Road Traffic Engineering for Improved Campus Mobility

Auteur : Imperiali d'Afflitto, Michele Jose
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité approfondie
Année académique : 2015-2016
URI/URL : http://hdl.handle.net/2268.2/1644

UNIVERSITÉ DE LIÈGE

Faculty of Applied Sciences

Master Degree Thesis

Road Traffic Engineering for Improved Campus Mobility

Master thesis in Computer Science
by
Michele José Imperiali d'Afflitto

Supervisor:
Prof. Laurent Mathy

Jury:
PhD Tom Barbette
Prof. Pierre Wolper
Prof. Damien Ernst

Academic Year 2015-2016

Abstract

Road Traffic Engineering for Improved Campus Mobility

Michele José IMPERIALI d'AFFLITTO

Master in Computer Science
University of Liège - Academic Year: 2015-2016

Promoter: Prof. Laurent Mathy

Accessing the University of Liège without a car is not optimal. Parking spots are a scarce resource. Moving from one building to another with public transportation is unpractical. Distances are too large to travel by foot. Biking is not really an option, because few secured parking spots and charging stations for electric bikes are available.

The goal of this thesis is to ease accessing, leaving, and moving around the university, by leveraging the empty space in the fleet of cars of the University members.

Practically, this means we want to create an ergonomic system to offer empty seats and to ask for lifts, to, from, and between ULg locations. This system should thus, from the end-user point of view, be available on smart-phones. Offers and demands should be centralised and re-dispatched in a smart way to propose simple and efficient combinations to end-users, and in real-time.

This work consists in the development of an Android application providing carpooling functionalities, enhancing the back-end system developed in the context of another project[1], authored by PhD student Thibaut Cuvelier, who implemented a first version of a website providing basic matching between drivers and passengers, based on a static graph including the most important towns around the University of Liège.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my promoter Prof. Laurent Mathy for the continuous support during my work and for his patience.

Besides my promoter, I would like to thank my supervisor, PhD student Tom Barbette, for his insightful comments and encouragement.

I would also like to thank my family: my mum and Massimo, my dad and Irene, my brothers and sisters for supporting me spiritually throughout my entire academic journey and my life in general.

I must also mention my uncle Franco, a really special person, who has always been next to me and supported me, especially during these last two years of master abroad.

And above all I would really like to express my deepest thank you to all my friends and especially to my girlfriend Daniela, who has been next to me during every moment and has always supported me with patience and love.

This work is the result of all the support I received throughout my career as a student and I will never thank everybody enough for it.

Contents

Abstract III

Acknowledgements V

1 Introduction 1
- 1.1 Previous Work . . . 1
- 1.2 Project Requirements . . . 2
- 1.3 Setting Up the Environment . . . 2

2 Architecture and Technology 3
- 2.1 Django Backend . . . 3
- 2.2 Tastypie API . . . 4
  - 2.2.1 Requests Authentication . . . 5
  - 2.2.2 Communication with API . . . 6
  - 2.2.3 Data Serialisation with Gson . . . 8
- 2.3 External Services: Google APIs . . . 8
- 2.4 Registration Mechanism and Automatic Login . . . 9
- 2.5 Ride-Demand: Matching Algorithm . . . 10
  - 2.5.1 Market Research . . . 10
  - 2.5.2 How It Works . . . 11
  - 2.5.3 Geopy . . . 12
- 2.6 Real-Time Notifications . . . 13
- 2.7 Connectivity Monitoring . . . 15
- 2.8 Errors Handling . . . 15
- 2.9 Localisation . . . 17

3 App Structure 19
- 3.1 Introduction . . . 19
- 3.2 Activity Package . . . 19
  - 3.2.1 Activities Hierarchy . . . 21

3.3 Adapter Package 21
3.4 Data Package 21

4 Use Cases 25
4.1 Introduction 25
4.2 Registration and Login 26
4.3 Driver Mode 26
4.3.1 Add new rides 27
4.3.2 Find potential passengers 28
4.3.3 Submit Feedback 30
4.4 Passenger Mode 31
4.4.1 Ask lift 32
4.4.2 Find potential drivers 32
4.5 User Profile 33

5 Conclusions 35
5.1 Final Result 35
5.2 Future works and Improvements 35
5.2.1 GPS Functionality 36
5.2.2 Payment System 36
5.2.3 JUnit Testing 36

6 Appendices 37
6.1 Data Model 38
6.1.1 User 38
6.1.2 Preference 38
6.1.3 Vehicle 38
6.1.4 Device 39
6.1.5 Trip 39
6.1.6 Ride 39
6.1.7 Ride Status 40
6.1.8 Demand 40
6.1.9 Tolerance 40
6.1.10 Graph 40
6.1.11 Node 40
6.1.12 Edge 41
6.1.13 Path 41
6.1.14 Passengership 41
6.2 Code Snippets 42
6.2.1 Authenticator class 42
6.2.2 Connectivity receiver 43
6.3 API Calls 44

6.3.1 Node API 45
6.3.2 User API 45
6.3.3 Ride API 48
6.3.4 Demand API 52
6.3.5 Passengership API 56

Bibliography 57

List of Tables

2.1 Authorisation Classes 5
2.2 Application HTTP requests 7
2.3 Application notifications 14
2.4 Custom exceptions 16
3.1 Application custom adapters 22
6.1 All nodes request - Optional parameters 45
6.2 User Registration - Exceptions 46
6.3 Ride Registration - Exceptions 48
6.4 Ride request - Optional parameters 48
6.5 All rides request - Optional parameters 49
6.6 Delete Ride - Exceptions 49
6.7 Potential passengers request - Optional parameters 50
6.8 Get Potential Passengers - Exceptions 50
6.9 Confirm Passenger - Exceptions 51
6.10 Confirm Driver - Exceptions 51
6.11 Reject Match - Exceptions 52
6.12 Propose Ride - Exceptions 53
6.13 Leave Ride - Exceptions 53
6.14 Get demand request - Optional parameters 54
6.15 Delete Demand - Exceptions 54
6.16 Potential drivers request - Optional parameters 55
6.17 Potential Drivers - Exceptions 55
6.18 Feedback - Exceptions 56

List of Figures

2.1 Car On The Hill - The Android app logo 4
2.2 HTTP requests hierarchy 6
2.3 Matching algorithm iterations, function of existing demands . . . 12
2.4 Google Cloud Messaging - Architecture 13
3.1 Android application code structure 20
3.2 Activities hierarchy sub-schema 23
4.1 Register and login views 27
4.2 Main driver views 28
4.3 Add new ride views 29
4.4 Find potential passengers 30
4.5 Ratings 31
4.6 Passenger main views 32
4.7 User profile management 33
6.1 API classes hierarchy 44

Chapter 1

Introduction

1.1 Previous Work

My thesis work is the continuation of another project, authored by PhD Thibaut Cuvelier, who implemented a carpooling website, based on a static graph of the main towns around the University of Liège.

The website is called CarOnTheHill[1] and, after registration, it allows the user to be either a driver or a passenger.

In driver mode users can create new rides and find potential passengers which are statically matched based on the ride path. In passenger mode, users can create ride requests and await to be accepted by a potential driver.

The main limitation of this first version was its static graph architecture, causing the impossibility for end users to define routes with addresses different from the ones that were used to pre-populate the application database. Secondly, for a carpooling application the number one requirement is to be easily accessible from smart-phones, possibly through an *ad hoc* application.

[1]The name CarOnTheHill originates from the fact that the majority of the faculties of ULg University reside on a hill, in the town of Sart-Tilman, in the South of Liège.

1.2 Project Requirements

Starting from what had been developed by Thibaut Cuvelier, I was asked to work on an Android application, being able to reuse most of the logic put in place for the website, using a predefined set of API calls, eventually to be enriched.

During the first weeks of my thesis I spent most of the time to understand the overall application structure and to get confident with the used frameworks. As it is the case for every work based on somebody else's strategic and technological decisions, it took quite a while to be confident with the code and its overall functioning.

Together with my promoter, Prof. Laurent Mathy, and my supervisor, PhD Tom Barbette, we agreed to first work on the development of an Android application retracing the existing website. Then we would have focused on new functionalities.

This first version of the Android application allowed a user to register, using the academic email, and then either add new rides or ask for a lift. Since the graph used by the application at this stage was static and pre-defined, users were forced to select origin and destinations of their rides from a limited list of nodes and the matching between demand and offer was done by an algorithm traversing all nodes of a given ride path, and checking for related pending requests.

1.3 Setting Up the Environment

To be able to work on my laptop I was offered by my supervisor Tom Barbette a virtual machine emulating the actual server. This is created and configured using Vagrant[2], a very straightforward solution to develop local environments.

The real server resides on a subdomain (called kaiss) of Montefiore webserver.

To speed up the setup process I wrote two very basic batch scripts to launch the virtual machine and to open a separate terminal to be used for database operations.

Throughout my work I was able to use great software which really helped me focusing on the actual code writing. For code changes tracking I used SourceTree[3], pushing the latest version of my work to a GitLab directory hosted on Montefiore server. For the back-end, written in Python, I used IntelliJ software PyCharm[4]. Finally, for the development of the actual Android application I used Android Studio[5].

Chapter 2

Architecture and Technology

This chapter contains the main technological aspects of my work. The project I have worked at consists of two parts:

- Android application, representing the core product for the end user;
- Django[6] back-end, implementing all the main functionalities used by the application.

The communication between the Android application and the Django back-end occurs through an *ad hoc* API, implemented using Tastypie[8]. Android application and back-end exchange JSON data which is serialised and deserialised using the Gson Java library[7].

2.1 Django Backend

As previously stated, the backend of my thesis is developed using Django, a high level web framework, on top of which I have implemented the Tastypie API. The Django backend was already developed when I started to work at this project, and so was a good part of the API, however the newly introduced functionalities forced me to change quite a bit of the original code.

Currently the backend consists of three main Python files (which are also the ones I have worked on the most):

- *Models*: this file contains all the data models that I have also replicated in the Android application. It also implements the system core functionalities.

Figure 2.1: Car On The Hill - The Android app logo

- *Resources*: this file implements the actual Tastypie API and defines all the listening points of the server, that is the URLs using which the Android application (and any other service) can exchange data with the backend.
- *Graph*: At the beginning of my work, since the application used a static graph this file had quite a bit of logic in it. Now, however, it *only* implements the actual matching algorithm, finding potential drivers or passengers for demands and rides.

2.2 Tastypie API

The application API has been developed using Tastypie, a web-service API framework for Django, following the RESTful approach[12]. For the purpose of my thesis I was provided with a first draft of API that I have followed almost entirely. A detailed list of all implemented API calls is reported in Appendix 6.3.

To better deal with the fetching of related objects, I have used the **lazy loading pattern**[13], therefore in some cases I have implemented requests were the user can specify with *ad hoc* parameters whether to perform a deep inspection of the resource or a shallow one. This is for example the case for the request retrieving ride data for which we are able to specify whether to retrieve the full ride path information or not.

With Tastypie I was able to define 3 levels of authorisation for the application resources. These have been implemented by deriving the Tastypie Authorization class and obtaining 3 classes described in Table 2.1.

Class Name	Description
UserObjectsOnlyAuthorization	Authorisation used to limit read and write operations only to authenticated users. This is the minimum level of protection for the application resources.
DriverObjectsOnlyAuthorization	Authorisation used to limit results of read list operations to objects owned by the connected user, in driver mode (e.g. if the connected user wants to get the list of his rides).
PassengerObjectsOnlyAuthorizati	Authorisation used to limit results of read list operations to objects owned by the connected user, in passenger mode (e.g. if the connected user wants to get a list of his lift demands).

Table 2.1: Authorisation Classes

In addition to the authorisation mechanism described above, I put particular attention in the definition of *ad hoc* security checks on the backend, making sure that every API request is allowed and returning descriptive error messages to inform the user about eventual unauthorised requests. More details about the API returned errors are available in Appendix, in section 6.3.

2.2.1 Requests Authentication

The authorisation mechanism used for the communication between Android application and server pairs the user credentials to each request. This approach was in fact preferred to a more modern token-based authentication system. This decision was taken because the former approach is much simpler to implement than the latter but it indeed comes with some costs, the biggest one of which is poor scalability.

The Android class implementing the authentication is called `Authenticator`, placed in the homonym package. This class is responsible for sending the user credentials to the server, paired with every HTTP request.

The `Authenticator` class is very simple and it is implemented using a singleton pattern. User credentials, once provided through the login form, are encoded using

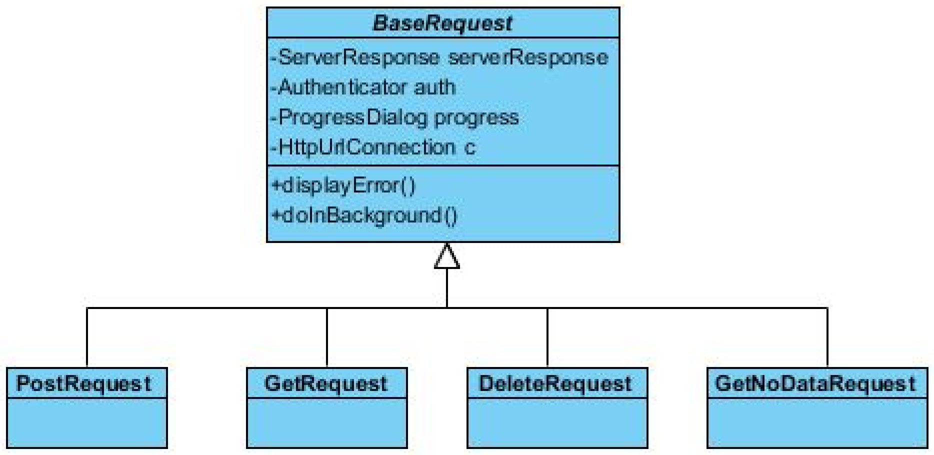


Figure 2.2: HTTP requests hierarchy

the `Base64` class[1] and stored in the singleton, making them available to every subsequent API request. The full class code is reported in listing 6.1 in Appendix.

2.2.2 Communication with API

The communication with the Tastypie API is managed through a set of classes, grouped in the HttpRequests package. These classes are all implemented extending the `AsynkTask` class, so to ensure that networking activities are not performed on the main thread, otherwise risking to freeze the application.

As shown in figure 2.2, the requests all derive from the base abstract class `BaseRequest`, which simply takes care of HTTP errors display and holds some common variables. Worth mentioning the `doInBackground` function: it performs a check for existing Internet connection and it has to be overridden by the other classes to perform the actual HTTP request. Thanks to this implementation mechanism no request is sent out if no connection is available, avoiding unpredictable

[1]Base64 is a group of similar binary-to-text encoding schemes that represent binary data in an ASCII string format by translating it into a radix-64 representation [...] - https://en.wikipedia.org/wiki/Base64

behaviour of the application.

The class taking care of the actual server response is called `ServerResponseObject`. It is in the same package of the other HTTP request classes and, after parsing the JSON of the server response, prints it in the Android console for debugging purpose. Together with the server data this class also holds the info regarding eventual error codes and messages.

Each of the request classes parses the JSON of the server response using the `ServerResponseObject` class and returns the object to the caller. The details of each class are reported in table 2.2.

Class Name	**Description**
GetRequest	This is the most common HTTP request, used every time the application wants to retrieve some data from the server. This may be returned in an array format in case of get list requests, or a simple JSON object. In either case, this class parses the payload and returns it to the requesting class which then uses Gson to store the result into the appropriate data model.
PostRequest	This request is used to send data in JSON format to the server.
DeleteRequest	This request is used to delete data from the server. For each delete request, for security reasons, the backend carefully verifies if the request comes from the resource owner.
GetNoDataRequest	As its name clearly states this request is a GET request which however does not produce any data being returned from the server, but simply an HTTP code, so it skips all the parsing performed by the similar `GetRequest` class.

Table 2.2: Application HTTP requests

The way these classes are used to communicate with the application backend is by creating new private classes extending the required request. The newly created class contains only one overridden method, called `onPostExecute`, which is automatically triggered after the completion of the HTTP request and returns an object of the `ServerResponseObject` class, representing the server response. The body of the function contains the code handling the obtained result, eventually parsing the returned JSON using the `GsonCustomBuilder` class.

2.2.3 Data Serialisation with Gson

As stated in the previous sections all the data exchanged between the Android application and the Tastypie API is in JSON format. This means that once the application receives some data, it has to parse it for storing it in the correspondent data model. This is achieved using the Gson Java library which is able to transform data to and from JSON format seamlessly.

In order to properly handle the serialisation and deserialisation of some data types, in particular for `GregorianCalendar` and `Status` data models, I had to build a customised Gson builder. For this manner it was sufficient to define *ad hoc* classes implementing the conversion from JSON to data type and viceversa and link them to the custom Gson builder.

Across the application, all the classes are using the custom Gson builder by retrieving the unique static instance of the `GsonCustomBuilder` class. Using the functions `toJson` and `fromJson`, the builder is able to automatically identify the type of serialisation/deserialisation to perform based on each data type it comes across.

2.3 External Services: Google APIs

The application uses several Google APIs for some of its core features. These are Google Maps, Google Maps Directions and Google Cloud Messaging.

The maps API is used to display to the driver the set of potential passengers on a Google map, as shown in figure 4.4a.
Furthermore, since the core of CarOnTheHill application is based on addresses, it is crucial not to leave the user providing this information unsupervised. The objective is to deal with a common address representation so to be able to ensure consistency across the application. With this premise, I have decided to use the `Geocoder` Android class to perform address auto completion, returning `Address` Java objects upon user selection.

More in details, the Android application communicates with Google Maps when the user is typing an address for a ride or demand creation. The classes handling the auto-completion and the communication with the Google API are in the `geoautocompletion` package and the main one is called `GeoAutocompletion-Adapter`. Results obtained by this adapter are limited to Belgium and every time up to five matches are displayed[2].

The custom class `DelayAutoCompleteTextView` inherited from `AutoComplete-TextView` makes sure to delay the requests sent to the Google API and eventually destroys pending ones if a new one is to be sent because the user continues typing. This is an optimisation which ensures that there is no waste of Google API requests.

On the other hand, the back-end uses the Google Maps Directions API to compute the shortest driving path between two provided locations. The result is then stored in the server database as a series of nodes connecting origin to destination. One advantage of this approach is that further improvements of this project will be able to use the stored information to implement routing algorithms, for example to improve traffic in the campus in case of congestion.

The Maps Directions API is also used by the matching algorithm to compute the shortest path a driver should follow to pick up a given passenger, and check if he would be able to take him to destination on time (more information on the algorithm functioning are in section 2.5).

The Google Cloud Messaging API is used for application notifications, described in details in section 2.6.

2.4 Registration Mechanism and Automatic Login

One of the project requirements was to ensure registration only for ULg personnel since the application is intended for people related to the ULg university. To enforce this requirement, at registration time, an *ad hoc* regex expression is used to ensure that the provided email belongs to the ulg.ac.be domain.

To successfully complete the registration, the user is also asked to provide a username which can be interchangeably used to log in.
When the registration data is sent to the server, the Django back-end checks for duplicated username and emails, therefore ensuring unique usage.

[2]For the implementation of the geo-autocompletion functionality I customised a very well written tutorial which I found on Google after some research, available on http://garbtech.co.uk/android-implementing-a-google-maps-search-box-with-autocompletetextview-and-geocoder-api/

The automatic login functionality is achieved using the Java `SharedPreferences` class. This class always stores the last used username or email and optionally the password (user has to check the auto-login option upon login). In case of automatic login enabled, upon the next launch attempt, the app will gather the credentials from `SharedPreferences` and will automatically try to perform a GET request of the user profile data. If the server responds with no error the user is automatically logged in.

2.5 Ride-Demand: Matching Algorithm

The first version of matching algorithm used by the application to find potential passengers for newly created rides was very performing. This algorithm, developed by Thibaut Cuvelier, consisted in traversing the path nodes of a given ride and for each one, check if there were associated pending demands (a demand was associated to the node corresponding to the demand origin).

Such approach was optimal for the old scenario, with a static graph and a very limited number of nodes. However, when I have moved forward and I have integrated Google maps into the application, the algorithm was no longer usable and I had to define another approach.

2.5.1 Market Research

First attempts to design the new matching algorithm showed to the driver a given number of alternative paths to reach the final destination and for each of those the number of potential passengers. With this approach the driver could eventually select the path allowing the biggest number of passengers, however, no freedom was left to the driver in defining completely new paths to find as many passengers as possible.

Looking at the way other carpooling applications, such as BlaBlaCar[10], deal with the problem (from the end user point of view), I have realised that there was one assumption that I had to change in order to reach the optimal solution. Indeed, during my first meetings with Professor Mathy and my supervisor Tom Barbette, we were always focusing on the actual path followed by a given driver and, based on that, the application was presenting him with possible passengers to pick up along the road.

A small market research I have run through 20 respondents, showed that one of the biggest concerns of a driver using a carpooling application is filling up the car and share expenses. As long as the original destination is reached at the wanted time, the driver does not really care about which path to follow.

With this result in mind I have designed my matching algorithm and the core of my Android application.

2.5.2 How It Works

When a driver creates a ride, the back-end communicates with Google API to find the quickest path to reach the destination (accepting that it could be different from what the driver has originally in mind). This information is stored as a list of nodes in a local graph and used later on to find potential passengers.

When it is time to fill up the car, the driver will have access to a view, with a map simply showing ride origin and destination and pins identifying potential passengers. But let's see more in details how this is done.

Before starting the computation the algorithm performs some automatic filtering of the pending demands, only selecting those with an arrival time which falls into a time range of 24 hours with respect to the ride start time (such a big time slot is used for testing purposes but it is reasonable to assume that using a time range of 3 hours would provide good matches). In other words, being D the set of pending demands, the filtering selects all demands `d` satisfying the following condition:

$$\forall d \in D : d.arriv\_time \in [ride.start\_time - 12hrs, ride.start\_time + 12hrs]$$

Once the set of original pending demands is filtered, the matching algorithm computation starts. The algorithm has 2 nested for cycles, one iterating over the list of edges of the ride path and one iterating over the list of pre-filtered demands. For every edge the algorithm checks whether the demand origin is within an acceptable distance from the edge origin or destination (this distance, unless otherwise stated, defaults to 2 kms). If this is the case, the demand is entered into a list of potential matches which is then inspected to identify whether also the demand destination is within the acceptable distance from one of the ride edges.

In other words, as long as origin and destination of a passenger demand are not too far away from the driver (assumed) riding path, the algorithm identifies the passenger as potential. Even though the algorithm assumes that the driver is riding along the shortest path provided by Google (which is not necessarily the one he will follow), the radius used to find potential passengers is a mechanism which overcomes pretty well this fake assumption.

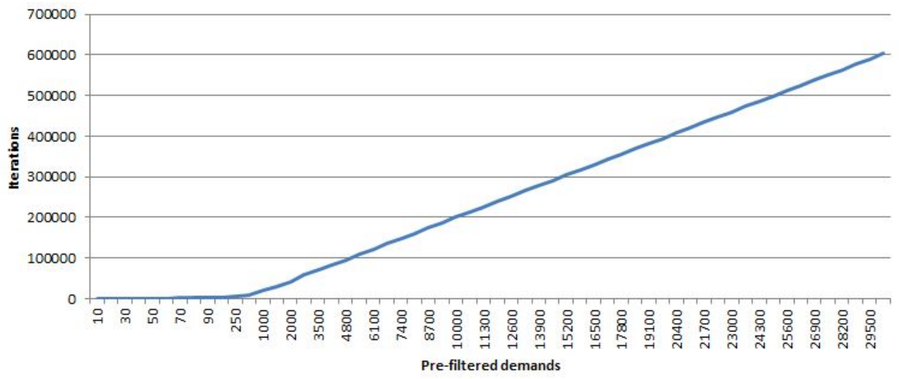


Figure 2.3: Matching algorithm iterations, function of existing demands

This algorithm, with very small modifications is also used to compute potential drivers for a given passenger. Since the approach is dual to what has been illustrated for the driver case, it will not be described. However, the fact that the passengers can also dynamically find potential drivers is a big enhancement compared to the previous version of the application.

After I designed the algorithm I tried to compute its complexity, focusing on the worst case scenario. According to Wikipedia, the Université de Liège counts roughly 30 thousands people among students, administrative and academic staff[11]. Observing the average number of edges of a given path during my tests, I can state that this number is close to 20. I can therefore represent the number of iterations of my matching algorithm, function of the total number of demands, with an upper bound of 30 thousands units (worst case, where every person has created a demand falling in the 24 hours time slot described above). As shown in figure 2.3 the number of algorithm iterations grows linearly with respect to the number of demands, which is still acceptable in terms of computational complexity.

2.5.3 Geopy

To compute the distance between two nodes, in the context of the matching algorithm, I have used a python library able to compute distances between geo-locations. Its name is Geopy[14].

Geopy can calculate geodesic distance between two points using the Vincenty distance[15], and this is exactly how the matching algorithm checks whether two geo-locations are close enough.

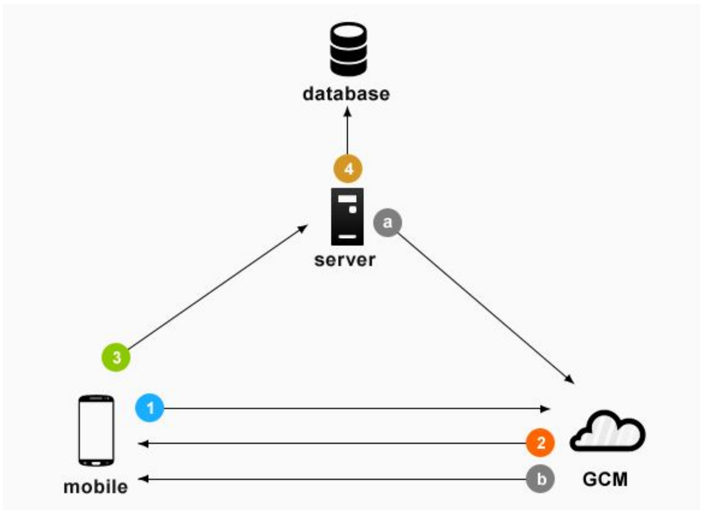


Figure 2.4: Google Cloud Messaging - Architecture

2.6 Real-Time Notifications

In order to design a helpful application, ready to be used, from the very beginning I have worked at integrating a notification system which would be able to alert users about main events such as ride demand being accepted, driver about to leave, etc.

Browsing on Google I realised that a new cross-platform messaging solution has recently been launched. Its name is Firebase Cloud Messaging[16] (FCM) and it replaces the previous solution called Google Cloud Messaging[17] (GCM). After some research I have eventually realised that little testing has been done for backends implemented using Django and I therefore had to use the older Google Cloud Messaging, despite the recommendations from Google to start using the new solution.

The notification system is very simple and its main steps are shown in figure 2.4. In details these are:

1. First Android device sends sender and application ID to GCM server for registration.
2. Upon successful registration GCM sever issues a registration ID to the Android device.
3. After receiving the registration ID, the device will send it to CarOnTheHill

server.

4. CarOnTheHill server will store registration ID in its database for later usage.

Whenever a push notification is needed, CarOnTheHill server sends a message to GCM server along with device registration ID (letter a in the picture).

GCM server will deliver that message to the corresponding mobile device using its registration ID (letter b in the picture).

Notifications are sent in the circumstances listed in table 2.3.

Notifications
If a driver proposes a ride to a passenger, the latter is notified and taken to the demand view, to be able to check the driver profile and eventually accept the lift.
If a driver accepts or rejects a lift demand the passenger is notified.
If a driver has no more seats available and there are outstanding passengers requests, all these passengers are automatically rejected and notified.
If a driver leaves, all accepted passengers are notified.
If a driver is asked a lift he is notified and taken to the ride view, to be able to check the passenger profile and eventually accept the request.
If a driver cancels a ride, all related passengers are notified that the ride was cancelled (these are either those who had asked a lift to the driver or to whom the driver had offered a seat).
If a passenger accepts or rejects a lift, the driver is notified.
If a passenger accepts a ride (or a driver accepts a lift request) and he has other outstanding requests (for the same demand) the related drivers are informed that the passenger has found a driver.
If a passenger cancels a lift all related drivers are notified that the demand was cancelled (these are either those to whom the passenger had asked a lift or drivers who offered a seat to the passenger).

Table 2.3: Application notifications

The code in charge of implementing the notifications system is organised in the `notifications` package. It consists of 4 classes, but clearly the most interesting ones are the following:

- **`MyGcmListenerService`**: this is the service running in the background which takes care of reproducing the notification upon reception. In this class I have implemented the code to handle the different types of notifications.

- `RegistrationIntentService`: this service registers the user device for push notifications. This registration is done in background every time the user performs the login. Initially I thought that a better approach would have been to do the push notifications registration only once at user registration time, however in this case only the device with which the user registers would receive the notifications. This other implementation allows a user to use multiple devices and still receive the notifications on each one of those.

2.7 Connectivity Monitoring

Developing the application I figured out that one critical point is to make sure that no HTTP request is attempted if no connection is available. I have already discussed about how I dealt with such problem in section 2.2.2, however I decided that I wanted a mechanism being able to be fired real-time in case of dropped connection, alerting the user about it.

I came up with the class `ConnectivityReceiver`, a receiver in charge to monitor variations of the Internet connection status (its code is reported in listing 6.2). Its functioning is very simple: it monitors the Internet connection and, if its status changes, it triggers a `onNextworkConnectionChanged` function (with connection status as parameter), implemented in the main activity class `CarOnTheHillActivity`. The function reads the connection status parameter and, in case of absent connection, takes the user to the login screen alerting him that no connection is currently available.

2.8 Errors Handling

Error handling throughout the application is performed with a combination of two mechanisms:

- Custom error messages, displayed using the `Toast` Android class for popup management;
- Custom exception classes.

.

Custom exception classes are all grouped in the **Exceptions Package**. They are all inheriting `RuntimeException` and they are equipped with a constructor accepting a string parameter which is used as error message to be displayed to the end user. Details about each custom exception and when it is commonly used are reported in table 2.4.

Custom Exception	**Description**
`MissingIdException`	Generally used when an API call is missing the ID to build the final API URL.
`WrongActivityResultId-Exception`	Used when an activity is launched by another one with the function `startActivityForResult` and the result code is unexpected.[3]
`MissingExtraException`	Across the entire application activities often load data passed by the previous one through an extra. This is done to limit the number of HTTP requests sent to the server. If the extra is not found this custom exception is used to throw an error.

Table 2.4: Custom exceptions

For what concerns error management on the backend, instead, a custom error class has been defined in order to send to the application some additional information. In particular, the server, in most of the cases, returns a JSON object containing an error code and a descriptive message about the error cause. In section 6.3, dedicated to API calls, it is possible to have additional information on all the exceptions triggered by the server.

2.9 Localisation

Android Studio offers a very easy way of localising an application. In fact all string resources can be centralised in one unique XML file which can then be translated in several languages, providing out of the box localisation.

Throughout the development of the application I was very attentive, never using hard-coded strings. The result is one XML file of less than 300 lines, written in English, containing all the text ever used throughout the application. Translating this file in French or any other language would produce an instantaneous localisation of CarOnTheHill application.

On the other side, unfortunately the backend is not yet equipped to handle multi-language functionality. However, currently the only contents coming from the backend are the location addresses, provided by Google API, and the error messages, both written in English.

[3]This approach is used when the activity being launched has to produce some data for the calling one. Once the called activity has completed its computation it passes the result to the caller with a request code (the same used by the caller to launch the activity). On resume the caller checks the returned request code to make sure to understand from which activity the result is coming from.

Chapter 3

App Structure

3.1 Introduction

The Android application code is divided into packages to better organise the code following common coding best practises. In particular, the structure, as we can see in Figure 3.1 presents 7 main packages.

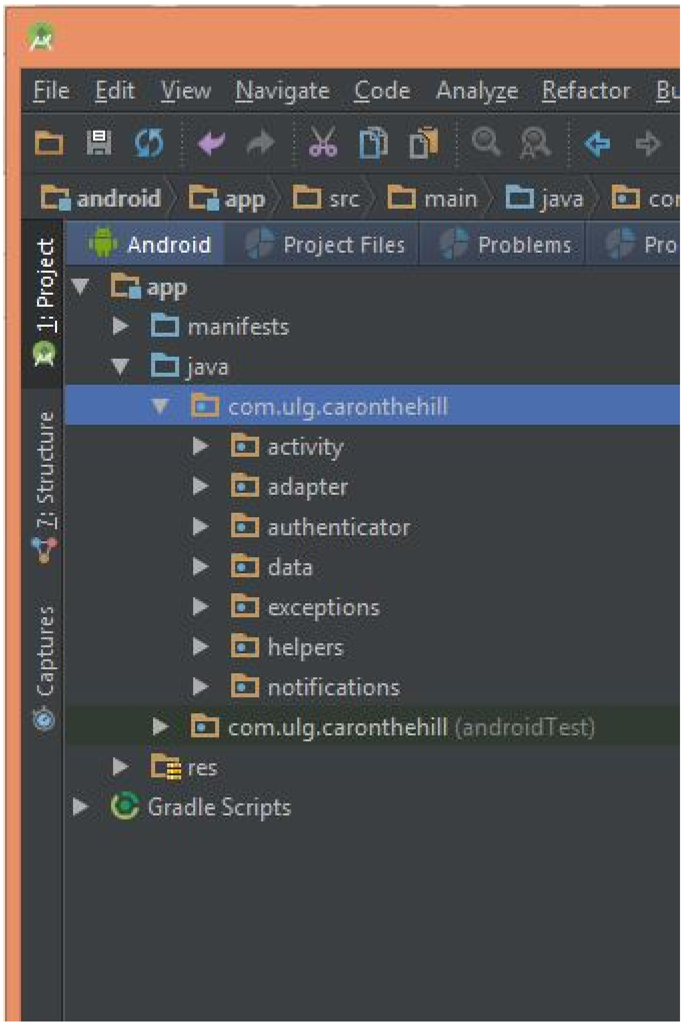
The main idea behind the organisation of the application code was to follow the Model-View-Controller (MVC) paradigm[9] as much as possible, even though the Android logic does not naively support such paradigm. With this goal in mind the various classes are mainly divided into activities, adapters and models, representing in order respectively (and with some approximation) the views, the controllers and the models of the MVC approach.

The following sections try to give additional details of the various Java classes implemented in the main packages.

3.2 Activity Package

The activity package is the core of the application. It contains all its activities, that is all its views, also grouped based on whether they regard the driver or the passenger mode.

With respect to the MVC paradigm, saying that the activity classes represent the views would not be correct. Indeed in Android the activity classes also carry on the actual logic of the application (and not simply the final rendering as it is


File Edit View Navigate Code Analyze Refactor Bu
android app src main java co
Android Project Files Problems Pro
1: Project
7: Structure
Captures

app
manifests
java
com.ulg.caronthehill
activity
adapter
authenticator
data
exceptions
helpers
notifications
com.ulg.caronthehill (androidTest)
res
Gradle Scripts

Figure 3.1: Android application code structure

supposed to be for a view). Therefore I would rather say that this package contains a combination of views and controllers.

3.2.1 Activities Hierarchy

All the activities are inherited from the base class `CarOnTheHillActivity`, derived from `AppCompatActivity`, as show in Figure 3.2. Using this structure allowed to centralise common code and variables across all the application activities.

One of the most important parts of this class is that it registers two receivers used by the application: one to monitor for Internet connectivity (see section 2.7) and the other which is in charge to handle incoming notifications (see section 2.6).

Other important parts of `CarOnTheHillActivity` are:

- A static reference to the connected user, so that its data can be accessed from everywhere in the app;
- The code to eventually hide the action bar or manage its buttons;
- The check for Google Play Services which verifies if the device has the library installed and if not it prompts the user with a message to download the library. In fact Google Play Services are required by the application for features such as Google maps and push notifications.
- A function to display an alert to the user in case of no existing connection.

3.3 Adapter Package

This package contains all custom adapters mainly used to properly represent list of data models and implement the logic behind them. As for the *Activities package*, also the classes belonging to this package cannot be properly defined pure controllers.

Table 3.1 lists the various adapters, describing their role in the application.

Where applicable the adapters implement the **ViewHolder pattern**. The ViewHolder design pattern enables to access each list item view without the need for the ids look up, saving valuable processor cycles. Specifically, it avoids frequent call of `findViewById()` during ListView scrolling, which make it nice and smooth.

3.4 Data Package

This package contains all the classes representing the model in the MVC pattern. In particular, it contains two sub-packages, one implementing the various data

Adapter Name	Description
PassengerAdapter	This adapter deals with the representation of list of `Passengership` data models. To provide sufficient information it also retrieves info related to the demand and the passenger associated to the `Passengership` object. Thanks to this adapter pending matches, that is those still to be accepted by both parties, can be accepted or rejected using *ad hoc* buttons. In case of confirmed matches, this adapter shows also a button to leave a feedback to either the passenger or the driver.
RideAdapter	This adapter is an abstract class used by other adapters dealing with `Ride` data models. It simply takes care of referencing the layout elements and update the view if a ride is removed from the displayed list.
RideDriverAdapter	This adapter is inherited from `RideAdapter` and displays a list or rides (both future and passed) to a driver. It implements the functions to find potential passengers and display existing ones. Thanks to this adapter the driver can also inform he has left or he can delete a ride.
RidePendingOfferAdapter	This adapter is inherited from `RideAdapter` and displays a list of rides in a pending state, because they have been offered to a passenger and the driver is awaiting for his confirmation. It implements the functions to confirm or reject the match.
RidePotentialDriverAdapter	This adapter is also inherited from `RideAdapter` and displays a list of potential rides to a passenger. The passenger can then decide to ask a ride to the given driver or not.
DemandAdapter	This adapter deals with the representation of a list of `Demand` data models (both passed and future). Using this adapter the passenger can display confirmed/pending drivers or find potential ones. He can also delete the request.
ULGNodesAdapter	This adapter is used to represent the list of ULg faculties in a drop-down list. It also implements a custom comparator to order the faculties, represented by `Node` data models, by name.
ViewPagerAdapter	This adapter is used for multi-tabs views.

Table 3.1: Application custom adapters

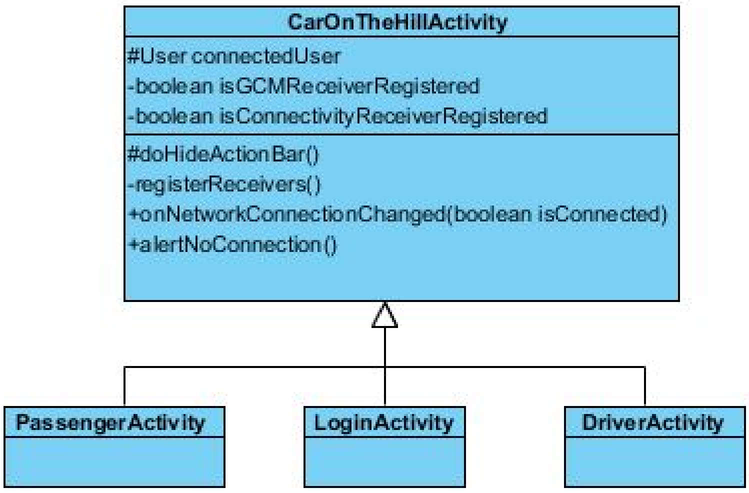


Figure 3.2: Activities hierarchy sub-schema

models used by the application and the other taking care of all API endpoints, that is the actual URLs to be used to reach the server.

For a detailed description of these sub-packages please refer respectively to Appendix 6.1 and 6.3.

Chapter 4

Use Cases

4.1 Introduction

This chapter focuses on the functionalities of the Android application, highlighting some of its key features, showing screen-shots and describing typical use cases.

The application design in many circumstances follows what has been done by BlaBlaCar. In fact, not wanting to reinvent the wheel, with the goal to maximise user usability, I looked very often at the way this leader of the carpooling industry designed its Android application and I replicated it for CarOnTheHill.

This is particularly the case for the user profile section, displayed in Figure 4.7a and the functioning of the activities to ask or create a ride.

4.2 Registration and Login

In order to register and use CarOnTheHill it is required to have a ULg email address. At registration stage a check is performed over the domain of the provided email in order to make sure that this requirement is satisfied. Further information are asked as first and last name, username (also usable to login, instead of the email) and password.

Automatic checks are performed client side to make sure that only when all fields are correctly filled the registration button is active and can be pressed. At this stage another validation is performed, for example to make sure that the two provided passwords match.

Once the registration is successful the user is redirected to the login page. Here he will have to enter email or username and password. He can check the option to be remembered so that next time the application will be launched it will automatically log him in.

After logging in the user can decide to use the application in two different modes:

- **Driver**: to create new rides and find potential passengers;
- **Passenger**: to create ride demands and find potential drivers;

Despite his choice, he will always be able to switch between the two modes, using the apposite buttons in the action bar.

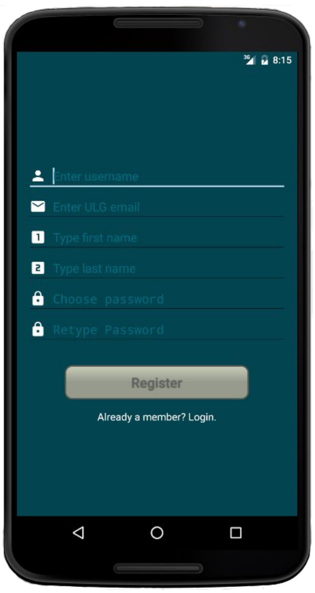
Furthermore, an *ad hoc* profile section allows the user to specify preferences or provide details about the owned vehicle. In the next sections I will review more in details all these functionalities.

4.3 Driver Mode

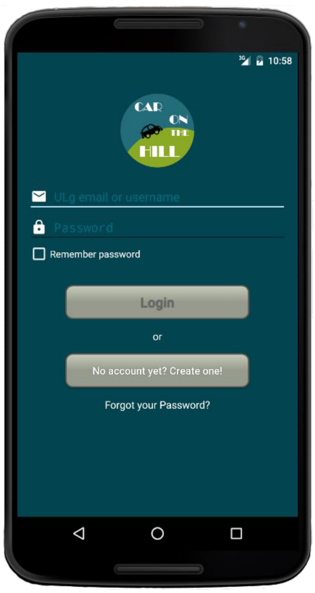
Selecting the driver mode, the user will be automatically redirected to the driver home page from which he can access to existing rides or create new ones.

The existing rides provide info on start and end location of each ride, together with the departure time and date. They are divided in two tabs:

- **Your rides**: it displays all future rides. From this tab each ride shows info on start and end location and departure time together with a set of passengers



(a) Registration View



(b) Login View

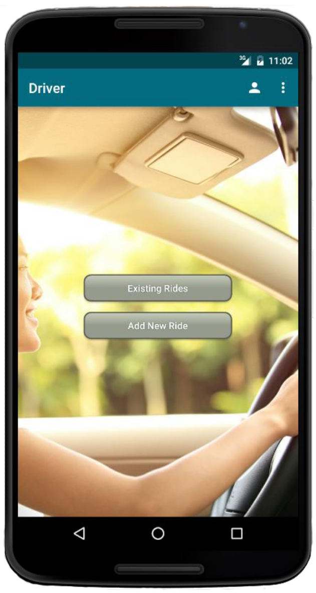
Figure 4.1: Register and login views

associated to the ride. These can be *pending*, *accepted* or *rejected*. Pending passengers are those who have asked a lift or to whom the driver has offered one; in both circumstances the other party is supposed to either accept or reject the offer/demand. Most importantly, the driver can also look for new potential passengers. Last but not least, with apposite buttons the driver can inform his passengers he is leaving or he can eventually delete the ride (if no passengers have been confirmed yet).

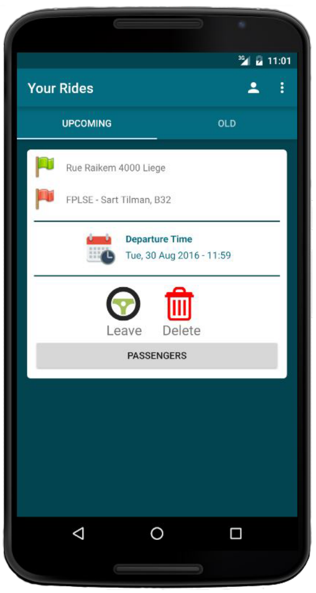
- **Old rides**: it is a list of rides which have passed or flagged as *left* by the driver. From this view the driver will be always able to access accepted passengers info and submit feedback to each one of them.

4.3.1 Add new rides

From the driver home page shown in figure 4.2a the user can add new rides. To do so he must specify origin and destination, number of free seats and departure time.



(a) Driver Home



(b) Driver existing rides

Figure 4.2: Main driver views

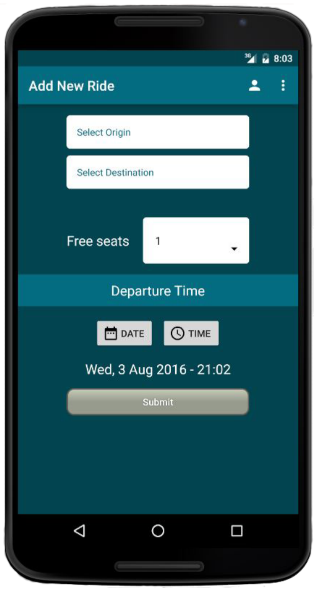
The application ensures that at least one end of the ride is one of the ULg faculties. In this case the user is presented with the list of the faculties which is hard-coded in the server database.

For generic addresses, instead, the application uses a geo auto-completion mechanism, as shown in Figure 4.3b, which uses Google API in order to help the selection of the address, but most importantly ensure consistency.

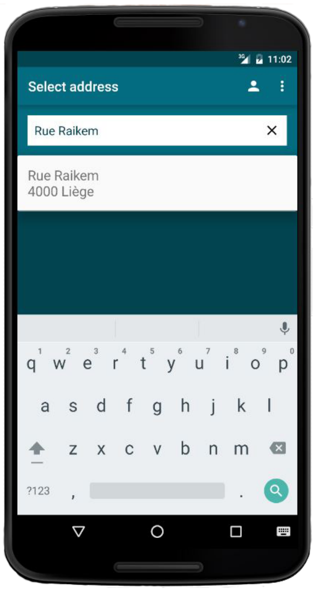
4.3.2 Find potential passengers

If a ride does not have any pending passenger demands and the driver wants to quickly find potential matches he can access to the passengers map, a Google map view where he can display potential passengers, filtering them based on distance from the ride path and their arrival tolerance, as displayed in Figure 4.4b.

As shown in Figure 4.4a, the map displays two flags identifying the ride start



(a) Add new ride



(b) Geo auto-completion for address selection

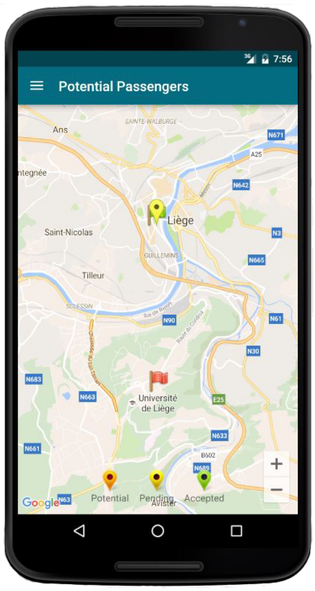
Figure 4.3: Add new ride views

and end locations and passengers are identified by pins. The map does not display only potential passengers but also pending and accepted ones. The difference is stated by the pin colour: orange for potential, yellow for pending and green for accepted.

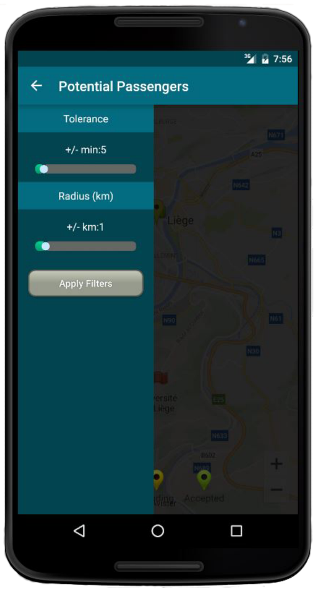
By clicking on the pin the driver can have a detailed description of the passenger demand and, by clicking on this description a new view opens up and the driver can eventually propose the ride in the case of potential passengers or accept/reject the request for pending ones.

In both cases the driver will be asked to provide a time at which he is intending to pick up the passenger.

If the results displayed on the map are not satisfactory the driver can always use the filters, located in the left hand side slider menu, and try to find more matches, adjusting their values. The filters are loaded by default with a radius of 2 kms and an arrival tolerance of more or less 5 minutes. For a detailed description of how the match between ride and demands is performed please refer to section 2.5.



(a) Potential passengers map



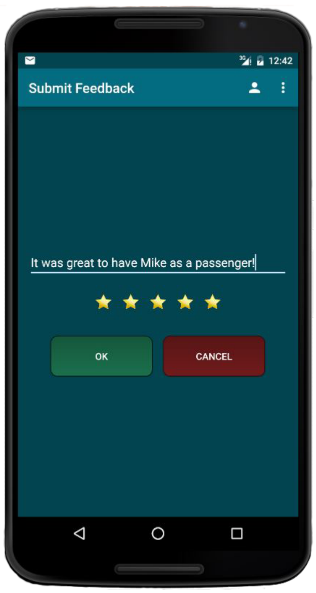
(b) Potential passengers filters

Figure 4.4: Find potential passengers

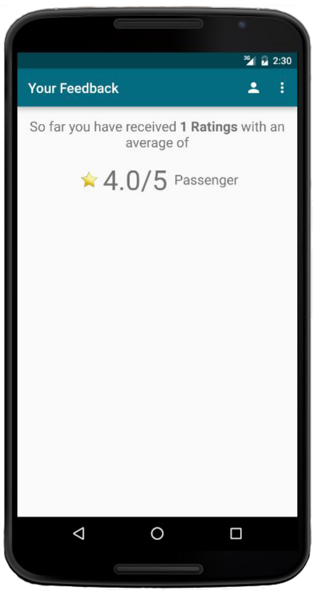
4.3.3 Submit Feedback

Once a driver indicates he has left he, and the accepted passengers, will be able to write each other a feedback. This is a 5-star vote, paired with a small description, as shown in Figure 4.5a. The application is able to distinguish between a feedback obtained as a driver and one obtained as a passenger, therefore each user will always have two ratings computed by averaging all feedback obtained as a driver and those obtained as a passenger.

At any moment the user can access to a view displaying the total number of feedback received and the average rating both as passenger and as driver, as shown in figure 4.5b.



(a) Write feedback to passenger or driver



(b) User ratings summary

Figure 4.5: Ratings

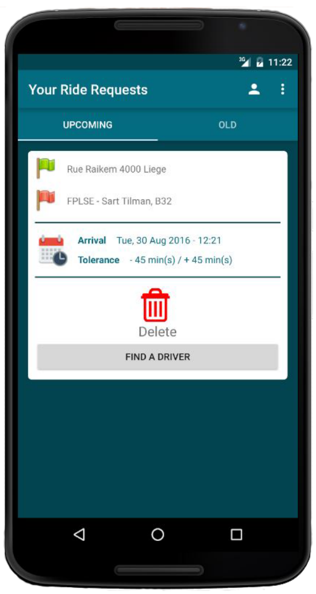
4.4 Passenger Mode

In passenger mode the user can monitor the status of existing demands and create new ones as shown in figure 4.6b.

The existing demands, following the same approach of the driver rides, are divided in two tabs:

- **Your Demands**: it displays info about future demands. For these the passenger can access to pending drivers, that is drivers who either have offered a lift or to whom the passenger has asked it. In both cases the other party is supposed to either accept or reject the demand/offer. The passenger can also try to find possible drivers without having to wait an offer, as described in more details in section 4.4.2. Finally, in case of no accepted matches the demand can also be deleted.

- **Old Demands**: it contains all past demands. From this view the passenger will always be able to access to the driver information and write a feedback.



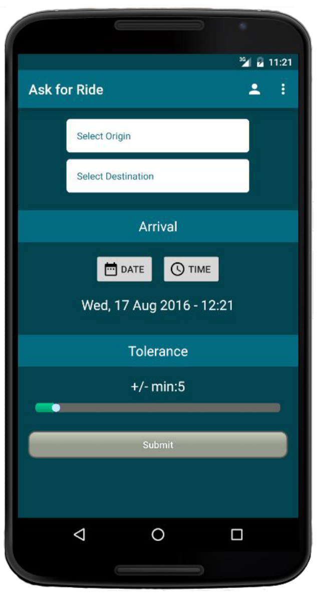
Rue Raikem 4000 Liege

FPLSE - Sart Tilman, B32

Arrival Tue, 30 Aug 2016 - 12:21

Tolerance - 45 min(s) / + 45 min(s)

(a) Passenger existing demands



Wed, 17 Aug 2016 - 12:21

+/- min:5

(b) Ask a lift

Figure 4.6: Passenger main views

4.4.1 Ask lift

In order to ask a lift the passenger has to specify origin and destination, with same approach illustrated for ride creation (see section 4.3.1), and in addition he has to specify an arrival tolerance. This is used to determine a time interval during which the passenger is accepting to reach destination.

4.4.2 Find potential drivers

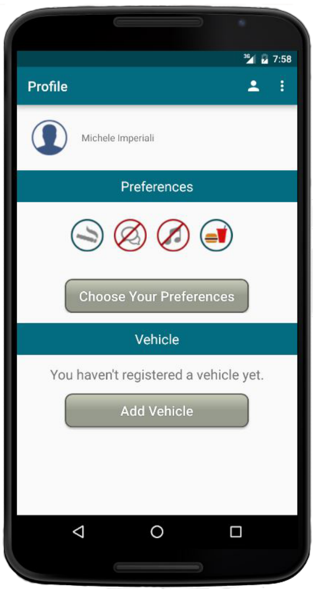
As described for drivers in section 4.3.2, also the passenger is able to find potential drivers autonomously. The mechanism is very similar to the one shown for the driver: the passenger can identify potential drivers by using two available filtering criteria, the radius, which represents the distance of demand origin and destination from a driver path (it defaults to 2 kms), and the arrival tolerance, which defaults to the one specified at demand creation time but can be overwritten here (helpful for example if the passenger realises that he must allow a bigger tolerance to find a potential driver).

4.5 User Profile

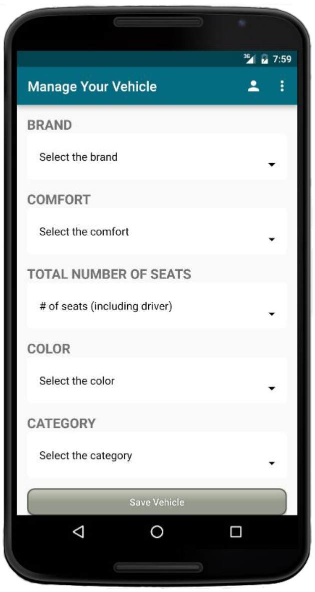
The user profile view contains a series of information about the user, like preferences and owned vehicle.

From this page the user can:

- Update his preferences, such as if he likes chatting, listening to music etc.
- Manage his vehicle information (if any), as shown in Figure 4.7b.



(a) User profile



(b) Manage vehicle

Figure 4.7: User profile management

This view is also used when a passenger or a driver wants to see additional info about a given user, an clicks on the user avatar (which for the time being defaults to a custom image).

Chapter 5

Conclusions

5.1 Final Result

As a conclusion to my work I can firmly state that I was able to implement an Android application that answers the need of any person wanting to find a lift or to fill up his car and go to university. CarOnTheHill is indeed the solution that we where thinking of when I first met with my promoter, Professor Laurent Mathy, and my supervisor, PhD student Tom Barbette, discussing about the work to be done and the project requirements.

The final Android application is the result of lots of thoughts and hard work trying to keep the code as clean and intuitive as possible, following best practises and using all the skills and the methods learnt throughout my university career.

Looking at the final result I am also satisfied about the fact that I put significant effort not only in the actual code but also in the design and user friendliness of the implemented solution. I did it following my motto which says that no matter how well your algorithm performs, the world will not use it if it is ugly.

5.2 Future works and Improvements

No matter the effort you make, there will always be something you can do to improve what you have done, and my thesis is not an exception to this phrase. Even though I think I managed to implement the most important functionalities, there are still many other that could improve the usability of the final application. In this section I try to list some of the most important ones.

5.2.1 GPS Functionality

Probably one of the biggest improvements for CarOnTheHill would be allowing the user to plan the entire trip on the application, looking at possible driving paths, editing them in order to pick up potential passengers and so further.

The biggest assumption I based my work on was indeed that the driving path was of secondary importance. What really matters, after origin and destination have been set, is finding a lift (for the passenger) and fill up the car (for the driver who wants to share the expenses). This means that if I take a good guess at the driving path (and the shortest path provided by Google must be good!), then I can go ahead and find the potential matches, without bothering about the actual path the driver will take. In other words I am accepting to face scenarios in which the driver is presented with possible matches which are too far away from his driving path, which differs considerably form the one computed using Google Maps API.

Integrating a GPS system in the application, for example, the driver could still inform the passenger on an estimated pick up time, but the GPS could alert him in real-time that the driver is approaching. Furthermore real-time congestion algorithms could be implemented to make sure the driver always takes the best and less crowded path to reach his destination.

5.2.2 Payment System

Integrating a payment system could have been so easy as to compute the fuel spent to take the passenger from origin to destination and share the expenses by paying cash, off the platform. Much more interesting solutions would involve the integration of an actual payment system (exactly like the one used by BlaBlaCar) which asks in advance to the passenger the payment before lift confirmation.

Despite the fact that the first solution requires very little implementation effort, I decided to give priority to other functionalities, like the feedback mechanism, and consequently this does not appear as one of the application features.

5.2.3 JUnit Testing

Unfortunately the application does not come with JUnit testing since the time required to implement the Android application, change and extend the backend and document the overall development efforts, forced me to leave it behind.

Chapter 6

Appendices

The appendices contain a detailed description of the application data model and the entire documentation regarding the implemented API. For every available API call I have reported the URL to be used and the available parameters, followed by a brief description.

This chapter also contains the code of some interesting classes I have described in this report.

6.1 Data Model

The Android application has its own data model which recalls what has been developed on the server side. This was done in order to optimise the exchange of information between client and server, using an *ad hoc* JSON parsing library called Gson.

In the following subsections I will document in details the data models used by the Android application and their role.

6.1.1 User

The User class contains all information related to the registered user, obtained feedback, preferences and eventually vehicle information. More in details, these info are:

- First and last name,
- ID,
- User-name,
- User preferences, such as smoking, eating, listening to music and chatting,
- Vehicle information,
- Feedback data, such as average driver and passenger rating and total number of received feedbacks.

6.1.2 Preference

This class handles all user preferences, such as whether the user likes to talk in the car, listening to music, is ok with eating or smoking in the car. Each preference has a scale from 0 to 2 and it defaults to neutral (1) unless otherwise specified.

6.1.3 Vehicle

This class contains all information regarding the user vehicle. In particular, the user can specify:

- Brand,
- Number of seats (including the driver),
- Comfort class, such as basic, normal, luxury, etc.,

- Colour, to ease being recognised,
- Category, such as Economy, 4x4, Cabriolet, etc..

6.1.4 Device

Each user is associated to a device that is stored on the server to trigger push notifications. This class handles three pieces of data:

- Registration token produced by the GCM server,
- Device ID,
- Device name, which is actually the device model.

6.1.5 Trip

The class Trip handles some information common to rides and demands. Therefore I have created this base class from which Ride and Demand are then inherited. The data collected in this class is the following:

- Resource ID and URI,
- The user the trip is associated to (driver in case of a ride and passenger in case of a demand),
- Origin and destination nodes.

6.1.6 Ride

The Ride class is inherited from Trip and contains the following additional information:

- Maximum number of free seats,
- If the ride is forward and backward or not,
- URI of the driver,
- Ride start time and date,
- Whether the ride has left and at what time,
- The ride path,
- A list of pending, confirmed and rejected passengers.

6.1.7 Ride Status

The status of a ride is managed using this enumeration. This helps identifying if the ride is potential for a passenger, if it has been proposed by a driver and awaits passenger confirmation (or viceversa it has been asked by a passenger and awaits driver confirmation), if it is accepted or rejected.

6.1.8 Demand

This class also is inherited from Trip. The additional information contained is:

- Arrival time and date,
- Arrival tolerance (earlier and later),
- List of pending, accepted and rejected drivers.

6.1.9 Tolerance

This class is used to better deal with tolerance information (and its display) for a given demand.

6.1.10 Graph

This class stores info about all ULg graph nodes, which are statically saved in the server database and retrieved the first time the app is launched. It is built using a singleton pattern and contains an hash map of nodes, for efficient and quick retrieval.

6.1.11 Node

This class contains the info of a basic node. In particular:

- Latitude and longitude coordinates,
- Address description,
- Whether the node represents a ULg faculty,
- ID and resource URI.

6.1.12 Edge

This class stores the info related to an edge of a given path. It contains:

- Edge origin and destination nodes,
- Length,
- Driving time (according to Google Maps).

6.1.13 Path

This class contains the collection of edges making a ride path, with info on the total length and time required to reach the destination, according to Google Maps.

6.1.14 Passengership

This class is used to represent matches between demands and rides. It contains information on the status of the request, whether it came from a passenger or was directly proposed by the driver and the related feedback. Its fields are:

- The ID of the passenger,
- The resource ID,
- A reference to both ride and demand objects,
- Whether it was proposed by the driver,
- Maximum number of available seats,
- Number of remaining seats,
- Whether the matching has been confirmed by both parties,
- Time at which the driver will pick up the associated passenger,
- The feedback associated to driver and passenger.

```java
public class Authenticator {
    String base64EncodedCredentials;
    static final Authenticator singleton = new Authenticator();

    private Authenticator(){}

    /**
     * Return the unique instance of the cookie
     * @return the authenticator singleton
     */
    public static Authenticator getSingleton(){
        return singleton;
    }

    /**
     * Stores locally the user credentials
     * @param userName string representing the username
     * @param password string representing the password
     */
    public void setCredentials(String userName, String password) {
        singleton.base64EncodedCredentials = "Basic " +
            Base64.encodeToString(
                (userName + ":" + password).getBytes(),
                Base64.NO_WRAP);
    }

    /**
     * Return the base64 encoding of the user credentials
     * @return string representing the base64 encoding
    * of the user credentials
     */
    public String getCredentials() {
        return singleton.base64EncodedCredentials;
    }
}
```

Listing 6.1: Authenticator class

6.2 Code Snippets

Some of the most interesting classes or code snippets are reported in this section, showing how some of the main architectural choices have been implemented.

6.2.1 Authenticator class

This class handles the user credentials, to be paired to every request sent to the server, ensuring the user authentication.

```java
public class ConnectivityReceiver
        extends BroadcastReceiver {

    public static ConnectivityReceiverListener connectivityReceiverListener;

    public ConnectivityReceiver() {
        super();
    }

    @Override
    public void onReceive(Context context, Intent arg1) {
        ConnectivityManager cm = (ConnectivityManager) context
                .getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
        boolean isConnected = activeNetwork != null
                && activeNetwork.isConnectedOrConnecting();

        if (connectivityReceiverListener != null) {
            connectivityReceiverListener.onNetworkConnectionChanged(isConnected);
        }
    }

    public static boolean isConnected() {
        ConnectivityManager
                cm = (ConnectivityManager)
                     CarOnTheHillActivity.getInstance().getApplicationContext()
                .getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
        return activeNetwork != null
                && activeNetwork.isConnectedOrConnecting();
    }


    public interface ConnectivityReceiverListener {
        void onNetworkConnectionChanged(boolean isConnected);
    }
}
```

Listing 6.2: Connectivity Recevier

6.2.2 Connectivity receiver

This receiver monitors changes of the Internet connection status.

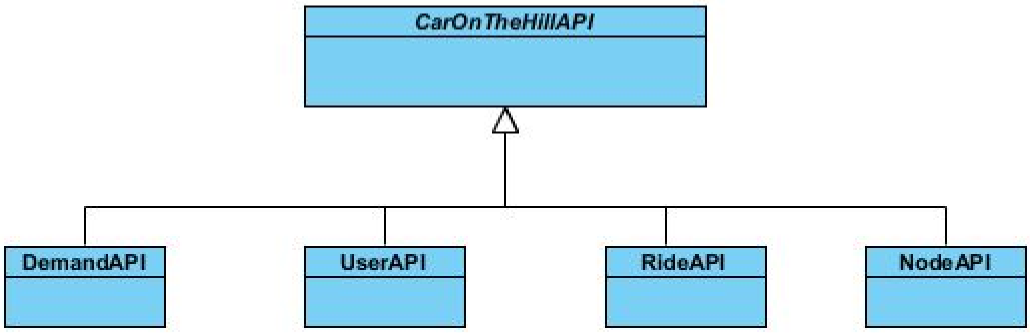


Figure 6.1: API classes hierarchy

6.3 API Calls

The API was developed using Tastypie, a web-service API framework for Django. This appendix lists all API calls implemented in the context of the Android application.

The API calls are divided among several classes, all inherited from the abstract class CarOnTheHillAPI, which defines all the API basic endpoints and some commonly used functions such as the one to append query parameters to a given request. The different API classes contain the functions returning the various API URLs to be used across the application in order to communicate with the server. The classes structure is displayed in figure 6.1.

Whenever possible the API calls return custom error messages through a JSON object made of two fields:

- code: the HTTP error code;
- message: the HTTP error message.

The following subsections are used to document in detail every API call with its expected behaviour.

6.3.1 Node API

This API class handles all requests dealing with graph nodes stored on the server. It allows some filtering, based on whether we are only interested in retrieving ULg nodes and also based on latitude and longitude coordinates.

Get node GET api/v1/node/:nodeID/
It allows the connected user to retrieve information about a given graph node. For more details, check out section 6.1.11.

Get all nodes GET api/v1/node/
It allows the connected user to retrieve information about all nodes stored on the server. The request allows some filtering based on parameters reported in table 6.1.

Request Parameters	
Parameter	Description
is_ulg	If specified and set to 1, the server will only return nodes representing ULg faculties
latitude	If specified, the server will only return nodes with the given latitude
longitude	If specified, the server will only return nodes with the given longitude

Table 6.1: All nodes request - Optional parameters

6.3.2 User API

This API class handles all requests regarding user registration, login and profile info. It also deals with device registration to enable push notifications.

Register User POST api/v1/register/
It allows to register a user providing first and last name, email, username and password.

In table 6.2 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
403	Username already in use.
403	Email already in use.
403	Missing required data.

Table 6.2: User Registration - Exceptions

Get Profile GET api/v1/profile/
It returns information about the connected user (including private info), such as:

- Joined and last login date
- First and last name
- ID and Resource URI
- User-name
- User preferences, such as smoking, eating, listening to music and chatting
- Vehicle information
- Feedback data

Get User GET api/v1/user/:userID/
It returns public information about a given user. Currently there is no distinction between the data gathered by this call and *Get Profile* but the two calls are in place for future enhancements.

Update Profile POST api/v1/profile/update/
It allows the connected user to update his profile info. At this stage the only information that can be updated are the user preferences and vehicle information. It accepts as payload a User object model with the updated information.

Register Device POST api/v1/device/register/
It allows the connected user to register his device for push notifications. Currently it is done automatically in the back-end at user login time. To perform the registration, the device sends the following information to the server:

- Device name (optional)
- Registration token, obtained by Google Cloud Messaging server
- Device model

Unregister Device POST api/v1/device/unregister/
It allows the connected user to unregister his device and stop push notifications. The device identifier must be passed as request payload.

6.3.3 Ride API

This API class deals with all requests related to a ride, that is ride creation and deletion, finding potential passengers and manage pending requests.

Register ride POST api/v1/ride/register/
It allows the connected driver to register a new ride. The request payload is a Ride object model.

In table 6.3 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
403	Missing required data.

Table 6.3: Ride Registration - Exceptions

Get ride GET api/v1/ride/:rideID/
It allows the connected user to get information about an existing ride. Currently the data can be read by any connected user (not just the object owner).

This request also has optional parameters, as shown in table 6.4 to be used in order to fully retrieve some related resources, such as the ride path or confirmed, rejected and pending passengers.

Request Parameters	
Parameter	Description
full_path	By passing the parameter full_path set to 1, the retrieved data will contain detailed information about the ride path, with a full list of nodes.
full_passengership	By passing the parameter full_passengership set to 1, the retrieved data will contain detailed information about passengers related to the ride and whether they are confirmed, pending or rejected.

Table 6.4: Ride request - Optional parameters

Get all rides GET api/v1/ride/

It allows the connected user to get a list of all owned rides.

This request also has optional parameters, as shown in table 6.5 to be used in order to filter the list of rides, for example only getting future ones.

Request Parameters	
Parameter	Description
start_time_gte	If provided, the server will only return rides with a start time greater than the one specified (datetime format: YYYY-MM-DDTHH:MM:SS).
start_time_lte	If provided, the server will only return rides with a start time less than the one specified (datetime format: YYYY-MM-DDTHH:MM:SS).

Table 6.5: All rides request - Optional parameters

Delete ride DELETE api/v1/ride/:rideID/cancel/

It allows the connected user to delete an existing ride. At the moment the application does not allow to delete a ride in two occasions:

- If the drive is marked as left;
- If the drive has confirmed passengers;

In table 6.6 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
404	The ride does not exist.
403	The ride is marked as "has left".
403	The ride has confirmed passengers.

Table 6.6: Delete Ride - Exceptions

Get potential passengers GET api/v1/ride/:rideID/potential_passengers/

It allows the connected driver to find potential passengers for a given ride. Potential passengers are identified based on two parameters, as reported in table 6.7.

In table 6.8 there are all custom errors returned by this API call.

Request Parameters	
Parameter	Description
radius	Distance of passenger origin and destination locations with respect to the ride path (this defaults to 2 kms). The bigger the radius, the more the identified potential passengers.
tolerance	The passenger arrival tolerance. The bigger the tolerance, the more the identified potential passengers (defaults to 10 mins earlier/later).

Table 6.7: Potential passengers request - Optional parameters

Exceptions	
Code	Reason
403	Ride does not belong to connected user.
403	Cannot find potential passengers for a ride marked as left.
403	Cannot find potential passengers for a ride which has passed.
403	Cannot find potential passengers for a ride with no more free seats.

Table 6.8: Get Potential Passengers - Exceptions

Confirm Passenger POST api/v1/ride/:rideID/confirm_passenger/:passengershipID/
It allows the connected user to confirm a passenger. The request carries a payload with a datetime object representing the pick up time proposed to the passenger (labelled `pickup_time`).

In table 6.9 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
404	The match the user tried to confirm does not exist.
404	The ride related to the match does not exist.
403	The driver cannot confirm his own offer.
403	The passenger cannot confirm his own request.
403	Cannot confirm a match which has already been confirmed.
403	Cannot confirm a match which has been previously rejected.

Table 6.9: Confirm Passenger - Exceptions

Confirm Driver GET api/v1/ride/:rideID/confirm_driver/:passengershipID/
It allows the connected user to confirm a driver.

In table 6.10 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
404	The match the user tried to confirm does not exist.
404	The ride related to the match does not exist.
403	The passenger cannot confirm his own request.
403	Cannot confirm a match which has already been confirmed.
403	Cannot confirm a match which has been previously rejected.

Table 6.10: Confirm Driver - Exceptions

Reject Match GET api/v1/ride/:rideID/reject/:passengershipID/
It allows the connected user to reject a match, that is either a pending lift demand or a pending ride offer.

In table 6.11 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
404	The match the user tried to reject does not exist.
404	The ride related to the match does not exist.
403	Cannot reject a match which has already been rejected.
403	Cannot reject a match which has been previously confirmed.

Table 6.11: Reject Match - Exceptions

Offer ride POST api/v1/ride/:rideID/propose/:passengershipID/
It allows the connected driver to propose a ride to a potential passenger and awaits for his answer. The request carries out a payload with a datetime object representing the pick up time proposed to the passenger (labelled `pickup_time`)

In table 6.12 there are all custom errors returned by this API call.

Leave ride GET api/v1/ride/:rideID/leave/

It allows the connected driver to inform he has left and therefore send a push notification to accepted passengers so they get ready to be picked up.

In table 6.13 there are all custom errors returned by this API call.

6.3.4 Demand API

This API deals with all requests regarding ride demands, that is demand creation and deletion, finding potential drivers and similar.

Register demand POST api/v1/demand/register/
It allows the connected passenger to register a new lift demand. The request payload is a Demand object model.

Exceptions	
Code	Reason
404	The demand to which the driver wants to propose the ride does not exist.
404	The request does not contain a pick up time JSON payload, or the format is not valid (format must be: YYYY-MM-DDTHH:MM:SS).
403	Cannot propose a ride to a demand already matched by another ride.
403	The driver cannot propose a lift more than once to the same demand.
403	Cannot propose a lift with a ride marked as left.
403	Cannot propose a lift with a ride with passed start time.
403	Cannot propose a lift if the ride has no more empty seats.

Table 6.12: Propose Ride - Exceptions

Exceptions	
Code	Reason
404	The ride does not exist.
403	The ride is already marked as left.

Table 6.13: Leave Ride - Exceptions

Get demand GET api/v1/demand/:demandID/
It allows the connected user to get information about an existing lift demand. Currently the data can be read by any connected user (not just the resource owner).

This request also has optional parameters to be used in order to fully retrieve some related resources, such as confirmed, rejected or pending lift requests or ride offers, as shown in table 6.14.

Request Parameters	
Parameter	Description
full_passengership	By passing the parameter full_passengership set to 1, the retrieved data will contain detailed information about drivers related to the ride and whether the lift is confirmed, rejected or still pending.

Table 6.14: Get demand request - Optional parameters

Get all demands GET api/v1/demand/
It allows the connected passenger to get a list of all owned ride demands.

Delete demand DELETE api/v1/demand/:demandID/cancel/
It allows the connected passenger to delete an existing demand.

In table 6.15 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
404	The demand does not exist.
403	Cannot delete a demand already accepted/rejected.

Table 6.15: Delete Demand - Exceptions

Get potential drivers GET api/v1/demand/:demandID/potential_drivers/
It allows the connected passenger to find potential drivers for a given demand. Potential drivers are identified based on two optional parameters, as reported in table 6.16.

In table 6.17 there are all custom errors returned by this API call.

Request Parameters	
Parameter	Description
radius	Distance of passenger origin and destination locations with respect to the ride path (this defaults to 2 kms). The bigger the radius, the more the identified potential drivers.
tolerance	The passenger arrival tolerance. The bigger the tolerance, the more the identified potential drivers (defaults to the tolerance provided at demand creation time).

Table 6.16: Potential drivers request - Optional parameters

Exceptions	
Code	Reason
404	The demand does not exist.
403	Cannot find potential drivers for a demand already accepted.
403	Cannot find potential drivers for an old demand.

Table 6.17: Potential Drivers - Exceptions

6.3.5 Passengership API

This API class only implements the request to submit a feedback related to a match.

Submit Feedback POST api/v1/passengership/:passengershipID/feedback/
It allows the connected user to submit a feedback for the driver or passenger associated to the given match. The request payload is a Passengership object model.

In table 6.18 there are all custom errors returned by this API call.

Exceptions	
Code	Reason
403	Missing required data (e.g. the vote or the description).
403	The user has already written a feedback.
403	The user is not allowed to write this feedback.

Table 6.18: Feedback - Exceptions

Bibliography

[1] Cuvelier Thibaut, June 2015, Developing a website for ULg carpooling.
http://www.montefiore.ulg.ac.be/~tcuvelier/?page=projects#car-on-the-hill

[2] Vagrant enables users to create and configure lightweight, reproducible, and portable development environments.
https://www.vagrantup.com/

[3] SourceTree is a free Git and Mercurial client for Windows or Mac.
https://www.sourcetreeapp.com/

[4] PyCharm is a lightweight IDE for Python with support of Web development with Django framework.
https://www.jetbrains.com/pycharm/

[5] Android Studio: the official IDE for Android.
https://developer.android.com/studio/index.html

[6] Django: web framework.
https://www.djangoproject.com/

[7] Gson: an open source Java library to serialise and deserialise Java objects to (and from) JSON.
https://github.com/google/gson

[8] Tastypie: a web-service API framework for Django.
https://django-tastypie.readthedocs.io/

[9] The Model-View-Controller (MVC) design pattern.
http://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

[10] BlaBlaCar: market leader in carpooling solutions.
https://www.blablacar.be

[11] Université de Liège: the Wikipedia page.
https://en.wikipedia.org/wiki/University_of_Li%C3%A8ge

[12] Representational state transfer: architectural style of application programming interfaces.
https://en.wikipedia.org/wiki/Representational_state_transfer

[13] Lazy loading: design pattern commonly used in computer programming to

defer initialisation of an object until the point at which it is needed.
https://en.wikipedia.org/wiki/Lazy_loading

[14] Geopy is a Python 2 and 3 client for several popular geocoding web services.
https://pypi.python.org/pypi/geopy

[15] Vincenty's formulae are two related iterative methods used in geodesy to calculate the distance between two points on the surface of a spheroid.
https://en.wikipedia.org/wiki/Vincenty%27s_formulae

[16] Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably deliver messages at no cost.
https://firebase.google.com/docs/cloud-messaging/

[17] Google Cloud Messaging (GCM) service handles all aspects of queueing of messages and delivery to client applications running on target devices.
https://developers.google.com/cloud-messaging/