POLITECNICO
MILANO 1863

PowerEnJoy

Design Document

Version 1.3

Authors:
Mosciatti Simone
Zanzottera Sara

Reference Professor:
Mottola Luca

December 11, 2016

Contents

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 acthitecture's details, the desing patterns we chosed 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 penalities in term of discounts or overprices 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. All these 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 (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.

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

Nearby Cars Available cars located within a maximum distance to a specific position.

Available Cars Cars whose Availability Status is set to "Available".

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.

Driver Whoever is driving a regularly booked PowerEnJoy car.

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

Notification A form of comunication 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 an user still need to pay to PowerEnJoy .

Safe Area An parking area, predefined by the company, where 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 (Power Station) Dedicated areas where is possible to plug the PowerEnJoy cars to charge their batteries.

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

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

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

Central System (Main Server) The central system for PowerEnJoy . All the command and all the data are streamed, analyzed and used here.

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.

Location Pair of integer values as provided by GPS sensors.

Payment Method Set of data relative to a credit card.

Email address (Email) Unique string identifiying an email box to which email messages are delivered.

Identity ID Personal code provided by local authorities to uniquely identify citizens.

Driving License ID The unique code reported on every legal driving license.

Session Key A string representing a key for a secured channel. Used to secure communications between the server and the nodes.

Scanned License An high quality image of the driving license acquired by the car's onboard system.

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. In detail:

   (a) **Design Process Description**: a detailed explanation of the design process we followed.
   (b) **Goals Analysis**: an accurate analysis of the functionalities requied to fulfill the systems' goals we defined in the RASD.
   (c) **Interfaces Design**: the logical organization of the functionalities we identified in the previous section into high-level interfaces.
   (d) **Communication Design**: an analysis of the communication needs and requirements, leading to the definition of the protocols we use between nodes.
   (e) **Components Design**: combining the previous two sections' conclusions, we define how system's component should implement the interfaces and which node will host them.
   (f) **Runtime View**: in order to ensure the designed system is coherent, we prepared four Sequence Diagrams for the most important processes and explained them in detail.
   (g) **Deployement View**: a diagram showing how we meant to depoly the real system.
   (h) **Selected Technologies**: a description of the techologies we would like to use for this project, and the motivations behind our choice.

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.

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)
- *Requ irements and Specification Document* (referring to this project)
- *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 alreay 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 Responsability" principle in order to provide highly decoupled and reusable components. Again, if those functionalities were already working our work 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 comunication mechanism to use between the nodes.

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

To double check the correctness of the overall design we produce several sequence diagrams, showing for the uses cases what functionalities are involved and what functionalities are 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 betwen 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 funcionalities to be provided by the backend. These are listed with each functionality and named **Sy/Functionality-Name**, 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

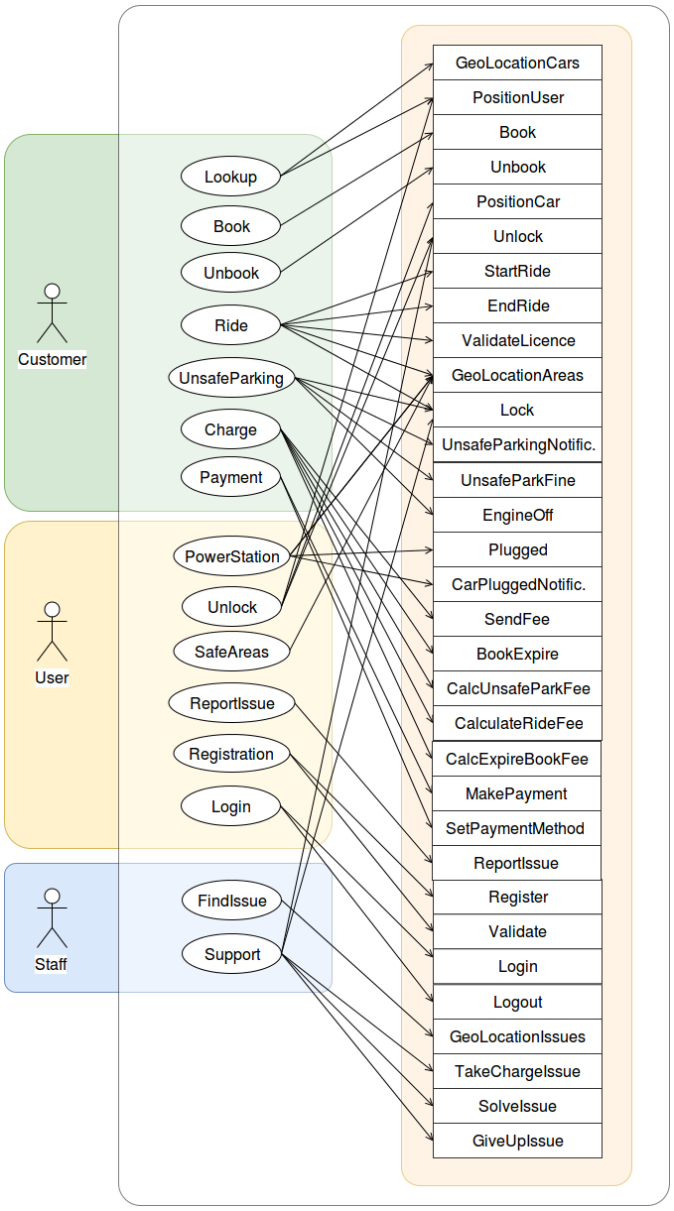

Figure 1: Graphical visualization of goals and system's required functions identified above, divided by tipology of users.

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

USER_MANAGER

Responsability Manages the users.

USER/Register ⇔ 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 ⇔ 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 as credit card informations.

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 ⇔ 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 ⇔ Sy/Login

Responsability Allows users to logout the system.

Input The login session key.

Output The session key is invalidated.

USER/SetPaymentMethod ⇔ 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 ⇔ 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 ⇔ 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 ⇔ 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 ⇔ 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 ⇔ Sy/PositionCar

Responsability Retrieves the position of a specific car.

Input The ID of the car.

Output The coordinates of the car.

POSITION/User ⇔ 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 ⇔ 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 ⇔ 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 ⇔ 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 ⇔ 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 ⇔ 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 ⇔ Sy/Lock

Responsability Locks the car.

Input The ID of the car.

Output The car is locked.

CAR/TurnOff ⇔ 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 ⇔ 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 ⇔ 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 fees and payments

BILL/CalculateRideFee ⇔ 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 ⇔ Sy/CalculateExipreBookFee

Responsability Calculates the amount of a expired prenotation fee.

Input The ID of the booking.

Output The ID of the fee refered to the expired prenotation.

BILL/CalculateUnsafeParkingFine ⇔ 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 refered to the fine for unsafe parking.

BILL/Pay ⇔ 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 ⇔ 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 ⇔ 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 responsable for the issue.

ISSUE/Solve ⇔ Sy/SolveIssue

Responsability Allows operators to close an issue marking it as Solved. The system is resposible 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 ⇔ 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 ⇔ {Sy/SendFee, Sy/UnsafeParkNotif., Sy/CarPluggedNotif.}

Responsability Notify the user of some event. It may also prompt the user to acknoledge 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, depending on the notification .

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 identified the abstract interfaces that, together, provide the required functionalities of our system, we are going to design how the nodes interacts, in order to have all the required information to design the actual components.

Physical structure

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

- 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.

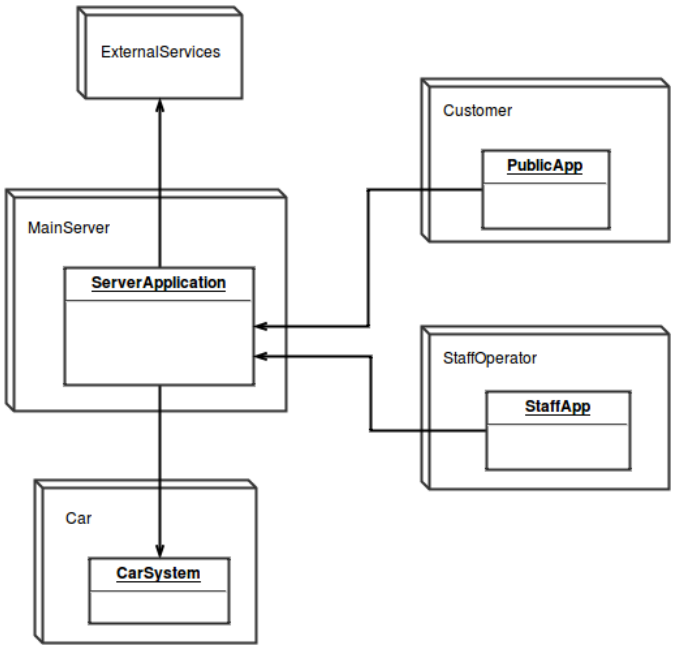

Figure 2: Higher level components of the system

Communication Strategy

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

It would have been possible to make each node comunicate 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 approach is more suitable.

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

- one between the main server and the 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, in order for the system to have full control on these otherwhise 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 and privacy concerns.

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 indipendently 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 troughput 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.

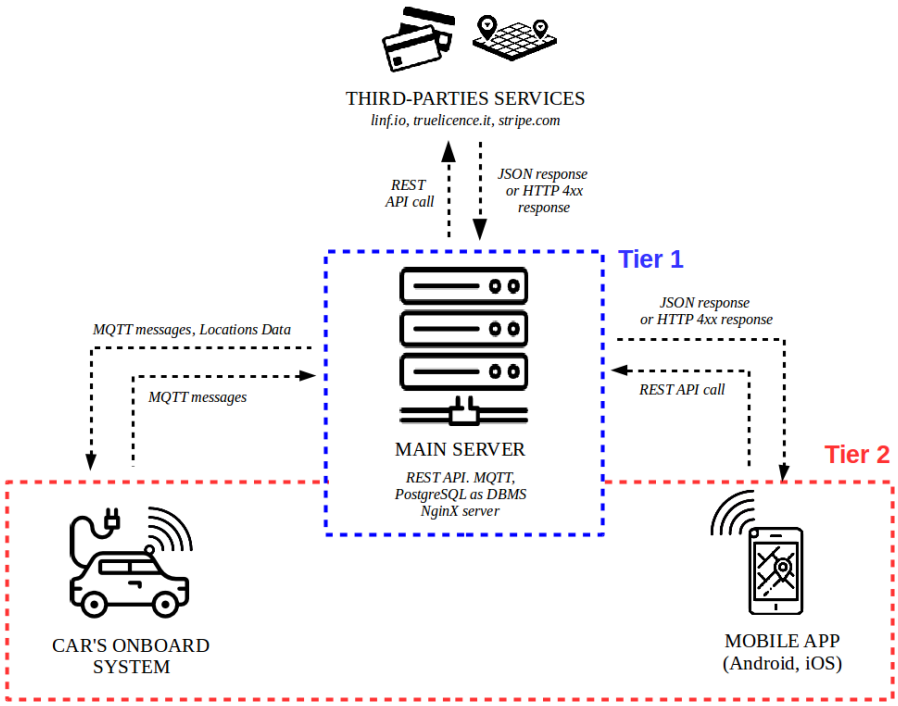

Figure 3: Communication design of the process (anticipating some technology choices we are going to justify in next sections).

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 with the deploy of the system 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 partecipation 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 incomig 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.

Moreover we provide a GUI for both apps and cars: so we deploy **PUBLIC_APP_GUI** on customer's apps, **STAFF_APP_GUI** on staff's apps, and **CAR_GUI** on the cars.

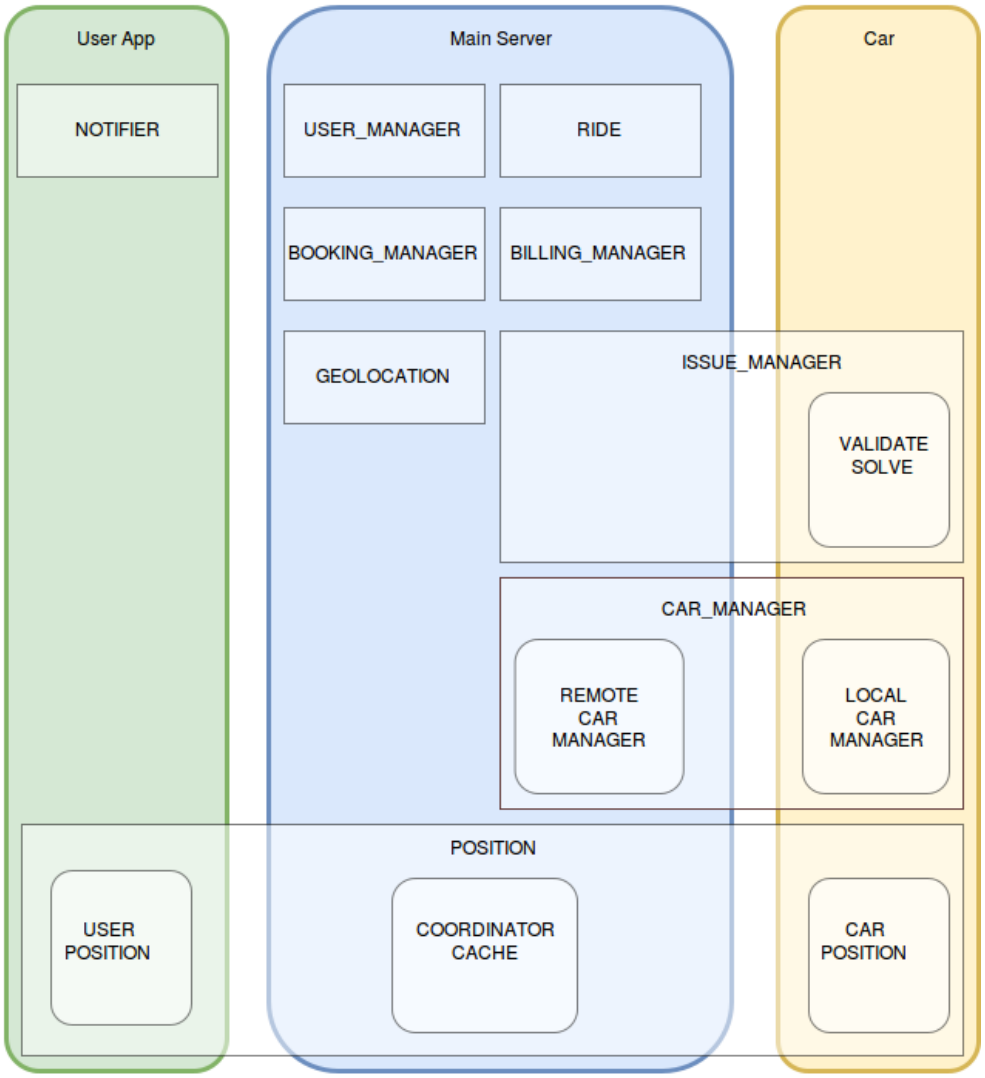

Figure 4: Graphical visualization of the abstract components needed.

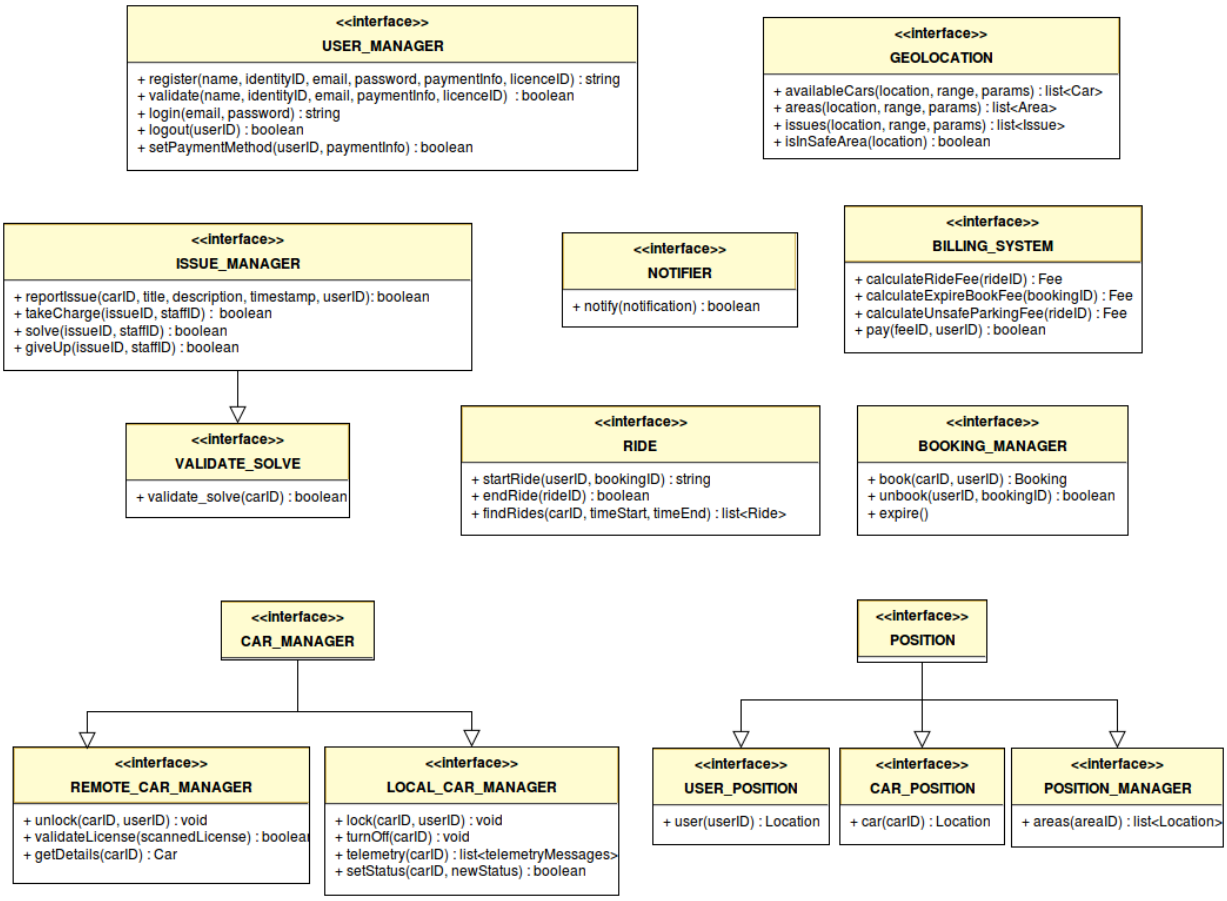

Figure 5: Description of the interfaces of the components identified above.

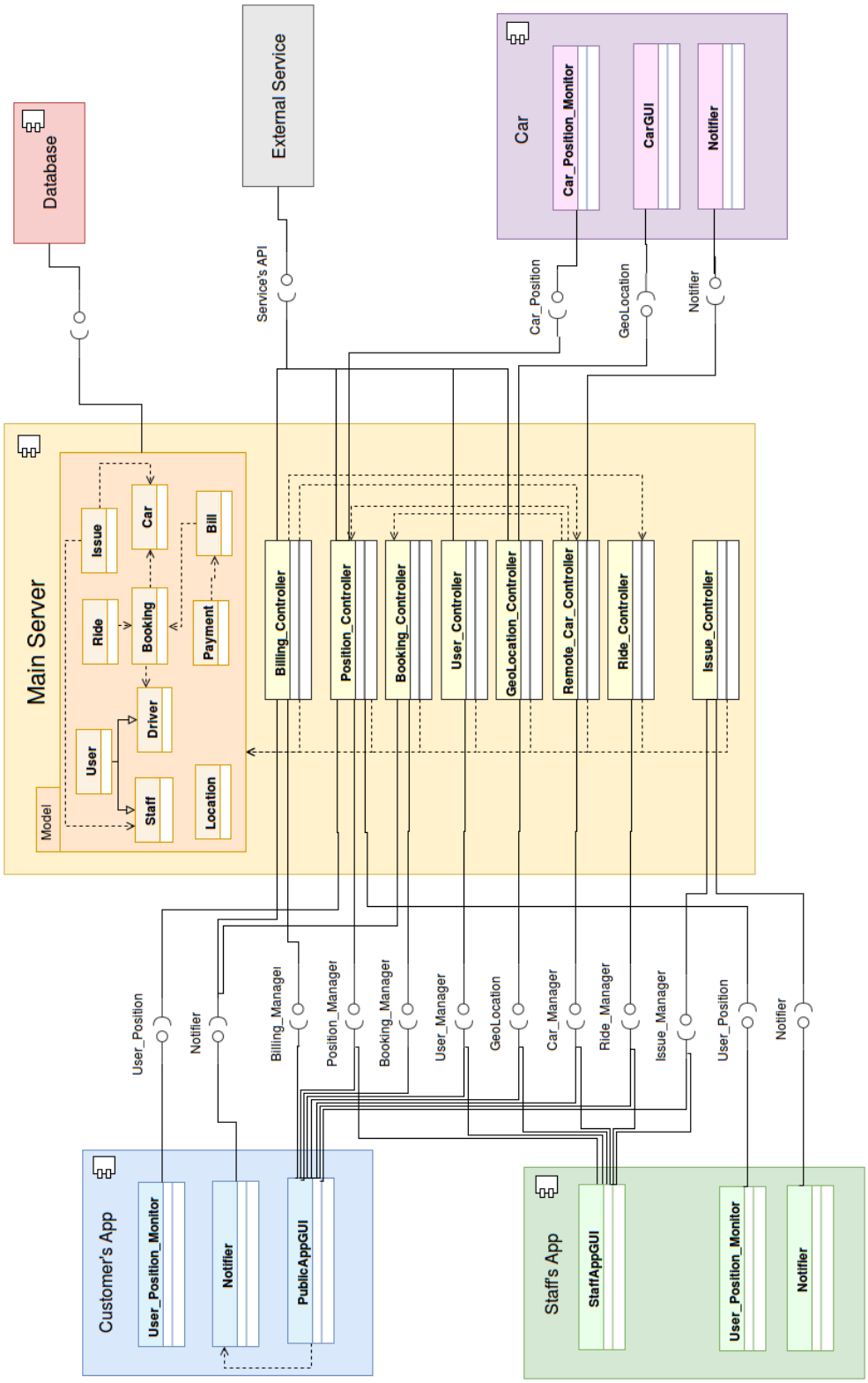

Figure 6: Components View Diagram of the entire system.

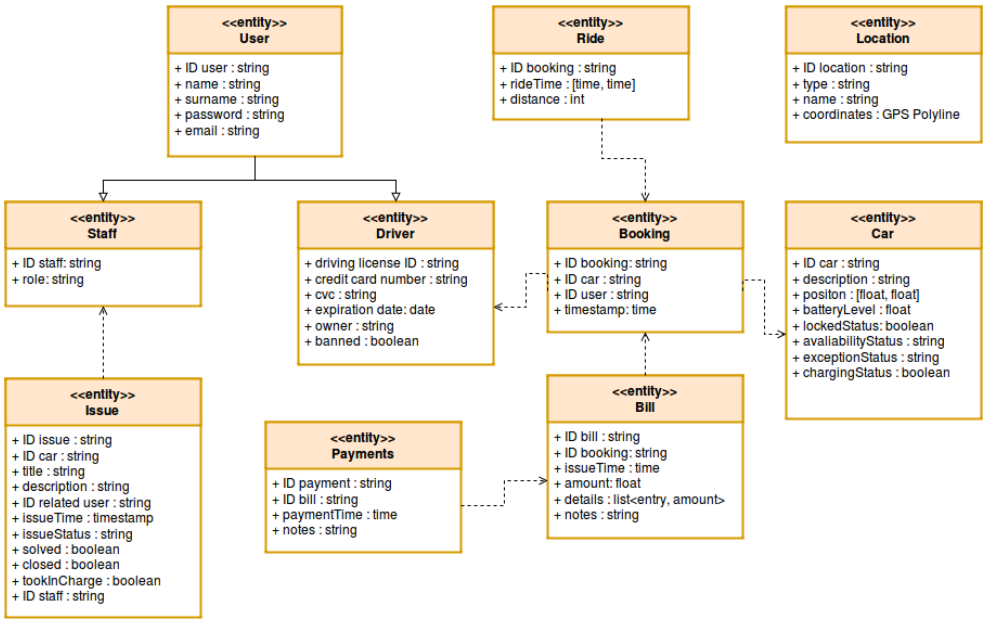

Figure 7: Class Diagram of the Model.

2.6 Runtime View

In this section we provide four Sequence Diagram and their description, to ensure the system is coherent.

Register and Login

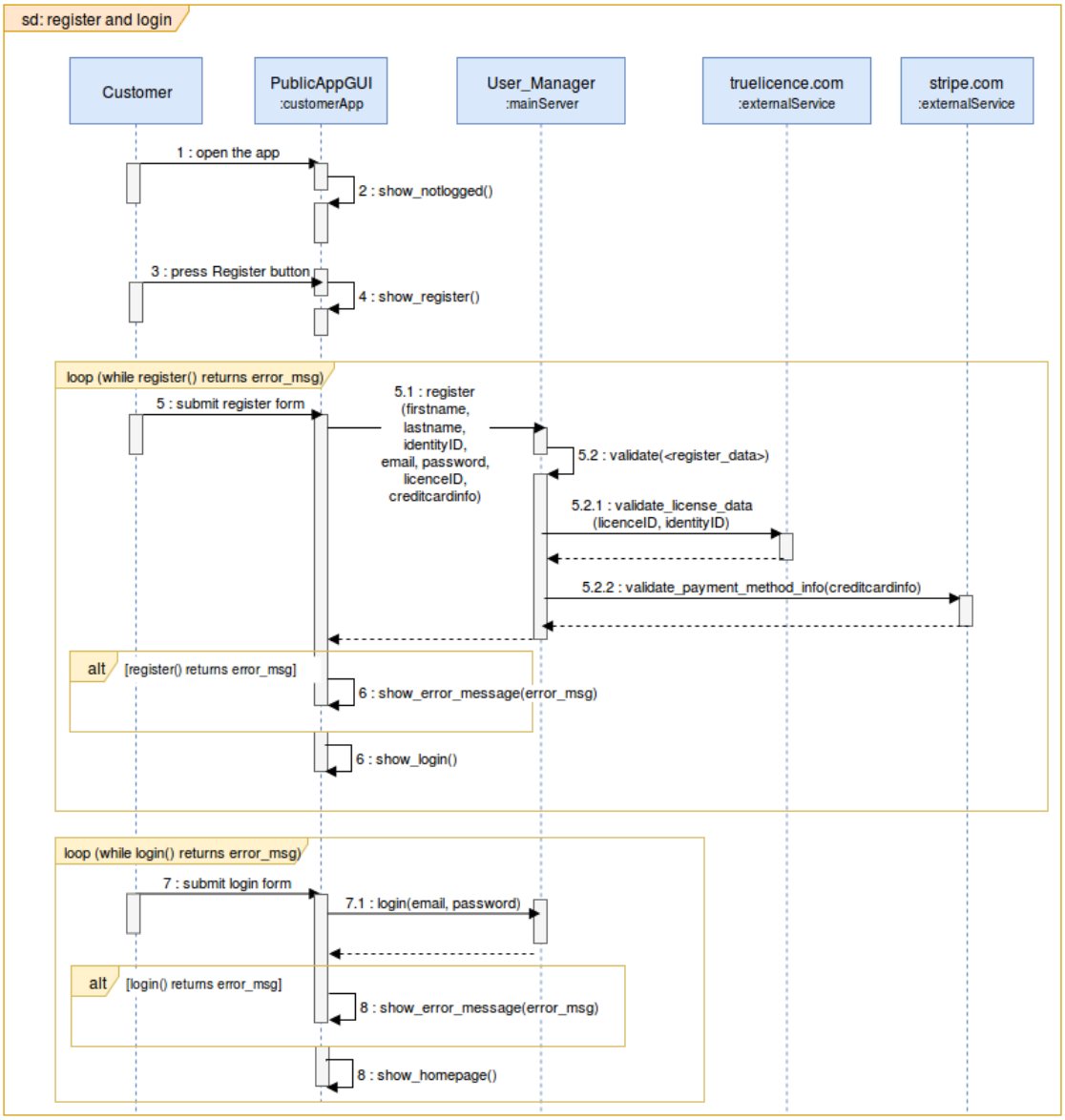

Figure 8: Sequence Diagram for Registration and Login process of customers.

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 when needed.

When the registration/login process is completed server-side, USER_MANAGER forward the results 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

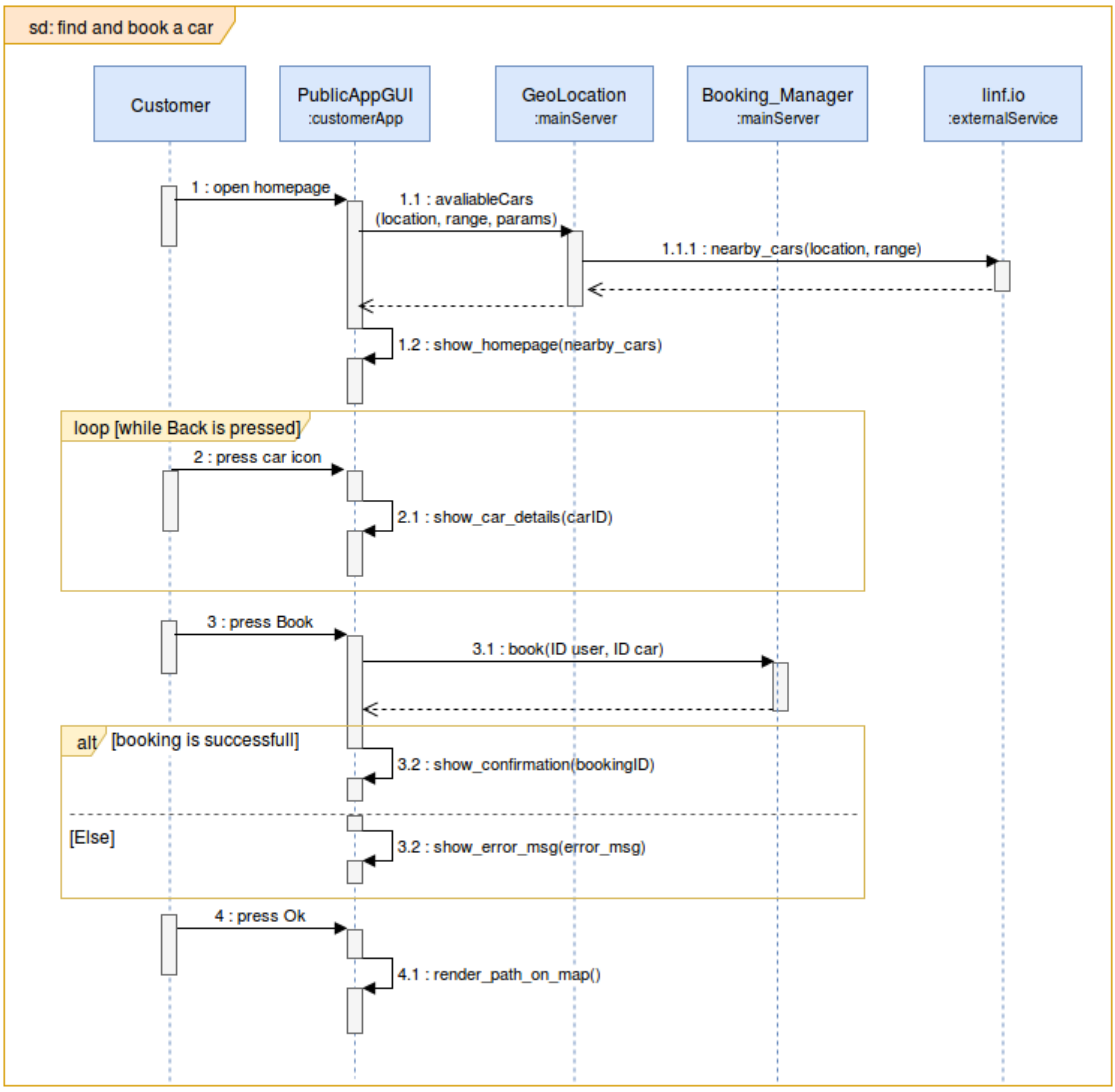

Figure 9: Sequence Diagram for Lookup and Booking process of customers.

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

The first goal is achieved with the sole need of GEOLOCATION, deployed 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

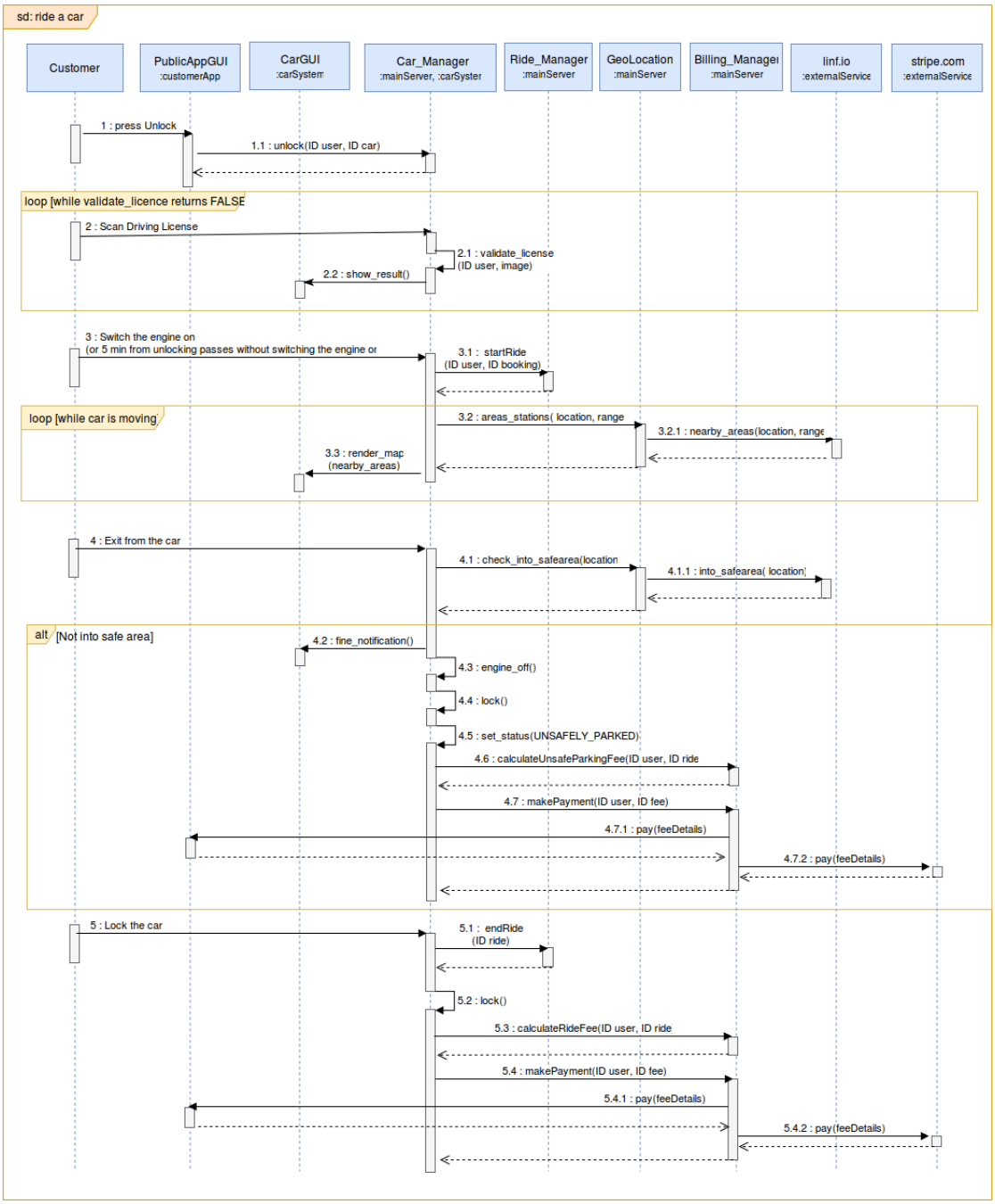

Figure 10: Sequence Diagram for the Riding process of customers.

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(userID, carID) 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 the license correspondes to the user who booked the car, it 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 lenght 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/CheckIntoSafeArea(Location).

If not, a series of operations are performed. First, a notification is sent to CarGUI in order to notify users and possibly have them back in the car. If the user does not come back within a short amount of time (e.g. a minute), 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.

Otherwhise, the system waits for the car to get locked. Once the locking requests arrives, first the ride is ended with a call to RIDE_MANAGER/EndRide(rideID) and the car is locked. Then the regular fee is calculated and the payment issued.

Issue Management

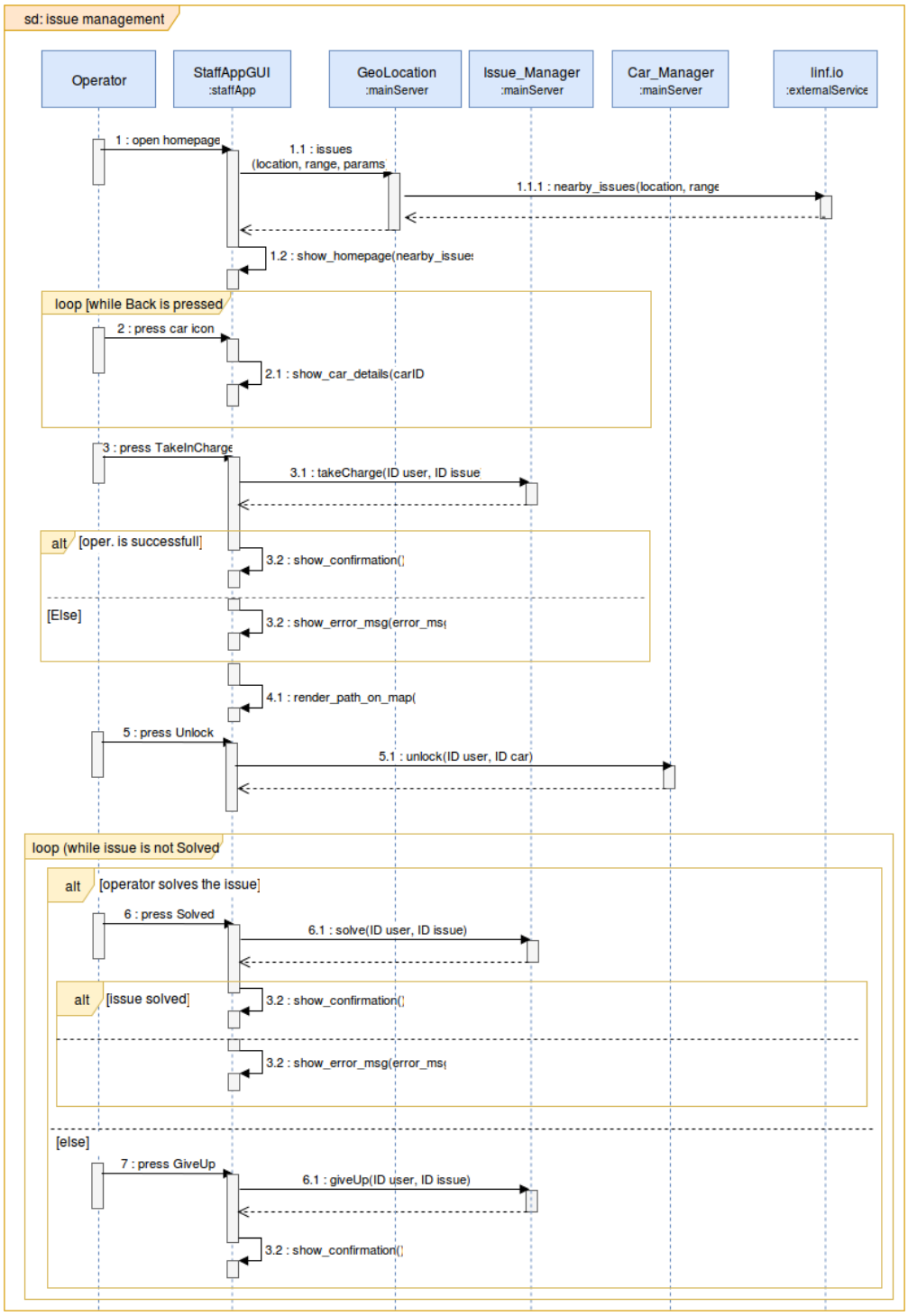

Figure 11: Sequence Diagram for the Issue Management process for staff operators.

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 Deployement View

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

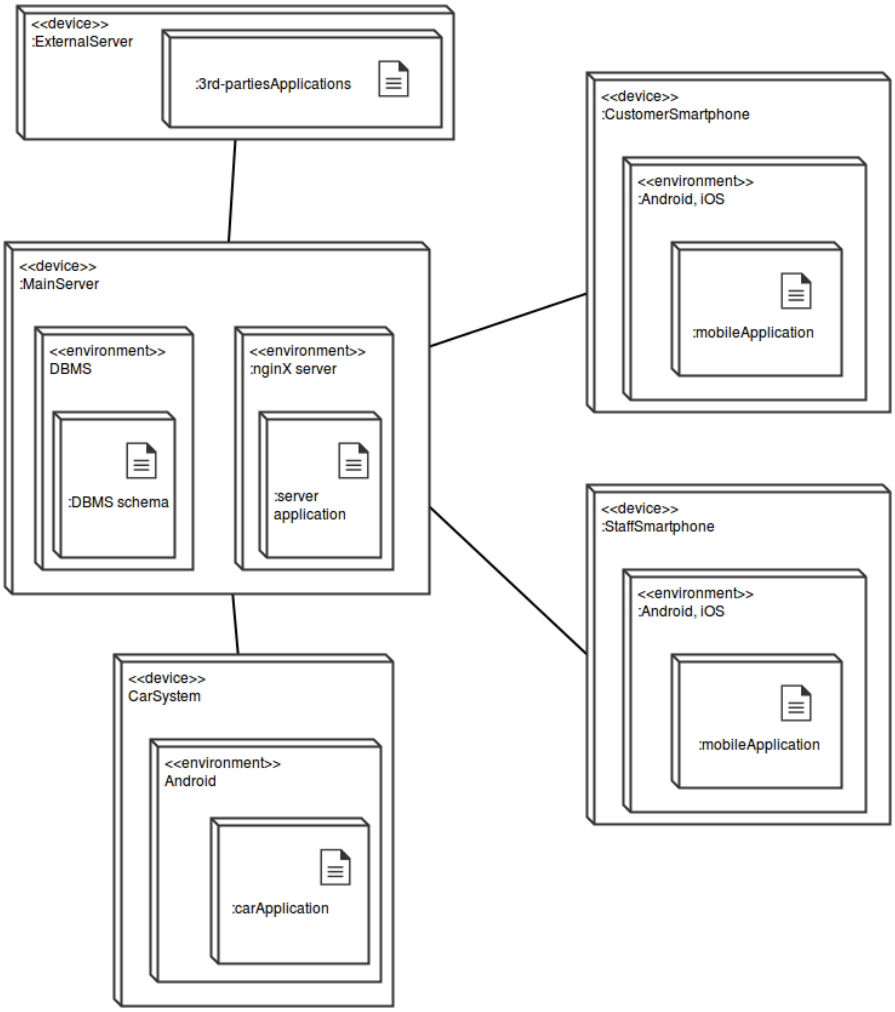

Figure 12: Deployement View of the system

2.8 Selected Technologies

Having defined the interfaces between the several components, all possible technologies may have been chosen to implement them. Obviously some choices are more apt than others with respect of latency, throughput, elegance of the design and other factors; however the whole design remains intact whichever technology we choose.

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

RESTful API

The main server exposes its API in the most conventional way, using the classical HTTP/TCP/IP stack. In particular we aim to provide RESTful interfaces. Modelling everything as an entity provides enough capabilities to implement the whole system while remaining constrainted to only the basic REST verb.

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

MQTT

As for now, most of the communications between the server and the car use 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 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.

PostgreSQL

As DBMS for our system we chose PostgreSQL because it is an enterprise-level, open-source software, widely known in the industry for its reliability, integrity and correctness. it supports all major operating systems, it is very standard compliant and at the same time highly customizable.

In addition, PostgreSQL features some spatial database extenders, like PostGIS, that adds geographical objects support and makes easier running location queries on the database.

3 Algorithm Design

In this section we define a first sketch of an implementation of BOOKING_MANAGER.

Clearly this implementation is extremely naive: it doesn't take care of persisting its status and it doesn't validate the input. However it is effective to show our intention and expectation from the developers.

```
class BookingManager():
    def __init__(self):
        self.reservation_running = min_heap()
        self.reservations = {}
        thread_expired = threading.Thread(target = self.manageExpired)
        thread_expired.start()

    def newBook(self, id_user, id_car):
        reservation = {"id_reservation" : newID(),
                       "id_car" : id_car,
                       "id_user" : id_user,
                       "expire_time" : time.now() + time.timedelta(hours = 1)
    }
        self.reservation_running.add((reservation["expire_time"], reservation
    ))
        self.reservations[reservation["id_reservation"]] = reservation
        return reservation["id_reservation"]

    def removeReservation(self, id_reservation):
        if id_reservation in self.reservations:
            reservation = self.reservations[id_reservation]
            self.reservation_running.remove(reservation)
            del self.reservation[id_reservation]
            return True
        return False

    def getReservation(self, id_reservation):
        return self.reservation[id_reservation]

    def manageExpired(self):
        while True:
            (expire_time, reservation) = self.reservation_running.pop()
            if expire_time > time.now():
                FineManager.expireReservation(reservation)
                self.removeReservation(reservation["id_reservation"])
            time.sleep(1)
```

4 User Interface Design

4.1 Mockups

Mockups have already been included in the RASD (section 3.3: Non Functional Requirements), so they are not being reported here.

4.2 UX Diagrams

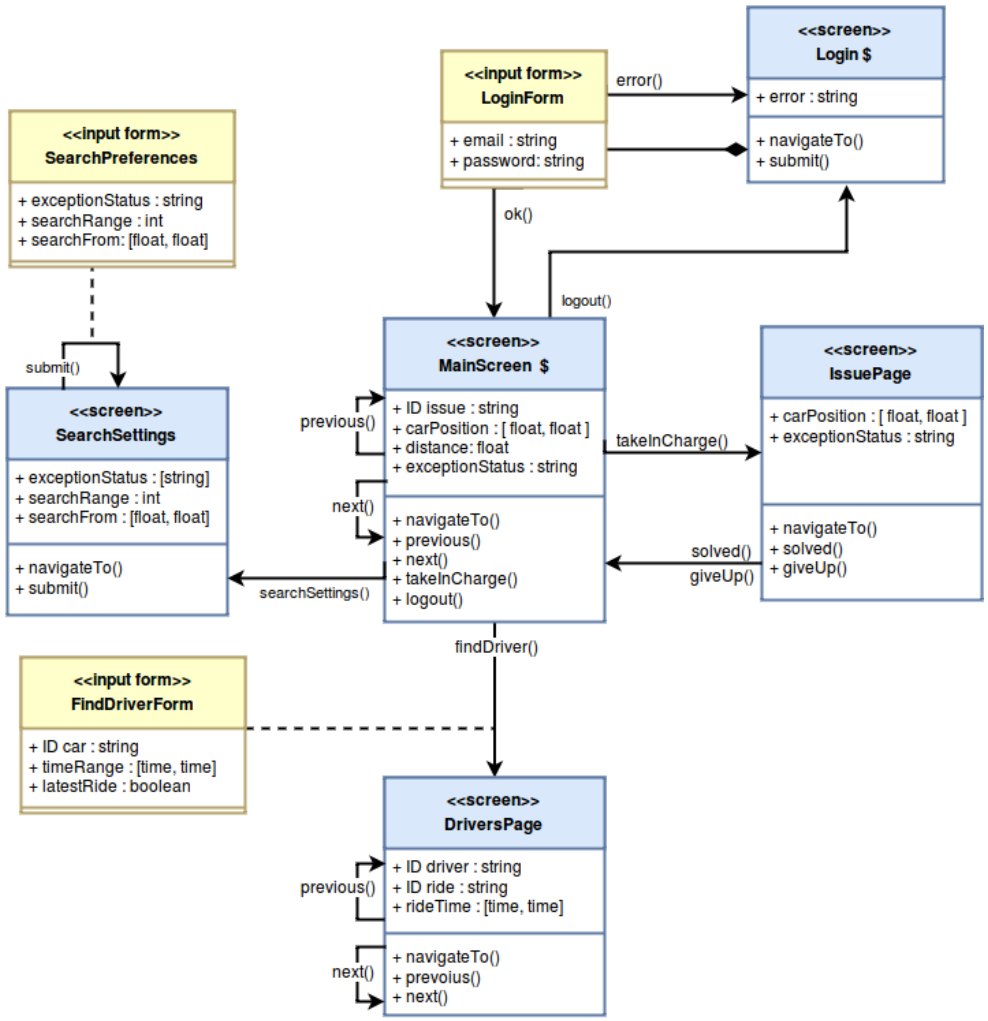

Figure 13: UX Diagram of the interface of staff's application

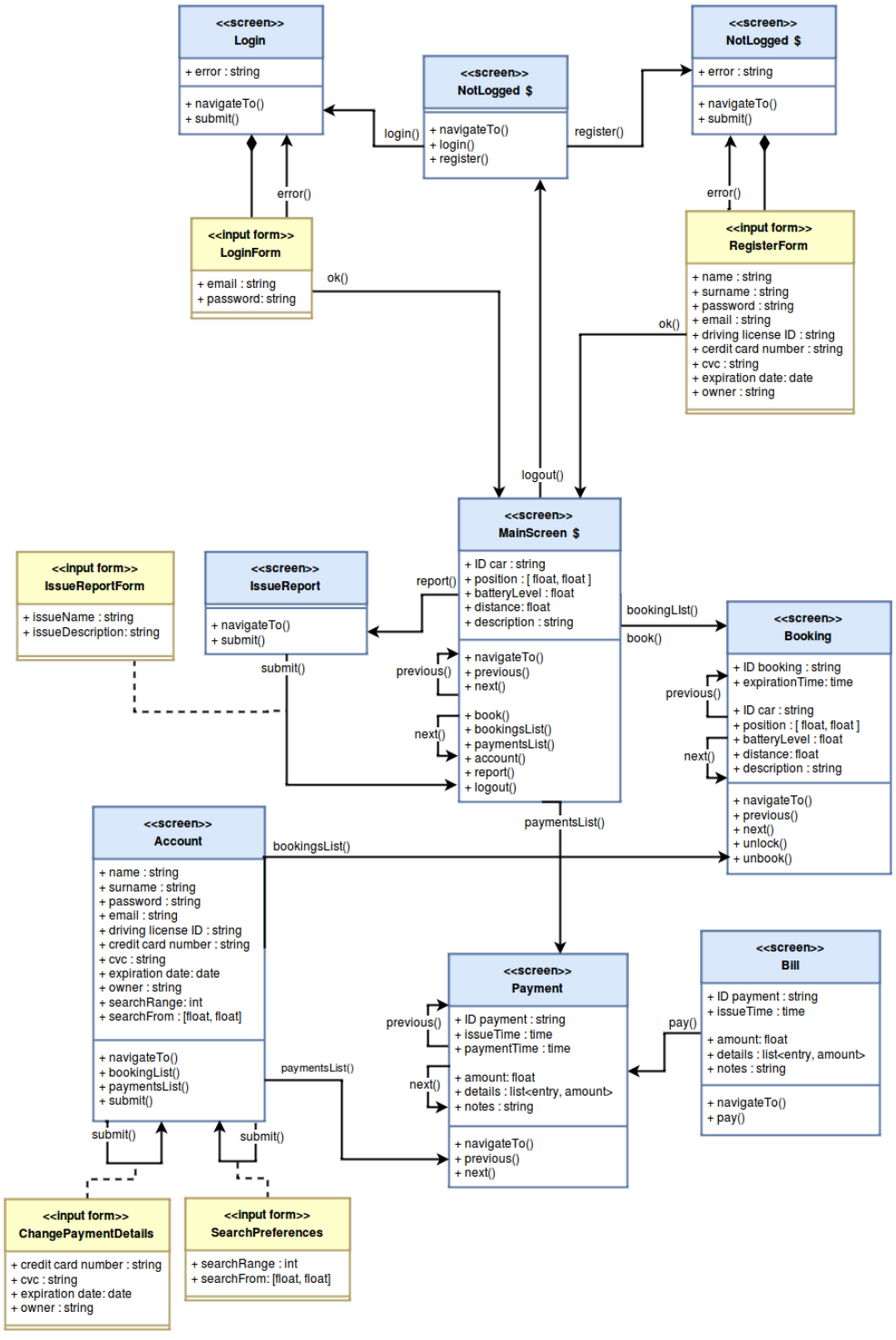

Figure 14: UX Diagram of the interface of customer's application

4.3 BCE Diagrams

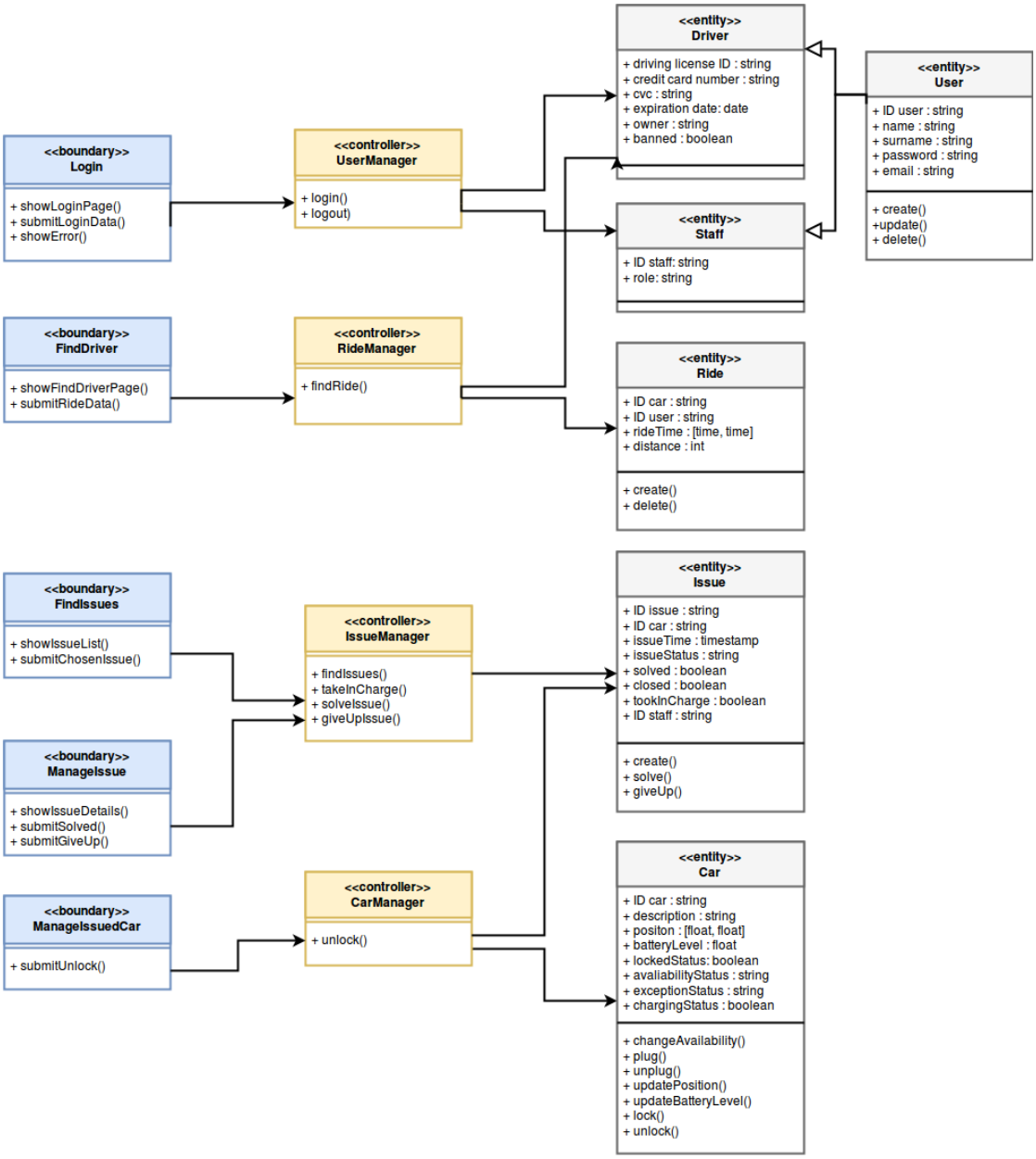

Figure 15: BCE Diagram of the interface of staff's application

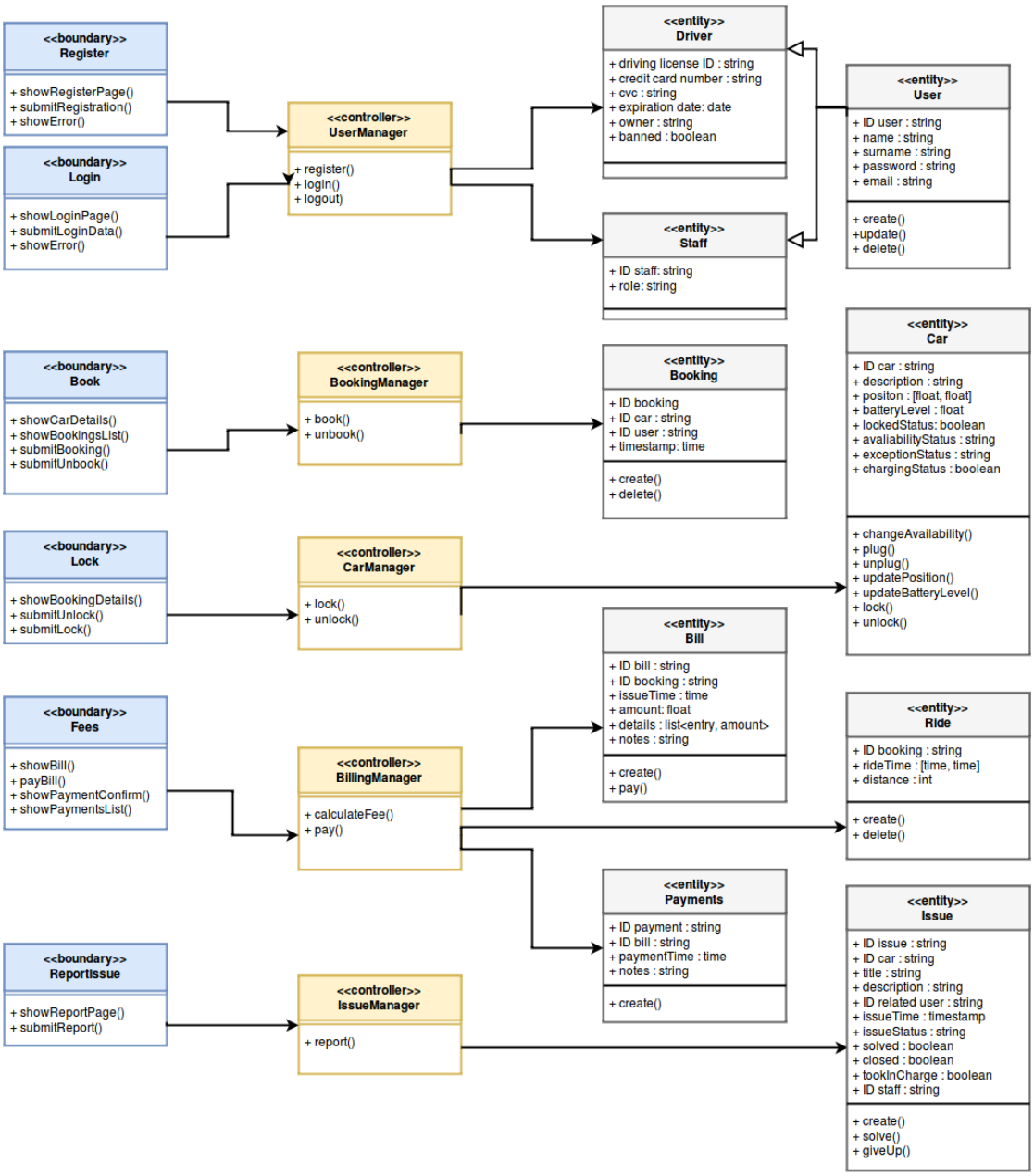

Figure 16: BCE Diagram of the interface of customer's application

5 Requirements Traceability

In this secton 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
- **LaTeX** on TeXworks to redact this document
- **www.draw.io** to draw UML graphs
- **LibreOffice Draw** to draw the system's overview at section 2.1

6.2 Hours of work

- SZ: 1h on 30/11
- SM: 2h on 1/12
- SM: 5h on 2/12
- SZ: 5h on 2/12
- SZ: 3h on 4/12 (RASD review)
- SM: 6h on 5/12
- SZ: 3h on 5/12
- SM: 5h on 6/12
- SZ: 4h on 6/12
- SM: 10h on 7/12
- SZ: 7h on 7/12
- SM: 8h on 8/12
- SZ: 10h on 9/12
- SM: 9h on 10/12
- SZ: 12h on 10/12
- SM: 7h on 11/12
- SZ: 7h on 11/12

Changelog

- **v 1.1**
  - Fix description of Figure 7
- **v 1.2**
  - Add RASD as a reference document.
- **v 1.3**
  - Add cover page.