



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

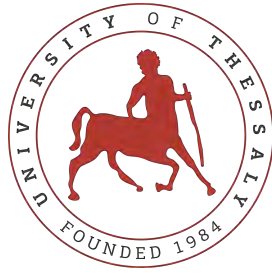
Full stack implementation of a car pooling web and mobile application

Diploma Thesis

NIKOLAOS PAPPAS

Supervisor: Georgios Stamoulis

Volos 2020



UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Full stack implementation of a car pooling web and mobile application

Diploma Thesis

NIKOLAOS PAPPAS

Supervisor: Georgios Stamoulis

Volos 2020



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη εφαρμογής διαμοιρασμού οχημάτων

Διπλωματική Εργασία

ΝΙΚΟΛΑΟΣ ΠΑΠΠΑΣ

Επιβλέπων/πouσα: Σταμούλης Γεώργιος

Βόλος 2020

Approved by the Examination Committee:

Supervisor **Georgios Stamoulis**

Professor, Department of Electrical and Computer Engineering,
University of Thessaly

Member **George Thanos**

Laboratory Teaching Staff, Department of Electrical and Com-
puter Engineering, University of Thessaly

Member **Hariklia Tsalapata**

Laboratory Teaching Staff, Department of Electrical and Com-
puter Engineering, University of Thessaly

Date of approval: 20-9-2020

Acknowledgements

Firstly, I would like to thank Laboratory Teaching Staff George Thanos for his advice, guidance and assistance throughout the whole project. Furthermore, i would like to thank my family for their support and patience. Moreover, I would like to thank my friends Thanos, Dimitris, Kostas, Kostas and Apostolis for their help and advice when needed! Lastly I would like to thank Ioanna for her tolerance, patience and persistence.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

NIKOLAOS PAPPAS

15-9-2020

Abstract

In recent years, although technology has diminished travel time, travel costs have increased due to the continuously increasing fuel prices and the privatization of public highways that result in the creation of more and more toll stations. Carpooling is becoming a common way to travel because it reduces each traveller's travel costs and is also a more environmental friendly and sustainable way to travel. It reduces air pollution, carbon emissions, traffic congestion on the roads and the need for parking spaces. This diploma thesis deals with the implementation of a web and mobile application that serves the purpose of carpooling. The development involves implementing firstly the front-end of the application, the web and the mobile user interface and secondly the back-end that serves all the requests a user makes from the front-end. The web frameworks used are Django and ReactJS with the programming languages Python and JavaScript respectively. The aim is to learn these two web frameworks from development mode to production state and generally to develop existing programming knowledge through this application development. More specifically on the application, it enables a user to browse rides offered, to login or register, join the rides, or add new ones.

Keywords

carpooling, web application, mobile application, Django, Django REST framework, ReactJS, React Native

Περίληψη

Τα τελευταία χρόνια, αν και η τεχνολογία έχει μειώσει τον χρόνο ταξιδιού, το κόστος του έχει αυξηθεί λόγω των συνεχώς αυξανόμενων τιμών των καυσίμων και της ιδιωτικοποίησης των δημόσιων αυτοκινητοδρόμων που έχουν ως αποτέλεσμα τη δημιουργία όλο και περισσότερων σταθμών διοδίων. Το Carpooling γίνεται ένας συνηθισμένος τρόπος για να ταξιδέψετε επειδή μειώνει το κόστος ταξιδιού κάθε ταξιδιώτη και είναι επίσης ένας πιο φιλικός προς το περιβάλλον και βιώσιμος τρόπος ταξιδιού καθώς η κοινή χρήση αυτοκινήτων για τα ταξίδια μειώνει την ατμοσφαιρική ρύπανση, τις εκπομπές διοξειδίου του άνθρακα, την κυκλοφοριακή συμφόρηση στους δρόμους και την ανάγκη για θέσεις στάθμευσης. Αυτή η διπλωματική εργασία ασχολείται με την υλοποίηση μιας εφαρμογής διαδικτύου και κινητών τηλεφώνων που εξυπηρετεί τον σκοπό του carpooling. Η ανάπτυξη της εφαρμογής περιλαμβάνει την υλοποίηση, πρώτον, του γραφικού περιβάλλοντος της εφαρμογής, του διαδικτύου και της διεπαφής χρήστη για κινητά και, δεύτερον, την υλοποίηση του σέρβερ που εξυπηρετεί όλα τα αιτήματα που κάνει ένας χρήστης από το γραφικό περιβάλλον. Οι τεχνολογίες ανάπτυξης διαδικτυακών εφαρμογών που χρησιμοποιούνται είναι Django και ReactJS με τις γλώσσες προγραμματισμού Python και JavaScript αντίστοιχα. Ο στόχος είναι να μάθουμε αυτές τις δύο τεχνολογίες ανάπτυξης διαδικτυακών εφαρμογών διεξοδικά και σε βάθος, από το αρχικό στάδιο ανάπτυξής τους έως την κατάσταση παραγωγής και γενικά να αναπτύξουμε τις υπάρχουσες γνώσεις προγραμματισμού εφαρμογών. Πιο συγκεκριμένα η εφαρμογή, επιτρέπει στον χρήστη να περιηγηθεί στις προσφερόμενες διαδρομές, να συνδεθεί ή να εγγραφεί, να συμμετάσχει στις διαδρομές ή να προσθέσει νέες.

Λέξεις Κλειδιά

carpooling, web application, mobile application, Django, Django REST framework, ReactJS, React Native

Table of contents

Acknowledgements	ix
Abstract	xi
Περίληψη	xiii
Table of contents	xv
List of figures	xix
List of Abbreviations	xxi
1 Introduction	1
1.1 Idea Description	1
1.2 Main Objective	2
1.3 Thesis Structure	2
2 Knowledge base	5
2.1 Software Stack	5
2.2 Back-end	5
2.3 Front-End	5
2.4 Full Stack Development	6
2.5 Client - server model	6
2.6 API	7
2.7 Representational State Transfer	7
2.7.1 Separation of client and server	8
2.7.2 Statelessness	8

2.7.3	Client and Server communication	8
2.8	MVC	9
2.9	Python Virtual Environment	10
3	Application Analysis	11
3.1	Functional requirements	11
3.1.1	Authorization	11
3.1.2	Rides	12
3.1.3	Cars	12
3.1.4	Notifications	12
3.2	Non-functional requirements	12
3.2.1	Back-end API	13
3.2.2	Front-end	13
3.2.3	Models Schema	13
3.2.4	Use cases	15
4	Application architecture	17
4.1	Introduction	17
4.1.1	Product perspective	17
4.1.2	Operating Environment	17
4.1.3	Constraints	17
4.1.4	Assumptions and considerations	18
4.2	System structure	18
4.2.1	General System overview	18
4.2.2	Authentication	21
5	Application Design	23
5.1	Back-End	23
5.1.1	Chosen Technologies	23
5.1.2	Used Libraries and Add-Ons	24
5.2	Front-End	25
5.2.1	Chosen Technologies	25
5.2.2	Used Libraries and Add-Ons	27
5.3	Used tools	29

5.3.1	GIT Version Control	29
5.3.2	PyCharm IDE	29
5.3.3	WebStorm IDE	29
5.3.4	PostMan	29
5.3.5	React Developer Tools	30
5.3.6	Redux DevTools	30
6	Back-End Development	31
6.1	Django Fundamentals	31
6.1.1	Django General Structure	31
6.1.2	Django App Structure	33
6.1.3	Django request cycle	33
6.2	Setting up prerequisites	35
6.2.1	Setting up MySQL and Redis	35
6.2.2	Setting up Django	35
6.3	Implementation	36
6.3.1	Django Apps	36
6.3.2	Django Models	37
6.3.3	Django Views	38
6.3.4	Django URLs	40
6.3.5	Django Serializers	40
6.3.6	Custom Middleware	41
6.3.7	Django webSocket Routing	42
6.3.8	Django Settings	44
6.3.9	Authorization and authentication	45
6.4	Deployment	46
6.4.1	System general configuration	46
6.4.2	NGINX configuration	47
6.4.3	Server Hardening	50
7	Front-End Development	53
7.1	React Fundamentals	53
7.1.1	JSX	53

7.1.2	State	54
7.1.3	Virtual Document Object Model	55
7.1.4	React Lifecycle Methods	56
7.1.5	Redux	57
7.2	Web Application Development	59
7.2.1	Setup	59
7.2.2	Application Structure	60
7.2.3	Implementation	62
7.2.4	Deployment	68
7.3	Mobile Application Development	69
7.3.1	Mobile Development approach	69
7.3.2	React Native	70
7.3.3	Expo	71
7.4	Mobile application Implementation	72
7.4.1	Setup	72
7.4.2	Mobile Application Structure	72
7.4.3	App.js	73
7.4.4	MainNavigation.js	75
7.5	Deployment	81
8	Application Preview	83
8.1	Back-End	83
8.1.1	Landing Page	83
8.1.2	Django Admin Page	83
8.1.3	REST API endpoints	86
8.2	Front-End	87
8.2.1	Web Application	87
8.2.2	Mobile Application	97
9	Production sites and code	111
10	Future development	113

List of figures

2.1	Client Server Model [6]	6
2.2	N-tier [5]	7
2.3	MVC architecture [61]	10
3.1	Application models	14
3.2	Use Cases	15
4.1	System Structure	20
6.1	Django Structure general overview [13]	32
6.2	Django Life Cycle [11]	34
7.1	React example [41]	54
7.2	State example [62]	55
7.3	Virtual DOM and Browser DOM Comparison [68]	56
7.4	React lifecycle methods [47]	57
7.5	Redux data flow [1]	58
7.6	Default React Structure [46]	60
7.7	My React Structure	61
7.8	Heroku carsharing1312 App dashboard	68
7.9	Mobile Application Structure	73
8.1	Back-End Landing Page [62]	83
8.2	Django admin login page	84
8.3	Django admin interface	85
8.4	Home Page	88
8.5	Rides	88

8.6	Ride	89
8.7	FAQ	89
8.8	Terms and Conditions	90
8.9	Privacy Policy	90
8.10	Login form	91
8.11	Register form	92
8.12	After Login/Registration	93
8.13	Add Ride	94
8.14	Add Ride	95
8.15	Edit Ride	95
8.16	Settings	95
8.17	My rides	96
8.18	Requests	96
8.19	Settings	96
8.20	Home Screen	97
8.21	Rides Screen	98
8.22	Ride Filters	99
8.23	FAQ Screen	100
8.24	Login Screen	101
8.25	Sign up Screen	102
8.26	Join Ride	103
8.27	My profile	104
8.28	Notifications	104
8.29	Cars	105
8.30	Add Car	106
8.31	Delete Car	106
8.32	Edit Car	107
8.33	Cars	108
8.34	Edit Profile	109
8.35	My Rides	109
8.36	user's Requests	110
8.37	Add Ride	110

List of Abbreviations

The next list describes several symbols that will be later used within the body of the document.

API Application Programming Interface

ASGI Asynchronous Server Gateway Interface

CA Certificate Authority

CORS Cross-Origin Resource Sharing

CRUD Create, Read, Update, Delete

CSRF Cross-Site Request Forgery

CSS Cascading Style Sheets

DBMS Database Management System

DOM Document Object Model

DRF Django Rest Framework

DVCS Distributed Version Control System

EFF Electronic Frontier Foundation

FAQ Frequently asked Questions

HTML Hyper Text Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IDE Integrated development environment

IoT Internet of Things

IP Internet Protocol

JS JavaScript

JSON JavaScript Object Notation

JWT JSON Web Token

MVC Model, View, Controller

OS Operating System

PK Primary Key

QR Quick Response

RDBMS Relational Database Management System

REDIS Remote Dictionary Server

REST Representational state transfer

SQL Structured Query Language

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UFW Uncomplicated Firewall

URL Uniform Resource Locator

VCS Version Control System

WS Web Socket

WSGI Web Server Gateway Interface

Chapter 1

Introduction

1.1 Idea Description

Transportation plays a key role in society's structure and development. It refers to the movement of goods and people from place to place and the networks by which such movement is achieved. Transportation enables trade and cultural dialog between people, which are essential for the development of civilizations. Relocation of travelers and cargo are the most common uses of transportation.

Passenger transportation is divided into public or private. The first refers to scheduled services on fixed routes defined by private or public companies, meanwhile the second refers to privately owned vehicles that provide travel services at the riders' desire.

As big is the role of transportation to society, that big are also its costs. Technology and mankind progress have diminished travel time between places however travels costs have not seen a similar drop down. Although vehicles became more efficient in terms of consumption, transportation infrastructure costs have risen and so has the petrol and gas prices. Most highways have been privatized and have introduced high priced tolls to profit from their use by vehicles. The need of a cost-efficient and sustainable way of travel has aroused.

Carpooling allows travelers to share a ride to a common destination. Carpooling has several social and environmental impacts. From the reduce of energy consumption and gas emissions to the congestion mitigation and the reduced parking demand. Furthermore, people who use carpooling as a way to travel share their travel costs so their individual travel cost is lower. It is an environmentally friendly and sustainable way to travel. It depicts how a collaborative way to travel can be beneficial for individuals and also the whole society in extend.

The idea of implementing a carpooling platform draws its origins from the most know carpooling platform, BlaBlaCar. It has a trusted community of 90 million drivers and passengers and it is used in 22 countries worldwide. Greece is not one of these countries and due to the fact that the need of a platform that provides a sustainable way to travel in Greece exists, an implementation of such platform would help Greek people and Greek society in many aspects.

1.2 Main Objective

This thesis focuses on implementing a web and mobile application that serves as a carpooling platform in which users will be able to offer rides. More specifically, this platform will connect people looking to travel long distances with drivers heading the same way, so they can travel together and share the cost. the main technical goals are to learn and extend the knowledge of two web frameworks, their programming languages and their tools, the client-server architecture, the creation of *RESTful APIs* and user-friendly interfaces and lastly the configuration of web servers that host the two web frameworks.

1.3 Thesis Structure

In Chapter 1 introduction is developed, in which the idea and purpose of the this diploma thesis is analyzed.

In Chapter 2 the technological and theoretical background that one must have to understand the structure and development of a modern internet application is analyzed.

In Chapter 3 the application analysis takes place, which includes analyzing the idea, the details and the scope of the application and defining the application requirements.

In Chapter 5 the technological overview of the application is analyzed including the programming language used, the back-end and front-end frames and the database.

In Chapter 6 the back-end development from development to production state is described.

In Chapter 7 the front-end development is described, including the development of both web and mobile applications.

In Chapter 8 the functions that are available to the user are shown as well as the user

interface and the services provided.

In Chapter 9 the production sites and code repositories are listed.

In Chapter 10 some future extensions are reported.

Chapter 2

Knowledge base

This chapter provides elucidation for all unfamiliar concepts and technologies used in this master thesis.

2.1 Software Stack

A software stack is a collection of independent components/functions that work together to deliver specified services to the user. The components, which may be operating systems, architectural layers, protocols, run-time environments, databases and function calls, are in a hierarchical order. Most common web stacks are: **LAMP** (*Linux, Apache, MySQL, PHP*) and **MEAN** (*MongoDB, Express, Angular and Node*).

2.2 Back-end

It is the server side of an application or website that focuses on how the application or website functions above the hood. Back-end implements all the logic and functions of a website and is responsible for managing the database. The most common back-end frameworks are Express(*JavaScript*), Django(*Python*), Rails(*Ruby*), Laravel(*PHP*) and Spring(*Java*) [65].

2.3 Front-End

It refers to the visible part of website or web application from which the user interacts with the application or website. The front-end portion is mainly built by using **HTML, CSS**

and in some cases also JavaScript. The most common frameworks and libraries are Bootstrap, React, AngularJS and VueJS.

2.4 Full Stack Development

It refers to the development of both front end(client side) and back end(server side) portions of web application.

2.5 Client - server model

Client-server model is an application architecture that partitions tasks and workload between a server that provides resources and the client that requests that resources. Clients and servers interact through network connections using predefined protocols(*HTTP,SMTP*, etc). The communication is unidirectional: The client requests a resource/service from the server, and the latter after processing the request, returns a response. There could be multiple client components issuing requests to a server that is passively waiting for them. The client-server model is a centralized networking model, unlike P2P which is decentralized. [6]

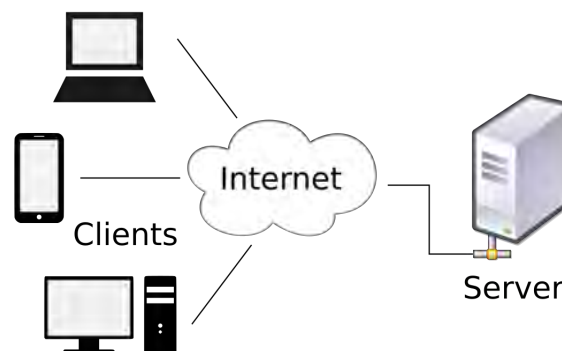


Figure 2.1: Client Server Model [6]

With the use of this architecture, the client side(Front-end) is free from interacting with the database. The server side handles business logic and database queries. The traditional client-server architecture involves two levels, a client level and a server level. But nowadays the common design of client-server systems uses three or more levels called tiers. The N-tier architecture is an industry-proven software architecture model. It is suitable to support enterprise level client-server applications by providing solutions to scalability, security, fault

tolerance, reusability, and maintainability. It helps developers to create flexible and reusable applications. These tiers are:

- **The presentation tier:** the front-end which deals with the interaction with the user.
- **The application tier:** The server, or the back-end that processes the requests of all clients.
- **The database tier:** A resource manager that stores data.

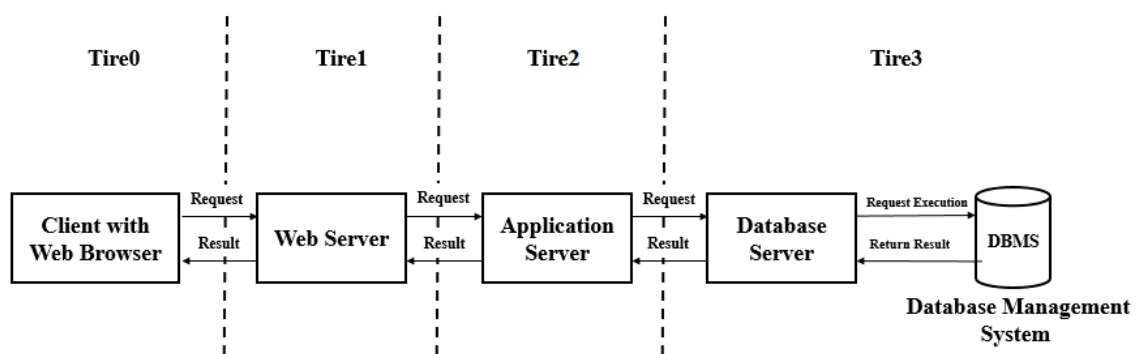


Figure 2.2: N-tier [5]

2.6 API

An Application Programming Interface is a set of functions and protocols for building an application software. *APIs* provide communication between services without the need of knowing how these services are implemented. *APIs* allow access to the application's resources without exposing how these resources are implemented, while maintaining security and control.

2.7 Representational State Transfer

REST, or Representational State Transfer, is an architecture for providing standards between computer systems on the internet, in order for the communication between them to be easier and guided. REST-compliant systems, are called RESTful systems and are characterized by how they are stateless and separate the concerns and aspects of client and server.

2.7.1 Separation of client and server

In the REST architecture, the implementations of both the client and the server are done independently. This means that the codes on either the client or the server side can be changed at any time. By using a REST, different clients may request resources from the same REST endpoints and receive the same responses.

2.7.2 Statelessness

Systems that follow the REST paradigm are stateless. This means that both the server and the client doesn't know about the state of the other. These constraints help RESTful applications achieve quick performance and scalability, as their components can be managed, updated, and reused without affecting the system as a whole.

2.7.3 Client and Server communication

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests.

The 4 most common HTTP request methods are[32]:

HTTP request	Meaning
GET	retrieve a specific resource
POST	create a new resource
PUT/PATCH	update a specific resource
DELETE	remove a specific resource

The most common HTTP response/status codes[33]:

Status Code	Meaning
200 (OK)	Successful HTTP request
201 (CREATED)	Successful HTTP request that resulted in a resource creation
204 (NO CONTENT)	Successful HTTP request where nothing is being returned
400 (BAD REQUEST)	Bad request syntax, excessive size or other client error
401 (UNAUTHORIZED)	Lacks of authentication credentials
403 (FORBIDDEN)	The client does not have permission to access the resource requested
404 (404)	The resource cannot be found on the server
500 (INTERNAL SERVER ERROR)	The generic response for a server error

2.8 MVC

MVC is short for Model, View, and Controller. MVC is a popular way of organizing your code. [21]

- **Model:** It is our data representation. It serves as an intermediate between the data stored in the database and the functions available. It allows to interact with your data without having to write complex database queries.
- **View:** It represents what a user see on its browser/mobile application.
- **Controller:** It provides the logic to either handle presentation flow in the view or update the model's data. It implements the logic to alter the model and in extend the database, or alter the visual contents the end user sees.

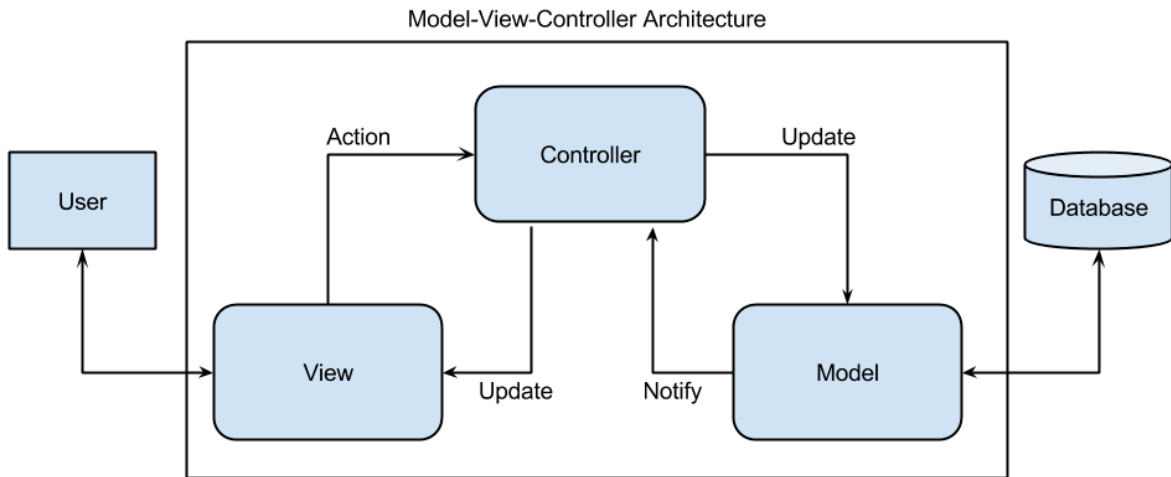


Figure 2.3: MVC architecture [61]

2.9 Python Virtual Environment

Since this application uses a set of dependencies that don't come as a part of the Python standard library they must be installed for all instances of the application, namely the development and testing on the local machines of developers and the production server. Python Virtual Environment was used for these purposes.

A **Virtual Environment** is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the “Project X depends on version 1.x but, Project Y needs 4.x” dilemma, and keeps global site-packages directory clean and manageable. [37]

A list of all the dependencies that are installed in the virtual environment can be found in the requirements.txt file in the project's root directory. The guide for installing and configuring the virtual environment is described in chapter 6.

Chapter 3

Application Analysis

This chapter will focus on the analysis of the project as a part of a software development that connects customer's requirements to the system and its subsequent design and development. Analysis of software project is intended to define the detailed description of the application, break it down into requirements to the system, their systematization, detection of dependencies, and documentation.

3.1 Functional requirements

Functional requirements specify the behaviors the product will exhibit under specific conditions. They describe what the developers must implement to enable users their requirements.

3.1.1 Authorization

- F1. Sign up / Login. User shall be able to sign up, providing username, password, email, avatar, phone etc, and will also be able to log in with those credentials.
- F2. Sign up / Log in via Facebook. User shall be able to sign up to the application with his Facebook account. The application shall load user's data such as name, surname, email, date of birth, etc.
- F3. Logout: Authorized user shall be able to log out. In this case, he/she shall also stop receiving any notifications from the application.
- F4. View profile: Only authenticated users have permission to see other users profiles.

3.1.2 Rides

- F5. Browse rides: All users, including not authenticated ones will be able to browse the rides offered.
- F6. View ride details: All users, including not authenticated ones will be able to view the details(*origin, destination, etc.*) of the ride offered.
- F7. Create / Edit / Delete Ride: Authenticated users will be able to create new rides or edit or delete theirs. The creation of a ride, requires the ownership of at least one car to host the ride.
- F8. Join ride: Authenticated users will be able to join a ride offered by another authenticated user.
- F9. Accept / Deny ride request: Users that offer a ride, can accept or deny others requests to join their ride.
- F10. Unjoin from ride: Users that already joined a ride, can unjoin.

3.1.3 Cars

- F11. Add / Edit / Delete car: Authenticated users will be able to add/edit/remove a car they own.

3.1.4 Notifications

- F12. View user notifications. User shall receive information about their requests to join rides.
- F13. Mark notifications as *read*. User shall be able to mark the notifications received as *read*.

3.2 Non-functional requirements

Non-functional requirements are requirements that describe not what the software will do, but how the software will do it and in extend what how will the developer implement

them.

3.2.1 Back-end API

NF1. RESTful. Back-end API shall follow architectural constraints of REST architectural style.

NF2. HTTPS. The server shall communicate with the client via *HTTPS*.

NF3. MySQL database. MySQL shall be used as the primary *DBMS*.

NF4. Server configuration and deployment. The virtual Machine that the back-end server is running on, must be properly configured and deployed in order to maintain the security, the integrity and the load balance of the application.

3.2.2 Front-end

NF5. User friendly. The user interfaces (both the web and mobile clients) must have simple, easy-to-understand and friendly components.

NF6. Performance optimized code. Every component, function and service implemented must be performance optimized, as not to consume many resources.

NF7. HTTPS. The front-end shall communicate with the users via *HTTPS*.

3.2.3 Models Schema

After analyzing the functional requirements we define three (3) models that have self-explanatory fields that help describe its model instance.

- User model, with the fields: username, password, email, avatar, date of birth, gender, country, phone number, has viber, has whatsapp.
- Ride model, with the following fields: origin, destination, date, time, vacant seats, car, create
- Car model, with the following fields: plate, brand, model, year, color.

To be more specific about the model relations between each other, the user and car models are connected via one to many relationship. The user and ride models are connected via many to many relationship. The ride and car models are connected via one to one relationship. The ride request and ride models are connected via many to one relationship. Finally, the ride request and the user models are connected via one to one relationship. Explanatory, each user can create many rides, with a car linked to each ride. If no car exists, then the user cannot create a ride. When a user joins a ride, a ride request model instance is created to link the requested user and the offered ride.

For simplification of understanding of the primary model classes and their behavior, it was decided to define so-called Domain model. The Domain model is the visual representation of conceptual classes. Domain model is visualization of things in real-world, not of software components such as C++ or Python class.

To better describe the models and their relations, the following schema/domain model was produced:

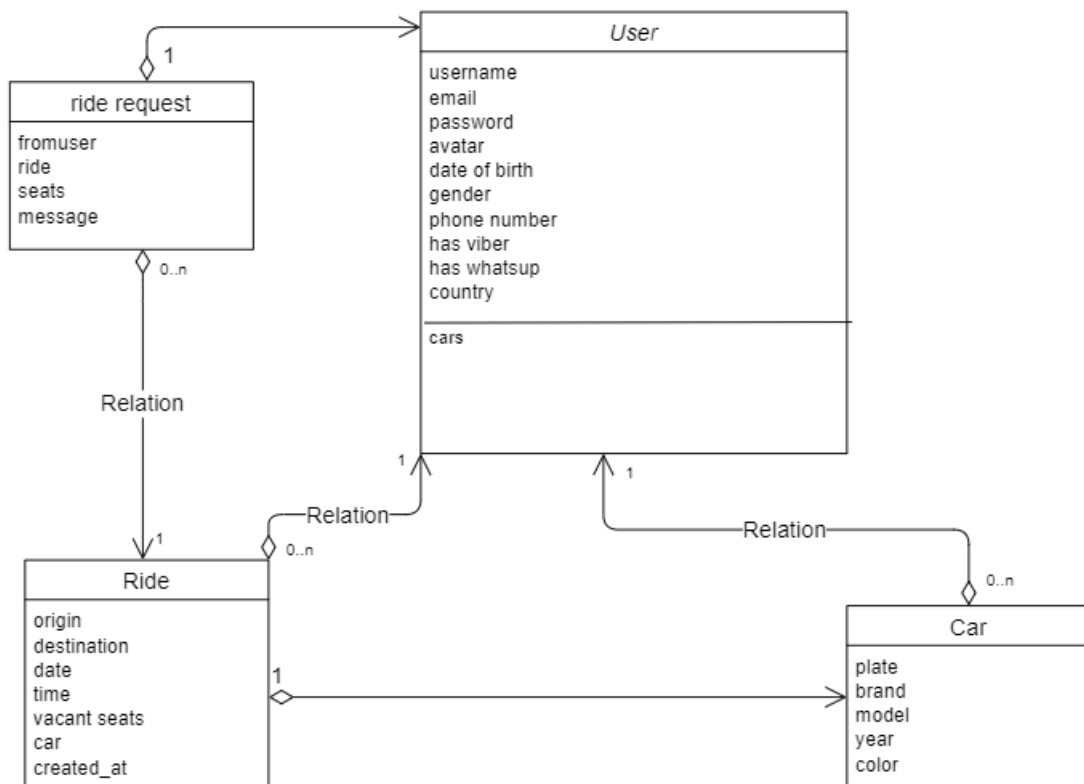


Figure 3.1: Application models

3.2.4 Use cases

Use cases were defined after analyzing both functional and non-functional requirements. Use cases documentation serves for better understandings of functionality required from the system.

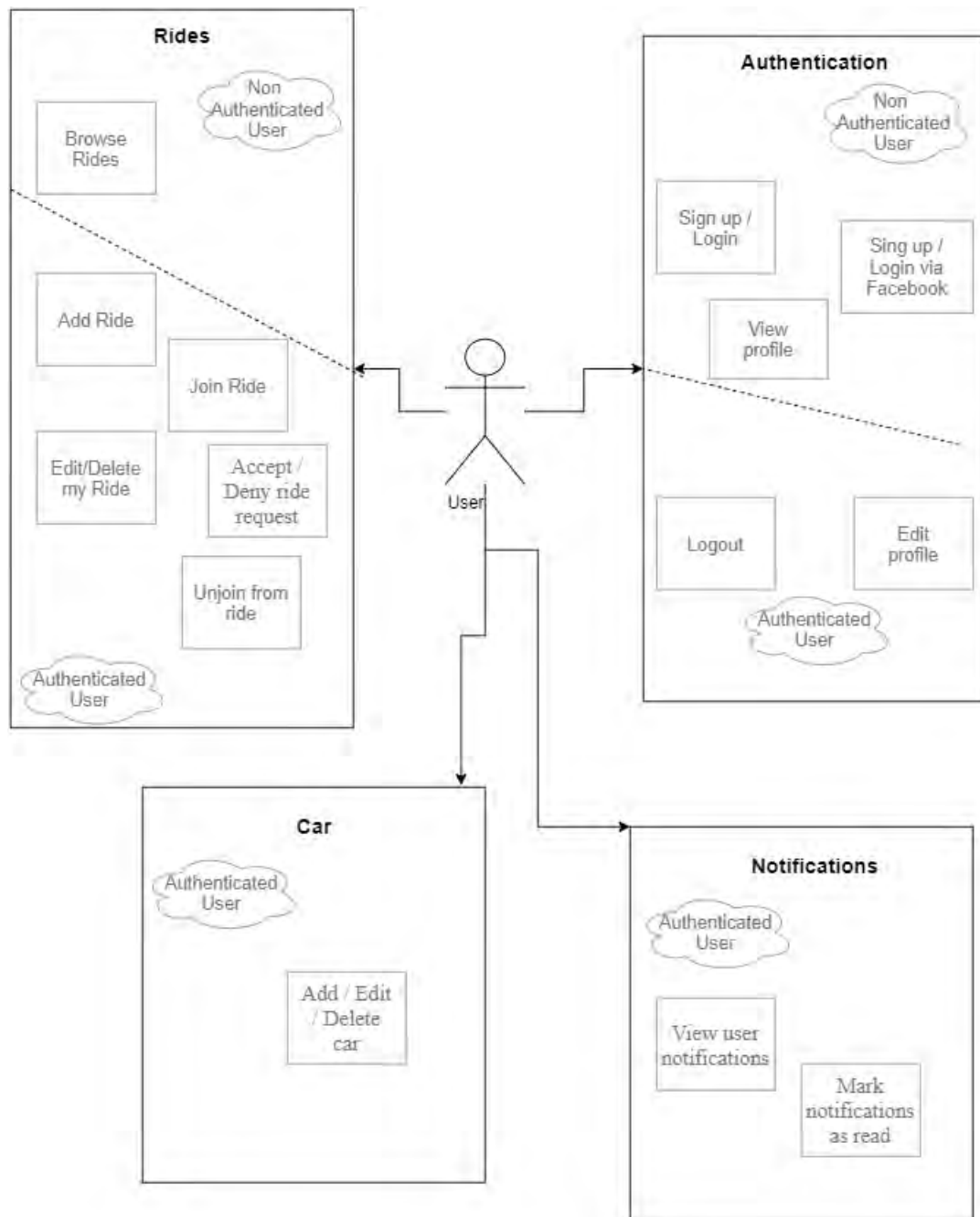


Figure 3.2: Use Cases

Chapter 4

Application architecture

This section provides an overall description of the project functions and requirements.

4.1 Introduction

4.1.1 Product perspective

The back-end will serve as a *RESTful API* that the web and mobile clients will consume. As stated before, by following this development path the complexity of back-end will be hidden from the user interface. It handles all logic computations and queries the database. On the other hand, the front-end handles all user actions and fetches resources from the back-end.

4.1.2 Operating Environment

The platform will be written in Python and JavaScript and therefore could be run on both Windows or Unix systems. It is also recommended to host the back-end service behind a web server like Apache or Nginx for load-balancing. Nginx has inbuilt wSGI support so it is the web server chosen. Although the platform could have any *RDMS* as database, MySQL is preferred. .

4.1.3 Constraints

This platform requires an active internet connection due to the fact that it is an online service. It enables the communication between users.

4.1.4 Assumptions and considerations

Although basic security measures and permissions checks will be implemented in order to secure both client and server from security threats, the initial version of this project will not be extremely focused on security. *CSRF* attack counter measures will be implemented and user input preprocessing will take place. The back-end server will be implemented using most known security measures and patterns. All data transmitted will be encrypted using *SSL/TLS*. Data storage on the back-end server and its database will not be encrypted, except for passwords storage. Front-end storage will use *HTML5* `localStorage` system in the web application and `AsyncStorage` system in the mobile application. In both of these systems, no data will be encrypted. Other possible security problems may arise in the future, but are out of the scope of the initial idea.

4.2 System structure

4.2.1 General System overview

The whole application system is divided into components. Principal components are

1. **The Server,**
2. **The Web client,**
3. **the Android and iOS client**

The detailed structure of the server and its connection with external interfaces are presented at figure 4.1. As seen in the diagram, the inside of the server is divided into components that are responsible for storing and processing data of application entities. This components are called *apps* in Django. Apps communicate with the database via Django models. Models in Django is an interface designed to simplify querying to the database. As seen in the diagram, the server provides interface for the client applications to communicate via Representational State Transfer (REST) API.

Django framework provides a web server for development only. So before stepping to production environment we have to configure a production ready web server. But traditional web servers do not understand or have any way to run Python applications. So Gunicorn and

Daphne application web servers are used to handle HTTP and Web-Socket requests respectively.

Further explanation of the chosen technologies and their components will be discussed on chapters 5, 6 and 7.

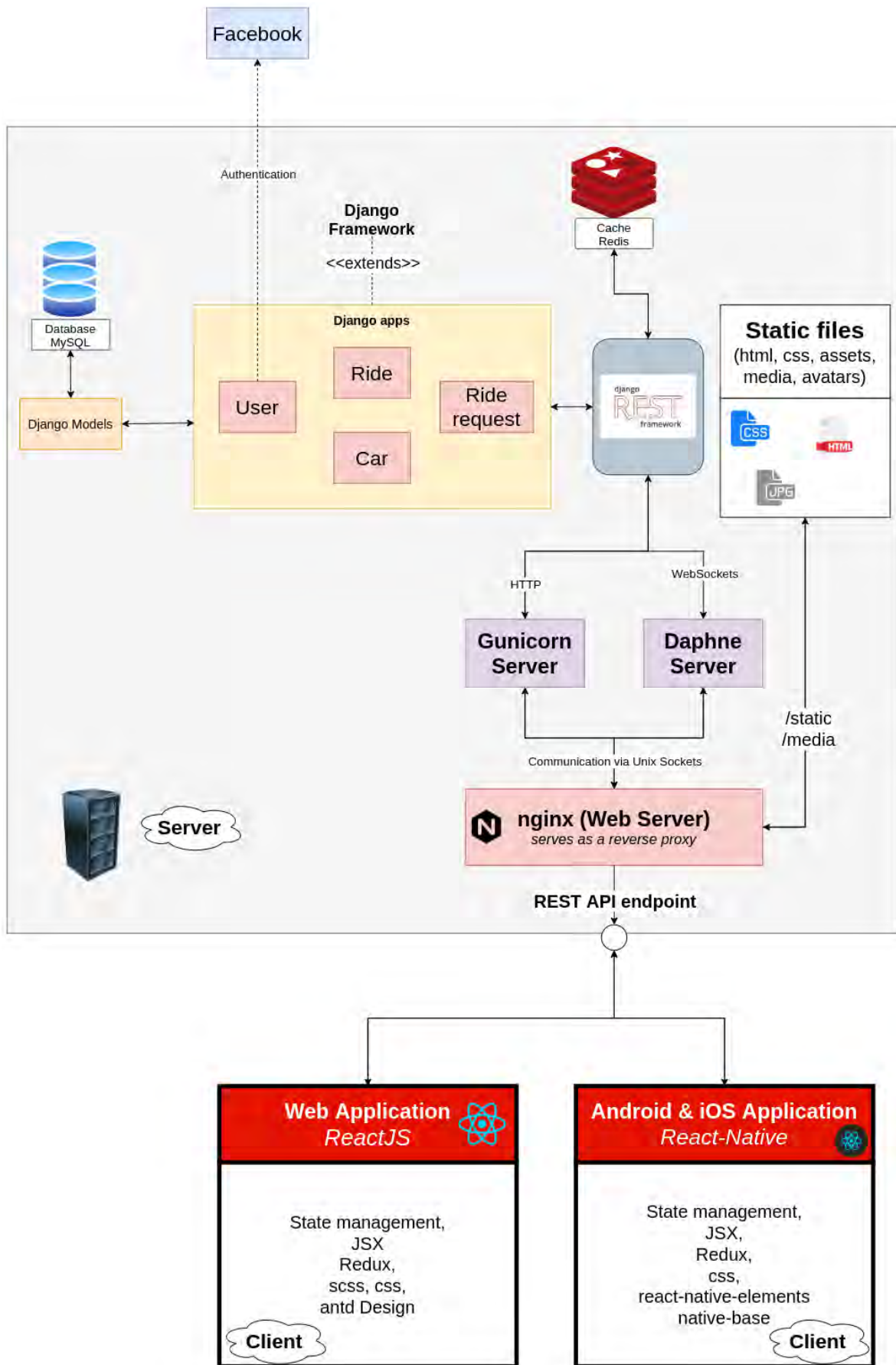


Figure 4.1: System Structure

4.2.2 Authentication

Although a non authenticated user can browse the rides users offer he/she cannot use the application in full extend. The user has to be authenticated in order to join or add rides. The carpooling application provides in-app registration as well as registration via the user's Facebook account. Authentication will be implemented with *JWT*[25]. When a user logs in/sign ups, the server will respond with a *JWT* token that the user will store. With this token, the user will be authenticated every time he makes a call to the back-end *API*.

Traditional authentication

The traditional authentication is conducted by registering the user's credentials to the server. The registration is served by django framework itself with the use of some 3rd-party add-ons which will be described later on 5. Passwords are hashed, but other data is stored plain-texted.

Facebook authentication

With the use of Facebook's Graph *API*, a user logs in to the application and the server receives all necessary information about the user.

Chapter 5

Application Design

5.1 Back-End

5.1.1 Chosen Technologies

Python

Python is the base of the server. It was chosen as a primary programming language because it was designed to be simple and highly readable, which is crucial for large-scale projects. Its syntax and standard library simplify and speed up development. [36]

Django

Django is an open source web framework for Python. It provides a high-level abstraction of common web development patterns. Django framework follows Model-View-Controller (*MVC*) design pattern. It uses *MVC* to separate model as a data and a business logic of the application, view as a representation of the information for the user, in this case, the client side of the application and controller as an interface of the application, in this case, set of URLs to communicate with front-end. [12]

Django REST

Django REST framework is an open source project built on Django framework. It contains needed tools for implementation of the *RESTful API* such as serializers, pagination, permissions and more. [14]

MySQL

MySQL is powerful, open source relational DBMS that provides tons of features and flexibility. It is used for small but also large scale applications. Django framework provides great *API* for working with MySQL databases. [27]

Nginx

Nginx[engine-x] is an *HTTP* and reverse proxy server, a mail proxy server, and a generic TCP1/UDP2 proxy server [30]. According to Netcraft, nginx served or proxied 25.58% busiest sites in July 2020 [31].

Gunicorn

Gunicorn [22] is a stand-alone *WSGI* web application server which offers a lot of functionality. It natively supports various frameworks with its adapters, making it an extremely easy to use drop-in replacement for many development servers that are used during development.

Redis

Redis(*Remote Dictionary Server*) is an in-memory data structure project implementing a distributed, in-memory key–value database with optional durability. Redis is often the most popular key-value database because it is considered at the same time a store and a cache. Redis provides a data model that is very unusual compared to a *RDBMS* which makes the retrieval of documents really fast.[57]

Daphne

As Gunicorn application server cannot handle `webSocket` requests, Daphne, a *HTTP*, *HTTP2* and `webSocket` protocol server for *ASGI* and *ASGI-HTTP*, is used. [8]

5.1.2 Used Libraries and Add-Ons

MySQLclient

MySQLclient adds Python 3 support to MySQLdb which is an interface to the popular MySQL database server that provides the Python database *API*. [28]

Django-cors-headers

A Django App that adds *CORS*[17] headers to responses. This allows in-browser requests to your Django application from other origins. Adding *CORS* headers allows your resources to be accessed on other domains. More on *CORS* headers on [7].

Django-allauth

Django-allauth is a python library developed for Django that supports multiple authentication schemes (e.g. login by user name, or by e-mail), as well as multiple strategies for account verification (ranging from none to e-mail verification). [16]

Django-rest-auth

django-rest-auth is a package that provides a set of REST API endpoints to handle User Registration and Authentication tasks, such as User Registration with activation, Login/Logout, Retrieve/Update the Django User model, Password change, Social Media authentication. [19]

Django-notifications

django-notifications is a GitHub notification alike app for Django. It is used to implement notifications into the application. [18]

Django Channels

Channels is a project that takes Django and extends its abilities beyond HTTP - to handle webSockets, chat protocols, *IoT* protocols and more. It's built on a Python specification called *ASGI*. It is used to implement webSocket connection and handling. [9]

5.2 Front-End

5.2.1 Chosen Technologies

JavaScript

JavaScript is a high-level, multi-paradigm language with a dynamic type system. JavaScript is considered to be a core technology of web development, present on more than 94% of all

websites. As a multi-paradigm language, JavaScript supports several programming styles, and even combines them; be it event-driven, functional, or an imperative paradigm. Although initially only aimed to be a client-side language for web browsers, JavaScript is currently also being used on web servers. Given that JavaScript is generally processed (using a JavaScript engine), the resulting interpreted (or compiled) code is typically platform agnostic. Furthermore it is easily debugged due to the existence of many implemented in-browser tools. [24]

React

React (also known as React.js or ReactJS) is an open-source JavaScript library for building user dynamic interfaces or *UI* components. It is maintained by Facebook, Instagram and a community of individual developers and companies. React utilizes so-called components as a main building block. Each component is either a JavaScript function (a stateless component), or a JavaScript class (a stateful component). A stateless component returns the HTML markup directly, whereas a stateful component extends the `React.Component` class and implements the `render()` method, which returns either another component, or HTML markup. [38]

React-Native

React Native is an addition to React universe. It is a framework developed for native application development for both iOS and Android. It uses the same design patterns and architecture principles as React and extends them, allowing us to build rich and friendly mobile user interfaces. While extending React, it also extends the native system (iOS or Android) giving access to native mobile components. [49]

Expo

Expo is a set of tools and services built around React Native and native platforms that help you develop, build, deploy, and quickly iterate on iOS, Android, and web apps from the same JavaScript/TypeScript codebase. React Native does not give you all the JavaScript *APIs* you need out of the box, but only most primitive features. Expo aims to enhance React Native and provide all the JavaScript *API* you need for the most common needs. [20]

5.2.2 Used Libraries and Add-Ons

Ant-design

Ant Design is a React *UI* library that has a set of components that are useful for building elegant user interfaces. It is created and maintained by Chinese Alibaba. [2]

Redux

Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. Its React binding is React-Redux. It lets your React components read data from a Redux store, and dispatch actions to the store to update data. [58]

Redux-Thunk

Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object. That function receives the store's dispatch method, which is then used to dispatch regular synchronous actions inside the body of the function once the asynchronous operations have completed. [60]

React Router

React Router is the standard routing library for React. From the docs: "React Router keeps your *UI* in sync with the *URL*". It has a simple *API* with powerful features like lazy code loading, dynamic route matching, and location transition handling built right in. [55]

Axios

HTTP client for the browser and node.js. [4]

MomentJS

MomentJS is a JavaScript library which provides an easy way of parsing, validating, manipulating and displaying date/time objects in JavaScript. [26]

React-facebook-login

A React component for Facebook Login. [42]

React-faq-component

A React package to render FAQ section. [43]

React-google-maps

A React Google Maps integration component. [45]

React Places Autocomplete

A React component to build a customized *UI* for Google Maps Places Autocomplete. [54]

React-awesome-button

React-awesome-button is a performant, extendable, highly customisable, production ready React Component that renders an animated set of 3D *UI* buttons. [39]

React Native Async Storage

An asynchronous, un-encrypted, persistent, key-value storage system for React Native. [3]

React Navigation

React Navigation provides routing and navigation for your React Native apps. [53]

NativeBase

Essential cross-platform and open-source *UI* components for React Native & Vue Native. [29]

React Native Elements

React Native Elements is a useful set of reusable components for your React Native application. [50]

React-native-maps

React Native Map components for iOS + Android. [51]

React-native-maps-directions

Directions component for react-native-maps. Helps draw a route between two coordinates, powered by the Google Maps Directions *API*. [52]

5.3 Used tools

5.3.1 GIT Version Control

Version control is a system that monitors changes in a file or a set of files and allows users backtrack to a previous state if needed.

Git is a distributed version control system for monitoring changes to source code during software and application development. It is designed to coordinate work between developers, but can also be used to monitor changes to any file set. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

5.3.2 PyCharm IDE

PyCharm is an *IDE* used in computer programming, specifically for the Python language. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (*VCSes*), and supports web development with Django. It is maintained by JetBrains and it provides educational licences for students. [35]

5.3.3 WebStorm IDE

WebStorm is a powerful *IDE* for modern JavaScript development. WebStorm provides full support for JavaScript, TypeScript, *HTML*, *CSS* as well as for frameworks such as React, Angular, and Vue. js right out of the box, no additional plugins are required. It is also maintained by JetBrains and it provides educational licences for students. [67]

5.3.4 PostMan

Postman is a software development tool. It enables people to test calls to *APIs*. It offers a sleek user interface with which to make HTTP requests, without the hassle of writing a bunch of code just to test an *API*'s functionality. [34]

5.3.5 React Developer Tools

React Developer Tools is a tool that allows you to inspect a React tree, including the component hierarchy, props, state, and more. It is used for debugging a React application. [40]

5.3.6 Redux DevTools

Redux Developer Tools to power-up Redux development workflow or any other architecture which handles the state change. It can be used as a browser extension (for Chrome, Edge and Firefox), as a standalone app or as a React component integrated in the client app. It is also used for debugging a React application combined with Redux. [59]

Chapter 6

Back-End Development

This chapter contains a description of the implementation of the project's server side. The first part will describe the structure of the project and the fundamentals of a Django project and the next parts will describe the implementation of the carpooling application. It is intended to familiarize the reader with the implementation of this application and to simplify the understanding of the structure of the project for future developers.

6.1 Django Fundamentals

6.1.1 Django General Structure

Django as a framework determines the structure of the whole system. Django project is divided into logical parts, apps. Apps contain a set of modules with classes, which implement interfaces and extend classes, which are provided by Django. A Django project composes of one or more Django apps.

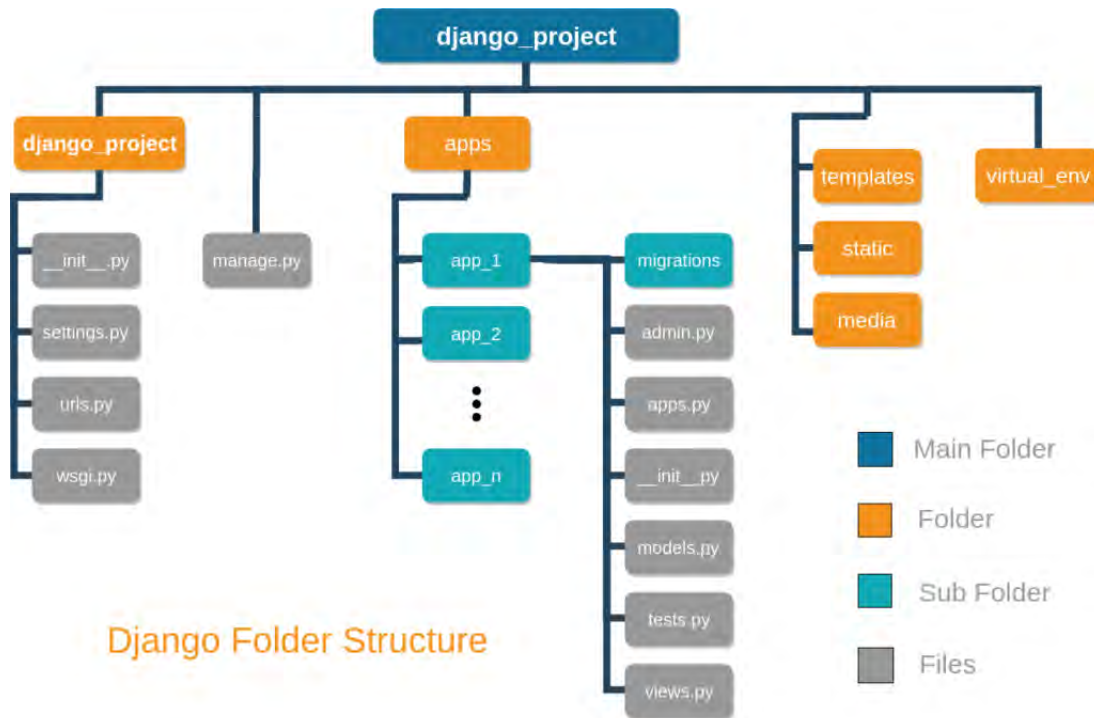


Figure 6.1: Django Structure general overview [13]

As shown in 6.1 we can define the following key modules:

- **django_project root folder.** This is the root folder of your Django application.
- **django_project.** The main Django application. It is the automatically generated. It is the entry point of all requests made to Django.
- **apps.** Our defined apps.
- **templates/static/media.** The folder that holds all static content that django application uses.
- **manage.py.** A command line utility for executing django commands.
- **virtual_env.** The folder in which virtual environment files(*installed packages, python variables etc*) are stored.

The main automatically generated app (*django_project* in the above figure), contains the following:

- **__init.py__.** The file that defines that the folder is a Python/Django package.

- **urls.py**. It contains URL configurations/paths.
- **settings.py**. Contains the global settings for the Django application.
- **wsgi.py & asgi.py**. Although not included in the above figure, they enable WSGI and ASGI compatibility.

Django doesn't require certain structure of files and folders, however the proposed way presented in Django Documentation has an underlying logic.

6.1.2 Django App Structure

To create a functional Django website, Django applications must be created. They add functionality, simplicity and re usability to the whole application because they divide the global application into several reusable parts.

Inside every app, Django creates the following files/folders:

- **migrations**. This folder stores migrations and changes to the database.
- **admin.py**. Where django models are registered to to the admin interface.
- **apps.py** is a configuration file for the app.
- **models.py**. Contains the models of the app.
- **tests.py**. Contains test procedures for testing the app.
- **views.py**. Contains the logic and the rendering process of the app.
- **__init__** The file that defines that this folder (your Django app) is a Python package.

6.1.3 Django request cycle

The flow shown in 6.2 depicts the request-response cycle of a Django application. Django architecture is divided in several layers that handle gradually a request made to Django.

In a general Django application, a client issues a request to the server following the *HTTP* protocol. The first layer to handle that request are **Request Middleware**. They provide functions like authentication, authorization and session management. Afterwards, the request is passed to the **URL Router** where the path of the specific request will be match

with the application's implemented URL paths. When a request is match it will be handled to the **View** to be processed. The logic business of the Django application takes place in the View layer. From there, access and processing of the database is made. The View is responsible of rendering the Response(*HTML/XML/JSON/etc.*) with the data that's retrieved from the database. [66]

In this project, Django REST framework is used to extend Django. In short, Django is transformed in a REST API. In depth, *DRF* extends the Django Middleware layer, adding parsers, functions and other services/modules. More on [10]

One of these modules are **Serializers**:

This component transforms Django data. They allow complex data as query sets to be converted to Python data types, that can me easily rendered into *JSON*, *XML* and other formats. They also provide deserialization, allowing parsed data to be transformed into complex model types.

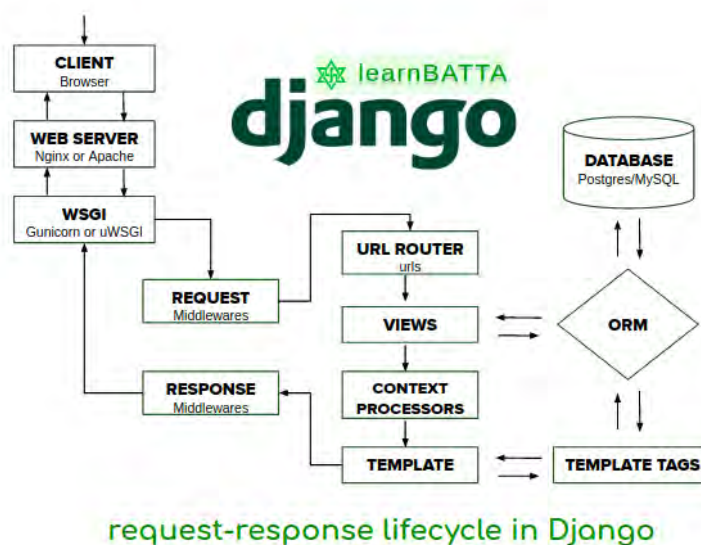


Figure 6.2: Django Life Cycle [11]

6.2 Setting up prerequisites

6.2.1 Setting up MySQL and Redis

In this section, we will briefly discuss, the database set up.

First of all, we have to setup our database and our cache server. We first install MySQL and Redis servers.

```
sudo apt install mysql-server && redis-server # installing mysql
and redis
sudo apt install python3-dev libmysqlclient-dev
default-libmysqlclient-dev # install dependencies for mysql
```

Secondly, we set up our MySQL database, We Login via the MySQL root user, create a database, create a new User and link the newly created database to the newly created user.

Lastly, we must ensure that MySQL and Redis servers are up and running.

6.2.2 Setting up Django

In this section we explain briefly some configurations a developer must do, in order for Django to communicate successfully with MySQL and Redis, as well as some other minimum configurations to get a project started.

Firstly we create our new virtual environment in which django would run.

```
python3 -m venv env # last parameter is the name of the new
virtual environment
. env/bin/activate # we activate our virtual env
pip install django # we install django
django-admin startproject.djangoproject # we start a project
named 'djangoproject'
```

Secondly we edit, *settings.py* and we make the following configurations:

```
TIME_ZONE. We set our time zone.
LANGUAGE_CODE. We set our language.
STATIC_ROOT = os.path.join(BASE_DIR, 'static')$
ALLOWED_HOSTS = ['your server IP address']$
DATABASES = {
```

```
'default': {
    'ENGINE': 'django.db.backends.mysql',
    'NAME': OUR DATABASE NAME,
    'USER': THE USER WE SET UP,
    'PASSWORD': THE PASSWORD OF THE USER,
    'HOST': '127.0.0.1',
    'PORT': '3306',
}
}
REDIS_URL = os.getenv('REDIS_URL', 'redis://localhost:6379')
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [REDIS_URL],
        },
    },
},
}
```

6.3 Implementation

6.3.1 Django Apps

The project was divided into the following apps:

- user. This app includes modules for storing and processing information about the user.
- cars. This app includes modules for storing and processing users' cars.
- rides. This app includes modules for storing and processing users' rides.
- notifier. This app, in addition with "django-notifications" is responsible for processing users' notifications.
- rideRequests. This app includes modules for storing and processing users' requests to offered rides.

6.3.2 Django Models

Django models is an interface for simplified querying to the database. Thus, each app, contains a module *models*. In this module, there are models that completely describe the database. Despite what kind of DBMS is used (PostgreSQL or SQLite or MySQL), managing it and configuring in is a complex process. Especially in SQL databases making queries for creating, deleting or updating can be complicated. Django models simplify every database-related task and provide an easy and simple environment for development, testing and deployment.

Model that represent a User in the database:

```
class User(AbstractUser):
    GENDER = Choices(
        ('M', 'Male'),
        ('F', 'Female'),
        ('O', 'Other')
    )
    dob = models.DateField(max_length=8,
        default=datetime.date(1999, 12, 31))
    phone_number = models.TextField(max_length=12, blank=True,
        null=True, verbose_name='Τηλέφωνο')
    avatar = models.ImageField(upload_to='avatar/',
        default='avatar/default-avatar.jpg', blank=True)
    gender = models.CharField(blank=True, null=True, max_length=1,
        default=GENDER.O, choices=GENDER, verbose_name='Φύλο')
    country = models.CharField(max_length=3, blank=True,
        verbose_name='Χώρα', null=True)

    has_whatsup = models.BooleanField(blank=True, null=True,
        verbose_name='Whats up messenger platform')
    has_viber = models.BooleanField(blank=True, null=True,
        verbose_name='Viber messenger platform')

    is_confirmed = models.BooleanField(default=True,
        verbose_name='Επιβεβαιωμένος χρήστης')
```

```
# for future confirmation by admins of user ID or PASSPORT
```

6.3.3 Django Views

Django view is a method that is called during request on certain *URL*. This function takes a *HTTP* request and returns a web response. In the case of Django *REST* framework, a *JSON* response is returned. The main logic of processing requests is in the views. Django *REST* framework simplify the implementation of such functions and the corresponding auxiliary classes (serializers, pagination, etc.)

Here's a view that returns the current date and time, as an *HTML* document:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Django provides a more generic way to implement views as Python objects instead of defined functions.

In a class-based view, the previous example would become:

```
from django.http import HttpResponse
from django.views import View
import datetime

class MyView(View):
    def get(self, request):
        now = datetime.datetime.now()
        html = "<html><body>It is now %s.</body></html>" % now
        return HttpResponse(html)
```

Django focuses on code reusability and has implemented generic views for that purpose. They take common patterns found in web development(for example *CRUD* operations) and abstract them so that a developer can quickly write common views without much complexity.

Using generic views can speed up development substantially. Django comes with a handful of built-in generic views to help implement list and detail views of objects, but also views that create or edit objects.

Say we have a model called Book. The example below shows the simplicity of returning a list of instances of Book model, to the front-end.

```
from django.views.generic import ListView
from .models import Book

class PublisherList(ListView):
    model = Book
```

The view of getting the list of offered rides looks like:

```
class RideListView(ListAPIView):
    queryset = Ride.objects.all()
    serializer_class = RideListSerializer
    permission_classes = [AllowAny, ]

    def get_queryset(self):
        queryset = Ride.objects.all().order_by('created')
        origin = self.request.query_params.get('origin', None)
        # print(origin)
        if origin is not None:
            queryset = queryset.filter(origin__contains=origin)

        destination = self.request.query_params.get('destination',
            None)
        if destination is not None:
            queryset =
                queryset.filter(destination__contains=destination)

        date = self.request.query_params.get('date', None)
        if date is not None:
            print(date)
            queryset = queryset.filter(date=date)
```

```
vacant_seats = self.request.query_params.get('passengers',
None)

if vacant_seats is not None:
    print(vacant_seats)
    queryset = queryset.filter(vacant_seats__gte=vacant_seats)

return queryset
```

6.3.4 Django URLs

The `urls` module in Django is responsible for linking the *URL* endpoints to their corresponding views. It contains a list of path objects. In `rides` app it looks like this:

```
urlpatterns = [
    path('', RideListView.as_view(), name='list_rides'),
    path('myrides/', MyRidesListView.as_view(), name='myrides'),
    path('create/', RideCreateView.as_view(), name='create_ride'),
    path('<int:pk>/', RideDetailView.as_view(), name='detail_ride'),
    path('<int:pk>/edit/', RideEditView.as_view(),
        name='edit_ride'),
    path('', include('rideRequests.urls')),
]
```

6.3.5 Django Serializers

Django REST serializers is an interface that provides the Django REST framework for simplifying the serialization and deserialization of instances of Django models. Serializers allow the two-way transformation of complex Django data to JSON format. One simple serializer in `ride` app looks like this:

```
class RideListSerializer(serializers.ModelSerializer):
    uploader = SimpleUserSerializer(read_only=True)
    time = serializers.TimeField(required=False)
```

```
class Meta:
    model = Ride
    fields = ('pk', 'origin', 'destination', 'type',
             'date', 'time', 'periodic', 'vacant_seats',
             'uploader',
             )

    depth = 1
```

6.3.6 Custom Middleware

As stated earlier, authentication is achieved with *JWT* tokens. After the client logs in, the server responds with a *JWT* token to be stored in the client for future *CRUD* operations. As the user requests a secured resource from the server with an *HTTP* request, it is necessary to identify who this user is and if he/she has permission to access that resource. So the client sends the *JWT* token with the requests headers and the server captures it into a custom middleware, decode it, and authenticates the user.

The custom middleware used in capturing *JWT* token and authenticating the user in *HTTP* requests is:

```
class AuthenticationMiddlewareJWT(object):
    def __init__(self, get_response):
        self.get_response = get_response
        # print('middlewareSTART', flush=True)

    def __call__(self, request):
        # print('middleware')
        request.user = self.__class__.get_jwt_user(request)
        # print(request.user)
        return self.get_response(request)

    @staticmethod
    def get_jwt_user(request):
```

```
# print(request.headers)

user = get_user(request)
# print(user, flush=True)
if user.is_authenticated:
    return user
try:
    user_jwt =
        JSONWebTokenAuthentication().authenticate(Request(request))
    # print(user_jwt, flush=True)
    if user_jwt is not None:
        return user_jwt
except:
    pass
return user # AnonymousUser
```

Similar to *HTTP* requests, *webSocket* requests must be authenticated. So, the client due to the fact that the browsers do not support passing *JWT* authentication headers on *webSocket* upgrade, sends the *JWT* as parameters in the URL. **This is totally insecure without using *HTTPS*.** I have also written another custom middleware for *webSocket* connections.

6.3.7 Django *webSocket* Routing

Normally, a client communicates with the Django application with *HTTP* protocol.

1. The client sends an *HTTP* request to the server.
2. Django parses the request, extracts a *URL*, and then matches it to a view.
3. The view processes the request and returns an *HTTP* response to the client.

Using *HTTP* protocol, for a server to send a response, firstly it has to receive a request from a client. By using *webSockets* protocol a server can send data to the client without the latter requesting it. They allow bi-directional communication. *WebSocket* messages are sent using the *ws(s)://* prefix, as opposed to *http(s)://* which is the prefix for *HTTP/HTTPS* requests.

The routing.py:

```
websocket_urlpatterns = [  
    path(r'ws/', consumers.MyConsumer),  
]
```

Consumers module, handle the connections between the clients and the server. Consumers are similar to Django views. If a client connects to the application, he/she will be added to the "users" group and will be able to receive messages on certain events. When this client disconnects from the application, the channel is removed from the group, thus the user will stop receiving messages.

My consumer:

```
class MyConsumer(WebSocketConsumer):  
    def connect(self):  
        # Checking if the User is logged in  
        if self.scope["user"].is_anonymous:  
            # Reject the connection  
            self.close()  
        else:  
            self.group_name = str(self.scope["user"].pk) # Setting  
                the group name as the pk of the user #  
            # primary key as it is unique to each user. The group  
                name is used to communicate with the user.  
            async_to_sync(self.channel_layer.group_add)(self.group_name,  
                self.channel_name)  
            self.accept()  
  
        # Function to disconnect the Socket  
    def disconnect(self, close_code):  
        self.close()  
        # pass  
  
        # Custom Notify Function which can be called from Views or api  
        # to send message to the frontend  
    def notify(self, event):
```

```
self.send(text_data=json.dumps(event["text"]))

def addRequests(self, event):
    self.send(text_data=json.dumps(event))

def removeRequests(self, event):
    self.send(text_data=json.dumps(event))

def removeMYRequests(self, event):
    self.send(text_data=json.dumps(event))

def updateRequests(self, event):
    self.send(text_data=json.dumps(event))

def sendNotification(self, event):
    self.send(text_data=json.dumps(event))
```

By that process we achieve bidirectional and asynchronous communication between the client and the server.

6.3.8 Django Settings

Django settings is a module that contains all the configuration of the Django project. The main configurations to notice are:

- **INSTALLED_APPS**. A list of all apps in the project.
- **ALLOWED_HOSTS**. A list of the host/domain names that this Django site can serve.
- **REST_FRAMEWORK.PAGE_SIZE**. The page size in terms of objects, in the JSON response of Django REST framework.
- **DEBUG**. Turn on/off debug mode.
- **DATABASES**. The settings for all databases to be used with Django.

A complete list of settings available in Django can be found in the official documentation [15].

6.3.9 Authorization and authentication

Due to the fact that the front-end is a single page application, JSON Web Tokens(*JWT*) were chosen as the best practice. On the back-end side Django Rest Framework provides many plugins that allow authentication through the use of *JWT* tokens. The use of *JWT* allows the storing of any kind of information in the "payload" part of the token. In our implementation we will be storing user information as the user identifier (User Primary Key), his/her token, and other vital information for the front-end.

In order to use *JWT* authentication effectively, we must configure again *settings.py*. We set the following:

```
JWT_AUTH = {
    'JWT_EXPIRATION_DELTA': timedelta(hours=100),
    'JWT_ALLOW_REFRESH': True,
    'JWT_SECRET_KEY': SECRET_KEY,
    'JWT_AUTH_HEADER_PREFIX': 'JWT',
}

REST_USE_JWT = True
REST_SESSION_LOGIN = False
```

In addition, because a user should be able to login/signup via Facebook, we configure *django-all-auth* as:

```
SOCIALACCOUNT_PROVIDERS = {
    'facebook': {
        'METHOD': 'js_sdk',
        'SDK_URL': '//connect.facebook.net/{locale}/sdk.js',
        'SCOPE': ['email', 'public_profile', 'picture', 'id'],
        'AUTH_PARAMS': {'auth_type': 'reauthenticate'},
        'INIT_PARAMS': {'cookie': True},
        'FIELDS': [
            'id',
            'email',
            'name',
            'first_name',
            'last_name',
```

```
        'verified',
        'locale',
        'timezone',
        'link',
        'gender',
        'updated_time',
    ],
    'EXCHANGE_TOKEN': True,
    'LOCALE_FUNC': lambda request: 'en_US',
    'VERIFIED_EMAIL': False,
    'VERSION': 'v2.12',
}
}
```

6.4 Deployment

The back-end is deployed on Okeanos. Okeanos is GRNET's cloud service, for the Greek Research and Academic Community. The Virtual Machine that Okeanos provides runs Ubuntu 18.04 OS. After finishing implementing the application and connecting it with the database, it is essential to configure Gunicorn and Daphne application servers. Daphne can serve *HTTP* requests as well as WebSocket requests. For stability and performance, we will use Gunicorn to serve HTTP requests and Daphne to serve webSocket requests. We will also set up Nginx in front of Gunicorn/Daphne to take advantage of its load-balancing mechanisms and its security features.

6.4.1 System general configuration

We will use supervisor as our process management system. Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems[63]. Supervisor will be responsible of starting, restarting, and stopping Gunicorn and Daphne services. The use of supervisor simplifies the control of services because it provides a web interface, served on port 9001.

```
sudo apt-get install supervisor
```

```
nano /etc/supervisor/conf.d/gunicorn.conf # creation of Gunicorn
configuration file
```

We enter the following configuration. We have to ensure that the path to gunicorn executable is correct.

```
[fcgi-program:gunicorn]
# TCP socket used by Nginx backend upstream
socket=tcp://localhost:8001

# Directory where your site's project files are located
directory=/home/user/api/carsharing_backend

# Each process needs to have a separate socket file, so we use
process_num

# Make sure to enter the correct path to gunicorn!
command=/home/user/api/venv/bin/gunicorn
    django_backend.wsgi:application --name "gunicorn" --workers=4
    --capture-output --log-level=debug --bind
    unix:/run/gunicorn/gunicorn.sock

# Automatically start and recover processes
autostart=true
autorestart=true

# Choose where you want your log to go
stdout_logfile=/var/log/supervisor/gunicorn.log
redirect_stderr=true
```

6.4.2 NGINX configuration

Gunicorn and Daphne application servers are designed to sit behind a reverse proxy server that handles load balancing, caching, and preventing direct access to internal resources.

Django and Gunicorn/Daphne in extent, are not designed to serve static files. So NGINX has to be configured to serve them separately.

In order for our production server to be secure, the use of HTTPS is mandatory. *HTTPS* is *HTTP* with *TLS()* encryption. *HTTPS* uses *TLS (SSL)* to encrypt normal *HTTP* requests and responses, making it safer and more secure. HTTPS provides critical security and data integrity for both the server and the users' personal information. It prevents intruders from being able to passively listen to communications between the server and the clients.

To set up HTTPS and its dependant certificates we make use of the global non-profit Certificate Authority (CA) Let's Encrypt, which is sponsored by Electronic Frontier Foundation (*EFF*), the Mozilla Foundation, Google and other organizations and companies. The **Certbot** tool is used to add HTTPS certificate to the back-end website. Certbot is a free, open source software tool for automatically using Let's Encrypt certificates and is developed by *EFF*, a 501(c)3 nonprofit based in San Francisco, CA, that defends digital privacy, free speech, and innovation. It provides an easy transition from *HTTP* to *HTTPS* technology without much hustle.

Below is the NGINX configuration file we have used for our deployment that combines both the configuration for Django and the Let's Encrypt certificates:

```
upstream channels-backend {
    server localhost:8000;
}

server {
    server_name snf-876572.vm.okeanos.grnet.gr;

    charset utf-8;
    # max upload size
    client_max_body_size 75M; # adjust to taste

    # Favicon
    location = /favicon.ico { access_log off; log_not_found off; }

    # Django static
    location /static {
        alias /home/user/api/carsharing_backend/static;
    }
}
```

```
# Django media
location /media {
alias /home/user/api/carsharing_backend/media;
}

# send all other http(s) requests to gunicorn
location / {
    include proxy_params;
    proxy_pass http://unix:/run/gunicorn/gunicorn.sock;
}

# send all ws requests to daphne
location /ws/ {
    try_files $uri @proxy_to_app;
}

location @proxy_to_app {
    proxy_pass http://channels-backend;

    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";

    proxy_redirect off;
    proxy_set_header Host $host;

    proxy_set_header Authorization $http_authorization;

    proxy_pass_header Authorization;

    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Host $server_name;
```

```
proxy_set_header X-Forwarded-Proto $scheme;
}

# lets encrypt auto
listen 443 ssl; # managed by Certbot
ssl_certificate
    /etc/letsencrypt/live/snf-876572.vm.okeanos.grnet.gr/fullchain.pem;
    # managed by Certbot
ssl_certificate_key
    /etc/letsencrypt/live/snf-876572.vm.okeanos.grnet.gr/privkey.pem;
    # managed by Certbot
include /etc/letsencrypt/options-ssl-nginx.conf; # managed by
    Certbot
ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by
    Certbot
}

server {
    if ($host = snf-876572.vm.okeanos.grnet.gr) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    listen 80;
    server_name snf-876572.vm.okeanos.grnet.gr;
    return 404; # managed by Certbot
}
```

6.4.3 Server Hardening

Firewall

Although that Okeanos Virtual Machines are under a firewall, the use of a local one is beneficial. There are a few options available when it comes to Linux firewalling, includ-

ing *UFW*(*Uncomplicated FireWall*) and *iptables*. We use *UFW*, because it is more user-friendly, easier to understand and will take care of generating the required rules for *iptables*.

SSH

SSH is the protocol that is used to connect to our server. We have implemented SSH passwordless authentication with the use of public-private keys. Also, our server does not accept ssh login to root user and there is a limit for password attempts.

Intrusion-Prevention System

Intrusion-Prevention systems monitor log files and search for particular patterns that correspond to a failed login attempt. Fail2ban is such a system. If a certain number of failed logins are detected from a specific IP address (within a specified amount of time), fail2ban blocks access from that IP address.

Chapter 7

Front-End Development

7.1 React Fundamentals

7.1.1 JSX

JavaScript XML (JSX) is a syntax extension to JavaScript. In short, it is JavaScript and *HTML* combined. It converts *HTML* tags into react elements. While *JSX* looks like *HTML*, it is actually just a neater and cleaner way to write a `React.createElement()` declaration.

With *JSX*:

```
const myelement = <h1>I Love JSX!</h1>;  
ReactDOM.render(myelement, document.getElementById('root'));
```

Without *JSX*:

```
const myelement = React.createElement('h1', {}, 'I do not use  
JSX!');  
ReactDOM.render(myelement, document.getElementById('root'));
```

```
App.js
1  import React, { Component } from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4
5  class App extends Component {
6    render() {
7      return (
8        <div className="App">
9          <div className="App-header">
10             <img src={logo} className="App-logo" alt="logo" />
11             <h2>Welcome to React</h2>
12           </div>
13           <p className="App-intro">
14             Hello Codecademy!
15           </p>
16         </div>
17       );
18     }
19   }
20
21   export default App;
22
```

Figure 7.1: React example [41]

7.1.2 State

React Class Components provide an instance of properties can control their behavior. In other words, the State of a component is an object that holds some information that may change over the lifetime of the component. [56]

```
import React, { Component } from 'react';

class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list: [1, 2, 3],
    };
  }

  render() {
    return (
      <div>
        <ul>
          {this.state.list.map(item => (
            <li key={item}>{item}</li>
          ))}
        </ul>
      </div>
    );
  }
}

export default App;
```

Figure 7.2: State example [62]

As shown in the example an array of 3 elements is set as State and is rendered as an ordered list in the *DOM*. If the user implements an `onChange` function that changes the State (the list array), the Component shall be re-rendered.

7.1.3 Virtual Document Object Model

The *HTML DOM* was designed to serve static content. When it updates, every node rebuilds and the page is repainted with the new data. Single Page Applications contain many dynamically generated nodes, that tend to change with every user reaction. So in order not to update the *HTML DOM* at a regular interval, the *Virtual DOM* was introduced.

The *Virtual DOM* is just a virtual/abstract representation of the *HTML DOM*. Each time the application state changes, the *Virtual DOM* is updated instead of the real *DOM*. When updates occur in *VDOM*, React makes these changes to the real *DOM* in the most efficient way. The image below shows the virtual *DOM* tree and the diffing process. More

on [64].

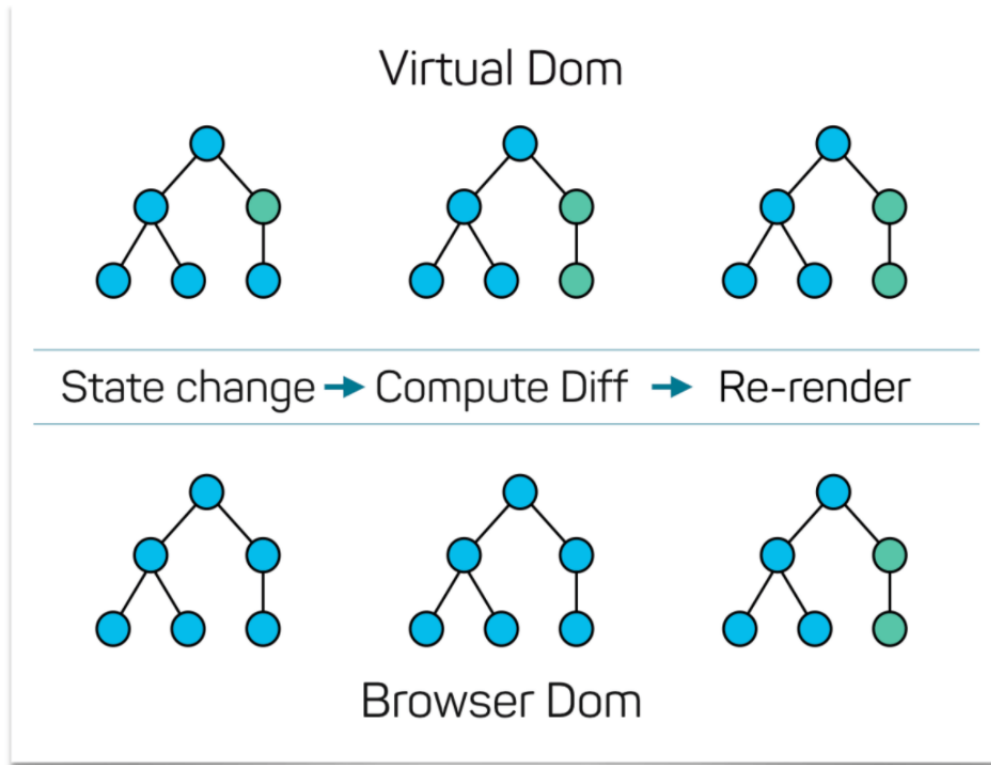


Figure 7.3: Virtual DOM and Browser DOM Comparison [68]

Every node represent a *UI* element. The nodes coloured green had their state updated. The difference between the previous version of the virtual *DOM* tree and the current virtual *DOM* tree is then calculated. The whole parent subtree then gets re-rendered to give the updated *UI*. This updated tree is then updated to the real *DOM*.

7.1.4 React Lifecycle Methods

React components go through a lifecycle, whether our code knows about it or not. *You can think of React lifecycle methods as the series of events that happen from the birth of a React component to its death.*[48]

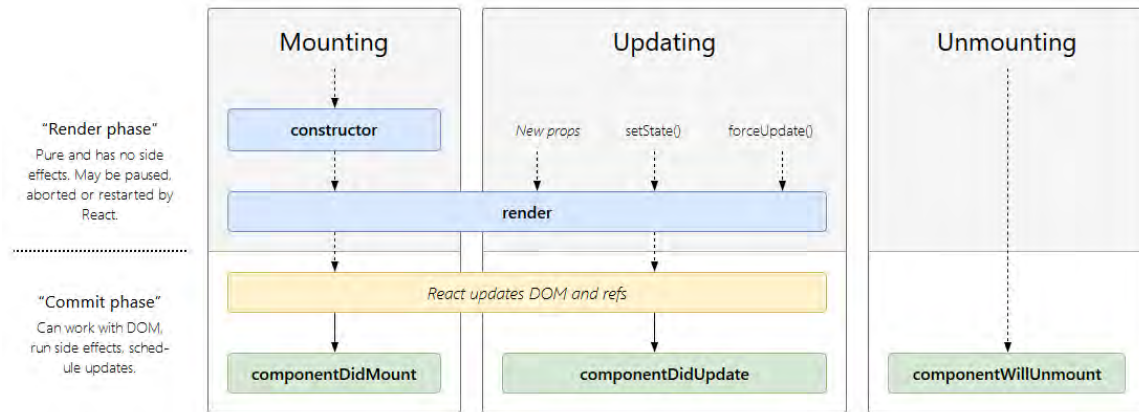


Figure 7.4: React lifecycle methods [47]

7.1.5 Redux

As we have seen, a typical JavaScript application is full of state. As the requirements for JavaScript single-page applications have become increasingly complicated, our code must manage more state than ever before. Managing this ever-changing state is hard and complicated. Big applications have big application states and managing them gets more and more inconvenient as applications grows. Redux is a popular JavaScript library for managing that state that our application holds. It uses a global state that is stored in a tree called *store*. Redux has 3 main parts:

1. **Actions**
2. **Reducers**
3. **Store**

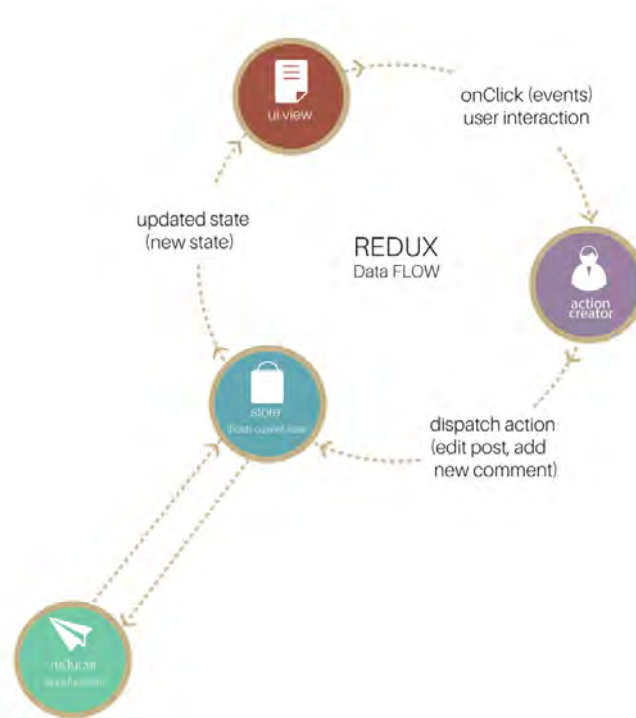


Figure 7.5: Redux data flow [1]

Actions

To put it simply, actions are events. They send data from the application (user interactions, API calls, form submissions etc) to the store. The store gets information only from actions. Internal actions are simple JavaScript objects that have a `type` property, describing the type of action and `payload` of information being sent to the store.

```
{
  type: LOGIN_FORM_SUBMIT,
  payload: {username: 'nick', password: '123456'}
}
```

Actions are created with action creators:

```
function authUser(form) {
  return {
    type: LOGIN_FORM_SUBMIT,
    payload: form
  }
}
```



```
}
```

To call an action anywhere in the app the use of dispatch method is mandatory:

```
dispatch(authUser(form));
```

Reducers

In Redux, reducers are functions (pure) that take the current state of the application and an action and then return a new state.

```
function handleAuth(state, action) {  
  return _.assign({}, state, {  
    auth: action.payload  
  });  
}
```

Store

Store is the object that holds the application state and provides a few helper methods to access the state, dispatch actions and register listeners. The entire state is represented by a single store. Any action returns a new state via reducers.

```
import { createStore } from 'redux';  
let store = createStore(rootReducer);  
let authInfo = {username: 'alex', password: '123456'};  
store.dispatch(authUser(authInfo));
```

7.2 Web Application Development

7.2.1 Setup

We start by setting up the React project. According to its documentation, there are several options available, depending on the application's purpose. **Create React App** is an officially

supported way to create single-page React applications. It offers a modern build setup with no configuration. Following its documentation[44] we execute:

```
npx create-react-app my-app  
cd my-app  
npm start
```

npx comes with npm 5.2+ and higher. The above commands will create a React app with name "my-app" and, will run the app in development mode. In order to view it, we have to access <http://localhost:3000>. The default react structure is shown in 7.6.

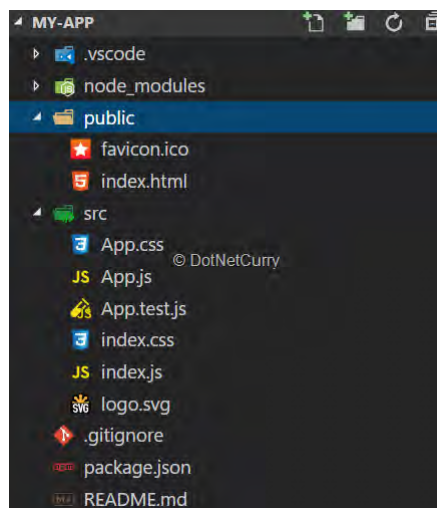


Figure 7.6: Default React Structure [46]

7.2.2 Application Structure

We have broken down our application into folders and separate files for our development to be more easy to understand and configure.

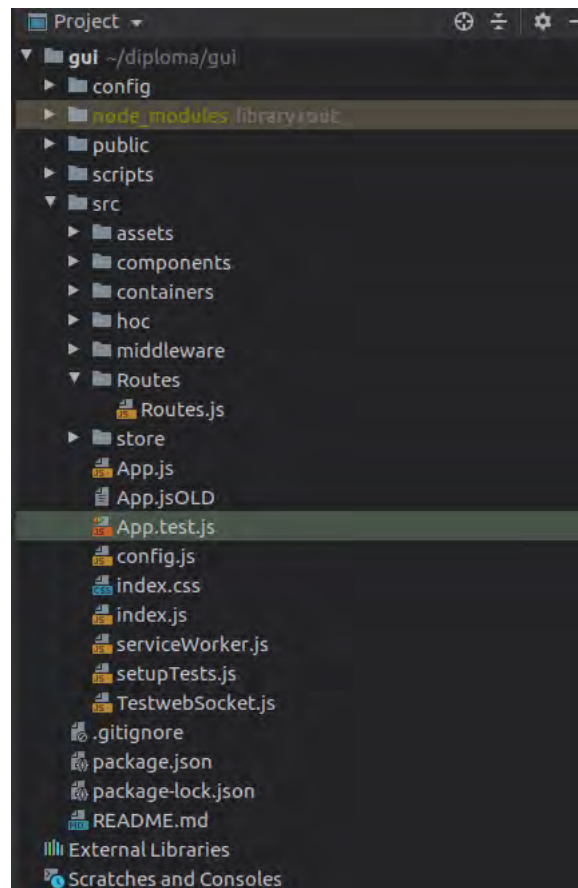


Figure 7.7: My React Structure

We can stand out the following modules:

- **public**. Contains public files, such as favicons, manifest.json and the entry file *index.html*.
- **scripts**. It contains the scripts required to run, build and test the React app.
- **package.json**. This file is kind of a manifest for your project. It contains our project dependencies, its description, name, version etc.
- **src**. This is the root source folder.
- **assets**. The assets(px images) used by React app.
- **containers**. Modules that contain state, statefull components.
- **components**. Modules that our free of state, stateless components.
- **hoc**. High Order Component. It contains the components defining the Layout of the application.

- **middleware.** Contains the middleware files for our app.
- **Routes.** Defines the routing endpoints of our React app.
- **store.** Defines the Redux Store. Contains actions and reducers.
- **App.js.** The second accessed file of our app. Here we add the previously defined routing in our app.
- **config.js.** Configuration variables are stored here like Facebook ID, GOOGLE MAPS API key etc.
- **index.js.** The main JavaScript file of our app. Defines the store and its dependencies.
- **serviceWorker.js.** Default script that your browser runs in the background, separate from a web page, opening the door to features that don't need a web page or user interaction. **We make not use of serviceWorkers.**

7.2.3 Implementation

In this chapter some major components implementations of the React app are shown.

index.js

```
// enhance a store with applyMiddleware and a few developer tools
// from the redux-devtools package.
const composeEnhancers =
  window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const rootReducer = combineReducers({
  auth: authReducer, // reducer for authentication
  rides: ridesReducer, // reducer for listing rides
  myrides: myRidesReducer, // reducer for listing my rides
  ride: rideReducer, // reducer for showing selected ride
  websocket: websocketReducer // reducer that store websocket
    connection
});

// redux middleware
```

```
// thunk: Allows to execute async code in Redux
// webSocketsMiddleware: custom middleware that provides websocket
// connections.
const middleware = [thunk, webSocketsMiddleware];

// defining one store!
export const store = createStore(rootReducer, composeEnhancers(
  applyMiddleware(...middleware)
));

const app = (
  <Provider store={store}>
    <App/>
  </Provider>
);

ReactDOM.render(app, document.getElementById("root"));
serviceWorker.unregister();
```

webSocketsMiddleware

This custom implemented middleware provides websocket support for our React application. It opens/closes a websocket connection and handles received messages via websockets from the server.

```
const webSocketsMiddleware = (function () {
  let socket = null;
  /**
   * Handler for when the WebSocket opens
   */
  const onOpen = (ws, store, host) => event => {
    // Authenticate with Backend... somehow...
    // console.log(event, host);
    store.dispatch(webSocketActions.websocketConnectSuccess(event.target.url));
  };
  /**
```

```
* Handler for when the WebSocket closes
*/

const onClose = (ws, store, host) => event => {
  store.dispatch(webSocketActions.webSocketDisconnect(host));
  console.log('Socket is closed', event.reason);
  // setTimeout(() => {
  //
  //   store.dispatch(webSocketsActions.webSocketConnect(host));
  // }, 5000);

};

/**
 * Handler for when a message has been received from the server.
 */

const onMessage = (ws, store) => event => {
  const payload = JSON.parse(event.data);
  // console.log(payload);
  switch (payload.type) {
    case actionTypes.WS_MESSAGE:
      store.dispatch(webSocketActions.webSocketMessage(event.host,
        payload));
      break;
    case "addRequests":
      // console.log('add');
      store.dispatch(requestsActions.addRequestsOfMyRides(payload.text));
      break;
    case "removeRequests":
      // console.log('remove');
      store.dispatch(requestsActions.removeRequestsOfMyRides(payload.text));
      break;
    case "removeMYRequests":
      // console.log('removeMY');
      store.dispatch(requestsActions.removeRequest(payload.text));
      break;
  }
}
```

```
    case "updateRequests":
      // console.log('updateRequests');
      store.dispatch(requestsActions.updateRequests(payload.text));
      break;
    case "sendNotification":
      // console.log('sendNotification');
      console.log(JSON.parse(payload.text));
      store.dispatch(notifActions.receiveNotification(JSON.parse(payload.text)));
      break;

    default: console.log('default'); break;
  }
};
/**
 * Middleware
 */
return store => next => action => {
  // console.log(action.type);
  switch (action.type) {
    case actionTypes.AUTH_SUCCESS:
      if (socket !== null) {
        socket.close();
      }
      // Pass action along
      next(action);
      // // Tell the store that we're busy connecting...
      store.dispatch(webSocketActions.webSocketConnectStart(API_WS));
      let user = JSON.parse(localStorage.getItem('user'));
      // console.log(user);
      // Attempt to connect to the remote host...
      socket = new WebSocket(API_WS + `?token=${user.token}`);

      // Set up WebSocket handlers
      socket.onmessage = onMessage(socket, store);
    }
  }
}
```

```

    socket.onclose = onClose(socket, store, action.host);
    socket.onopen = onOpen(socket, store, action.host);
    break;
  case 'NEW_MESSAGE':
    // console.log('sending a message', action.msg);
    socket.send(JSON.stringify({ command: 'NEW_MESSAGE',
      message: action.msg }));
    break;
  case actionTypes.AUTH_LOGOUT:
    if (socket !== null) {
      socket.close()
    }
    socket = null;
    // Tell the store that we've been disconnected...
    store.dispatch(webSocketActions.webSocketDisconnect(action.host));
    break;
  default:
    return next(action);
}
};
})(());
export default webSocketsMiddleware;

```

Routes

This is the routing configuration used for implementing this React app.

```

const Routes = (props) => {
  let isMobile = props.isMobile;
  let [myrides, setMyRides] = useState();
  let location = useLocation();
  const pk = location.pathname.split('/')[2];

  useEffect(() => {
    if (!myrides) {

```



```

        setMyRides (props.myrides);
    }
}, [myrides, props.myrides]);

```

```

return (
  <Switch>
    <Route exact path="/" component={Home}/>
    {props.isAuthenticated ? <Route exact path="/user/:id"
      component={User}/> : null}
    {props.isAuthenticated ? <Route exact path="/myaccount"
      render={(props) => <MyAccount {...props}
        isMobile={isMobile}/>}/>: null}
    {props.isAuthenticated ? <Route exact path="/mysettings"
      render={(props) => <MySettings {...props}
        isMobile={isMobile}/>}/>: null}
    {props.isAuthenticated ? <Route exact
      path="/mynotifications" render={(props) =>
        <Notifications {...props}/>}/>: null}
    {props.isAuthenticated ? <Route exact path="/myrides"
      render={(props) => <MyRides {...props}/>}/>: null}
    {props.isAuthenticated && checkIfOwner(pk, myrides) ?
      <Route exact path="/rides/:ridePK/edit"
        render={(props) => <EditRide {...props}/>}/>: null}

    <Route exact path="/rides" component={Rides}/>

    <Route exact path="/rides/:ridePK" component={Ride}/>
    <Route exact path="/faq" component={MyFaq}/>
    <Route exact path="/terms" component={TermsConditions}/>
    <Route exact path="/privacypolicy"
      component={PrivacyPolicy}/>
    {props.isAuthenticated ? <Route exact path="/ridesadd"
      component={addRide}/> : null}
  </Switch>
)

```

```

    {props.isAuthenticated ? <Route exact path='/requests'
      component={Requests}/> : null}
    <Route path='/' component={error404}/>
  </Switch>
);
};

```

7.2.4 Deployment

The deployment of our React web application is based on Heroku. Heroku provides its users a free tier to host web applications, including React-based ones. No complex configuration is needed to deploy this React Application to Heroku. It only needs an active heroku account and a valid git repository that hosts our React code. [23]

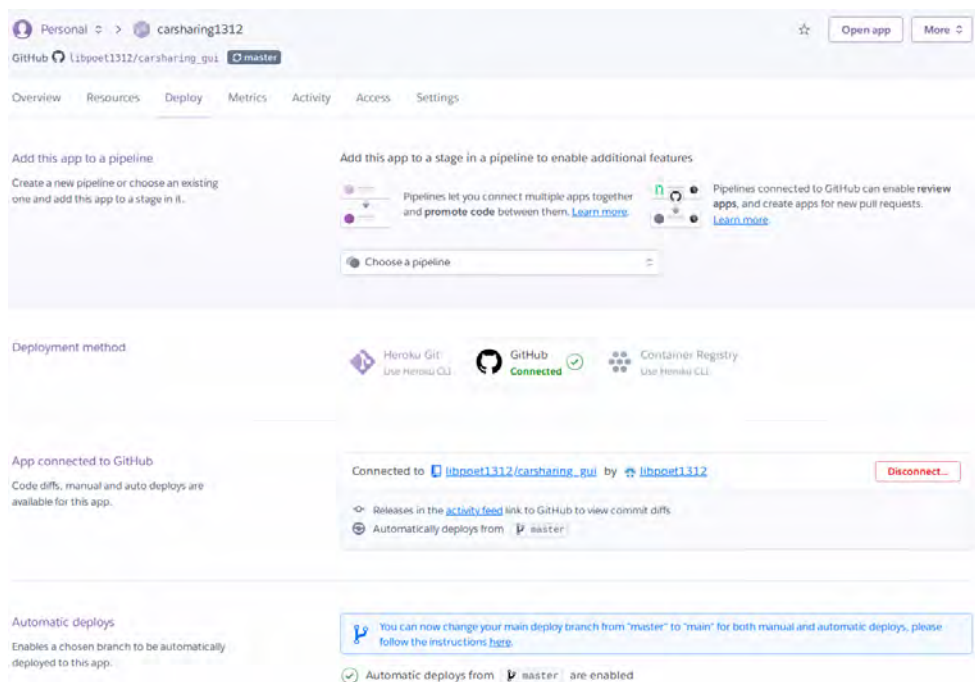


Figure 7.8: Heroku carsharing1312 App dashboard

7.3 Mobile Application Development

7.3.1 Mobile Development approach

Mobile development is a branch of application development that focuses on applications created solely for mobile devices. Although many web applications are mobile-friendly they are web sites, so they use a web browser in order for their content to be viewed. Over the last years different techniques and frameworks have risen in order to provide a mobile only development approach, that focuses and uses the mobiles' operation systems(Android, iOS) and their capabilities.

We can distinct two main approaches concerning the development on mobile applications.

Native approach

Native mobile development refers to the traditional way of developing mobile applications, using the mobile operation system architecture and programming language. For Android, the native programming languages used are *Java* or *C++* and *Kotlin*. For iOS Objective-C was the main developing language until 2014, until Apple introduced their programming language *Swift*.

Because native application development uses the mobile architecture and its language, it forces developers to learn and use the native languages and also the platforms and their architecture in their extend.

The biggest disadvantage with native mobile development is the separate codebases. When developing an application both in Android and iOS, a developer must have knowledge of their architecture and their programming languages. This increases the cost and complexity of development.

Cross-Platform approach

Cross-platform mobile development refers to the development of a mobile application which runs in different mobile platforms/multiple devices regardless of the system that it is run on. Cross-platform mobile applications can be divided in four major categories:

- **Hybrid Development:**

What makes hybrid applications different is that while a web application is viewed

via a web browser, the hybrid one is produced by a framework and is installed on the mobile device. They are developed with common web technologies (*HTML*, *CSS* and *JavaScript*) but they are wrapped into a container that acts like an installable mobile application. The most common hybrid development frameworks are *PhoneGap*, *Apache Cordova* and *Ionic*. Their main disadvantage is that they lack on the User Interface.

- **Cross-Compiled Native Development:**

As the name implies, cross-compiled applications are written in common programming languages and then compiled to native binaries of each operation system (Android or iOS). They can be very close to a fully native applications but they have their disadvantages. Most common frameworks are *Flutter* and *Xamarin*.

- **Native Scripting Development:**

Applications that were developed with native scripting utilize an interpreter to execute code during run time. They use a scripting language (mostly *JavaScript*) but utilize the native platform in its extend, so the result is really close to a native developed application.

7.3.2 React Native

React Native is a Native Scripting framework. React Native is very similar to *ReactJS*, but there are differences that distinct those two frameworks. They share the main ideas and conceptions. A main difference is that React-Native doesn't use pure *HTML* to render the app, but provides alternative components that work in a similar way. Those React-Native components map the actual real native iOS or Android *UI* components that get rendered on the app.

Most components provided can be translated to something similar in *HTML*, where for example a *View* component is similar to a *div* tag, and a *Text* component is similar to a *p* tag.

```
import React, { Component } from 'react';
import { View, Text } from 'react-native';

export default class App extends Component {
  render() {
```

```
    return (  
      <View style={styles.container}>  
        <Text style={styles.intro}>Hello world!</Text>  
      </View>  
    );  
  }  
}
```

Because your code doesn't get rendered in an *HTML* page, this also means you won't be able to reuse any libraries you previously used with React that renders any kind of *HTML*.

```
const styles = StyleSheet.create({  
  container: {  
    flex: 1,  
  },  
  content: {  
    backgroundColor: '#fff',  
    padding: 30,  
  },  
  button: {  
    alignSelf: 'center',  
    marginTop: 20,  
    width: 100,  
  },  
});
```

7.3.3 Expo

Expo is a framework built around React Native that provides a set of tools that simplify the development of a React Native application.

Through the development of the React-Native application, Expo is used because it provides:

- Fast and simple project installation.
- A command utility *Expo CLI* that opens in a browser and provides a graphical interface

of tools to configure and run the application.

- Expo client is a mobile app both for iOS and Android. It allows opening projects on the smartphone during development of the application without firstly building it via XCode or Android Studio.
- You can develop apps for ios without macOS with ios device and test them with Expo client.

7.4 Mobile application Implementation

7.4.1 Setup

Firstly we have to setup expo and its pre-requisites.

```
sudo apt install nodejs
npm install -g expo-cli
expo init PROJECTNAME #replace PROJECTNAME with your desired
                        project name
npm start
```

7.4.2 Mobile Application Structure

The main structure of the mobile application is:

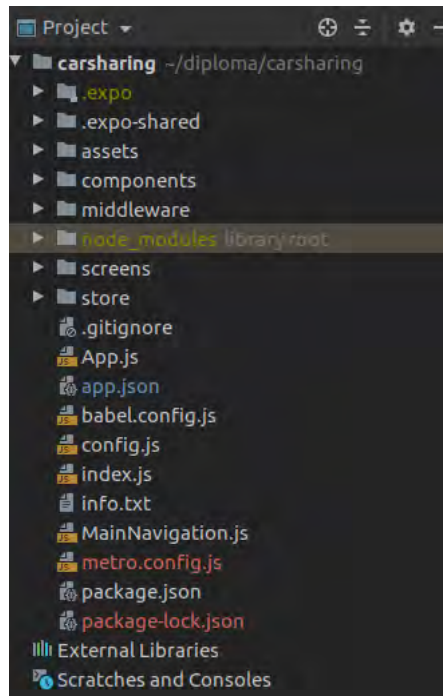


Figure 7.9: Mobile Application Structure

The most notable folders are:

- **screens**: This folder contains all components-screens that hold state, that the user navigates to.
- **components**: This folder contains all components that the user interacts through the screens.
- **assets**: Contains all static assets of the application. Favicon, images etc.

7.4.3 App.js

The main component of the mobile application. It initializes the application and checks if the *API* is online.

```
class App extends Component {
  state = {
    isReady: false,
    loading: true,
    online: false,
    error: null
  }
}
```

```
};

render() {
  if (!this.state.isReady) {
    return <AppLoading
      startAsync={this._cacheResourcesAsync}
      onFinish={()=>this.setState({isReady: true})}
      onError={console.warn}
    />;
  }

  return(
    <Provider store={store}>
      <StatusBar hidden/>
      <MainNavigation/>
    </Provider>
  )
}

async _cacheResourcesAsync() {

  await Font.loadAsync({
    Roboto: require('native-base/Fonts/Roboto.ttf'),
    Roboto_medium:
      require('native-base/Fonts/Roboto_medium.ttf'),
    ...Ionicons.font,
  });

  await axios.get('https://snf-876572.vm.okeanos.grnet.gr/')
    .then(() => {
      // console.log(res.status);
    }).catch( error => {
      console.log(error);
    });
}
```



```
        // return new Promise;  
    }  
}  
export default (App);
```

7.4.4 MainNavigation.js

This component is the second most important JavaScript file. It handles navigation, notifications and authenticates the user!

```
const RideStack = (props) => {  
  return (  
    <RideStackNav.Navigator  
      initialRouteName="Rides"  
      screenOptions={  
        {  
          headerRight: () => (<MyHeader  
            navigation={props.navigation}/>),  
        }  
      }  
    >  
    <RideStackNav.Screen name="Rides" component={Rides}  
      options={{  
    }}/>  
    <RideStackNav.Screen name="Ride" component={Ride}  
      options={{  
        headerTitle: '',  
        headerLeft: (props) => (  
          <HeaderBackButton  
            {...props}  
          />  
        )  
      }}  
    >
```

```

    }}/>
  </RideStackNav.Navigator>
)
};

const AuthStack = (props) => {
  return (
    <AuthStackNav.Navigator
      screenOptions={
        {headerRight: () => (<MyHeader
          navigation={props.navigation}/>)}
      }
    >
      <>
        {props.route.params.isAuthenticated ?
          <>
            <AuthStackNav.Screen name="MyProfile"
              component={MyProfile}/>
            <AuthStackNav.Screen name="Profile"
              component={Profile}/>
            <AuthStackNav.Screen name="Settings"
              component={Settings} />
            <AuthStackNav.Screen name="MyRides"
              component={MyRides}
              options={{headerTitle: 'My
                Rides' }}
            />
            <AuthStackNav.Screen name="AddRide"
              component={AddRide}
              options={{headerTitle: 'Add a
                ride' }}
            />
          </>
        }
      </>
    </AuthStackNav.Navigator>
  );
};

```

```
<AuthStackNav.Screen name="EditRide"
  component={EditRide}
  options={{headerTitle: 'Edit
              Ride' }}
/>
<AuthStackNav.Screen name="MyRequests"
  component={MyRequests}
  options={{headerTitle: 'My Requests' }}
/>
<AuthStackNav.Screen name="RequestsOfMyRides"
  component={RequestsOfMyRides}
  options={{headerTitle: 'Requests
              of my rides' }}
/>
<AuthStackNav.Screen name="Cars"
  component={Cars} />
<AuthStackNav.Screen name="AddCar"
  component={AddCar} options={{
  headerTitle: 'Add Car'
  }}/>
</>
:
<>
<AuthStackNav.Screen name="SignIn"
  component={Login}
  options={{
    title: 'Sign in',
    // When logging out, a pop
    // animation feels intuitive
    // You can remove this if you
    // want the default 'push'
    // animation
    animationTypeForReplace:
      !props.route.params.isAuthenticated
```

```

        ? 'pop' : 'push',
      }}/>
    <AuthStackNav.Screen name="SignUp"
      component={Signup} />

    { /*<AuthStackNav.Screen name="ResetPassword"
      component={ResetPassword} />*/ }

  </>
}
</>
</AuthStackNav.Navigator>
);
};

class MainNavigation extends Component {
  constructor(props) {
    super(props);
  }

  state = {
    notifications: [],
    unreadNotificationsCount: 0
  };

  componentDidMount() {
    this.props.onTryAutoSignup(); // auto-signin

    const newArray = this.props.notifications.filter( notif => {
      // console.log(notif);
      return notif.unread===true
    });
    this.setState({ unreadNotificationsCount: newArray.length});
  }
}

```

```
componentDidUpdate(prevProps, prevState, snapshot) {
  // when user logs in update notifications!
  if (prevProps.notifications !== this.props.notifications) {
    console.log('edw edw edw');

    const newArray = this.props.notifications.filter( notif
      => {
        // console.log(notif);
        return notif.unread===true
      });
    this.setState({ unreadNotificationsCount:
      newArray.length});
  }
}

render() {
  return (
    <Root>
      <NavigationContainer>
        <Tab.Navigator
          screenOptions={({ route }) => ({
            tabBarIcon: ({ focused, color, size }) => {
              let iconName;

              if (route.name === 'FAQ') {
                iconName = focused
                  ? 'ios-information-circle'
                  : 'ios-information-circle-outline';
              } else if (route.name === 'Rides') {
                iconName = focused ? 'ios-list-box' :
                  'ios-list';
              } else if (route.name === 'Home') {
                iconName = focused ? 'ios-home' :
                  'md-home';
              }
            }
          })
        />
      />
    />
  )
}
```

```

    }else if (route.name === 'Login'){
      if(this.props.isAuthenticated){
        iconName="ios-person"
      }else{
        iconName="ios-log-in"
      }
    }else if(route.name === 'MyProfile'){
      iconName="ios-person"
    }

    // You can return any component that you
    // like here!
    return <Ionicons name={iconName}
      size={size} color={color} />;
  },
  )))
  tabBarOptions={{
    activeTintColor: 'tomato',
    inactiveTintColor: 'gray',
  }}
>
<Tab.Screen name="Home" component={Home}/>
<Tab.Screen name="Rides" component={RideStack}/>
<Tab.Screen name="FAQ" component={FAQ}/>
<Tab.Screen name={!this.props.isAuthenticated ?
  "Login" : "MyProfile"} component={AuthStack}
  initialParams={{isAuthenticated:
    this.props.isAuthenticated}}
  options={this.state.unreadNotificationsCount
    ?
    { tabBarBadge:
      this.state.unreadNotificationsCount
    }
  : { tabBarBadge: null }

```

```
        }
      />
    </Tab.Navigator>
  </NavigationContainer>
</Root>
)
}
}

const mapStateToProps = state => {
  return {
    isAuthenticated: state.auth.user !== null,
    user: state.auth.user,
    notifications: state.auth.notifications
  };
};

const mapDispatchToProps = dispatch => {
  return {
    logout: () => dispatch(authActions.logout()),
    onTryAutoSignup: () => dispatch(authActions.authCheckState())
  }
};

// connect(mapStateToProps, mapDispatchToProps)(AuthStack);
export default connect(mapStateToProps,
  mapDispatchToProps)(MainNavigation);
```

7.5 Deployment

The mobile application is deployed through *Expo* framework. Registered users in expo framework have the ability to deploy their application in Expo cloud. It can be downloaded from Expo and later uploaded to Google Play or iOS store.

Chapter 8

Application Preview

8.1 Back-End

8.1.1 Landing Page

The back-end server due to the fact that works as a REST API that serves requests from the front-end clients, has no response methods that return user-friendly data, formatted as HTML. It only returns JSON formatted data.

For testing purposes, the `/'` path returns a landing HTML page that welcomes the user.

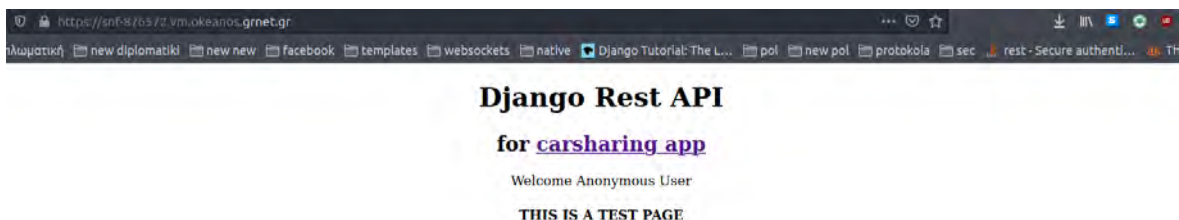


Figure 8.1: Back-End Landing Page [62]

8.1.2 Django Admin Page

Django framework provides a powerful admin interface. It lists all models and their instances and provide an easy management system of them. Only admin users have access to this interface.

The login page:

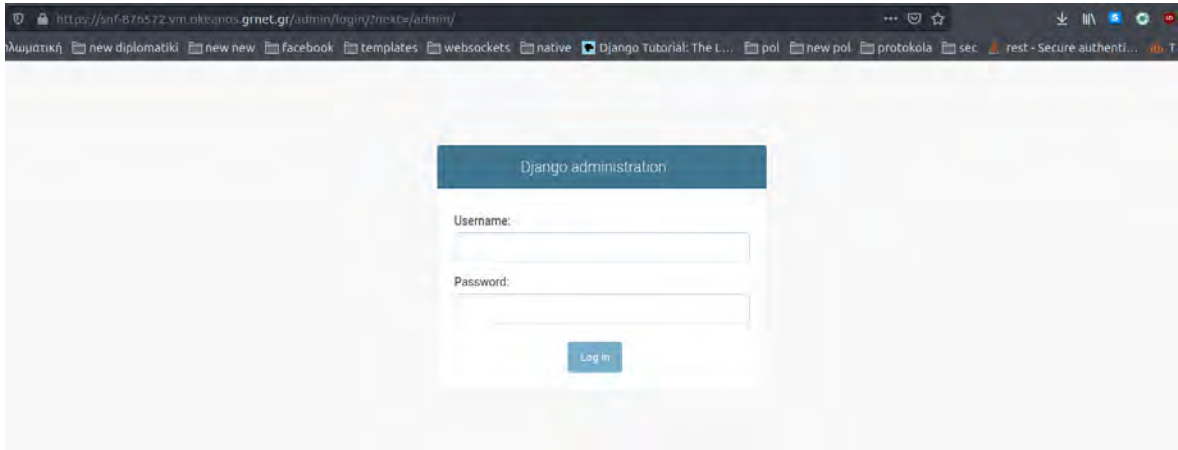


Figure 8.2: Django admin login page

The admin interface:



Figure 8.3: Django admin interface

8.1.3 REST API endpoints

Endpoints are divided to categories, in accordance with their functionality:

1. **Auth.** Endpoints used for users' authentication/authorization patterns.
 - i. **API_URL/rest-auth/registration/**. Register a new user.
 - ii. **API_URL/rest-auth/logout/**. Logout a user.
 - iii. **API_URL/rest-auth/login/**. Login a user.
 - iv. **API_URL/rest-auth/facebook/**. Login/Register a user via Facebook.
 - v. **API_URL/refresh-token/**. Refresh token.
 - vi. **API_URL/rest-auth/registration/verify-email/**. Verify email endpoint. Used only if email verification is on.
2. **Users.** Endpoints for getting users' data.
 - i. **API_URL/rest-auth/user/**. Retrieve/Update user information.
 - ii. **API_URL/rest-auth/logout/**. Logout a user.
 - iii. **API_URL/user/USER_PK**. Retrieve user information based on User's Primary Key(ID).
3. **Rides.** Endpoints used for rides patterns.
 - i. **API_URL/api/**. Get a list of rides offered.
 - ii. **API_URL/api/RIDE_PK**. Get details of a ride, based on its Primary Key.
 - iii. **API_URL/api/create/**. Create a ride.
 - iv. **API_URL/api/RIDE_PK/edit/**. Edit/Delete a ride.
 - v. **API_URL/api/myrides**. Get a list of my rides.
4. **Requests.** Endpoints for joining/unjoining a ride etc.
 - i. **API_URL/api/RIDE_PK/join**. Join a ride based on its Primary Key.
 - ii. **API_URL/api/RIDE_PK/getrequests**. Get join requests of a ride.
 - iii. **API_URL/api/getallrequests**. Get all join requests for a user rides.
 - iv. **API_URL/api/getmyrequests**. Get all join requests that a user has made.

- v. **API_URL/api/RIDE_PK/declinejoin/USER_PK/**. Decline a user's request to join a ride.
 - vi. **API_URL/api/RIDE_PK/acceptjoin/USER_PK/**. Accept a user's request to join a ride.
 - vii. **API_URL/api/RIDE_PK/unjoin**. Unjoin a ride in which the user has already joined.
5. **Notifications**. Endpoints for users' notifications.
- i. **API_URL/notifier/getall/**. Get user's notifications.
 - ii. **API_URL/notifier/mark-as-read/NOTIFICATION_PK**. Set notification as read, based on its Primary Key.
 - iii. **API_URL/notifier/mark-all-as-read/**. Mark all user's notifications as read.
6. **Cars**. Endpoints for users' cars.
- i. **API_URL/cars/car/**. Get user's list of cars, Add/Delete car.
 - ii. **API_URL/cars/car/CAR_PK**. Edit user car, based in its Primary Key.

Further explanation of the endpoints provided, their methods, their arguments needed and their description, will be available in the API documentation.

8.2 Front-End

8.2.1 Web Application

The home page is the first page a user sees when fires up CarSharing App. As stated in the project description users can browse rides, see their details, login/signup via Facebook or via their provided credentials, add or join existing offered rides.

A non-authenticated user can browse the following pages:

- **Home Page.**

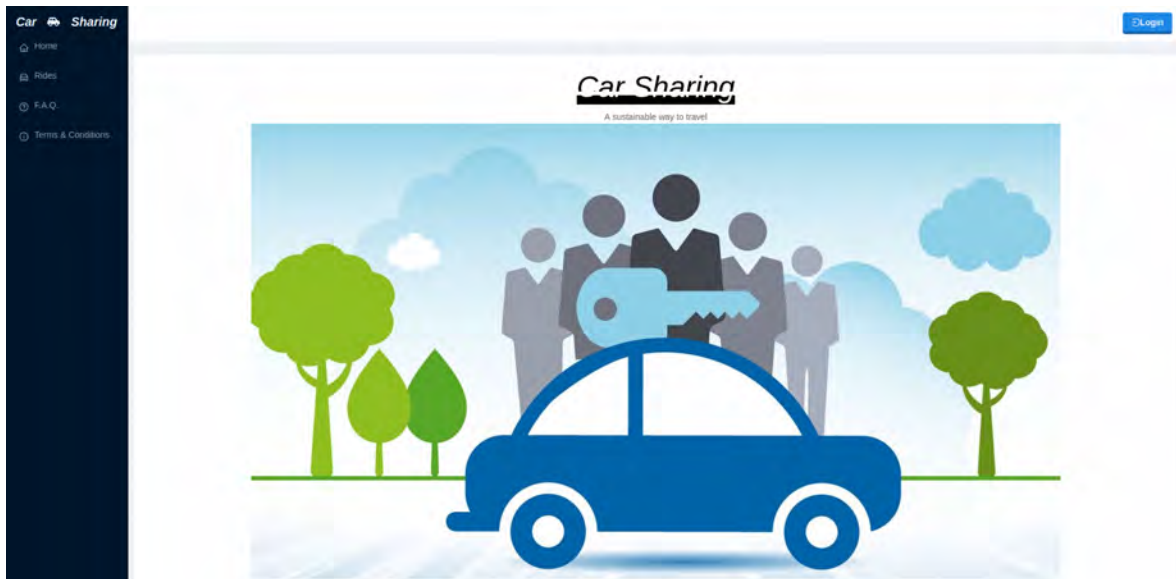


Figure 8.4: Home Page

- Rides.

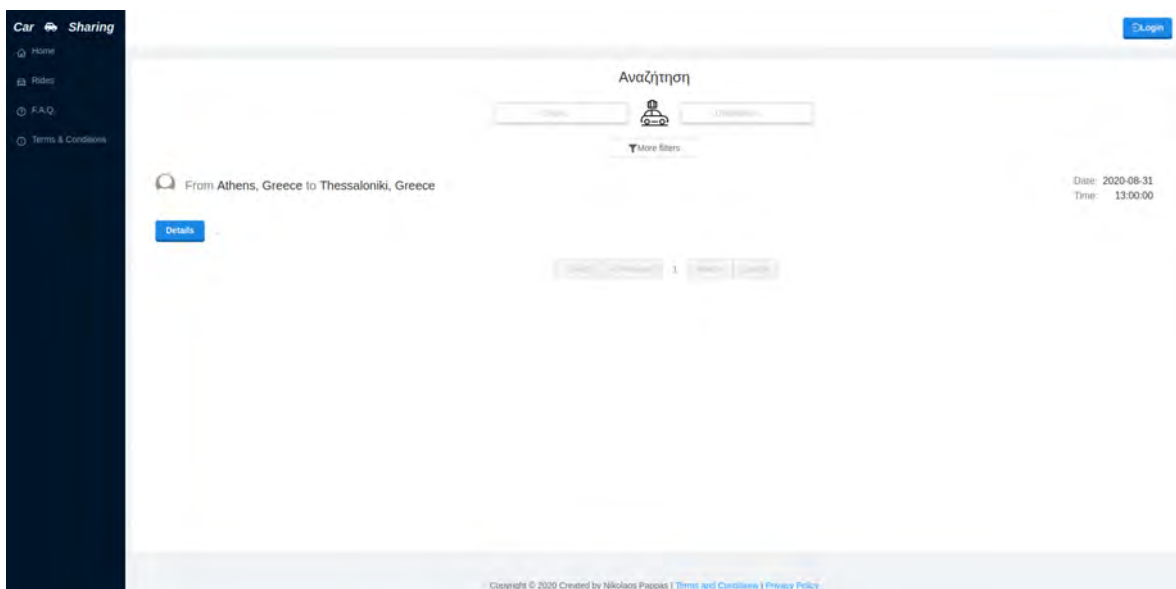


Figure 8.5: Rides

- Ride.

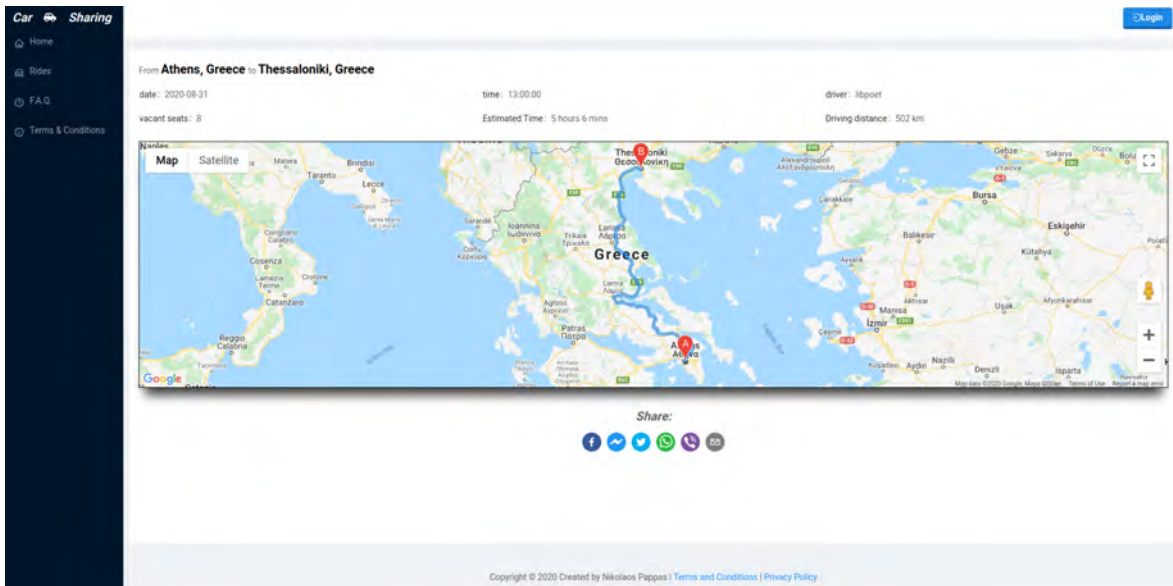


Figure 8.6: Ride

- F.A.Q.

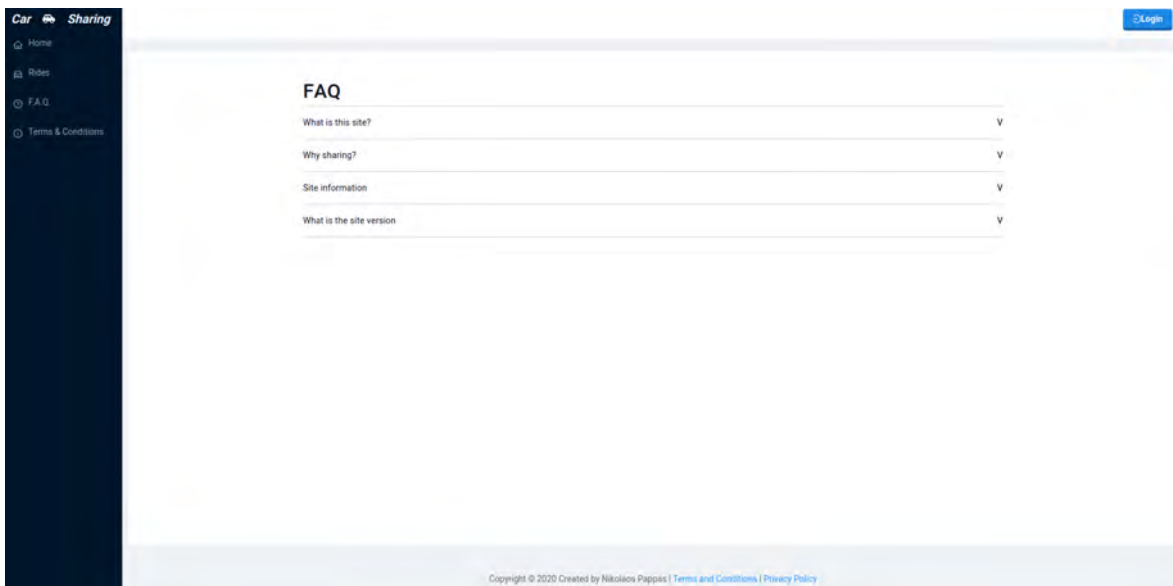


Figure 8.7: FAQ

- Terms and Conditions.

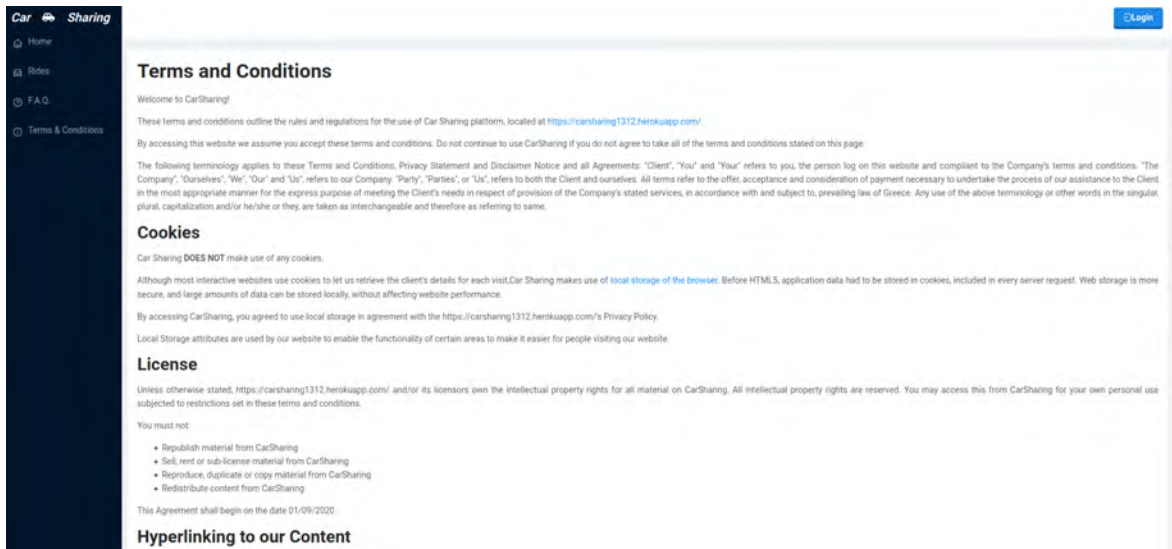


Figure 8.8: Terms and Conditions

- **Privacy Policy.**

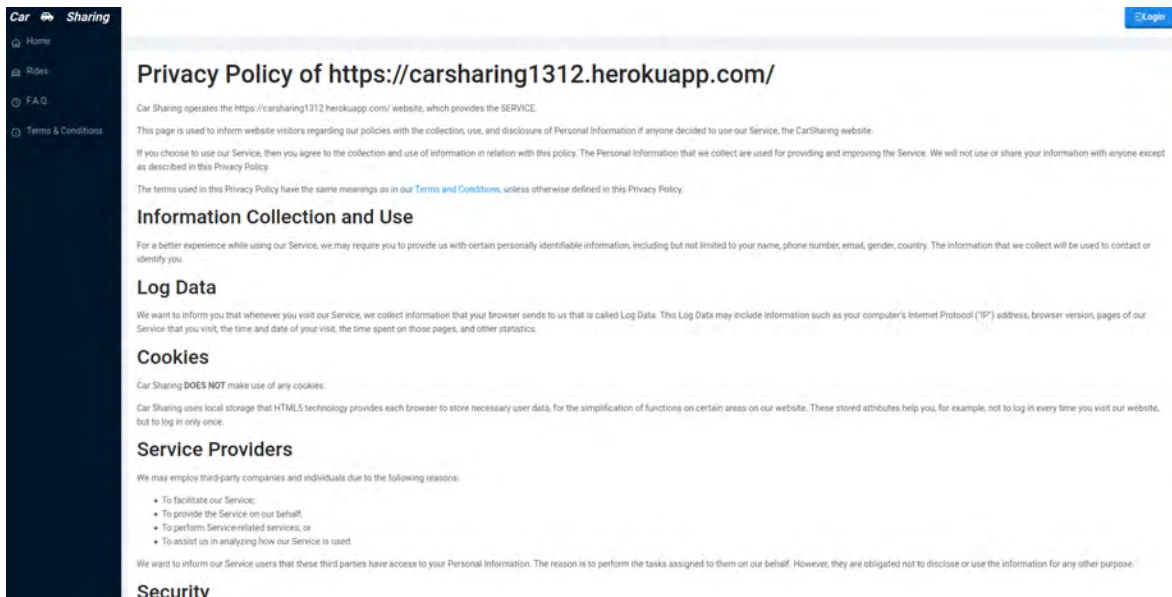
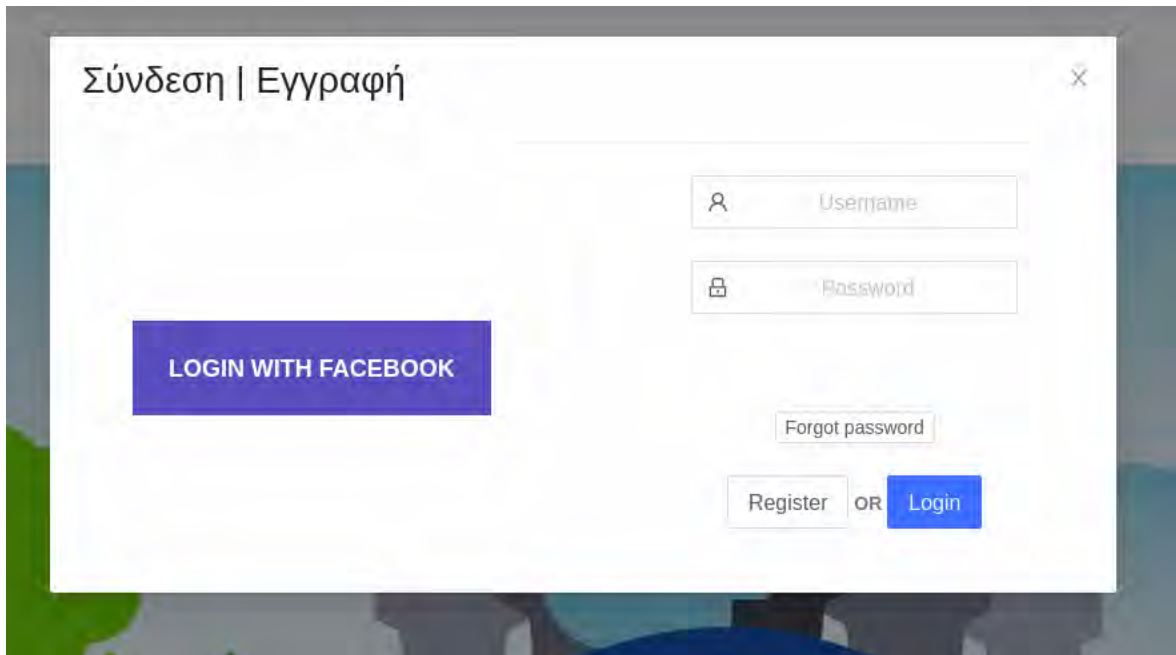


Figure 8.9: Privacy Policy

A user can login or register via the below forms:



Σύνδεση | Εγγραφή

Username

Password

LOGIN WITH FACEBOOK


Forgot password


Register OR Login


Figure 8.10: Login form

Σύνδεση | Εγγραφή

* E-mail:

* Username :



* Password: 

* Confirm Password: 

* Date of birth:



Gender:

Αριθμός Τηλεφώνου:

Country:  IN 

Avatar:

Επιπλέον μέσα επικοινωνίας

 What's up  Viber

Συμφωνώ με τους [όρους και προϋποθέσεις χρήσης](#) της υπηρεσίας.

Figure 8.11: Register form

After Log In or Registration the header and the drawer menu change and they provide links to other site functions and services.




Figure 8.12: After Login/Registration


The authenticated user can perform the following tasks:


- **Add a Ride.**

Σύνδεση | Εγγραφή

* E-mail:

* Username :



* Password: 

* Confirm Password: 

* Date of birth:



Gender:

Αριθμός Τηλεφώνου:

Country:  IN 

Avatar:

Επιπλέον μέσα επικοινωνίας

 What's up  Viber

Συμφωνώ με τους [όρους και προϋποθέσεις χρήσης](#) της υπηρεσίας.

[Πίσω στην Σύνδεση](#) [Καθαρισμός Πεδίων](#) [Εγγραφή](#)

Figure 8.13: Add Ride

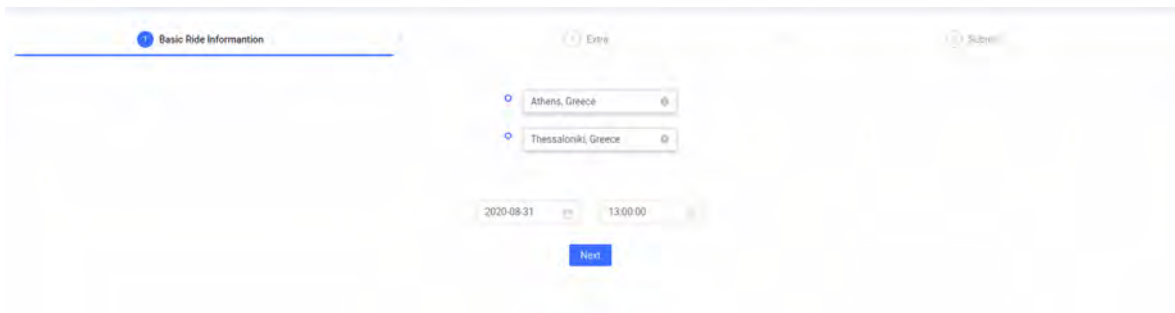
- **Add a Ride.**



The screenshot shows a web form titled "Basic Ride Information". The form is enclosed in a red rectangular border. It contains several input fields: a "City" dropdown menu, an "Expenses" text input, and two "Amount" text inputs. There are also "Cancel" and "Next" buttons at the bottom of the form.

Figure 8.14: Add Ride

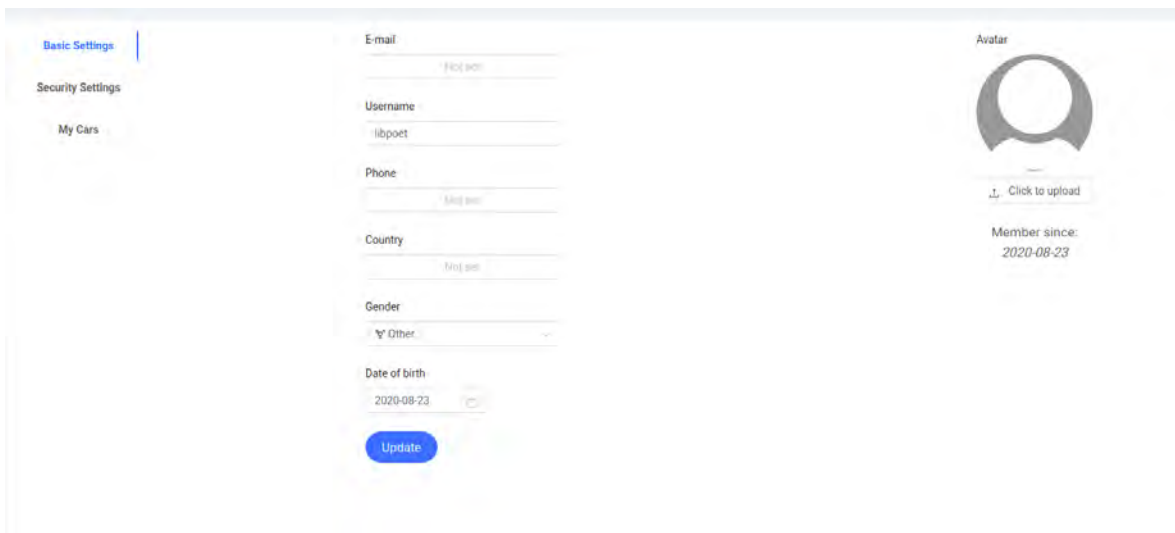
- **Edit a Ride.**



The screenshot shows a web form titled "Basic Ride Information" for editing a ride. It features dropdown menus for "City" (with "Athens, Greece" selected) and "Location" (with "Thessaloniki, Greece" selected). There are also input fields for "Start Date" (2020-08-31) and "End Date" (13:00:00). A blue "Next" button is positioned below the date fields.

Figure 8.15: Edit Ride

- **Edit his information through Settings.**



The screenshot displays the "Basic Settings" page. On the left, there are navigation tabs for "Basic Settings", "Security Settings", and "My Cars". The main content area contains a form with the following fields: "E-mail" (with a "No id set" message), "Username" (libpoet), "Phone" (with a "No id set" message), "Country" (with a "No id set" message), "Gender" (set to "Other"), and "Date of birth" (2020-08-23). An "Update" button is located at the bottom of the form. On the right side, there is an "Avatar" section with a circular profile picture placeholder, a "Click to upload" button, and the text "Member since: 2020-08-23".

Figure 8.16: Settings

- **List his Rides.**

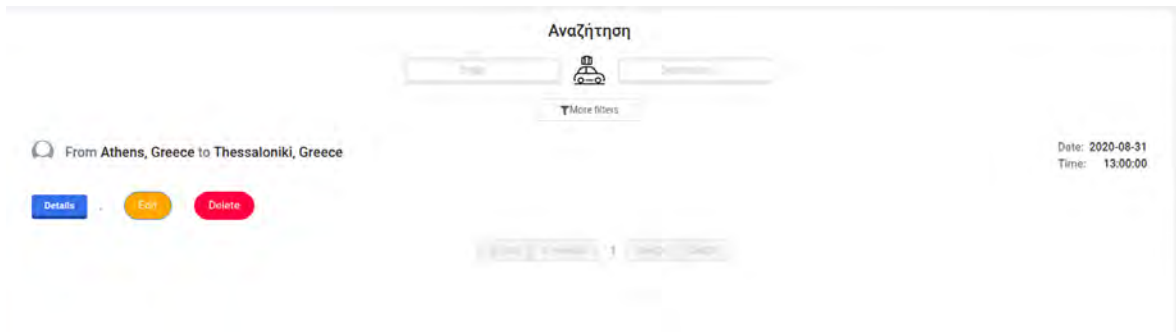


Figure 8.17: My rides

- See the requests his has made or the requests made to his rides.

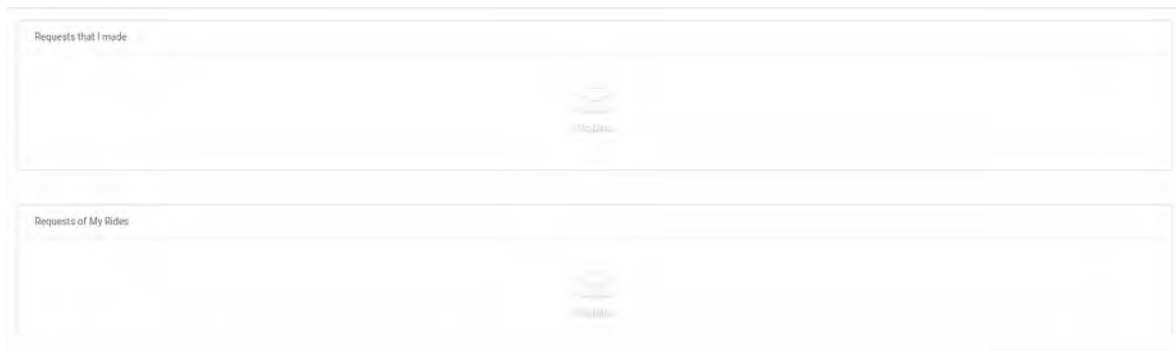


Figure 8.18: Requests

- Edit his information through Settings.

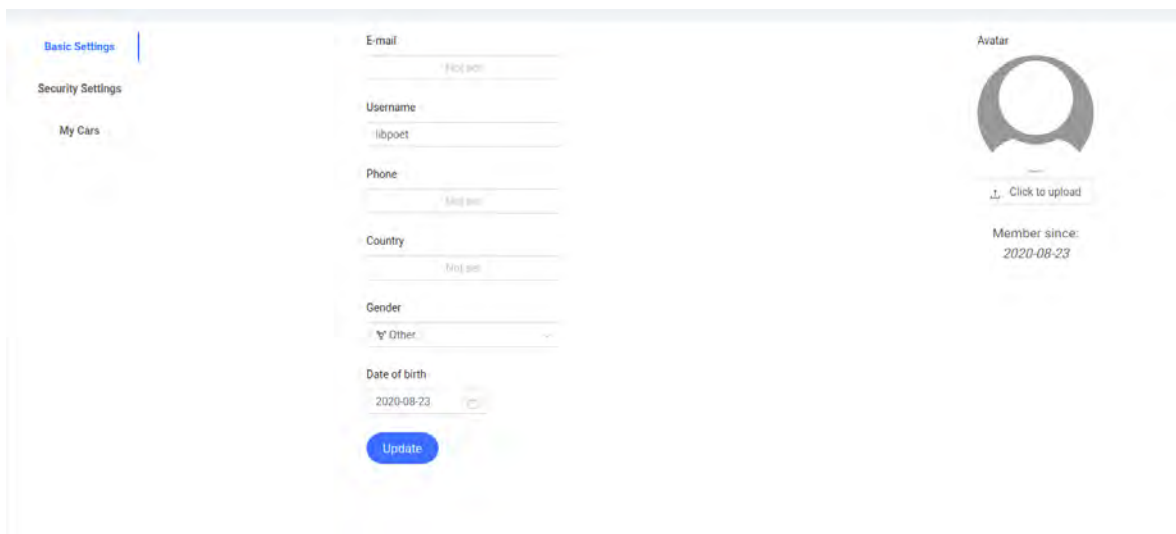


Figure 8.19: Settings

8.2.2 Mobile Application

A non-authenticated user can browse the following screens:

- **Home Screen**

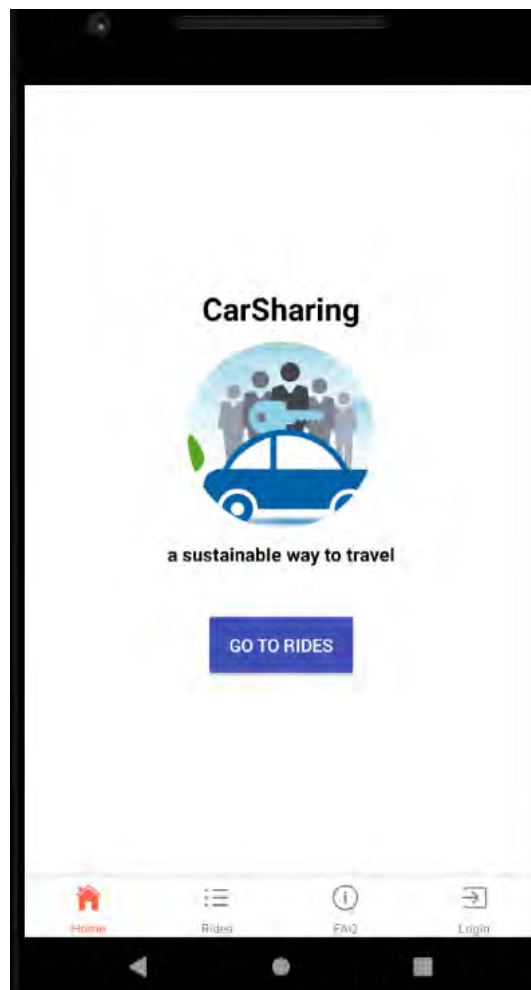


Figure 8.20: Home Screen

- **Rides Screen**

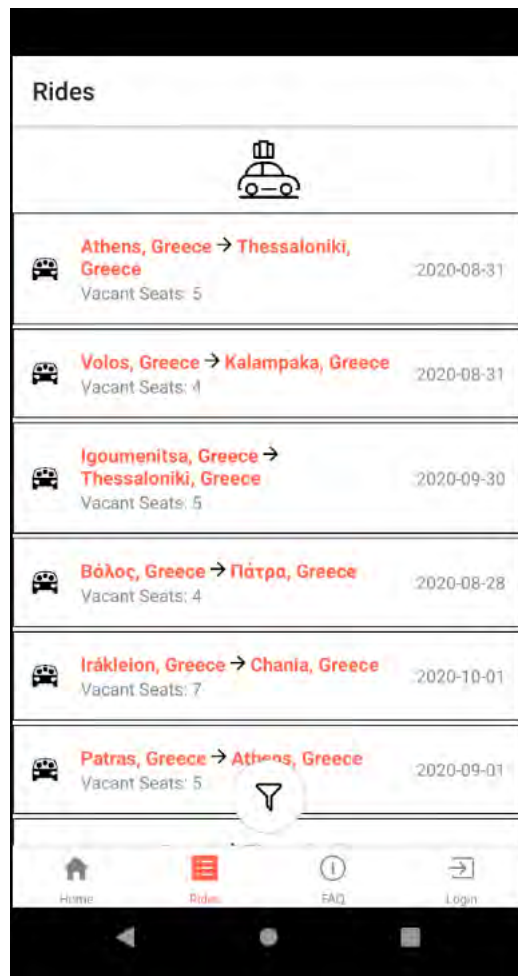


Figure 8.21: Rides Screen

- Ride Filters

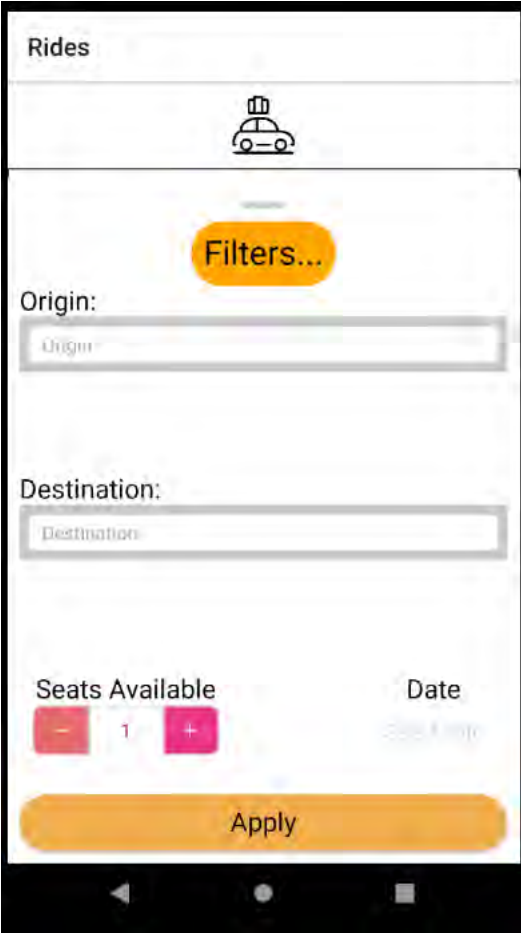


Figure 8.22: Ride Filters

- FAQ Screen



Figure 8.23: FAQ Screen

- Login Screen

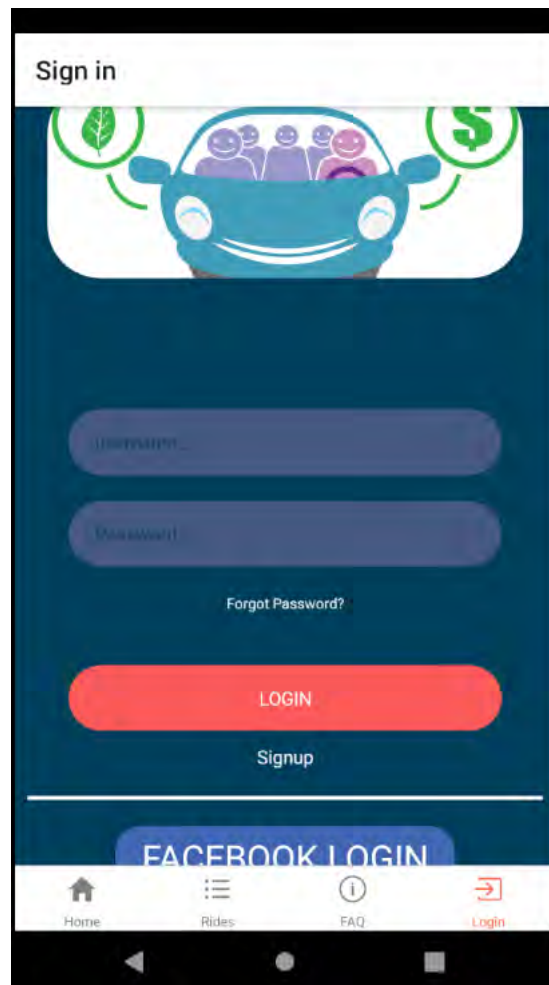
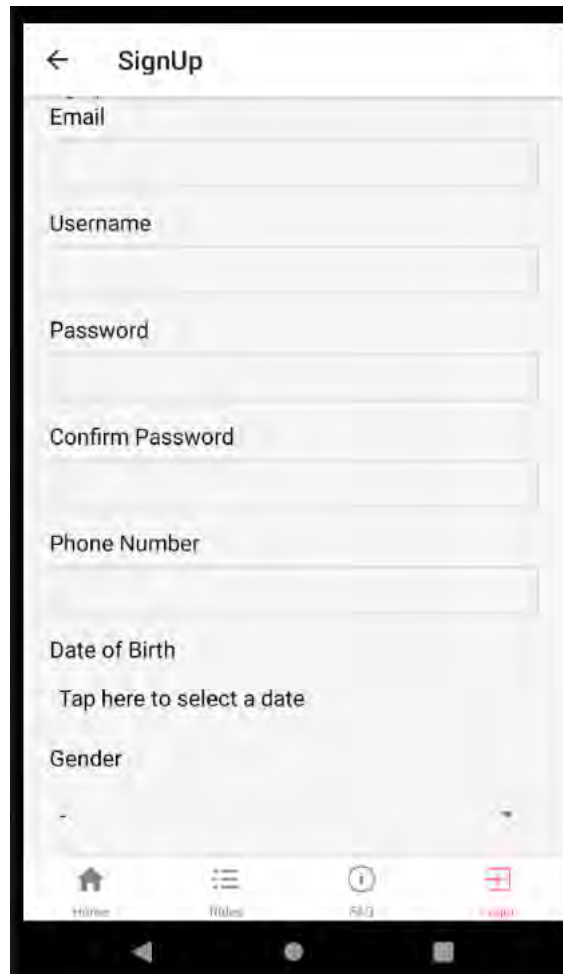


Figure 8.24: Login Screen

- Sing up Screen



The image shows a mobile application's 'SignUp' screen. At the top, there is a back arrow and the text 'SignUp'. Below this, there are several input fields: 'Email', 'Username', 'Password', 'Confirm Password', 'Phone Number', 'Date of Birth' (with a prompt 'Tap here to select a date'), and 'Gender'. At the bottom, there is a navigation bar with icons for 'Home', 'Rides', 'Profile', and 'Logout'.

Figure 8.25: Sign up Screen

After login/sign-up, a user can navigate through multiple screens that allow authenticated actions (add/join rides, edit profile etc) :

- **Join Ride**

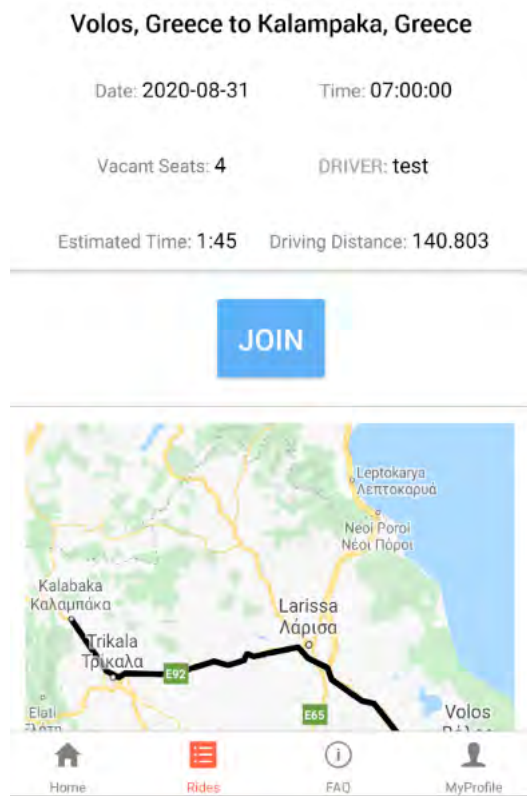


Figure 8.26: Join Ride

- **MyProfile**

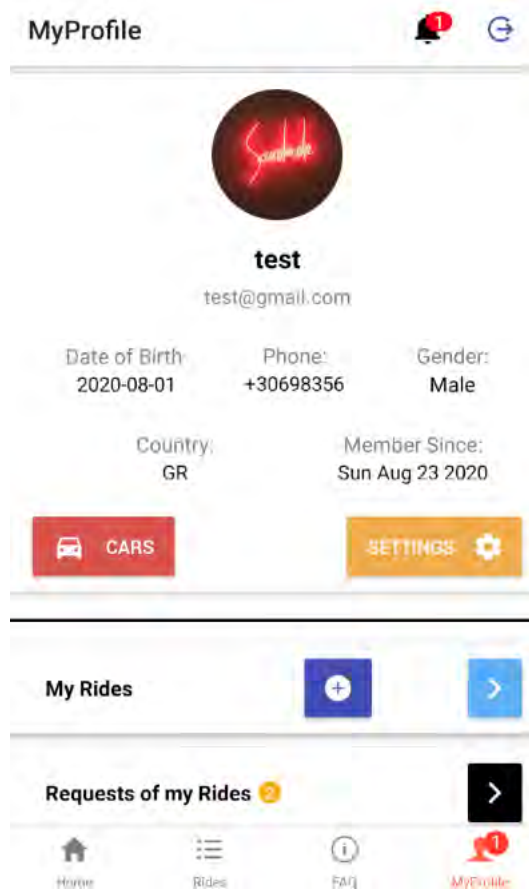


Figure 8.27: My profile

- **Notifications**

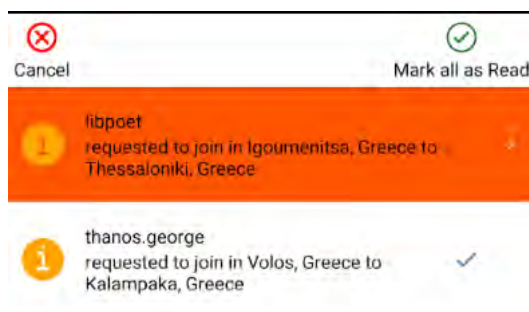


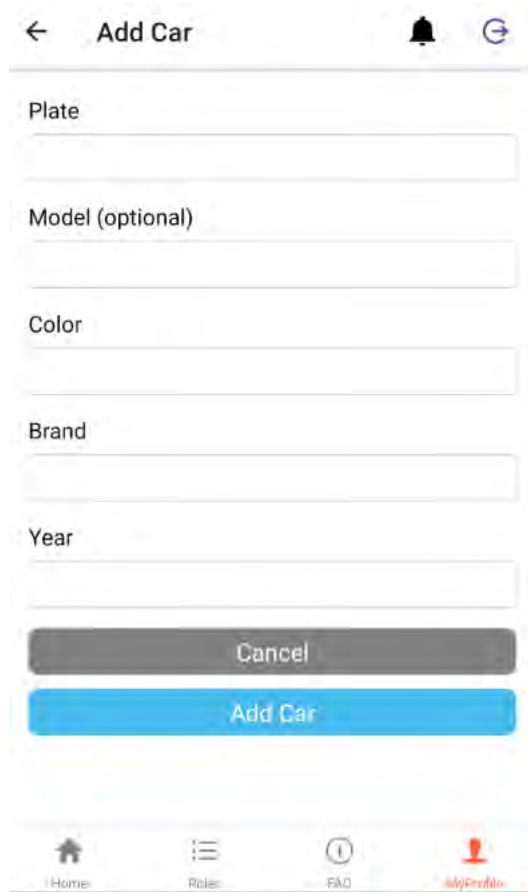
Figure 8.28: Notifications

- **Cars**



Figure 8.29: Cars

- **Add Car**



The screenshot shows a mobile application interface for adding a car. At the top, there is a navigation bar with a back arrow, the title "Add Car", a notification bell icon, and a refresh icon. Below the navigation bar, there are five text input fields labeled "Plate", "Model (optional)", "Color", "Brand", and "Year". At the bottom of the form, there are two buttons: a grey "Cancel" button and a blue "Add Car" button. The bottom of the screen features a navigation bar with four icons: a home icon labeled "Home", a list icon labeled "Role", a clock icon labeled "FAQ", and a profile icon labeled "My Profile".

Figure 8.30: Add Car

- **Delete Car**

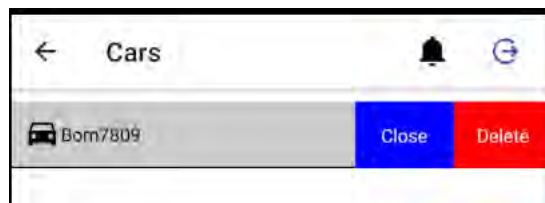
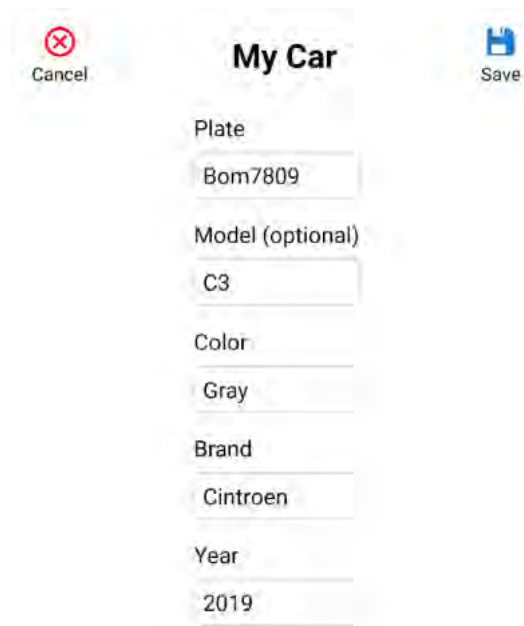


Figure 8.31: Delete Car

- **Edit Car**



The image shows a web form titled "My Car" for editing car details. The form is centered and has a light gray background. On the left side, there is a "Cancel" button with a red 'X' icon. On the right side, there is a "Save" button with a blue floppy disk icon. The form contains five input fields, each with a label above it:

- Plate:** Bom7809
- Model (optional):** C3
- Color:** Gray
- Brand:** Cintroen
- Year:** 2019

Figure 8.32: Edit Car

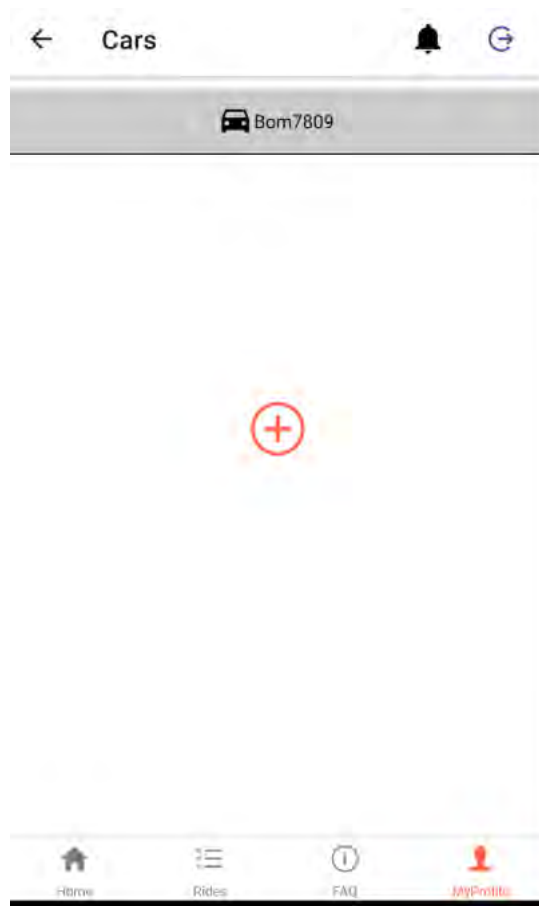


Figure 8.33: Cars

- **Edit profile**

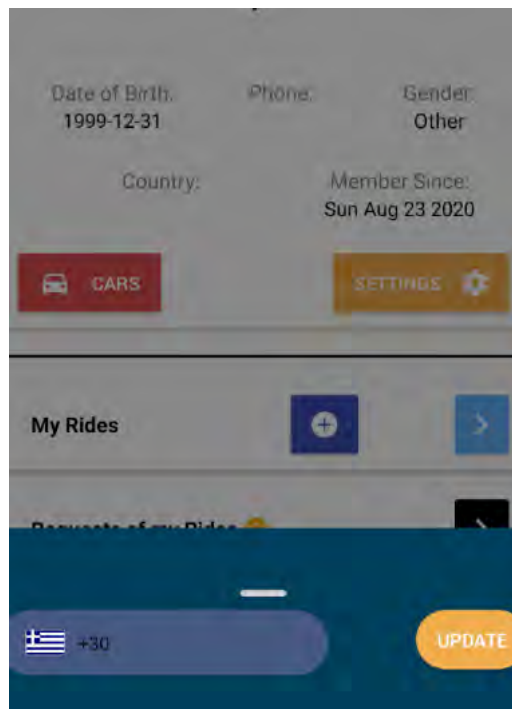


Figure 8.34: Edit Profile

- **My Rides**

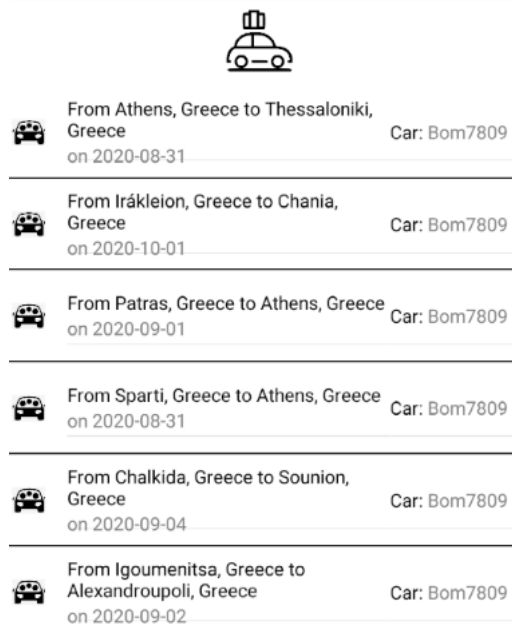


Figure 8.35: My Rides

- **User's requests**

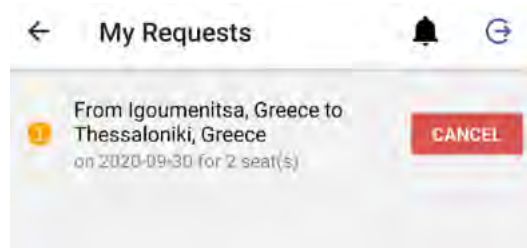


Figure 8.36: user's Requests

- **Add Ride**

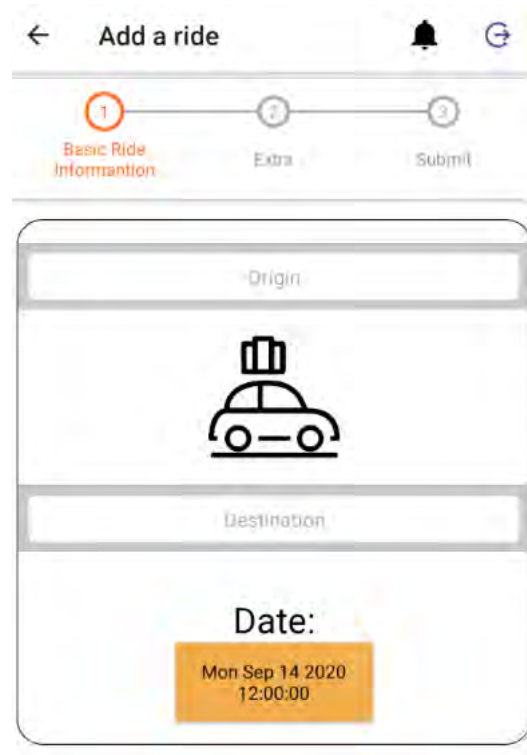


Figure 8.37: Add Ride

Chapter 9

Production sites and code

Carpooling application is fully deployed and is in production state.

- The Web Application is hosted on heroku at: <https://carsharing1312.herokuapp.com>.
- The Android app is hosted on my personal Expo space: <https://expo.io/@npappas/carsharing>.
- The back-end is hosted on: <https://snf-876572.vm.oceanos.grnet.gr/>.

The full code of all front-end and Back-end is hosted on repositories on my github account.

Specifically:

- Back-End
- Web Application
- Mobile Application

Chapter 10

Future development

Many services in the platform can be extended and others can be added to expand and improve its functionality.

Some are:

- A Chat service. A mechanism that allows users to exchange messages for their rides.
- Way-points. A ride, except for its origin and destination can contain also some way-points. With that extension, people can join rides and be dropped off in places between the origin or the destination.
- Other ride properties. A ride offerer could be able to define if a ride allows pets, chatting amongst the riders, smoking in the car etc.
- Ride ratings. A ride, after being shared, could be rated, thus the community could choose to ride or not with the rider based on its rates.
- Price. When implementing such platforms, their development has costs.

Both the web and mobile applications could be more User Friendly. More styling could be implemented to both the clients for them to be more attractive to the user.

Bibliography

- [1] *An Introduction to Redux*. <https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/>.
- [2] *Ant Design*. <https://ant.design/>.
- [3] *AsyncStorage*. <https://github.com/react-native-community/async-storage>.
- [4] *Axios*. <https://github.com/axios/axios>.
- [5] Wikimedia Commons contributors. *Client-Server N-tier architecture - en.png*. https://commons.wikimedia.org/w/index.php?title=File:Client-Server_N-tier_architecture_-_en.png&oldid=403628289.
- [6] Wikipedia contributors. *Client-server model*. https://en.wikipedia.org/w/index.php?title=Client%E2%80%93server_model&oldid=969629934.
- [7] *CORS*. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>.
- [8] *Daphne server*. <https://github.com/django/daphne/>.
- [9] *Django Channels*. <https://channels.readthedocs.io/en/latest/>.
- [10] *Django Life Cycle*. <https://medium.com/@ksarthak4ever/django-request-response-cycle-2626e9e8606e>.
- [11] *Django Life Cycle Diagram*. https://learnbatta.com/assets/images/django/request_response_lifecycle_Django.png.
- [12] *Django project*. <https://www.djangoproject.com/>.

- [13] *Django Project Structure*. <https://studygyaan.com/django/best-practice-to-structure-django-project-directories-and-files>.
- [14] *Django REST framework*. <https://www.django-rest-framework.org/>.
- [15] *Django Settings*. <https://docs.djangoproject.com/en/3.1/topics/settings/>.
- [16] *django-allauth*. <https://django-allauth.readthedocs.io/en/latest/index.html>.
- [17] *Django-cors-headers*. <https://pypi.org/project/django-cors-headers/>.
- [18] *django-notifications-hq*. <https://pypi.org/project/django-notifications-hq/>.
- [19] *django-rest-auth*. <https://django-rest-auth.readthedocs.io/en/latest/>.
- [20] *Expo*. <https://expo.io>.
- [21] *geeksforgeeks.org. MVC architecture*. <https://www.geeksforgeeks.org/mvc-design-pattern/>.
- [22] *Gunicorn*. <https://gunicorn.org/>.
- [23] *How to deploy a React + Node app to Heroku in 3 minutes without the command line*. <https://www.freecodecamp.org/news/deploy-a-react-node-app-to/>.
- [24] *JavaScript*. <https://django-allauth.readthedocs.io/en/latest/index.html>.
- [25] *JSON Web Tokens*. <https://jwt.io/introduction/>.
- [26] *MomentJS*. <https://momentjs.com/>.
- [27] *mySQL*. <https://www.mysql.com/>.
- [28] *Mysqclient*. <https://pypi.org/project/mysqclient/>.
- [29] *NativeBase*. <https://nativebase.io/>.
- [30] *NGINX*. <https://nginx.org/en/>.

-
- [31] *NGINX Percentage of busiest sites*. <https://news.netcraft.com/archives/2020/07/27/july-2020-web-server-survey.html>.
 - [32] Tutorials Point. *HTTP Methods*. https://www.tutorialspoint.com/http/http_methods.htm.
 - [33] Tutorials Point. *HTTP status codes*. https://www.tutorialspoint.com/http/http_status_codes.htm.
 - [34] *PostMan*. <https://www.postman.com/>.
 - [35] *Pycharm IDE*. <https://www.jetbrains.com/pycharm/>.
 - [36] *Python*. <https://www.python.org/>.
 - [37] The Hitchhiker's Guide to Python. *Python Virtual Environment*. <https://python-guide-kr.readthedocs.io/ko/latest/dev/virtualenvs.html>.
 - [38] *React*. <https://reactjs.org/>.
 - [39] *React <AwesomeButton /> UI Component*. <https://github.com/rcaferati/react-awesome-buttons>.
 - [40] *React Dev tools*. <https://www.npmjs.com/package/react-devtools>.
 - [41] *React Example*. <https://www.codecademy.com/articles/how-to-create-a-react-app>.
 - [42] *React Facebook Login*. <https://github.com/keppelen/react-facebook-login>.
 - [43] *React Faq Component*. <https://binodswain.github.io/react-faq-component/>.
 - [44] *React Getting Started*. <https://reactjs.org/docs/getting-started.html>.
 - [45] *React Google Maps*. <https://github.com/tomchentw/react-google-maps>.
 - [46] *React JS tutorial*. <https://www.dotnetcurry.com/reactjs/1353/react-js-tutorial>.
 - [47] *React Life Cycle Methods - Diagram*. <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>.

-
- [48] *React Lifecycle Methods - A Deep Dive*. <https://programmingwithmosh.com/javascript/react-lifecycle-methods/>.
 - [49] *React Native*. <https://reactnative.dev/>.
 - [50] *React Native Elements*. <https://react-native-elements.github.io/react-native-elements/>.
 - [51] *React Native Maps*. <https://github.com/react-native-community/react-native-maps>.
 - [52] *React Native Maps Directions*. <https://github.com/bramus/react-native-maps-directions>.
 - [53] *React Navigation*. <https://reactnavigation.org/>.
 - [54] *React Places Autocomplete*. <https://github.com/hibiken/react-places-autocomplete>.
 - [55] *React Router*. <https://reactrouter.com/>.
 - [56] *ReactJS | State in React*. <https://www.geeksforgeeks.org/reactjs-state-react/>.
 - [57] *Redis*. <https://redis.io/>.
 - [58] *Redux*. <https://redux.js.org/>.
 - [59] *Redux DevTools*. <https://github.com/reduxjs/redux-devtools>.
 - [60] *Redux Thunk*. <https://github.com/reduxjs/redux-thunk>.
 - [61] *Sap.com. MVC architecture figure*. <https://blogs.sap.com/2017/04/06/ui5-architectural-pattern-mvc-mvvm-or-mvwhatever/>.
 - [62] *State Example*. <https://medium.com/hootsuite-engineering/everything-you-need-to-know-about-setstate-8233a7042677>.
 - [63] *SuperVisor*. <http://supervisord.org/>.
 - [64] *The difference between Virtual DOM and DOM*. <https://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/>.
 - [65] *Top 8 Best Backend Frameworks*. <https://www.keycdn.com/blog/best-backend-frameworks>.

-
- [66] *Understanding The Request-Response Lifecycle In Django*. <https://learnbatta.com/blog/understanding-request-response-lifecycle-in-django-29/>.
- [67] *WebStorm*. <https://www.jetbrains.com/webstorm/>.
- [68] *Whatis the Virtual DOM*. <https://mfrachet.github.io/create-front-end-framework/vdom/intro.html>.