

# Design Document

Simone Mosciatti & Sara Zanzottera

December 11, 2016

# Contents

1	Introduction	3
1.1	Purpose . . . . .	3
1.2	Scope . . . . .	3
1.3	Definitions, Acronyms, Abbreviations . . . . .	3
1.4	Document Structure . . . . .	5
1.5	Reference Documents . . . . .	5
2	Architectural Design	6
2.1	Design Process Description . . . . .	6
2.2	Goals Analysis . . . . .	6
2.3	Interfaces Design . . . . .	10
	Notifier . . . . .	16
2.4	Communication Design . . . . .	17
	Physical structure . . . . .	17
	Communication Strategy . . . . .	18
	Interactions with 3rd-parties . . . . .	19
	Communication Protocols . . . . .	19
2.5	Components Design . . . . .	20
2.6	Runtime View . . . . .	25
	Register and Login . . . . .	25
	Lookup and Book . . . . .	26
	Ride . . . . .	28
	Issue Management . . . . .	30
2.7	Deployment View . . . . .	31
2.8	Selected Technologies . . . . .	32
	RESTful API . . . . .	32
	MQTT . . . . .	32
	Nginx . . . . .	32
	Database . . . . .	32
3	Algorithm Design	33
4	User Interface Design	34
4.1	Mockups . . . . .	34
4.2	UX Diagrams . . . . .	34
4.3	BCE Diagrams . . . . .	36
5	Requirements Traceability	38
6	Conclusions	40
6.1	Tools used . . . . .	40
6.2	Hours of work . . . . .	40

# 1 Introduction

## 1.1 Purpose

This Design Document aims to provide to everyone involved in the actual development of the application specific insights about the structure of PowerEnJoy, its architecture's details, the design patterns we chose to implement, but also some details about its high level components, their interactions and general behavior.

## 1.2 Scope

PowerEnJoy is a digital management system for car sharing that exclusively employs electric cars to provide its service. The system provides all the functionalities normally provided by a car sharing service: registering to the service, find the location of nearby available cars, reserve cars up to a short amount of time, unlock the chosen car once found, ride it and then park it in a safe area, when it will be automatically locked and the fee paid.

In addition, the system gives bonuses and penalties in term of discounts or over-prices depending on the behavior of the user, in order to promote virtuous behaviors.

PowerEnJoy is therefore a inherently distributed system, based on a central server interactions with many distributed nodes. In detail the system can be divided into four main parts:

- a public app, used by customers to access the service
- a centralized backend that provides the service
- the cars' onboard system, that communicates only with the centralized backend
- a reserved frontend, used exclusively by the staff members to better organize their job

All these four components will be examined in more detail in the subsequent sections of the document.

## 1.3 Definitions, Acronyms, Abbreviations

**RASD** Requirements and Specification Document.

**DD** Design Document.

**User** A customer of PowerEnJoy using the service.

**Staff Operator** An employee of PowerEnJoy which takes care of the cars.

**Ride** The action of getting onboard of a PowerEnJoy car, start its engine, drive to destination and park.

**Running Time** The time an user spends using the PowerEnJoy service.

**Issue** Any problem a car may incur in, or a user may face while using the service.

**Nearby Cars** Cars located within a maximum distance to a specific position.

**Nearby Issues** Issues that are affecting cars close to a specific position.

**Booking (Reservation)** The act to reserve a car for a limited amount of time for future use by a user.

**Reservation's maximum time** The maximum amount of time a car can be reserved.

**Driver** Whoever is driving a regularly booked PowerEnjoy car.

**Passenger** Whoever is inside a PowerEnjoy car but is not the driver.

**Driving License** The state's issued driving license of the user.

**Notification** A form of communication where the user is actively notified of some event.

**Issue Report** An incoming notification that states a car incurred in an issue.

**Fine** A fine issued by the local law enforcing officers to a user while driving a PowerEnjoy car.

**Pending Bills** Bills that a user still needs to pay to PowerEnjoy .

**Safe Area** A parking area, predefined by the company, where it is possible to safely park the cars of the PowerEnjoy fleet.

**Battery Charge** The amount of charge that is kept inside the car's battery.

**Charging Station** Dedicated areas where it is possible to plug the PowerEnjoy cars to charge their batteries.

**Car's Onboard System** The control system of the car that is able to exchange data with the central system and to reevaluate operation parameters.

**Customer's App** An implementation of the system frontend tailored to the need of the customers.

**Operator's App** An implementation of the system frontend tailored to the need of the staff.

**Central System** The central system for PowerEnjoy . All the commands and all the data are streamed, analyzed and used here.

**Credentials** Pair {Username, Password} necessary to access the PowerEnjoy system.

**GPS** : Global Positioning System is a global navigation satellite system (GNSS) that provides location and time information in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

**System's Frontend** The interface provided to the user of the PowerEnJoy system.

**System's Backend** The whole technical infrastructure necessary to PowerEnJoy .

## 1.4 Document Structure

### 1. Introduction

This sections aims to explain the purpose and the scope of the document, introducing the reader to subsequent sections of the document itself.

### 2. Architectural Design

This sections will explain the main architectural decision we made.

### 3. Algorithm Design

In this section we focus on the most critical code section and we provide an in-depth analysis of how they should be structured, eventually providing pseudocode for them.

### 4. User Interface Design

In this section we carry on the UX design with the help of UX and BCE diagrams, eventually completing them with updated and extended application mockups.

### 5. Requirements Traceability

In this section we map the requirements stated in the RASD to the actual component or processes that fulfill these requirements.

### 6. Conclusions

In this section we enumerate the tools we used to redact this document, the hours of work spent by each group member and the (eventual) revision history of the document itself.

## 1.5 Reference Documents

- *Assignments AA 2016-2017.pdf* (Assignments document given by the teacher)
- *Sample Design Deliverable Discussed on Nov. 2.pdf* (Sample document provided by the teacher)

## 2 Architectural Design

The overall design process has been carried in a bottom-up approach, starting from the analysis of goals and requirements moving upwards to the definition of the higher level components of the system. In the following sections we provide more details on the designed architecture.

### 2.1 Design Process Description

The overall design process starts from the analysis of the goals.

Taking forward the considerations made in the RASD, we analyse the interface between the world and the machine: given the list of goals and requirements, we identify what interfaces the system need to provide to the users. If those interfaces would have been already implemented our work would have been completed, however such is not the case, and those interfaces require some functionalities.

Once all the necessary functionalities have been identified we organize them into high-level components taking care to respect the “Single Responsibility” principle in order to provide highly decoupled and reusable components. Again, if those functionalities were already working our works would have been done, however these components need to be actually implemented and deployed into physical machines.

At this point we proceed to identify the physical nodes of our system and where our components should be deployed in order to fullfil their functionalities. We also analyze what communication mechanism to use between the node.

Finally we proceed to deploy the components into the physical node.

To double check the correctness of the overall design we produce several sequence diagram showing for the uses cases what functionality is involved and what functionality is called.

The rest of this section follows this flow, starting from the goals analysis and finally providing the overall design.

The next section will show the uses cases.

### 2.2 Goals Analysis

In this part of the document we analyze Goals as defined in RASD, in order to list and describe in detail which interactions between the world and the machine will be performed and how to provide them.

These are named **SB/FunctionalityName** in order to highlight that these functions are related to the system’s boundaries. Indeed we know already that some goals are specific for customers, some are specific for the staff operators, and some are shared. In order to highlight this difference, we decided to enforce the following naming schema:

- SB/ALL/FunctionalityName for shared functionalities
- SB/CUST/FunctionalityName for customer’s reserved functionalities
- SB/STAFF/FunctionalityName for staff’s reserved functionalities

Many of the following use cases requires some specific functionalities to be provided by the backend. These are highlighted and named **Sy/FunctionalityName**, where “Sy” indicate an abstract system that we are going to model in detail in the next paragraphs.

#### **SB/ALL/Registration**

**Description** Users can register to the system.

**Requires**

- Sy/Register
- Sy/Validate

#### **SB/ALL/Login**

**Description** Users can log into the system providing their email and password and logout from the system.

**Requires**

- Sy/Login
- Sy/Logout

#### **SB/CUST/Lookup**

**Description** Users can look for cars near them or near a specific position and range.

**Requires**

- Sy/GeoLocationCars
- Sy/PositionUser

#### **SB/CUST/Book**

**Description** Users can book cars to use within the next hour.

**Requires**

- Sy/Book

#### **SB/CUST/Unbook**

**Description** Users can remove a previously made reservation.

**Requires**

- Sy/Unbook

#### **SB/ALL/Unlock**

**Description** Users can unlock a nearby car that was previously booked.

**Requires**

- Sy/PositionUser

- Sy/PositionCar
- Sy/Unlock

### **SB/CUST/Ride**

**Description** Users can ride a reserved car.

#### **Requires**

- Sy/StartRide
- Sy/EndRide
- Sy/ValidateLicense
- Sy/GeoLocationAreas
- Sy/Lock

### **SB/ALL/SafeAreas**

**Description** Users can locate safe parking areas while driving.

#### **Requires**

- Sy/GeoLocationAreas

### **SB/CUST/UnsafeParking**

**Description** The system reacts to unsafe parking. It first tells users they are going to be fined if they leave the car unsafely parked, then it fines users who leaves the car in an unsafe area.

#### **Requires**

- Sy/EngineOff
- Sy/Lock
- Sy/UnsafeParkNotification
- Sy/UnsafeParkFine

### **SB/ALL/PowerStation**

**Description** Users can locate and use charging station.

#### **Requires**

- Sy/GeoLocationAreas
- Sy/Plugged
- Sy/CarPluggedNotification

### **SB/CUST/Charge**

**Description** Users are charged a fee at the end of the ride or if a reservation expires.

#### **Requires**



- Sy/SendFee
- Sy/BookExpire
- Sy/CalculateUnsafeParkingFee
- Sy/CalculateRideFee
- Sy/CalculateExpireBookFee

#### **SB/CUST/Payment**

**Description** Users can pay bills through the app and set their default payment method.

##### **Requires**

- Sy/MakePayment
- Sy/SetPaymentMethod

#### **SB/ALL/ReportIssues**

**Description** Users can report issues to the system.

##### **Requires**

- Sy/ReportIssue

#### **SB/STAFF/FindIssues**

**Description** The staff can locate cars that need their intervention.

##### **Requires**

- Sy/GeoLocationIssues

#### **SB/STAFF/Support**

**Description** The staff can identify and solve car's issues.

##### **Requires**

- Sy/Lock
- Sy/Unlock
- Sy/TakeChargeIssue
- Sy/SolveIssue
- Sy/GiveUpIssue

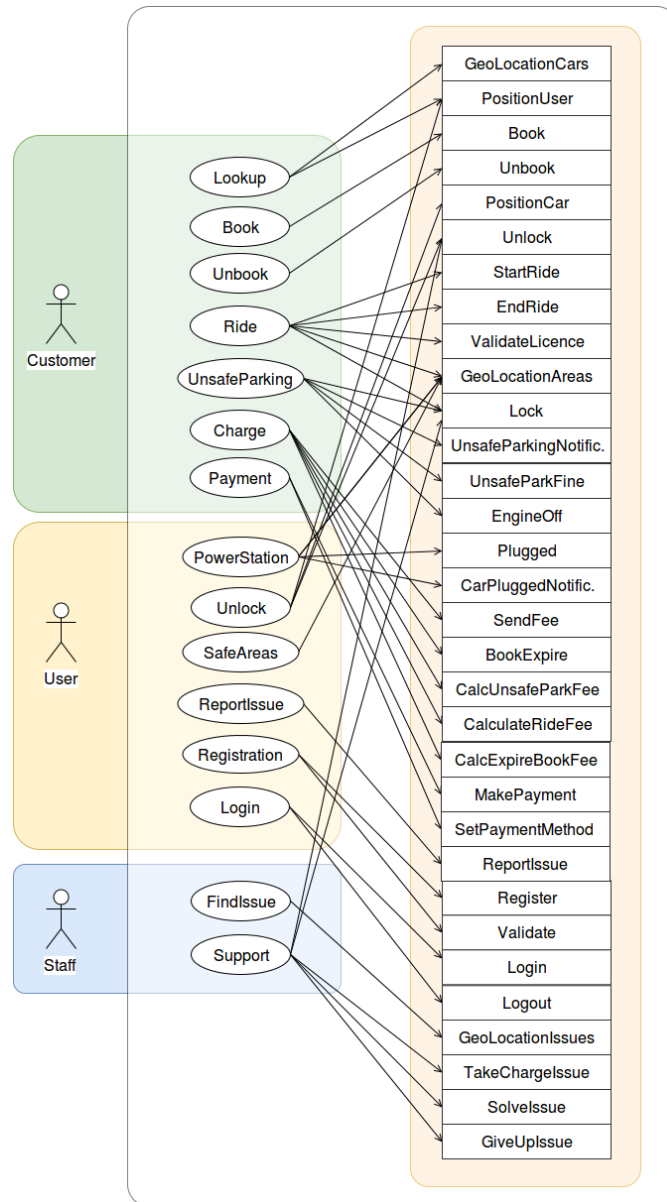


Figure 1: Graphical visulaization of the UI divided by tipology of users.

## 2.3 Interfaces Design

Now we gather all the functionalities we just described and we organize them into higher-level interfaces, being careful at respecting the responsibilities given to each one of them.

In order to clarify how previously defined “Sy/FunctionalityName” maps with the interface’s specific functions, we used a notation **INTERFACE/FunctionalityName** ⇔

**Sy/FunctionalityName.** The functionality name can vary, according to the context.

## **USER\_MANAGER**

**Responsability** Manages the users.

**USER/Register**  $\Leftrightarrow$  **Sy/Register**

**Responsability** Registers a new user into the system.

**Input** Information from the user such as:

- First name
- Last name
- Identity ID
- Password
- Email
- License ID
- Credit card informations: credit card number, control code, expiry date, owner, etc.

**Output** The ID of the newly created user.

**USER/Validate**  $\Leftrightarrow$  **Sy/Validate**

**Responsability** Validate the information provided by the user, it makes sure that the user is not already registered into the system, that the License is valid as well the credit card information.

**Input** Information from the user such as:

- First name
- Last name
- Password
- Email
- License ID
- Credit card informations: credit card number, control code, expiry date, owner, etc.

**Output** Confirm correctness and validity of the informations.

**USER/Login**  $\Leftrightarrow$  **Sy/Login**

**Responsability** Allows users to log into the system.

**Input** Email (considered a unique user ID) and password.

**Output** A session key, meaning that the user is logged into the system.

**USER/Logout**  $\Leftrightarrow$  **Sy/Login**

**Responsability** Allows users to logout the system.

**Input** The login session key.

**Output** The session key is invalidated.

**USER/SetPaymentMethod**  $\Leftrightarrow$  **Sy/SetPaymentMethod**

**Responsability** Update user's information about the preferred payment method.

**Input** The ID of the user and new payment informations.

**Output** The payment method data related to this user is updated.

## **GEOLOCATION**

**Responsability** Locates elements, points and areas of interest around a specific coordinate. "Search" service for elements of interest.

### **GEOLOCATION/AvailableCars $\Leftrightarrow$ Sy/GeoLocationCars**

**Responsability** Retrives the position of available cars.

**Input** Search parameters such as:

- Geographical coordinates of the center of the search range (latitude and longitude as provided by GPS sensors)
- Maximum walking distance from the specified position
- Other search settings, like minimum battery level, etc.

**Output** A set of available cars matching the search parameters.

### **GEOLOCATION/Areas $\Leftrightarrow$ Sy/GeoLocationAreas**

**Responsability** Retrives the position of areas of interest, such as power stations and safe parking areas.

**Input** Geographical coordinates of the center of the search range (latitude and longitude as provided by GPS sensors) and a search radius.

**Output** A set of areas of interest inside the circle of radius provided centered on the coordinates provided.

### **GEOLOCATION/Issues $\Leftrightarrow$ Sy/GeoLocationIssues**

**Responsability** Retrives the position of cars with some issues.

**Input** Search parameters such as:

- Geographical coordinates of the center of the search range (latitude and longitude as provided by GPS sensors)
- Radius of the search
- Issue type, Exeption status, and other similar search settings.

**Output** A set of cars with issues matching the search parameters inside the circle of radius provided centered on the coordinates provided.

### **GEOLOCATION/IsSafeArea $\Leftrightarrow$ Sy/GeoLocationIssues**

**Responsability** Given the coordinates check if those coordinates are inside a safe area.

**Input** Coordinates.

**Output** A boolean indicating if the coordinates are inside a safe area.

## **POSITION**

**Responsability** Locates elements of interest given their ID. "Lookup" service for elements of interest.

### **POSITION/Car $\Leftrightarrow$ Sy/PositionCar**

**Responsability** Retrieves the position of a specific car.

**Input** The ID of the car.

**Output** The coordinates of the car.

**POSITION/User  $\Leftrightarrow$  Sy/PositionUser**

**Responsability** Retrieves the position of an user.

**Input** The ID of the user.

**Output** The coordinates of the user.

**POSITION/Areas**

**Responsability** Retrieves the position of an area of interest.

**Input** The ID of the area.

**Output** The coordinates of the area as a set of boundary points.

## **BOOKING\_MANAGER**

**Responsability** Manages reservations.

**BOOKING/Book  $\Leftrightarrow$  Sy/Book**

**Responsability** Books one available car.

**Input** The ID of the car and the ID of the user.

**Output** The car is booked and the ID of the reservation is provided.

**BOOKING/Unbook  $\Leftrightarrow$  Sy/Unbook**

**Responsability** Removes a reservation.

**Input** The ID of the user and the ID of the reservation.

**Output** The reservation is cancelled.

**BOOKING/Expire  $\Leftrightarrow$  Sy/BookExpire**

**Responsability** Removes an expired reservation and fines the related user.

**Input** ID of the reservation.

**Output** The reservation is cancelled and the user is fined.

## **CAR\_MANAGER**

**Responsability** Manages the interactions between users and cars.

**CAR/Unlock  $\Leftrightarrow$  Sy/Unlock**

**Responsability** Unlocks the car.

**Input** The ID of the car and the ID of the user asking to unlock.

**Output** The car is unlocked.

**CAR/ValidateLicense  $\Leftrightarrow$  Sy/ValidateLicense**

**Responsability** Confirms the scanned license is related to the user who booked the car.

**Input** The scanned image of the driving license and the ID of the booking

**Output** The car is unlocked.

**CAR/Lock  $\Leftrightarrow$  Sy/Lock**

**Responsability** Locks the car.

**Input** The ID of the car.

**Output** The car is locked.

#### **CAR/TurnOff $\Leftrightarrow$ Sy/EngineOff**

**Responsability** Turns off the engine of a car.

**Input** The ID of the car.

**Output** The car is turned off.

#### **CAR/Telemetry**

**Responsability** Retrieves real-time, updated informations about a car.

**Input** The ID of the car.

**Output** All the latest informations available about the car.

#### **CAR/SetStatus**

**Responsability** Sets the Exception status of a car to a new value.

**Input** The ID of the car and the new status.

**Output** The new status is set.

#### **CAR/GetDetails**

**Responsability** Returns all available data about a given car (such as battery level, charging status, etc.).

**Input** The ID of the car.

**Output** All available details about the car

### **RIDE\_MANAGER**

**Responsability** Manage the rides.

#### **RIDE/Start $\Leftrightarrow$ Sy/StartRide**

**Responsability** Start counting the time of the ride.

**Input** The ID of the user and the ID of the booking.

**Output** The ID of the ride.

#### **RIDE/End $\Leftrightarrow$ Sy/EndRide**

**Responsability** Stop counting the time of the ride.

**Input** The ID of the ride.

**Output** Stop the count of time for the ride.

#### **RIDE/FindRides**

**Responsability** Retrieve the list of rides done with a specific car in a defined time range.

**Input** The ID of the car and a time range.

**Output** The list of rides performed with that car in that time range.

### **BILLING\_MANAGER**

**Responsability** Manages all the fees.

**BILL/CalculateRideFee  $\Leftrightarrow$  Sy/CalculateRideFee**

**Responsability** Calculates the amount of a riding fee.

**Input** The ID of the ride.

**Output** The ID of the fee with a complete total which include eventual discounts or overprices.

**BILL/CalculateExpireBookFee  $\Leftrightarrow$  Sy/CalculateExpireBookFee**

**Responsability** Calculates the amount of a expired prenotation fee.

**Input** The ID of the booking.

**Output** The ID of the fee referred to the expired prenotation.

**BILL/CalculateUnsafeParkingFine  $\Leftrightarrow$  Sy/CalculateUnsafeParkingFee**

**Responsability** Requires user to pay a fine for an unsafe parking.

**Input** The ID of the ride which left the car unsafely parked.

**Output** The ID of the fee referred to the fine for unsafe parking.

**BILL/Pay  $\Leftrightarrow$  Sy/MakePayment**

**Responsability** Requires user to pay a specific fee.

**Input** The ID of the user and the ID of a fee.

**Output** The fee is paid.

**ISSUE\_MANAGER**

**Responsability** Manages car's issues.

**ISSUE/New  $\Leftrightarrow$  Sy/ReportIssue**

**Responsability** Rise a new issue.

**Input** ID of the car, ID of the user raising the issue, a title and a description of the issue.

**Output** The ID of the reported issue.

**ISSUE/TakeCharge  $\Leftrightarrow$  Sy/TakeChargeIssue**

**Responsability** Allows operators taking in charge of a particular issue.

**Input** The ID of the issue, the ID of the operator.

**Output** The operator is now responsible for the issue.

**ISSUE/Solve  $\Leftrightarrow$  Sy/SolveIssue**

**Responsability** Allows operators to close an issue marking it as Solved. The system is responsible of allowing this operation only if it can confirm the issue has been solved.

**Input** The ID of the issue, the ID of the operator.

**Output** The issue is set as Solved, or set as Closed and another new issue is opened.

**ISSUE/GiveUp  $\Leftrightarrow$  Sy/GiveUpIssue**

**Responsability** Allows operators to give up over an issue.

**Input** The ID of the issue, the ID of the operator.

**Output** The issue is no more related to the specified operator and available for others to be took in charge.

## NOTIFIER

**Responsability** This component takes care to notify user of events.

**NOTIFY/Notify**  $\Leftrightarrow$  {Sy/SendFee, Sy/UnsafeParkNotification, Sy/CarPluggedNotification}

**Responsability** Notify the user of some event, it may also prompt the user to acknowledge some fact or complete some action.

**Input** A notification object.

**Output** The notification is show to the user and the user may be prompted to do some action.

---

In this part of the design we made some choices where alternative approach may have been considered as well. We now motivate some of those choices.

### Notifier

To make the user able to receive notification from server we could use several strategies. In a static world, where the product doesn't change, all of the choice are equivalent. However, if we put ourselves in a dynamic situation where the goals and requirements of the software evolve, our choice are more critical and must be justified.

Before deciding for an isolated component responsible only to forward a notification to the user, we evaluated other options for the notifier:

1. Each component could have its own NOTIFIER: eg. instead of NOTIFY/Notify(Notification) we could have something like CAR/NotifyUnsafePark(). Indeed, as more components need to notify the user, more and more different versions of the same functionality will probably be reimplemented in different part of the code base: a problem we would like to avoid.
2. Each kind of notification could have its own function: eg. instead of NOTIFY/Notify(Notification) we could have something like NOTIFY/UnsafePark(). We believe this choice would make the overall developing time too slow, since every new functionality which requires a notification would have to be implemented into NOTIFY, making it unstable.

We believe that our choice allows a very fast development cycles while maintaining an high decoupling between different components.



## 2.4 Communication Design

Once we have identified the abstract interfaces that, together, provide the functionalities of our system, we are going to design the components which implements them.

In order to actually implements the components we first need to know where those components are deployed, and the form of communication used between or inside them.

### Physical structure

As for the RASD (section 2: Overall Description), the system is be divided into four parts:

- the **customer's app**, used by customers to access the service.
- the **staff's app**, used exclusively by the staff members to better organize their job.
- the **main server**, a centralized backend that provides the service.
- the **cars' onboard system**, that communicates only with the centralized backend.

As said in the RASD, also in this Design Document we are using the term "App" to refer to a User Interface deployed on a smartphone, and we will keep such terminology for sake of clarity. However it must be clear that a whole spectrum of technologies could be used to actually implement the user Interface, not only mobile application.

Also, for the scope of this analysis we often consider the customer's app and the staff's app as a single entity, called simply "app".

At a first glance, the above elements can be organized in a two-tier Client-Server architecture as follows:

- Tier 1, the main server, which handles Application Logic and Data Management.
- Tier 2, comprising mobile apps and cars, hosts the User Interface.

From now on, we design the system basing on these fundamentals elements, defining their roles and interactions.

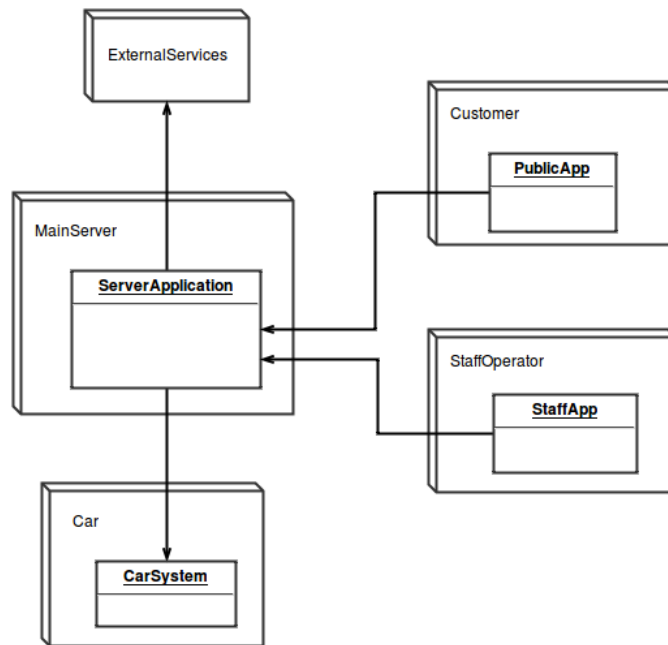


Figure 2: Higher level components of the system

### Communication Strategy

Now that our nodes are defined, we need to design the communication protocols between them.

It would have been possible to make each node communicate with each other node: however this approach raises some concerns about the security of the system, especially for the communication between the users and the cars (see below).

Of course a well done, and a well tested system should not raise these concerns, however real word engineering is constrained and the authors believe that a more defensive and conservative approach is more suitable.

Following these consideration we decided that would be more maneagable to have only two communication channels:

- one between the main server and the user apps;
- another one between the main server and the cars.

We decide to completely avoid all kinds of communication between the cars and the apps.

The main server always acts as an intermediary for app-to-car and car-to-app communications, in order for the system to have full control on these otherwise completely hidden interactions.

## Interactions with 3rd-parties

We decided to prevent any access to third-part services to every component but the main server, mainly for security reasons.

The main server will expose the necessary APIs to apps and cars to allow them retrieving all the informations they need without having them communicating independently with any external service.

## Communication Protocols

The communication between the main server and the user will employ a simple **Client-Server** approach which seems to naturally fit the domain space, it is a well know industry standard and is widely used.

On the other hand, communications between the server and the fleet is clearly more suitable for a **Publisher-Subscriber** pattern. The car has to communicate very often a lot of valuable information to the main server, and a PubSub pattern achieves high throughput and low latency. Even if the Pub-Sub pattern may not be so widely used and know, our experience suggested that the trade-off is well worth.

Moreover, we decided to model the most part of external services as Web services, so the server will communicate with them using a **Service-Oriented** approach.

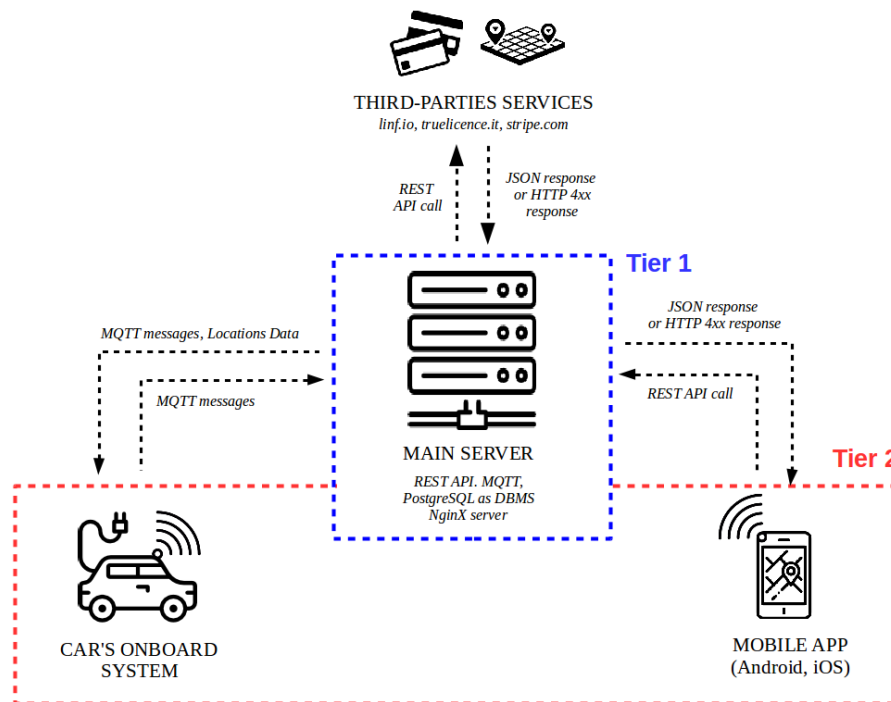


Figure 3: Communication design of the process (anticipating some technology choices we are going to justify in next sections).

## 2.5 Components Design

Up to this point we defined the logical components of the system in terms of interfaces and communication protocols. We now proceed deploying them on the physical nodes identified above in order to end up with a complete, high-level logical architecture for our system.

### **USER\_MANAGER**

Considering that all its functionalities are exposed as API by the server to the apps, the component is deployed on the server entirely.

### **GEOLOCATION**

Considering that all its functionalities are exposed as API by the server, the component is deployed on the server entirely.

### **POSITION**

This component need the participation of both users and cars. Thus the component is deployed on all the three nodes: user apps, cars and the main server.

**POSITION/Users** is deployed as **USER\_POSITION** on the user apps.

**POSITION/Cars** is deployed as **CAR\_POSITION** on the cars.

**POSITION/Areas** is deployed as **POSITION\_MANAGER** on the main server, as it acts as a dispatcher for all incoming requests, keeps a cache of the retrieved data, and so on.

### **BOOKING\_MANAGER**

Considering that all its functionalities are exposed as API by the server, the component is deployed on the server entirely.

### **CAR\_MANAGER**

This component requires a deeper analysis, as its interfaces are exposed by different elements. In details:

- CAR/Unlock is exposed by Server to Apps
- CAR/GetDetails is exposed by Cars to Server
- CAR/ValidateLicense is exposed by Server to Cars
- CAR/Lock is exposed by Cars to Server
- CAR/TurnOff is exposed by Cars to Server
- CAR/Telemetry is exposed by Cars to Server
- CAR/SetStatus is exposed by Cars to Server

Thus we create two **CAR\_MANAGER** subcomponents:

### **REMOTE\_CAR\_MANAGER**

Comprises all the functions exposed by the server:

- CAR/Unlock
- CAR/GetDetails
- CAR/ValidateLicense

#### **LOCAL\_CAR\_MANAGER**

Comprises all the functions exposed by the cars:

- CAR/Lock
- CAR/TurnOff
- CAR/Telemetry
- CAR/SetStatus

#### **RIDE\_MANAGER**

Considering that all its functionalities are exposed as API by the server to the customer's apps, the component is deployed on the server entirely.

#### **BILLING\_SYSTEM**

Considering that all its functionalities are exposed as API by the server to the customer's apps, the component is deployed on the server entirely.

#### **ISSUE\_MANAGER**

Considering that all its functionalities are exposed as API by the server to the customer's apps, the component is deployed on the server entirely.

Indeed, ISSUE/Solve involves an interface that is exposed by the car: thus we define another subcomponent, **VALIDATE\_SOLVE**, to be deployed on the car to fulfill this function.

#### **NOTIFIER**

Considering that its functionality is exposed by the user app, this component is deployed entirely on the user app.

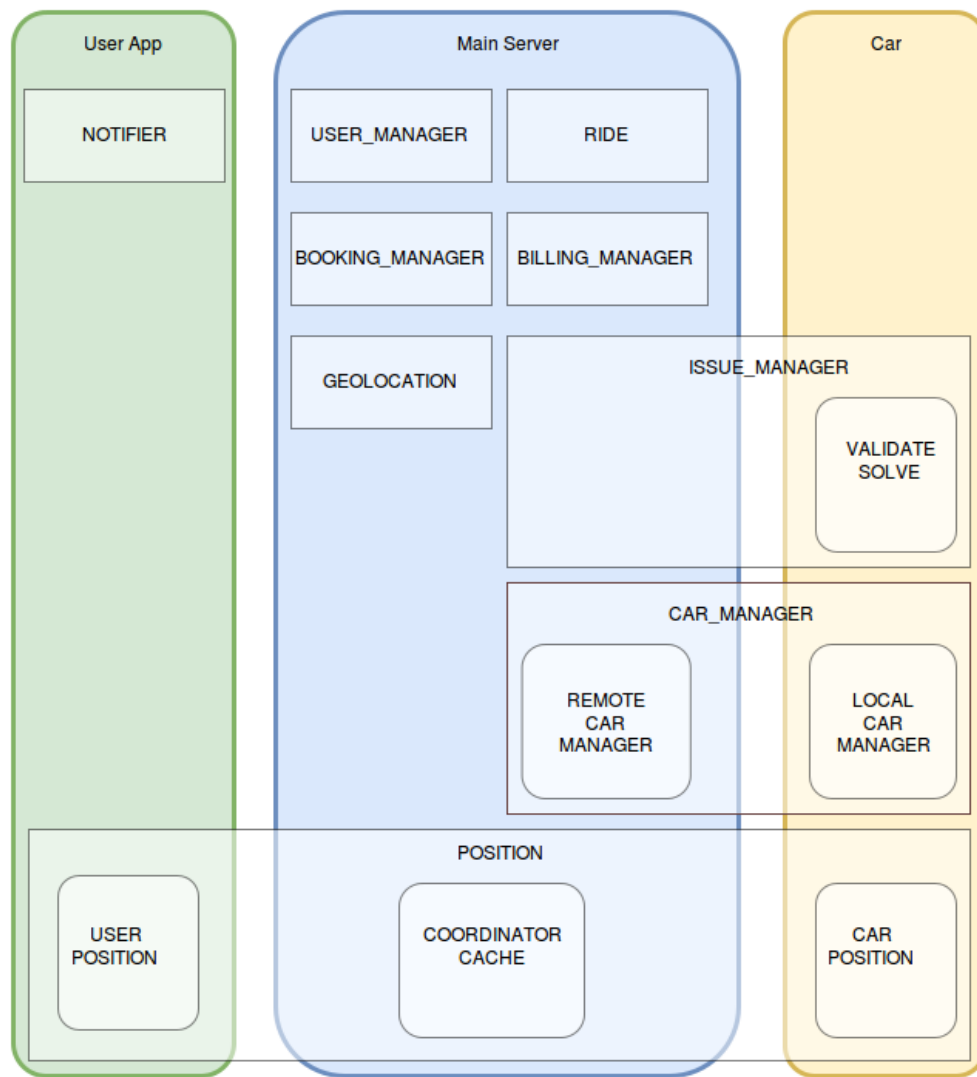


Figure 4: Graphical visualization of the abstract components needed.

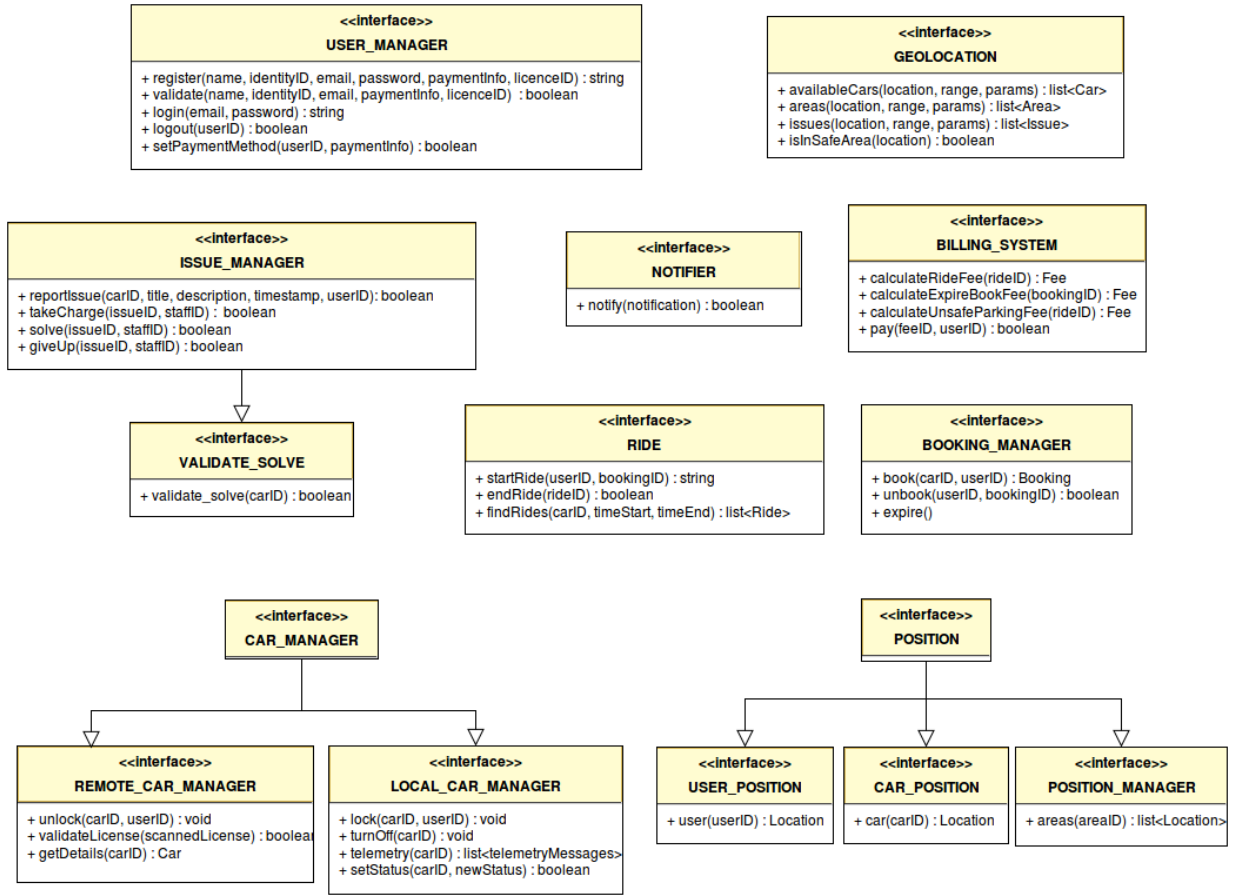


Figure 5: Diagram of the interfaces identified above.

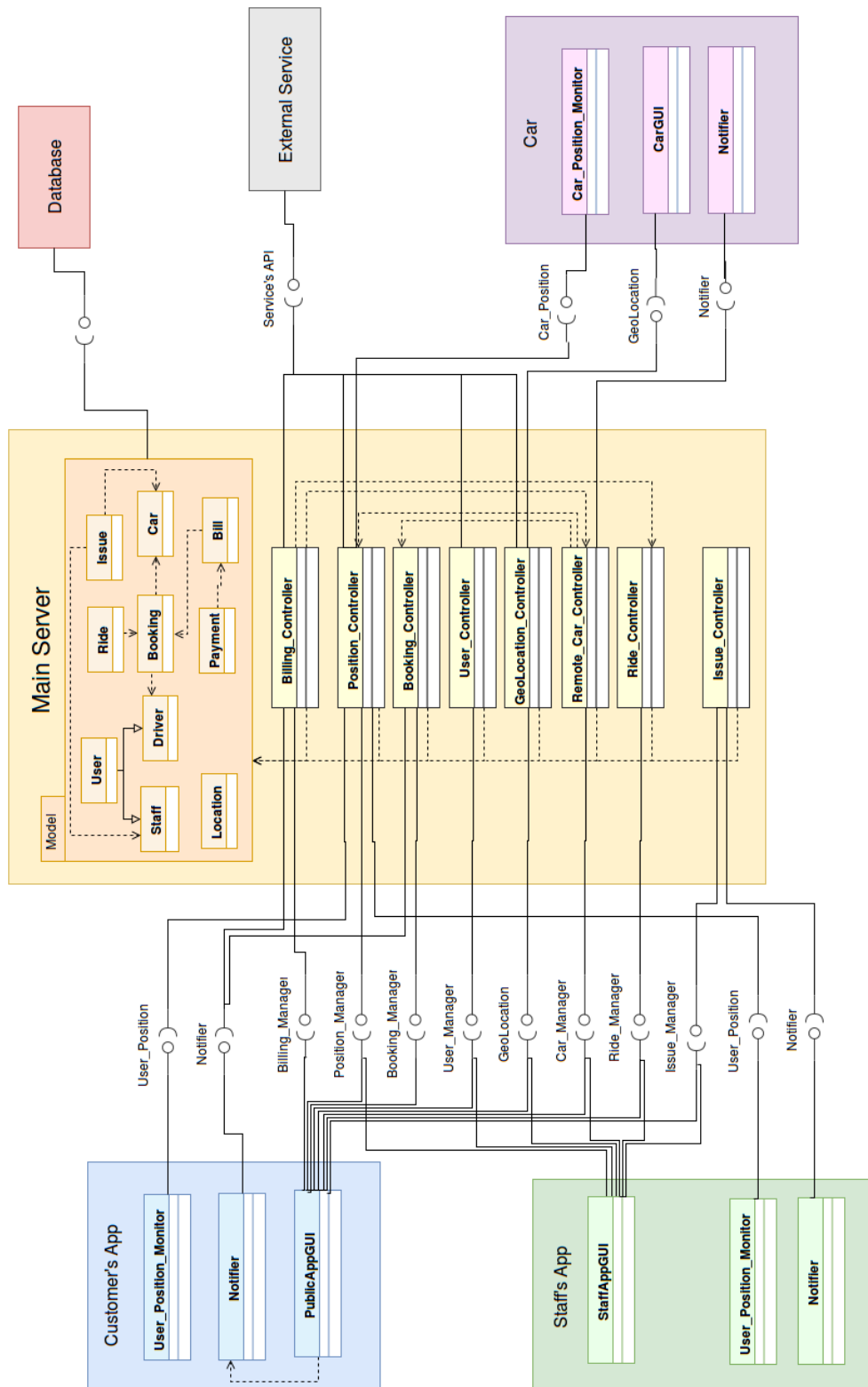


Figure 6: Components View Diagram of the entire system.



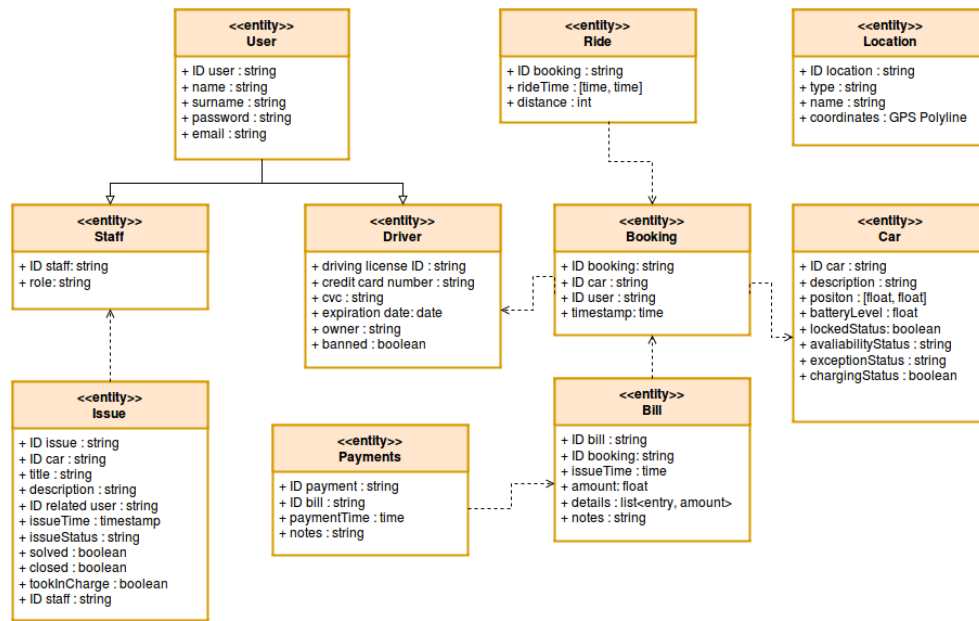


Figure 7: Model zoom of the Components View Diagram.

## 2.6 Runtime View

### Register and Login

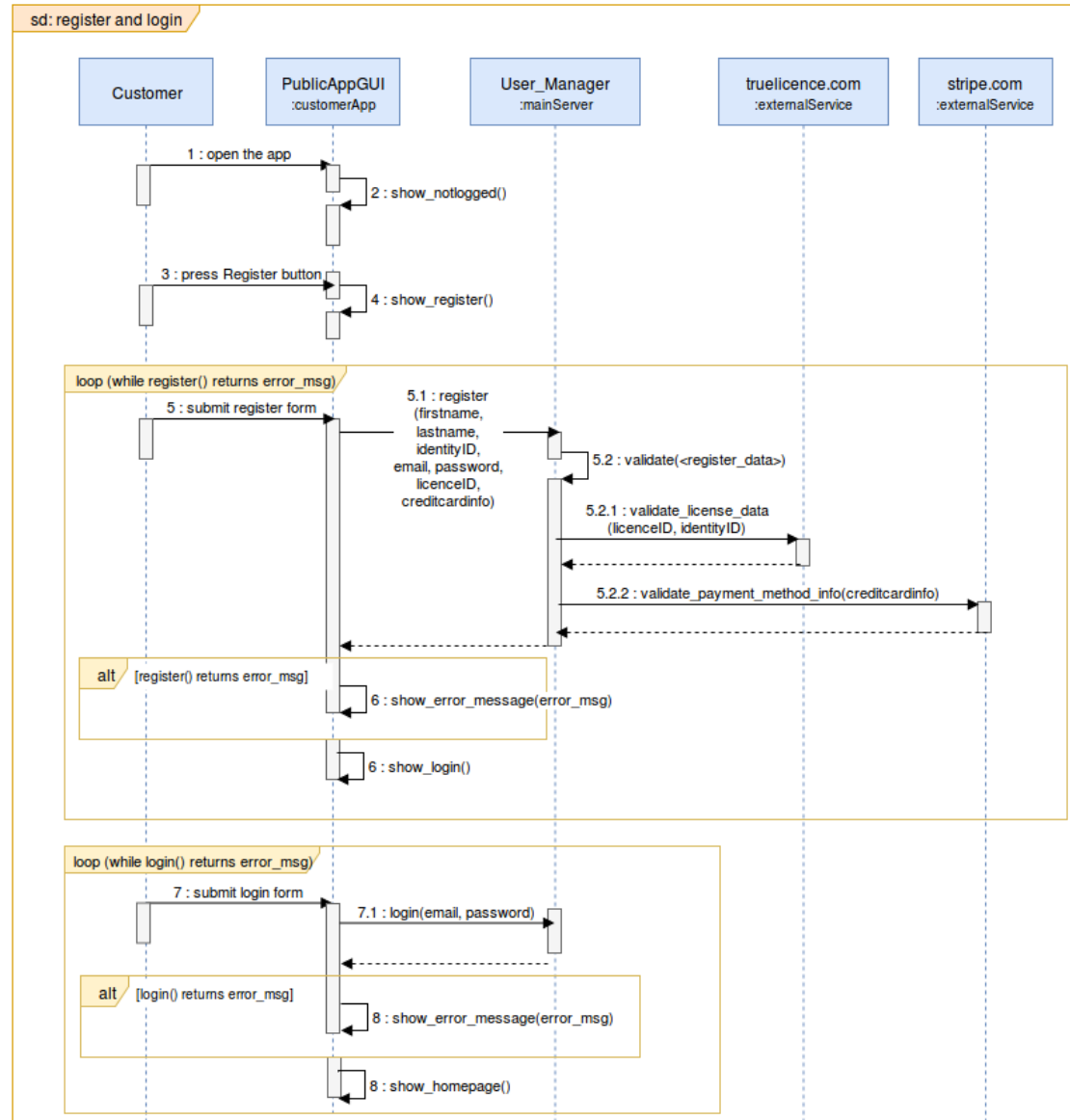


Figure 8: Sequence Diagram for Registration and Login process of customers.

Here we can clearly see how the responsibility of managing users is set to the **User\_Manager** component, fully deployed on the main server.

PublicAppGUI is simply responsible of gathering useful data and sending it to User\_Manager, which takes care of contacting external services if needed.

When the registration/login process is completed server-side, User\_Manager returns control to PublicAppGUI, that is in charge of informing the user about the result of their request and eventually redirected to another page (login after registration is completed successfully, homepage after login is completed successfully).

## Lookup and Book

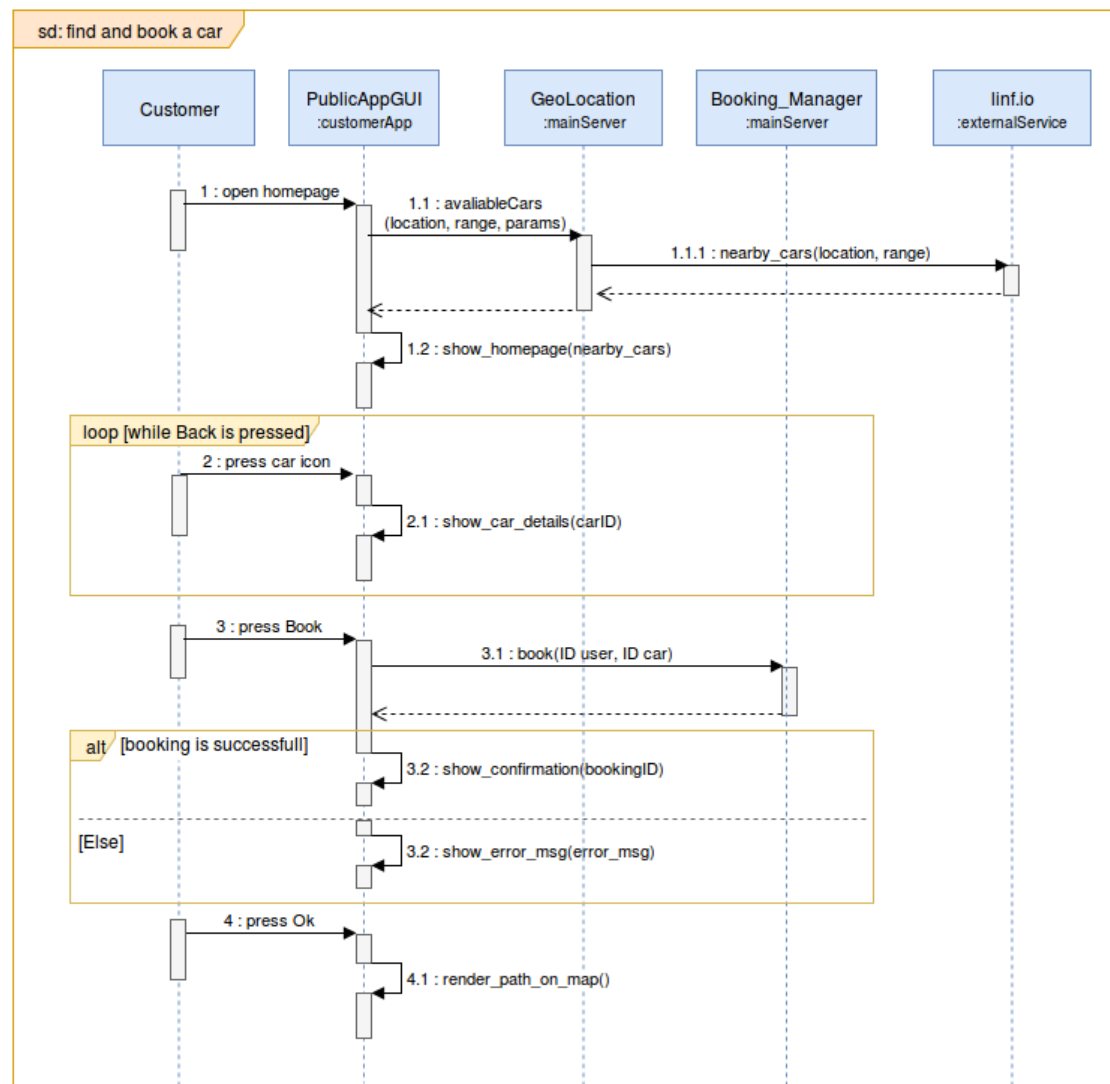


Figure 9: Sequence Diagram for Lookup and Booking process of customers.

This diagram shows clearly the two processes of looking for available cars and booking.

The first goal is achieved with the sole need of GeoLocation, deployed only server-side, that hides to the final user the interaction with linf.io, which is an external service.

The app is only responsible of rendering the informations into an interactive map the user can easily navigate.

Once on the map, users can navigate cars details without the need to connect anymore with the server (all the required data has been cached from the previous call) until they decide to book a car. At that moment, a call to `Booking_Manager` is issued, which handles the full procedure internally.

The app is responsible only to provide the user feedback about the successfullness of the operation once `Booking_Manager` returns.

Note that, after a booking is completed successfully, the user is redirected to the Bookings Page, because it is not possible for a user to issue multiple reservations at the same time.

## Ride

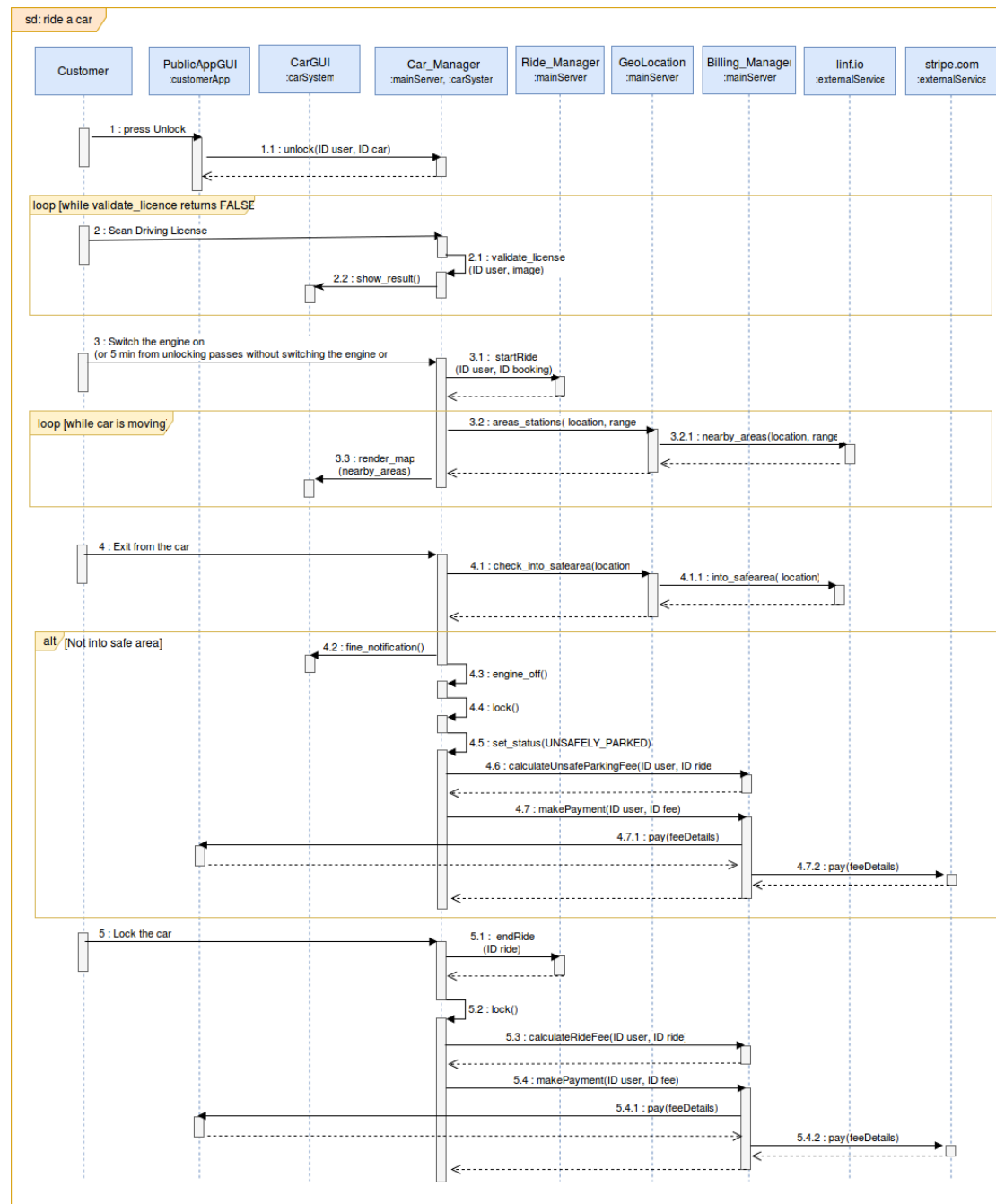


Figure 10: Sequence Diagram for the Riding process of customers.

This diagram describes what's probably the most complex interaction between users, cars and the main server.

The first step, unlocking the car, is straightforward: Car\_Manager receives a call from PublicAppGUI requiring the unlock, and Car\_Manager handles internally the process. Notice that, as Car\_Manager is deployed both on the car and on the main server, the call to Car\_Manager/unlock( IDUser, IDCar ) hides another internal process where the server communicates with the car.

Then the driver has to scan its driving licence in order to unlock the engine of the car. Again, Car\_Manager handles some communication between the car's system and the server internally, validate the licence and, if everything is fine, unlocks the engine (step not shown, as it is considered an internal call).

Once the engine is switched on, or 5 minutes from unlocking has passed (regardless of the engine status), Car\_Manager calls Ride\_Manager and starts a new Ride. The length of the ride is calculated from this moment until the Ride is closed (see later).

During the whole ride, CarGUI receives continuous updates from Car\_Manager about nearby safe areas and charging station: this data has been retrieved from GeoLocation, that in turn obtained them with a call to linf.io.

When the user exits from the car (regardless of the engine status), Car\_Manager reacts checking if the car has been left in a safe area or not calling GeoLocation/Check-IntoSafeArea( location ).

If not, a series of operations are performed. First, a notification is sent to CarGUI in order to notify the user and possibly have them back in the car. If the user does not come back within a short amount of time (a minute, for example), the engine is switched off, the car is locked, its Exception status is set to "Unsafely parked" and a fine is sent to the driver's app.

Otherwise, the system waits for the car to get locked. Once the locking request arrives, first the ride is ended with a call to Ride\_Manager/endRide( IDRide ) and the car is locked. Then the regular fee is calculated and the payment issued.

## Issue Management

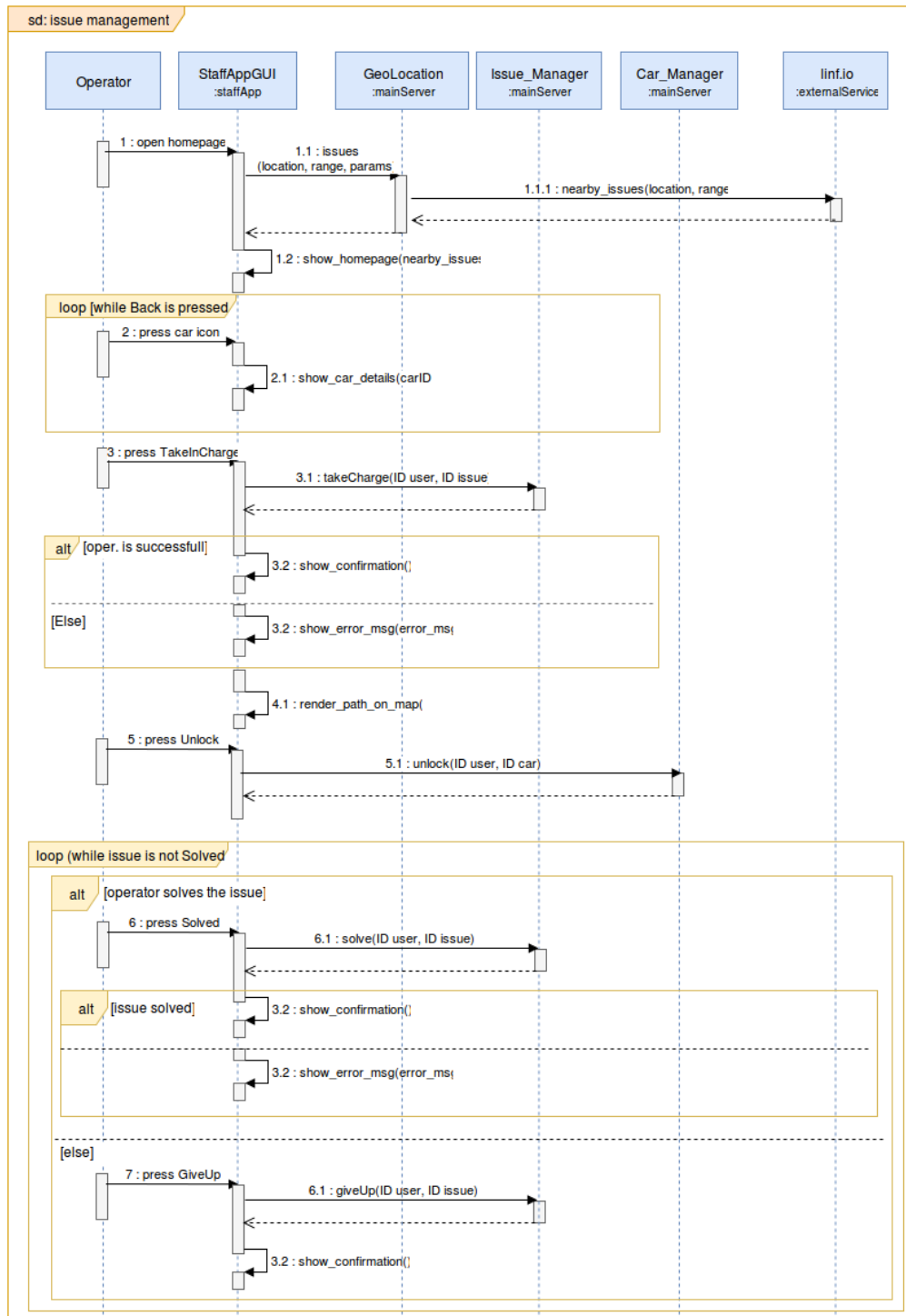


Figure 11: Sequence Diagram for the Issue Management process for staff operators.

In the above diagram we can see how the process of managing issues is carried on by staff operators.

The process share more or less the same workflow as the booking and unlock process for customers, plus some steps related specifically to issues.

It is evident how these additional steps are all managed by Issue\_Manager alone, as to highlight the single, coherent responsibility we assigned to components.

## 2.7 Deployment View

In order to better understand the technology we chosed to design this deployment diagram, refer to section 2.8: Selected Technologies

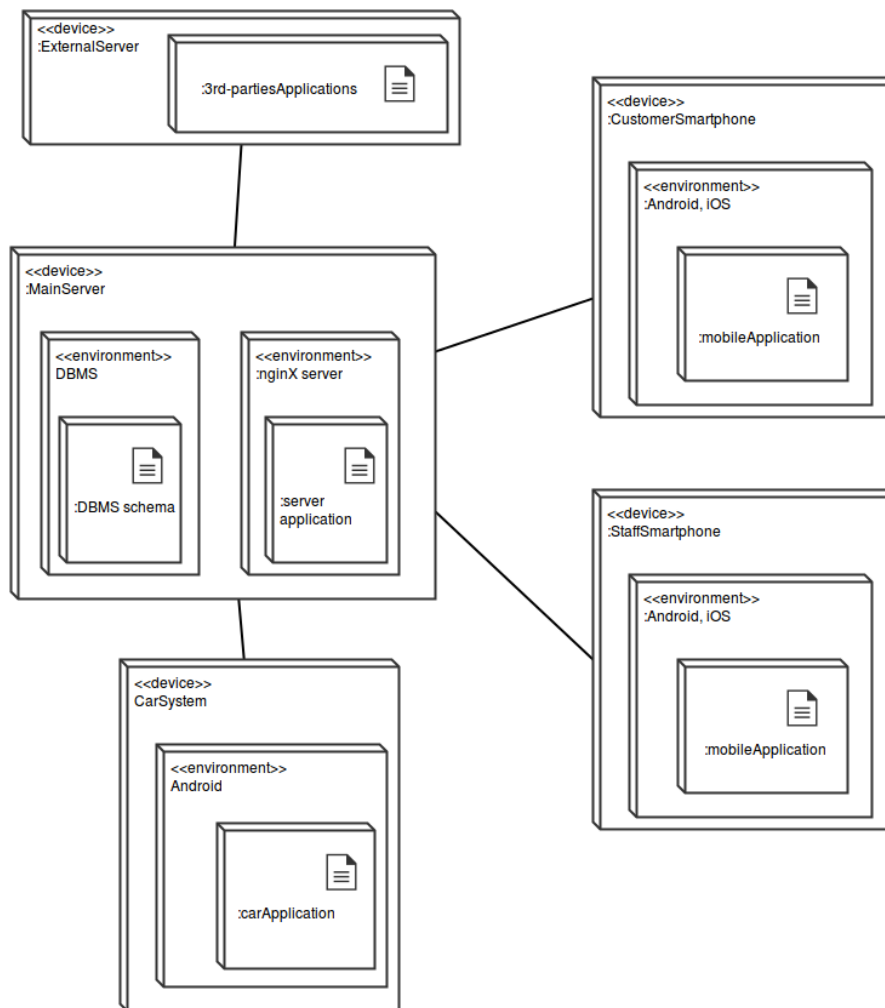


Figure 12: Deployment View of the system



## 2.8 Selected Technologies

Having defined the interface between the several functionality provided by the components, all possible communication technologies may have been chosen. Obviously some choices are more apt than others with the respect of latency, throughput, elegance of the design and other factors; however the whole design will remain intact whichever technology we choose.

Here we simply give some reasonable suggestions for protocols and technologies that seems us a better fit for the system we modeled.

### RESTful API

The main server will expose its API in the most conventional way, using the classical HTTP/TCP/IP stack. In particular we provide RESTful interfaces. Model everything as an entity provides enough capabilities to actual implement the whole system while remaining constrained to only the basic REST verb. This helped a lot in keeping the whole API scheme simple to understand and to implement.

Apps consume the REST interface provide by the server, however to implement the communication between the server and the app we will use the long polling strategy.

### MQTT

As for now, most of the communication between the server and the car will happens using the PubSub protocol, while some special messages (like, for example, the ValidateSolve function) will stay on a more basic Client-Server approach.

Between all the implementation of the PubSub protocol we chose to use MQTT for its low overhead, its QoS and because it is widely used in the industry.

### Nginx

We decided to use Nginx on the main server instead of a more common Apache server. This choice has been made with care, because Apache is surely a more widely used standard for running servers, whereas Nginx is a younger, less used solution.

Indeed, considering our server is going to handle a lot of parallel operations, we chose a software designed exactly to tackle this issue better than standard Apache-based solutions.

what about this? =

### Database

what about this?

### 3 Algorithm Design

In this section we define a first sketch of a BOOKING\_MANAGER, clearly this implementation is extremely naive, it doesn't take care of persisting its status nor it doesn't validate the input, however is effective to show our intention and expectation from the implementator.

```
1 class BookingManager():
2     def __init__(self):
3         self.reservation_running = min_heap()
4         self.reservations = {}
5         thread_expired = threading.Thread(target = self.manageExpired)
6         thread_expired.start()
7
8     def newBook(self, id_user, id_car):
9         reservation = {"id_reservation" : newID(),
10                        "id_car" : id_car,
11                        "id_user" : id_user,
12                        "expire_time" : time.now() + time.timedelta(hours = 1)
13                    }
14         self.reservation_running.add((reservation["expire_time"], reservation))
15         self.reservations[reservation["id_reservation"]] = reservation
16         return reservation["id_reservation"]
17
18     def removeReservation(self, id_reservation):
19         if id_reservation in self.reservations:
20             reservation = self.reservations[id_reservation]
21             self.reservation_running.remove(reservation)
22             del self.reservation[id_reservation]
23             return True
24         return False
25
26     def getReservation(self, id_reservation):
27         return self.reservation[id_reservation]
28
29     def manageExpired(self):
30         while True:
31             (expire_time, reservation) = self.reservation_running.pop()
32             if expire_time > time.now():
33                 FineManager.expireReservation(reservation)
34                 self.removeReservation(reservation["id_reservation"])
35             time.sleep(1)
```

## 4 User Interface Design

### 4.1 Mockups

Mockups have already been included in the RASD (section 3.3: Non Functional Requirements)

### 4.2 UX Diagrams

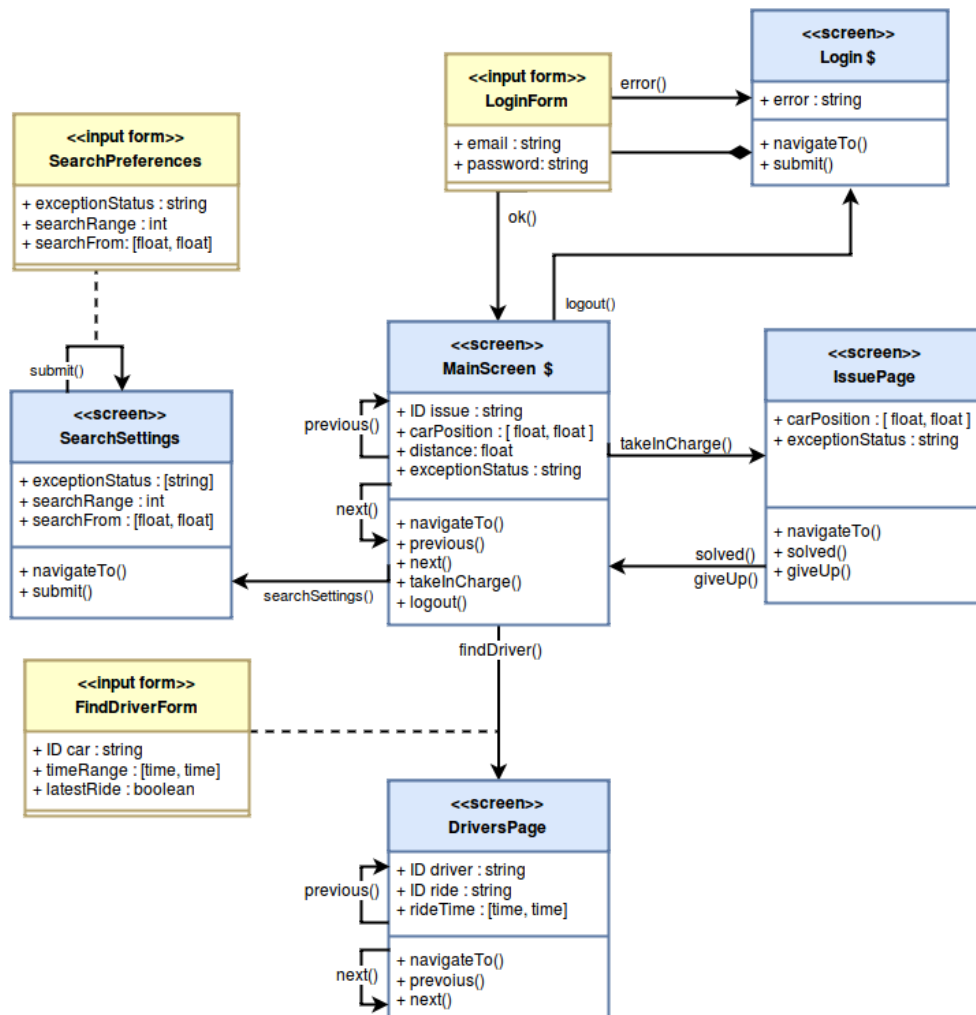


Figure 13: UX Diagram of the interface of staff's application

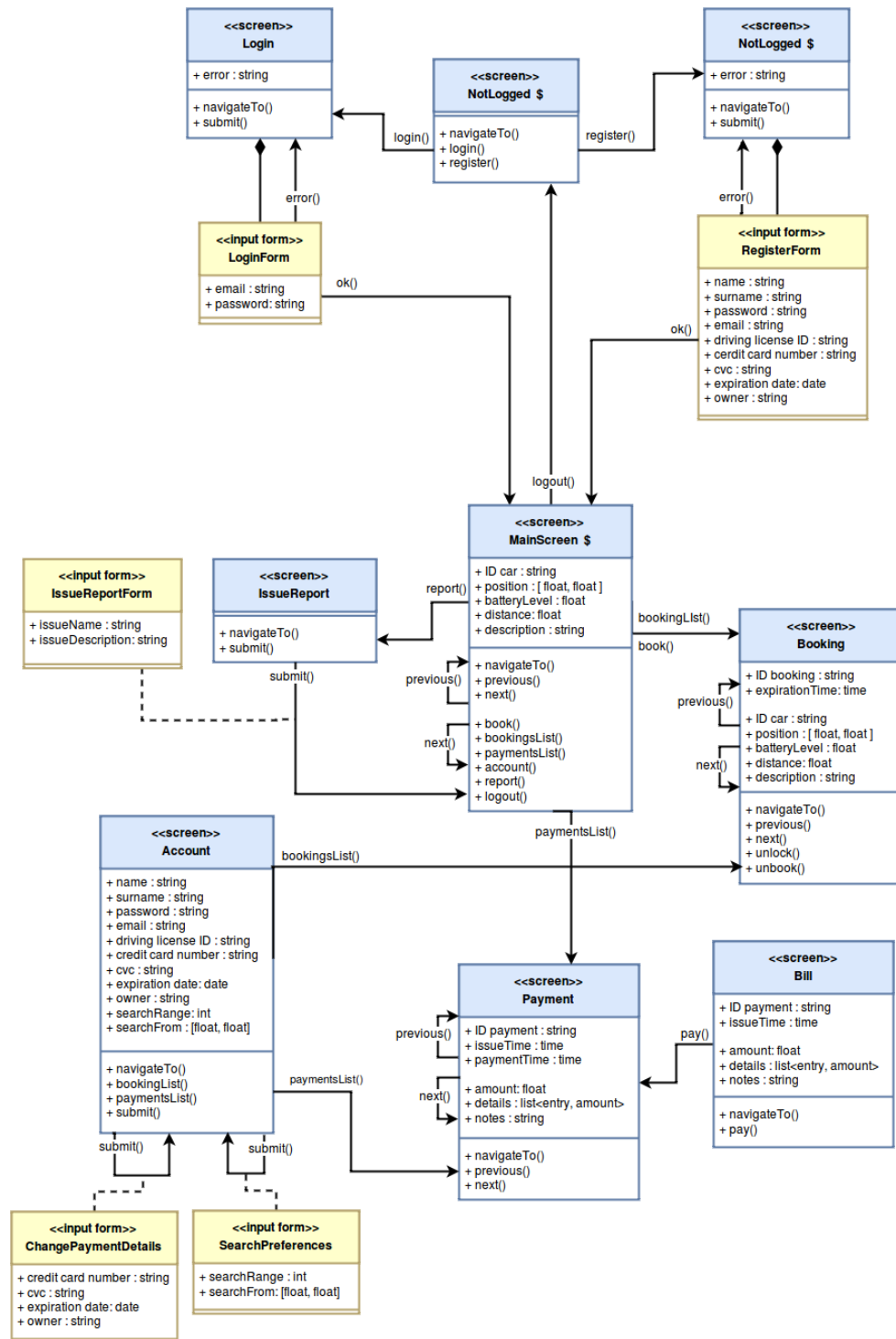


Figure 14: UX Diagram of the interface of customer's application

### 4.3 BCE Diagrams

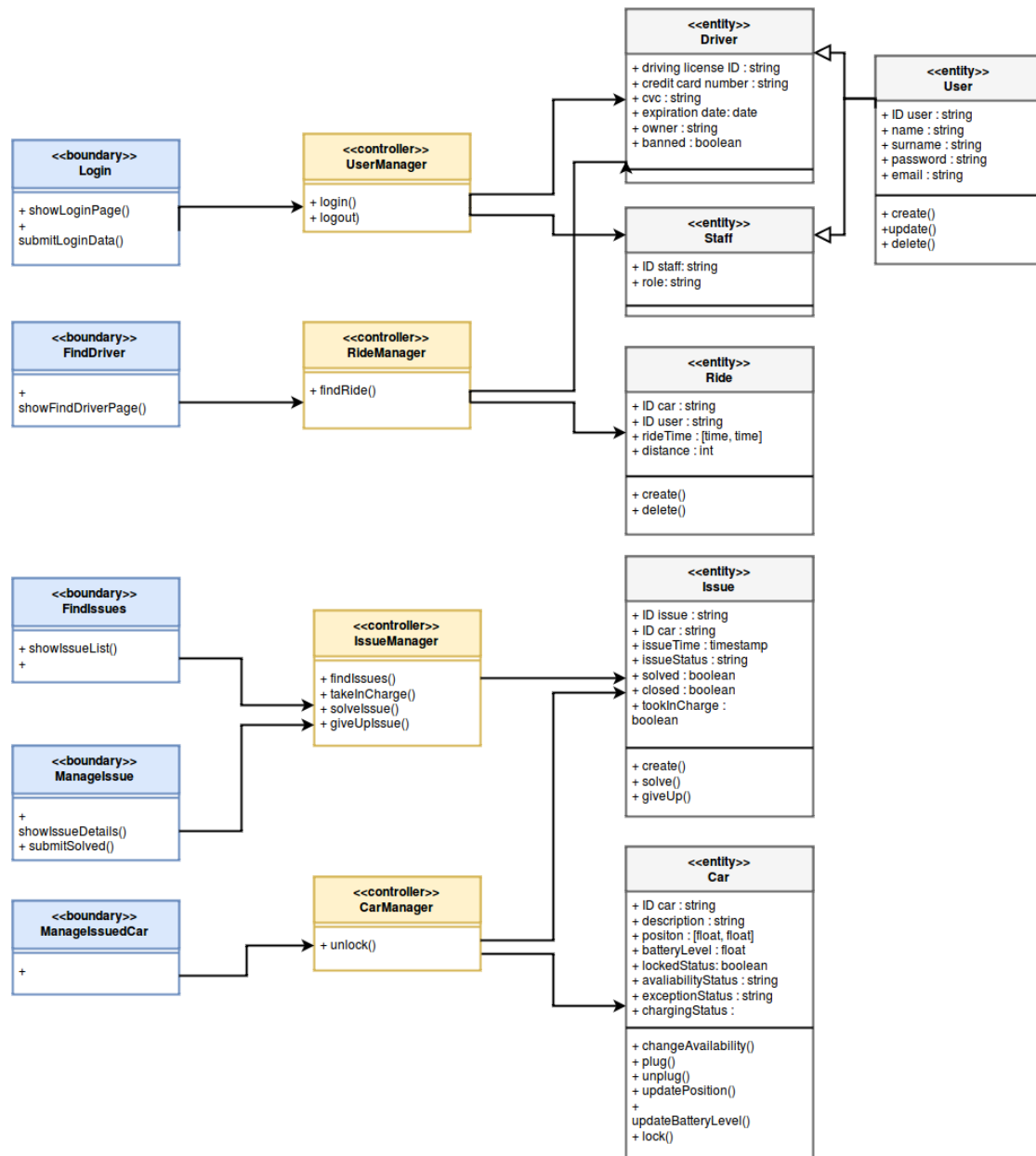


Figure 15: BCE Diagram of the interface of staff's application

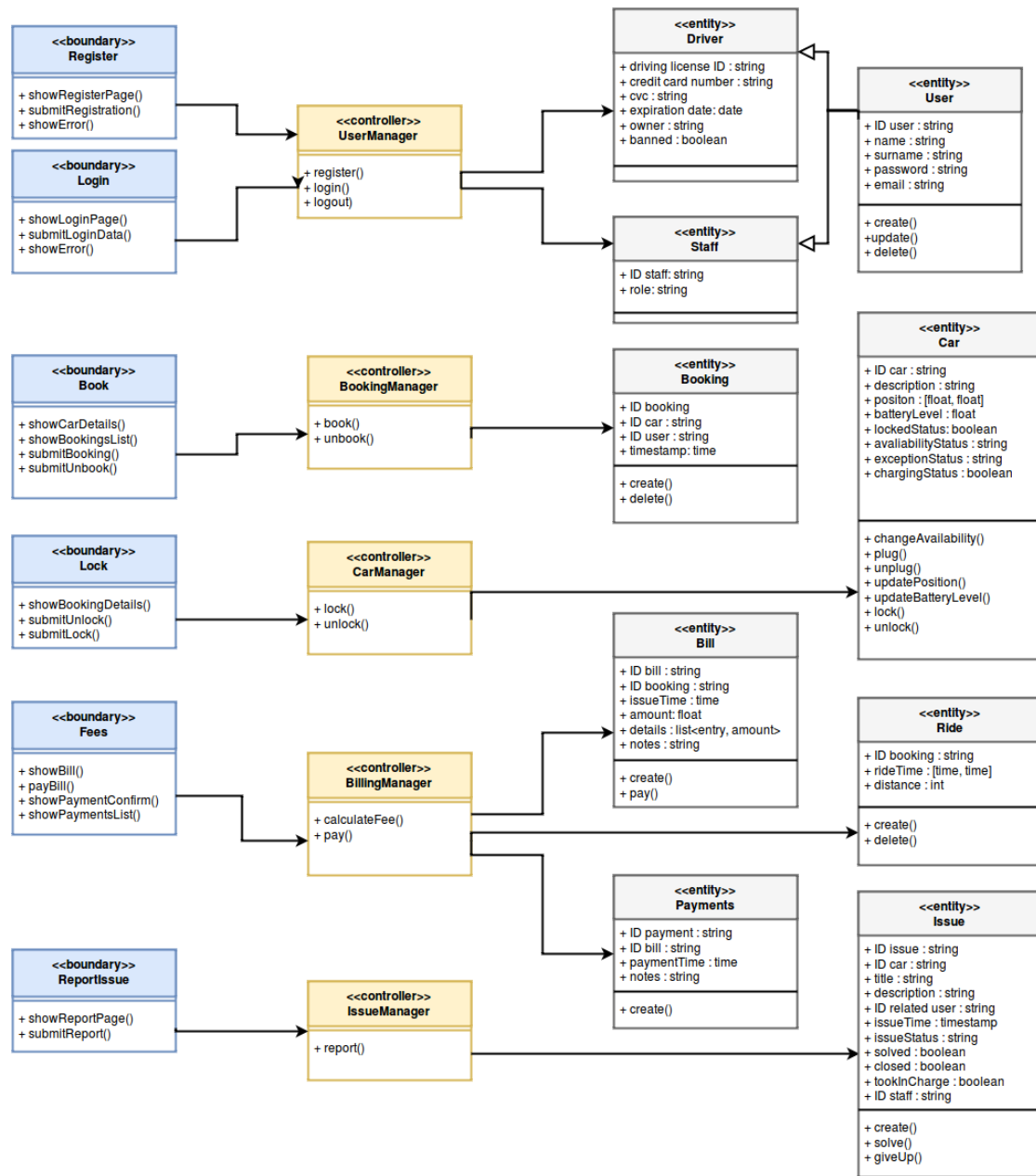


Figure 16: BCE Diagram of the interface of customer's application

## 5 Requirements Traceability

In this section we highlight which components ends up satisfying the goals defined in RASD once the design process is concluded.

**REGISTRATION** Users can register to the system.

- USER\_MANAGER

**LOGIN** Users can log into the system providing their email and password and logout from the system.

- USER\_MANAGER

**LOOKUP** Users can look for cars near them or near a specific position and range.

- GEOLOCATION
- USER\_POSITION

**BOOK** Users can book cars to use within the next hour.

- BOOKING\_MANAGER

**UNBOOK** Users can remove a previously made reservation.

- BOOKING\_MANAGER

**UNLOCK** Users can unlock a nearby car that was previously booked.

- USER\_POSITION (to ensure users are near the car they want to unlock)
- CAR\_POSITION
- REMOTE\_CAR\_MANAGER

**RIDE** Users can ride a reserved car.

- RIDE\_MANAGER
- REMOTE\_CAR\_MANAGER
- POSITION\_MANAGER
- LOCAL\_CAR\_MANAGER

**SAFE AREAS** Users can locate safe parking areas while driving.

- GEOLOCATION

**UNSAFE PARKING** The system reacts to unsafe parking. It first tells users they are going to be fined if they leave the car unsafely parked, then it fines users who leaves the car in an unsafe area.

- LOCAL\_CAR\_MANAGER
- NOTIFY
- BILLING\_MANAGER

**POWER\_STATION** Users can locate and use charging station.

- GEOLOCATION
- LOCAL\_CAR\_MANAGER
- NOTIFY

**CHARGE** Users are charged a fee at the end of the ride or if a reservation expires.

- NOTIFY
- BOOKING\_MANAGER
- BILLING\_MANAGER

**PAYMENT** Users can pay bills through the app and set their default payment method.

- BILLING\_SYSTEM
- USER\_MANAGER (to set default payment method)

**REPORT\_ISSUE** Users can report issues to the system.

- ISSUE\_MANAGER

**FIND\_ISSUES** The staff can locate cars that need their intervention.

- GEOLOCATION

**SUPPORT** The staff can identify and solve car's issues.

- LOCAL\_CAR\_MANAGER
- REMOTE\_CAR\_MANAGER
- ISSUE\_MANAGER



## 6 Conclusions

### 6.1 Tools used

During the development of this document we used the following tools:

- **Github** to version control the project
- **L<sup>A</sup>T<sub>E</sub>X** on TeXworks to redact this document
- **www.draw.io** to draw UML graphs
- **Gimp v.2.8** to mockup the application
- **LibreOffice Draw** to draw the system's overview at section 2.1

### 6.2 Hours of work

- SZ: 1h on 30/11
- SM: 5h on 2/12
- SZ: 5h on 2/12
- SZ: 3h on 4/12 (RASD review)
- SZ: 3h on 5/12
- SZ: 4h on 6/12
- SZ: 7h on 7/12
- SZ: 10h on 9/12
- SZ: 12h on 10/12