Design Document

Simone Mosciatti & Sara Zanzottera

December 13, 2016

Contents

1 Introduction

2 Architectural Design

2.1 Design Process Description

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.

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/CUST/Lookup

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

Requires

- Sy/GeoLocationCars
- Sy/PositionUser

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

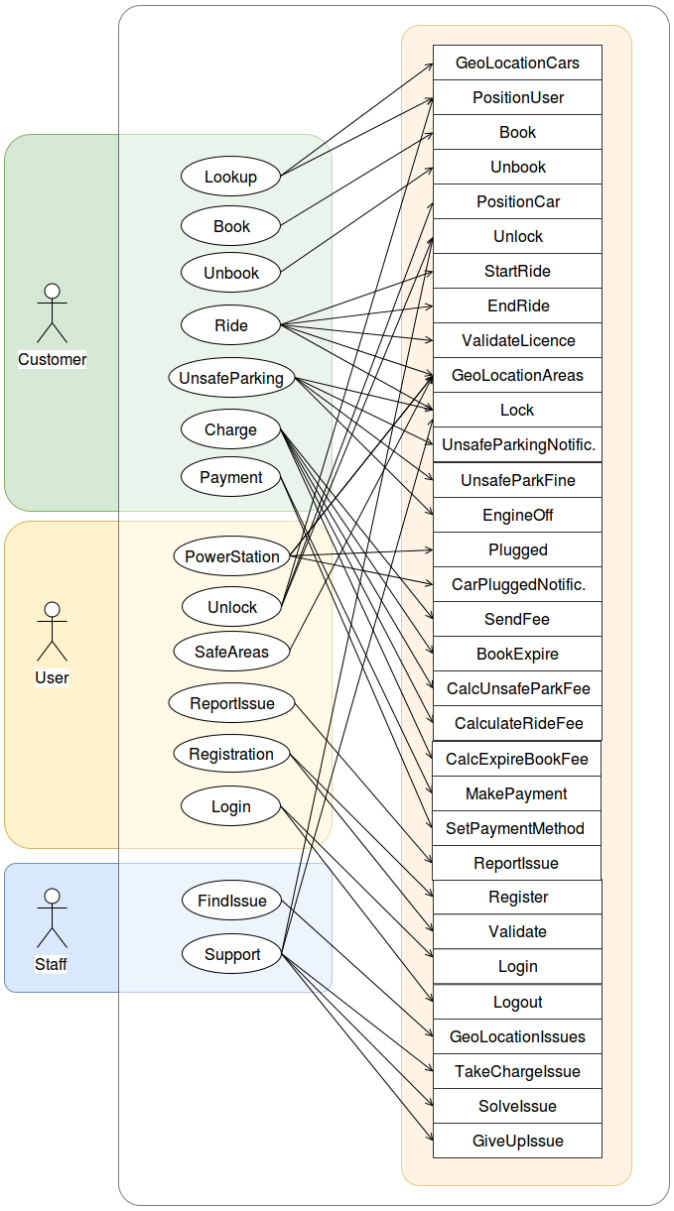

Figure 1: Graphical visualization of goals and system's required functions identified above, divided by tipology of users.

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/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.

POSITION

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

POSITION/User ⇔ Sy/PositionUser

Responsability Retrieves the position of an user.

Input The ID of the user.

Output The coordinates of the user.

BOOKING_MANAGER

Responsability Manages reservations.

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.

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.

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

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.

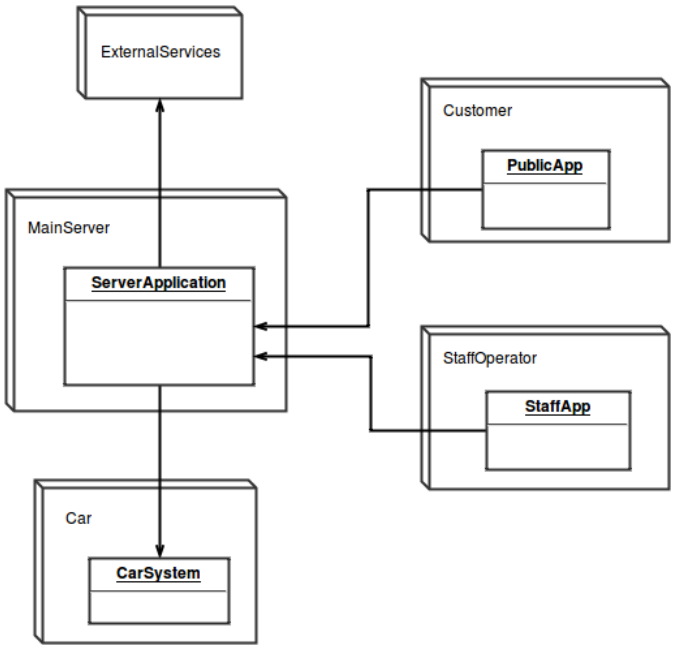

Figure 2: Higher level components of the system

Communication Strategy

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

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.

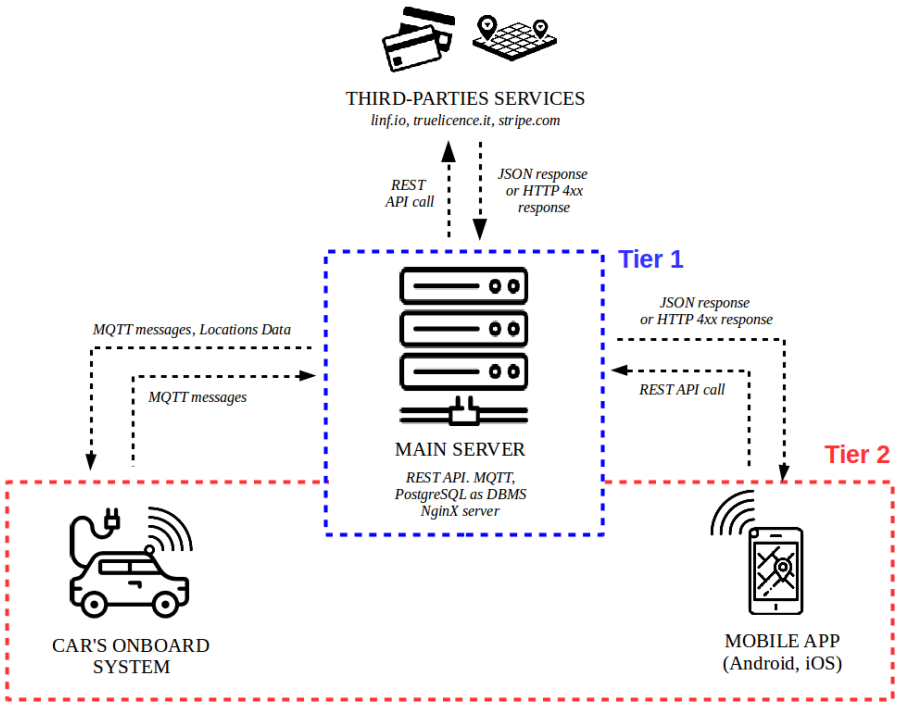

Figure 3: Communication design of the process (anticipating some technology choices we are going to justify in next sections).

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.

BOOKING_MANAGER
Considering that all its functionalities are exposed as API by the server, 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.

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

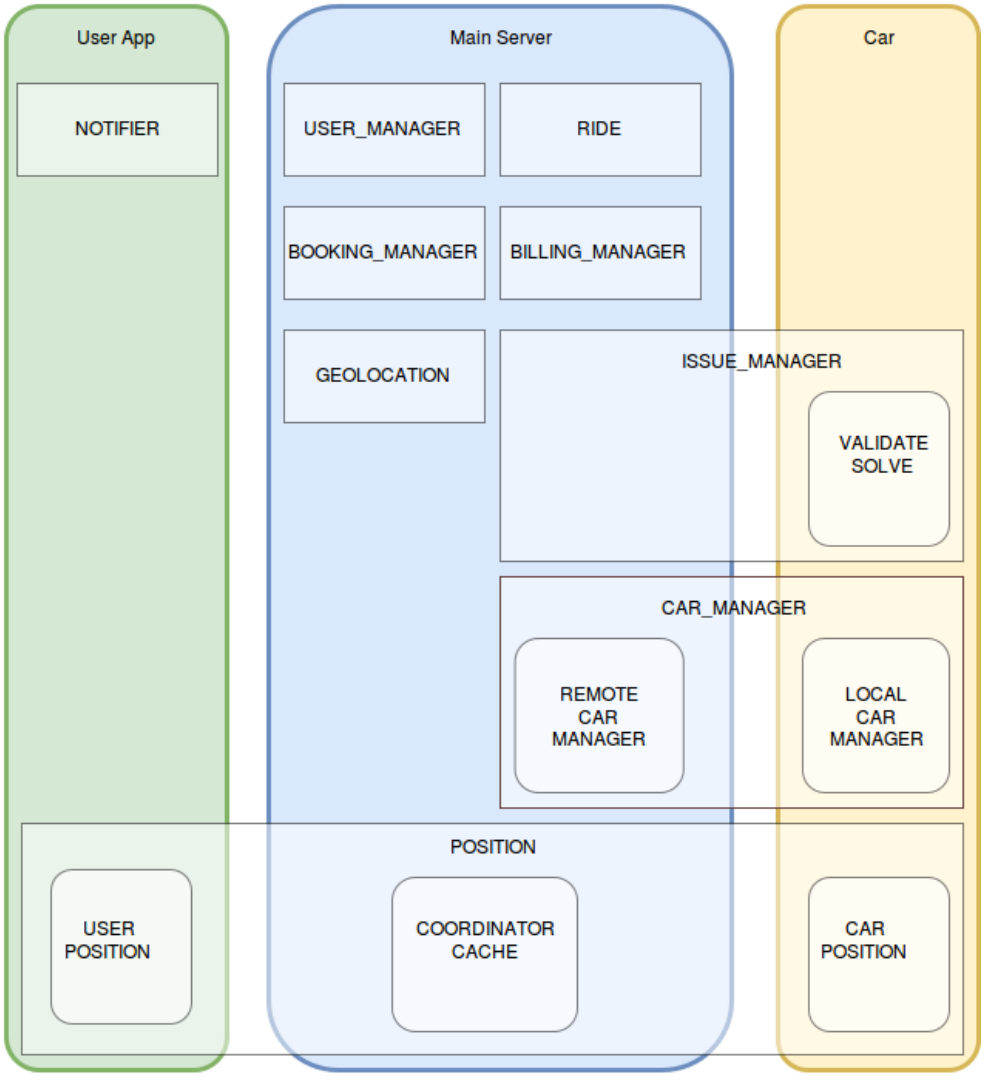

Figure 4: Graphical visualization of the abstract components needed.

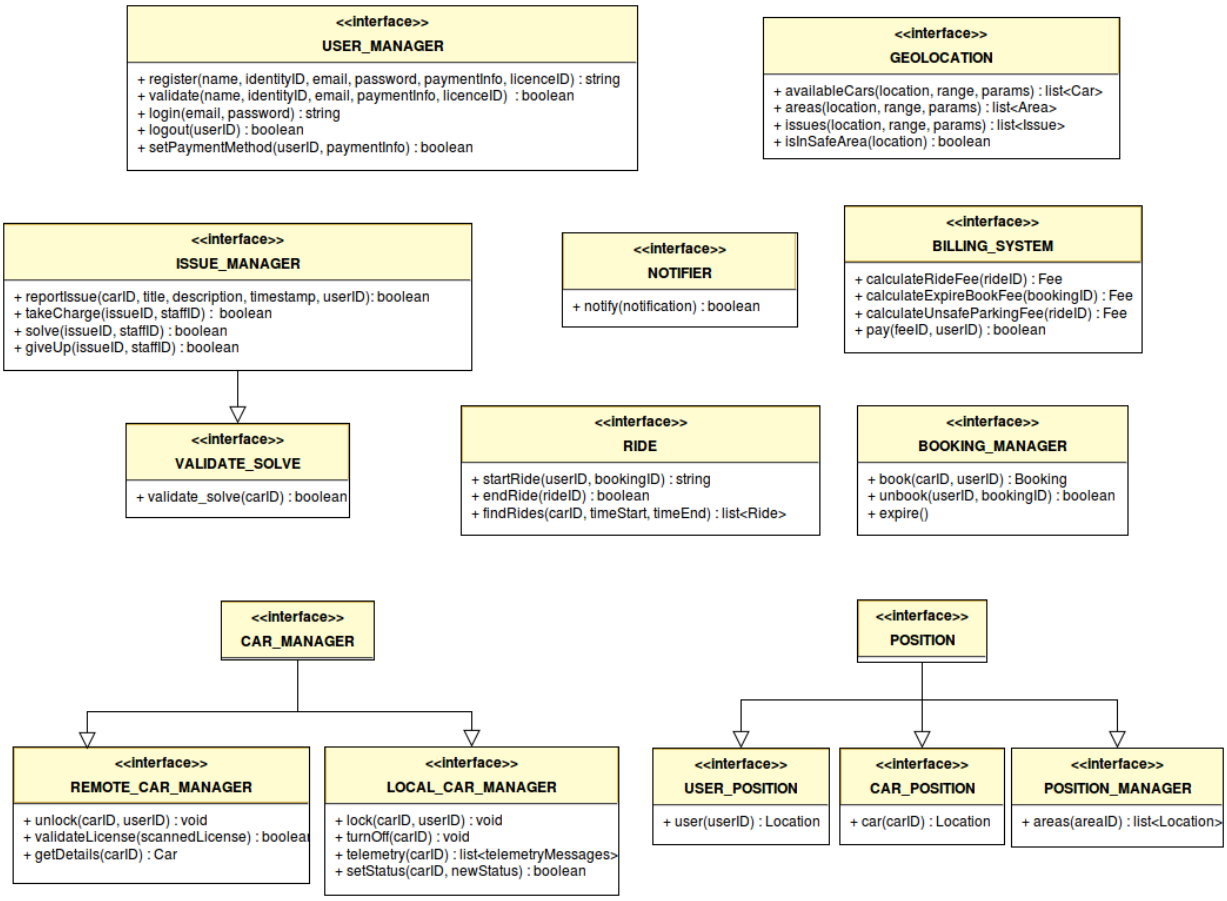

Figure 5: Description of the interfaces of the components identified above.

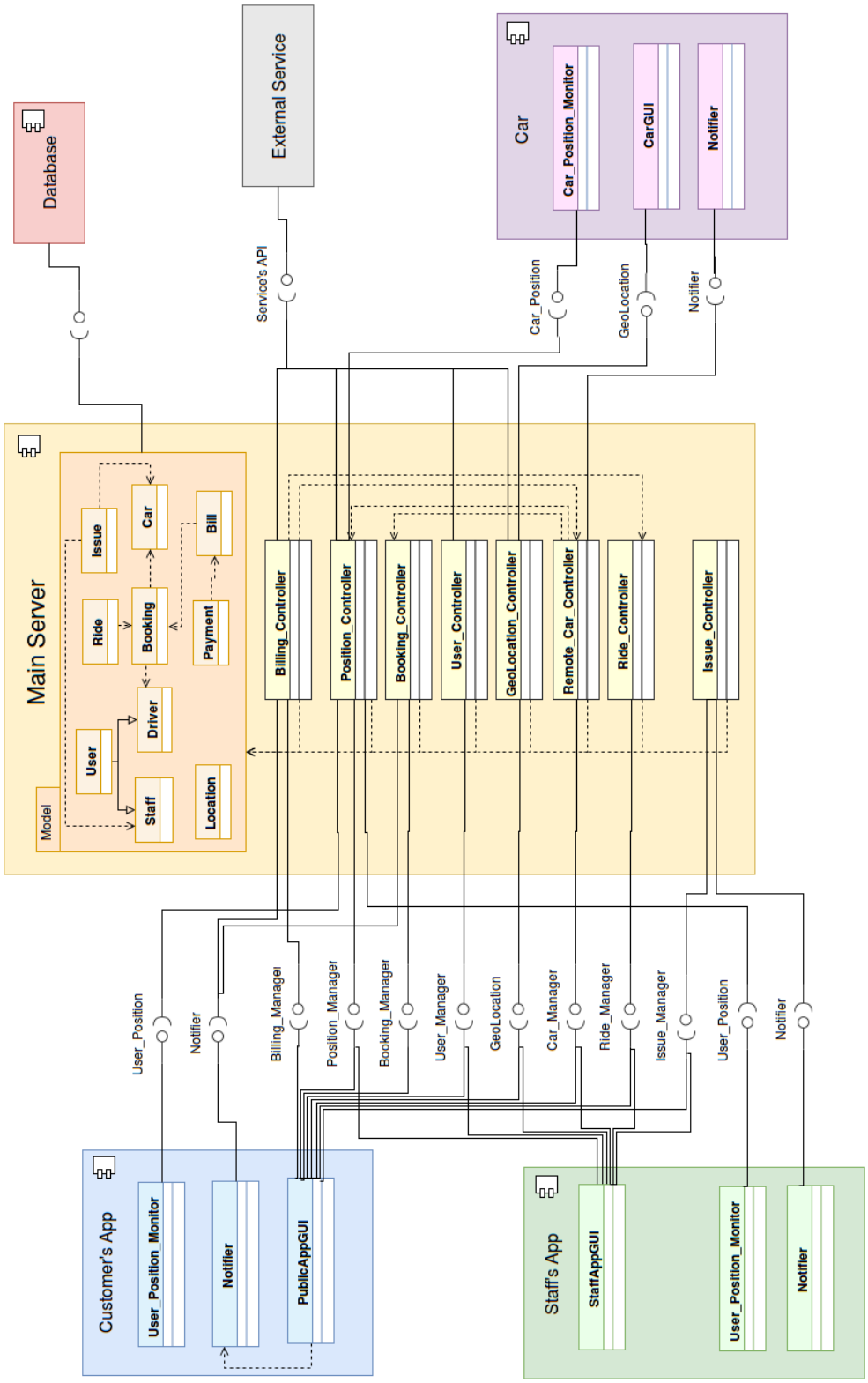

Figure 6: Components View Diagram of the entire system.

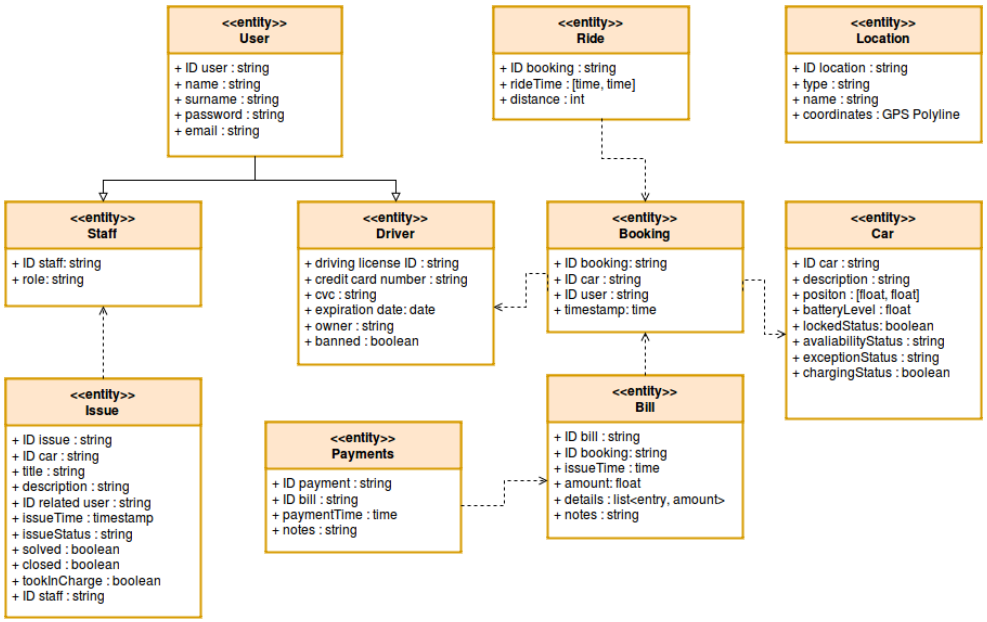

Figure 7: Model zoom of the Components View Diagram.

2.6 Runtime View

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

Register and Login

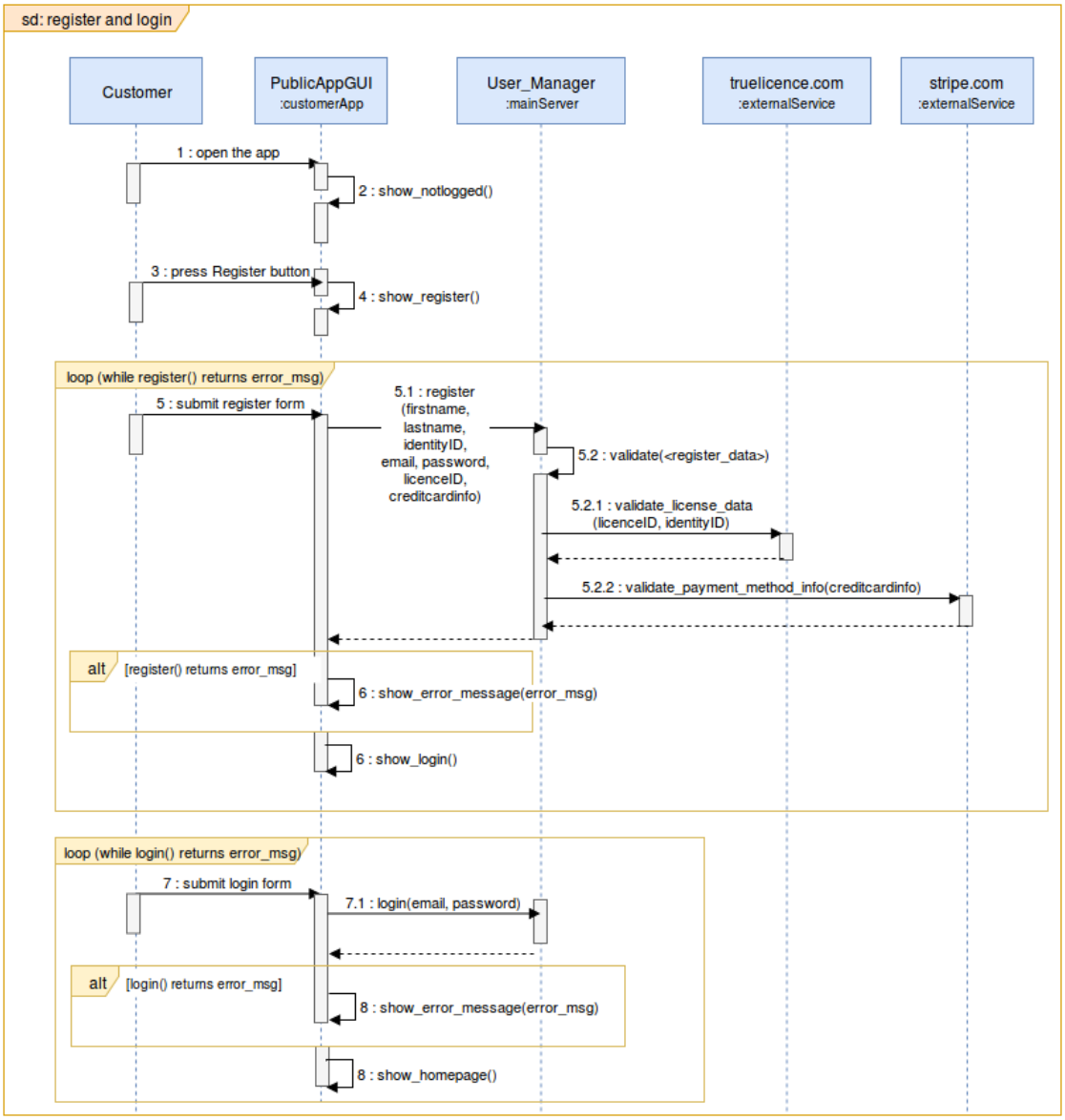

Figure 8: Sequence Diagram for Registration and Login process of customers.

Lookup and Book

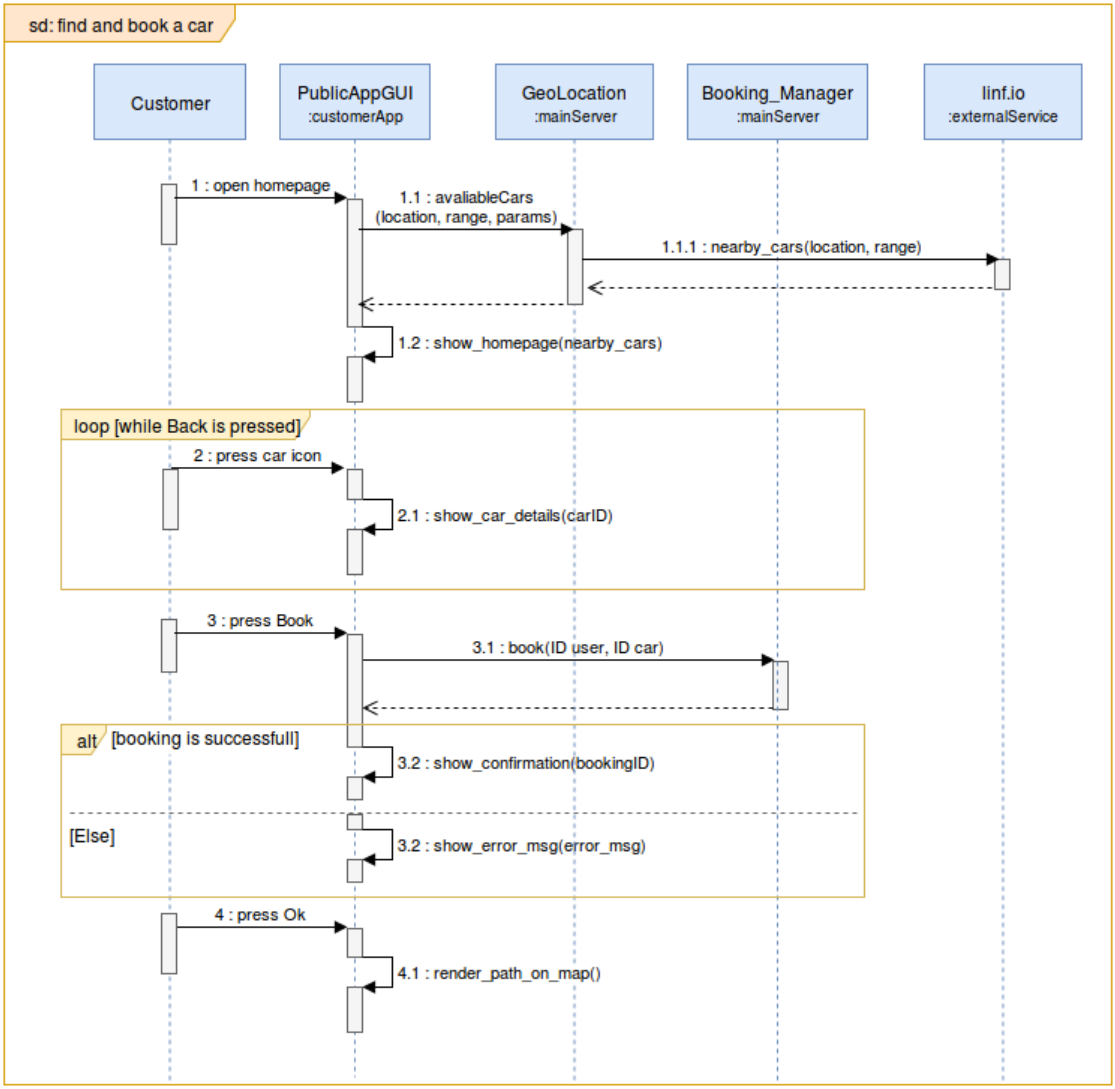

Figure 9: Sequence Diagram for Lookup and Booking process of customers.

Ride

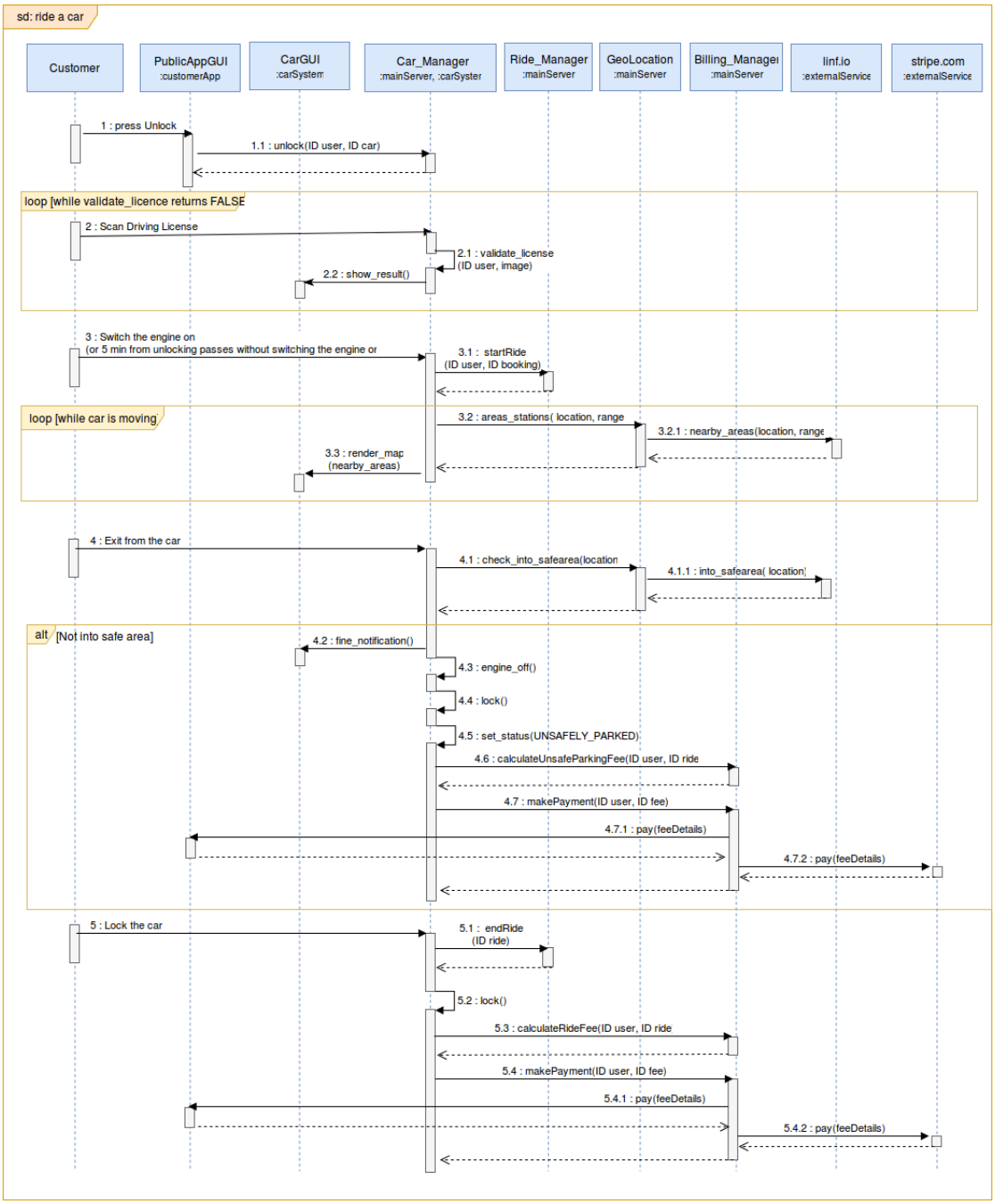

Figure 10: Sequence Diagram for the Riding process of customers.

Issue Management

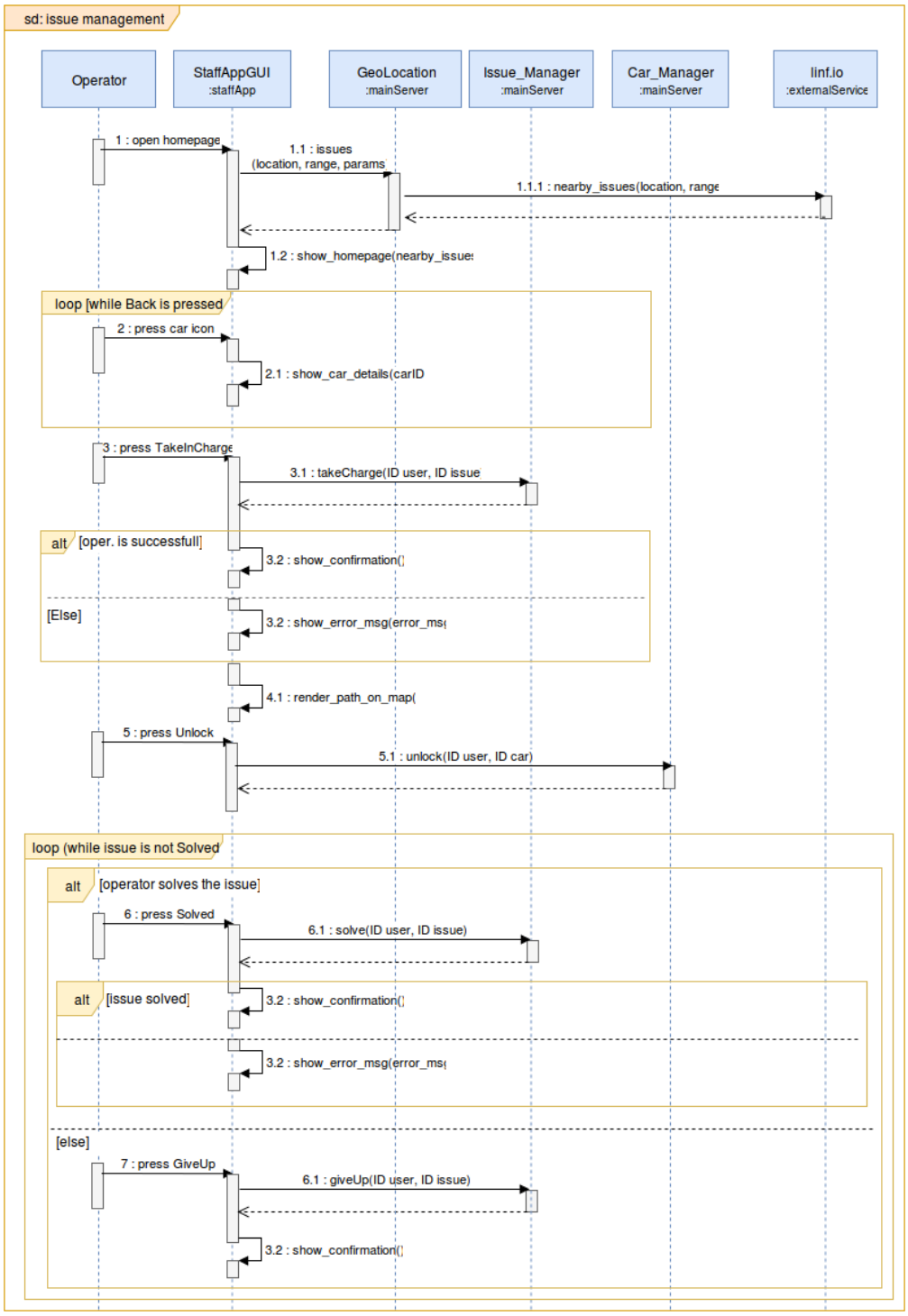

Figure 11: Sequence Diagram for the Issue Management process for staff operators.

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.

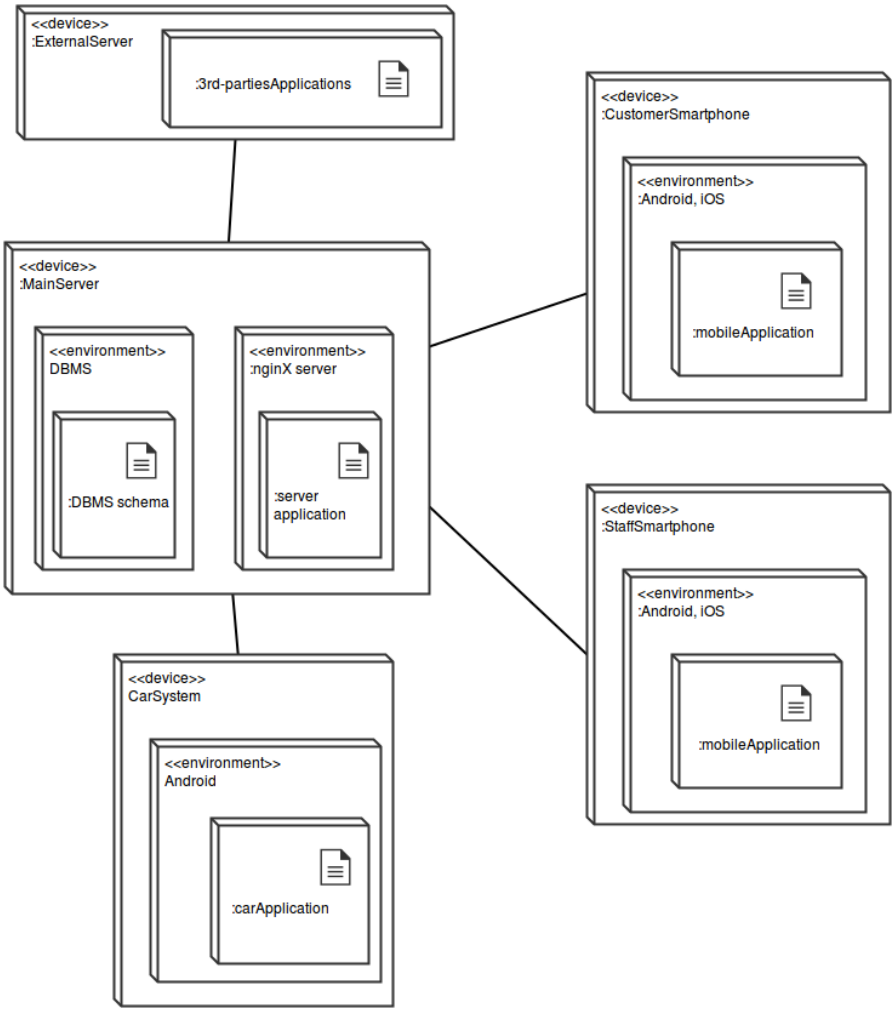

Figure 12: Deployement View of the system

2.8 Selected Technologies

RESTful API

MQTT

Nginx

PostgreSQL

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

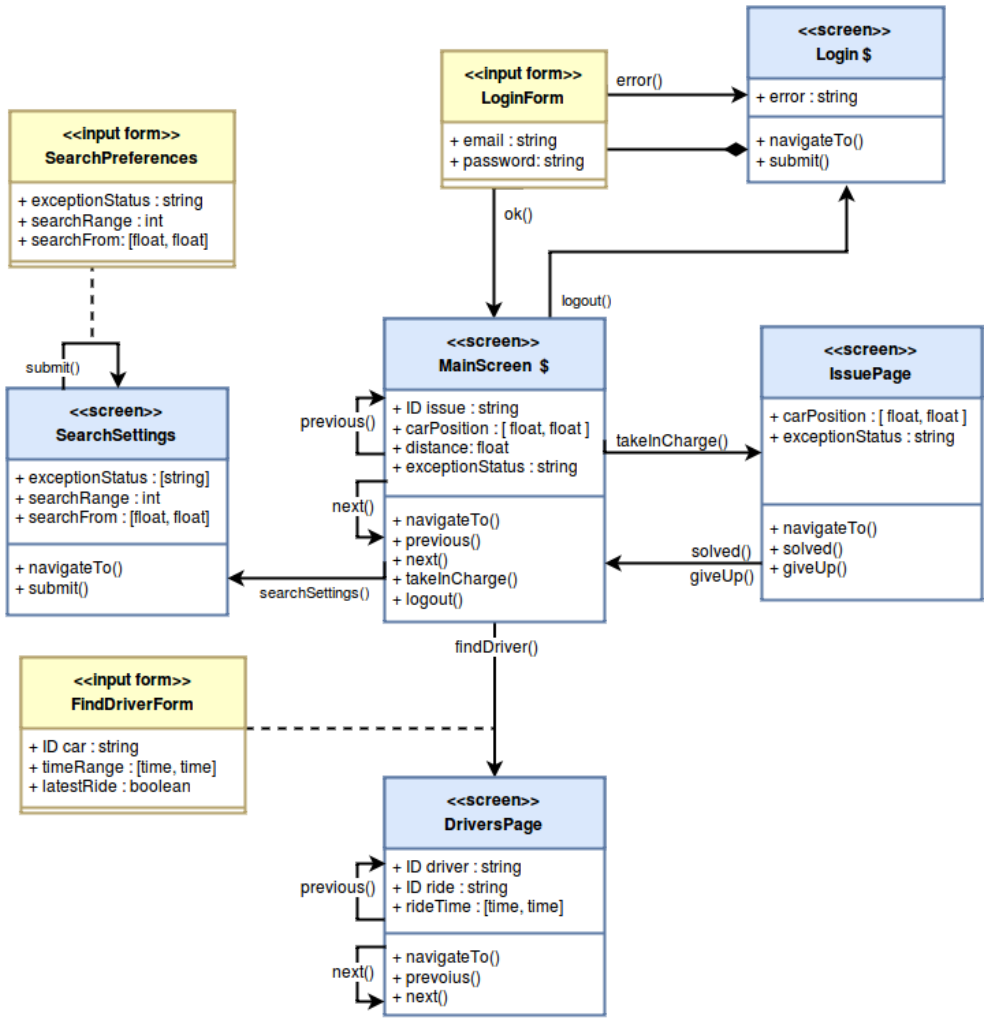

Figure 13: UX Diagram of the interface of staff's application

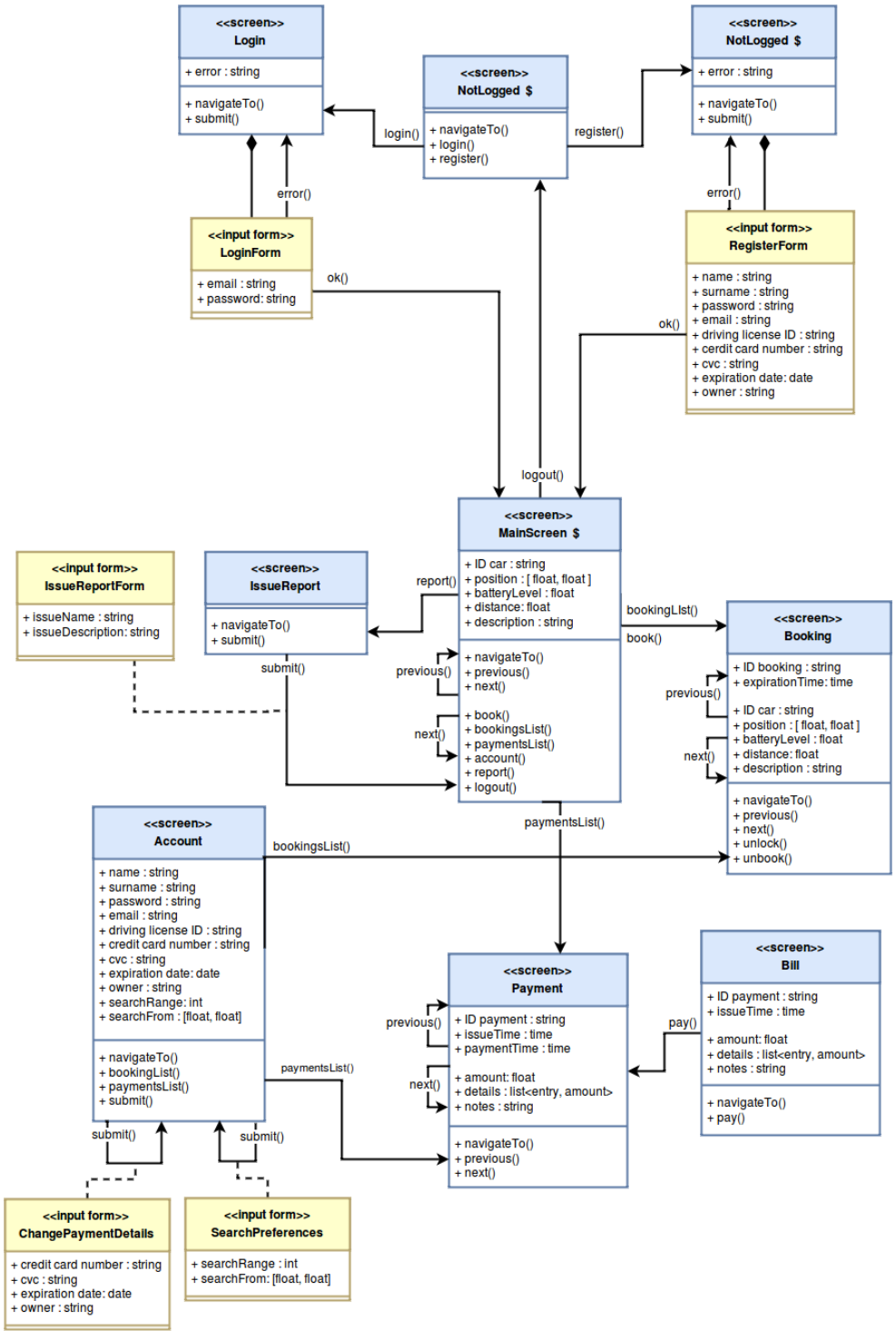

Figure 14: UX Diagram of the interface of customer's application

4.3 BCE Diagrams

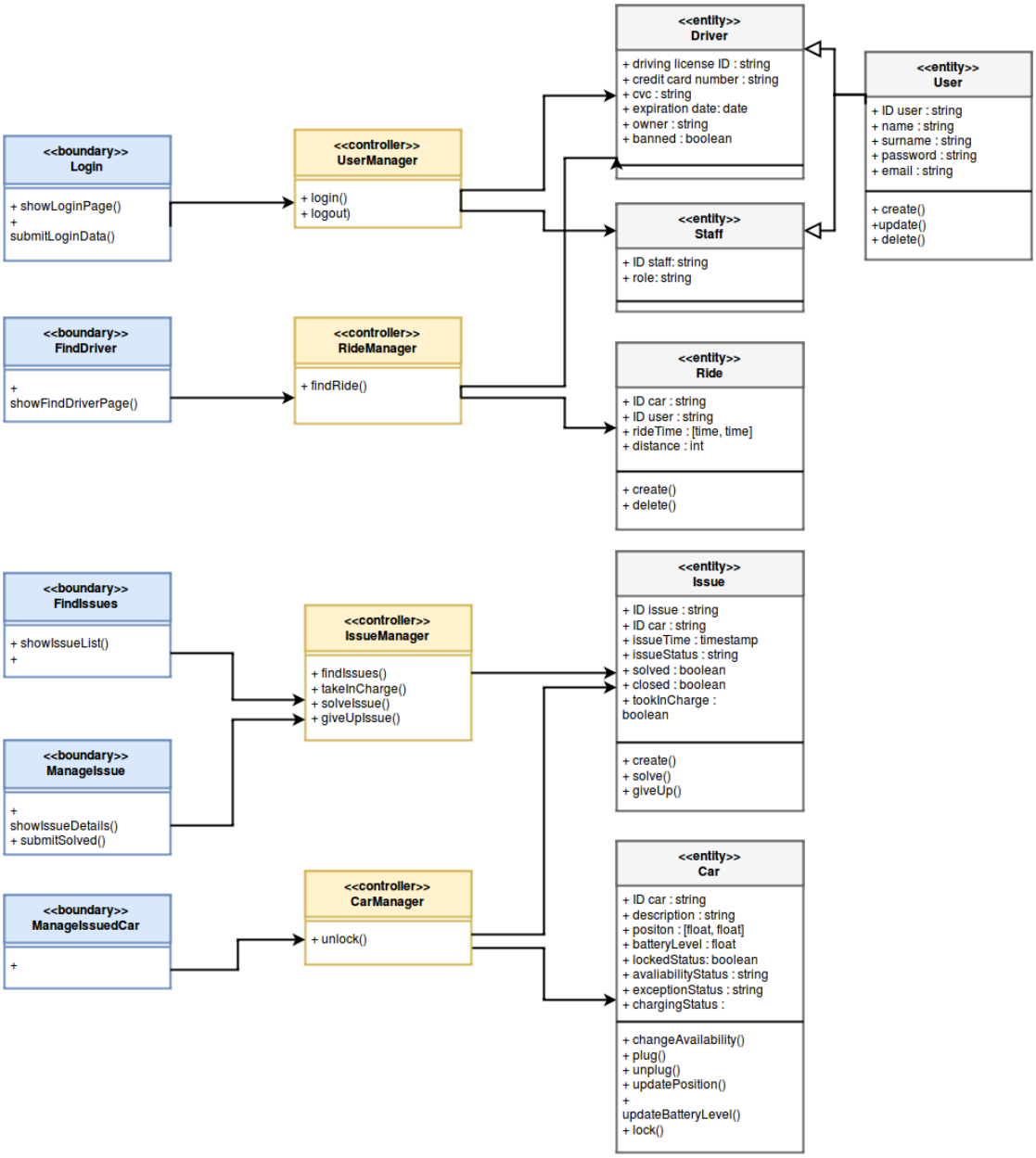

Figure 15: BCE Diagram of the interface of staff's application

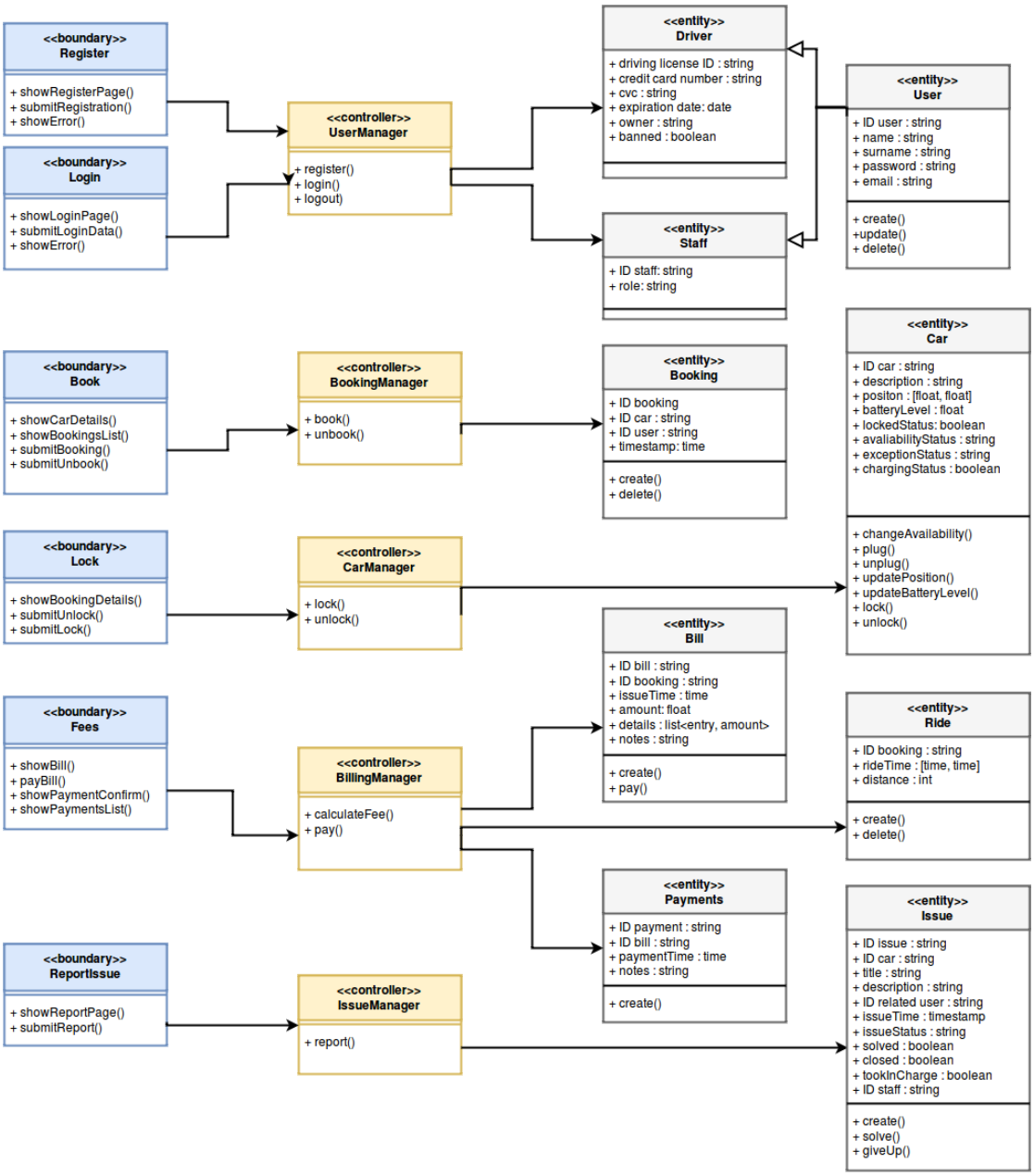

Figure 16: BCE Diagram of the interface of customer's application

5 Conclusions