PowerEnJoy, Design Document

Riccardo Cattaneo 873647
Fabio Chiusano 874294

Version: 1.0.0
Release date: 09-12-2016

Table of Contents

1 Introduction

1.1 Purpose

This document describes the hardware and software architecture of the PowerEnjoy System. Therefore, it outlines hardware tiers and all the components of the software and how they will work and cooperate together.
In particular, this document contains information about:

- Architecture Design with related technologies used;
- Main software components and interfaces description;
- Runtime behaviour of the system;
- Algorithm Design;
- User Interface Design.

1.2 Scope

PowerEnjoy is a car-sharing service based on a mobile web application and a web site.
It allows the user to see, thanks to the help of an external search-on-map handler, where the electric cars are, only if they are close to an address provided by either the user or his/her GPS Location. Hence, it allows users to reserve an electric car and to get on board when he/she is close to it.
The car software takes into account the minutes of usage of the car, the number of passengers, the battery level and the location of release.
The system then calculates the cost of the ride that user has to pay.
The main purpose of the system is to create a new and smart car-sharing service, which incentivizes virtuous and green behaviours.

1.3 Definitions, Acronyms, Abbreviations

- Reservation Code: is a code, displayed on the reserved car, that is needed by the user to unlock it;
- DD: Design document;
- ER: Entity-Relationship diagram;
- RASD: Requirements Analysis and Specification Document;
- JSE: Java Serial Edition;
- JEB: Java Enterprise Bean;
- REST: REpresantional State Transfer;
- RESTful service: REST compliant service;
- UX: User eXperience;
- JEE: Java Enterprise Edition;
- JAX-RS: Java API for RESTful web Services;
- JBOSS: JEE open source application server;

- JPA: Java Persistence API;
- Cordova: mobile cross-platform development framework;
- PhoneGap: mobile cross-platform development framework that works over Cordova;
- JSP: Java Server Pages.

1.4 Reference Documents

- Specification Document: Assignments AA 2016-2017.pdf;
- TheraWii Software Design Document;
- Example Design Documents from previous years:
  - MyTaxiService;
  - SWIMv2.

1.5 Document Structure

This document specifies the architecture of PowerEnJoy spreading from the general into the specific. In addition, it describes the architectural decisions and tradeoffs and justifies them. The design was guided by a top-down process approach and the document structure reflects this tactic.

The document is organized as follows:

- Introduction: provides a synopsis of the architectural descriptions;
- Architectural design: provides a general description of PowerEnJoy including its functionality and matters related to the overall system and its design;
- Algorithmic design;
- User Interface design;
- Requirements traceability;
- Effort spent and tools used.

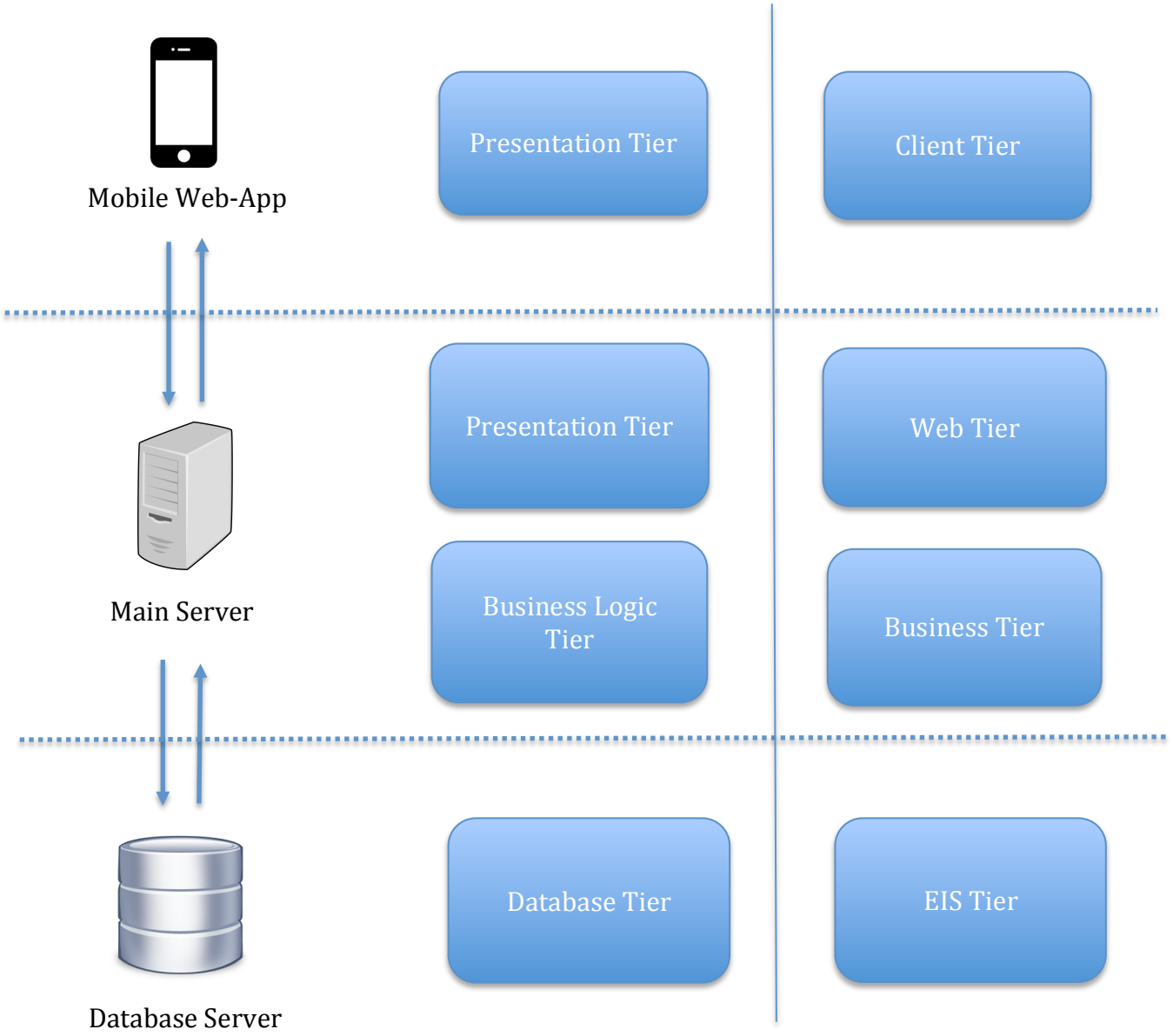
2 Architectural Design

2.1 Overview

We propose to make, at first, a web app that will give users a comfortable way to use our service.
There will be also a responsive web site for users who do not want to download an app on their smartphones.

The architecture of PowerEnjoy Service has three tiers:

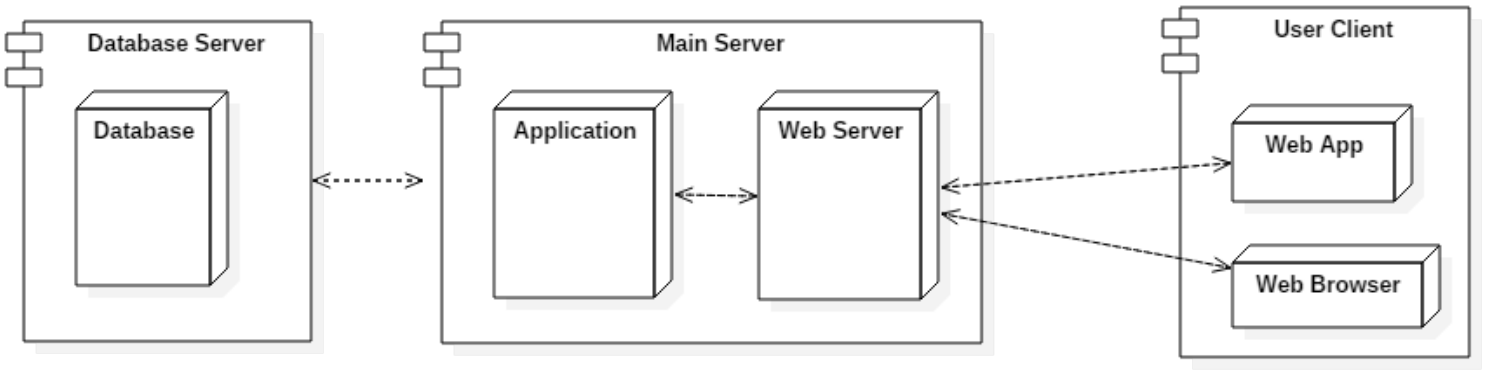
- The presentation will be provided to the user thanks to the cooperation between the client and the Web Tier of the Server;
- The Logic will be provided by the Main Server. Cars will have some internal logic too but they will work for the Main Server;
- The Persistence functionalities will be provided by a Database Server, which communicate directly to the Main Server.



As concerns the web site, the browser of course receives the front-end from the server and therefore there is no presentation tier located at the client.

2.2 High Level Components and their interactions

2.2.1 Overview

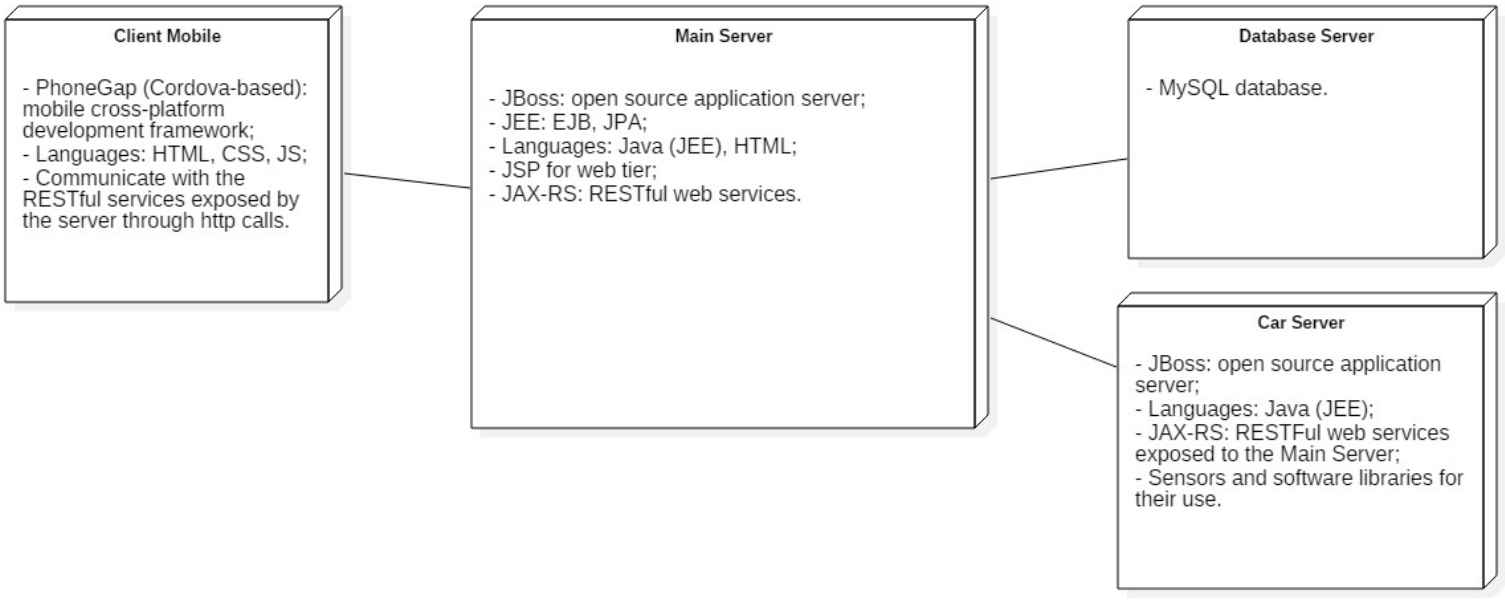


- Client Tier, represented by the web app, that will be available to all the major mobile operating systems and will be developed in a way to communicate with the PowerEnjoy servers through RESTful API and HTTP request;
- On the server side, the Main Server contains two software tiers:
  - Web Tier, that manages the communication with the web app. It receives requests from the app and provides it and updated front-end, thanks to Java Server Pages (JSP);
  - Business Logic Tier, that elaborates requests from client thanks to the embedded logic and manages interactions with Database, thanks to the Java Persistence API (JPA);
- Database Tier, that contains and manages persistent data in an efficient way.

Of course, the electric cars must be able to communicate with the server, so they must be provided with an Internet connection and an on-board computer that must be able to run Java software. Cars will have small logic, mainly dedicated to communicate sensors data with Logic Tier of the Main Server.

The Main Server will also take advantage of external handler, such as Payment Handler and Search-on-map Handler, in order to manage successfully functionalities provided to users.

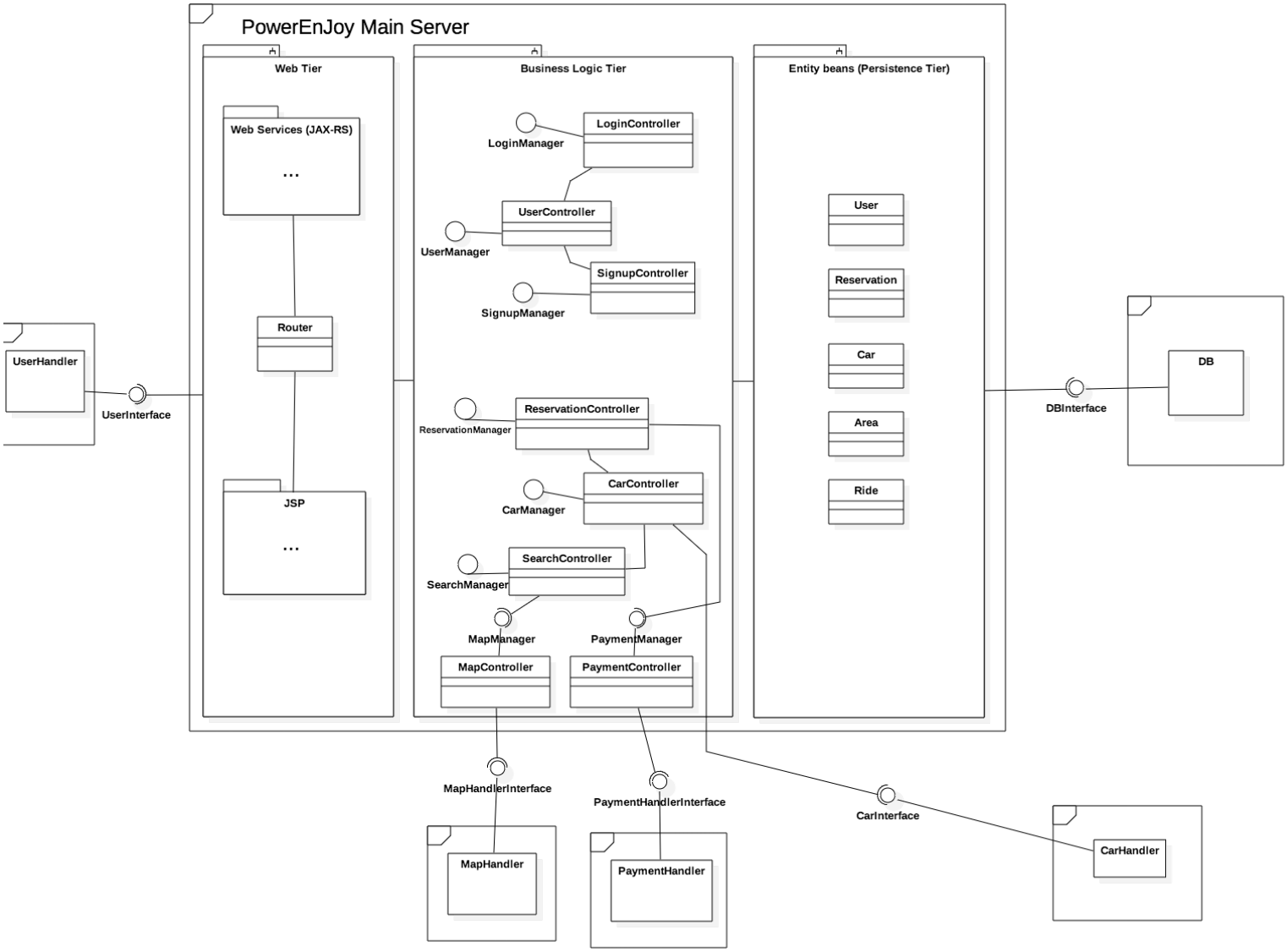
2.2.2 Technologies used



- The web app will be implemented using PhoneGap framework:
  - Interactions with the Server will be done via HTTP calls, using RESTful services offered;
- The Main Server will be implemented with Java Enterprise Edition (JEE):
  - Communication with the web app will be possible thanks to Java API for RESTful Web Services (JAX-RS);
  - The front-end functionalities, that clients need, will be provided with Java Server Pages (JSP);
  - Business Logic will be implemented using Session Java Beans, and communication with WebTier will be managed with the EJB Container;
  - Data management and interaction with Database Server will be implemented thanks to Java Persistence API (JPA), and will be composed of Entity Java Beans;
- For the Database Server we will use MySQL.

2.3 Component view

2.3.1 Main server component view

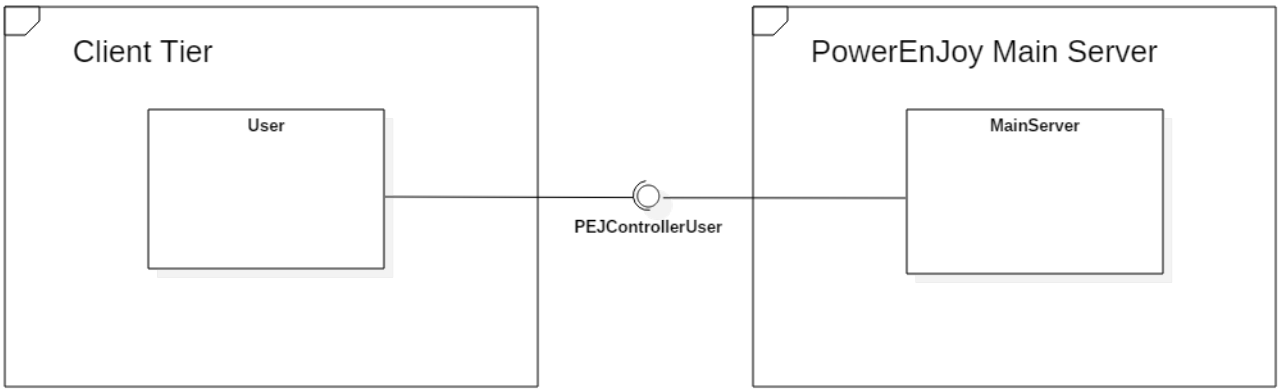


This is a view of the components located at the main server. These components are:

- Login Controller;
- User Controller;
- Signup Controller;
- Reservation Controller;
- Car Controller;
- Search Controller;
- Map Controller;
- Payment Controller.

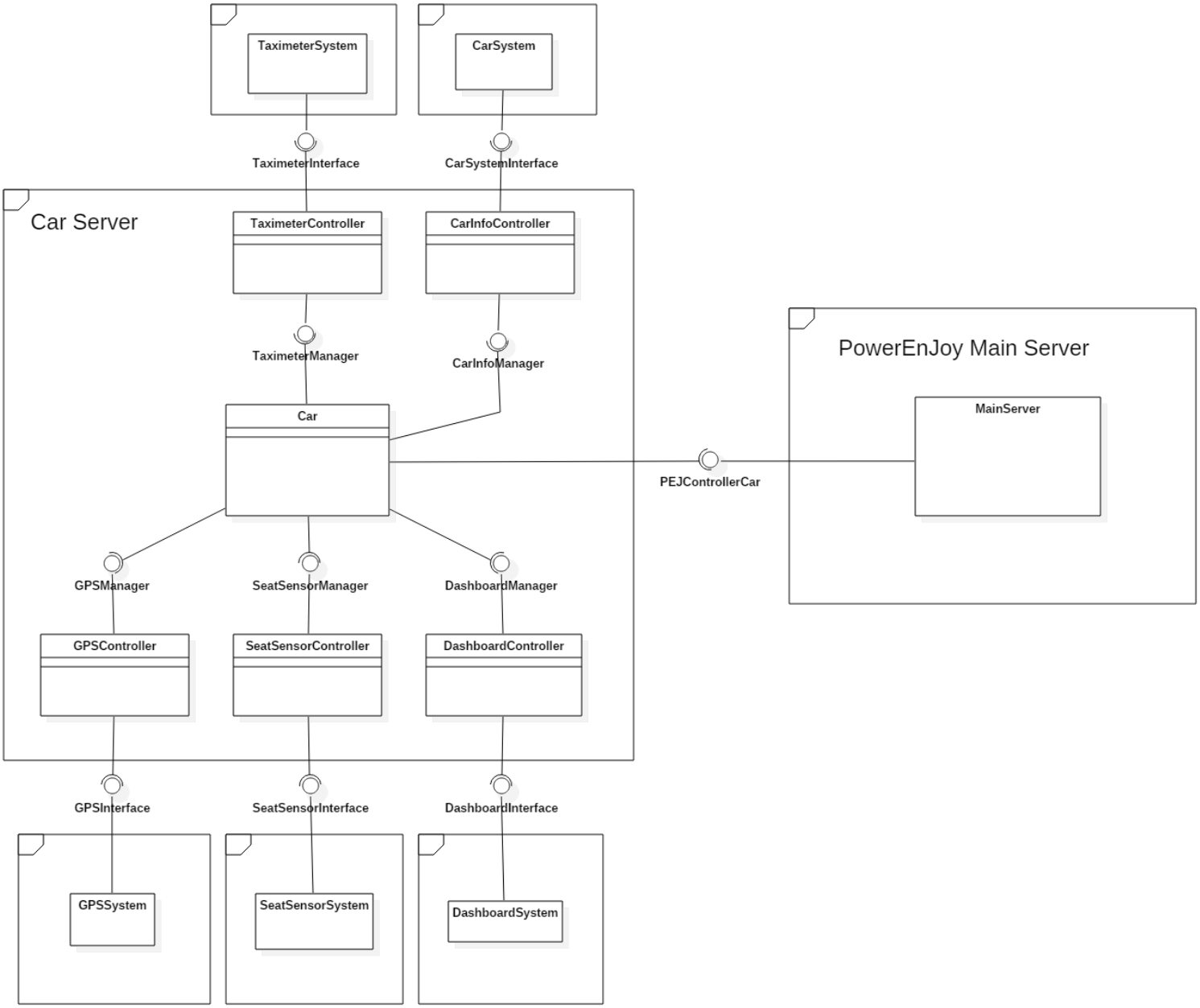
Interfaces with the external components are specified too.

2.3.2 User component view



This is a view of the components located at the client tier. There is no business logic at the client and therefore there's only an interface with the server (i.e. PEJControllerUser), that represents a sort of façade of the server functionalities that the client needs (e.g. get the map with the cars locations).

2.3.3 Car component view

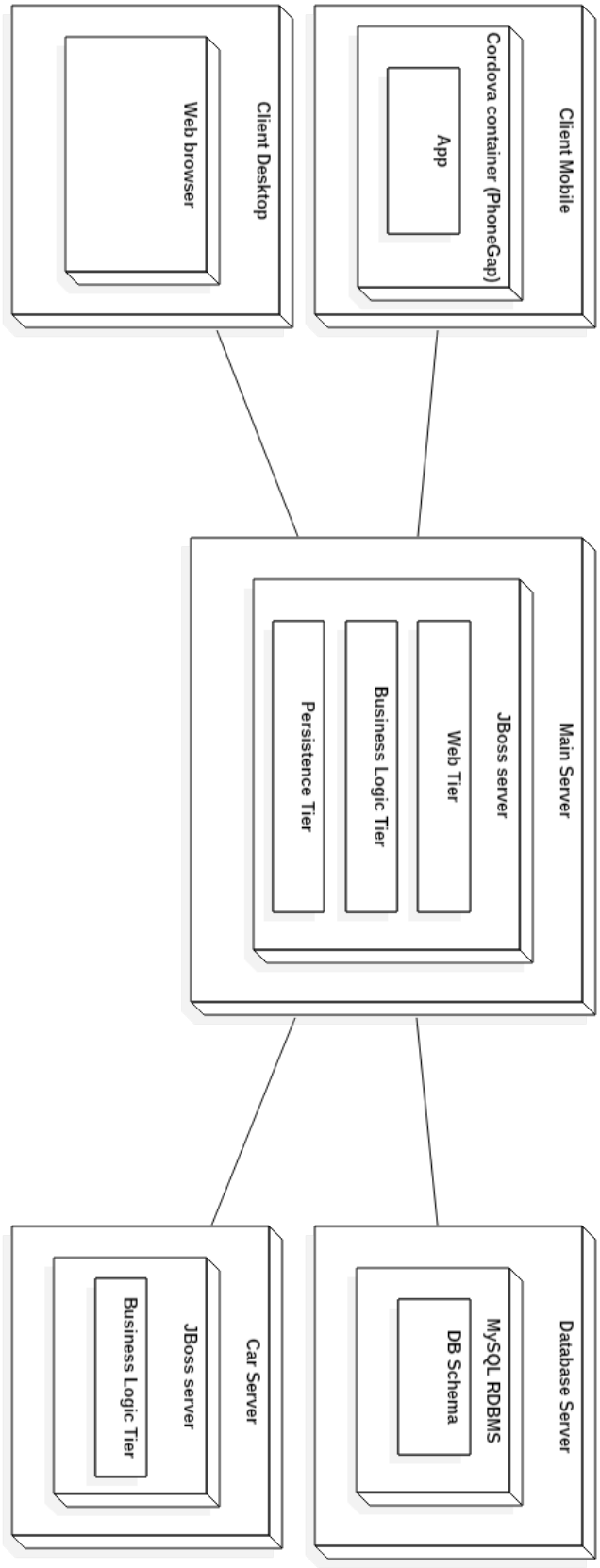


This is a view of the components located at the car server. The PowerEnJoy cars have more functionalities than normal cars, for example they must show the reservation code and they must be able to be located through GPS.
All these additional functionalities are embodied by controllers, such as the Taximeter Controller and the GPS Controller.

A car can use the server functionalities it needs thanks to an interface (PEJControllerCar, that represents how the car sees the Main Server) that acts as a sort of façade.

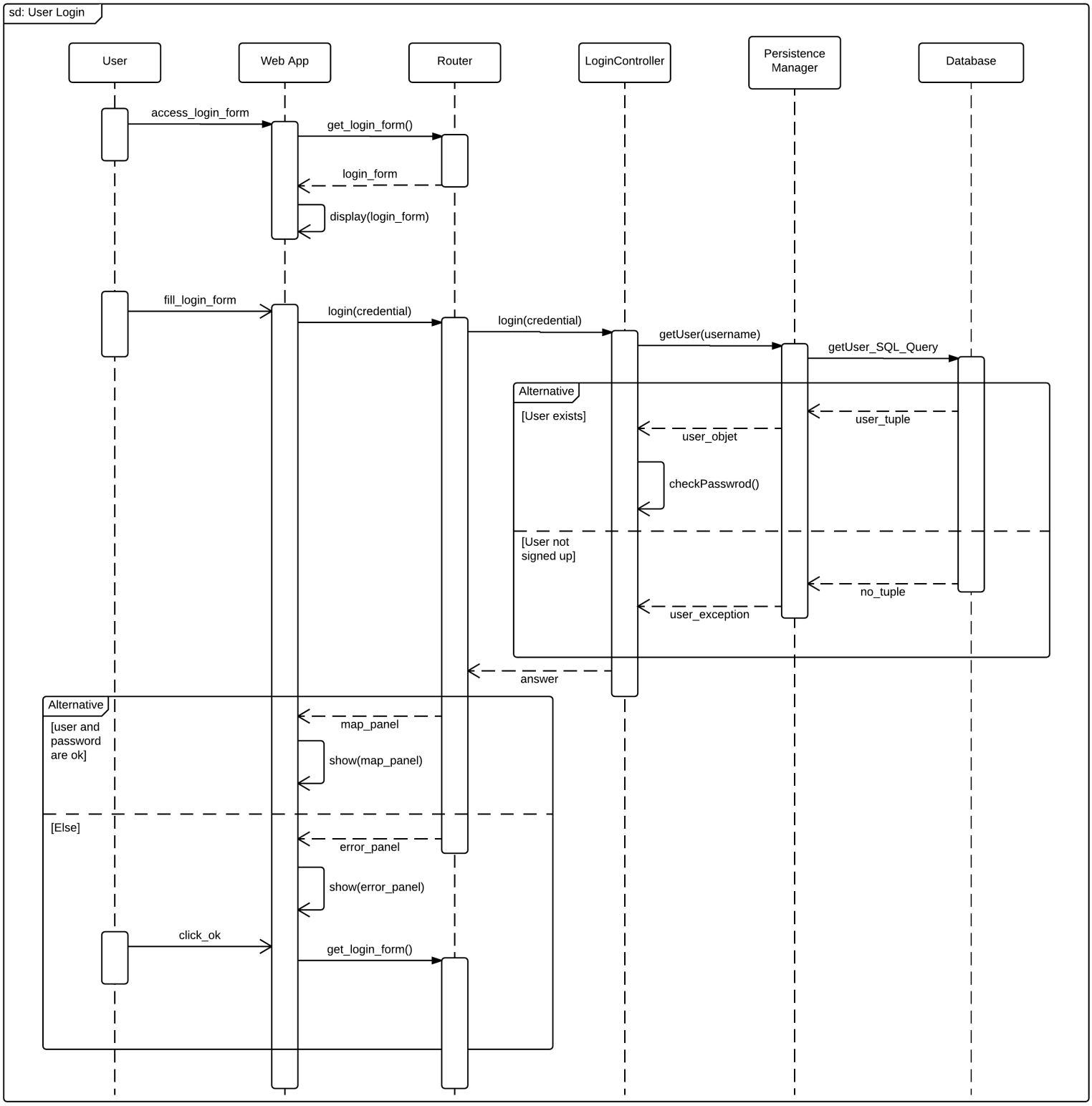
2.4 Deployment view

The client mobile app will run on the Cordova containers. We'll use the JBOSS open source application server for the Main Server and MySQL for the database.



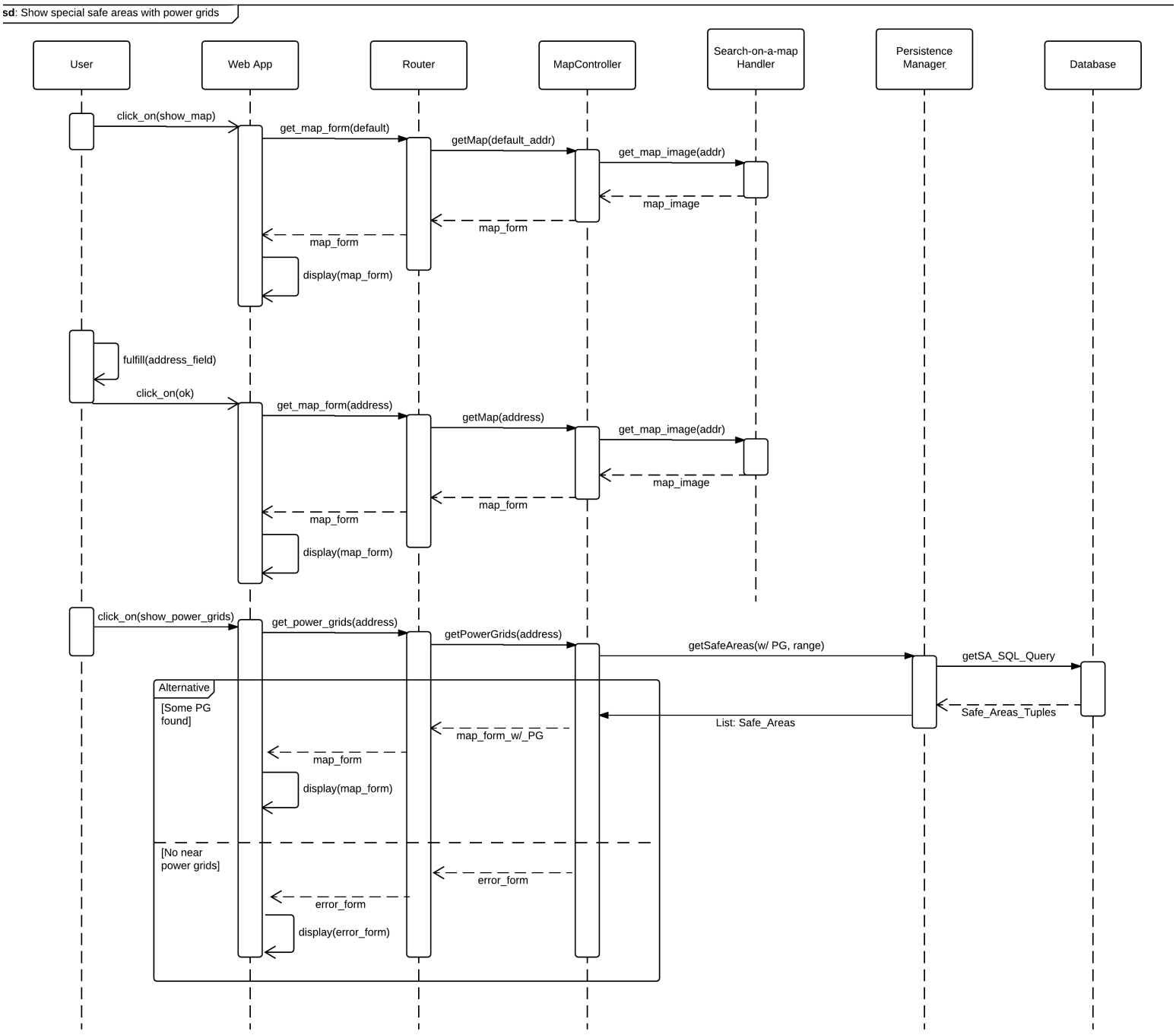
3 Runtime view

3.1.1 Log-in Sequence Diagram



This sequence diagram represents the user login.
The user has to select to login and fulfil the form with his credential.
The login request is then sent to the server. Once arrived to the system's router, the request is transferred to the LoginController, which first requires user information to the persistence tier. Then checks if the user exists or, if it does, checks if the password is correct. The web app receives either the map panel if credential was correct or an error panel if it was not, and display it to the user. If the user has to insert again credential, the app then displays again the login panel.

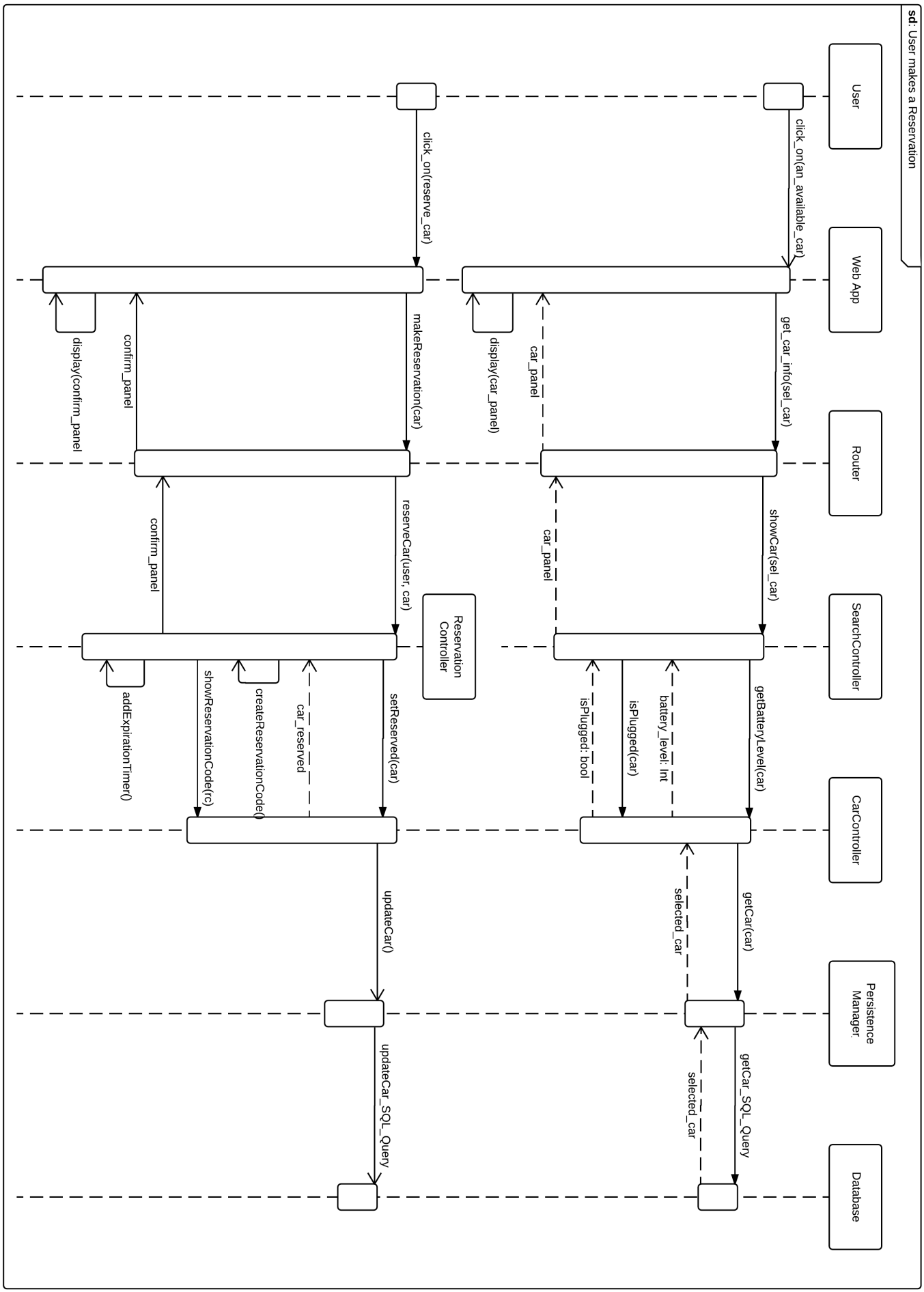
3.1.2 Show Special Safe Areas with Power Grids



This sequence diagram represents interactions between components in order to provide the user a map filled with Safe Areas with Power Grids in the right positions.
After user has chosen to go to the map page, system provides him a paged with a map centred in a default address. Then, thanks to the GPS sensor or, as in this case, after that the user has fulfilled the address field, the system loads the map centred on the address provided. This process can happen thanks to the

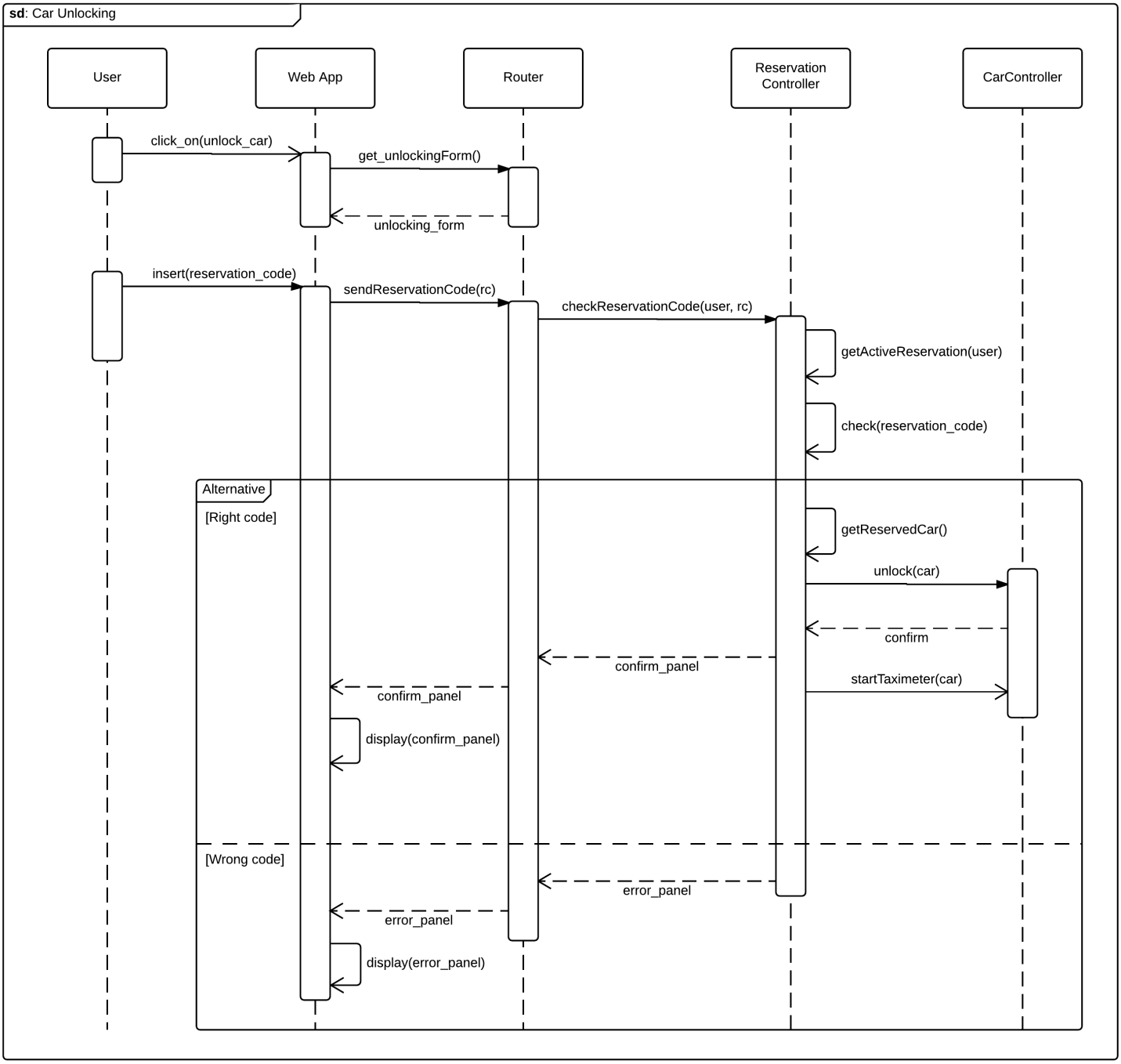
interaction between the MapController, in PowerEnjoy Main Server, and the search-on-a-map handler.
Then, the user selects to show positions of Power Grids. The request is moved to the MapController, which gets Power Grids Positions from the database, elaborates the map image overlaying a symbol in each position, and sends the answer back with the help of above layers. If there are not Power Grid within a certain defined range, the user gets an error message.

3.1.3 Reserve a Car



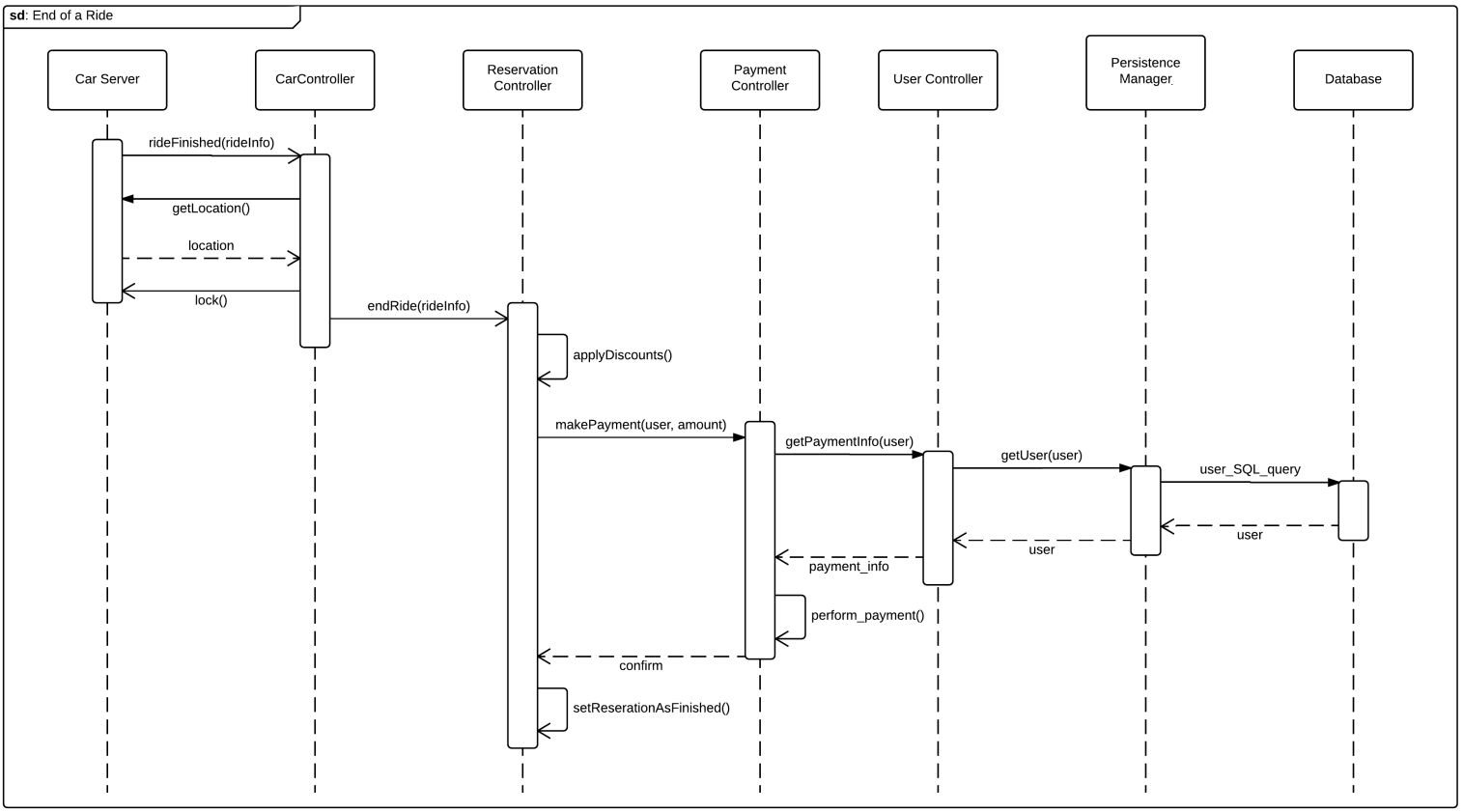
This sequence diagram represents the reservation of a car. We assume that the user has already logged-in and selected a Safe Area with at least one available car.
After user has selected one of them, the system provides him a page with all the information about that car. In this example, the Search Controller provides the information about battery level and if the car is plugged or not, but there can be many other. Then, the Reservation Controller deals with the creation of a new reservation. Thanks to the Car Controller, a Reservation Code is displayed on the interior screen of the car in order to give the ability to the user to unlock it. The Reservation Controller deals also with the creation of a timer that checks if a reservation should be expired or not, and manages consistency of data, marking the car as not available anymore.
Finally, the system informs the user about the reservation created.

3.1.4 Unlocking the Reserved Car



This sequence diagram is about unlocking of a car by the user who reserved it. We assume that the user has already logged-in and the reservation has not expired.
After user has inserted the code shown on the screen inside the car, the Reservation Controller finds the active reservation for that user and check if the code is right. If it is, the car is unlocked by Car Controller and the System starts charging user. Otherwise, an error message is shown to the user.

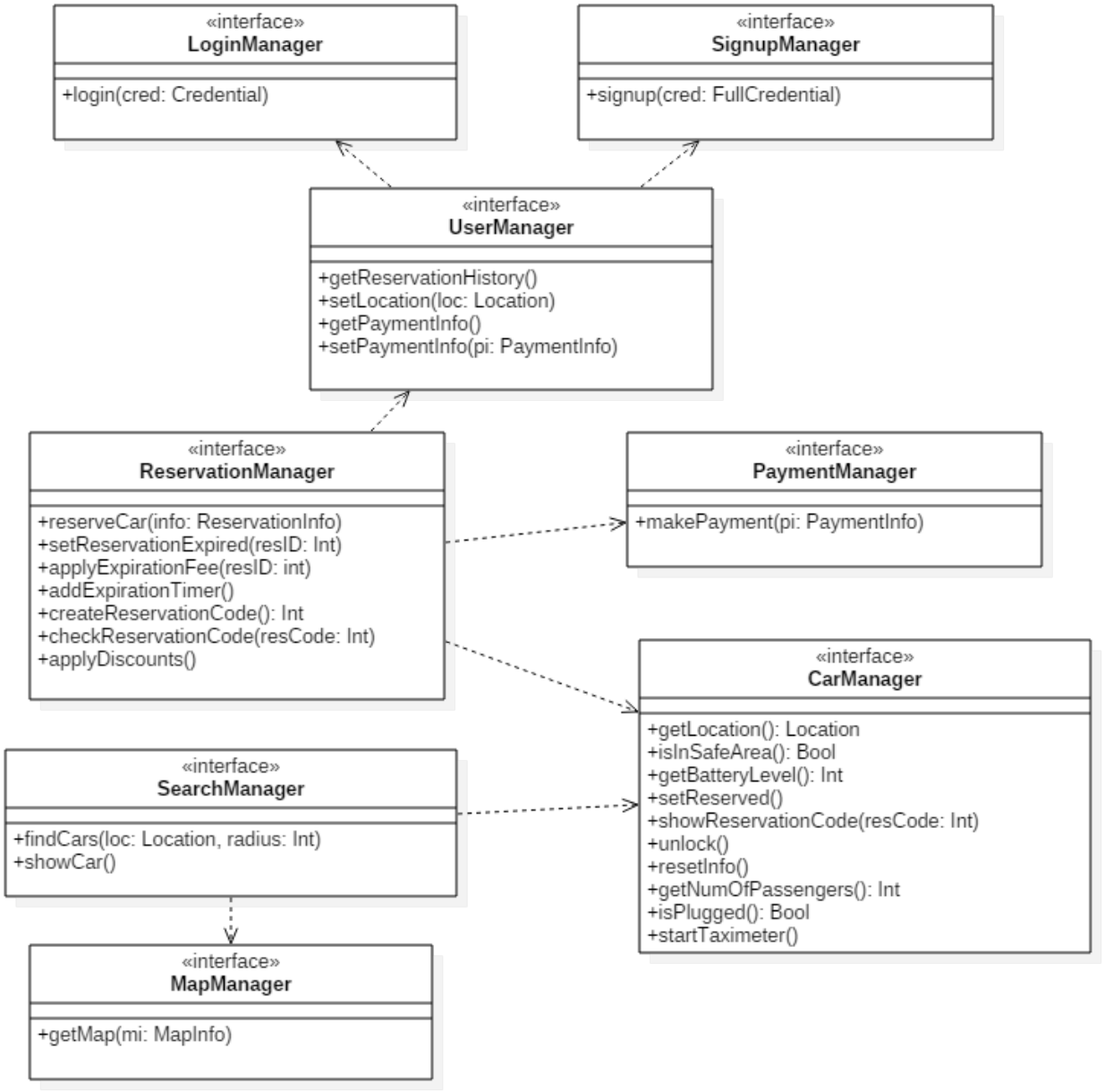
3.1.5 End of a Ride



This sequence diagram describes system interactions when a user exits the car and finish a ride. A cooperation between controllers is needed in order to achieve data consistency and to perform the payment.

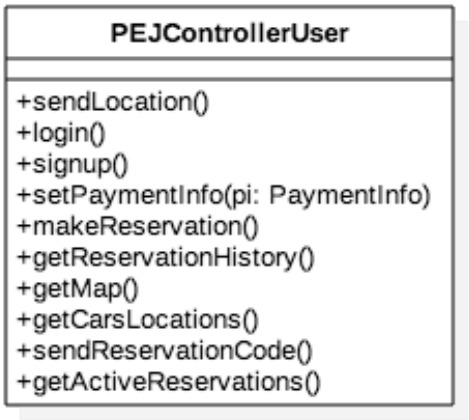
3.2 Component interfaces

3.2.1 Main server component interfaces



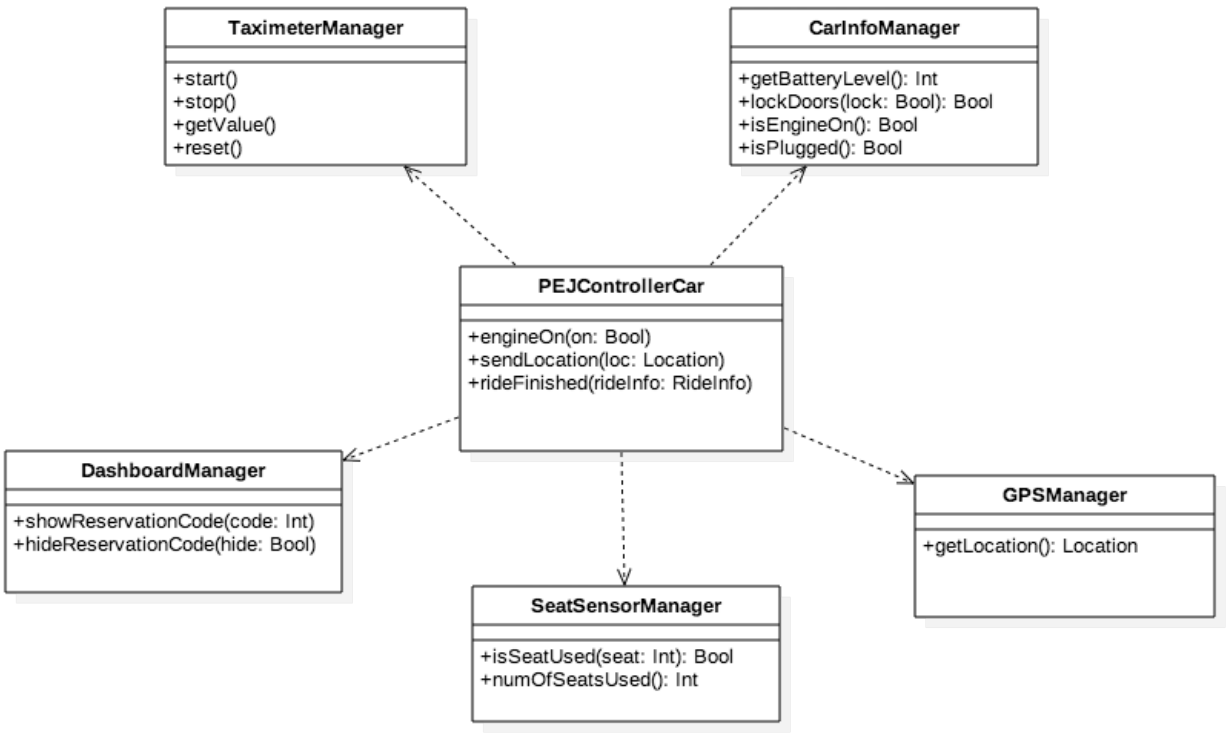
These are the component interfaces located at the Main Server. Their main functions can be found in the Runtime Analysis and in the Requirement Traceability chapters of this documents and do not represent the totality of functions of the future implementation, since complex functions can be splitted into more low level functions.

3.2.2 User component interfaces



This is the only component interface located at the client, that acts as a façade of the Main Server.

3.2.3 Car component interfaces



These are the component interfaces located at the car. They are used to retrieve car information such as its location (with the GPS Manager) or the number of passengers (with the Seat Sensor Manager).

3.3 Selected architectural styles and patterns

3.3.1 Architectural styles

The selected architecture has three tiers, with a thin client, a main server and a database, as already specified before.
We focused on give a high level of abstraction to the components. In particular, we have:

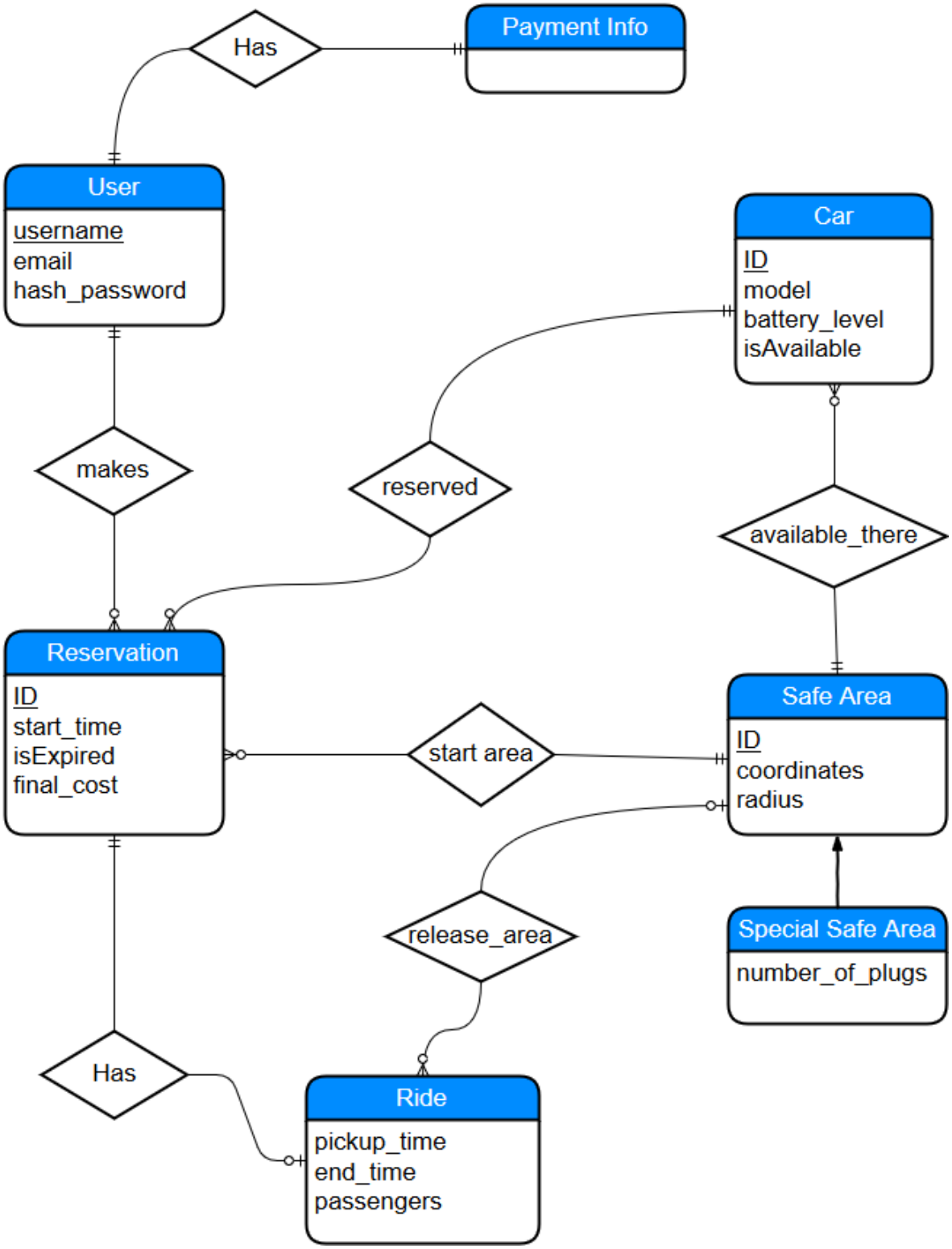
- ComponentManager: these interfaces, such as ReservationManager or CarManager, provide high level functionalities to the other components of the system, hiding low level procedures needed to achieve them;
- ComponentController: these objects are the implementation of the related ComponentManager interface. In this way, it is possible to develop multiple version of a controller, and change it without modifying other components;
- ComponentInterface: these interfaces, such as CarInterface or PaymentHandlerInterface, provide low-level functionalities to controllers, hiding communication protocols and allowing optimizing the process.
- ComponentHandler: these objects are the implementation of the related ComponentInterface. They provide functionalities communicating with external component or services. Thanks to the abstraction of interfaces, it is possible, for example, to implement two different Handlers, allowing different optimization strategies, such as a proxy.

3.3.2 Design Patterns

In order to achieve a software structure with good qualities, it has been designed following some design patterns. For example, the whole tier division is comparable to the architectural MVC (Model-View-Controller) Pattern: the Database Server contains and manages the model, the Main Server works as Controller, and the Web-app (or the web site, in the feature), in collaboration with the web tier in the main server, provides the View of the system.
Furthermore, the whole system is strongly based on a Client-Server communication model, because it allows a great scalability, maintainability, but also a good security. As a consequence of the MVC pattern, the Observer pattern is used.
The Façade pattern is used with external components.
The Composite pattern and the Singleton pattern is used to implement the algorithm presented in the Algorithm Design chapter.

3.4 Other design decisions

Here we show a high-level representation of internal relational structure of the database, through an ER (Entity-Relation) Diagram.



3.5 Extra: class diagrams for main server components

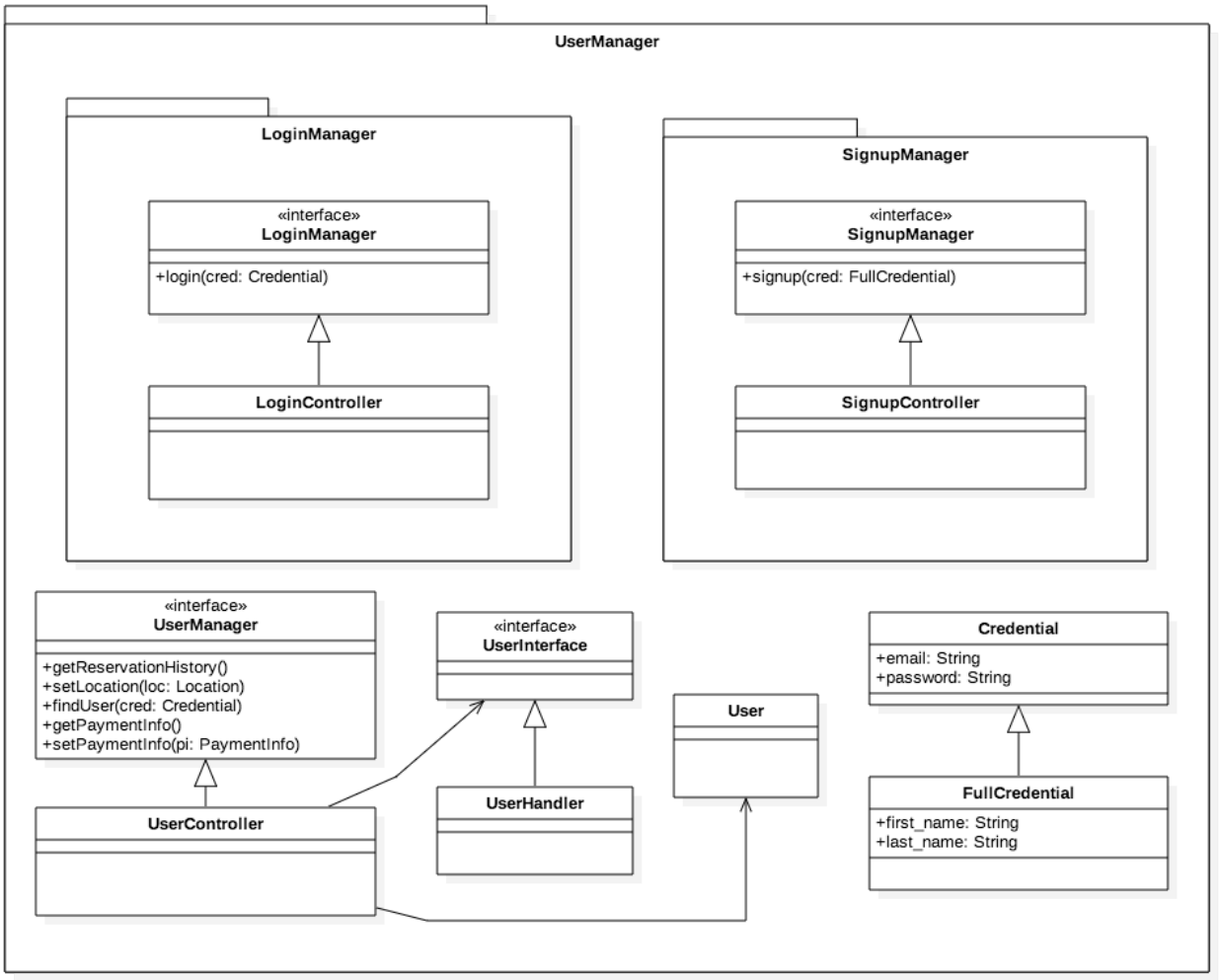
Here we show briefly some class diagrams for the components located at the Main Server. They are by no means complete but they are usefull since they give a little more low-level view of how our system works.

We splitted the components in three sections according to their main use:

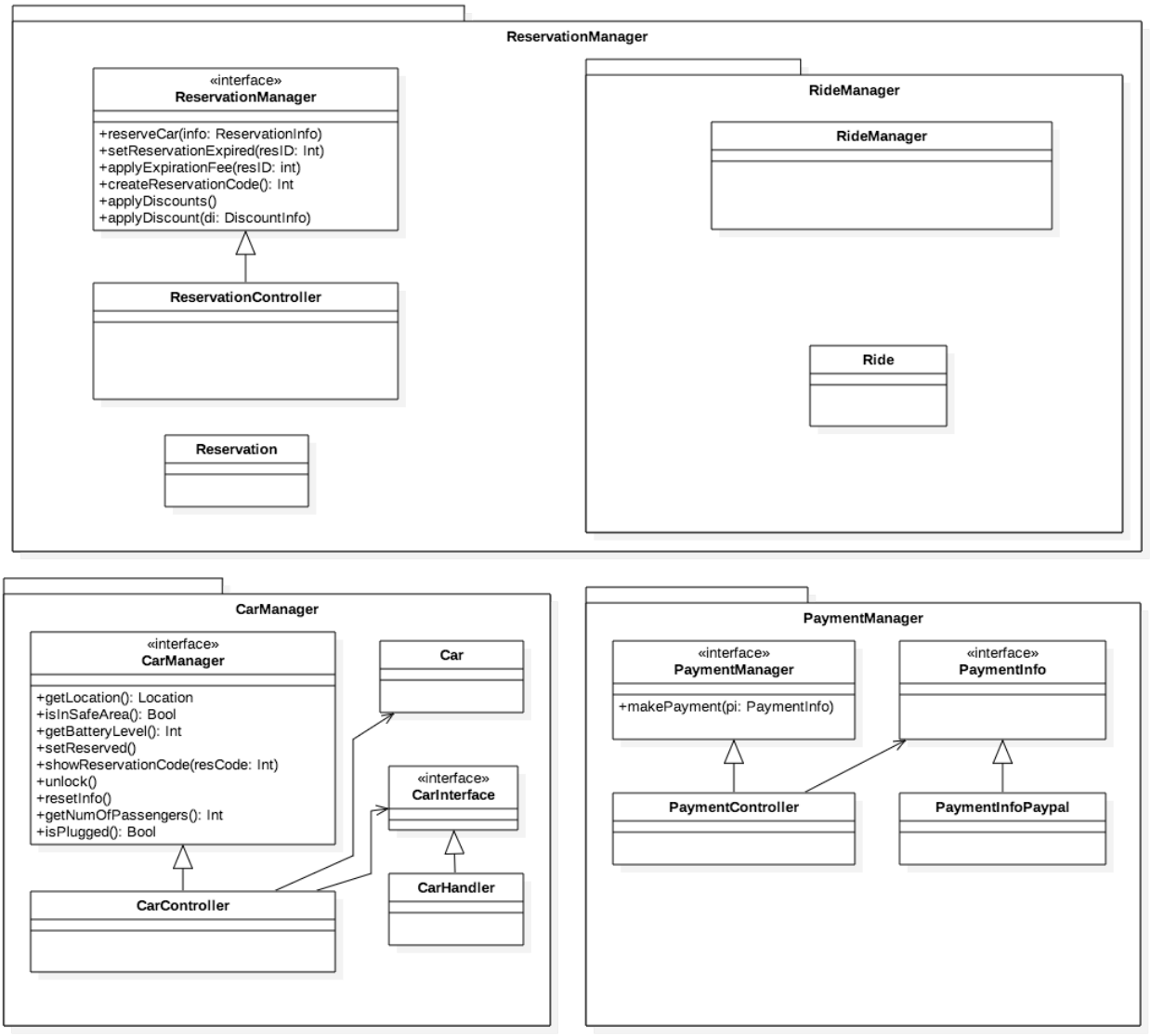
- User related components;
- Reservation related components;
- Search related components.

Of course some components should belong to more than one of these sections.

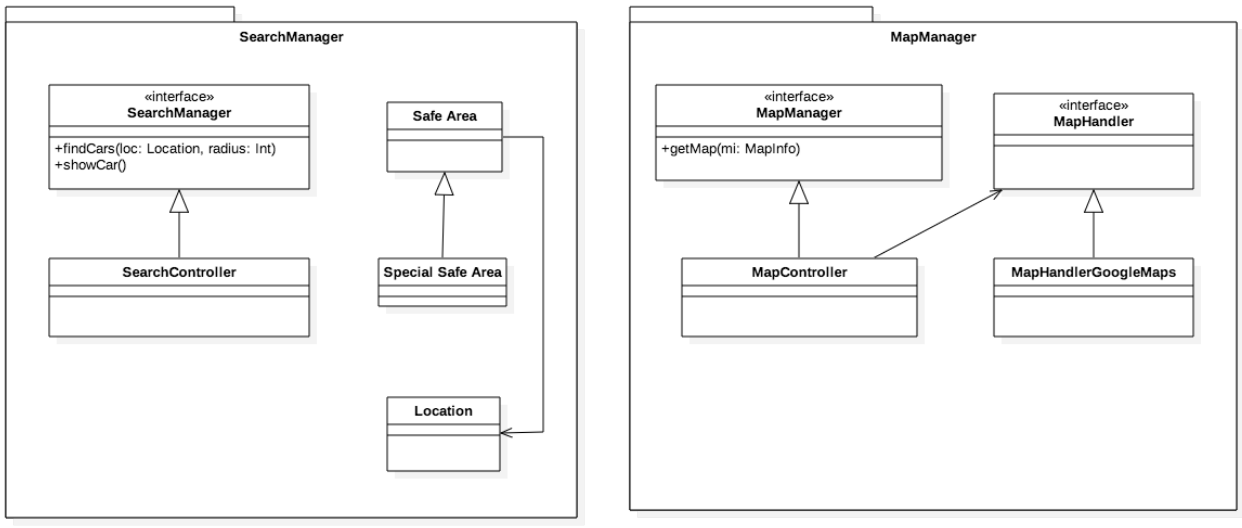
3.5.1 User related



3.5.2 Reservation related

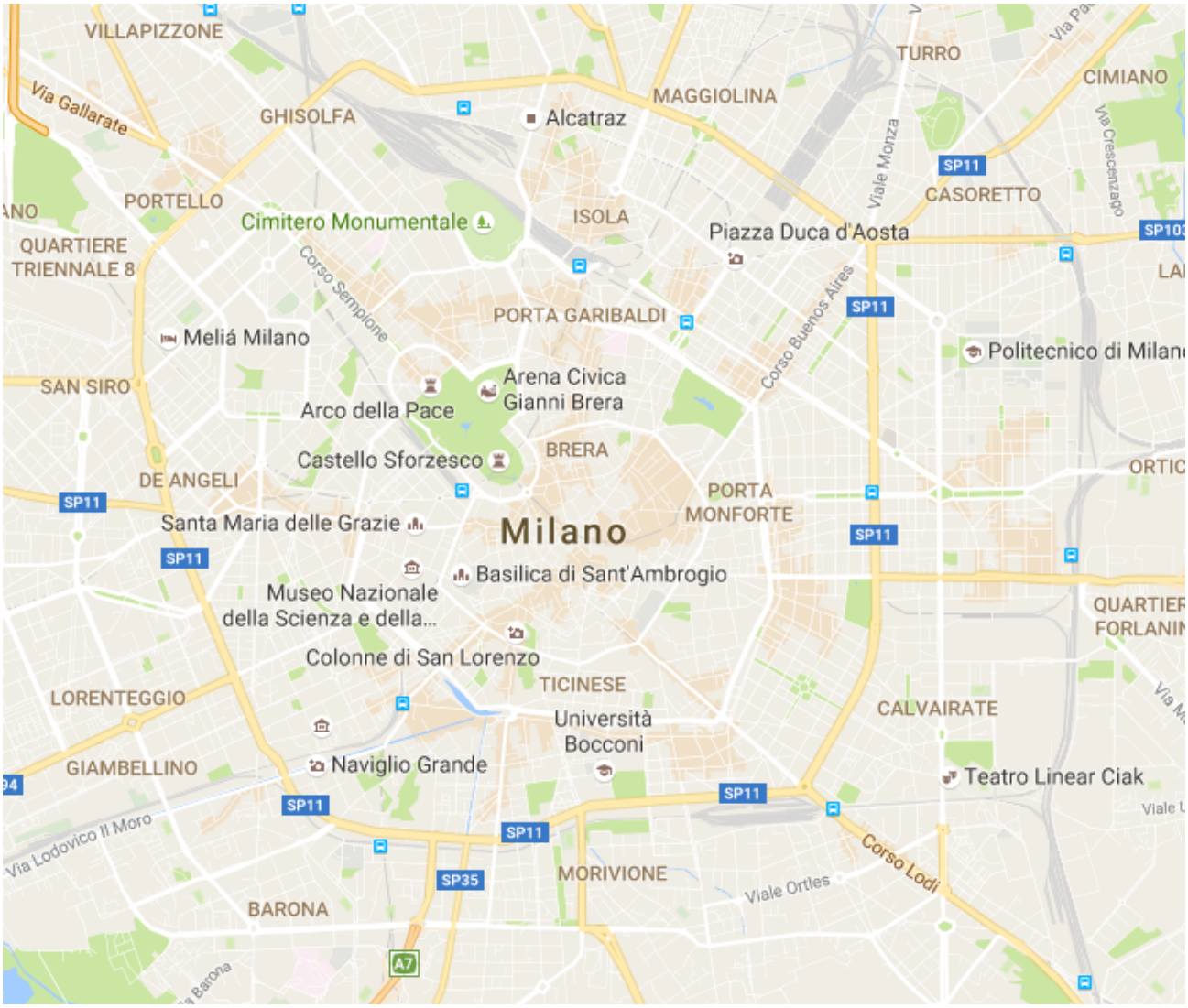


3.5.3 Search related

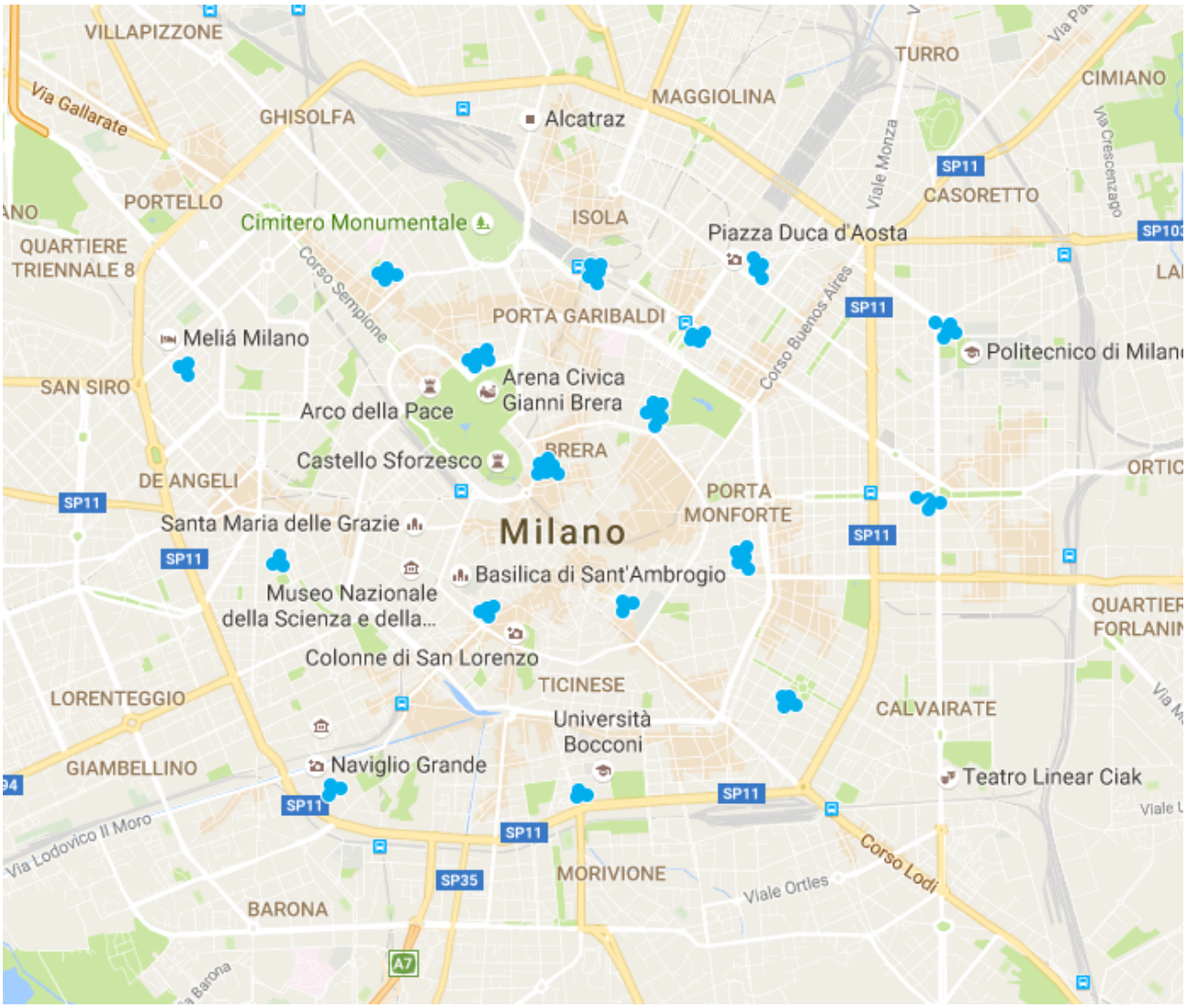


4 Algorithm Design

An interesting and important algorithm is the one related to the search for near cars.
It may be too expensive comparing the distances between the user location and the location of each car in the database.
An important thing to point out of our system is that the database stores the cars locations, but they are not constantly updated. Indeed, they are updated only at the end of the ride of a reservation. This implies that cars locations in the database are reliable only when a car is parked in a safe area without an active reservation. However, this is exactly the moment in which such cars must be shown to the users.
In order to do this efficiently, we'll use spatial data structures.
Consider this map of Milan.

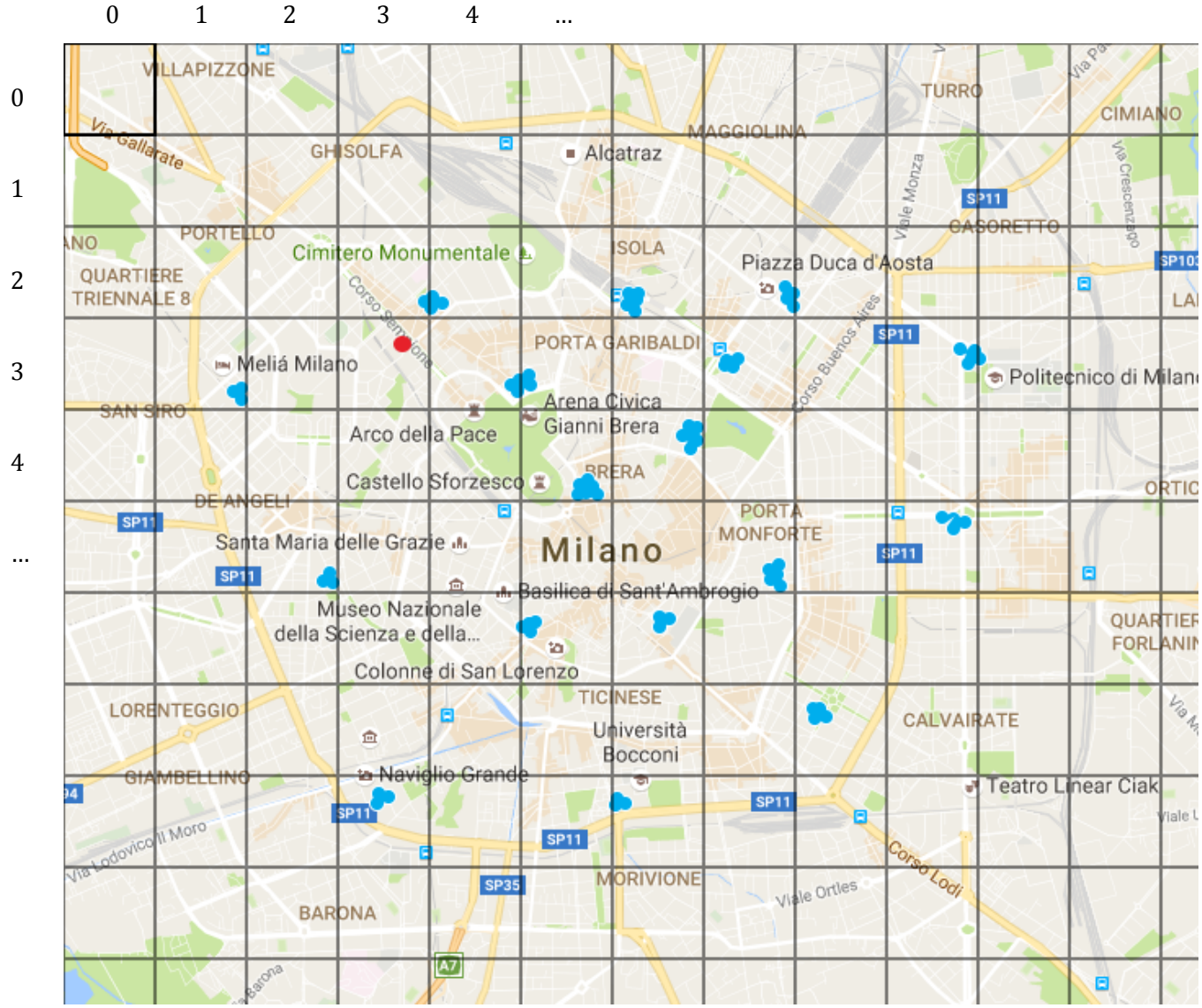


Now, this is the same map but with the PowerEnJoy cars drawn on it.



There is a total of 17 parkings drawn. However, PowerEnJoy will probably have more.

Instead of comparing the distance between the user condition and the locations of all the cars of our system in Milan, we can split Milan in sectors, as shown in the next image.



The red circle is the user position.

In order to find the near cars, we just have to check which cars belong to the sections near the user position.
The user is in section S(3, 3). Suppose we are interested in cars near the user in a radius of 2km and each square has side long 1km.
Then, we should check cars in the same section of the user and in the sections around it that intersect with the circumference with radius 2km and the user position as center. The sections we'll check are S(2, 2), S(2, 3), S(2, 4), S(3, 2), S(3, 3), S(3, 4), S(4,2), S(4, 3) and S(4, 4). In this way we only consider 7 cars out of the total 50+ cars.

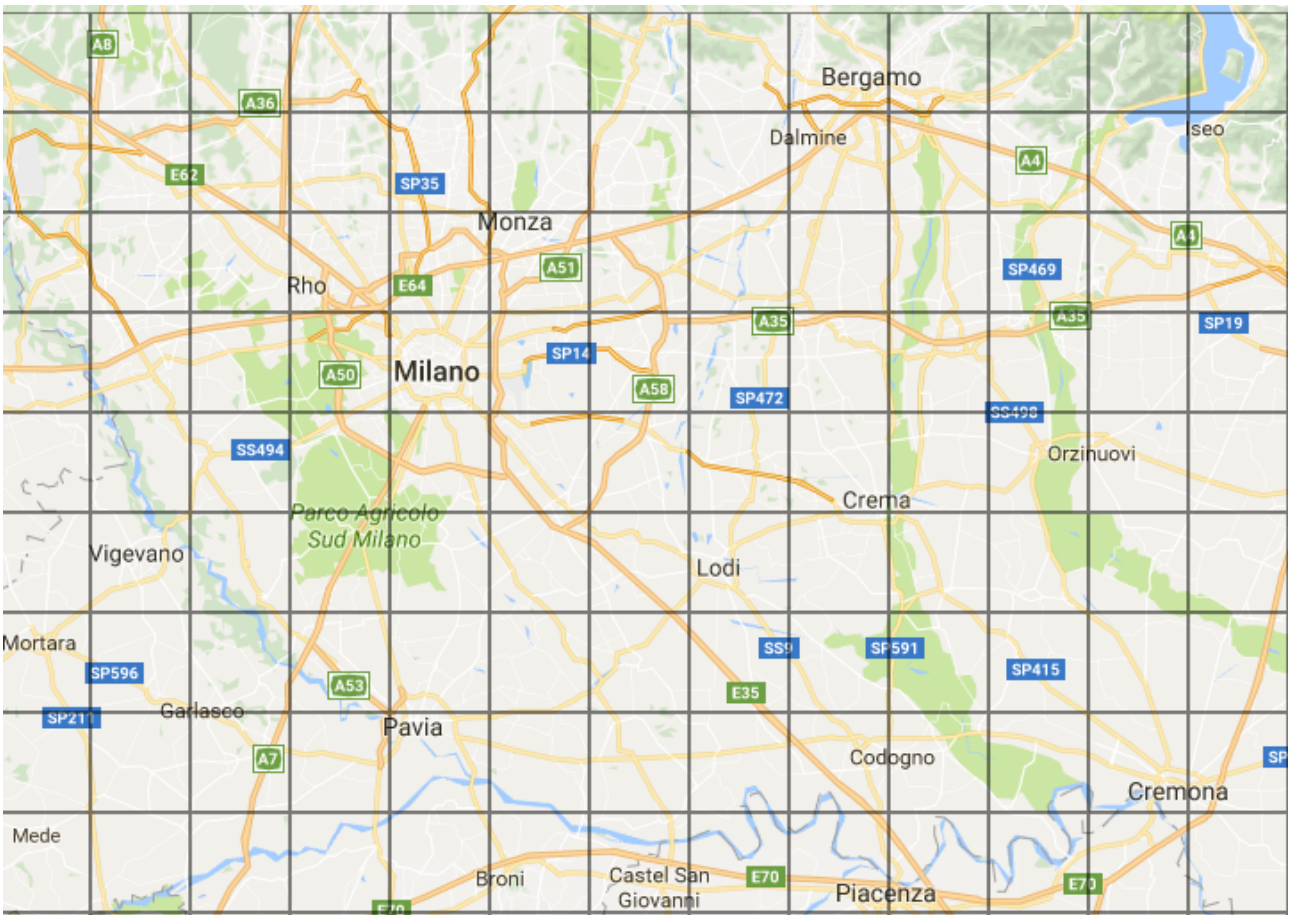
The downsides of this algorithm are:

1. There should be always a matrix allocated containing all the sectors. A matrix represent a big city and therefore it would be difficult to find near cars if we found ourselves in positions near its borders.
2. More is the section sides length, more is the precision of our algorithm in cutting off cars from the computation (more speed), but it will need more

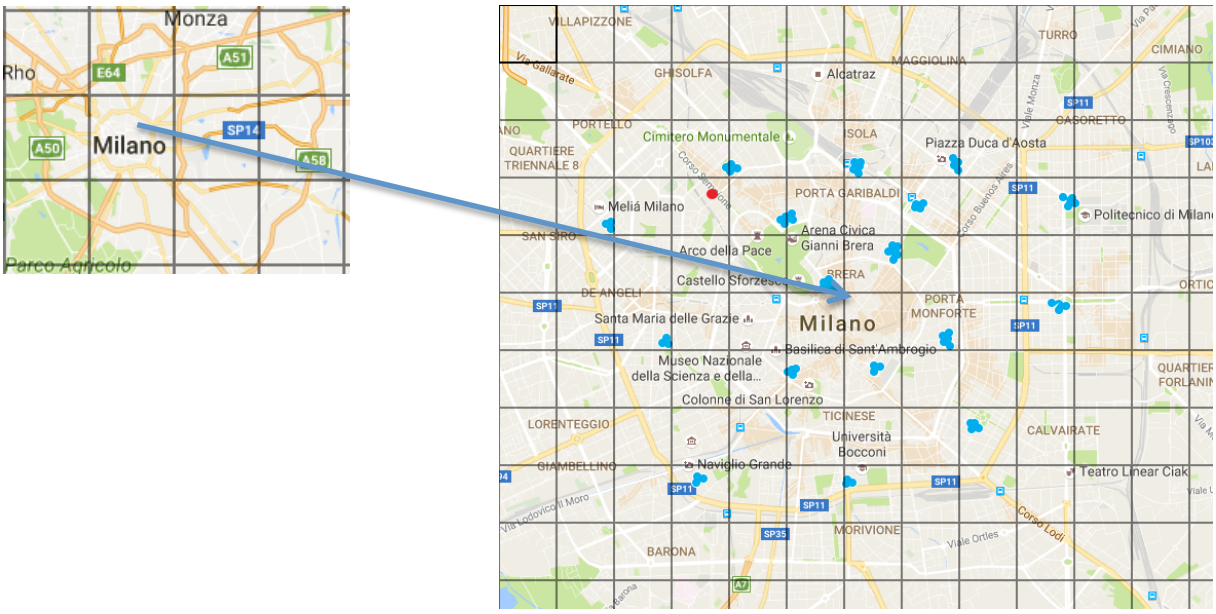
memory to memorize the big number of sectors (less memory). This works viceversa too.

3. We should allocate cars to sectors in a continuous way. However, since we are interested in the cars positions only when they are available, it will be enough to update their position in the database only at the end of a ride. Therefore this is not a downside in our case.

It is possible to solve the first downside by organizing sectors in a hierarchical way. Consider the following image of a part of Lombardia.



The sector of Milan will contain the matrix of such city that we previously saw.



Another advantage of organizing sectors in a hierarchical way is that we are not forced to always use the same sector side length. For example, if Milan is more densely populated with cars of our system than Bergamo, then we can use a smaller sector side length for Milan in order to increase the number of cars that will be cut off during searches.

This fact doesn't solve completely the second downside, but give us more control on the tradeoff between speed and memory.

This hierarchy can of course be seen as a tree. With a careful implementation, it's possible to inject different types of matrix in the same one (i.e. some sectors of Lombardia can be expandible in other sectors and others don't).

We can now provide the pseudocode for our data structures and for the search of near cars.

```
public class SectorCar {
	private int id;
	private double latitude;
	private double longitude;
	private LeafSector sector;

	/* Getters and setters */
	…
}

public interface Sector {
	// Returns true if contains a list of cars, false if contains a
matrix of other sectors.
	boolean isLeafSector();

	// Also updates car.sector.
	void addCar(SectorCar car);
	void removeCar(int carID);
}

public class LeafSector implements Sector {
	private List<SectorCar> cars;

	public LeafSector() { … }

	public boolean isLeafSector() {
		return true;
	}
	public void addCar(SectorCar car) { … }
	public void removeCar(int carID) { … }
}

public class NodeSector {
```

```
	private int sideLength; // in km
	private int rows;
	private int cols;
	private Sector[][] matrix;

	// By default, each Sector it's a LeafSector.
	public NodeSector (int rows, int cols, int sideLength) { … }

	/* Getters and setters */
	…

	public boolean isLeafSector() {
		return false;
	}
	public void addCar(SectorCar car) { … }
	public void removeCar(int carID) { … }

	// This method does two things:
		1 - it finds the previous sector of the car by doing
	car.sector and removes the car from its list.
		2 - it finds the new sector of the car using latitude and
	longitude and assigns the car to this sector.
		void updateCarPosition(SectorCar car, double latitude, double
	longitude);

	/* The following method replace a sector with another one.
	There are different cases:
	- LeafSector -> LeafSector: useless.
	- LeafSector -> NodeSector: we want more precision in such area.
All the SectorCar are reassigned to the right Sector.
	- NodeSector -> LeafSector: we want less precision. All the cars in
the NodeSector will be assigned to the LeafSector.
	- NodeSector -> NodeSector: some variables between rows, cols and
sideLength may vary. In this case, all the SectorCar are reassigned to
the right Sector. */
	public void replaceSector(Sector newSector, int row, int col) { … }
}

/* This class contains the NodeSector that contains the area in which
our system operates */
public class CarTreeSingleton {
	private static CarTreeSingleton instance;
	private NodeSector root;
	private Map<Integer, SectorCar> sectorCarMap;
	private CarTreeSingleton() {
		this.instance = new CarTreeSingleton();
		this.instance.root = new NodeSector(TREE_SECTOR_ROWS,
		TREE_SECTOR_COLS, TREE_SECTOR_SIDELENGTH);
		this.instance.sectorCarMap = new HashMap<Interger,
		SectorMap>();
	}
```

```
	public static CarTreeSingleton getInstance() { … }

	public NodeSector getRoot() { … }
	public Map<Integer, SectorCar> getSectorCarMap() { … }
}
```

In the following code, we fill the CarTreeSingleton for the first time (to be done when the server launches).

```
CarTreeSingleton cts = CarTreeSingleton.getInstance();
NodeSector root = cts.getRoot();
Map<Integer, SectorCar> sectorCarMap = cts.getSectorCarMap;

foreach Car car in database {
	SectorCar sectorCar = new SectorCar(car.id, car.latitude,
car.longitude);
	root.addCar(sectorCar);
	sectorCarMap.put(car.id, sectorCar);
}
```

The following code instead shows how to substitute a LeafSector with a NodeSector, so that more precision can be achieved in that area.

```
CarTreeSingleton cts = CarTreeSingleton.getInstance();
NodeSector root = cts.getRoot();

NodeSector sectorMilan = new NodeSector(rows, cols, sideLength);

root.replaceSector(sectorMilan, sectRow, sectCol);
```

Finally, the following code shows how to update the position of a car.

```
public void onRideConcluded(Ride ride, …) {
	int carID = ride.car.id;
	int carLatitude = ride.car.latitude;
	int carLongitude = ride.car.longitude;

	CarTreeSingleton cts = CarTreeSingleton.getInstance();
	NodeSector root = cts.getRoot();
	Map<Integer, SectorCar> sectorCarMap = cts.getSectorCarMap;

	SectorCar sectorCar = sectorCarMap.get(carID);
	root.updateCarPosition(sectorCar, carLatitude, carLongitude);
}
```

The problem is now divided in many easier subproblems. The most difficult methods to implement are NodeSector::updateCarPosition and NodeSector::replaceSector. They are explained in their comments and can be easily implemented with the help of Sector::addCar and Sector::removeCar and by making some comparisons with the dimensions of the varius sectors.

It would be great to make the system automatically manage the sectors hierarchy in order to achieve performance closer to optimum without human intervention.

This can be done, for example, by finding simple and efficient rules such as "when a LeafSector contains more than 20 cars, replace it with a NodeSector whose matrix is 3x3" and "when a NodeSector contains less than 10 cars in total, replace it with a LeafSector". Such sector transformations may occur after each concluded ride or at specific instances of time during the day. Some performance testing are necessary.

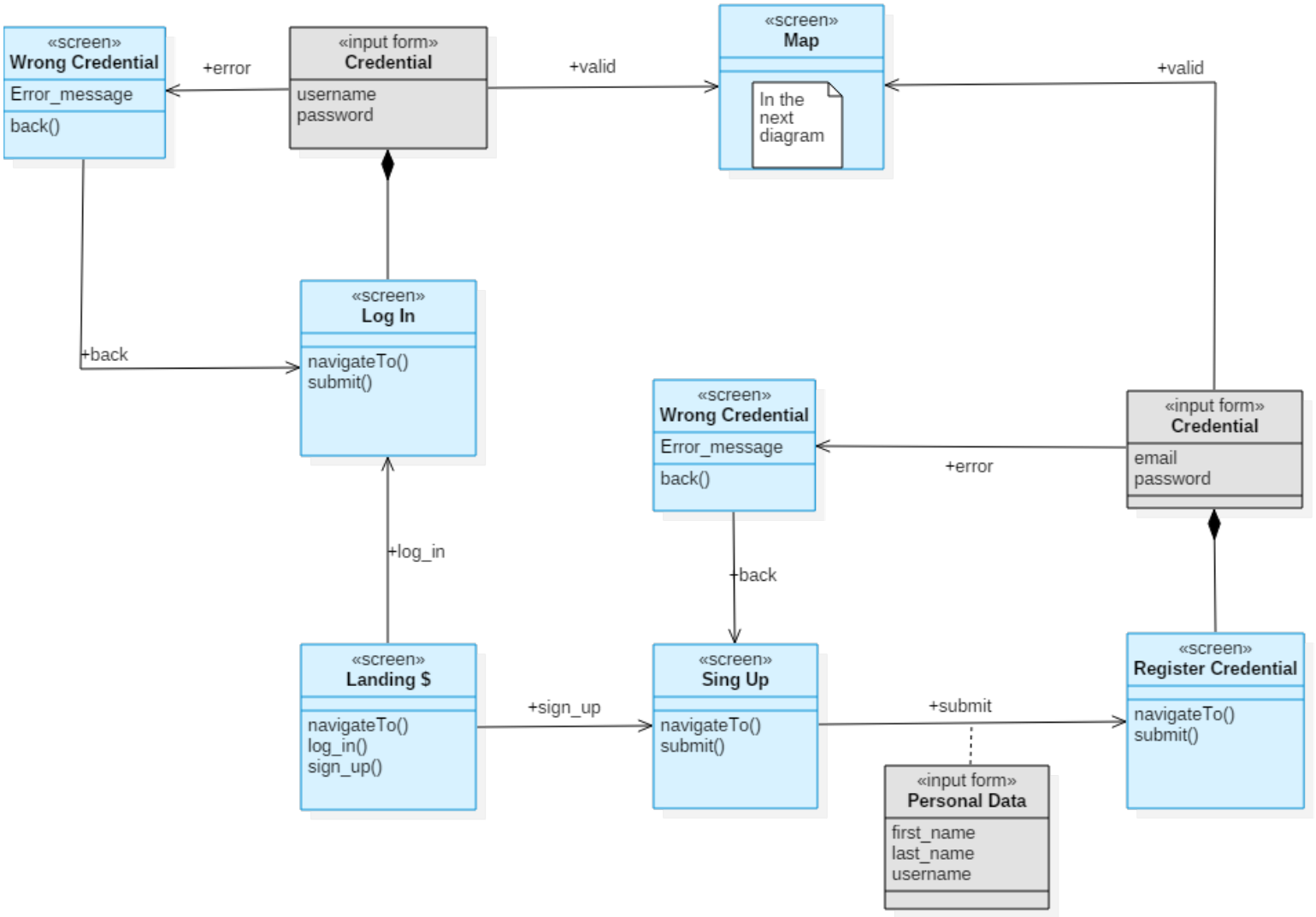
A similar algorithm can be used to manage the search of parkings near to a certain location. However the improvements in this case would be less than in the case of the serach for near cars, since probably the number of parkings is less than the number of cars and moreover the parkings never move themselves.

5 User Interface Design

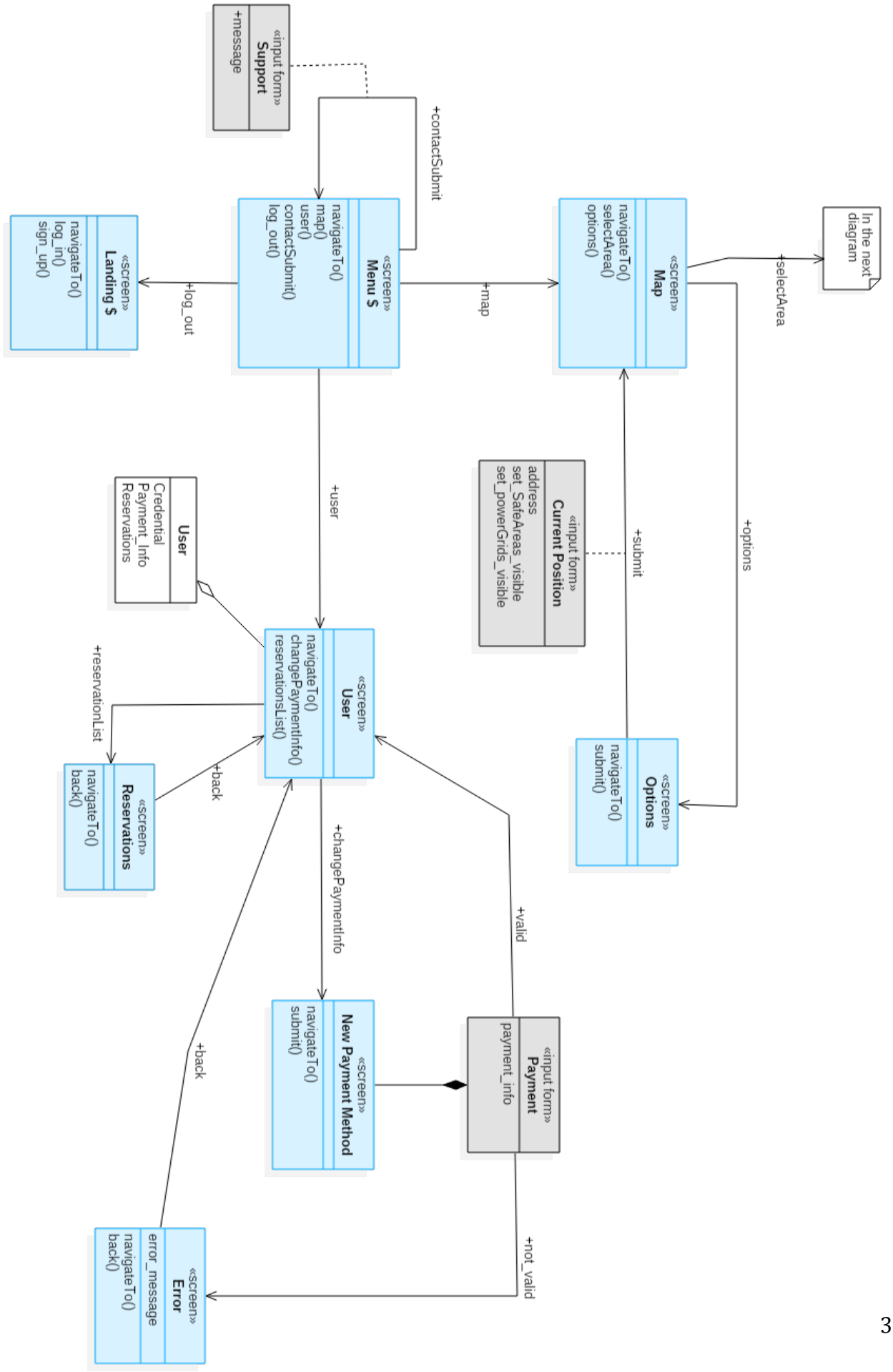
Here we provide User eXperience Diagrams and corresponding Mock-Ups related to the web-app. The web-browser will have a similar look and feel, and user interactions will be very similar.

5.1 UX Diagrams

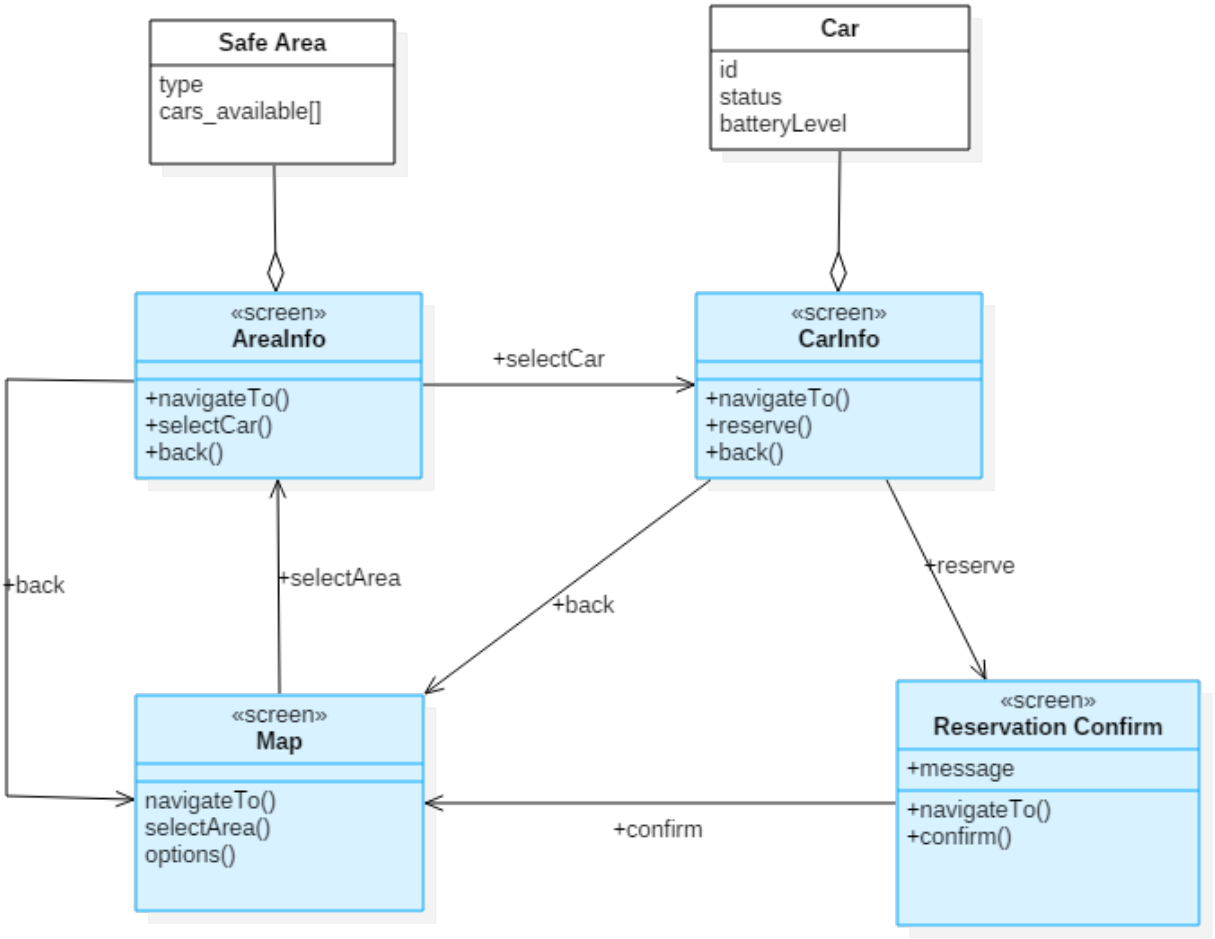
5.1.1 Log-in and sign-up



5.1.2 Search and user management

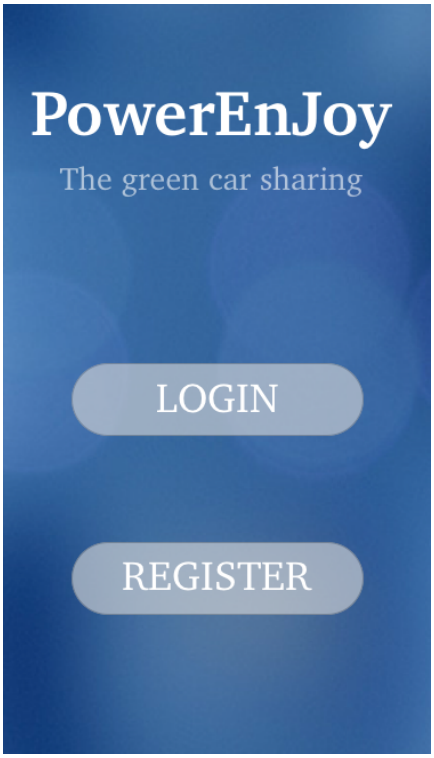


5.1.3 Reservation



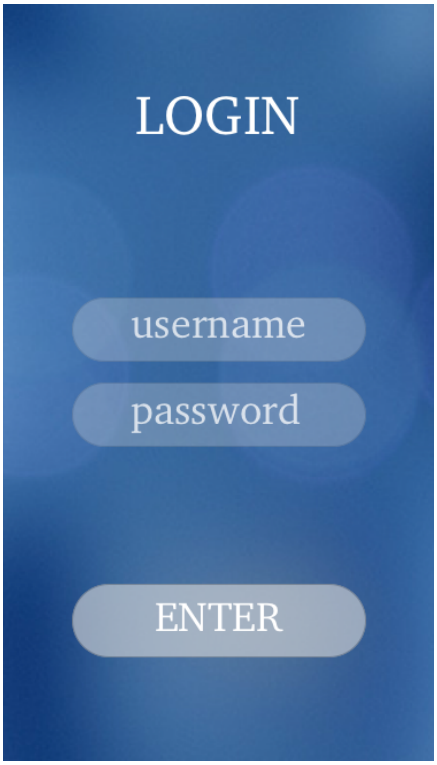
5.2 Mock-ups

5.2.1 Log-in and sign-up



1 - Landing

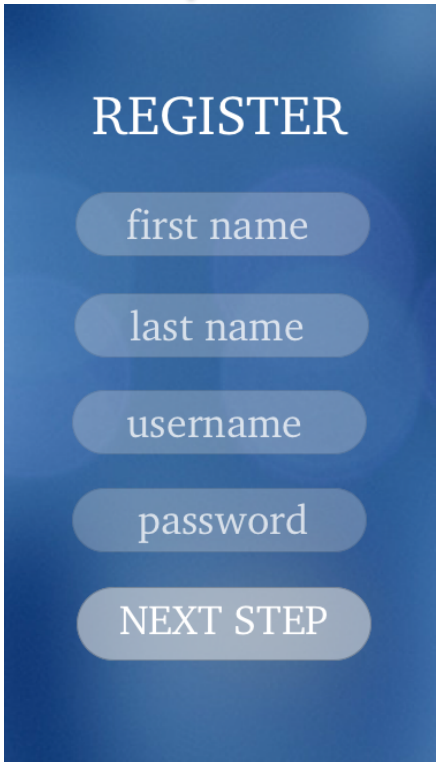
Login



2 - Login

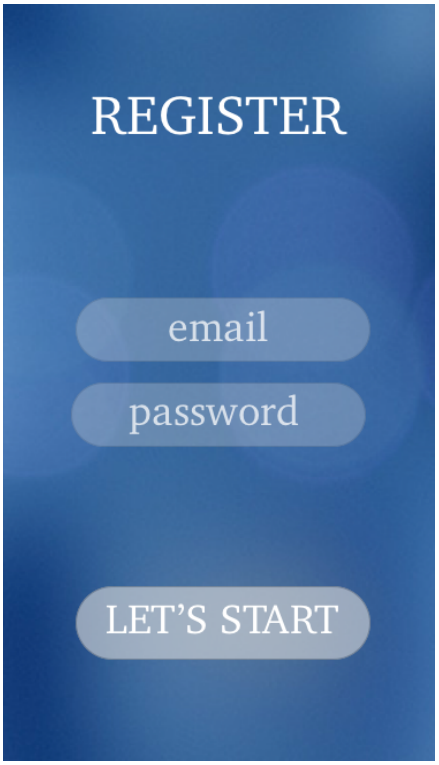
Enter

Register



3 - Sign Up

Next Step



4 - Credential

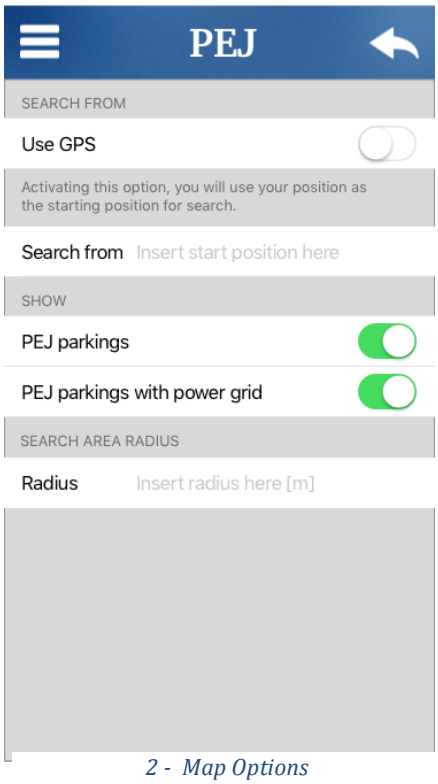
Let's Start

5.2.2 Search and user management

1 - Map View

Back

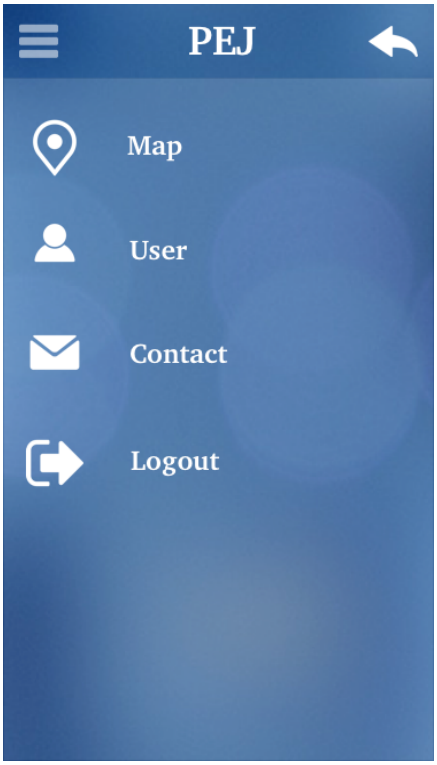
Options



2 - Map Options

Menu

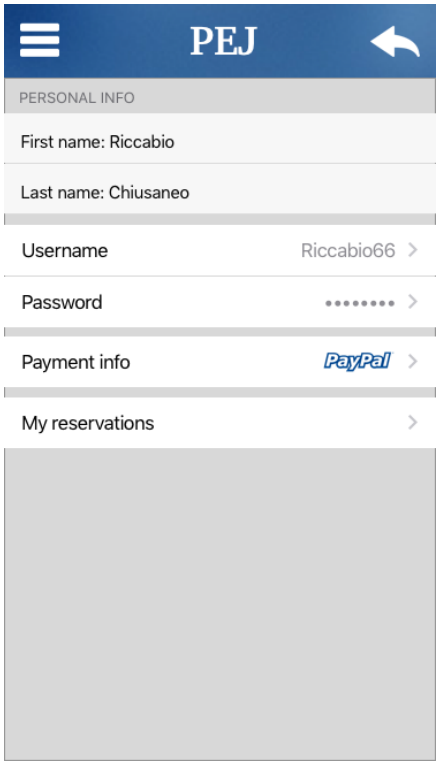
Map



3 - Menu

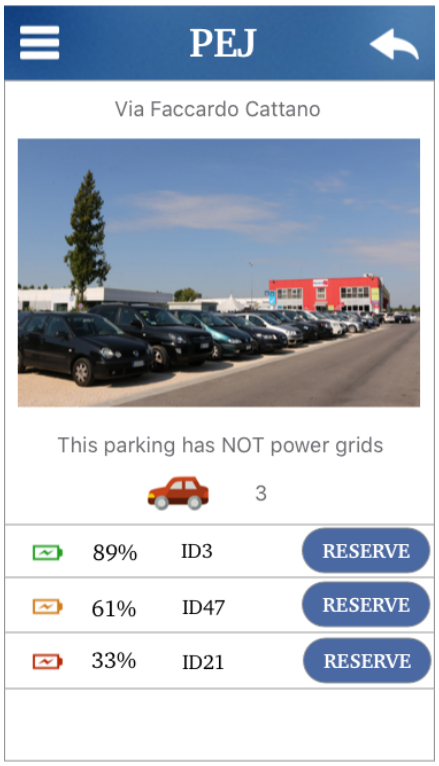
User

Back



4 - Personal Panel

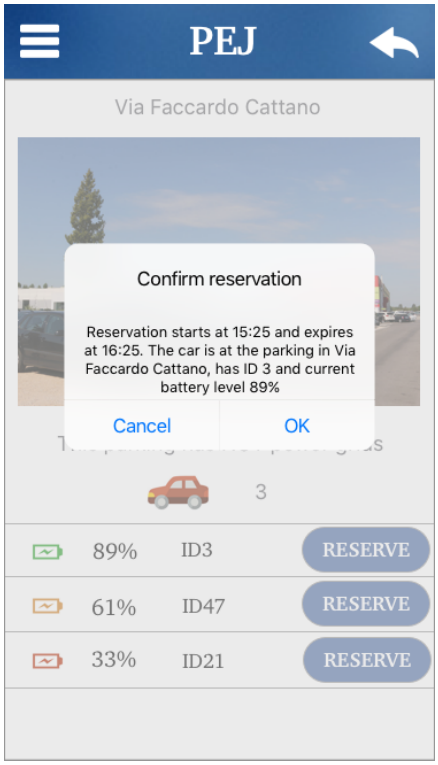
5.2.3 Reservation



1 - Safe Area Information

Reserve

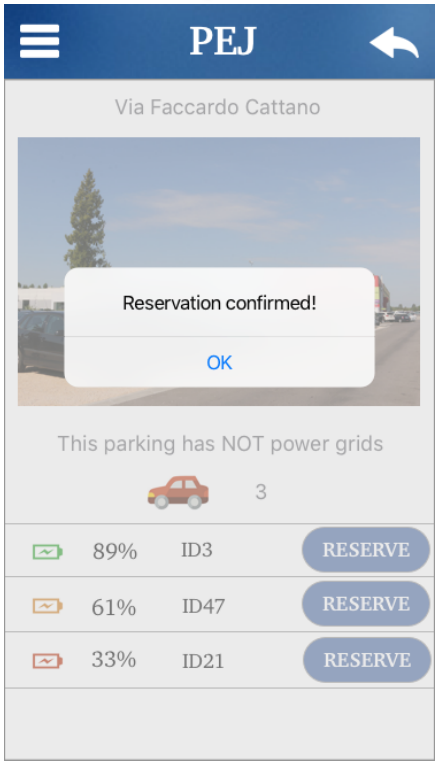
Cancel



2 - Reservation Confirm Message

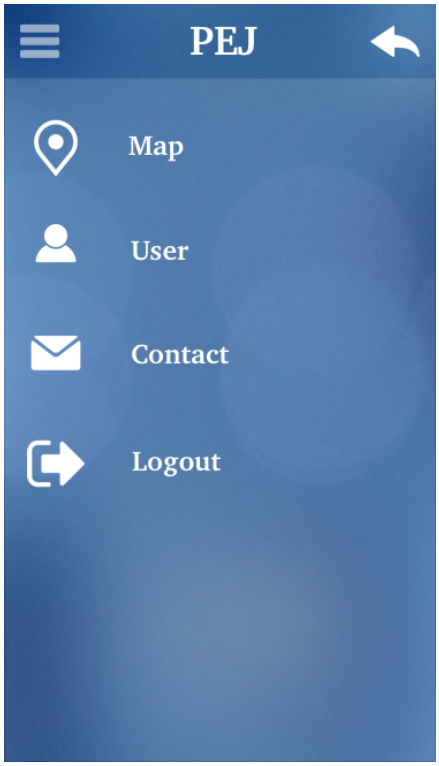
OK

OK



3 - Car successfully reserved

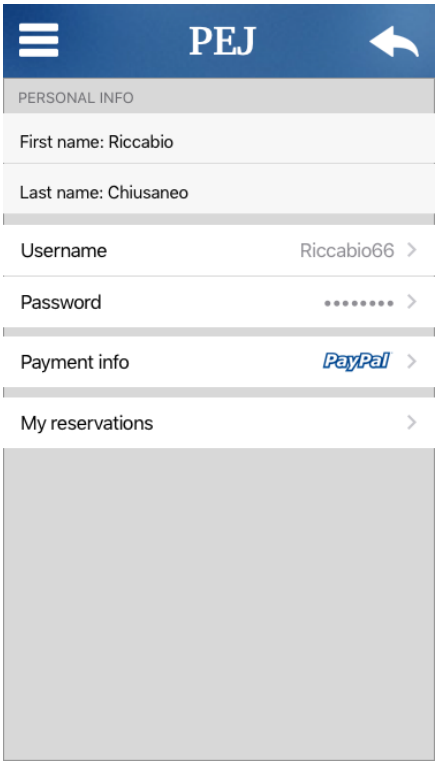
5.2.4 Unlock car



1 - Menu

User

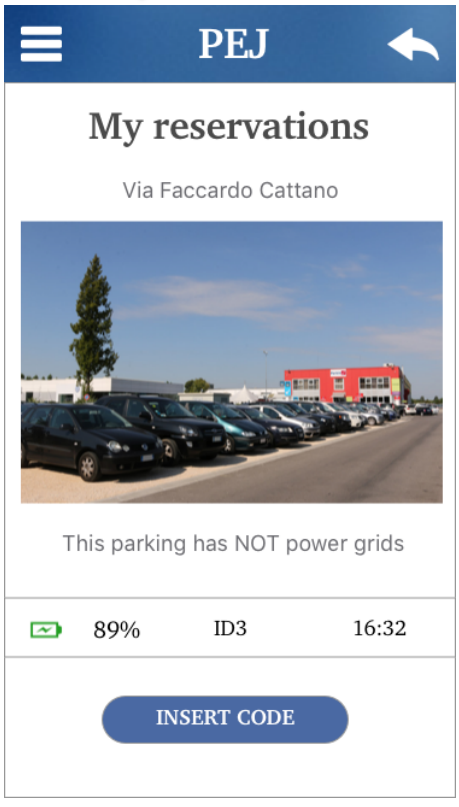
Back



2 - Personal Panel

My reservations

Back



3 - Reservations Panel

6 Requirements Traceability

The following tables show how each requirement specified in the RASD document is satisfied by the architecture proposed. The *components* and *description* columns give and idea of all the functions involved.

- [G1] *Users can see and select an available car close to him, or close to a specified address, and reserve it for up to one hour before they pick it up.*

REQUIREMENT	COMPONENTS	DESCRIPTION
R1: The system has to detect if a car is parked in a Safe Area	Car server: • GPSController Main server: • CarController	Car server: • GPSController.getLocation Main server: • CarController.getLocation, • CarController.isInSafeArea
R2: The system has to detect the battery level of each car	Car server: • CarInfoController Main server: • CarController	Car server: • CarInfoController.getBatteryLevel Main server: • CarController.getBatteryLevel
R3: The system has to detect car position and display it on the map	User browser/app: • PEJControllerUser Car server: • GPSController Main server: • MapController, • CarController, • SearchController	User browser/app: • PEJControllerUser.getMap • PEJController.getCarsLocations Car server: • GPSController.getLocation Main server: • MapController.getMap, • CarController.getLocation, • SearchController.showCar
R4: The system has to be able to identify the location of a user or through his/her GPS, if he/she gives the consent, or through some input	User browser/app: • PEJControllerUser	User browser/app: • PEJControllerUser.sendLocation
R5: The system has	Main server: • SearchController	Main server: • SearchController.findCars

to provide a list of available cars close to a given address		
R6: The system has to give the possibility to reserve a car at most by one user at a time	User browser/app: • PEJControllerUser Main server: • ReservationController, • CarController Car server: • DashboardController	User browser/app: • PEJControllerUser.makeReservation Main server: • ReservationController.reserveCar, • ReservationController.createReservationCode, • CarController.setReserved Car server: • DashboardController.showReservationCode
R7: The system has to mark the reservation as expired for a car after one hour if the user has not picked it up	Main server: • ReservationController (+ JEE Timer Service)	Main server: • ReservationController.setReservationExpired
R8: The system has to apply a fee of 1€ if the reservation has expired	Main server: • ReservationController	Main server: • ReservationController.applyExpirationFee

- [G2] *Users can get in a car only if they are near it and they reserved it.*

REQUIREMENT	COMPONENTS	DESCRIPTION
R1: Cars, if reserved, should show a reservation code on their dashboard	Main server: • CarController Car server: • DashboardController	Main server: • CarController.showReservationCode Car server: • DashboardController. showReservationCode
R2: The system has to unlock the	User browser/app: • PEJControllerUser Main server:	User browser/app: • PEJControllerUser.sendReservationCode

car if the user that reserved it sends the reservation code to the system	• CarController Car server: • CarInfoController	Main server: • CarController.unlock Car server: • CarInfoController.lockDoors

- [G3] *Users should pay proportionally to minutes they have used the car, and they should see in real time the amount of the bill.*

REQUIREMENT	COMPONENTS	DESCRIPTION
R1: The system has to reset trip information when a user get on the car	Main server: • CarController Car server: • TaximeterController • DashboardController	Main server: • CarController.resetInfo Car server: • TaximeterController.reset • DashboardController.hideReservationCode
R2: The system has to be able to understand when the car engine ignites	Car server: • PEJControllerCar	Car server: • PEJControllerCar.engineOn
R3: The system has to start charging the user when the car engine ignites	Car server: • TaximeterController	Car server: • TaximeterController.start
R4: The system has to display the current charge	TaximeterSystem	The taximeter system is configured to always show its current value when active
R5: The system has to identify when a car is parked in a safe area	Main server: • CarController Car server: • GPSController	Main server: • CarController.getLocation • CarController.isInSafeArea Car server: • GPSController.getLocation
R6: The system has to identify	Car server: • SeatSensorController	Car server: • SeatSensorController.isSeatUsed

when there is no one sit in the driver's seat		
R7: The system has to stop charging the user when the car is parked in a safe area and there is no one sat in the driver's seat	Car server: • SeatSensorController, • GPSController, • CarInfoController, • TaximeterController • PEJControllerCar	Car server: • SeatSensorController.isSeatUsed, • GPSController.getLocation, • CarInfoController.isEngineOn, • TaximeterController.stop • TaximeterController.getVaule, • PEJControllerCar.rideFinished

- [G4] *Users can register to the system and have their personal area.*

REQUIREMENT	COMPONENTS	DESCRIPTION
R1: The system has to provide log-in functionalities to the users: • The user should be able to check his/her active reservations	Main server: • LoginController, • UserController User browser/app: • PEJControllerUser	Main server: • LoginController.login, • UserController.findUser User browser/app: • PEJControllerUser.login • PEJControllerUser.getActiveReservations
R2: The system has to provide sing-up form to users: • The system has to check that there are not two users with the same username • The system has to store the password and personal information of every user • The system has to provide the possibility to enter a payment method	Main server: • SignupController, • UserController, • PaymentController User browser/app: • PEJControllerUser	Main server: • SignupController.signup, • UserController.findUser, • PaymentController.setPaymentInfo User browser/app: • PEJControllerUser.signup

• The system has to check if the payment method provided by the user is valid and usable		
R3: The system has to provide the possibility to change personal information or payment methods even after the registration	Main server: • UserController, • PaymentController User browser/app: • PEJControllerUser	Main server: • UserController.findUser, • PaymentController.setPaymentInfo User browser/app: • PEJControllerUser.setPaymentInfo
R4: The system has to provide the possibility to each user to see the personal "Reservation History"	User browserapp: • PEJControllerUser Main server: • UserController	User browser/app: • PEJControllerUser.getReservationHistory Main server: • UserController.getReservationHistory

- [G5] *Virtuous behaviours by users should be incentivized.*

REQUIREMENT	COMPONENTS	DESCRIPTION
R1: The system has to apply a discount of 10% on the final bill if there were at least three passengers on the last ride: • The system has to identify and store how many passengers there were on the car in the last ride	Main server: • ReservationController, • CarController Car server: • SeatSensorController	Main server: • ReservationController.applyDiscounts • ReservationController.applyDiscount, • CarController.getNumOfPassengers Car server: • SeatSensorController.numOfSeatsUsed
R2: The system has to apply a discount of 20% on the final bill if the car is left with at least 50% of battery level • The system has to be able to identify the battery level of	Main server: • ReservationController, • CarController Car server: • CarInfoController	Main server: • ReservationController.applyDiscounts • ReservationController.applyDiscount, • CarController.getBatteryLevel Car server: • CarInfoController.getBatteryLevel

the car		
R3: The system has to apply a discount of 30% on the final bill if the car is left plugged-in in a Special Safe Area: • The system has to identify if the car is plugged-in	Main server: • ReservationController, • CarController Car server: • CarInfoController	Main server: • ReservationController.applyDiscounts • ReservationController.applyDiscount, • CarController.getLocation • CarController.isPlugged Car server: • CarInfoController.isPlugged
R4: The system has to apply an extra-charge of 30% on the final bill if the car is left at least 3Km from the nearest Special Safe Area and the battery level is less than 30%: • The system has to be able to calculate the distance between the actual position of the car and the nearest Special Safe Area	Main server: • ReservationController, • CarController Car server: • GPSController • CarInfoController	Main server: • ReservationController.applyDiscounts • ReservationController.applyDiscount, • CarController.getLocation Car server: • GPSController.getLocation • CarInfoController.getBatteryLevel

7 Effort spent

We managed to distribute the workload fairly between days and team members in a way that allowed us to finish a few days before the deadline and have time for an accurate check in the last days.
The total amount of time required to build this document is about 24 hours for Chiusano Fabio and 20 hours for Riccardo Cattaneo.

8 References

The tools used for this document are:

- Microsoft Word: for text editing;

- Github, Git: for version control and synchronization between team members;
- LucidChart.com : to create Sequence UML Diagrams;
- Draw.io : to create ER Diagram;
- StarUML: to create other UML Diagrams;
- Sketch: to create mockups;
- Flinto: to create a prototype app.