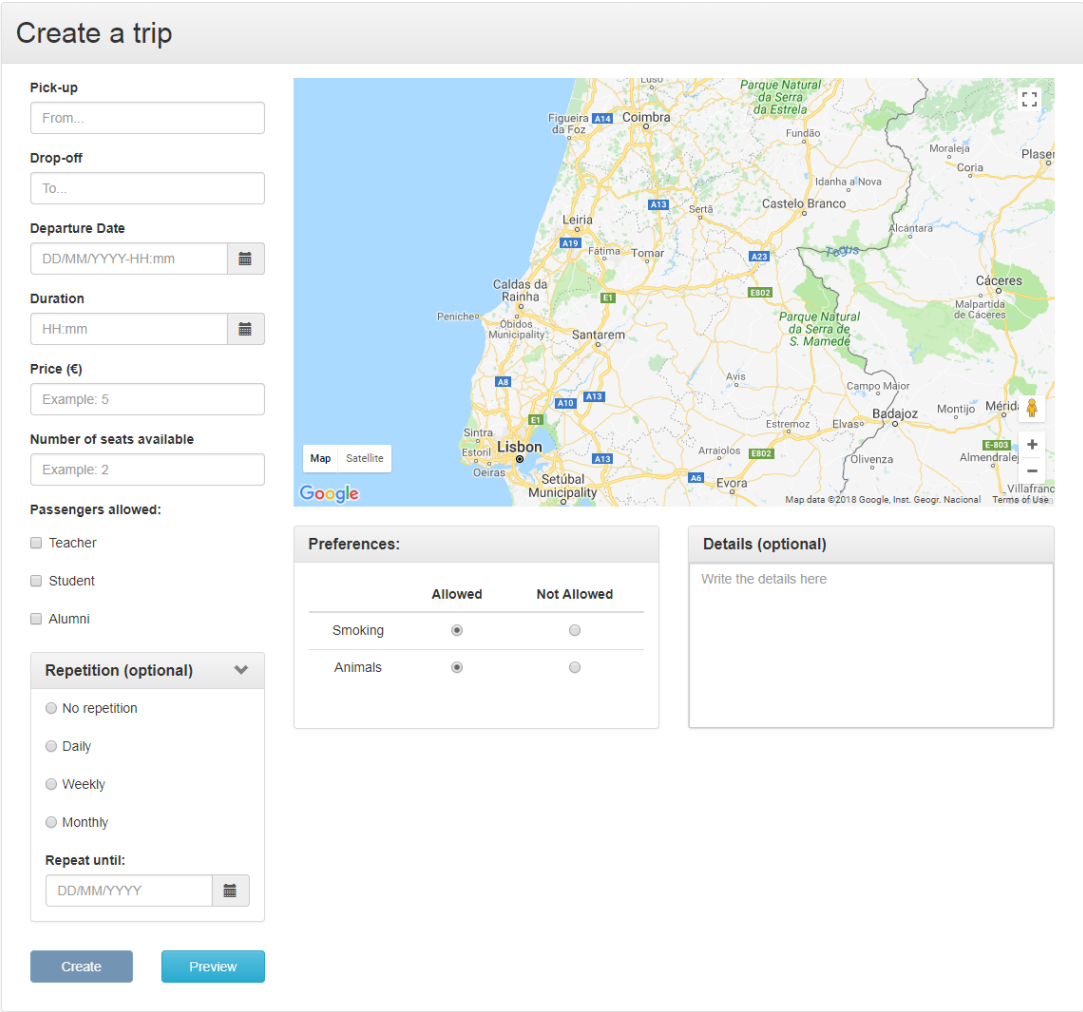

Figure 3.15: Create a trip interface

- Preferences (these preferences are read from the user profile but the user can change them at will, e.g. Smoking, Animals);
- Details (this is an optional field where the user can write some more details about the trip);
- Repetition (this is an optional field that allows the user to create trips with repetition starting on the date selected as “Departure Date” and ending on the date selected on “Repeat until”, e.g. Daily, Weekly, Monthly).

A “Preview” button is also provided that displays the trip path on the map, after both origin and destination have been selected. When all the desired information is filled the user can press the “Create” button to make the trip available in the system. This action is followed by with an alert informing whether the system was able to create the trip successfully or not.

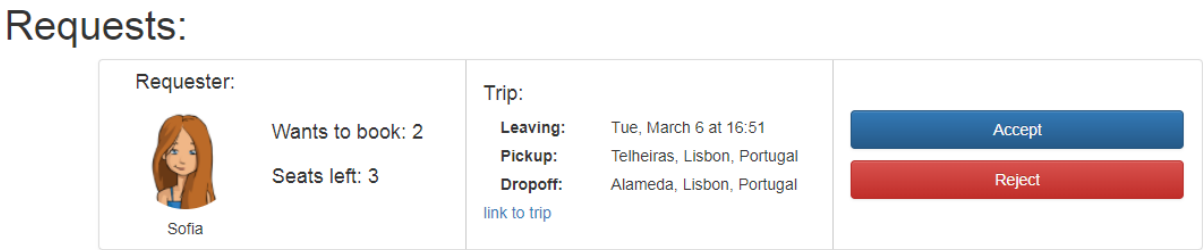

Figure 3.16: Driver Requests interface

3.1.3.2 Requests

If a passenger submits a booking request for a driver's trip, that driver can view it in the interface "Requests" of the Driver menu, as depicted in Figure 3.16. These requests have the following information:

- Who made the request;
- How many seats the passenger wants to book;
- How many seats are left on the trip;
- The departure date of the trip;
- The pick-up and drop-off locations;
- A link to view that trip with more details.

With this information the driver can then decide whether to accept or reject each request.

When viewing the details a trip where you are the driver the option to book seats will not appear as the driver cannot book seats in his own trip, as well as the menu to rate the trip because only passengers can rate trips. However, in order to help the driver to be more aware of the pending booking requests, on this interface the driver can also see if there are any booking requests for that particular trip, as depicted in Figure 3.17

3.1.3.3 My trips

In this interface the user can see every trip he has created as a driver, current trips (that did not happen yet) and old trips (that already happened), with the possibility of deleting current trips, as well as clicking on them to view their details, as depicted in Figure 3.18.

3.1.4 Administrator

This interface, depicted in Figure 3.19, is only accessible by an Administrator, since it allows the modification important aspects of the operation of the system with the following actions:

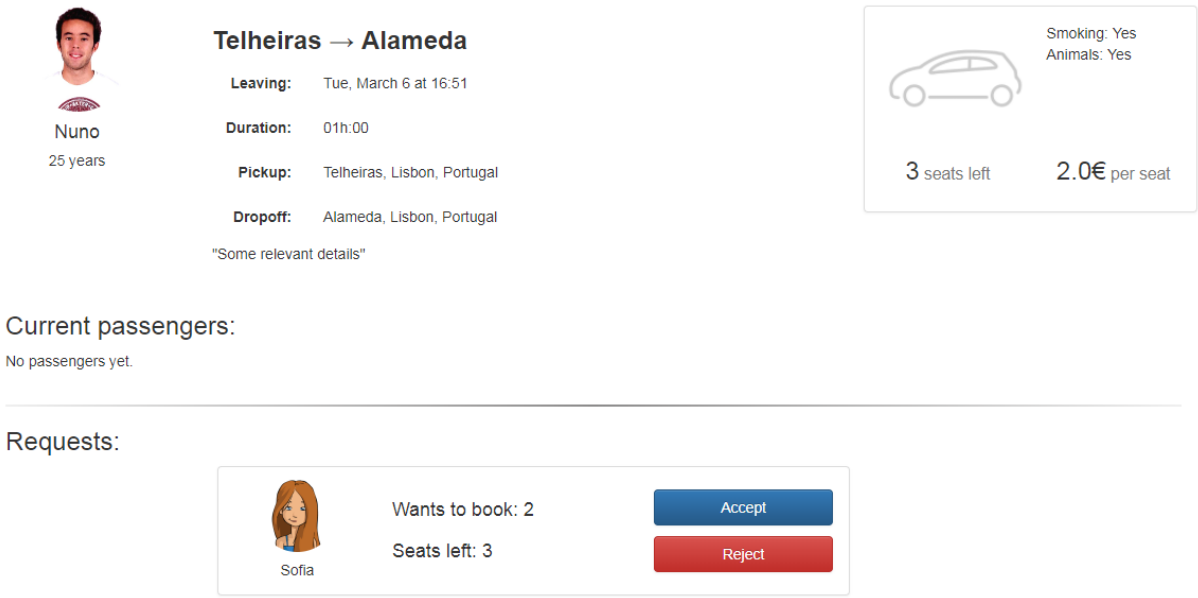

Figure 3.17: View Trip interface

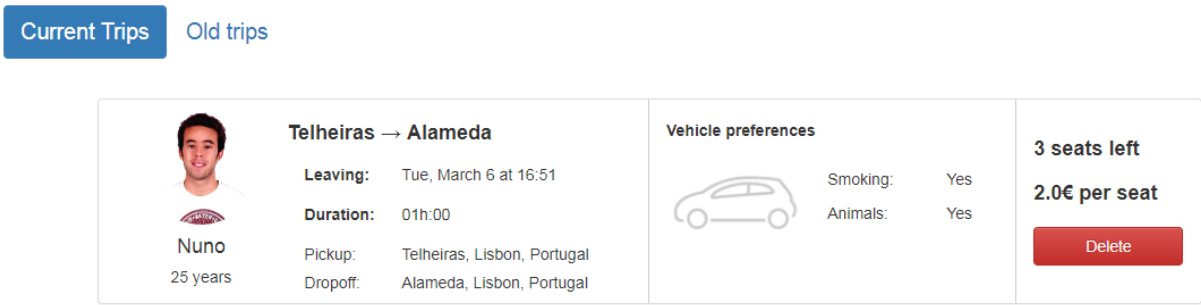

Figure 3.18: Driver My trips interface

- Add/remove preferences: these preferences can be seen/edited on the user profile, e.g. ”smoking” and ”animals”;
- Add/remove roles: these roles are the roles that are matched with the roles that are extracted when a user authenticates himself;
- Add root permissions: adds the role of ”root” to a user so that he can perform administrator operations.

3.2 Mobility Portal

The Mobility Portal targets organizations where, usually, there is a good level of trust between users. It leverages all available transportation methods of such organization by creating plugins able to extract

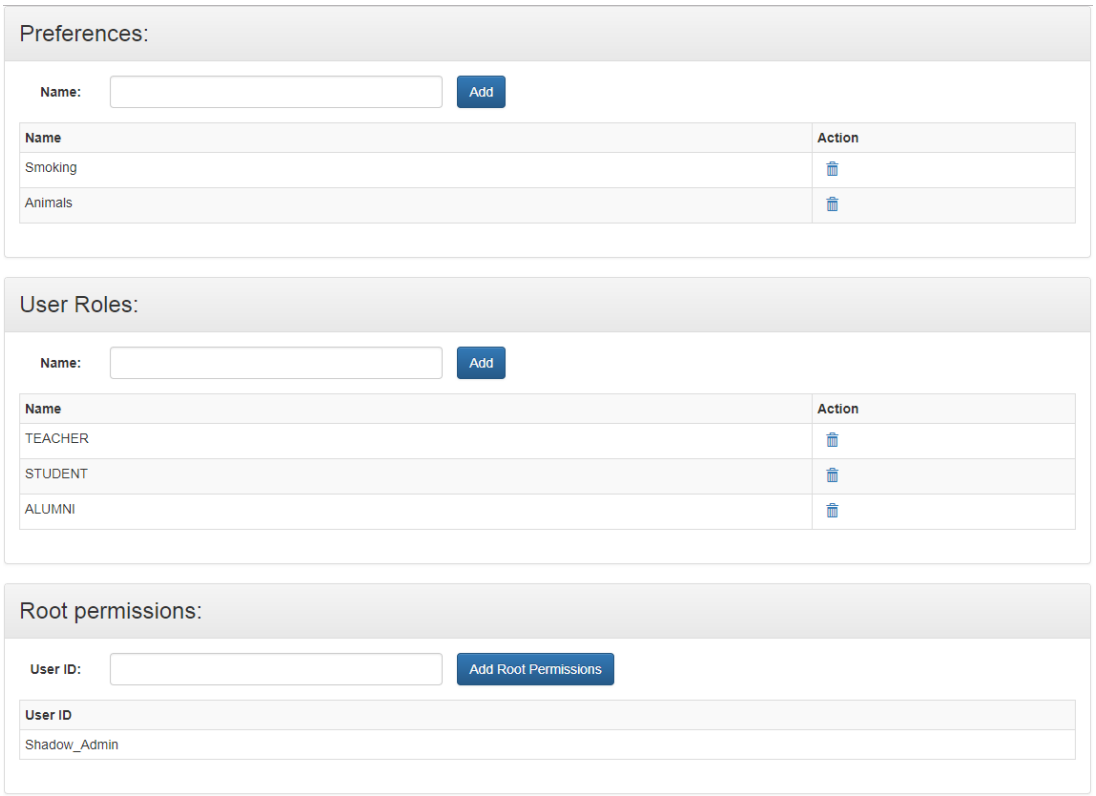

Figure 3.19: Administrator page

routes with schedules from the systems that handle those transportation methods. The main goal of the Mobility Portal is to allow a user to submit a request to search for trips from an origin point to a destination point with a certain departure date. This request is then process by the system which will use every available plugin to obtain possible routes that fulfill the request. The routes obtained may not fulfill the user request, in that case the system tries to find additional routes that can connect to the ones already received (e.g. walking 500m to be able to take the carpool).

We analyzed how the Mobility Portal could communicate with external services to get route information and their schedules. In order to have a modular solution which was a very important requirement to be considered in our system's design, we decided that our system would communicate with external services through plugins. This plugin architecture enables our system to be easily extended to collect information from many different services by adding new plugins that can query those services. The main external service that the Mobility Portal queries is our Carpooling service, although in the future more options can be added, such as a bike-sharing services and the IST Shuttle (which handles transportation between campi, Alameda and Taguspark).

We then moved to the algorithmic part of our system, more precisely to how our system will compute a result for a given input. After reviewing how related approaches handled this part we decided to implement the logic in the following way:

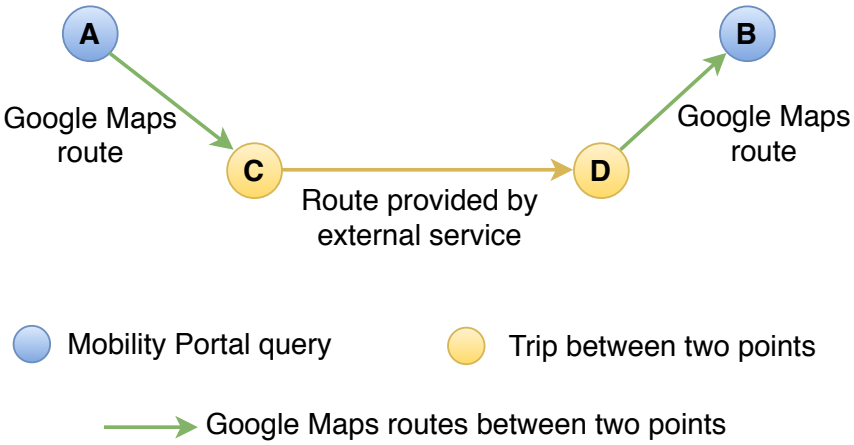

Figure 3.20: Mobility Portal logic example

- To a user input to search for trips (origin, destination and departure date) the system takes the departure date and queries every available system through its plugins to get every trip on that day;
- If after asking all plugins for information the system does not get any results, it will delegate that task to a Google Maps Plugin. This Plugin is the last resort to complete what is left of our current task;
- Else the system got some results, in case the origin and destination of those trips are a direct match with the ones given by the user the system does not need to do any more processing. Otherwise the system takes the points that do not match (origin and/or destination) and tries to connect them to the ones given by the user with routes from Google Maps.

An example of this is depicted in Figure 3.20.

We also analyzed the best way to present the results of a query to the user, such that he can easily understand them. Since the results are mainly routes and schedules we believe that the best way to understand them is with a list of possible routes and their respective schedules, from which the user can then select one and see it on a map. He can also follow a link present on each trip to book them.

We designed a simple interface for the Mobility Portal, which is depicted in Figure 3.21. Just like the Carpooling interface, we offer the Google maps auto-complete functionality to help choosing real locations for the origin (pick-up) and destination (drop-off) points, as well as the departure date that can be either written or chosen from a widget.

After the user has filled all the required parameters, he can then press the ”Search” button to submit the request. The results will then be displayed on a list below, as depicted in Figure 3.22. These results are organized by:

- Name of the service that answered with available routes;

Figure 3.21: Mobility Portal interface before a query

- List of answers, which can be just one trip, or a series of trips that connect each other in a way that there is one trip that starts on the desired origin and one that ends on the desired destination.

Each trip of that list can be clicked which will display the route on the map. There is also a link to that trip which will redirect the user to the information of that trip on the respective service (e.g. go to the Carpooling ”View trip” interface which allows the submission of booking requests).

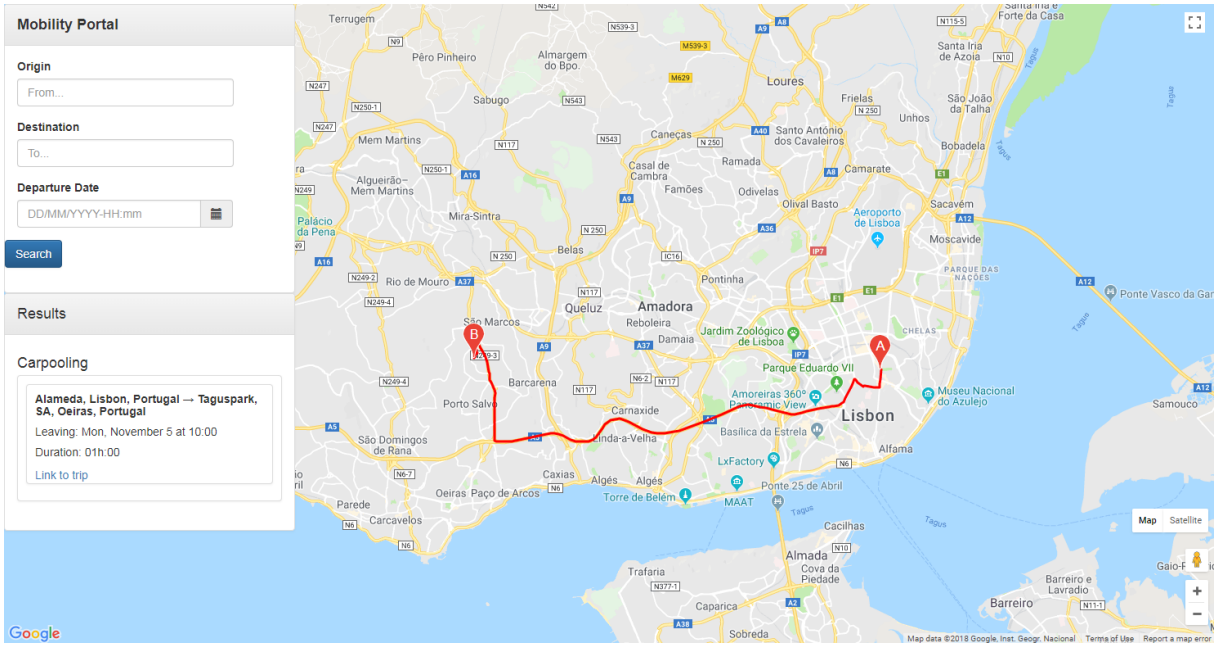

Figure 3.22: Mobility Portal interface after a query

4

Implementation

Contents

In this chapter we detail the most relevant aspects pertaining the implementation of both the Carpooling and the Mobility Portal systems. We start by describing the architecture of each system and will follow up with the description of their modules.

4.1 Carpooling

The main software modules of the carpooling system, as well as their most relevant interactions, are illustrated in Figure 4.1.

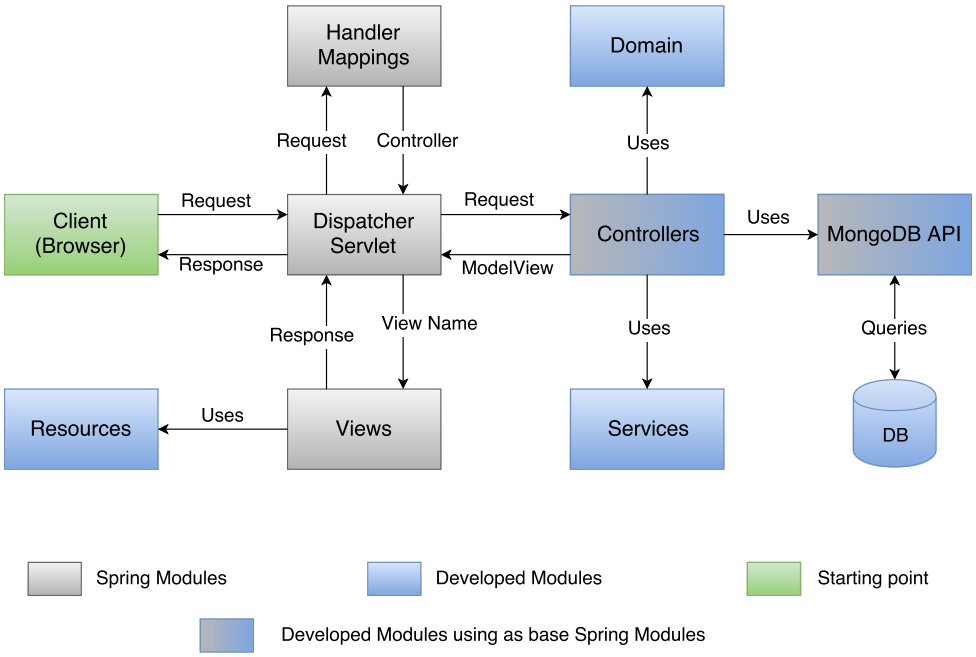

Figure 4.1: Carpooling system architecture

The grey modules are already defined/implemented by Spring and are used by the Spring engine without any need for customization. The blue modules were either developed or customized in the context of this work. Although the Controllers and the MongoDB API are also part of Spring they needed to be customized so that they could use the other developed modules (Domain, Services and the Database).

The Carpooling system is implemented in Java using Spring MVC (introduced in Section 2.2.2) which handles HTTP requests and returns the appropriate HTML content. Before this content is returned to the user (by Spring MVC) we use Thymeleaf to perform a server-side rendering of the HTML which enables us to easily fill an HTML page with dynamic content from the server. After all the dynamic content is inserted, Thymeleaf returns the HTML to the Spring MVC control which then returns it to the user (more precisely, to its browser).

The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements:

- The **Model** which encapsulates the application data;
- The **View** which is responsible for rendering the Model data and, in general, generate HTML output that the client (browser) can interpret;
- The **Controller** which is responsible for processing user requests, building an appropriate Model and pass it to the View for rendering.

Recapping what was introduced in Section 2.2.2 we describe the Spring MVC modules of the Carpooling system architecture:

- **DispatcherServlet** is the front controller of the framework and is responsible for delegating control to the various modules during the execution phases of an HTTP request;
- **HandlerMapping** selects which Controller handles each incoming request;
- **Controller** interfaces between the Model and the View to manage incoming requests, gets processed data from the Model and advances that data to the View for rendering;
- **Views** encapsulate both the **ViewResolver** and the **View**
  - **ViewResolver** selects a View based on a logical name for the view;
  - **View** is responsible for returning a response to the client. Some requests may go straight to view without going to the model part; others may go through all three.
- **MongoDB API** provides an interface for querying the database;
- **Domain** defines the system classes and what information they hold in a structured fashion;
- **Services** have the logic that is not handled by the Controllers;
- **Resources** is responsible for the templates used to build the HTML response to the user;
- **DB** is the database that will store the system data.

The typical flow of information would be starting on the module **Client (Browser)** which is where the HTTP requests that are sent to our system are generated as a result of the interaction of the user with the interfaces provided by the system. The **Dispatcher Servlet** and the **Handler Mappings** are handled by the Spring engine, and their function is to find the adequate Controller to process the current request. The **Controllers** then use several modules like the **MongoDB API** to query the **Database** to get or save **Domain** objects. The **Services** to use their logic in order to process the request. The **Views** module will

then find the requested HTML page on the **Resources** module, perform the necessary rendering of the HTML with Thymeleaf and return it to the Client.

The Carpooling system has two different types of communication, server-server and server-client(browser):

- **Server-server**: this type of communication is conducted between our server and an external server (e.g. Mobility Portal), since the data will only be used by servers it does not need to be presented with a user-friendly interface, thus the communication is done with a basic REST API (Spring) and the information is encoded in JSON format/XML, which is a structured human-readable format, but not very friendly for the user;
- **Server-client(browser)**: this type of communication is done between our server and a browser, since here there is user interaction the data needs to be presented in a user-friendly interface, thus the communication is done using Spring MVC and Thymeleaf.

The key difference between a traditional Spring MVC controller (server-client) and a RESTful web service controller (server-server) is the way the HTTP response body is created. While the traditional MVC controller relies on the View technology, the RESTful web service controller simply returns the object and the object data, which is written directly to the HTTP response as JSON/XML.

The remainder of this chapter is dedicated to describe the following modules, which were either developed or customized in the context of this work:

- Domain;
- Controllers;
- Services;
- Database (MongoDB API);
- Resources.

4.1.1 Domain

This module defines the classes (without logic) needed to store all necessary information in an organized way, as depicted in Figure 4.2. We are going to describe some of the most relevant entities of the system and match them with the corresponding implementation in the Domain:

User - Someone who uses the system, implemented with the class ”User” that models all its relevant properties in the context of the Carpooling system.

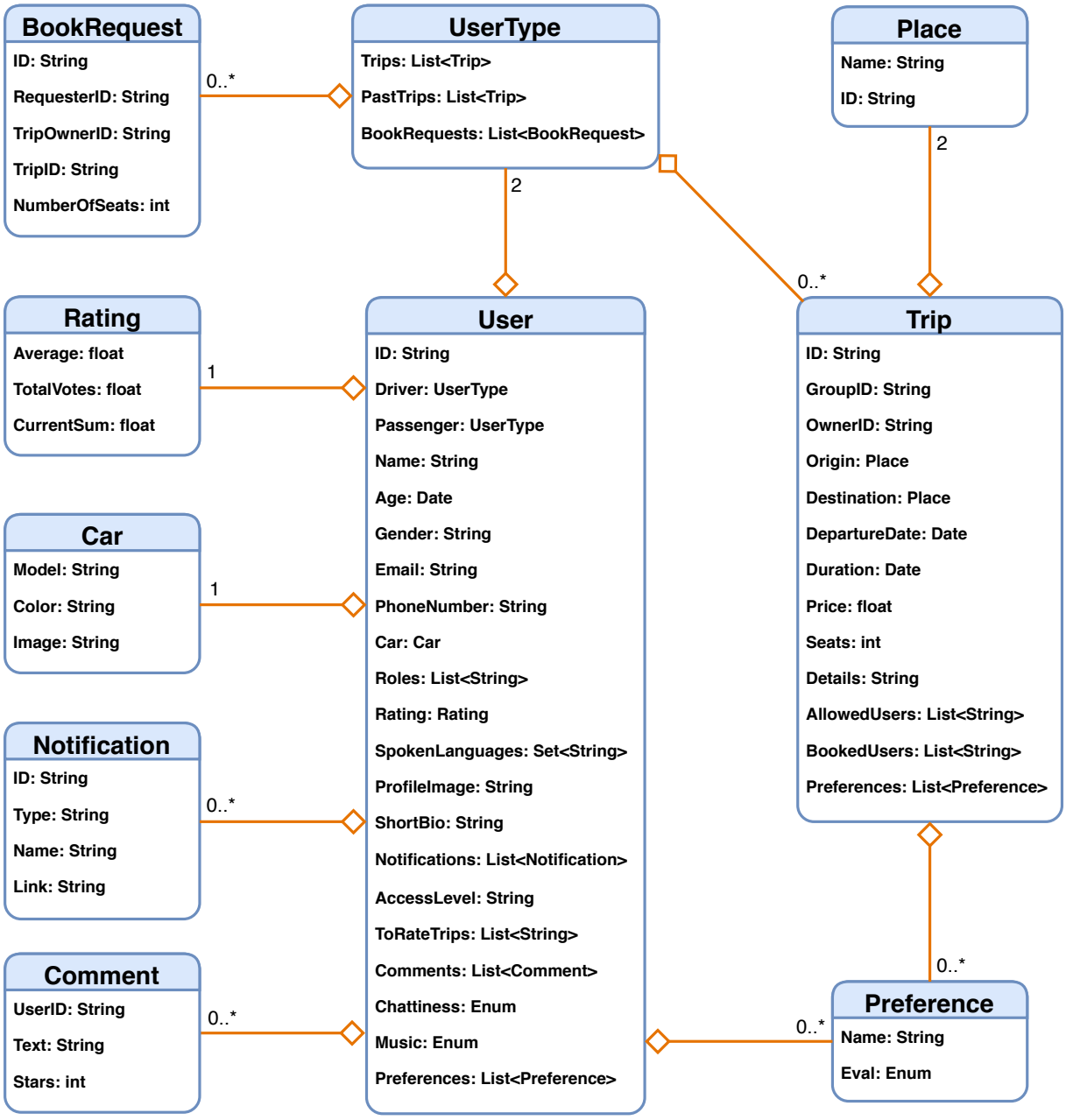

Figure 4.2: Carpooling UML

Trip - A route between two points with a departure date, duration, price, number of seats and preferences, implemented with the class ”Trip” that models all its relevant properties in the context of the Carpooling system.

Role - A user type in a particular organization, implemented with the class ”Role” composed by a single field called name, which is an unique identifier (e.g. TEACHER, STUDENT, ALUMNI). This is simplified in the UML diagram by the User attribute ”Roles” which is a list of Role (simplified as just a list of ”String”).

Access Level - Describes the clearance of a user (the permissions that a user has when interacting with the system), implemented by the attribute ”AccessLevel” in the class ”User”. Currently the system has two types of access levels:

- User;
- Root (Administrator).

Fixed preferences - These are preferences that are not modifiable without changing the code, their purpose is only to inform other users when viewing a profile of a user with these preferences. Modeled by these two attributes in the class ”User”:

- Chattiness (describes the willingness to talk);
- Music (describes the willingness to listen to music).

Dynamic preferences - Unlike the fixed preferences, these are used to perform matching when searching for a trip as a passenger. Modeled by the class ”Preference”, which contains two fields:

- Name;
- Eval.

The names of the dynamic preferences come from a database (PreferenceRepository) which stores only unique names of preferences. Every name on that repository must be on the list of dynamic preferences of a user.

Eval is a set of fixed values (YES, NOT RELEVANT, NO) that are used to compare preferences on the matching process when finding a trip, as well as to define default values to make it easier to create a trip (these default values can only be YES or NO). We defined two dynamic preferences for our system:

- Smoking;
- Animals.

An example would be:

- PreferenceRepository [Smoking, Animals];
- UserA dynamic preferences [[Smoking, YES], [Animals, NO]].

Notification - Used when the systems needs to notify a user about something that happened in the system (e.g. ”Rate the trip you went on”), modeled with the class ”Notification”.

Comment - Consists of a number of stars and a text, which are used to rate a trip that a user went on as a passenger, modeled with the class ”Comment”.

Rating - The average stars (from one to five) that other passengers rated a particular driver with, modeled with the class ”Rating”.

User type - The Carpooling system requires a user to be aware of his trips, past trips and booking requests both as a driver and as a passenger, modeled with the class ”UserType”.

Booking request - When a user issues a booking request to a driver, modeled with the class ”BookRequest”.

Place - Defines a place on a map, modeled with the class ”Place” with the full name of that place and the corresponding ID (taken from the Google Maps Places API introduced in Section 2.2.4).

Car - Describes a car, its model, color and picture, modeled with the class ”Car”.

4.1.2 Database

The Carpooling system uses a set of different repositories, which are a Spring abstraction to significantly reduce the effort to implement data access layers for various persistence stores. Such repositories are implemented using MongoDB (as discussed in Section 2.2.6). In order to achieve this we used Spring Data MongoDB [24], which focuses on automating data storage and query in MongoDB. As a consequence, no queries need to be written in the MongoDB language. All that needs to be done is to write handful of methods in Java and the necessary queries are automatically generated.

By using an interface with the MongoDB API for Spring we create an abstraction level that allows us to be able to switch the database type (to MySQL for example), with just minor changes to the code. The main reasons for choosing MongoDB over MySQL were:

- By default it prefers high insert rate over transaction safety;
- We do not need complex transactions;

- It provides better availability in an unreliable environment;
- Scales horizontally (which means that the scaling is done by adding more machines into the pool of resources, whereas vertical scaling is done by adding more power (CPU, RAM) to an existing machine).

The system does not need much safety on transactions. The only time we would need to be more careful with transaction safety is if a driver could somehow accept two booking requests for the same trip at the same time causing him to overbook it. This should never happen but if by some case it does happen we created a custom repository with a custom method that locks the seats of a trip until the processing of booking requests is finished, thus protecting our system against this overbooking scenario.

The repositories used in our system are the following:

- UserRepository: stores users;
- TripRepository: stores trips that did not happen yet, and some that might have already happened;
- PastTripRepository: stores trips that have already happened;
- PreferenceRepository: stores dynamic preferences;
- RoleRepository: stores the various roles that a user can have.

By having two different repositories to store trips we are able to improve the performance of our system, since when we need to find a trip for a passenger we only search on the TripRepository. Thus sparing our system from having to go through all the past trips.

In order to have a better understanding of how the repositories are created and handled let us take our UserRepository as an example, as shown in Listing 4.1.

```
public interface UserRepository extends MongoRepository<User, String> {

    User findByName(String name);

    User findByID(String ID);

}
```

Listing 4.1: UserRepository

What was done here was create an interface, in this case “UserRepository ” that extends the MongoRepository interface which out-of-the-box already provides many operations, including standard CRUD

operations (Create-Read-Update-Delete, which are the four basic functions of persistent storage). Then we define the information we want to store on the map, which will be the set of Users and associate them with a key which will be a String. After this all that is left is to define method signatures (name and arguments). The name of a method is very important since it is what allows Spring to automagically generate the queries. Signatures have a specific pattern that must be followed. There are several keywords that can be used for the method name that will result in different queries to find the desired object (in this example a User). Some of these keywords and examples of method names using them are the following:

- **FindByX**: e.g. `findByName("SomeName")` which will return Users that match the given field, in this case Users with the same name;
- **StartingWith**: e.g. `findByNameStartingWith("SomeName")` which will return Users with names that start with the one given;
- **EndingWith**: e.g. `findByNameEndingWith("SomeName")` which will return Users with names that end with the one given;
- **Between**: e.g. `findByAgeBetween(18, 30)` which will return Users with age between 18 and 30;
- **Like** and **OrderBy**: e.g. `findByNameLikeOrderByAgeAsc("SomeName")` which will return Users that have names containing the on given, ordered by age, in ascending order.

In case we need a more complex query that cannot be written using these keywords we need to define a custom repository and build our own query using the Query and Criteria classes which very closely mirror native operators. This was the case of our "TripRepositoryImpl", shown in Listing 4.2 where we defined a function "reserveSeatsOnTrip" and implemented the query ourselves since it was a more complex query.

```
1 public class TripRepositoryImpl implements TripRepositoryCustom {
2
3     @Autowired
4     private MongoTemplate mongoTemplate;
5
6     @Override
7     public boolean reserveSeatsOnTrip(String tripID, int seats) {
8
9         Query query = new Query();
```

```
        query.addCriteria(Criteria.where("ID").is(tripID).andOperator((
Criteria.where("seats")).gte(seats)));

        Update update = new Update();
        update.inc("seats", -seats);

        Trip trip = mongoTemplate.findAndModify(query, update, new
FindAndModifyOptions().returnNew(true), Trip.class);

        return trip != null;

    }
}
```

Listing 4.2: TripRepositoryImpl

With this we build our own query to select the trip where the ID is the given tripID and where the trip seats are greater than the given seats. Then we build the update we want to do if we find it, which is to decrease the number of seats on that trip. After that we call the `findAndModify` to apply the provided `Update` on documents matching `Criteria` of given `Query`.

Dynamic Preferences

These preferences, as previously explained are stored in the PreferenceRepository. Every time our system needs to expose dynamic preferences to a user it asks the PreferenceRepository for all the preferences available and uses them. This allowed us to implement functions to add and remove preferences from the repository, (which can only be used by administrators) either of these actions will result in a loop through every user in the UserRepository in order to add or remove the preference. Processing the preferences like this enables our system to support runtime updates of the preference repository. The first time our system is run we setup our predefined dynamic preferences (Smoking and Animals).

4.1.3 Controllers

This module holds all the controllers needed to communicate with a user (back-end to front-end communication). The controllers we defined, illustrated in Figure 4.3, are the following:

- `PassengerController`: handles all the operations of the passenger menu (find a trip, request to book seats on a trip, view requests, cancel requests, view passenger trips, rate trips);

- `DriverController`: handles all the operations of the driver menu (create a trip, view book requests, accept/reject requests, view driver trips, delete a trip, delete a series of trips with the same frequency);
- `ProfileController`: handles all the operations of the profile menu (view notifications, view public profile, edit profile);
- `GenericController`: handles all the generic operations (login, maintenance);
- `SuperUserController`: handles all the operations that only administrators can do (add/remove preferences, add/remove roles, add administrator permissions to a user).

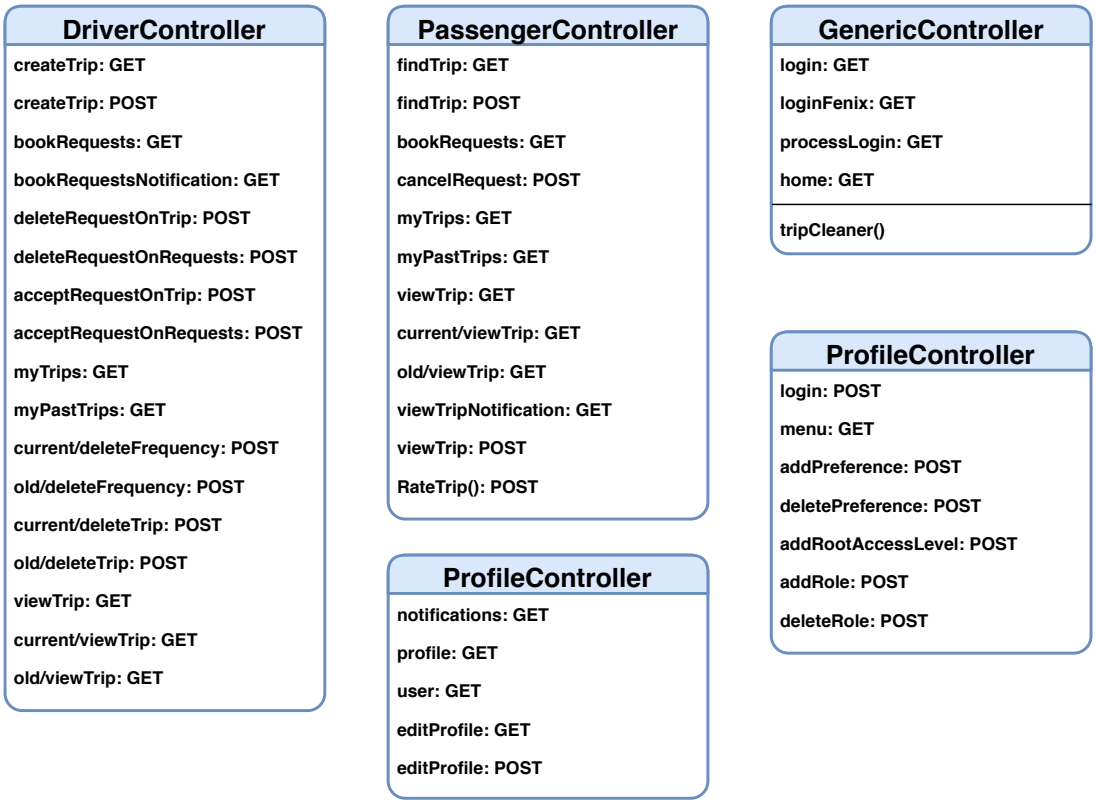

Figure 4.3: Carpooling Controllers

Most of such controllers use a specific annotation to define how the HTTP requests are mapped to each controller. For example our DriverController has the annotation `@RequestMapping(value = "/driver/*")`, which means that every request starting with `/driver/` will be mapped to this controller (e.g. `/driver/createTrip`). The methods of the `DriverController` are also annotated to identify if they will handle a HTTP GET or POST requests, like:

- `@GetMapping("/createTrip");`
- `@PostMapping("/createTrip").`

Finding a trip

The first step to find a trip is to take the user input and validate it, starting with the origin and destination locations. As explained on Chapter 3, these two fields should come from an auto-complete widget on the interface, however the user can chose to ignore this widget and write the location himself. To avoid ending up with places that do not exist, we take the origin and destination given by the user and do an auto-complete query on the server side to ensure that the location actually exists. This is performed using the Google Maps API, more precisely we used the `GeoApiContext` class to get the context and then we pass it to a `PlacesApi` autocomplete query to populate an array of `AutocompletePrediction`. This array contains the auto-complete predictions given by the Google Maps API, starting with the closest prediction to the user input. We then take the closest prediction (which should be the index zero of the array) for the origin and destination and use that instead of the user input, this way we ensure the places are more likely to exist. If, after this process, we still cannot find a place that matches the user input, an error message is presented to the user.

We can now move to the next step that we called “Matching”, which is the process of finding the trips that are the best matches for a particular passenger when he submits a request to find trips. The way this matching works is by getting all trips from the TripRepository and applying a series of filters to them. For this we use a class from the Services module called ”TripFinder” which will contain the list of trips and the desired filters that are to be applied to the list. Each Filter goes through the entire list of trips either removing or leaving them, according to the filter conditions. In the end this will result in a list with trips that fulfill every filter condition.

The first filter is the Place matching which will check if the origin and destination places of the trip that is being processed match the origin and destination given by the user. If they do not match, we remove that trip from the list and move to the next trip, otherwise we just move to the next trip.

The second filter is the User Role matching which will check if the passenger has any of the roles that are allowed on the trip. This means that if the passenger has at least one of the roles allowed on the trip, it is a match and the trip is kept on the list, otherwise that trip is removed from the list.

The third filter is the Preference filter which does the matching of the dynamic preferences of the trip and the passenger. In order to match these preferences we identified all the possible matching cases, illustrated in Figure 4.4, where the lines represent acceptable matches. Instead of doing all those comparisons to see if it is a match we do the opposite, which is to search for the cases that are not a match (which are less), as illustrated in Figure 4.5, and if we cannot find any it means that it’s a match. This results in the following:

- If the trip does not match the passenger we remove it from the list;
- Otherwise it means it is one of the cases from Figure 4.4, thus the trip matches the passenger and is kept on the list.

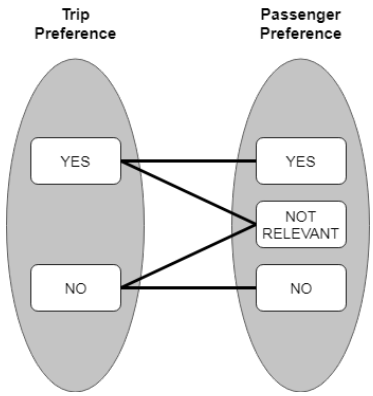

Figure 4.4: All the possible matching cases

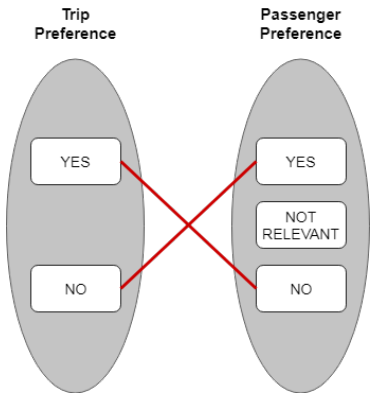

Figure 4.5: Cases where there is not a match

And finally the fourth filter is the DateRelevance filter which filters the trips by date and relevance. First, we remove every trip that is not on the same year, month and day as the departure date given by the user, which leaves us with all the trips on the same day. Then we check if the departure date of the trips are within a +/- 30 minutes window of the departure date given by the user. If they are within that window then we add them to a new list. By doing this we get all the trips that have the date closer to the one desired, then we take all the trips that are still on the same day but that were not within the time window and add them as well. This leaves us with a list of trips sorted by relevance which can now be returned to the user.

Creating a trip

As we have already discussed, to create new trips the user must be on the ”Driver” menu and on the “Create trip” sub-menu (discussed on Section 3.1.3.1). This interface is given to the client (browser) via HTTP GET when he clicks on the sub-menu. After all the necessary fields on the interface (origin, destination, departure date, duration, price, number of seats available, passengers allowed) are filled by the user and the “Create” button is clicked, an HTTP POST request will be sent to our system. We now

proceed explain how our system handles that POST request.

Once the POST request is received, we check if the user has selected at least one Role (e.g. STUDENT) to allow on the trip, if he did not then we can return immediately with an error message to the user saying that he needs to select at least one Role. If he did select at least one Role then we proceed to check if the origin and destination locations exist, just like we did on the "Find trip" procedure, already explained, to make sure that we have locations that exist. After that we take all the remaining user inputs and create the trip which we do not save to the database yet. Before that we check if the user selected a frequency which consists in one of the following patterns and an end date:

- Daily - repeat the trip every day;
- Weekly - repeat the trip on the same week day (e.g. Monday) every week;
- Monthly - repeat the trip every month on the same week of the month and the same day of the week, e.g. Start date: 29/01/2018; End date: 30/04/2018; Result: 29/01/2018, 26/02/2018, 26/03/2018, 30/04/2018.

If so we need to create additional trips with that frequency until the end date, otherwise we save the trip on the database, add the trip reference to the list of trips of the driver and save (update) that driver on the database as well. In case we need to create additional trips we get the desired frequency and, depending on that, we use the **Frequency** service of the Services module with a different recurrence pattern (plus one day, plus one week, plus one month). For this we must provide to the Frequency service the trip that was previously created, the start date, the end date, the duration and a groupID, which will then create the desired trips starting at the day given by the departure date until the end date given. Let us take the example of the `generateTripsEveryDay` function of this service, shown in Listing 4.3 and explain how it works.

```
1     public static List<Trip> generateTripsEveryDay(Trip trip, Date startDate,
2 Date endDate, Date duration, ObjectId groupID) {
3
4         List<Trip> tripList = new ArrayList<>();
5         Calendar calendar = Calendar.getInstance();
6
7         while (!startDate.after(endDate)) {
8             calendar.setTime(startDate);
9
10            Trip clonedTrip = trip.clone();
11            clonedTrip.setDepartureDate(startDate);
12            clonedTrip.setDuration(duration);
```

```
            clonedTrip.setGroupID(groupID);
            tripList.add(clonedTrip);

            startDate = DateUtil.addDays(startDate, 1);
            duration = DateUtil.addDays(duration, 1);

        }

        return tripList;
    }
```

Listing 4.3: generateTripsEveryDay function of Frequency service

This function will keep cloning the trip passed as the first argument, changing the departure date of the cloned trip according to the recurrence pattern (which in this case is adding one day), stamping it with a groupID and finally adding it to the result list until the desired end date is reached. It is not really necessary to update the duration since we only look at the hours when using it but in case we need it in the future to be on pair with the start date we updated it as well. All these trips are stamped with the same group identifier (although this groupID is different each time a user submits a request to create trips to our system) which allows us to easily find every instance of a recurrent trip, in case the driver wants to delete them all at the same time. We can then save the trips to the database and add the trip references to the list of trips of the driver, save and finally return to the user a message confirming that the trip(s) was(were) successfully created.

Requests

Creating booking requests

In order for a passenger to book seats on a trip he needs to issue a request to the driver stating how many seats he wants to book. If the booking request is created successfully it is saved in the passenger's request list and in the driver's request list. A notification is then generated so that the next time the driver from that trip logins into the system he is notified that he has a new booking request.

Deleting booking requests

There are two ways of deleting booking requests:

- The driver on his requests menu chooses to reject the request;
- The passenger on his requests menu chooses to withdraw his request.

Since the same request is linked to both the passenger and the driver (although exhibited in a different way in the interface) if one decides to delete it, it is also deleted from the other. Such an example

would be: the driver rejects (deletes) the request, then the system finds the passenger that made that request and deletes the request from that passenger also.

Notifications

On the profile menu and notifications sub-menu a user can see his notifications. A notification is just and alert box with some text (e.g. ”Rate your trip!”) and a link which is not visible, if the user clicks on the notification he will be redirected to this link (e.g. ”View trip” interface which will have the menu to rate that trip) and the notification is then deleted. The first time a user enters the Carpooling system we generate a notification for him to choose if most often he is going to be a driver or a passenger. In case the user decides to choose either driver or passenger this will define the initial page that will be presented to him in the next time he logs into the system, which in the long run saves a lot of clicks on the navigation interface.

Maintenance

As any other software system there are maintenance tasks that need to be executed regularly. We implemented in the Carpooling system a periodic function, in the GreetingController, which runs once every 24h. This function goes through all the available trips in the tripRepository and checks if each trip has already happened, if so such trips are moved to the repository for past trips. During this process we also need to update the trip references on the users (driver and passengers of the trips that already happened) and remove pending booking requests. Finally we create a notification for every passenger of those trips saying that they can rate the trip.

4.1.4 Services

This module is divided into three different areas:

- The Data Objects used to transfer data between the back-end and the front-end;
- The logic that is not handled by the Controllers themselves;
- A REST controller that offers server-server communication.

The **Data Objects** are just plain information containers with minor logic, or most of the times, with no logic at all, that are used to do the front-end back-end communication. This creates and additional layer between the front-end and back-end, thus not exposing unnecessary information to the front-end, since most of the times the information that is to be displayed on the front-end is not all the information that is

available in the Domain. This also allows us to create additional information on the data objects that is only required to the front-end.

The Controllers usually handle all the **logic** but there were some cases where the same logic was used on multiple methods (either in the same Controller or other ones) so we extracted the repeated logic into this Services module in order to make it reusable and avoid code repetition. This module stores all the major logic of the system. We have the **TripFinder** and some filters that can be used by it (DateRelevanceFilter, PlaceFilter, PreferenceFilter, RoleFilter) which were already explained before, there is also a RoleComparator which is used by the RoleFilter.

Another service is the **RequestRemoval** which offers the following functionalities:

- Find every booking request associated with a given trip ID on the list of a given driver;
- Find a booking request with a given ID on the list of a given passenger;
- Remove the trip with the given ID from the passenger trip list and add a notification;
- Remove the book request with the given ID from the passenger book request list and adds a notification.

Then we have the **Frequency** service which creates a set of trips depending on the pattern chosen (every day, every week, every month) which was already explained before. Although only being used by one Controller it is an important functionality of the system, thus being on this module.

The Services module also holds some authentication services, **AuthFenix** and **AuthFenixMap** that are used to extract information about an authenticated user. All the authentication services implement an interface called **Auth** which we defined with the following functionality:

- Get the unique username of the user;
- Get the name of the user;
- Get the birth date of the user;
- Get the gender of the user;
- Get the email of the user;
- Get the phone number of the user;
- Get the photo of the user;
- Get every role assigned to that user.

This interface allows the system to support several types of authentication handlers by implementing a new class for each type. The system has two classes that implement this interface, both for Fenix users:

- AuthFenix: makes use of the FenixEdu API [25];

- AuthFenixMap: makes use of the Spring Security API and the OAuth2 Spring API.

Currently the system is only using the second one, AuthFenixMap in order to be able to use those Spring APIs to improve security (which were not used with the AuthFenix).

In order to provide some basic server-server communication we developed a **REST Controller** with two endpoints with the following parameters:

- `/getRoutes`: origin, destination, departure date;
- `/getRoutesOnDay`: departure date.

The endpoint `/getRoutes` uses the same service as the ”Find trip” functionality from the passenger menu, which is the TripFinder. Except here we only use two filters, the PlaceFilter and the DateRelevanceFilter. After finding every trip that matches the desired parameters we build the data objects that are used to transmit this information. This data objects consists in a list of ResponseRoutes, each ResponseRoute has a list of Routes in a way that following that list the user will start on the desired origin, at the departure date, and end on the desired destination. A simple example, assuming that the user wants to go from point A to point B, can be one ResponseRoute that contains just one trip:

- Trip[origin=A, destination=B].

A more complex example, assuming that the user wants to go from point A to point C, can be one ResponseRoute that contains two trips:

- Trip[origin=A, destination=B];
- Trip[origin=B, destination=C].

The endpoint `/getRoutesOnDay` is identical to the one we just explained, with the difference that it only uses one filter which is the DateRelevanceFilter, thus it finds every trip on the day required.

4.1.5 Resources

This module stores the application properties, as well as the ”fenixedu” properties file, which defines security parameters necessary for authentication. It also contains two packages:

- Templates - holds all the HTML files;
- Static - holds all the JavaScript and CSS files.

The way we specify which HTML page is returned to the user on each Controller method is by simply writing the name of that HTML file on the return command, which Spring will use to find that same HTML file on the “resources/templates” package. All these HTML files use special Thymeleaf keywords which allow us to fill the HTML with dynamic content from the server, before returning them. To an existing

HTML file we can add Thymeleaf keywords and behavior, which if opened on a web browser (which does not process Thymeleaf modifications) will still look and work like a normal HTML, although without the Thymeleaf behavior we added. The key feature is being able to use Thymeleaf freely without it modifying the HTML in a way it can not be read and displayed properly by a browser, which eases the development and testing of such HTML pages.

A simple example of how this works, with a HTTP GET request:
On the HTML we can add inside a tag, `th:text="${name}"`, which tells Thymeleaf that it should expect some value that will be the text of that tag element. Then on the controller method that will use that HTML file we can specify what value we want it to be e.g. `model.addAttribute("name", user.getName());`. This "model" variable is an instance of the class Model from Spring, which we define on the method signature e.g. `methodName(Model model)` and then add attributes as needed. A Controller method for this is shown in Listing 4.4, where we have the endpoint "test", the HTML to be returned "index" and the model with the attribute "name" to be modified. After Thymeleaf finishes modifying the HTML it will return to the user the "index.html" file with the desired modifications.

```
@GetMapping("/test")
public String testGet(Model model) {

    User user = new User();

    model.addAttribute("name", user.getName());

    return "index";

}
```

Listing 4.4: Thymeleaf GET endpoint

We present another example but now with an HTTP POST request. With POST we want to receive information from the front-end, lets say we want to receive the input from a text-box, this can be done in several ways. One is to create an instance of the class `Query` with one `String` attribute `input` (define an empty constructor, getters and setters) which will be used to store the input we want. Then we go to the HTML of the GET that has the text-box and on the "form" tag we add `th:object="${query}"` which tells Thymeleaf that the form will submit that object as well as the `th:action="${/test}"` which tells what endpoint is should POST to. After that we need to bind the input text-box to the field `input` of that object, this can be accomplished doing `th:field="*{input}"`, as shown in Listing 4.5.

```
<form th:action="@{/test}" th:object="${query}" method="post" role="form">

    <input type="text" th:field="*{input}"/>

</form>
```

Listing 4.5: Thymeleaf POST example

This object `Query` needs to be created on the back-end beforehand, this is done by doing `model.addAttribute("query", new Query());` on the GET endpoint, as shown in Listing 4.6.

```
@GetMapping("/test")
public String testGet(Model model) {

    model.addAttribute("query", new DirectionsQuery());

    return "index";

}
```

Listing 4.6: Thymeleaf GET endpoint

Then to receive that object on the POST endpoint we just define it on the method signature with the help of a Spring anotation e.g. `methodName(@ModelAttribute Query query)`. After this we can use that object at will with e.g. `query.getInput();`, as shown in Listing 4.7.

```
@PostMapping("/test")
public String testPost(@ModelAttribute Query query) {

    System.out.println(query.getInput());

    return "index";

}
```

Listing 4.7: Thymeleaf POST endpoint

Thymeleaf also allows us to define HTML fragments and reuse them on other HTML pages. A fragment is defined on an HTML file with the keyword `th:fragment`. Lets take the example of our `simpleRequest.html` which has a fragment defined called `simpleRequest`, as shown in Listing 4.8. These fragments are under the `resources/templates/fragments` package.

```
<div th:fragment="simpleRequest">
    <!-- HTML code -->
</div>
```

Listing 4.8: Thymeleaf fragment

We want to use this fragment on the `tripDetails.html`, for this we use the keyword `th:replace`, as shown in Listing 4.9.

```
<div th:replace="fragments/simpleRequest :: simpleRequest"></div>
```

Listing 4.9: Thymeleaf replace

In a similar way we defined a `baseTemplate.html` to work as base for most of our other HTML pages. This allows us to only have the navigation bar HTML code in just one file instead of in all the pages that use it.

4.1.6 User input

In order to provide a user friendly interface, when the user input is required, our system offers several helping tools. When our system requires a date as input we offer a date widget (from Bootstrap), as illustrated in Figures 4.6 and 4.7, where the user can easily select a day, month, year, hour and minutes. When our system requires a place as input we offer an auto-complete tool (from Google Maps), as illustrated in Figure 4.8, which will autocomplete the user input with valid suggestions, while the user is typing, thus making the process of finding and choosing a place an easy task. This is done using Javascript and the class `google.maps.places.SearchBox` to bind the input field to the search box with the autocomplete functionality. When creating and finding trips we offer a map (from Google Maps) that can be used to show the route after having filled the origin, destination and pressing the "Preview" button. This functionality is achieved by using Javascript and some classes from Google Maps like `DirectionsService`, `Polyline`, `DirectionsRenderer` and `Map`.

4.1.7 Authentication

Before a user can access the Carpooling system he must first go through an authentication procedure, which is not implemented in our system, therefore we do not need to store any authentication data (e.g. username and password). Instead we use the feature of "Login with..." which in our case is login with Fénix (which uses CAS, as explained on Chapter 2).

The first time a user tries to use our system he will be prompted with a dialog box asking if he grants access to certain information of his Fénix account. In case the user grants that access he will then login

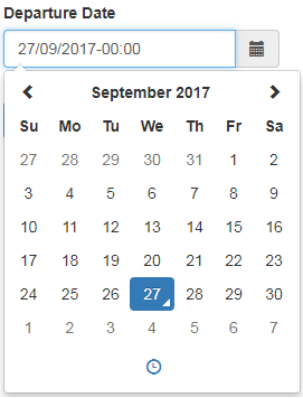

Figure 4.6: Selecting day, month and year

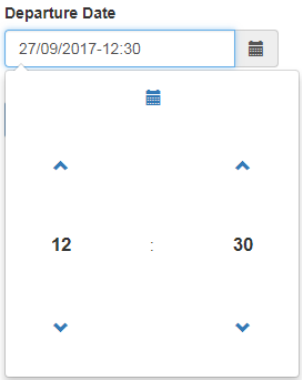

Figure 4.7: Selecting hour and minutes

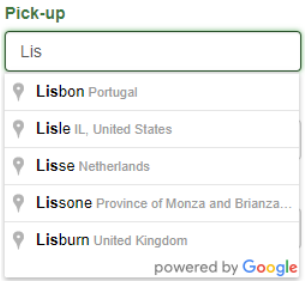

Figure 4.8: Auto-complete text box

with his Fénix credentials, which will give our system a token that can be used to extract information from his Fénix account. After some time the token will expire and the user has to login again.

This is achieved by using the Spring Security API and the OAuth2 Spring API. With these we can configure our Application class by making it extend `WebSecurityConfigurerAdapter` and implementing the `configure` method, as shown in Listing 4.10.

```
public class Application extends WebSecurityConfigurerAdapter{

    @Autowired
    private CustomAuthenticationSuccessHandler customAuthenticationSuccessHandler;

    @Override
    protected void configure(HttpSecurity http)throws Exception{

        http
                .antMatcher("/**")
                .authorizeRequests()
                .antMatchers("/", "/index", "/webjars/**")
```

```
                    .permitAll()
                    .anyRequest()
                    .authenticated();

        }
}
```

Listing 4.10: Application class

On this `configure` method we can configure security parameters, for example, what endpoints require the user to be authenticated. We do not want the index page to require authentication because this page is where the user chooses if he wants to login with Fénix or as an Administrator.

We also defined a `CustomAuthenticationSuccessHandler` where we register a `Listener` to the `AuthenticationSuccessEvent`, as shown in Listing 4.11.

```
@Component
public class CustomAuthenticationSuccessHandler implements ApplicationListener<AuthenticationS

    @Override
    public void onApplicationEvent(AuthenticationSuccessEvent event) {

    // first time setup code
    // update if necessary code

    }
}
```

Listing 4.11: CustomAuthenticationSuccessHandler

This enables us to intercept the flow of authentication, right after a user has successfully authenticated, extract user information from Fénix (using the `AuthFenixMap` explained before), update it on our system if it is different and perform initial setup.

4.2 Mobility Portal

The main goal of the Mobility Portal is to allow a user to submit a query to search for trips from an origin point to a destination point with a certain departure date. This query is then process by the system which will communicate with several other external systems and obtain possible routes that fulfill the request. This communication is performed through modular plugins that must be developed and deployed

to enable interfacing with each external system. Currently the Mobility Portal only communicates with our Carpooling system.

The Mobility Portal architecture has a similar architecture to that of the Carpooling system (described on Section 4.1) since the underlying technologies are the same. One difference, is that the Mobility Portal does not use a Database and thus the MongoDB API is not used either, as illustrated in Figure 4.9. A high-level view of the architecture of the system is illustrated in Figure 4.10, where the "Spring MVC" module is an abstraction of the previous diagram.

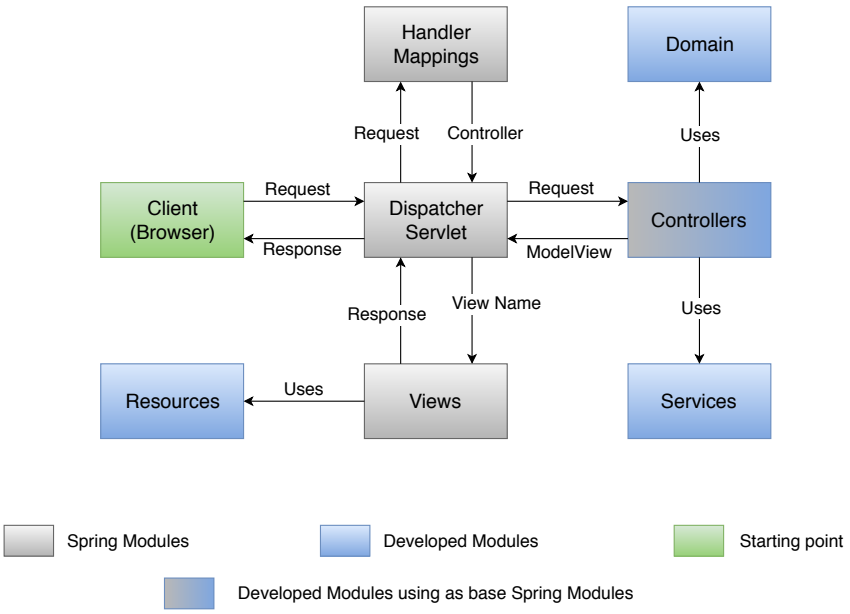

Figure 4.9: Mobility Portal architecture

4.2.1 Domain

This module holds all application containers needed to handle the information in a structured manner, as illustrated in Figure 4.11:

- **Route** - Specifies how the information of a route should be organized, it is used to preview the trip on the interface with the help of a map (even specifying the travel mode)
- **ResponseRoute** - Holds a list of Routes
- **Response** - Holds a list of ResponseRoutes, this will be the response of each plugin

4.2.2 Controllers

This module holds all the controllers needed to communicate with other servers (in this case the Carpooling). We only defined one Controller called `MainController` with the following endpoints:

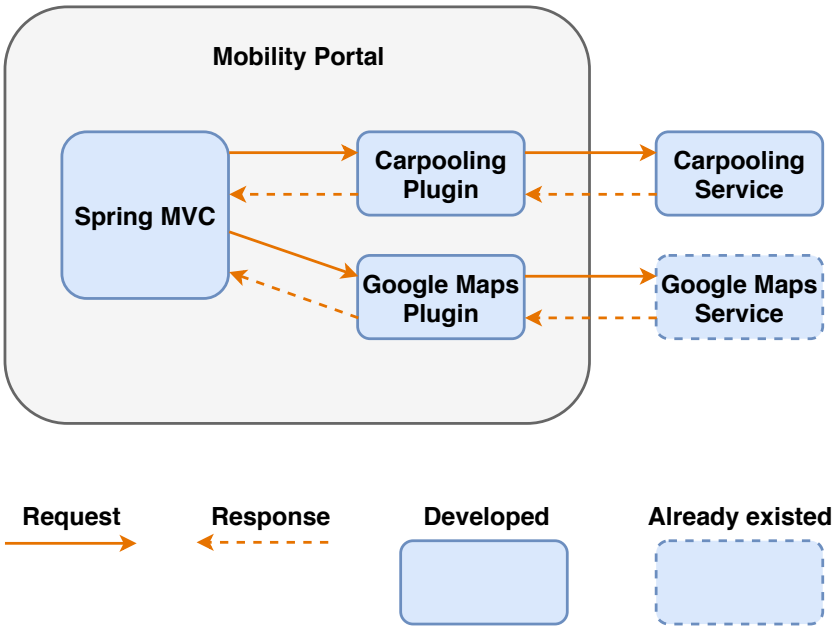

Figure 4.10: Mobility Portal high-level architecture

- `/test GET`
- `/test POST`

The `MainController` holds a list of Plugins which are in charge of querying other servers for routes that fulfill the user request. Currently there are only two Plugins which are the `CarpoolingPlugin` and `GoogleMapsPlugin`, but since all Plugins must implement a specific interface that we defined, shown in Listing 4.12, we can use an abstraction that allows us to easily add more Plugins in the future.

```
public interface Plugin {

    public List<ResponseRoute> getRoutes(String origin, String destination, Date
departureDate, Date arrivalDate);

    public String getName();

}
```

Listing 4.12: Plugin Interface

The **GET** will return the user the main HTML page where he can submit a query to search for trips from an origin point to a destination point with a certain departure date.

The **POST** will then receive the user input, loop through all the Plugins and ask them to query their

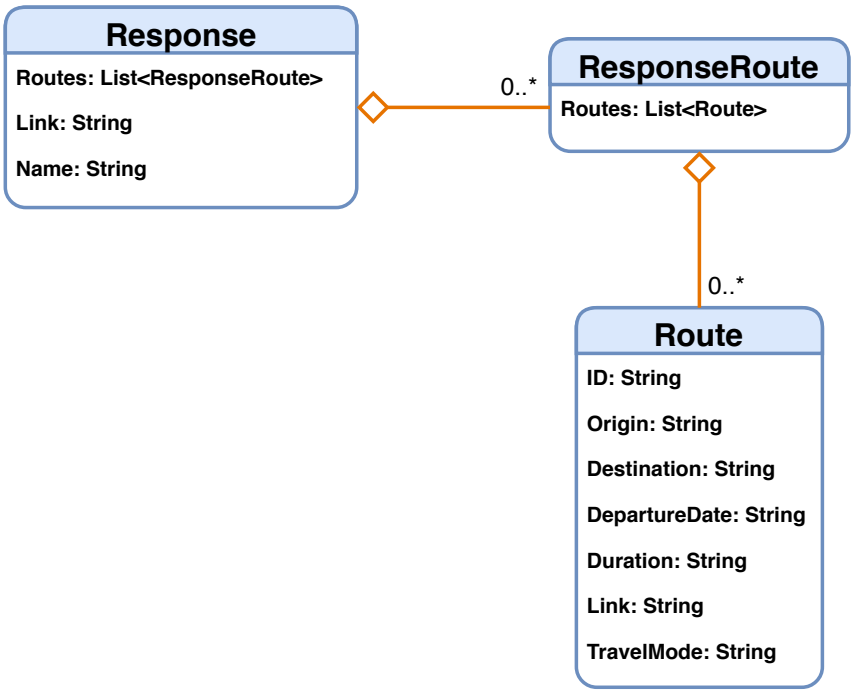

Figure 4.11: Mobility Portal UML

interfacing external systems for routes that fulfill the user request.

The `CarpoolingPlugin` allows four operations:

- `getRoutes(String origin, String destination, Date departureDate);`
- `getRoutesStartingAt(String origin, Date departureDate);`
- `getRoutesEndingAt(String destination);`
- `getRoutesOnDay(Date departureDate).`

All these operations will query the Carpooling system through predefined endpoints for the desired routes, either with the origin, destination and departure date defined (`getRoutes`), just the origin and departure date defined (`getRoutesStartingAt`), just the destination defined (`getRoutesEndingAt`) or just the departure date defined (`getRoutesOnDay`). The `getRoutes` operation is the most basic one, which just finds single trips from the origin to the destination leaving at the given departure date. The purpose of the `getRoutesStartingAt` and `getRoutesEndingAt` operations is to be able to find trips in the middle of the origin and destination in case they exist.

Currently the MainController only uses the `getRoutesOnDay` functionality of the `CarpoolingPlugin` which finds every trip available on that day. This interaction is illustrated in Figure 4.12 although with a different method name `getRoutes` which is the method name of the interface `Plugin`.

On that same functionality (`getRoutesOnDay`) we use another Plugin called `GoogleMapsPlugin` which enables the `ResponseRoutes` of a given Plugin to be filled with `Routes` from Google Maps. This

is useful when the Plugins response are Routes that either the origin does not match the origin requested or the destination does not match the destination requested. The `GoogleMapsPlugin` connects those points that do not match, as previously described in Section 3.2 in Figure 3.20.

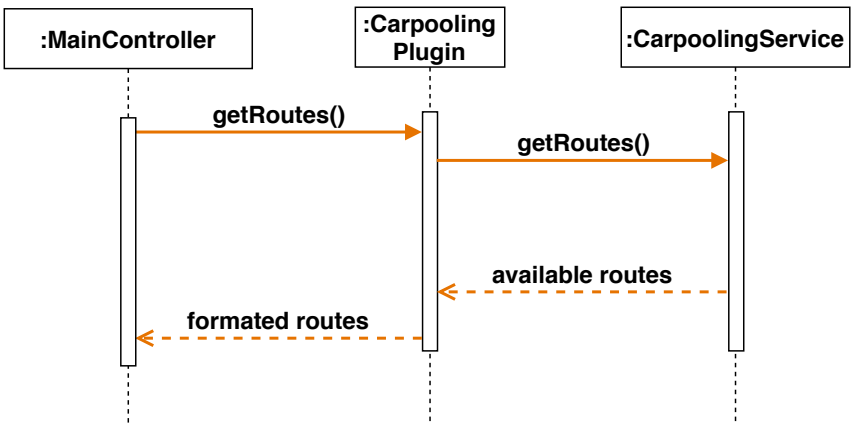

Figure 4.12: Carpooling Plugin interaction

4.2.3 Services and Resources

The **Services** module is divided into two different areas:

- The data objects used to transfer data between the back-end and the front-end;
- The logic that is not handled by the Controllers themselves.

The data objects purpose is the same as on the Carpooling system explained before. Since the interface of this system is much simpler than the Carpooling one we only need one HTML page (on the **Resources** module) to serve the user, thus the data objects we need are also much less, in this case just one to transmit the user input for the route query from the front-end to the back-end.

The logic that is not handled by the Controllers refers to all the Plugins of the system (previously explained) and the interface class that they all must implement:

- `Plugin` (interface);
- `GoogleMapsFiller`;
- `CarpoolingPlugin`.

5

Evaluation

Contents

As with every software system, or any other kind of system for that matter, it is important to test it with a sample of target users before deploying it to assess how well it performs as well as to detect major flaws and to be able to correct them before release to a larger audience. In the beginning of this document we described a series of requirements that our final solution should comply with. These requirements were addressed by features that need to be tested so that we can understand if they work as intended, to what extent they are easy to use and if there are any improvements that can be implemented. With this in mind we decided to conduct initial testing (also called Alpha Testing) with a few users when the Carpooling system was 70% - 90% complete, and later on some Beta Testing when the Carpooling system was 90% - 95% complete [26].

5.1 Alpha testing

The main objective of this initial testing stage was to check whether we were on the right path, if there were any bugs that went undetected and if there were features that could be further improved. This proved to be very helpful since we were able to identify several problems and thus improve the system still during the main development stage. Some of the problems that were identified were:

- When creating a trip the driver could select as a preference value “Not relevant”, which was confusing and did not make much sense for the users, thus we changed it so that the driver only has two options, which are either to allow or not allow a given preference;
- When a driver inspected one of his own current trips he could not see if there were any booking requests for that trip; to check if there were booking requests for that trip the user had to go to a different interface. Clearly, this was suboptimal, therefore we added that functionality. Now the user can still check all the booking requests on a dedicated interface, but he can also view them when inspecting a trip (only the requests for that trip);
- The rating was only given with a star classification, but some people wanted to leave a free text comment as well, thus we added the possibility to leave a comment as well;
- When finding trips, some people did not understand how to proceed to select a trip and book seats, therefore we added a specific button saying “Book seats” (although the user could just click any area inside the rectangle that contains the trip to open the trip details where he could book the seats);
- The navigation bar on the top of the page was not very clear in the sense that the user needed to press it to open the various menus, hence we added some hovering effects to make it more clear.

5.2 Beta testing

The main goal of this more advanced testing stage was to grasp if users, in practice, understood the interface of the Carpooling system, were able to navigate/use it easily, measure user satisfaction and collect additional suggestions. In this stage, a set of users were given the system to follow a script with a series of predefined tasks that were meant to cover most of the interactions that a user can have with the system. For each task the user was asked to rate it in terms of difficulty from 1 (Hard) to 5 (Easy) and comment on its use. The script starts with a brief introduction explaining its purpose and what is expected from the test users that were asked to follow it. Afterwards the user is presented with a sequence of eight tasks (plus task zero which is just to write down the time at which the test was started so that in the end we can estimate its duration). Additionally, in the end the user is presented with two open questions, described on Table 5.2. In total, 12 users tested the system, for which, according to Table 5.1 taken from Faulkner (2003) [27], the mean percentage of problems found relative to the number of participants (12 users) is between 94% and 97%, with the minimum being between 82% and 90%, thus suggesting that our testing pool of users can provide valid and significant results. Each user took, on average, 14 minutes to complete the testing, although not all provided an answer to the open questions (which were optional), the results were positive in general and are shown in charts of Figure 5.2. In the following paragraphs we will analyze these results and the answers to the open questions.

Number of Participants	Minimum % Found	Mean % Found
5	55	85.55
10	82	94.68
15	90	97.05
20	95	98.40
30	97	99.00
40	98	99.60
50	98	100.00

Table 5.1: Numbers of problems found relative to the number of participants

All the tasks had at least 8 users that rated them as easy (5) which is 66% of the test users, with the exception of the second task that only had 7. Tasks 2 and 3 had some users that found them hard to complete, that might be due to the initial contact with the system, since the users never had used the system before and they might need some time to learn how to properly navigate on it.

A few users (not the majority) said that the tasks were not clearly described, that they had to go back and forth between the web pages to get all the details. They also said that perhaps the first tasks were always more difficult because it was the first time they were using the system. This supports our analyses of tasks 2 and 3. We tried to soften the initial learning curve by inserting two buttons (”I’m a passenger” and ”I’m a driver”) which redirect the user to the respective roles (passenger or driver). We also had the opposite case were the users said that the tasks were very simple and the system was very

Tasks	Description
1	Open the application on the following page and login with Fénix: http://51.254.115.204:8090/index
2	Find trips from “Alameda” (Lisbon) to “Taguspark, SA” (Oeiras) leaving on November 5, 2018 at 10:00h
3	Request to book one seat on the first trip found
4	Create a trip from “Lisbon” to “Porto” with departure 15 November 2018 at 9:00h, with 3h duration, 5€ per seat, with 4 seats available, only for Students, for non smokers, with animals allowed, with the description “this is a test”
5	Rate the trip that you went on as a passenger from “Amadora” to “Taguspark”
6	Accept a request from Sofia for the trip that the system created for you as driver (from “Telheiras” to “Alameda”)
7	Add the language “English” to your profile and save
8	Choose if most often you are going to be a passenger or a driver and save your profile
Optional	If you found any task particularly hard can you explain why or how would it be easier to do?
Optional	Is there any functionality that you would like to see in this system?

Table 5.2: Tasks and respective description

intuitive.

Tasks 5, 6 and 7 had one user rate them with 2 (between hard and medium), which was the user that said that the tasks were not clearly described, as discussed above. Thus we believe that this score was due to the user not fully understanding how to navigate the system.

If we analyze from task 4 to task 8 we can see that more than 83% of the users rated between 4 and 5 (easy). With this results we can conclude that after the first 3 tasks, the users were able to navigate the system easily and complete the remaining tasks swiftly. Analyzing the average difficulty for each task, shown in the chart of Figure 5.1, we can conclude that the Carpooling system had a very good feedback from the users, with an average difficulty rating of 4 or more in each task.

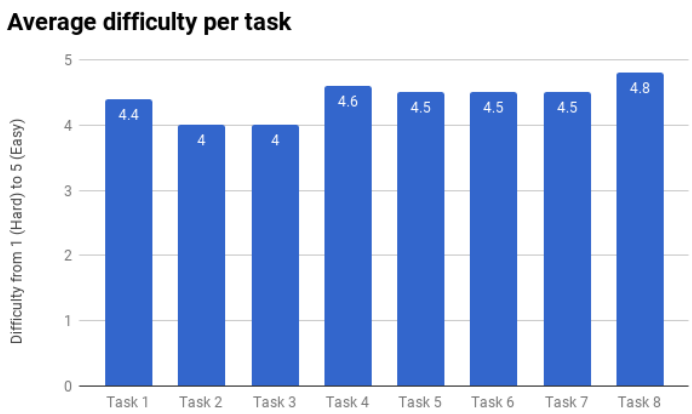

Figure 5.1: Average difficulty per task

On the questions with open answer we got some useful comments and suggestions of improvement

to our system, namely:

- ”Use different colors for the menu (and sub-menu) passenger and driver. Because they look very alike, the information that you can read and choose once you opt for each one. If they had different colors on the buttons selected or on the background that you could easily understand if you are using the system as a driver or as a passenger, it would be a visual communication improvement”. Doing this clear distinction of roles with colors should help on the initial understanding of the system;
- ”Having a distance limit that the user does not mind covering by foot”, which would allow a more flexible search of trips;
- ”The option to carry volumes or bulky items on the trunk”;
- ”When the system created the request to my Driver I was in Passenger mode and took a while to realize I had to switch roles. Maybe a red number near the mode (Driver or Passenger) indicating how many notifications/requests we have on those modes”. This suggests that the notifications system can still be improved.

Beta testing has shown us that after the initial learning curve the Carpoling interface was easily manipulated and understood, and that users could easily find and create trips in little time. Thus achieving the main objective of *Build our own Carpooling service and add the respective plugin to our Mobility Portal that can query that service*.

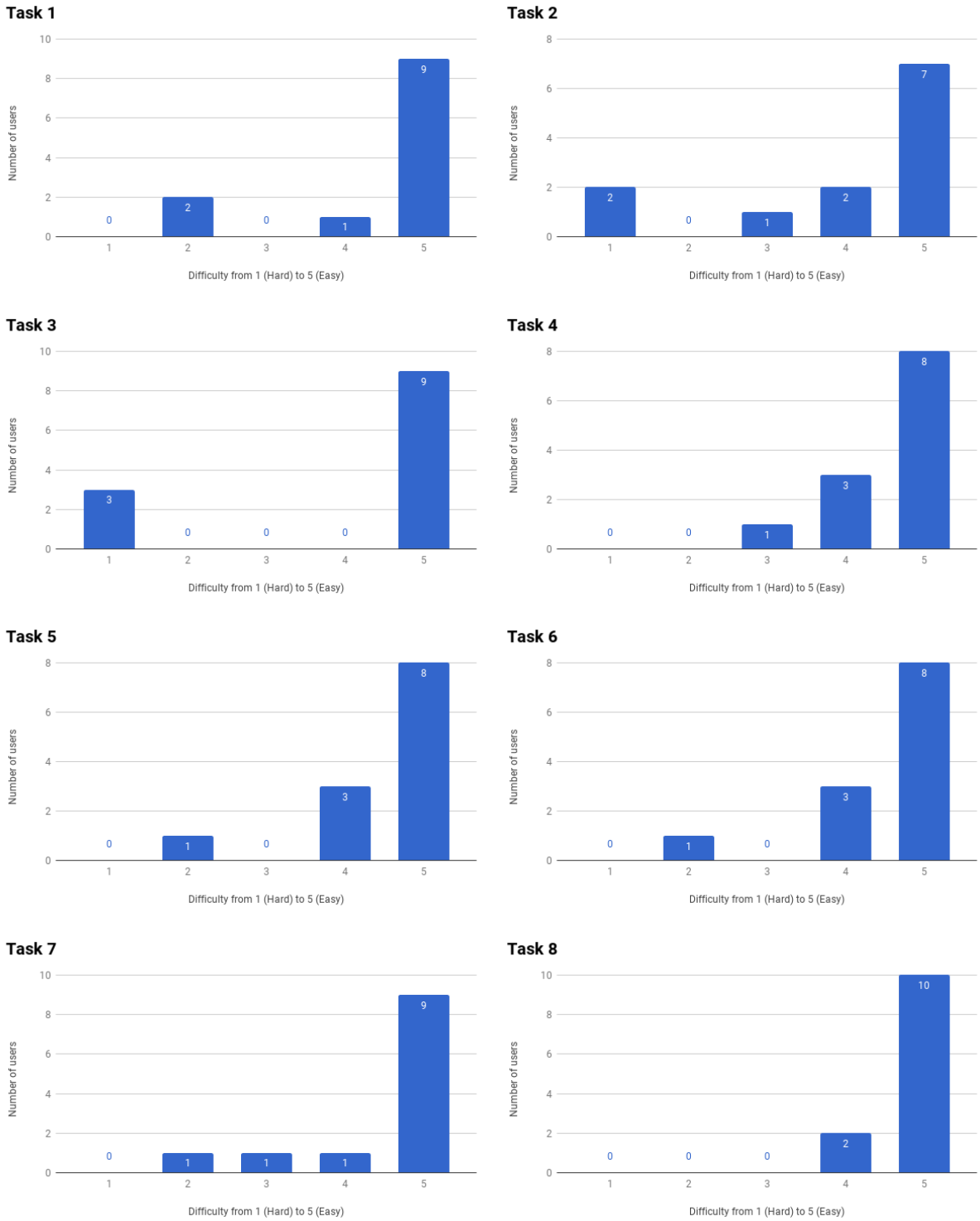

Figure 5.2: Tasks results

6

Conclusions

Contents

With all the transportation options available nowadays it becomes difficult to search for the most convenient option to travel from one place to another. As we have seen in this work there is no particular system that allows, for example, college students to search for transportation options that only their college provides and only they can use, while at the same time searching for general transportation options (e.g. public transports). This is, in general, true for most other large organizations with multiple campi, thus we found the need to develop a solution that allows an authenticated user from a given organization to search for the most convenient option to travel from one place to another using several different transportation services including the ones provided by the organization itself to its users (students, employees, affiliates, etc). The Mobility Portal targets organizations where, usually, there is a good level of trust between users. It leverages on all available transportation methods of such organization by creating plugins able to extract routes with schedules from the systems that handle those transportation methods. Since this system aims to be deployed on any organization we thought it would be interesting to develop a Carpooling system and use it as one of the external services of the Mobility Portal.

To achieve this we first identified the different inputs necessary to both our systems and explored how other systems accomplish required functionality. After analyzing this we defined all the information that users would have to handle. In order to have a modular solution we decided that the Mobility Portal should communicate with external services through plugins (including our Carpooling service). This plugin architecture enables our system to be easily extended to collect information from many different sources just by adding new plugins that can perform queries on such systems. We then moved to the algorithmic part of our system, more precisely to how both our systems would compute a result for a given input and display it to the user in an intuitive and useful way. In order to understand the best approach to use we analyzed several similar platforms, so that we could understand their advantages and disadvantages. After considering this we defined the Mobility Portal's behavior which is based on looping through plugins, and the Carpooling behavior which looks up for matching trips. Since the results of both systems are mainly routes and schedules, we decided that the best way for the user to understand their output would be with a list of possible routes and their respective schedules, which he can select and see on a map.

After the Carpooling system was 70% - 90% complete we decided to run some basic tests with users (Alpha Testing) to see if we were on the right path, if there were any bugs that we did not cover and if there were features that could be improved. This testings proved very helpful as stated in Chapter 5. Then when the Carpooling system was 90% - 95% complete we did some Beta Testing which had the purpose of checking if users, in practice, understood the interface of our system, were able to navigate/use it easily, measure user satisfaction and collect suggestions.

Our initial goals of: *Build our own Carpooling service and add the respective plugin to our Mobility Portal that can query that service* and *Create a modular system (Mobility Portal) that allows users to*

define an origin, destination and departure date and search for different ways to travel from one point to the other on that departure date were achieved, as our evaluation shows in Chapter 5 (although the Mobility Portal was not tested with real users).

The systems truly work, despite having a few minor flaws to be improved in the future.

6.1 System Limitations and Future Work

Regarding the current implementation, we identified some features that would be interesting to have in the future. Both systems would benefit from supporting multiple languages, which would allow both applications to reach more users; being able to choose the origin and destination points by selecting a waypoint on the map interface; having the interface design improved. The Mobility Portal would benefit from the development of a plugin able to query the IST Shuttle (that is a transportation service between the IST facilities) for route and schedule information; developing a plugin that is able to take a file with route and schedule information, in a standard widespread format and translate it into information that can be looked up. The Carpooling system would benefit from allowing the addition of waypoints when creating a trip, which would enable the system to find subtrips within trips.

Some of the limitations that we identified were: when finding a trip the Carpooling system is not flexible with the locations, for example if a user searches for a trip with origin ”A” and the system only has one trip with origin ”A B” (a more detailed place within ”A”), he will not find any trips. There are also places that should be the same, e.g. ”Lisboa” and ”Lisbon”, but are viewed as different places (although even their Google Maps Place ID is different).

Bibliography

[1] Amsterdam Arena Mobility Portal. [Online]. Available: http://www.amsterdamarena.nl/stadium-surroundings/arena-mobility-portal.htm

[2] 511 SF Bay. [Online]. Available: https://511.org/

[3] Boleia.net. [Online]. Available: http://www.boleia.net/

[4] Blablacar. [Online]. Available: https://www.blablacar.pt/

[5] Poparide. [Online]. Available: https://www.poparide.com/

[6] SINGU Car Pooling. [Online]. Available: https://carpooling.tecnico.ulisboa.pt

[7] Fielding, Roy Thomas (2000). ”Chapter 5: Representational State Transfer (REST)”. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

[8] Spring MVC. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[9] Jakarta Struts Web framework. [Online]. Available: http://struts.apache.org

[10] Thymeleaf. [Online]. Available: http://www.thymeleaf.org

[11] ”What is the Google Maps API?”. [Online]. Available: https://developers.google.com/maps/

[12] Google, Inc. General Transit Feed Specification Reference. [Online]. Available: http://developers.google.com/transit/gtfs/reference

[13] Google Maps Directions API usage limits. [Online]. Available: https://developers.google.com/maps/documentation/directions/usage-limits

[14] Marcin Kossakowski, Finding shortest path using Dijkstra’s algorithm and weighed directed graph.

[15] The OAuth 2.0 Authorization Framework (RFC 6749).

[16] ”OpenID Connect”. OpenID Foundation.

[17] ”JASIG CAS Protocol Page”. Apereo/JASIG.

[18] Whitson Gordon. ”Understanding OAuth: What Happens When You Log Into a Site with Google, Twitter, or Facebook”.

[19] The OAuth 1.0 Protocol (RFC 5849).

[20] ”User Authentication with OAuth 2.0”. OAuth.net.

[21] Eldon, Eric (2009-04-14). ”Single sign-on service OpenID getting more usage”.

[22] OpenID FAQ. [Online]. Available: http://openid.net/connect/faq/

[23] Drew Mazurek. ”CAS Protocol Specification 3.0”.

[24] Spring Data MongoDB. [Online]. Available: https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/

[25] FenixEdu. [Online]. Available: http://fenixedu.org/dev/api/

[26] Alpha Testing and Beta Testing (A Complete Guide). [Online]. Available: http://www.softwaretestinghelp.com/what-is-alpha-testing-beta-testing/

[27] Laura Faulkner (2003). Beyond the five-user assumption: Benefits of increased sample sizes in usability testing.