Design Document

Simone Mosciatti & Sara Zanzottera

December 9, 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 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 fronted, 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 maximun amount of time a car can be reserved.

Driver Whoever is driving a regularly booked PowerEnJoy car.

Passenger Whoever is in inside a PowerEnJoy car but is not the driver.

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 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.

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 command 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 each node should provide to the world and to other nodes in order to accomplish such goals.

Once the interfaces are identified, we proceed organizing those interfaces into higher-level components, caring at respecting the Single Responsability principle in order to provide highly decoupled and reusable components.

Once components have been defined, we organize them into an high level architecture and proceed deploying them on the physical nodes of the system. Then we define in detail how they interact with the help of some Sequence Diagrams.

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

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 users and the system will be performed and how to provide them.

These are nemed **UI/FunctionalityName** in order to highlight that these functions are related to the interface with users. Indeed a deeper and detailed design of the actual UI is carried out in section 4: User Interfaces.

As we are going to see, many use cases require some specific funcionalities to be provided by the backend. These are named **Sy/FunctionalityName**, where "Sy" indicate an abstract system that we are going to model in detail in the next paragraphs.

UI/Registration

  Description Users can register to the system.

  Requires
  - Sy/Register
  - Sy/Validate

UI/Login

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

6

Requires

- Sy/Login

UI/Lookup

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

Requires

- Sy/GeoLocationCars
- Sy/PositionUser

UI/Book

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

Requires

- Sy/Book

UI/Unbook

Description Users can remove a previously made reservation.

Requires

- Sy/Unbook

UI/Unlock

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

Requires

- Sy/PositionUser
- Sy/PositionCar
- Sy/Unlock

UI/Ride

Description Users can ride a reserved car.

Requires

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

UI/SafeAreas

Description Users can locate safe parking areas while driving.

Requires

- Sy/GeoLocationAreas

UI/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

UI/PowerStation

Description Users can locate and use charging station.

Requires

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

UI/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

UI/Payment

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

Requires

- Sy/MakePayment
- Sy/SetPaymentMethod

UI/ReportIssues

Description Users can report issues to the system.

Requires

- Sy/ReportIssue

UI/FindIssues

Description The staff can locate cars that need their intervention.

Requires

- Sy/GeoLocationIssues

UI/Support

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

Requires

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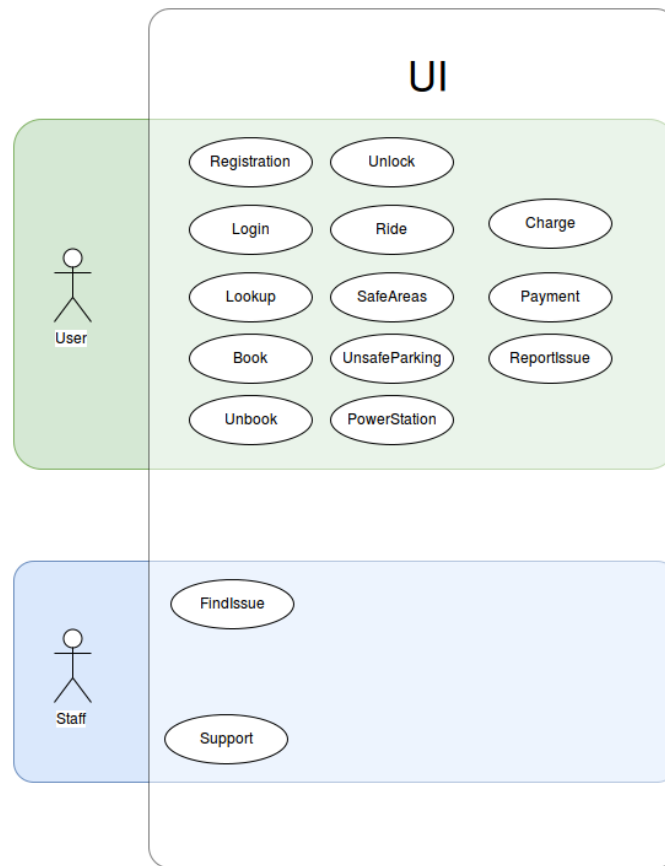- Sy/TakeChargeIssue
- Sy/SolveIssue
- Sy/GiveUpIssue

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

2.3 Components Design

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

In order to clarify how previously defined "Sy/FunctionalityName" maps with the component's specific functions, we used a notation **COMPONENT/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 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 ⇔ 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/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.

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 iteractions 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/Rides

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.

RIDE

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.

BILLING_SYSTEM

Responsability Manages all the fees.

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 user and 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/SendFee ⇔ Sy/SendFee

Responsability Ask for payment to the user before to actually execute the transaction.

Input The ID of the user and the total to pay.

Output The user is ask to confirm the payment.

NOTIFY/UnsafePark ⇔ Sy/UnsafeParkNotification

Responsability Communicate at the user that he parked in an usafe area.

Input The ID of the user.

Output The user is prompt to acknoledge that he is aware to have parked in an unsafe area.

NOTIFY/CarPlugged ⇔ Sy/CarPluggedNotification

Responsability Communicate at the user that he succesfully plugged the car into a power station.

Input The ID of the user.

Output The user is prompt to acknoledge that he plugged the car into a power station.

Qui un Components View simile a quello a pagina 8 dell'esempio... possibilmente con un font un po' piu' grande che nell'esempio.
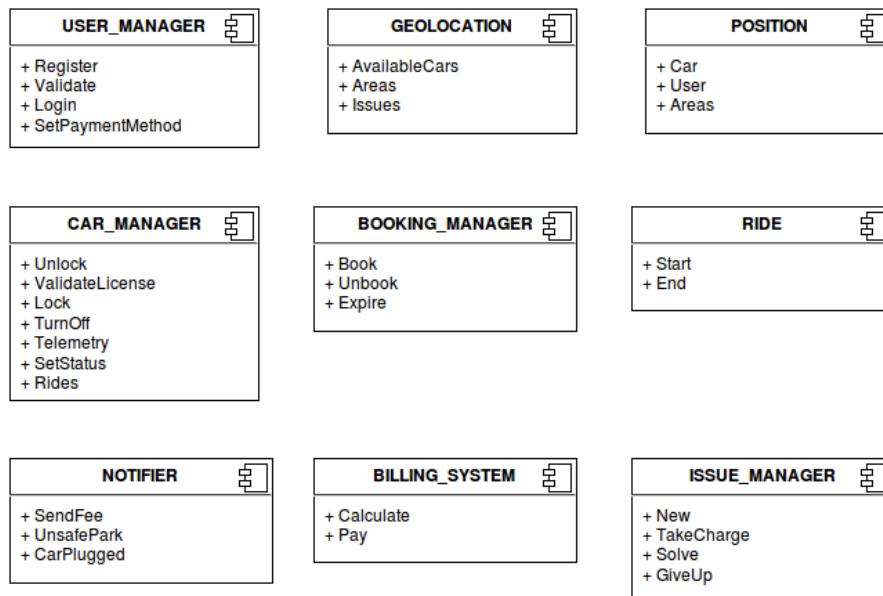Non so bene se qui o in fondo alla sezione successiva.

15

Figure 2: Graphical visualization of the abstract components needed.

Motivations

In this part of the design we made several choice where alternative approach may have been considered as well; we are going to motivate some of those choices.

Notifier

To notify the user about something we could have 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 the situation of a dynamic world where the need of the software always evolves our choice makes more sense.

1. Each component that needed to notify the user could have its own NOTIFIER (eg. instead of NOTIFY/UnsafePark we could have something like CAR/NotifyUnsafePark), indeed as more component will need to notify the user more and more different version of the same functionality risk to be reimplemented in different part of the code base, a problem we would like to avoid.

2. The NOTIFIER itself could have been more dynamic (eg. instead of multiple functionality NOTIFY/SendFee, UnsafePark, CarPlugged we could just have a single NOTIFY/Notify). This option while appealing risk to provide too much freedom to the implementators and it may be abused or not used correctly.

Overall we believe that our choice require to think carefully before to add a new notification, but, at the same time, having everything in the same component, will make the actual implementation capable to re-use pre-developed pieces easily.

2.4 From Abstract Components to Physical Nodes

Now that we have identify the abstract components that, together, provide the fucntionality of our project we are going to makes those components real.

In order to actually implements the components we - first - need to know where those components will be deployed and the form of comunication between and intra them.

In the next sub section we are going to explain what physical nodes will compose our system.

Physical structure

Overall there are 3 main identifiable nodes in our system:

- Main Server
- Car Onboard system
- User interface (App, web site, etc...)

We have refer to the User Interface as App throught the whole document and we will keep such terminology for sake of clarity, however, 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".

2.5 Communication between nodes

Now that our nodes are defined we need to understand the comunication between them.

It would have been possible to make each node comunicate with each other node, however this raise some concern about the security of the system, especially for the communication between the users and the cars. 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 channel, one between the main server and the user, and another between the main server and the cars.

Actual channel implementation

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 servet, 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.

2.6   Actual deploy of components into nodes

Now that we have identify the components that we need to deploy, the nodes where those components can be deployed and the communication mechanism between the nodes we are ready to actually decide where to put each component.

Most of the functionality will be invoked by the user, however, logically, the componentthey will reside on the main server.

Some component will be shared between the main server and the user or the car or both.

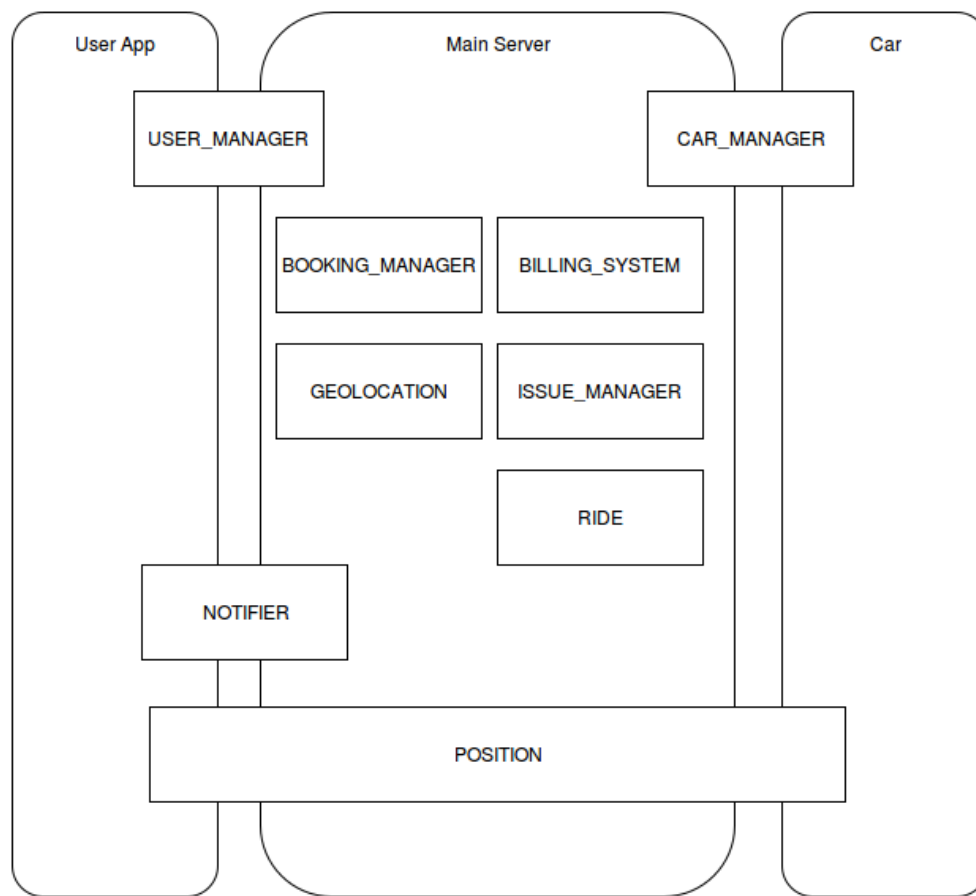Figure 3: Graphical visualization of the abstract components needed.

REEXPLAIN WHY SOME COMPONENTES ARE SHARED

2.7 Main Design Choices

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

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.

Now we list the design choices we made, first of all, to enforce the architecture identified above.

Car to app communication

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

The main server is always an intermediary for app-to-car and car-to-app communications, in order for the system to have always 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.

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

Communication protocols

More specific choices has been made on the communication protocols too.

A pure **Client-Server** communication protocol is implemented only for server-to-app and app-to-server communications, as they seem to naturally fit this model.

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 servet, and a PubSub pattern achieves high troughput and low latency.

To describe the communication approach in terms of the protocol itself, the car publishes messages about it own status and the server subscribes to those messages. The server is meant to host brokers too.

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.

Here we provide a high-level diagram of the system and some proposed technologies that we will discuss in the last sections of the document.
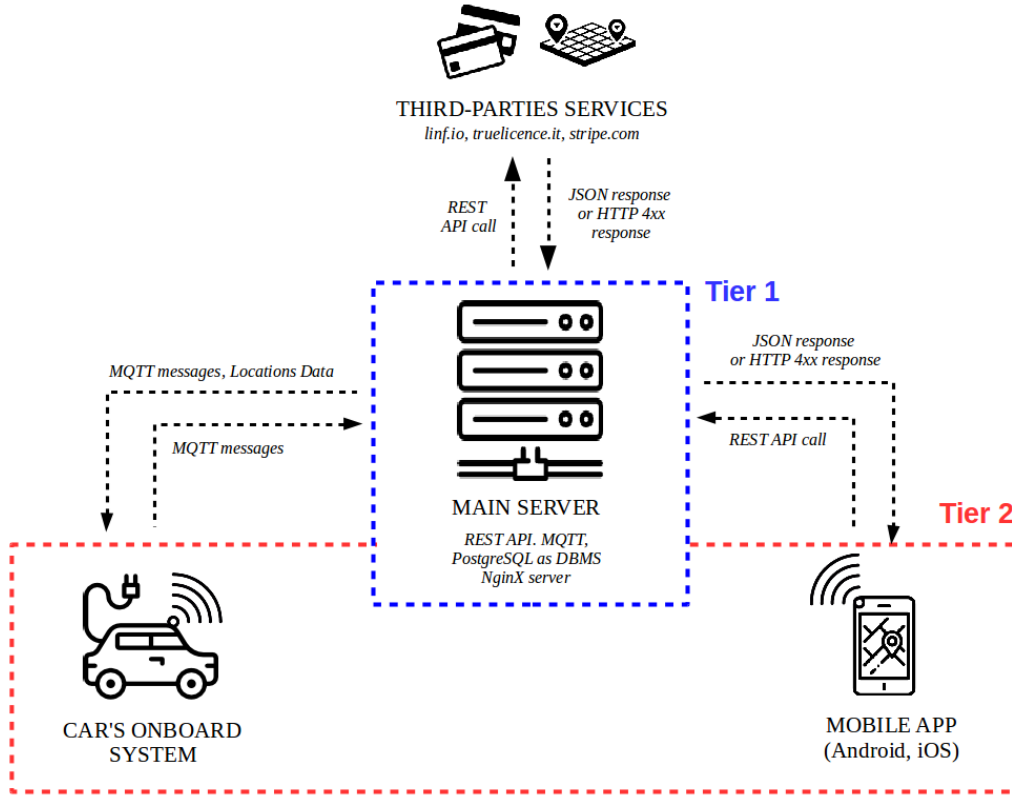

Figure 4: High-level organization of the system's required elements

2.8   High-Level Architecture

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

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

LOCATION

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

POSITION

Considering that all its three functions are exposed as API by the server, the component is deployed on the server entirely. Define this better

BOOKING_MANAGER

Considering that all its three functions are exposed as API by the server, the component is deployed on the server entirely. What about EXPIRE?

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/Rides is exposed by Server to Apps
- 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/Rides
- CAR/ValidateLicense

LOCAL_CAR_MANAGER

Comprises all the functions exposed by the cars:

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

BILLING_SYSTEM

Considering that all its three functions are exposed as API by the server to the customer's apps, the component is deployed on the server entirely. Recheck PAY : is it on server or on the app?

ISSUE_MANAGER

Considering that all its three functions 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.
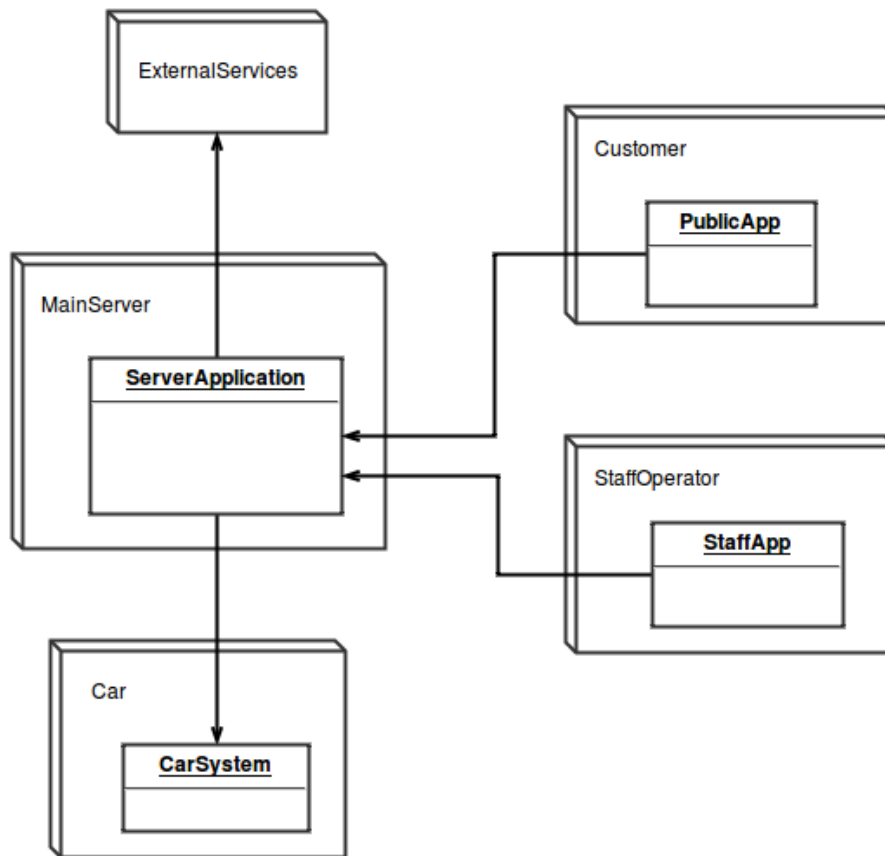

Figure 5: Higher level components of the system
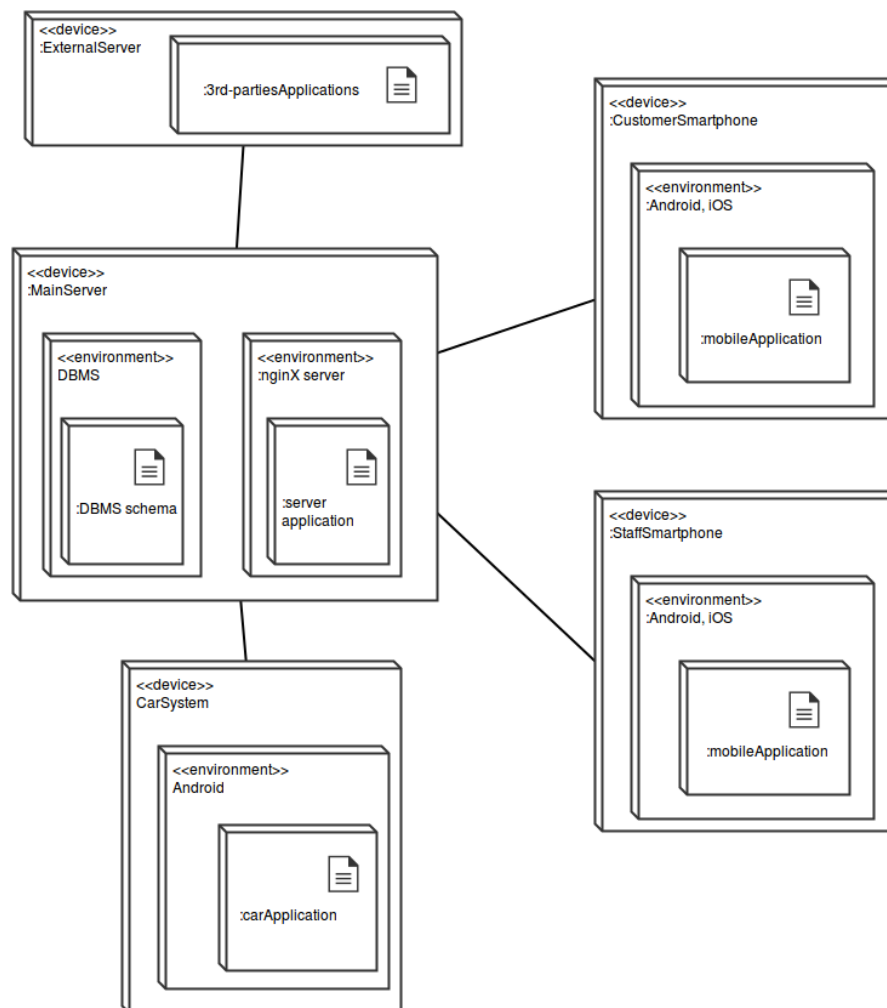
2.9   Deployement View

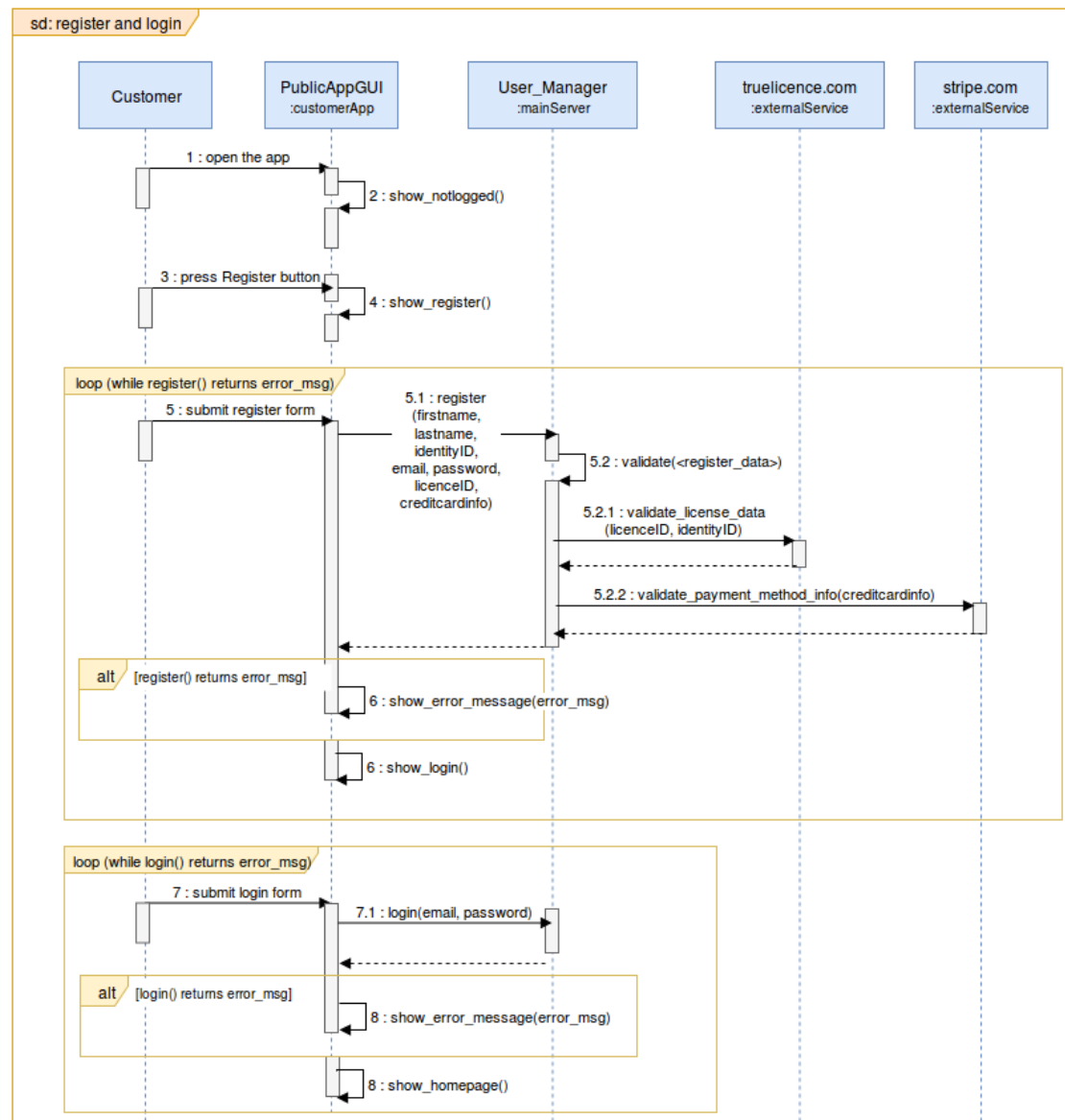Figure 6: Deployement View of the system

2.10 Runtime View

Figure 7: 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).

2.11 Selected Technologies

Having defined the interface between the several functionality provided by the components, all possible communication technologies may have been choosen. 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 constrainted 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.

Distributing the main server

what about this?

Database

what about this?

3   Algorithm Design

[Definition of critical sections of code]

4 User Interface Design

4.1 Mockups

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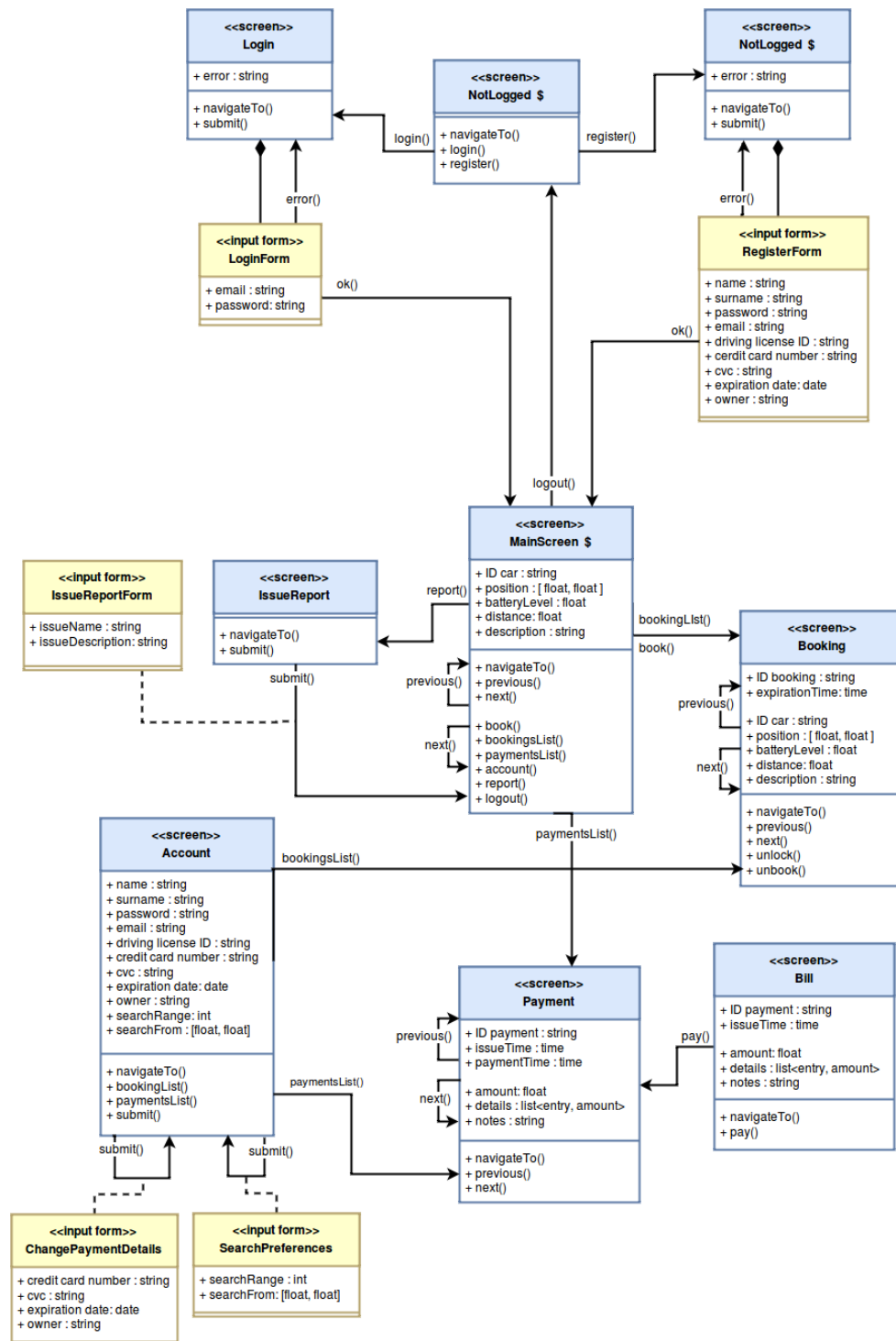
4.2 UX Diagrams

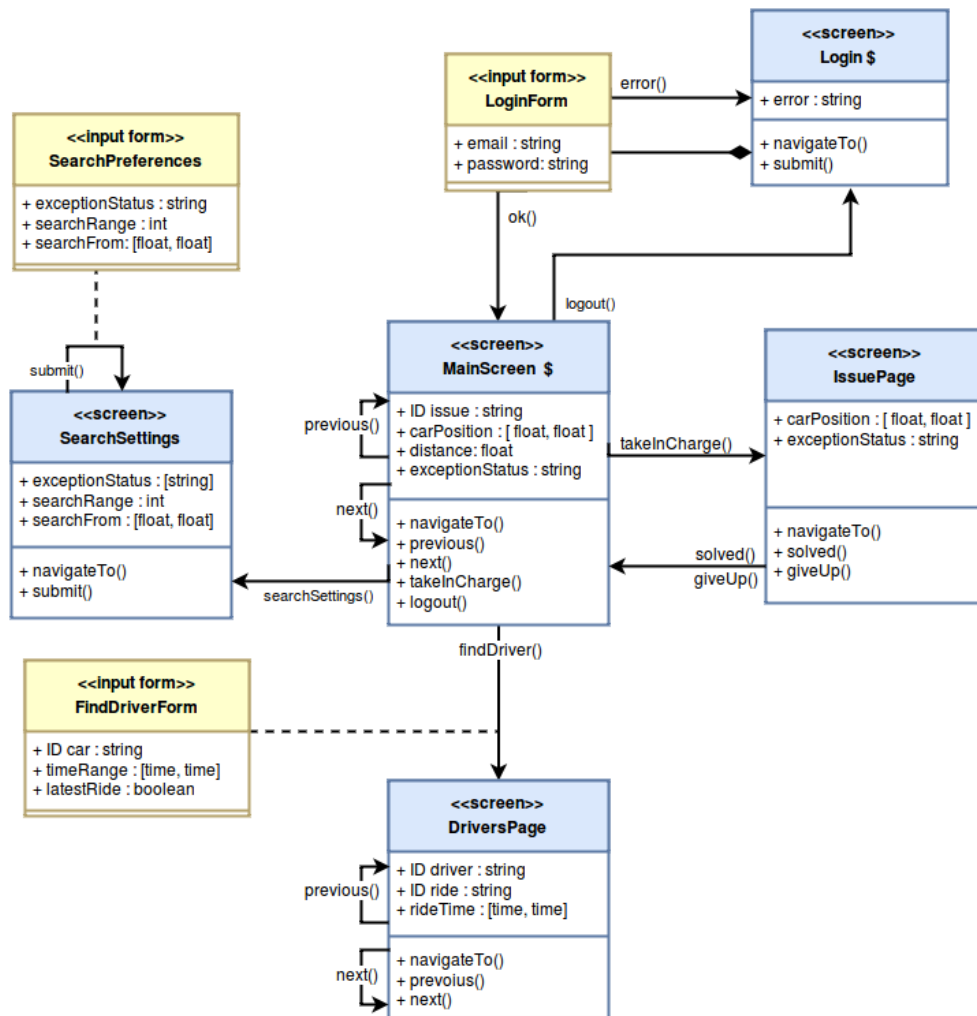Figure 8: UX Diagram of the interface of customer's application

Figure 9: UX Diagram of the interface of staff's application
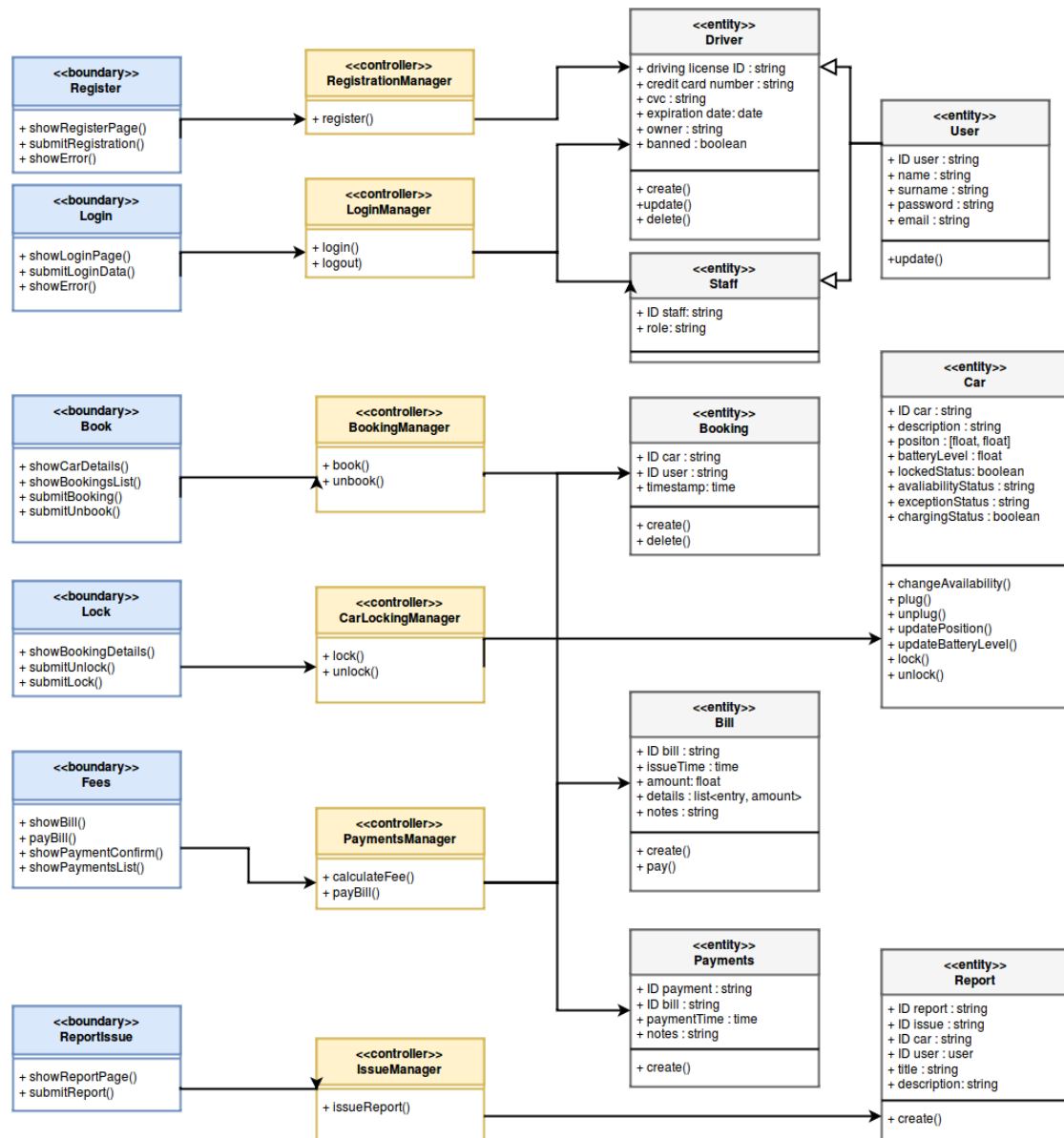
4.3 BCE Diagrams

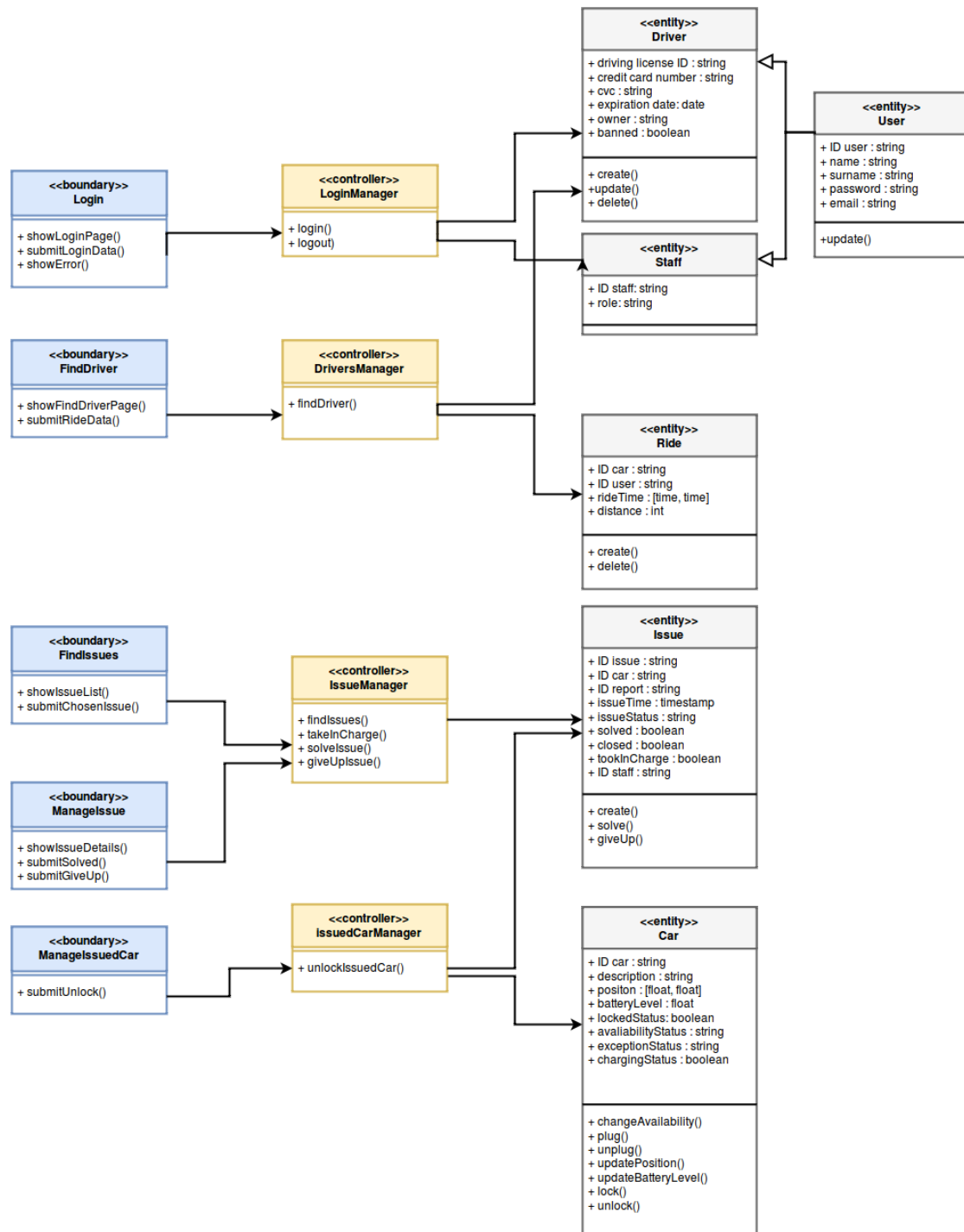Figure 10: BCE Diagram of the interface of customer's application

Figure 11: BCE Diagram of the interface of staff's application

5 Requirements Traceability

Lo stesso lavoro fatto nel definire le interfacce, ma a rovescio. Non piu' Interfaccia -¿ Req soddisfatti, ma Req-¿ interfaccia che lo soddisfa.

Considering we designed the system using a bottom-up approach, designed components maps in a straightforward way to the goals specified in the RASD. Hoever we provide an explicit mapping of the two.

REGISTRATION Users can register to PowerEnJoy .

- Server: RegistrationController
- Customer's App: RegistrationView (?)

LOGIN Users can login to PowerEnJoy .

- Server: LoginController
- Customer's App: LoginView (?)
- Staff's App: LoginView (?)

LOOKUP Users can find cars nearby a given position, it could be its position or a point in the map.

- Server: CarsLocation
- Customer's App: CarsView (?)

BOOK Users can book a car for a short amount of time.

- Server: BookingController
- Customer's App: BookingView (?)

UNLOCK When users are in proximity of the car they booked, the system can unlock it.

- Server: CarLockController
- Car: LockController
- Customer's App: CarNearby (——————— SEE ISSUE #14)

RIDE Users can drive to their destination.

- Server: AuthDriver
- Car: LicenceScanner, EngineController, MQTT publishers, CarGUI

SAFE_AREAS Users can locate safe parking areas.

- Server: AreasLocation
- External Services: linf.io

- Car: MQTT publishers, CarGUI

UNSAFE_PARKING The system must react to an unsafe parking.

- Server: UnsafeParkingController, IssuesController (?)
- Car: EngineController, LockController (MQTT publishers too?)

POWER_STATIONS Users can locate charging stations.

- Server: AreasLocation
- External Services: linf.io
- Car: CarGUI

CHARGE At the end of the ride, users are charged a fee.

- Server: BillingController
- Customer's App: PaymentDetailsView

PAYMENTS Users can pay bills through the app.

- Server: PaymentController
- External Services: stripe.com
- Customer's App: PaymentDetailsView

FIND_ISSUES The staff can locate cars that need their intervention.

- Server: IssuesLocation
- Staff's App: IssuesView (?)

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

- Server: IssuesController
- Staff's App: IssueDetailsView (?)

FINES The system can provide enough details for the staff to manage correctly the fines they receive from local authorities.

- Server: FindDriverController (?)
- Staff's App: FindDriverView (?)

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

- **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 of work on 4/12 (RASD review)

- SZ: 3h on 5/12

- SZ: 4h on 6/12

- SZ: 7h on 7/12