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.
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.
This chapter is divided in 4 majors subparts :

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

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.

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
}
```

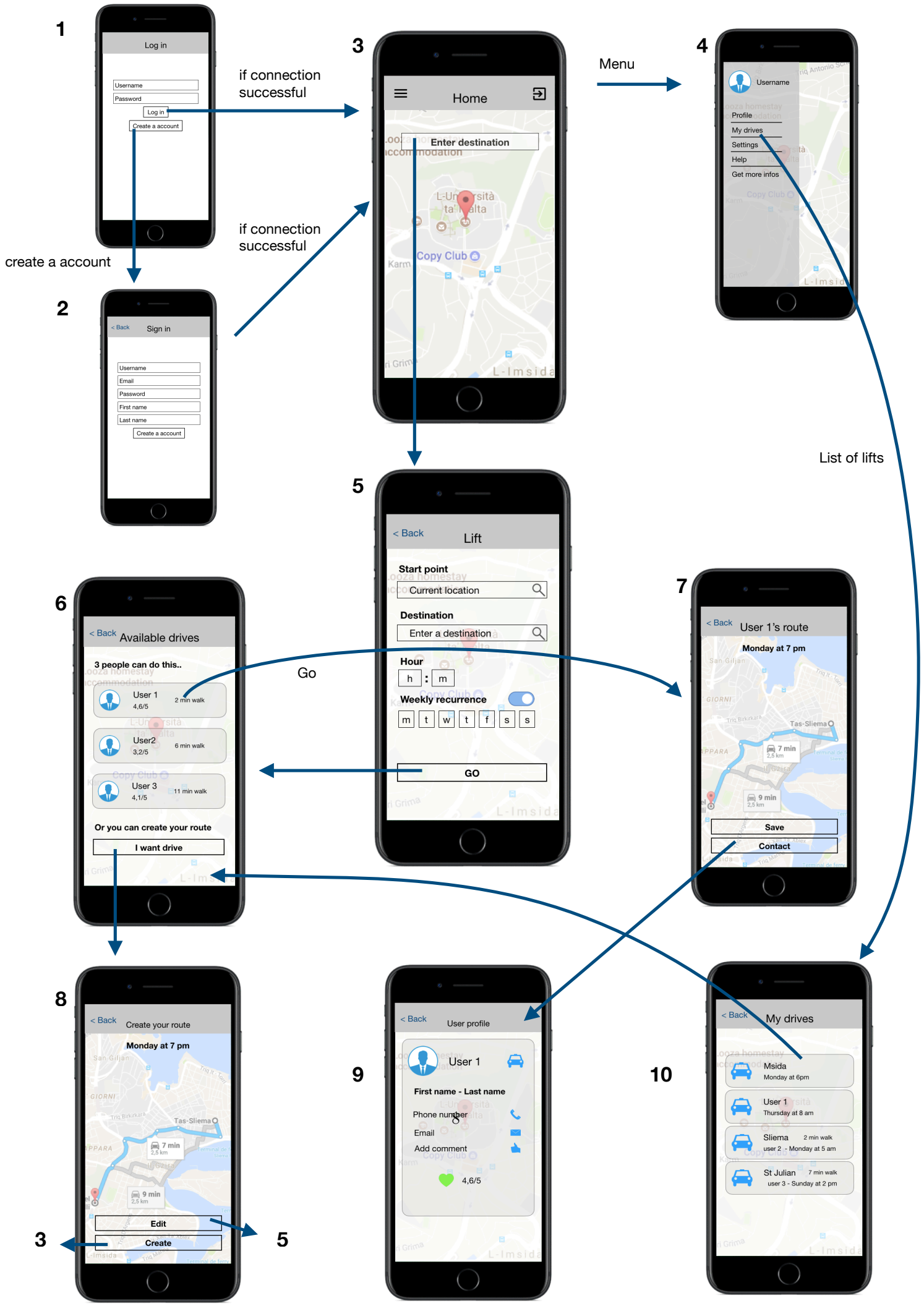
A routePoint is defined by :

```
{
        "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.


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.
- 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.
- 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.
- POST "/passenger"
  - This route can create a passenger in the database.
- PUT "/passenger/:passid"
  - This route update the information about the chosen passenger.
- DELTE "/passenger/:passid"
  - This route deletes the chosen 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.
- POST "/ratings"
  - This route create a rate in the database.
-
  -
- DELETE "/ratings/:rateid"
  - This route update the information about the chosen rating.

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

TODO

Chapter 5

iOS application

TODO