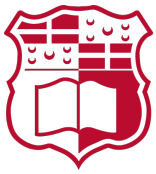
GetALift

Mobile application project aimed at connecting people to make daily trips.

Christian Colombo

Collaborators :
Loan AUBERGEON
Argann BONNEAU
Laouenan LE CORGUILLE
Alexandre DONY
Charly JONCHERAY



L-Università ta' Malta

Faculty of ICT
University of Malta
Malta
August 2018

Part I

Project Description

Chapter 1

High level

The project consists in creating a mobile application aimed at connecting people to make daily trips. This app is close to other apps carpool like BlaBlaCar for example, but it favors mainly course and regular trips. The application is part of the eco-responsibility domain. Indeed, we often notice that vehicles traveling in cities contain only one person. Carpooling offers significant advantages: to meet new people, to divide the journey price by the number of passengers, and of course to reduce the number of cars on the island, so the pollution but also the amount of traffic jams. The car traffic will be smoother and you will spend less time on the road. Indeed, the traffic on the island of Malta is difficult, the roads are narrow and old. The infrastructure of the Maltese roads is not suitable for such a high number of cars, especially in the summer when the island welcomes many tourists, 2.2 million expected for 2018.
The app favors drivers. They must not wait or make a detour, theoretically of course, free to them to be accommodating. It is the passenger who must be at a given time and a given position. Knowing that the driver and the passenger will have a different path, the application will find the closest driver and calculate the distance to reach the meeting point.**CCcomment: if would be great if you could add some figures and examples of maps**
The application is intended to connect drivers and passengers only, it will not deal with the management of payments, if any, or the management of places. The app will track driver's journeys and suggest the one that's best for users seeking a ride. The application must be simple and intuitive.

Chapter 2

Technical description

This application will be available on iOS and Android platform. The client-server environment refers to a mode of communication across a network between several programs: one, qualified as a client, sends requests; the other , qualified as server wait for and respond to customer requests.

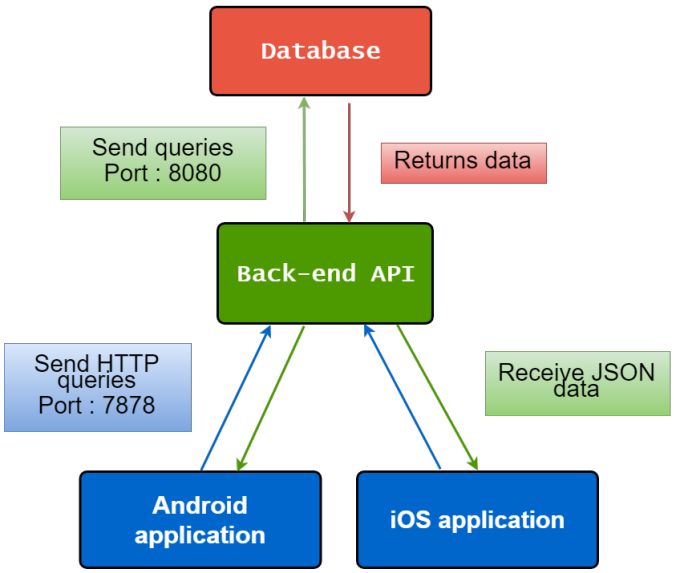


Figure 2.1: Diagram showing the architecture of the project

This chapter is divided in 5 majors subparts :

- Back-end API
- Database settings
- How to use
- Search algorithm
- Application design

2.1 Back-end API

The back-end server is an API using the NodeJS (JavaScript) technology. It is related to a database using MySQL to store the data of the application. The clients send HTTP request to the server, which answer

back some datas. This is how the clients and the server communicate. The chapter 3 list all of the functions available on the back-end API.

2.2 Database settings

The database is using MySQL and PHPMyAdmin to interact with. The database is composed of 8 tables :

- **User** : This table store all the users data.
  - id : The ID of the user
  - username : The user's username
  - password : User's password
  - name : User's first name
  - surname : User's last name
  - email : User's email
  - mobileNumber : User's mobile number
  - isVerified : Is the user verified ?
- **Route** : This table store all the routes created by the users.
  - id : The ID of the route
  - startingPoint : The geographical point at the beginning of the route
  - endPoint : The geographical point at the end of the route
  - driver : The user id that created this route
  - originAdress : The adress of the origin of the route
  - destinationAdress : The adress of the destination of the route
  - distance : The distance between the origin and the destination of the route (in meters)
  - duration : The duration of the route (in seconds)
- **RouteDate** : This table associate a route with a date. This date represents the starting time of the driver.
  - id : The ID of the route meta line
  - route : The route ID linked to this meta
  - route_date : First date of the repeat. It's assumed to be every week starting from this day
  - weekly_repeat : Is this route must be repeated every week ?
- **RoutePoints** : This table is the heaviest of the database. It stores every key points of every routes. Routes are divided in RoutePoints, which fits the path of the driver.
  - id : The ID of the route point
  - route : The route ID linked to this point
  - point_rank : It's the order of this point in the point list of the route
  - point : The geographical coordinates of this point
  - square_id_lng : The longitude id of the square of the routePoint
  - square_id_lat : The latitude id of the square of the routePoint
  - seconds_from_start : An estimation of the number of seconds passed since the start of the route
- **Ride** : A ride is an instance of a route.

  - id : The ID of the ride
  - route : The route attached to this ride

- **Rating** : This table store the rates of the users.
  - id : The comment ID
  - author : The user ID of the author of the comment
  - target : The user ID of the target of the comment
  - ride : The ride ID linked to this comment
  - stars : How many stars did the author reward the target ?
  - comment : The text content of the comment
  - postDate : The date of the rate

- **Passenger** : This table associate a passenger to a driver's route.
  - id : The ID of the row
  - ride : The ride ID linked to the passenger
  - passenger : The passenger ID linked to the route

- **FavoriteRoute** : This table is used to add some favorites routes.
  - id : The ID of the row
  - routeId : The route attached to this save
  - userId : The user attached to this save

2.3 How to use

2.3.1 Set-up the backend API

First, to set up the backend API, you need to be be connected to the remote server, by ssh. To connect by ssh, you have to write in a command prompt :

```
user@desktop:~$ ssh username@ip_server
```

Then, using git, clone the github's repository :

```
user@desktop:~$ git clone https://github.com/ccol002/getalift.git
```

Finally, you need to start the server using npm :

```
user@desktop:~$ cd getalift/getalift_backend
user@desktop:~/getalift/getalift_backend$ npm start
```

Once you will see the following displayed in the console, it will mean that the server is ready.

```
> getalift-backend@0.0.0 start /home/user/getalift/getalift_backend
> node index.js

[S] Server is listening on port 7878.
```

2.3.2 Set-up the database

To set-up the database, you need to connect to phpmyadmin through the following URL :

```
http://ip_server/phpmyadmin/index.php
```

The username is "root", and the password is also "root". Then, to create the database, import the script "database.sql" located in the getalift_backend folder.

2.3.3 Interact with the backend API

To interact with the API, use Postman. Postman is a complete API development environment, for API developers, used by more than 5 million developers and 100000 companies worldwide. With Postman, you can send HTTP request to the api, and see the response.

If the request is POST, or PUT, you need to set some keys in the body part of the request. Theses key depends of the route of the request, you can find all theses parameters details in the getalift_backend/index.js file.

Note : The API is securised with json web token. For all the requestw made to the API, you need to set a key "x-access-token" in the header part of the request, and the value of this key have to be a valid token. A valid token is delivered in the response of successful authentication request to the API. There is only 2 routes that can be reached without token, the authentication and the register route.

2.4 Search algorithm

To find the best driver's route that match with the passenger's route, a search algorithm has been created. The biggest challenge was to try to find the best route, with an answer time lower than 1 second. To launch this algorithm, we need 5 parameters :

- startLat : The latitude of the starting point of the passenger
- startLng : The longitude of the starting point of the passenger
- endLat : The latitude of the ending point of the passenger
- endLng : The longitude of the ending point of the passenger
- startDate : The date where the passenger is able to start travelling

2.4.1 Part 1 : Square ID selection

Knowing that the numbers of routes in the database can be huge, this first step goal is to refine the numbers of routes that I will analyze in step 2.

We divided the world map into squares of 500 meters square approximately. All theses squares have unique square ID. To create a unique identifier to every square, the square ID will be composed of 2 elements : the latitude ID and the longitude ID. Each time a driver add a route to the database, for every routePoints of the route, the square ID is calculated and stored into the database.

Then, we know the starting point and the ending point of the passenger. So we calculate the square ID of theses 2 points. And we look in the database, for all the routes that matches theses two conditions :

- The route have a routePoint with the same squareID than the starting point of the passenger
- The route have a routePoint with the same squareID than the ending point of the passenger

2.4.2 Part 2 : Vector direction angle

All the routes that passed the first step and that their startingDate is older than the startDate of the passenger are subjected to this second step. This step is use to hold only the routes that goes in the same direction than the passenger. Why ? Because, for example, if a driver move from A to B, and the passenger move from B to A, the driver's route will pass the first step, but this route is not interesting for the passenger because it is the inverse route.

For each routes, I create two vectors. One from the starting point to the ending point of the driver. And the other one from the starting point to the ending point of the passenger. Then, I am calculating the angle between the two vectors, and I hold only routes where the angle is less than 90 degrees.

2.4.3 Part 3 : K-d tree algorithm

All the routes that passed the second step are subjected to this third step. This is the last step. This step goal is to find the x routes which have routePoints close to the passenger key points.

A k-d tree is a space-partitioning data structure for organizing points in a k-dimensional space. K-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key. It is used to find the closest neighbors of a point. I am using it to find, for each routes, the closest routePoint of the driver depending on the passenger starting point. Idem for the passenger ending point.

Then, the routes array is sorted by totalDisance (increasing), and the 3 first elements of this array are returned using JSON. Here is the JSON format returned :

```
{
        [Array of routes]
}
```

A route is defined by :

```
{
        "id" : the ID of the route,
        "closestPointStart" : the routePoint of the route that is the closest to the
                passenger starting point,
        "closestPointEnd" : the routePoint of the route that is the closest to the
                passenger ending point,
        "distancePointStart" : the distance in meters between the passenger starting
                point and the closestPointStart,
        "distancePointEnd" : the distance in meters between the passenger ending point
                and the closestPointEnd,
        "totalDistance" : the sum of the two above distances (in meters),
        "user_id" : the user ID of the route's driver,
        "user_name" : the username of the route's driver,
        "route_date" : the starting time of the driver's route,
        "routePoints" : an array of all the routePoints of the route
}
```

A routePoint is defined by :

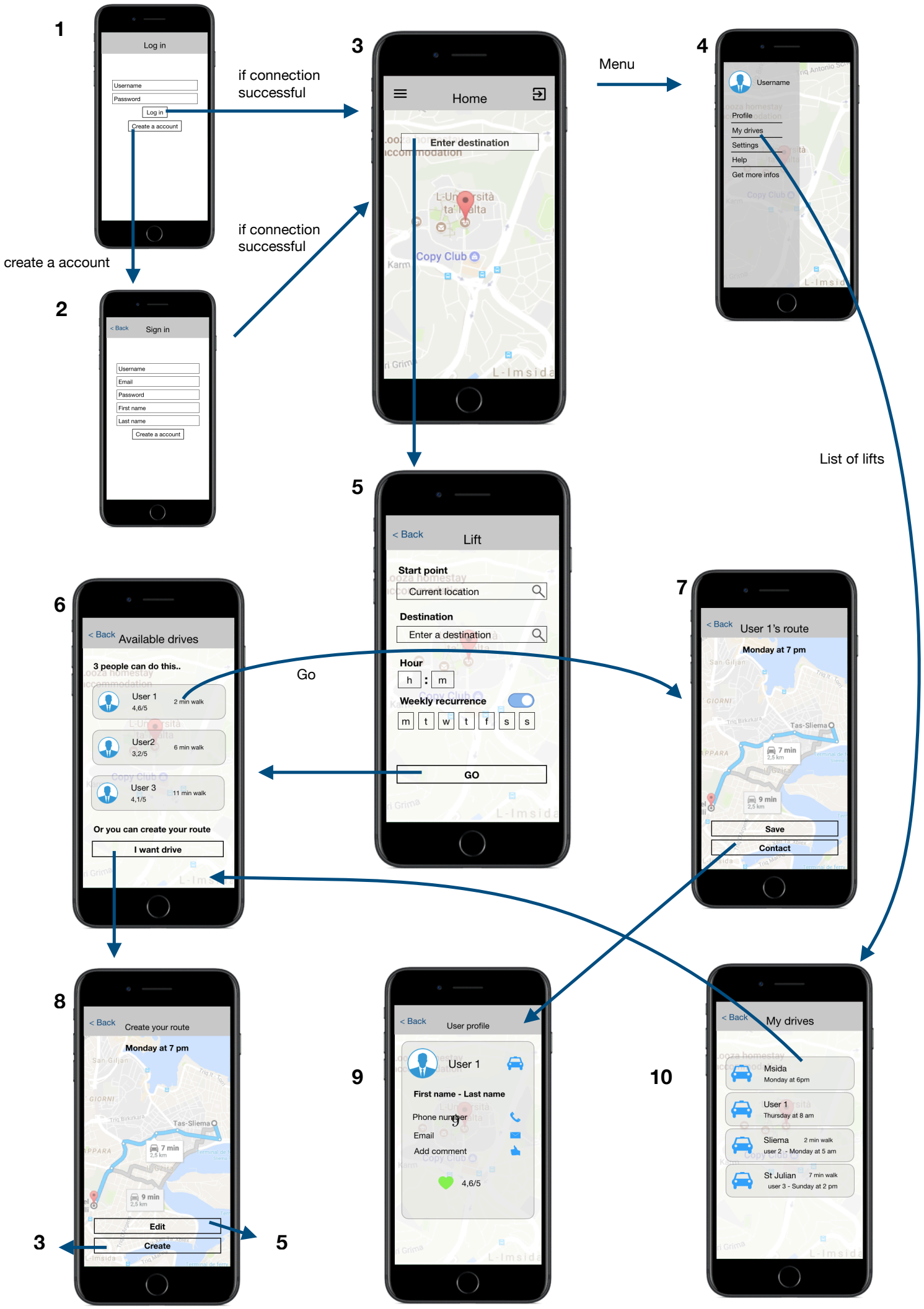
```
{
        "id" : the ID of the routePoint,
        "point" : {
                "x" : the latitude of the point,
                "y" : the longitude of the point
        },
        "route" : the route ID,
        "seconds_from_start" : an estimation of the number of seconds
                passed since the start of the route
}
```

2.5 Application design

The following page is the application design realised by Loan AUBERGEON. Here is a description the design :

- **Page 1 : Login**. This is the first page of the application. It is used to either log the user, or he also can create an account by clicking on the button "Create an account"
- **Page 2 : Sign in**. This page is use to register the user in the database.
- **Page 3 : Home**. This is the home page of the application. There is a map from Google Map in the background, focused on the user's localisation. From this page, you can search for rides, or access to the navbar.
- **Page 4 : Navbar**. This page is the navigation bar. There is 5 items in the menu : "Profile", "My drives", "Settings", "Help" and "Get more infos".
- **Page 5 : Lift**. This page is to search for a route. The user has to choose his starting point, his destination, the date and the hour, and also if there is a weekly recurrence. When the "GO" button is pressed and the form is filled, we moove to the page 6.

- **Page 6 : Availables drives**. This page list the availables drives on the database that matches the user's parameters. By clicking on a drive, the user is redirected to page 7. If there is no drive available, or if the user want to drive, he can click on the button "I want to drive", which redirect to page 8.
- **Page 7 : User x's route**. This page is to display a specific route. If the user want to take this route, he can contact the driver by clicking on "Contact". The user will be redirected to page 9.
- **Page 8 : Create your route**. This page is to display the route that the user is going to create. He can edit this route, by clicking on the "Edit" button, to change a parameter. He will be redirected in page 5. Else, he can also click on the button "Create" to create his route. He will be redirected to page 3.
- **Page 9 : User profile**. This page is to display a specific user profile. It shows some informations, such as the username, the first name and last name, the phone number, the email and also the rating of the user.
- **Page 10 : My drives**. This page is to display the drives of the connected user. If he clicks on a drive, he will be redirected to page 8.


1
Log in
Username
Password
Log in
Create a account
if connection successful
create a account
2
< Back
Sign in
Username
Email
Password
First name
Last name
Create a account
if connection successful
3
Home
Enter destination
Menu
4
Username
Profile
My drives
Settings
Help
Get more infos
List of lifts
5
< Back
Lift
Start point
Current location
Destination
Enter a destination
Hour
h : m
Weekly recurrence
m t w t f s s
GO
Go
6
< Back
Available drives
3 people can do this..
User 1
4,6/5
2 min walk
User2
3,2/5
6 min walk
User 3
4,1/5
11 min walk
Or you can create your route
I want drive
7
< Back
User 1's route
Monday at 7 pm
7 min
2,5 km
9 min
2,5 km
Save
Contact
8
< Back
Create your route
Monday at 7 pm
7 min
2,5 km
9 min
2,5 km
Edit
Create
3
5
9
< Back
User profile
User 1
First name - Last name
Phone number
Email
Add comment
4,6/5
10
< Back
My drives
Msida
Monday at 6pm
User 1
Thursday at 8 am
Sliema
2 min walk
user 2 - Monday at 5 am
St Julian
7 min walk
user 3 - Sunday at 2 pm

Part II

Features

Chapter 3

Backend API

3.1 Ready

Here is a list of the routes already availables from the backend API. All the routes below are preceded by `http://ip_server:port/api`

3.1.1 Global

- GET "/"
  - Returns "Hello World"

3.1.2 Users

- POST "/users"
  - Create a user
- POST "/auth"
  - Authenticate a user
- GET "/users"
  - This route send back every public informations about every users. It can be a bit heavy with a lot of users.
- GET "/users/:usrid"
  - This route send back the public informations about the chosen user
- PUT "/users/:usrid"
  - This route update the information about the chosen user.
- DELETE "/users/:usrid"
  - This route deletes the chosen user.

3.1.3 Routes

- GET "/routes"
  - This route send back every informations about all the routes. It can be a bit heavy with a lot of routes.
- GET "/routes/:routeid"
  - This route send back the public informations about the chosen route.
- GET "driverroutes/:driverid"
  - This route send back the public informations about all the routes from a specific driver.
- GET "driverroutesdate/:driverid"
  - To get the route where the user's id is the driver order by date.
- PUT "/routes"
  - This route create a new Route in the database. It search the optimal directions with the google maps API, in order to store the best route.
- DELETE "/routes/:routeid"
  - This route deletes the chosen route.
- POST "/routes/findTarget"
  - This route can be used in order to search for a route that match specific parameters.

3.1.4 Route Dates

- GET "/routedate/:routeid"
  - This route send back the public informations about the chosen routedate.
- GET "/routedate"
  - This route returns all the routedates for all the routes.

3.1.5 Rides

- GET "/rides"
  - This route send back every informations about all the rides. It can be a bit heavy with a lot of rides.
- GET "/rides/:rideid"
  - This route send back the public informations about the chosen ride.
- GET "/rides/route/:passengerId"
  - This route show all the routes that the passenger add to his rides.
- GET "/rides/routeId/:routeId"
  - This route show the ride id corresponding to this route id.
- POST "/rides"

  - This route create a ride in the database.

- PUT "/rides/:rideid"
  - This route update the information about the chosen ride.

- DELETE "/rides/:rideid"
  - This route deletes the chosen ride.

3.1.6 Passenger

- GET "/passengers"
  - This route send back every informations about all the passengers. It can be a bit heavy with a lot of passengers.

- GET "/passengers/:passid"
  - This route send back the public informations about the chosen passenger.

- GET "/passengers/route/:passId"
  - Query that returns the information of route where the user is a passenger order by date.

- GET "/passengers/information/:routeId"
  - Query that returns the information of passengers who are using the ride.

- POST "/passenger"
  - This route can create a passenger in the database.

- POST "/passenger/existingRide"
  - This route create a passenger for a already existing Ride where the id of the route is in parameter with the id of the passenger.

- PUT "/passenger/:passid"
  - This route update the information about the chosen passenger.

- DELTE "/passenger/:passid"
  - This route deletes the chosen passenger.

- GET "/passenger/alert/:driverId"
  - Query that returns the name of all passengers in relation to a driver (For the alert message at startup)

- POST "/passenger/alert/:driverId"
  - This route create a passenger for a already existing Ride where the id of the route is in parameter with the id of the passenger

- PUT "/passenger/alert/:passId"
  - This route create a passenger for an un-existing Ride where the id of the route is in parameter with the id of the passenger

3.1.7 Ratings

- GET "/ratings"
  - This route send back every informations about all the ratings. It can be a bit heavy with a lot of rates.
- GET "/ratings/:rateid"
  - This route send back the public informations about the chosen rate.
- GET "/ratings/:targetid"
  - This route send back the the average of rate concerning the target.
- POST "/ratings"
  - This route create a rate in the database.
- POST "/ratings/existingRate"
  - This route can modify a rate in the database if the rate already exist.
- DELETE "/ratings/:rateid"
  - This route update the information about the chosen rating.
- GET "/ratings/Comment/:targetid"
  - This route send back the comments regarding the chosen target

3.1.8 Favorite Route

- GET "/favoriteRoute/:userId"
  - This routes returns all the favorites routes of a chosen user.
- POST "/favoriteRoute"
  - This route create a favorite route.
- DELETE "/favoriteRoute"
  - This route delete a favorite route.

3.2 To Do

Currently, there is no need to add additional features to the backend API.

Chapter 4

Android applcation

4.1 Ready

- **Page 1 : Login**. This page is fully working. The user can log in or register by clicking on the "Sign in" button.
- **Page 2 : Sign in**. This page is fully working. The user can register by filling the form and pressing the button "Create an account".
- **Page 3 : Home**. The map is displayed on the background. Also, we can search for a ride, or navigate trough the navbar.
- **Page 4 : Navbar**. The navbar is fully working. The user can access to his profile, his lifts, page help, settings and logout.
- **Page 5 : Lift**. This page is working. The search is working also. The user can select his origin, his destination, and the date where he want to travel.
- **Page 6 : Availables drives**. This page is fully working. You can click on a available ride to display it in the page 7. You can also click on "I want to drive" to create your own ride.
- **Page 7 : User x's route**. This page is fully working. You can see the driver's route, and also the path between your starting point and the meeting point with the driver. Idem for the your ending point and the dropping point.
- **Page 9 : User profile**. This page is working. You access to the user's details, including first and last name, username, rating, and mobile number.

4.2 To Do

- **Page 3 : Home**. The map is not focused on the user's location.
- **Page 5 : Lift**. There is no button to enter the current location of the user in the origin field. Also, the management of the weekly recurrence is not implemented.
- **Page 8 : Create your route**. This page is not implemented currently.
- **Page 9 : User profile**. Clicking on the user's number should start a call on the phone.
- **Page 10 : My drives**. This page is not implemented currently.
- **Page 11 : Help**. This page is not implemented currently.
- **Page 12 : Settings**. This page is not implemented currently.

Chapter 5

iOS Application

5.1 How to install GetALift (GEA) on your iPhone

5.1.1 Xcode

To work on the GAL project, you have to use Xcode. Xcode downloads directly on the App Store.

To open the project on Xcode, launch Xcode and click on *File/Open* and select in the *getalift-ios* folder the *GALDev.xcworkspace* and click on *Open*

5.1.2 GALDev.xcodeprojet

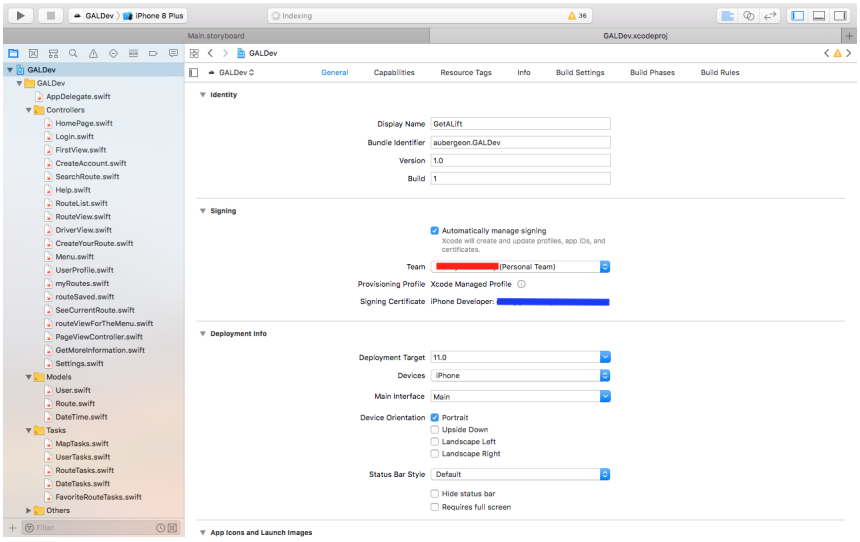


Figure 5.1: GALDev - General

When you open it for the first time you should have errors concerning the *Bundle Identifier*. Take care that it is *yourname.nameoftheproject*.

To install the GEA application on your personnal iPhone, you should create a developper account. To create a developper account, go in *Xcode Preferences* and then in *Acount*

Click on the "+" and create a new account (cf. figure 5.2).

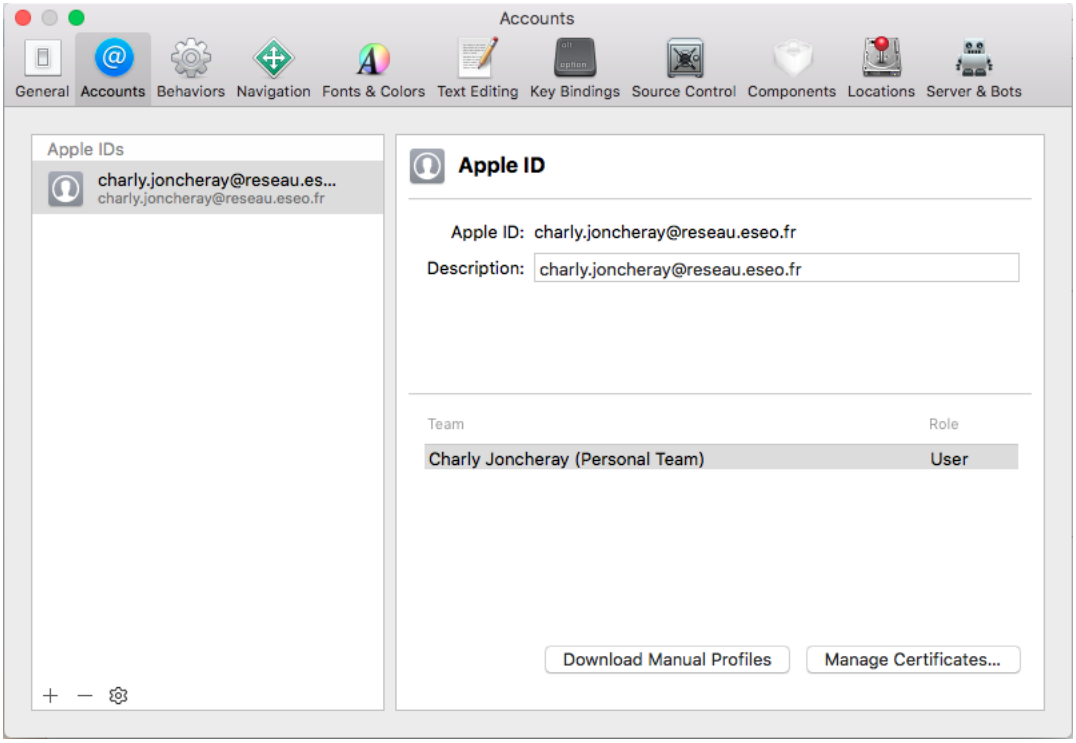


Figure 5.2: Xcode - Account

Then, in the *General* page of your project (cf. figure 5.1), select your account as being your *Team* (red box).

In the blue box, you should see your email adress.

You should have an error concerning the *NotificationBannerSwift*. It because of CocoaPods.

5.1.3 CocoaPods

CocoaPods is a dependency manager for Swift and Objective-C Cocoa projects. It has over 50 thousand libraries. It is used in the GAL application and it is necessary to install it is necessary to install it on your Mac.

How to install CocoaPods on your Mac

To install CocoaPods, you have to use the Terminal on your Mac. Open the Terminal.

In the folder that contains the project : */getalift/getalift-ios/* make sure there is the file : *Podfile*

If it is not the case, in the Terminal, in the folder containing the xcodeproject file for your project, run the command:

```
pod init
```

This will create a new text file named Podfile (no extension), with the following content:

```
# Uncomment this line to define a global platform for your project
# platform :ios, "7.0"

target 'GALDev' do

end
```

You can delete the first 2 lines of comments. Replace this code by :

```
use_frameworks! #Because we use Swift in the project
target 'GALDev' do
pod 'NotificationBannerSwift'
pod 'GoogleMaps'
end
```

Then, run the following command line in the Terminal in the folder where the *Podfile* file is located :

```
pod install
```

5.2 GetALift iOS Application

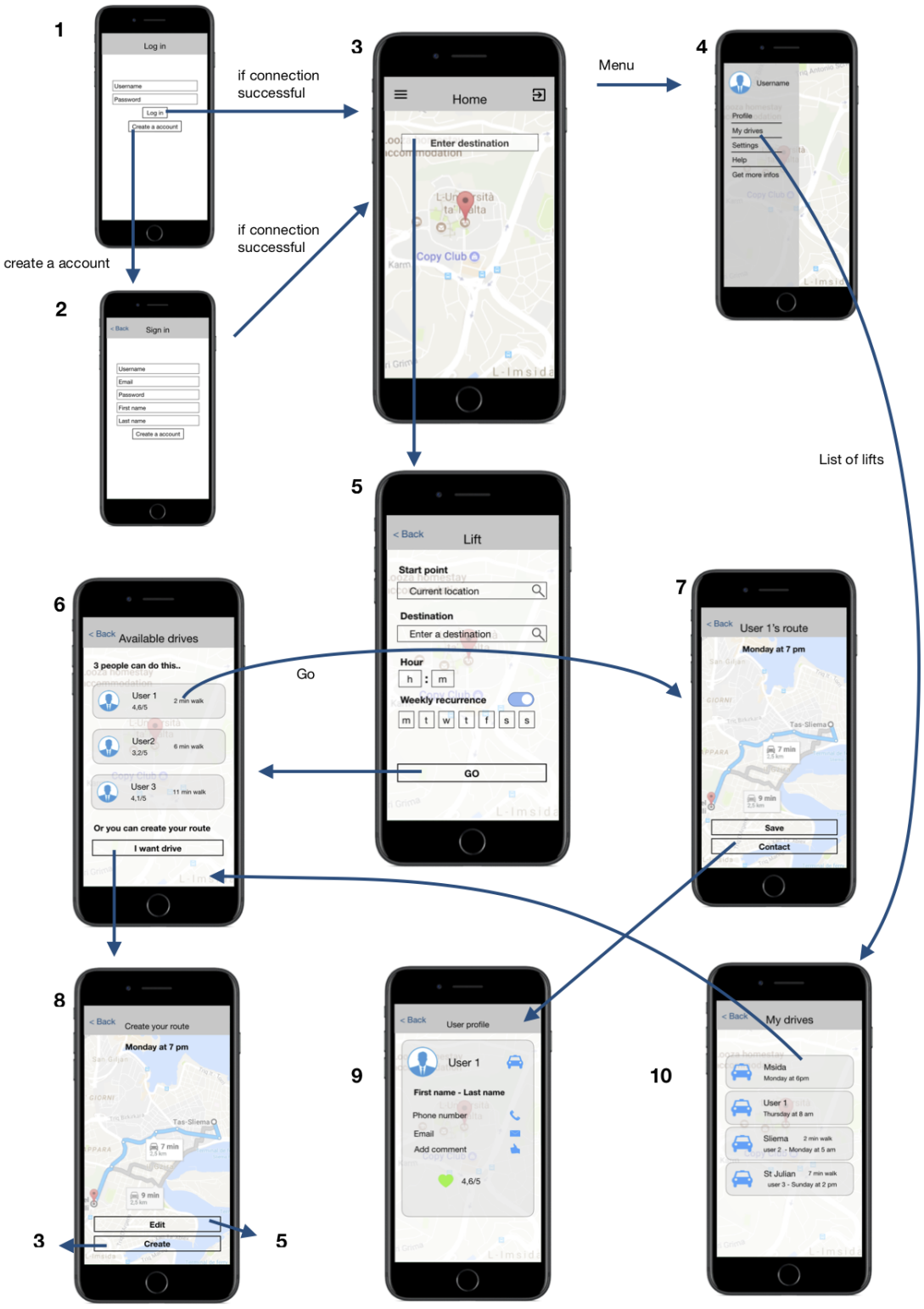


Figure 5.3: GALDev iOS Application

Don’t forget to open "GalDev.xcworkspace" in Xcode and not GALDev.xcodeproject otherwise you will have errors.

5.2.1 App preview

Application homepage

This is the home page of the app, the one that first appears when the user opens the app. He can access the authentication and account creation page by clicking on the buttons or by making a continuous gesture from left to right on the screen.

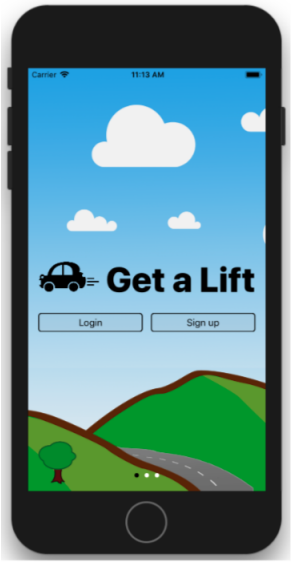


Figure 5.4: Application homepage

Authentification page

This page allows the authentication of the user. When the "Connection" button is pressed, the system checks that all the fields are filled in, otherwise it returns an error message in the form of a notification of this type:

Then the system sends the collected information to the database which verifies the authentication.

Account creation page

This page allows the creation of an user account.

When the "Sign up" button is pressed, the system checks that all the fields are filled in, otherwise it returns an error message in the form of a notification.

Then the system sends the collected information to the database that creates an account in the database.

Main page

Once authenticated, the user arrives on this page, which is the main page of the application. It provides access to the menu and search interface of a route. It also presents a map showing the current position of

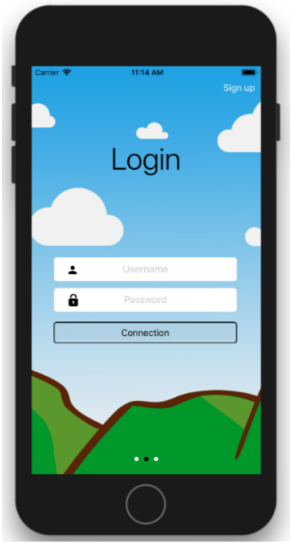


Figure 5.5: Authentification Page

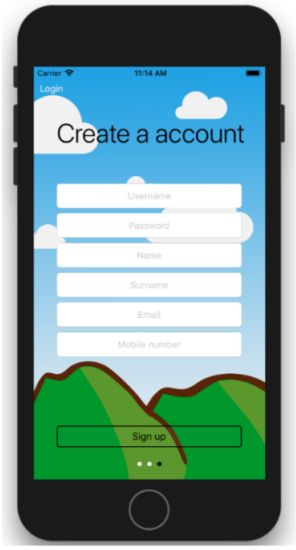


Figure 5.6: Creation Account Page

the user, after authorization of the user.

At the top left, there is the button to access the menu and the right is the button to access the search interface of a trip. The button at the bottom right makes it possible to refocus the map on the position of the user.

If the user is the driver for routes where the are passengers, alert message will be displayed when he connect. He had to confirm if the passenger was in his car or not. If he click on "Yes", the passenger could see the route in his "MyRides" tab. If he click on "No", the passenger is deleted in the database of the Passenger tab.

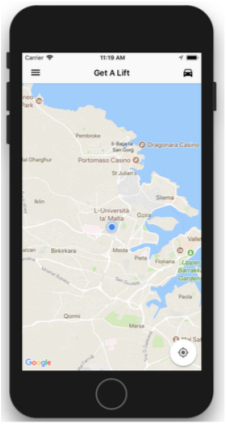


Figure 5.7: Main Page

Menu

We can access to different functionality from the menu :

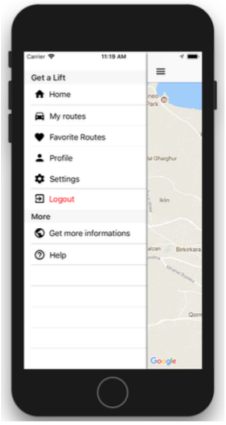


Figure 5.8: Menu

We can :

- go to the main page
- see information about his profile
- get more information about the app
- access a tutorial to discover how the application works
- access the application settings
- Sign out.
- see the section "My routes" which is the list of the routes that the authenticated user has created as a driver
- see the section "Favorite Routes" which is the list of trips that the user has saved as a passenger.

To access the route search interface, click on the button:

Search interface of a route

This interface groups together the search and the creation of a trip in order to highlight carpooling. Indeed, if a user wants to create a route as a driver he will in any case access the list of available routes according to his criteria. This will show this user that other people are making the same trip as him and that he is not obliged to create another trip. This makes it possible to promote carpooling and to simplify the use of the application.

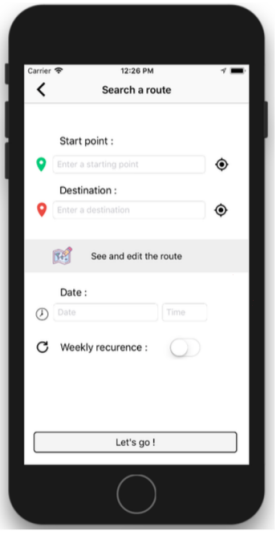


Figure 5.9: Search interface of a route

Here we specify its point of departure and its point of arrival. When you click on the text entry field a drop-down menu is displayed to make suggestions to the user. He can also click on the locate button to display the current position of the user in the text entry field.

Once this information is entered, the user can display and fine-tune his journey using a map. By clicking on the "See and edit the route" button.

Preview of the desired route

On this map you can directly edit the points on the map "by hand". The user can zoom, move the map to draw the path that suits him. Whenever the position of a point is changed, the path between the two points is updated. Once the route has been modified, the user can return to the search page of a trip and the input fields will be immediately modified to display the coordinates of the points he has previously modified on the map.

The user can now enter the date and time of his trip in the search interface of a route. He can also notify if his trip will be done every week or not. The purpose of this application is to connect the users of the daily road.

Once all fields are completed, he can click on the "Let's go" button to display the corresponding results.

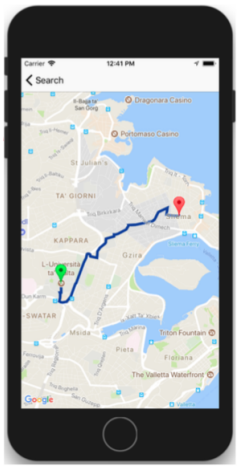


Figure 5.10: Preview of the desired route

List of available routes

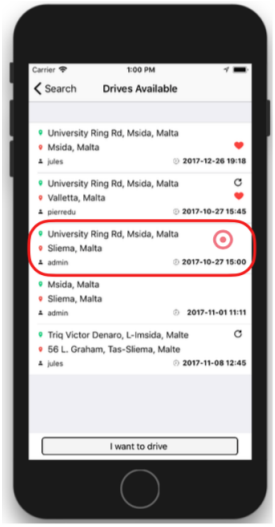


Figure 5.11: List of available route

We can observe several routes available with information about them (date, time, driver, recurrence). The red heart indicates that it is a road registered as a favorite road.

The button offers the possibility to create its own route.

If we click on the route, the next page is displayed with the route projection on a map and the different route information.

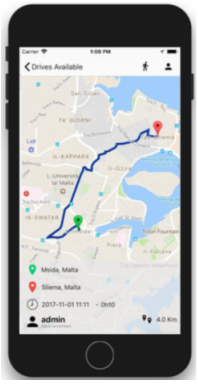


Figure 5.12: Overview of a route

Overview of a route

In this view we find all the information on the trip. We can see the road on a map and if we click on the button representing a man who walks, we will have the way to walk between our starting point and that of the path that will be displayed, and the same for the point of departure. Just press this button again to remove the path to walk.

The user can also access the driver's information to contact him or save the trip by clicking on his or her pseudonym or by clicking on the small contact icon at the top right of the screen.

Overview of a driver

Figure 5.13: Overview of a driver

The user can call him driver, send him an SMS or an email directly from the application by pressing the dedicated buttons. It can also save this trip to its preferred route list to keep it.

Interface to create a route

In the case where the user does not find corresponding paths to his trip, he can create a trip by returning to the list of available trips and clicking on the button "I want to drive".

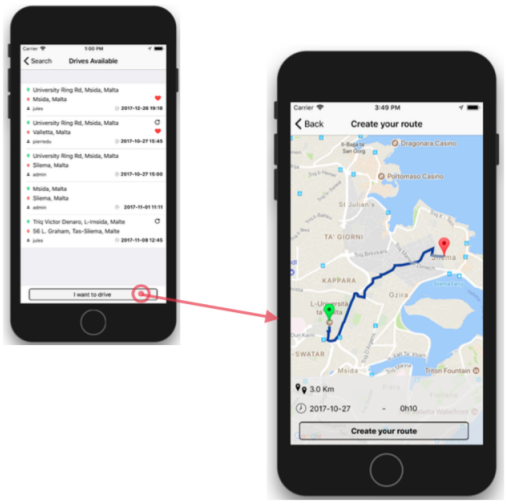


Figure 5.14: Interface to create a route

We get a summary page of the route that we want to create, if we want to change this it will be enough to go back and change what we want in the search interface / creation. The user presses the "Create your route" button to finalize the creation, a confirmation alert will be displayed on the screen and the user will be redirected to the main page of the application.

List of created routes

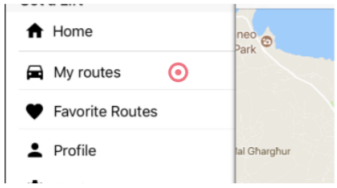


Figure 5.15: List of created route

The user to find this trip in the list of routes he has created, available in the application menu. From this list, the user can see in more detail the routes he has created and delete them.

Likewise with the list of his favorite journeys.

5.3 GetALift iOS Classes

5.3.1 Model View Controller (MVC)

MVC is an acronym that stands for Model View Controller. With the VMC, we divide our program into 3 parts:

- Model: that's what the application does. This is the logic, the brain of the application. He is responsible for the manipulation of the data. In the case of the GetALift application, this part deals with tasks such as saving data, saving, retrieving user data, searching for routes by search parameters, etc.

- Controller: It retrieves model information and posters in the view.
- View: that's what the user sees, it's the interface of the application (storyboard)

5.3.2 Controller

The code is directly detailed on each corresponding file but this part of the documentation generally describes the classes and their main functions.

In each UIViewController class :

- the ViewDidLoad function is the function that is called when the page is created.
- the ViewWillAppear is the function that is called even time that the page appears after it creation

HomePage

The home page when the user open the application.

Login

The first page that opens when the user launch the application.

Function(s) :

- Authentification : This function creates a new user object in the case where the fields "username" and "password" have been correctly completed and the authentication on the backend has worked.

FirstView

The first view of the app when the user is authenticated.

Function(s) :

- createUIAlertController : Create an object of type alert.
- showAlert : Display the alert and the 2 actions « Yes » and « No »
- confirmPassenger : Execute the changeInTheCarColumn request which change the value of the « inTheCar » column in the dataBase to confirm the presence of the passenger in the car
- notConfirmPassenger : Execute the deletePassenger request which delete the user from the database in the case where the driver don't confirm the presence of the passenger in his car.

CreateAccount

View for that the user create an account on the dataBase.

Function(s) :

- createAccount : This function creates a new user object and a new user in the database in the case where the different fields have been correctly completed and the creation of a new user on the database has worked.

SearchRoute

The view to research or create a route.

Help

Help page.

RouteList

The page which display the list of route after a research.

RouteView

Class to display the information of an existing route.

Function(s) :

- addToRide : This function execute first the ddRide request. If the POST request works (if « success == true »), a route is associated to a new ride on the database. A passenger is created for the corresponding ride in the database. In the case, where « success == false » which means that an already created ride exist for the current route, there is only a new passenger for the corresponding ride who is created on the database by executing the « addPassengerExistingRide » request.

DriverView

Class to display information of the driver of the selected route.

Function(s) :

- addToFavoriteRoute : This function execute the addFavoriteRoute request to allow the user to save a route in his Favorite Route tab. If the route is already in your favorite route tab, an error message appears. Otherwise, the route is added to your Favorite Route.
- message, mail, phone : Several function in this class allow the user to call, send a message or a mail to the driver

CreateYourRoute

Class which allow creation of the route on the database and allow the display of the route

Menu

UserProfile

Class to display the information of the user

myRoutes

Class to display the different routes the connected user created

routeSaved

Class to display the different routes the connected user add to his favorite routes

SeeCurrentRoute

Class to show the route being edited on the research interface

routeViewForTheMenu

Class allowing to display informations of an existing route

PageViewController

GetMoreInformation

Get Morte information page

Settings

Settings Page

Rides

Class to display the different Rides (route that user made and where the driver confirmed his presence.

Comments

Class to display the different comments relative to a driver.

Rating

Class which allow to the user to rate and comment the driver for a ride he made. If it is the first time, he rate the driver, it use a POST request and if he modify his rate, it use a PUT request.

Function(s) :

- rateFunction : This function execute first the postRating request. If the POST request works, the rate is creating in the database and a notification banner inform the user that his rate have been created. If the POST request didn't work, it means that a rate already exist for this route by the user in the database. The function then execute the putRate function which modify the existing rate if the user click on « Yes » on the alert message which appear or not if he click on « No »

EditProfile

Class which allow the user to modify his personal information on the database.

Function(s) :

- editInformations : This function allow the user to modify his information. The user must to enter his actual password if he want change an information of his profile. If he want to change his password, he must to complete the two fields in the same way or an error banner appears. If his information are correctly modified on the database, a notification banner appears.

5.3.3 Model

User

This class define the object User

Route

This class define the object Route

DateTime

This class defines the object date

Passenger

This class defines the object Passenger

Comment

This class defines the object Comment

5.3.4 Tasks

MapTasks

Regroup all the requests concerning the map.

UserTasks

Regroup all the requests concerning users.

Function(s) :

- user: Perform GET request which create a user object when it is called.
- editUser: Perform PUT request which allow to modify an existing user on the database regarding his id
- authentification: Perform POST request which allow to confirm or not the authentication of a user.

RouteTasks

Regroup all the requests concerning routes.

Function(s) :

- route POST: It is the POST request which is perform when a user search a route. It create one or several object of type Route in an array to display them after.
- route (date) GET: Perform GET request which throw back a route regarding it date.
- route (driverId) GET: Perform GET request which throw back a route regarding it driver.
- deleteRoute: Perform DELETE request which delete a route regarding it id.

DateTasks

Regroup all the requests concerning the date.

Function(s) :

- date(routeId): Perform GET request which create a date object regarding the route id when it is called.

FavoriteRouteTasks

Regroup all the requests concerning favorite routes.

Function(s) :

- favoriteRoute: Perform GET request which create object of type Route regarding the user id to display the different route saved by the user.
- deleteFavoriteRoute: Perform DELETE request to delete a favorite route regarding the route id.
- addFavoriteRoute: Perform POST request which save a route regarding a user id in the favoriteRoute tab on the database

PassengerTasks

Regroup all the requests concerning passengers.

Function(s) :

- passengerNames: Perform GET request that returns a table of all passengers in relation to a driver
- deletePassenger: Perform DELETE request which allows to delete a passenger according to the id
- changeInTheCarColumn: Perform PUT request that allows to pass inTheCar to the value 1 if the driver confirms the presence of the passenger
- addpass: Perform POST request that adds a passenger in the Database regarding the ride Id
- addPassengerExistingRide: Perform POST request that adds a passenger in the database regarding his id and the route id when a ride already exist regarding the route.

RatingTasks

Regroup all the requests concerning rates.

Function(s) :

- getRating: Perform GET request which allow to recover the average of the rate o a driver.
- postRating: Perform POST request which allow to create a rate on the database regarding a driver and a route
- putRate: Perform PUT request which allow to modify on the database an already existing rate

RideTasks

Regroup all the requests concerning rides.

Function(s) :

- rides: Perform GET request hat returns a table of all rides in relation to a passenger.
- addRide: Perform POST function which create a ride regarding a Route if a ride doesn't already exist for the route

CommentsTasks

Regroup all the requests concerning the display of the comments

Function(s) :

- commentaries: Perform GET request that returns all the comments regarding a driver id.

5.3.5 Others

SearchTextField

Class that manages the operation of the different search bars

CalculationForMapDisplay

Class that manage the the display of the map on the screen regarding the zoom.

5.3.6 Custom Item

MyCustomCell

Class that manage customCell cells in UITableView

BoutonArrondi

Class that manage BoutonArrondi button.

SelectedButton

Class that manage the selected button

BoutonArrondi2

Class that manage BoutonArrondi2 button.

RidesTableViewCell

Class that manage RidesTableViewCell cell in the UITableView table on the Rides page.

CommentaryTableViewCell

Class that manage CommentaryTableViewCell cell in the UITableView table on the Comment page.