Software Engineering 2 Project: PowerEnJoy

Design **D**ocument

Marco Ieni, Francesco Lamonaca, Marco Miglionico
Politecnico di Milano, A.A. 2016/2017

February 7, 2017
v1.1

Contents

Chapter 1

Introduction

1.1 Purpose

This is the Design Document of PowerEnJoy, a digital management system for a car-sharing service that exclusively employs electric cars. The aim of this document is to show our design choices and the rationale behind them.

We will analyse the components of the system and how they interact between each other. Furthermore, we will show the most important algorithms of the project, in order to underline the key aspects of their implementation. Finally, we will add details to the first description of the user interfaces that was described in the RASD.

1.2 Scope

The main goal of the system is to let the users easily look for and reserve a vehicle.

In particular we will present different views of the system representing different levels of abstraction: the user's point of view, an internal sight of the subsystems, their high level interactions and the communication interfaces that they use to interact.

We will describe the chosen architectural styles and pattern, some fundamental algorithms and how our choices are mapped on the requirements elicited in the RASD (Requirement Analysis and Specification Document).

The software system is divided into three layers: presentation, application and data. They will be presented in detail in the following chapters.

The architecture has been designed to be easily extensible and maintainable in order to provide new functionalities. Every component must be conveniently thin and must encapsulate a single functionality, in order to ensure an high cohesion. The dependency between components is unidirectional and coupling is avoided as much as possible in order to increase the reusability of the modules.

In order to simplify the system comprehension and to help the developers during the implementation phase, design patterns and architectural styles were used for solving the architectural problems of the system.

The Design Document would also demonstrate how the design will accomplish all the requirements captured in the RASD.

1.3 Definitions, acronyms, abbreviations

JPA: The Java Persistence API (JPA) is a Java application programming interface specification that describes the management of relational data.

MySQL: It is an open-source relational database management system,the term SQL is the abbreviation for Structured Query Language.

InnoDB: It is a storage engine for MySQL. MySQL 5.5 and later use it by default

ACID: It is a set of properties of database transactions: Atomicity,Consistency,Isolation,Durabili

ER: The entity–relationship(ER) model,it describes inter-related things of interest in a specific domain of knowledge

EJB: Enterprise JavaBeans (EJB) is a managed, server software for modular construction of enterprise software, and one of several Java APIs.

JPQL: The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.

JAX-RS: Java API for RESTful Web Services (JAX-RS) is a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.

RESTful: Representational state transfer (REST) or RESTful web services are one way of providing interoperability between computer systems on the Internet.

JSON: JSON (JavaScript Object Notation) is a lightweight data-interchange format.

RASD: Requirements Analysis and Specification Document.

DD: Design Document.

UI: User Interface.

Application Server: The layer which provides the application logic and interacts with the DB and with the front-ends

Font-End The components which use the application server services, like the mobile applications.

Back-End Term used to identify the Application server

1.4 Reference Documents

This document refers to the project rules of the Software Engineering 2 project, to the template for the Design Document contained into it and to Requirement Analysis and Specification Document (the previous delivered document)

1.5 Document Structure

This document is divided in five chapters:

Chapter 1. Introduction: It provides a general presentation of the Design document and of the system to be developed.

Chapter 2. Architectural Design: It goes into the detail of the architecture design of the system, describing its basic structure and the interactions of the main subsystems. It also contains the architectural style and pattern decisions description.

Chapter 3. Algorithm Design: It focuses on the definition of the most important algorithms of the system, independently from their concrete implementation.

Chapter 4. User Interface Design: It shows how the user interfaces of the system will look like and behave, by means of mock-ups and user experience modelling.

Chapter 5. Requirements Traceability: It explains how the requirements we have defined in the RASD map to the design elements that we have defined in this document.

Chapter 2

Architectural Design

In this chapter we provide a comprehensive view over the system components, both at a physical and logical level. The system will be firstly described at a very high level, showing the different components and how they interact (section 2.1). Then the system will be described and detailed in the section 2.2, following a top-down approach.

In section 2.3 we will focus our attention more on the physical level, analysing the deployment of the system on physical tiers, while in section 2.4 we will describe the dynamic behaviour of the system.

Furthermore, section 2.5 will focus on the interface between the different components of the system. Finally, the design choices and patterns used will be presented and discussed in section 2.6.

2.1 Overview

In this section we will present the high level components of the system and their interaction:

Mobile application: This presentation layer consists in the mobile client. It communicates directly with the application server.

Application Server: This layer contains all the application logic of the system. All the policies, the algorithms and the computation are performed here. This layer offers a service-oriented interface.

Database: The data layer is responsible for the data storage and retrieval. It does not implement any application logic. This layer must guarantee ACID properties.

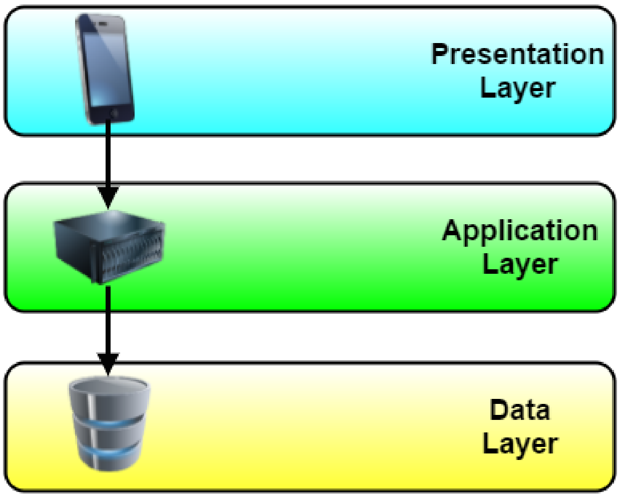


Figure 2.1: Layers of the system.

HandyCar board: A board to which all the actuators and sensors of the car are connected. It communicates with the application server, abstracting all the low-level details of the cars.

On-Board tablet: A tablet that is on board of the vehicles. It used for the communication between the system and the driver, so the tablets communicates with the application server.

PowerEnJoy Operator Program: A program that the operators of PowerEnJoy can use to interact with the system, for example modifying the safe areas. It communicates with the application server.

External Systems: There are three external systems:

- **External System for Payment:** checks the validity of the payment information and handles the payment process;
- **External System for Driving License Validation:** checks if the driving license informations provided by the user;
- **External System for Maintainers:** signals to the maintainers the unavailable cars;

The system is structured in three layers as we can see in picture 2.1.

This design choice makes it possible to deploy the application server and the database on different tiers. It also improves scalability and fault tolerance.

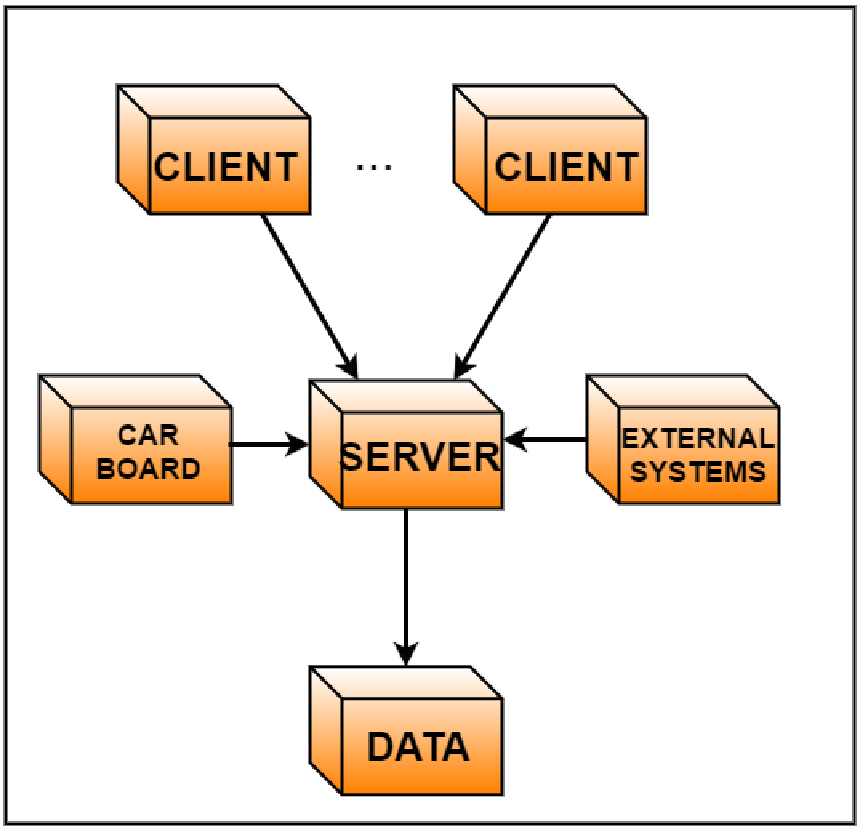


Figure 2.2: Tiers of the system.

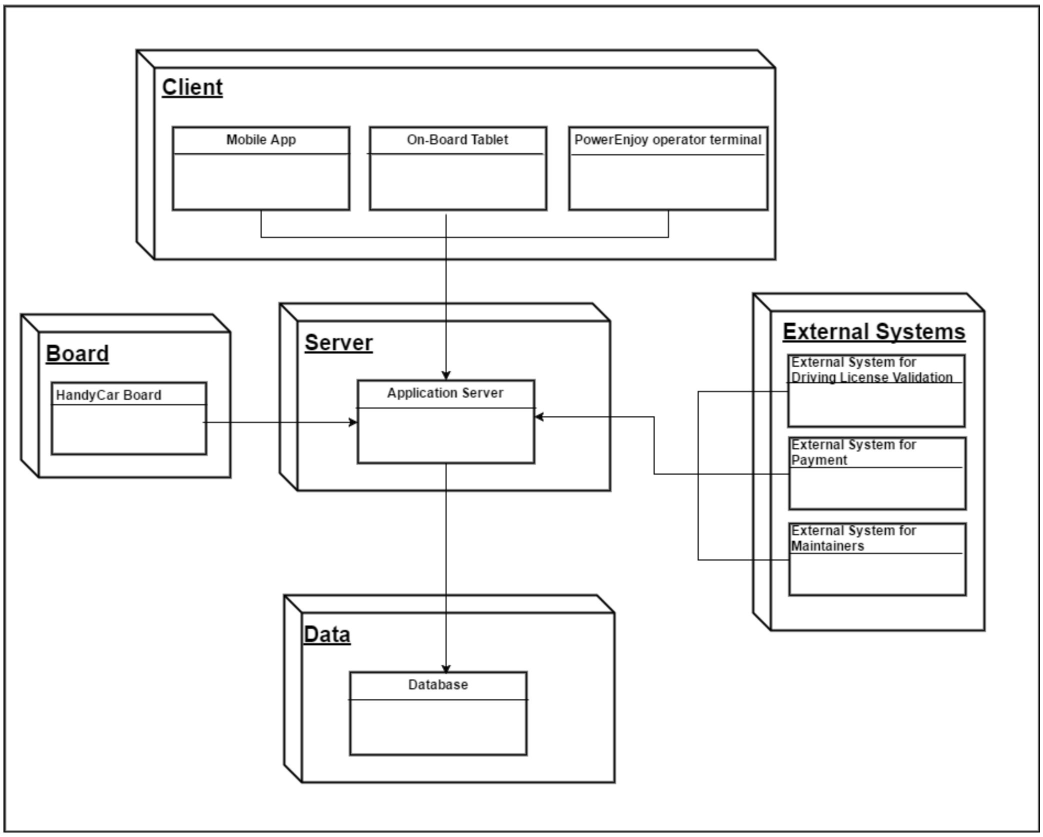

Client
Mobile App
On-Board Tablet
PowerEnjoy operator terminal
Board
HandyCar Board
Server
Application Server
External Systems
External System for Driving License Validation
External System for Payment
External System for Maintainers
Data
Database

Figure 2.3: High level components of the system.

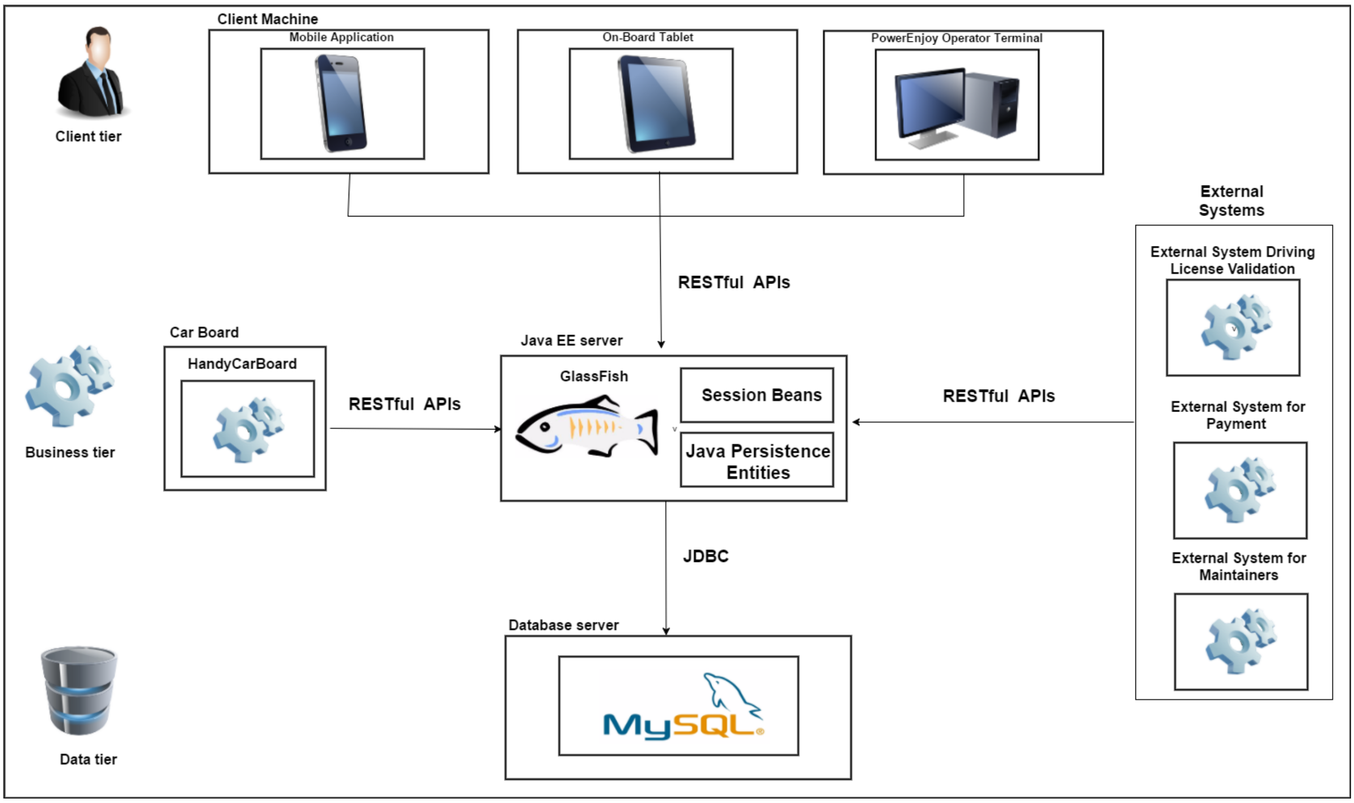


Figure 2.4: Description of the tiers, detailed with Java EE components.

The figure 2.2, instead, shows the three tiers from a very high level point of view: Client,Server and Data.

The interactions between the main components are shown in the figure 2.3.

A more detailed description of the three different tiers is shown in picture 2.4.

2.2 Component View

2.2.1 Database

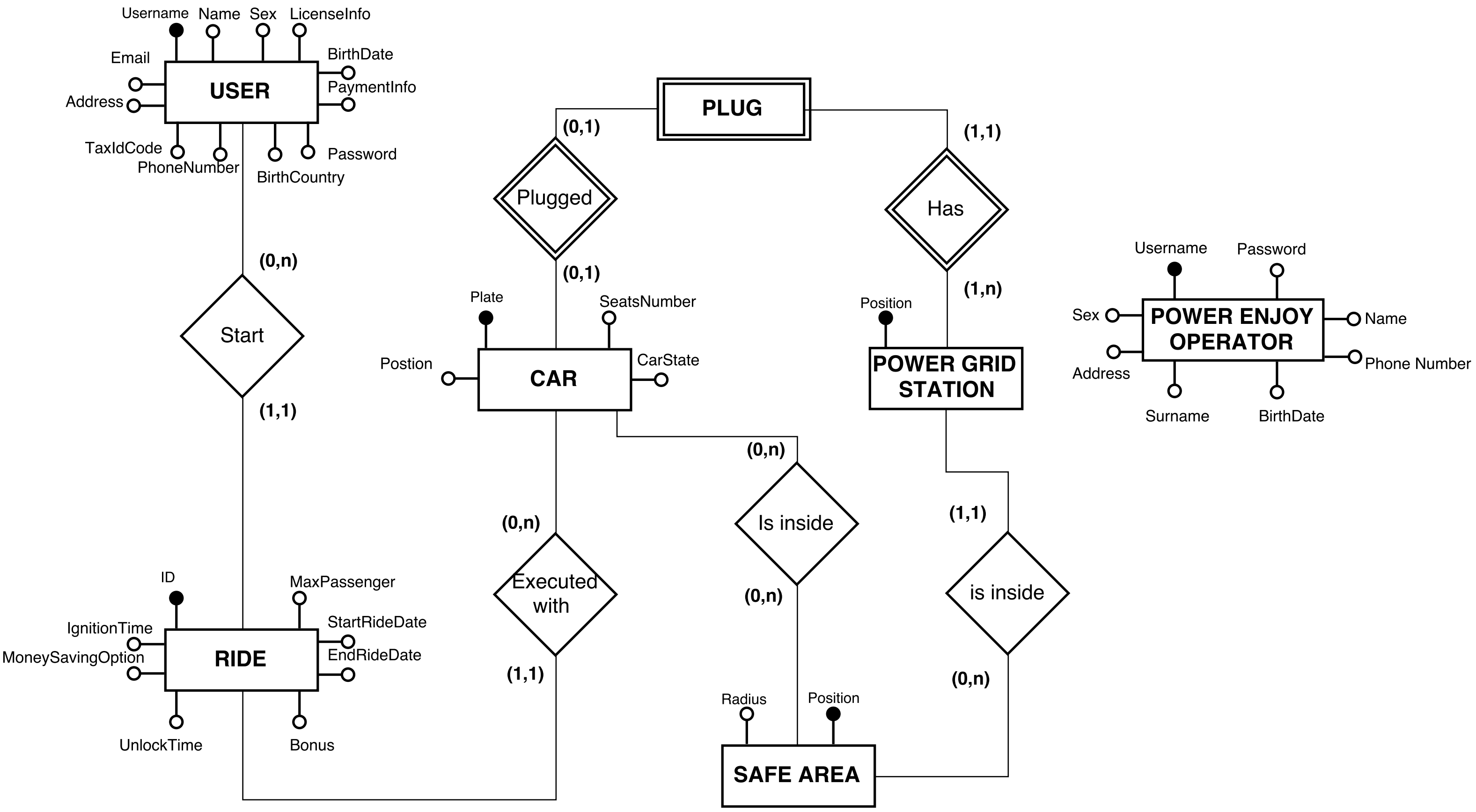
The database tier runs MySQL Community Edition and uses InnoDB as storage engine: the DBMS is fully transactional with rollback and commit, besides it ensure ACID properties and provides automatic recovery from crashes via the replay of logs.

The DBMS will not be internally designed because it is an external component used as a "black box" offering some services: it only needs to be configured and tuned in the implementation phase. The database can communicate only with the business logic tier using the standard network interface, described in section 2.6.

Security restrictions will be implemented to protect the data from unauthorized access: the database must be physically protected and the communication has to be encrypted. Access to the data must be granted only to authorized users possessing the right credentials and system privileges allow only administrators to perform administrative actions in the database, including privileges such as: create database, create procedure, create view, backup database, create table, and execute.

Every software component that needs to access the DBMS must do so with the minimum level of privilege needed to perform the operations. All the persistent application data is stored in the database. The conceptual design of the database is illustrated by the ER diagram.

Foreign key constraints and triggers are not used: the dynamic behaviour of the data is handled entirely by the Java Persistence API in the Business Application tier.


Username
Name
Sex
LicenseInfo
Email
BirthDate
USER
PaymentInfo
Address
TaxIdCode
PhoneNumber
BirthCountry
Password
(0,n)
Start
(1,1)
ID
MaxPassenger
IgnitionTime
StartRideDate
MoneySavingOption
RIDE
EndRideDate
UnlockTime
Bonus
PLUG
(0,1)
Plugged
(0,1)
Plate
SeatsNumber
Postion
CAR
CarState
(0,n)
Executed with
(1,1)
(0,n)
Is inside
(0,n)
Radius
Position
SAFE AREA
(1,1)
Has
(1,n)
Position
POWER GRID STATION
(1,1)
is inside
(0,n)
Username
Password
Sex
POWER ENJOY OPERATOR
Name
Address
Phone Number
Surname
BirthDate

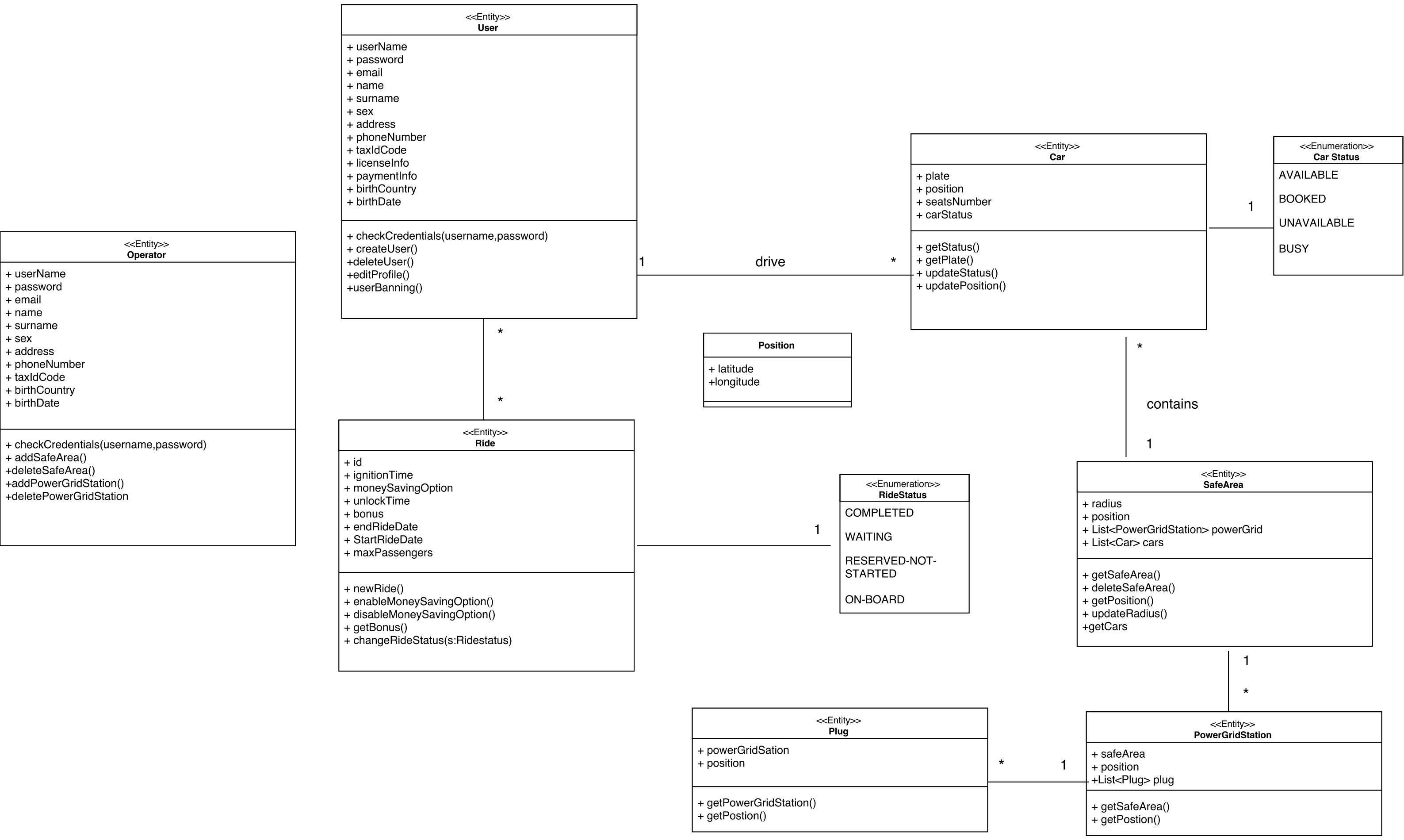
2.2.2 Application Server

The application server is implemented in the business logic tier, it runs on an open-source application server, GlassFish, that use Java EE and supports Enterprise JavaBeans. The access to the DBMS is not implemented with direct SQL queries but using Java Persistence API (JPA), in particular the Java Persistence Query Language (JPQL) makes queries against entities stored in DBMS. Queries resemble SQL queries in syntax, but operate against entity objects rather than directly with database tables. The object-relation mapping is done by entity beans.

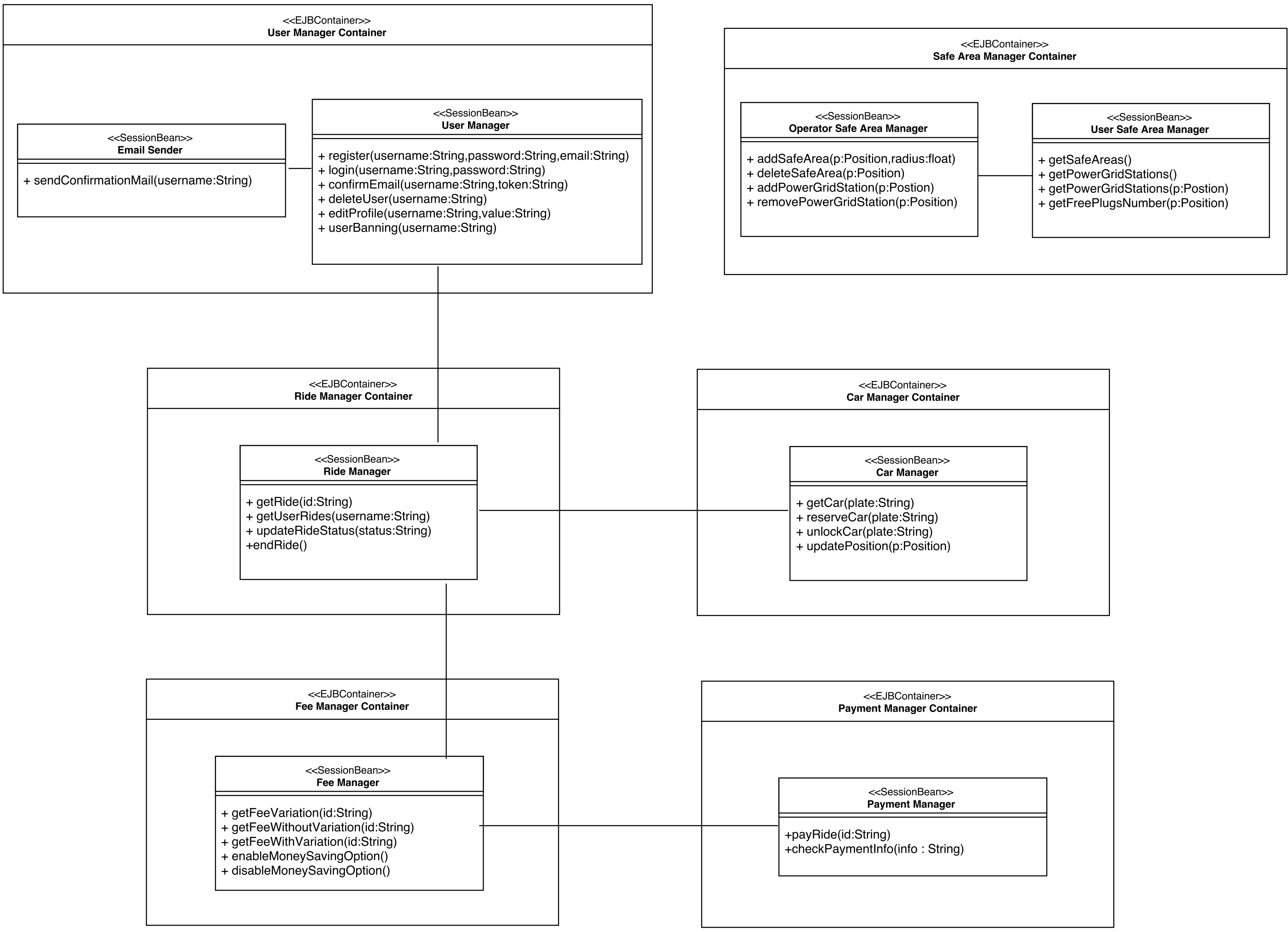
The Entity Beans representing the database entities are strictly related to the entities of the ER diagram,the data are stored automatically using container-managed persistence. They are persistent because their data is stored persistently in the database and they do survive a server failure, failover, or a network failure. The business logic is implemented by custom-built stateless Enterprise JavaBeans (EJB).

Our application is quite simple, the state of the cars is stored in the DB, so we do not need stateful EJBs which can be more expensive,but just stateless EJBs. Concurrency management and performance are fundamental, so the reuse of EJBs for many requests is a desirable behaviour. The application server implements a RESTful API using JAX-RS to allow the clients to use the services offered by the EJBs.

ENTITY BEANS



SESSION BEANS


<<EJBContainer>>
User Manager Container
<<SessionBean>>
Email Sender
+ sendConfirmationMail(username:String)
<<SessionBean>>
User Manager
+ register(username:String,password:String,email:String)
+ login(username:String,password:String)
+ confirmEmail(username:String,token:String)
+ deleteUser(username:String)
+ editProfile(username:String,value:String)
+ userBanning(username:String)
<<EJBContainer>>
Safe Area Manager Container
<<SessionBean>>
Operator Safe Area Manager
+ addSafeArea(p:Position,radius:float)
+ deleteSafeArea(p:Position)
+ addPowerGridStation(p:Postion)
+ removePowerGridStation(p:Postion)
<<SessionBean>>
User Safe Area Manager
+ getSafeAreas()
+ getPowerGridStations()
+ getPowerGridStations(p:Postion)
+ getFreePlugsNumber(p:Position)
<<EJBContainer>>
Ride Manager Container
<<SessionBean>>
Ride Manager
+ getRide(id:String)
+ getUserRides(username:String)
+ updateRideStatus(status:String)
+endRide()
<<EJBContainer>>
Car Manager Container
<<SessionBean>>
Car Manager
+ getCar(plate:String)
+ reserveCar(plate:String)
+ unlockCar(plate:String)
+ updatePosition(p:Position)
<<EJBContainer>>
Fee Manager Container
<<SessionBean>>
Fee Manager
+ getFeeVariation(id:String)
+ getFeeWithoutVariation(id:String)
+ getFeeWithVariation(id:String)
+ enableMoneySavingOption()
+ disableMoneySavingOption()
<<EJBContainer>>
Payment Manager Container
<<SessionBean>>
Payment Manager
+payRide(id:String)
+checkPaymentInfo(info : String)

Text

2.2.3 User Manager

This bean manages all the user management features: user registration, user login user deletion, profile editing and user banning. It also provides a function to confirm the email address provided by the user with the token sent by email.

This bean also communicates with two external systems:

- External Payment System, that checks that the payment information provided by the user are correct and handle the payment process;
- External System for Driving License Validation, that checks the driving license of the user during the registration process.

2.2.4 Car Manager

This bean manages manages all the car's features: get car, reserve car, unlock car and update position. In this way it keeps references and update all the informations about the car, including the maximum number of passengers the battery level and the numbers of seats. It also provides a function to handles the changes in the status of the car.

This bean also communicates with the External System for Maintainers, because the bean signals the unavailable cars to it.

2.2.5 Ride Manager

This bean manages all the ride features:create new ride, get ride,get user's ride and end ride. It provides functions that return all the rides informations like the number of passenger on the car,the unlock time and the user that has reserved the car.

2.2.6 Fee Manager

This bean manages all the functionalities about the fee of the ride: calculate fee variation, calculate fee with variation and calculate the fee without variation. It handles all the information about the bill that the user has to pay and apply discounts when it is necessary.

2.2.7 Safe Area Manager

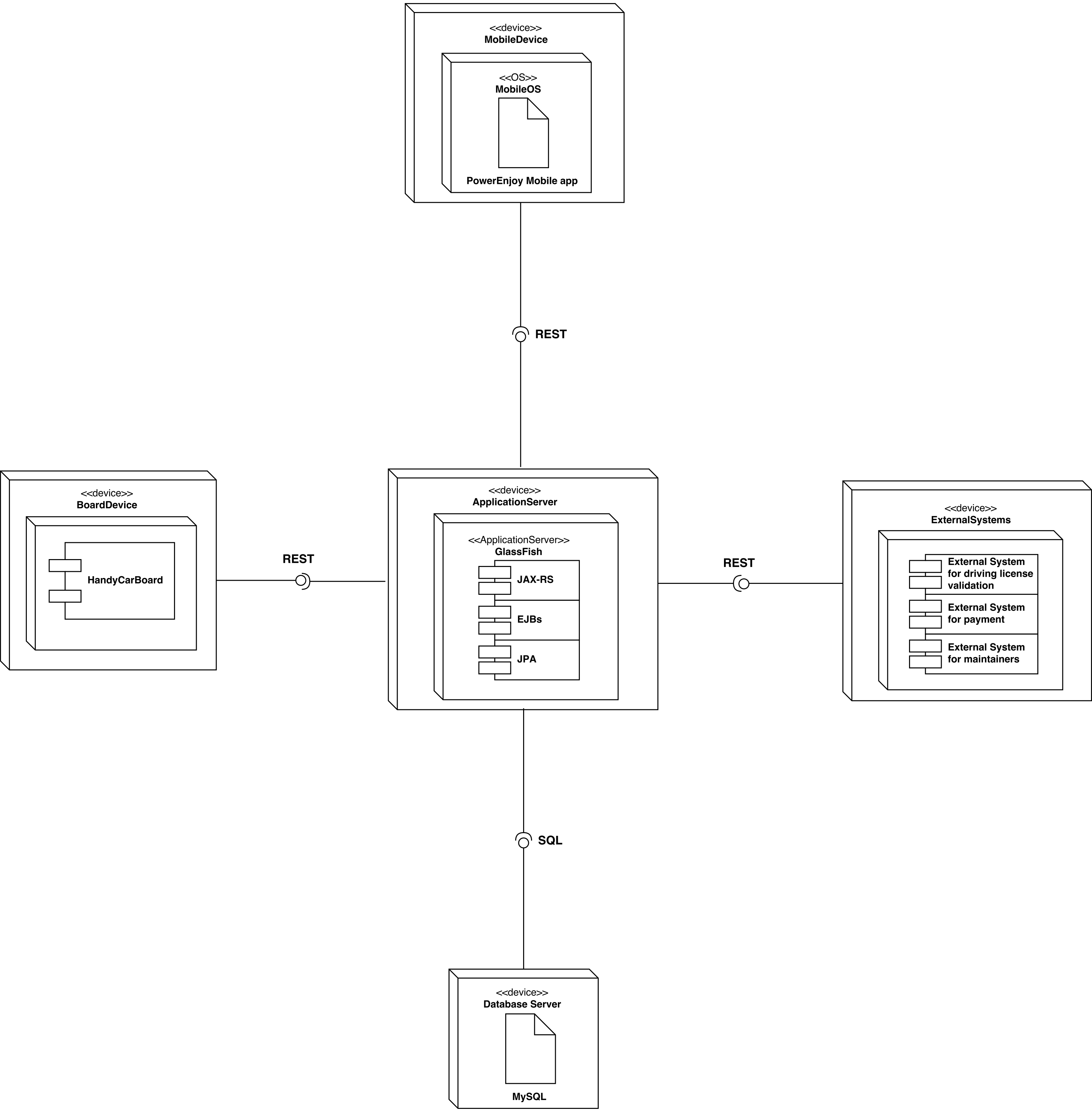
This bean allows the operator to manage the safe area and provides functions to add a new safe area and to delete it. It also provides function to add and delete a power grid station in a specific position. Besides it offers also some function available also to the user that allows him/her to know the safe areas and power grid stations positions.

2.2.8 Payment Manager

This bean provide a function to pay the ride and another one to check the payment information provided by the user thanks to an external system of payment information validation.

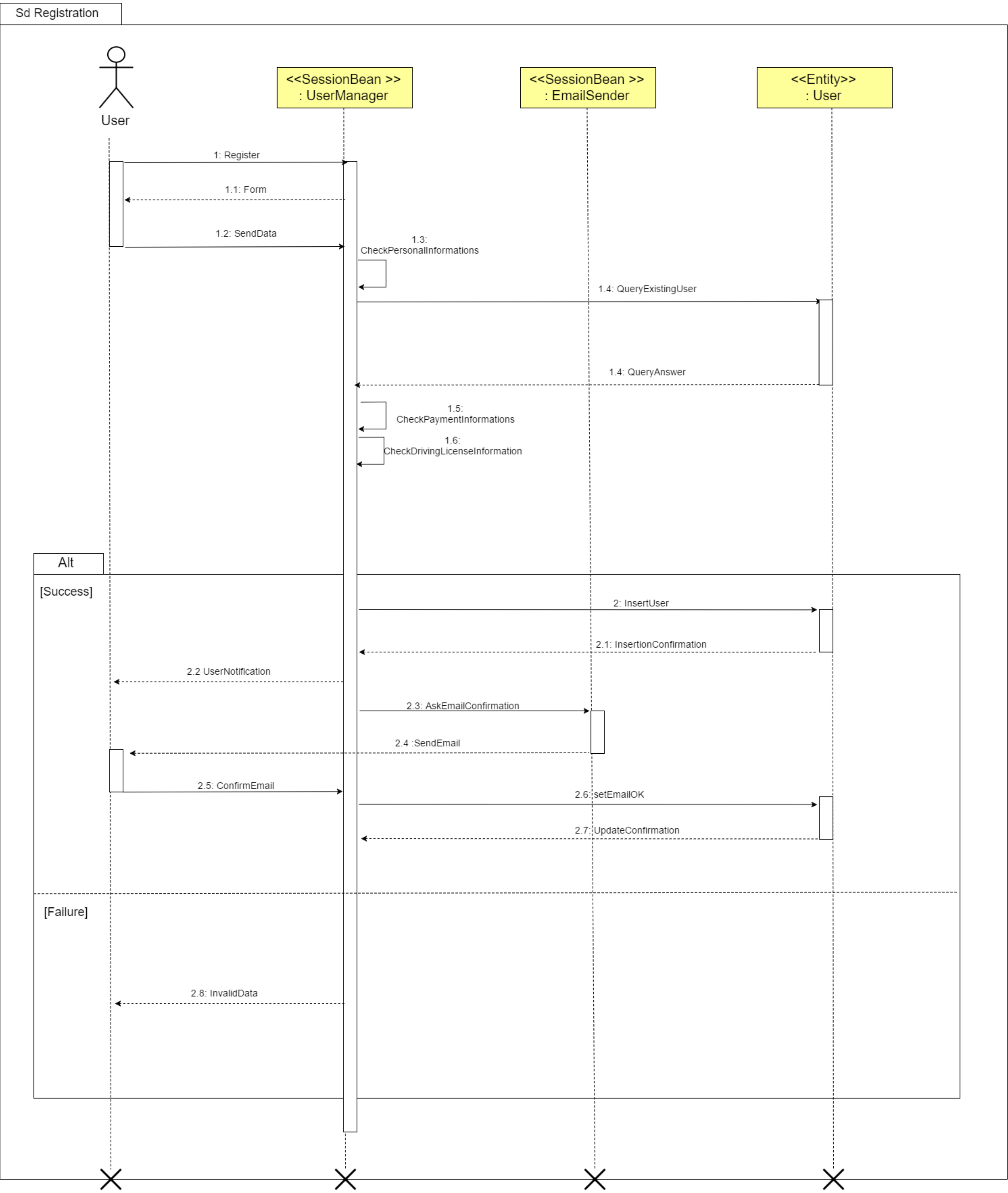
2.3 Deployment View

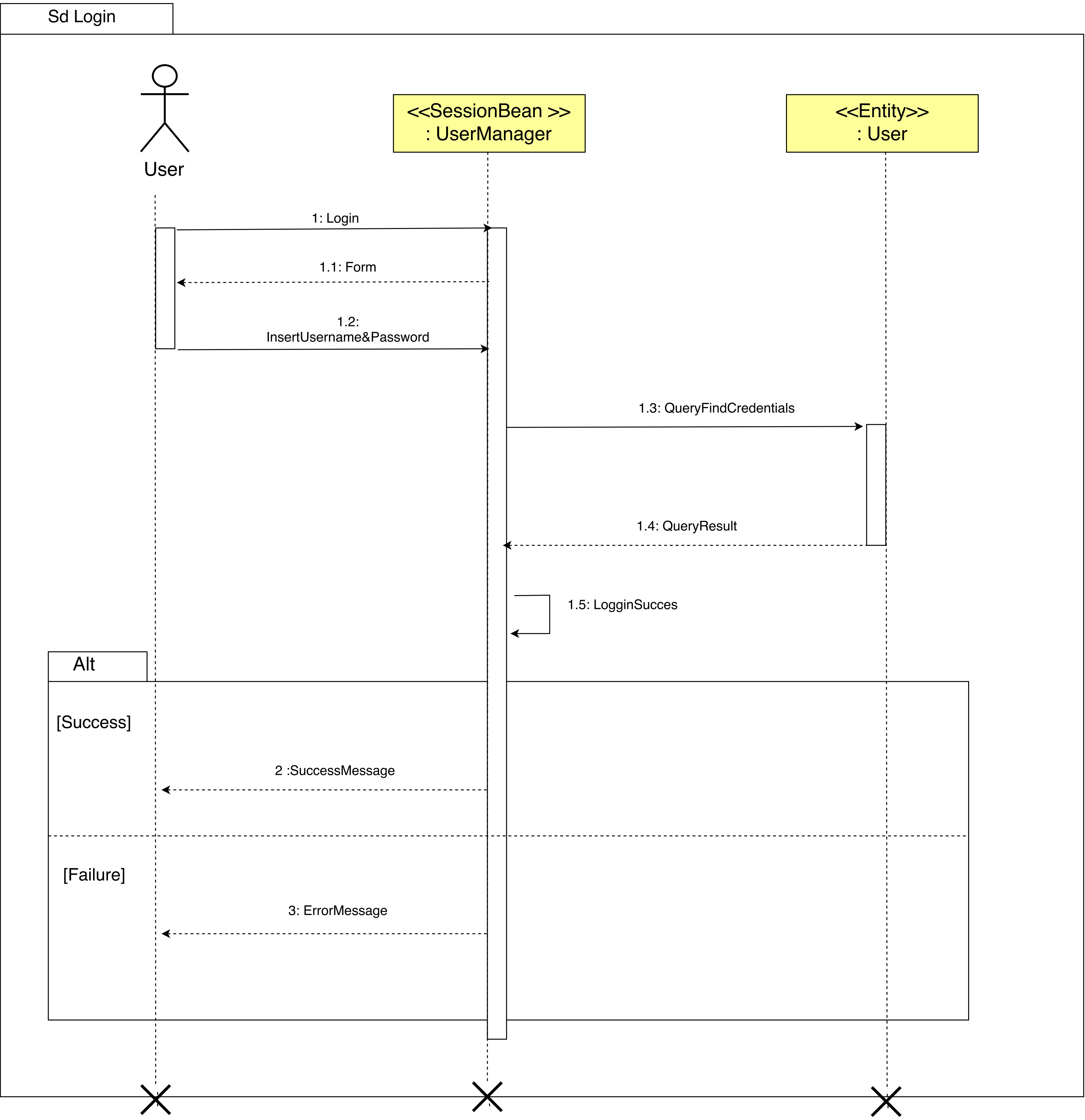
The the business logic layer containing the application server is allocated on one physical machine (tier), while the data are on another dedicated machine.The external system provides services for payments and driving license validation;while the HandyCardBoard use the information coming from the sensors and actuators of the car. Both communicate with the application server. GlassFish Server is used as the Java EE application server for the business logic.



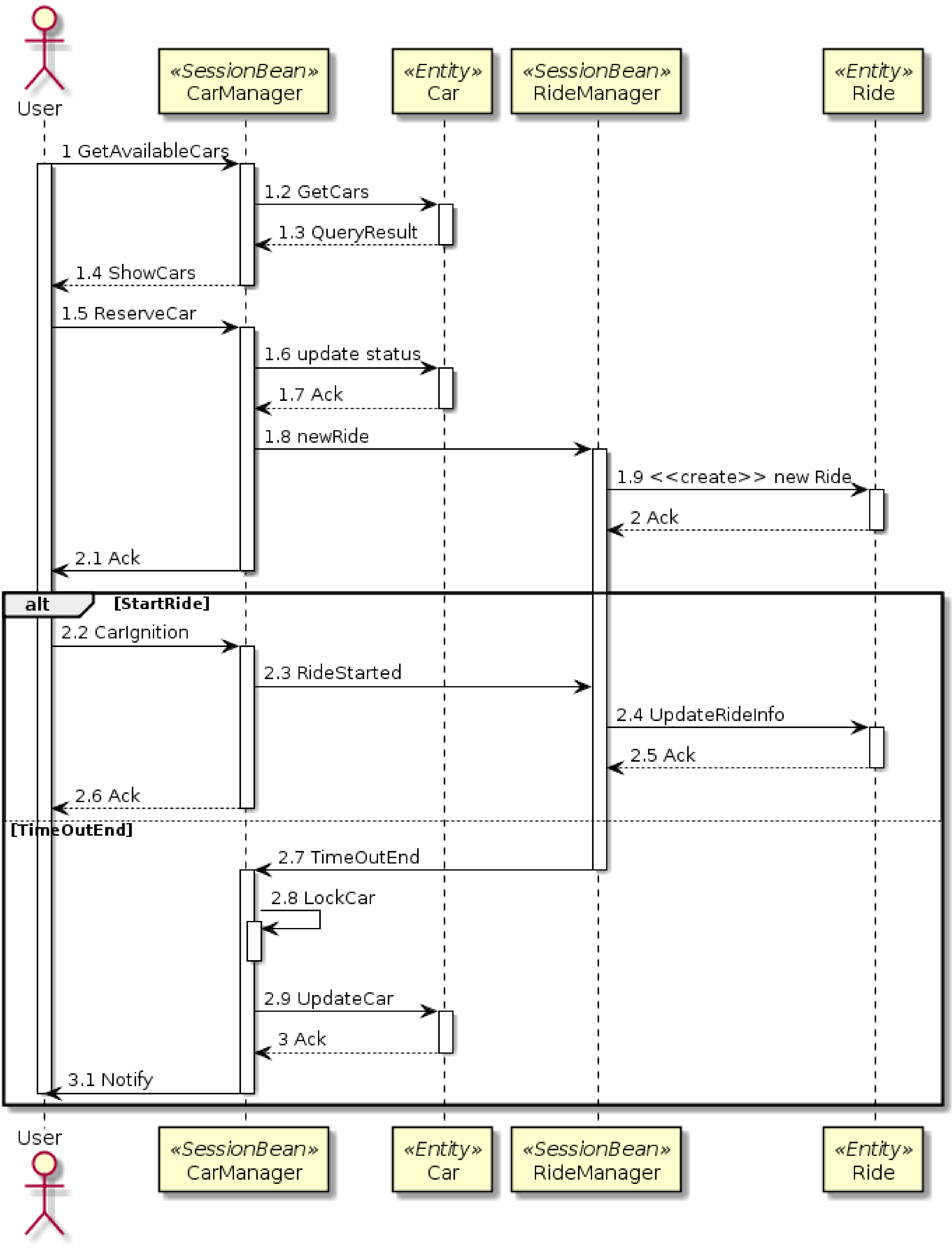
<<device>>
MobileDevice
<<OS>>
MobileOS
PowerEnjoy Mobile app
REST
<<device>>
BoardDevice
HandyCarBoard
REST
<<device>>
ApplicationServer
<<ApplicationServer>>
GlassFish
JAX-RS
EJBs
JPA
REST
<<device>>
ExternalSystems
External System for driving license validation
External System for payment
External System for maintainers
SQL
<<device>>
Database Server
MySQL

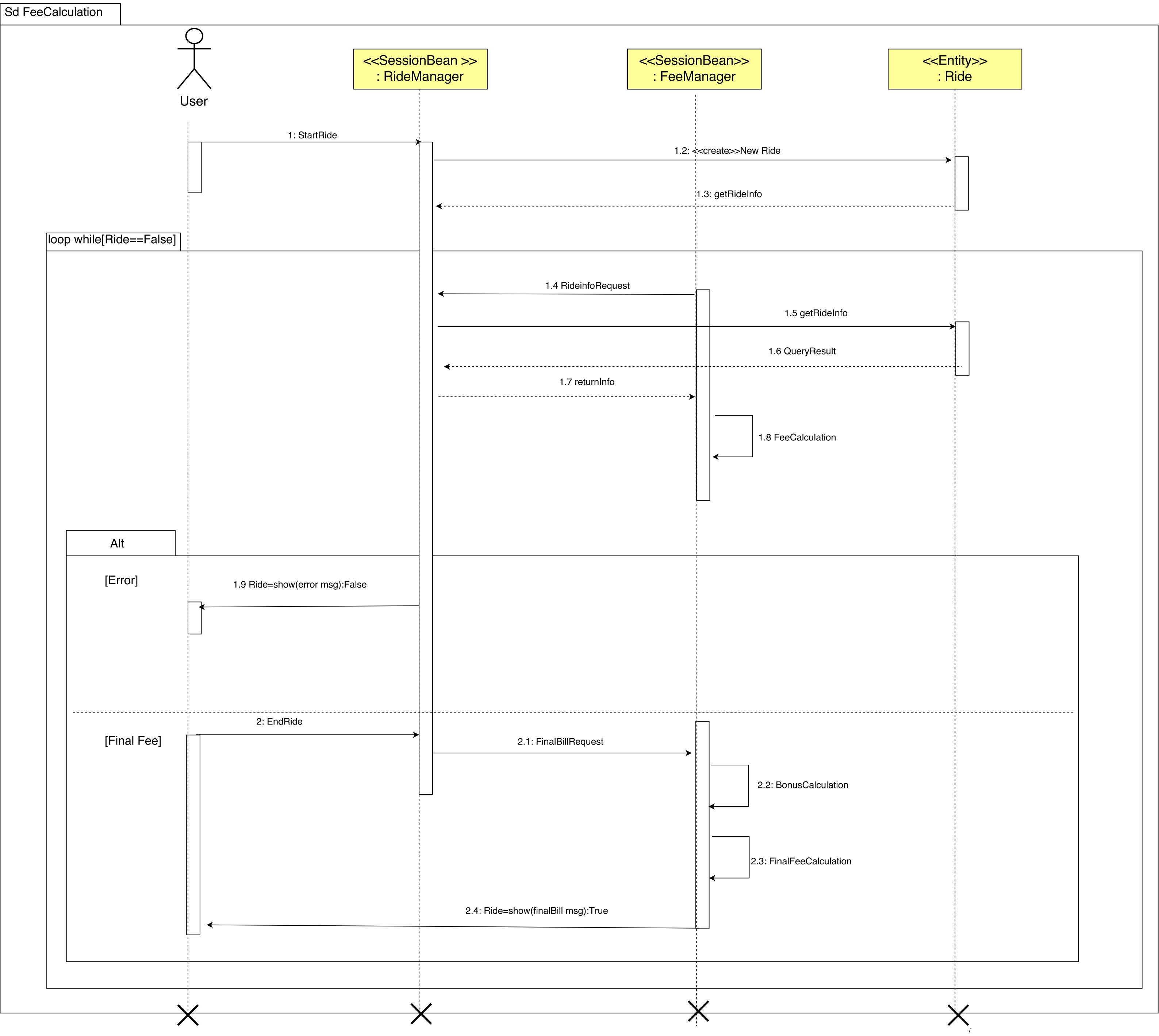
2.4 Runtime View


Sd Registration
User
<<SessionBean >>
: UserManager
<<SessionBean >>
: EmailSender
<<Entity>>
: User
1: Register
1.1: Form
1.2: SendData
1.3:
CheckPersonalInformations
1.4: QueryExistingUser
1.4: QueryAnswer
1.5:
CheckPaymentInformations
1.6:
CheckDrivingLicenseInformation
Alt
[Success]
2: InsertUser
2.1: InsertionConfirmation
2.2 UserNotification
2.3: AskEmailConfirmation
2.4 :SendEmail
2.5: ConfirmEmail
2.6: setEmailOK
2.7: UpdateConfirmation
[Failure]
2.8: InvalidData


Sd Login
User
<<SessionBean >>
: UserManager
<<Entity>>
: User
1: Login
1.1: Form
1.2:
InsertUsername&Password
1.3: QueryFindCredentials
1.4: QueryResult
1.5: LogginSucces
Alt
[Success]
2 :SuccessMessage
[Failure]
3: ErrorMessage

"Sd Ride"


User
«SessionBean»
CarManager
«Entity»
Car
«SessionBean»
RideManager
«Entity»
Ride
1 GetAvailableCars
1.2 GetCars
1.3 QueryResult
1.4 ShowCars
1.5 ReserveCar
1.6 update status
1.7 Ack
1.8 newRide
1.9 <<create>> new Ride
2 Ack
2.1 Ack
alt
[StartRide]
2.2 CarIgnition
2.3 RideStarted
2.4 UpdateRideInfo
2.5 Ack
2.6 Ack
[TimeOutEnd]
2.7 TimeOutEnd
2.8 LockCar
2.9 UpdateCar
3 Ack
3.1 Notify
User
«SessionBean»
CarManager
«Entity»
Car
«SessionBean»
RideManager
«Entity»
Ride


Sd FeeCalculation
User
<<SessionBean >>
: RideManager
<<SessionBean>>
: FeeManager
<<Entity>>
: Ride
1: StartRide
1.2: <<create>>New Ride
1.3: getRideInfo
loop while[Ride==False]
1.4 RideinfoRequest
1.5 getRideInfo
1.6 QueryResult
1.7 returnInfo
1.8 FeeCalculation
Alt
[Error]
1.9 Ride=show(error msg):False
[Final Fee]
2: EndRide
2.1: FinalBillRequest
2.2: BonusCalculation
2.3: FinalFeeCalculation
2.4: Ride=show(finalBill msg):True

2.5 Component Interfaces

2.5.1 Logic layer to data storage layer

The logic layer communicates with the data storage layer via the Java Persistence API (JPA) over standard network protocols. In this way, the two layers can be deployed both in different tiers or in the same one.

The JPA specification uses an object/relational mapping approach to bridge the gap between an object-oriented model and a relational database in order to focus more on the object model rather than on the actual SQL queries used to access data stores.

2.5.2 Logic layer to presentation layer

The application server communicates with the other elements of the system through a RESTful interface over the HTTPS protocol. The RESTful interface is implemented using JAX-RS and uses JSON as the language data format.

User Manager

All the following functions can be called by the mobile app of the user:

- **void register(String username, String password, String email):** Add a new user in the system with the provided data if these are correct. After this, an email with a token is sent to the user email address in order to confirm the latter.

- **Token login(String username, String password):** Allows any registered user to log into the system using his own username and password. If these credentials are correct, the function returns a token to be used in the future requests to identify the user, otherwise it returns an error.

- **void confirmEmail(String username, Token emailToken):** Validates the email address that was inserted by a registered user using the token sent to that email address after the registration.

- **void deleteUser(String username):** Allows users to delete the user account and information, except for his essential information and rides because they can be requested from authorities. As parameters it takes the username of the user and his password.

- **void editProfile(String fieldName, String newValue):** Allows users to edit their information. If they user wants to change its email, he/she has to confirm it as for the registration.

- **void userBanning(String username):** Blocks the user's account.This function is available only for PowerEnJoy operators.

Ride Manager

- **Ride getRide(String id):** Returns the ride info that corresponds to a given ID. The function will return an error if a user wants to retrieve the info about a ride that is not assigned to him/her. The PowerEnJoy operators can get all the rides, without restrictions. The info about a ride contain: the plate of the car, the username of the user that reserved the car, unlock time, ignition time, end time, the fee with variations and the fee variations and the maximum number of passengers.

- **List <Ride> getUserRides(String username):** Returns the rides info that corresponds to a given users. The function will return an error if a user wants to retrieve the info about another user. The PowerEnJoy operators can get all the rides, without restrictions.

Car Manager

- **Car getCar(String plate):** Returns the car info that corresponds to a given plate. The info about a car contain: the battery level, the state, if the engine is on or off, the number of seats, the model and the manufacturer.

- **List <Car> getAvailableCars(Position center, float radius):** Returns the cars that are available in a given circle.

- **void reserveCar(String plate):** The mobile app calls this function to allow the user to request a car. This function returns an error if the request is not valid.

- **void unlockCar(String plate):** The mobile app calls this function to allow the user to unlock a car that he reserved. This function returns an error if the request is not valid.

- **void endRide(int maxNumberOfPassengers):** The HandyCar Board use this function to report to the system that the user ended a ride.

At this point, the application server saves the end time, the fee variation that were applied and the maximum number of passengers. Every HandyCar Board has its own ID, so the system can recognize from which car the request is sent.

- **void reportFailedCar():** The HandyCar Board use this function to report to the system that the a car is failed. At this point, the Car Manager in the application server turn the state of that car to unavailable and signal it to the maintainers server.

Safe Area Manager

These functions can be called only from the user of the ride or from PowerEnJoy operators:

- **void addSafeArea(Position center, float radius):** Adds a safe area with a given center and radius. The function returns an error if the new safe area is contained into another pre-existing safe area.
- **void deleteSafeArea(Position center):** Removes the safe area with the given center. The function returns an error if there is not a safe area with that center.
- **void addPowerGridStation(Position position):** Adds a power grid station in a given position. The function returns an error if there is not a safe area in that position.
- **void removePowerGridStation(Position position):** Removes a power grid station in a given position. The function returns an error if there is not a power grid station in that position.

These other functions are available to all the users:

- **List <SafeArea> getSafeAreas():** Returns all the safe areas.
- **List <PowerGridStation> getPowerGridStations():** Returns all the power grid stations.
- **List <PowerGridStation> getPowerGridStations(Position safeAreaCenter):** Returns all the power grid stations that belongs to the safe area with the given center.
- **int getFreePlugsNumber(Position powerGridStationCenter):** Returns the number of free plugs in the power grid station with the given center.

Fee Manager

All these functions can be called only from the user of the ride or from PowerEnJoy operators.

- **float getFeeVariation(String ID):** Calculates a fee variation for a given ride ID.
- **float getFeeWithoutVariation(String ID):** Calculates a fee for a given ride, without including the variations of bonuses and penalty.
- **float getFeeWithVariation(String ID):** Calculates a fee for a given ride, including the variations of bonuses and penalty.

These functions can be called only from the car tablet:

- **PowerGridStation enableMoneySavingOption():** This function activates the Money Saving Option and returns the power grid station where the user has to plug the car.
- **void disableMoneySavingOption():** This function disable the Money Saving Option in order to be able to leave the car in every safe area.

Payments Manager

- **void payRide(String ID):** Pay the fee for the ride of the given ID. The function returns an error if the ride was already paid or if the user is not the owner of the ride
- **Boolean checkPaymentInfo(String info):** Check if the payment informations provided by the user are correct or no. The function return true if they are correct, false if there's an error in the informations.

2.5.3 HandyCar Board

The HandyCar Board provides some REST APIs, that allow the application server to interact easily with the sensors and the actuators of the electric cars.

- **void lock():** Lock the car.
- **void unlock():** Unlock the car.
- **CarValues getCarValues():** Returns the values of all the sensors and actuators of the car.

2.6 Selected architectural styles and patterns

We used the following architectural styles and patterns:

Client-Server: The client–server model is used between these communications:

- the application server (client) queries the DB (server);
- the presentation layer (client) communicates with the application server (server);
- the application server (client) and the on-board tablet (server);
- the HandyCar Board and the application server (both act like client and server);
- the application server (client) and the payment validation external system (server);
- the application server (client) and the driving license validation external system (server).

We choose this type of architecture to have a centralized control and because it is simple and well known by our development team.

Service-oriented architecture: The SOA is used by the system for the communication between the application server and the presentation layer.

We used SOA in order to have an high level interaction between these two layers, by looking only at the component interfaces, that represent a black box for its consumers.

In this way, a service logically represents a business activity with a specified outcome, so you want to use it, you do not have to look inside its specific implementation.

Thin client: The thin client is implemented in the mobile app and in the on-board tablet. We used this paradigm in order to have light-weight apps, that can run in a large number of devices with a smooth user interaction.

Instead, all the application logic is on the application server, on which the client depends heavily on. This can help to reduce the number of updates, because if you want to change something in the application logic, it is not necessary to release a new application of the client.

2.7 Other design decisions

2.7.1 Storage of passwords

Users' passwords are not be stored in plain text, but they are hashed and salted. In this way it is more difficult for an attacker to retrieve the user information after a possible case of data theft.

In addition to this, the hash algorithm used must be specifically designed to hash passwords, in order to slow down a possible brute force attack.

2.7.2 Maps

In order to have a knowledge of the maps, the system uses the external service Google Maps. We choose it because it is really reliable and its APIs are well documented and maintained.

Of course it was more convenient to adopt an external service like this instead of reconstruct the maps where the PowerEnJoy service will operate.

Chapter 3

Algorithm Design

In this section we will describe some algorithms used by the system in different phases during the car rent management. We cannot describe all the algorithms used by the system, due to its complexity. However we have chosen the most important ones, in order to show some development choices we made. The chosen algorithms are the following four:

1. Given a ride, calculate the final fee
2. Given a circle with center c and radius r, find all the Safe Areas that intersect the given circle
3. Given a position, find all the car far at most 2km
4. Given a destination, find the Safe Area where to park

In the algorithm description there is not the final code. Instead we will use a mix of common speech and pseudo-code (easy to translate into the desired programming language). This pseudo-code is referred to an object-oriented programming language.

3.1 Algorithm 1: How to calculate the final fee

When a ride ends it is important to calculate the final fee to charge to the user. This may differ to the amount shown on the car screen due to some bonus or malus (called fee variator) unlocked by the user. So at the end of each ride the system will call this function. It will have a ride in input and a float as output, that represents the final fee to charge.

Just to remember, a ride has the following fields:

- Reservation Time
- Unlock Time
- Ignition Time
- End Time
- User
- Car
- max Passengers Number
- a set in which are stored the fee variator already unlocked

We also remember that each fee variator has a float field called variator that represents the percentage variation made by that fee variator. These float is positive for the malus (because it has to increase the fee) and is negative for the bonus (because it has to reduce the fee).

In order to describe the algorithm in the easiest way we assume:

- the function can accede to the cost per minute of the rent (CPM) and a set of all the fee variator that can be unlocked (called feeVariatorSet)
- there is a function called rideLength(Ride ride) that returns the length of a ride as a number of minutes
- each fee variator has a function check(Ride ride). Given a ride this function returns true if the relative fee variator can be unlocked, false otherwise.

```
Input  : A ride
Output: The final fee to charge
foreach var in feeVariatorSet do
    if var is check(ride) then add var in ride.feeVariator;
end
result = rideLength(ride) * CPM;
foreach var in ride.feeVariator do
    result = result + result * var.variator;
end
return result;
```

Algorithm 1: How to calculate the final fee of a ride

3.2 Algorithm 2: How to find the Safe Areas near a given position

This algorithm will never be used directly but it will be included into other algorithms, such as algorithms 3 and 4 described below. This algorithm will produce as output a set of safe areas that intersect a circle with a given center and radius. We remember that:

- Two circles intersect iff the distance between the two centers is lesser than the sum of the two radius.
- Given two positions P_1 and P_2, with lat_1 and lat_2 their latitude and $long_1$ and $long_2$ their longitude, the distance d between them is given by:

$$d = R \times (\arccos(\sin(lat_1) \times \sin(lat_2) + \cos(lat_1) \times \cos(lat_2) \times \cos(long_1 - long_2)) \tag{3.1}$$

We assume also that:

- There is a function called distance(Position p_1, Position p_2) that given two positions return the distance between them, according to the equation 3.1.
- The algorithm can accede to a set, called $safeAreaSet$, in which all the Safe Area existing in the system are stored.

So the algorithm is:

```
Input   : A position and a max distance
Output: The set of chosen Safe Areas
foreach safeArea in safeAreaSet do
    if distance(center of safeArea, position) < sum of maxDistance
      and safeArea radius then add safeArea in result;
end
return result;
```

Algorithm 2: How to find Safe Areas that intersect a circle with given center and radius

3.3 Algorithm 3: How to find cars near a given position

When a user wants to start a ride he or she will search for a car near his or her position in order to rent it. This algorithm takes care of the car searching.

We decided that a user can reach a car by foot if its location is 2km or less from his or her one. So when a user wants to start a ride the system has to provide to him or her all the available cars that are far at most 2km.

This algorithm is divided into two parts. In the first part it uses the algorithm 2 to find all the safe areas that intersect a circle with the center into the user's position and a radius equal to 2km. In the second part the algorithm analyzes all the cars parked in the safe areas found in the first part and returns the only ones parked 2km or less from the user's position.

But why is it important to use algorithm 2? We decided to implement algorithm 2 in order to reduce the time to search a car. The number of safe area is much less than the number of car. Furthermore safe area position is fixed, instead car position changes. Using safe areas position instead of cars one lets the algorithm to cycle fewer objects, cutting all the cars parked in safe areas too much distant from the user, saving time and resources.

```
Input  : The user's position
Output: The set of chosen cars
call algorithm2(userPosition, 2km);
foreach safeArea in algorithm2Result do
    foreach Car parked in safeArea do
        if distance(userPosition, Car position) $< 2km$ then add Car
          in result;
    end
end
return result;
```

Algorithm 3: How to find cars far at most 2km from user's position

3.4 Algorithm 4: How to find a Safe Area where to park

These algorithm fulfill two purposes: one it is find a safe area near a location where the user can park. The second one is try to balance cars in the city. The second purpose is as important as the first one. In fact we want that cars are equally distributed among all the safe areas. We do not want that some safe areas have much more cars than others. As the algorithm 3, algorithm 4 is divided into two parts, with the first one utilizing algorithm 2. And as algorithm 3, the max distance allowed is 2km, the max distance reachable in a reasonable time by foot.

In order to understand the algorithm we assume that there is a function (called carDensity(SafeArea)) that return the density of parked cars in a given safe area (the ratio between number of cars parked in that safe area and the area of the safe area).

```
Input   : The user's destination
Output: A set of safe areas, ordered by car density
call algorithm2(destination, $2km$);
sort algorithm2Result ordered by carDensity(safeArea) from lowest to highest;
return result;
```

Algorithm 4: How to find a safe area were to park

If the algorithm was called in order to find a safe area, the system will show the first element of the result. If the algorithm was called to find a power grid station, the system will show the first element of the result with a power grid station with a free plug far at most 2km from user's destination.

Chapter 4

User Interface Design

4.1 User Experience

We have already dealt with user interface designs in the RASD, where we have shown some mock-ups of the screens of our application.

To maintain reasonable the initial development cost, the PowerEnJoy operators will access to the system through ssh, that can ensure a certain level of security. In the future, a web app for the PowerEnJoy operators will be created.

In this section we will model the user experience of the mobile app, showing how the interface changes with the input of the user and external factors, so we will map the sequence of actions and events with the screens flow.

We will use the class diagram for the user experience modelling. This approach needs some clarification about the used notation, so below we are going to explain the meaning of the symbols used:

Directed arrow: It defines a transition from a particular user interface element to another. It may be due to two possible cause:

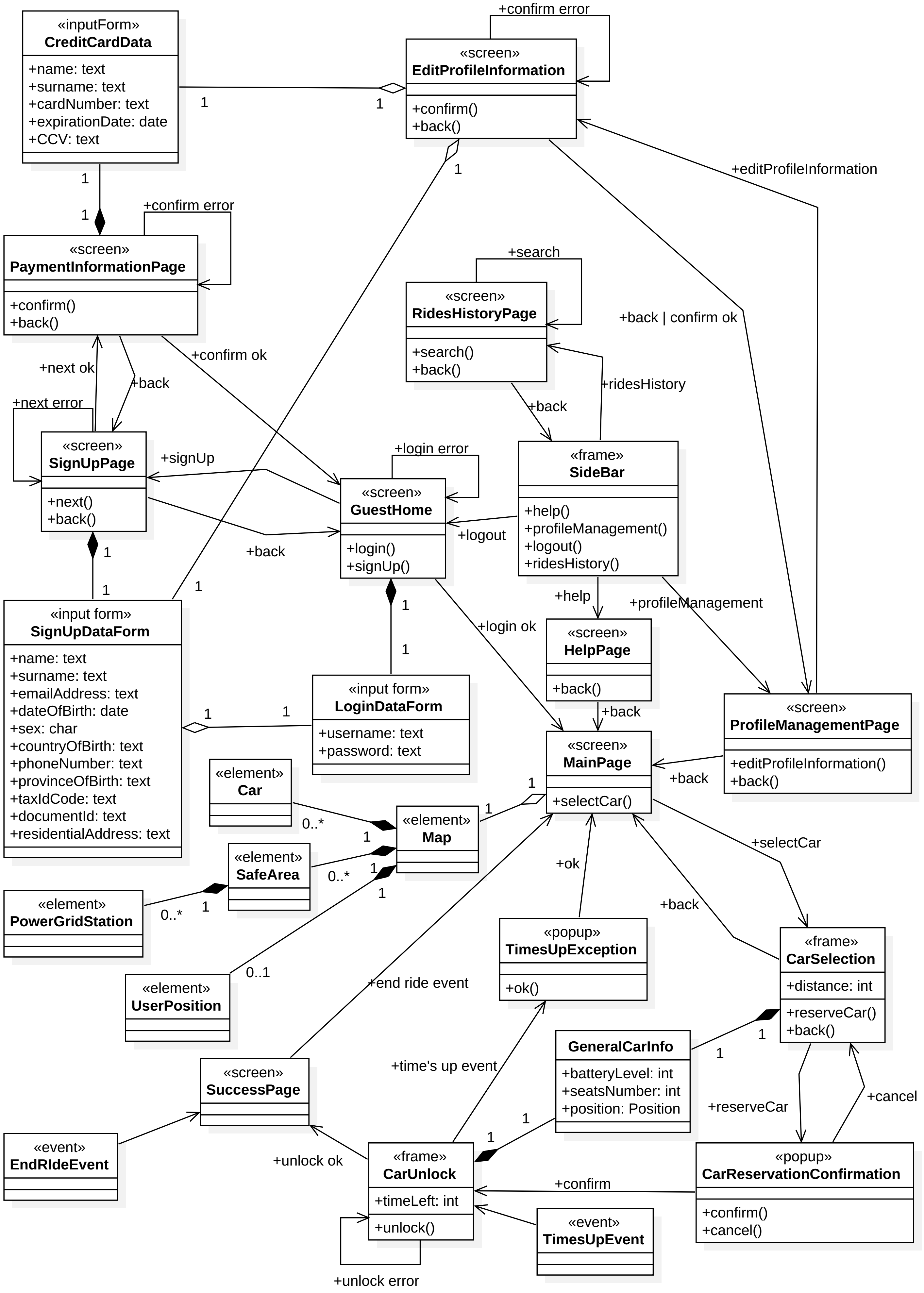
- **User input:** This type of transition is triggered by the invoking of a particular method *function()* in the source interface element. The name of the user input transaction is the same of the function that was invoked by the user. Sometimes the method can rise an error, so in these cases the name of the transaction is followed by "ok" if the outcome of the function is positive or by "error" if it is negative.
- **Event:** This type of transition is triggered by an event that can be generated by the client itself or by a message sent by the server

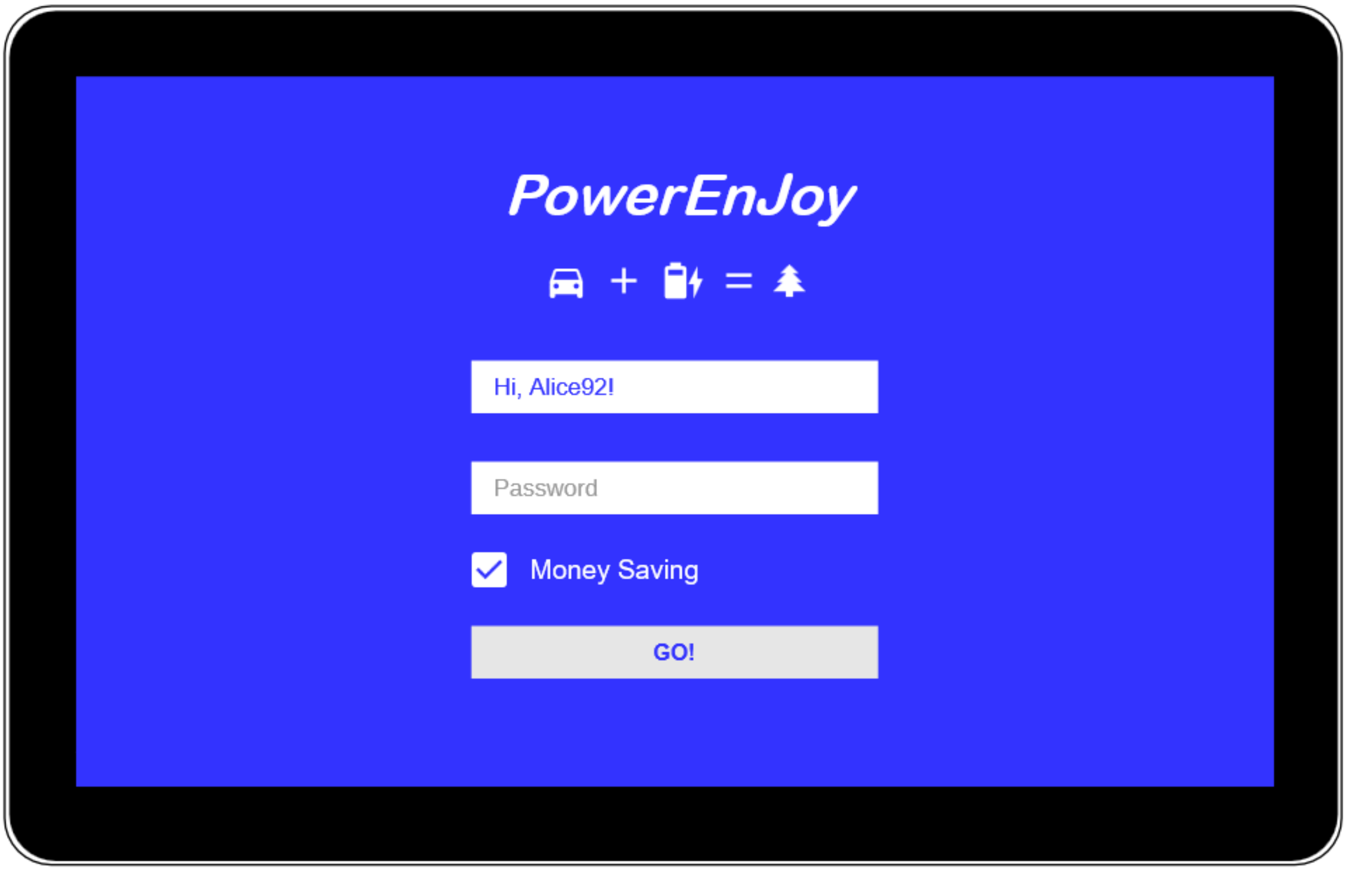
to the client. This transaction is identified by the ”event” at the end of the name.

Class symbol: It determines a specific user interface element among *Screen*, *Input form*, *Element*, *Pop-up* and *Frame* or a specific event (stereotype *event*). In both cases the type of element is specified in the stereotype of the class symbol.

Composition symbol: Means that a specific user interface element is contained in another. Typically is used when a form or a list of elements is contained in a particular screen. It can be also used to say that a generical user interface element displays some info.

It is important to underline that, in order to maintain readability, only the main elements are shown in the following diagram.


«inputForm»
CreditCardData
+name: text
+surname: text
+cardNumber: text
+expirationDate: date
+CCV: text
+confirm error
«screen»
EditProfileInformation
+confirm()
+back()
1
1
1
1
1
+confirm error
«screen»
PaymentInformationPage
+confirm()
+back()
+search
«screen»
RidesHistoryPage
+search()
+back()
+back | confirm ok
+editProfileInformation
+confirm ok
+next ok
+back
+next error
+ridesHistory
+back
«screen»
SignUpPage
+next()
+back()
+signUp
+login error
«screen»
GuestHome
+login()
+signUp()
«frame»
SideBar
+help()
+profileManagement()
+logout()
+ridesHistory()
+logout
+back
1
1
1
«input form»
SignUpDataForm
+name: text
+surname: text
+emailAddress: text
+dateOfBirth: date
+sex: char
+countryOfBirth: text
+phoneNumber: text
+provinceOfBirth: text
+taxIdCode: text
+documentId: text
+residentialAddress: text
1
1
+help
+profileManagement
+login ok
«screen»
HelpPage
+back()
«input form»
LoginDataForm
+username: text
+password: text
1
1
+back
«screen»
ProfileManagementPage
+editProfileInformation()
+back()
«screen»
MainPage
+selectCar()
+back
«element»
Car
0..*
1
1
«element»
Map
1
+selectCar
«element»
SafeArea
0..*
1
1
+ok
+back
«element»
PowerGridStation
0..*
1
«popup»
TimesUpException
+ok()
«frame»
CarSelection
+distance: int
+reserveCar()
+back()
«element»
UserPosition
0..1
+end ride event
GeneralCarInfo
+batteryLevel: int
+seatsNumber: int
+position: Position
1
1
«screen»
SuccessPage
+time's up event
+reserveCar
+cancel
1
1
«event»
EndRIdeEvent
+unlock ok
«frame»
CarUnlock
+timeLeft: int
+unlock()
+confirm
«popup»
CarReservationConfirmation
+confirm()
+cancel()
«event»
TimesUpEvent
+unlock error



PowerEnJoy

Hi, Alice92!

Password

☑ Money Saving

Figure 4.1: Car login page.

4.2 On-Board Tablet

We have already shown the mobile app user interface in the RASD, so in this section we will show some mock-ups of the tablet that is present in every car.

When the user enters the car, the screen in figure 4.1 is what he/she will see in the tablet. The tablet receives the nickname of the user that unlocked the car, so it is able to display it. If the password is correct than the user can turn on the car.

If the authentication phase is successful, then on the tablet appears the Main Page. The mock-up of this screen is presented in figure 4.2. Here you can navigate through the map, where you can find also the safe areas and the power grid stations. In the bar at the bottom, you can find the current percentage of the battery level of the car, the time since the reservation started, the current fee and the buttons to activate the money saving option and to end the ride.

In this picture the money saving option is off and the end ride bottom is enabled because the user is in a safe area and so he/she can park and leave the car. When the user is not in a safe area the button becomes grey, to indicate that it is disabled. Using the search bar, the user can select a

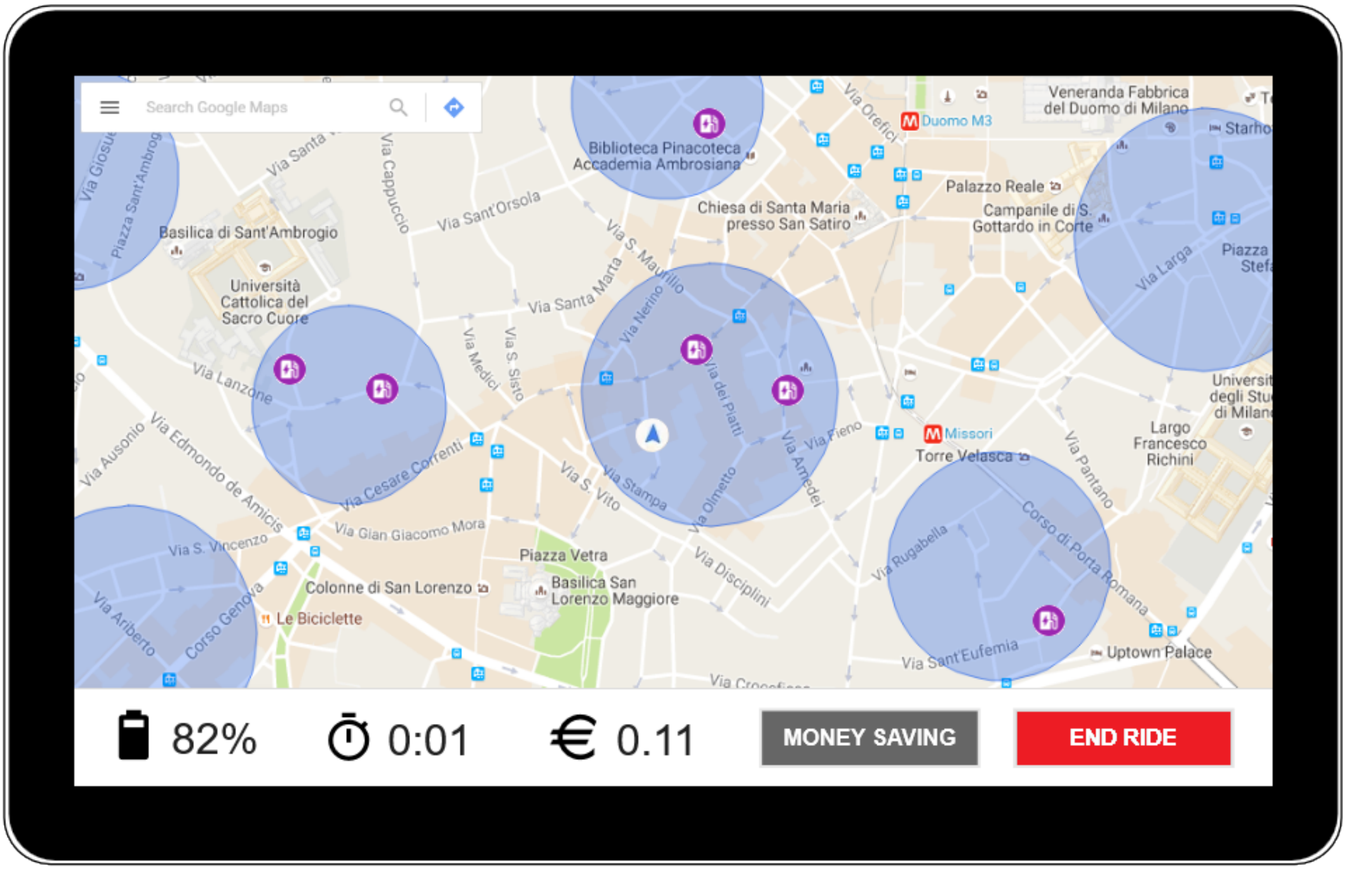


Figure 4.2: Car main page.

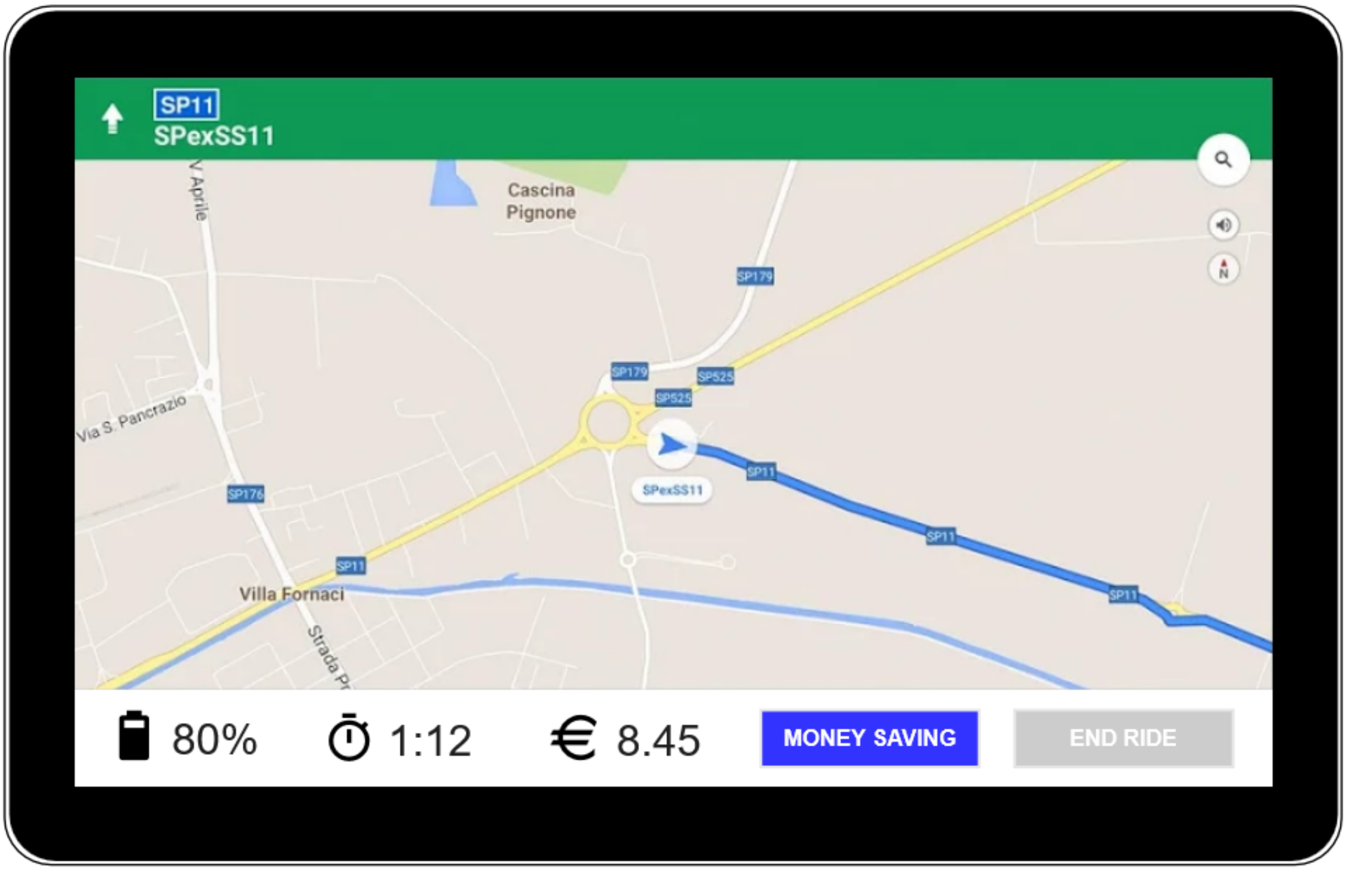


Figure 4.3: Car main page.

destination and set the navigation.

If he/she do so, the screen of the tablet will be something similar to figure 4.3. In this example the money saving button is blue, so it means that this option is activated.

Chapter 5

Requirements Traceability

In this chapter we will describe how the design element defined in the previous sections will meet the requirements listed in the RASD document.

Requirement		Design Solution	
R.1.1	The system must turn the state of the car from reserved to available if the user did not unlock it within an hour since the moment of the reservation.	CMP: Car Manager	The CarManager monitors every reservation and it takes care of doing this.
R.1.2	The system must turn the state of the car from available to reserved after the user reserves it.	CMP: Car Manager	The CarManager does it.
R.2.1	The system must be able to check the position of the user through the GPS position of his mobile phone.	CMP: Car Manager	When a user wants to unlock a car through the mobile app, it does this check.

R.2.2	The system must be able to show all the available cars, with their current level of battery, within a range of 1 km from the position of the user.	CMP: Car Manager	The function *List <Car> getAvailableCars(Position center, float radius)* does it if the center is the position of the user and if the radius is equal to 1000.
R.2.3	If there are not any available cars within a range of 1 km from the position of the user the system must notify the user.	CMP: Car Manager	The function *List <Car> getAvailableCars(Position center, float radius)* returns an empty list and the client can advise the user.
R.3.1	The system must be able to identify a specified location.	CMP: Google Maps API	Through the Google Maps APIs the system is able to do it.
R.3.2	The system must be able to show all the available cars, with their current level of battery, within a range of 1 km from that specific location.	CMP: Car Manager	The function *List <Car> getAvailableCars(Position center, float radius)* does it if the center is the desired location and if the radius is equal to 1000.
R.3.3	If there are not any available cars within a range of 1 km from that specific location the system must notify the user.	CMP: Car Manager	The function *List <Car> getAvailableCars(Position center, float radius)* returns an empty list and the client can advise the user.
R.4.1	The system must be able to check if the user who wants to unlock the car is the same who reserved it.	CMP: Car Manager	The function *void unlockCar(String plate)* does this check.
R.5.1	The system must be able to show to the user the position of the safe areas on the map.	CMP: Safe Area Manager	The function *List <SafeArea> getSafeAreas()* does it.

R.5.2	The system must notify the user when he/she is inside a safe area.	CMP: Google Maps APIs	Through the Google Maps APIs the client is able to do it.
R.5.3	The system must end the reservation only when the car is parked in a safe area and the user exits the vehicle.	CMP: Car Manager	The function *void endRide(int maxNumberOfPassengers)* does this check.
R.5.4	When the reservation ends the system must change the status of the car in available and lock the car.	CMP: Car Manager	The function *void endRide(int maxNumberOfPassengers)* does it.
R.6.1	The system must be able to calculate the distance between the car and the mobile phone of the user using the GPS positions.	CMP: Car Manager	The Car Manager is able to do it.
R.6.2	The system must show to the user the option to unlock the car at least when the distance between the car and the mobile phone of the user is less than 6 meters.	Mobile app UI	The Mobile app UI shows this option when the user has a reserved a car.
R.6.3	The system must be able to unlock the cars remotely.	CMP: Car Manager	The function *void unlockCar(String plate)* does it.
R.6.4	The system must be notified when the user enter inside the car.	CMP: Car Manager	The function *void unlockCar(String plate)* does it.
R.6.5	The system must be able to lock the car again if the user doesn't enter inside the car within 3 minutes.	CMP: Car Manager	The Car Manager does it.

R.7.1	The system must be able to calculate the current fee with respect to the amount of money per minute defined by the company.	CMP: Fee Manager	The Fee Manager does it.
R.7.2	The system must be able to show on the tablet present on the car the current fee that the user has to pay.	CMP: Fee Manager	The function *float getFeeWithVariation(String ID)* can send the current fee to the car tablet.
R.8.1	The system must be able to show to the user the position of the power grid station.	CMP: Safe Area Manager	The function *List <Position> getPowerGridStations()* does it.
R.8.2	The system must be able to show to the user the number of free plugs in every power grid station.	CMP: Safe Area Manager	The function *int getFreePlugsNumber(Position powerGridStationCenter)* does it.
R.9.1	The system must allow only the users who has not been banned to search for a car.	CMP: Car Manager	The Car Manager does this check.
R.9.2	The system must allow only the users who has not been banned to reserve a car.	CMP: Car Manager	The Car Manager does this check.
R.9.3	The system must be able to know the result of the payment operation.	CMP: Payments Manager	The function *void payRide(String ID)* does it.
R.9.4	The system must ban a user who has a pending fee.	CMP: User Manager	The User Manager does it.

R.9.5	The system must be able to mark a fee as paid if and only if the payment operation is successful.	CMP: Payments Manager	The Payments Manager does it.
R.9.6	The system must be able to remove the ban from the user when his/her pending fee has been extinguished.	CMP: User Manager, Payments Manager	The User Manager and the Payments Manager do it.
R.10.1	The system must allow the user to enable the saving mode option in every moment from when he/she starts ride.	CMP: Fee Manager	The function *PowerGridStation enableMoneySavingOption()* does it.
R.10.2	The system must be able to recognize the best power grid to station where to park a specified car based on its position and the distribution of the other cars of PowerEnJoy.	CMP: Fee Manager, Safe Area Manager. ALG: 4	The function *PowerGridStation enableMoneySavingOption()* does it.
R.10.3	The system must be able to provide information about the station where to leave the car to obtain a discount.	CMP: Fee Manager	The function *PowerGridStation enableMoneySavingOption()* does it.
R.10.4	The system must ask the user his/her destination.	Car tablet UI	The car tablet UI does it.
R.10.5	The system must be able to check the availability of power plug in a specific station.	CMP: Safe Area Manager	The function *int getFreePlugsNumber(Position powerGridStationCenter)* does it.

R.11.1	There must be some maintainers who repair every broken car after at maximum two working days after the time they broke.	CMP: Car Manager	The function *void reportFailedCar()* signals that the car is broken, then the car Manager send a request to the maintainers external service, which ensure us that the car will be repaired in that period of time.
R.11.2	There must be some maintainers who recharge the cars after at maximum one working day after the time they discharged.	CMP: Car Manager	The HandyCar Board signals to the application server when a car is discharged, then the car Manager send a request to the maintainers external service, which ensure us that the car will be charged in that period of time.
R.11.3	The system must be able to identify a lack of cars in some parts of the city.	CMP: Car Manager. ALG: 4	The Car Manager is able to do it.
R.11.4	If there is a lack of cars in some parts of the city, the system has to notify the maintainers.	CMP: Car Manager	The Car Manager does it.
R.12.1	The system must be able to detect that there are at least two passengers (not including the driver) on the car and in this case it applies a discount of 10% on the final fee of that ride.	CMP: Car Manager, Fee Manager	The function *void endRide(int maxNumberOfPassengers)* communicates the maximum number of passengers and the Fee Manager applies the discount if it is compatible.

R.12.2	The system must be able to detect if the battery level at the end of the ride is greater or equal to the 50% of the total battery level and in this case it applies a discount of 20% on the final fee of that ride.	CMP: Car Manager, Fee Manager	The Car Manager does this check and the Fee Manager applies the discount if it is compatible.
R.12.3	The system must be able to detect if the car is in a recharging station area and it is plugged to the power grid when the reservation ends, in this case it applies a discount of 30% on the final fee of that ride.	CMP: Car Manager, Fee Manager	The Car Manager does this check and the Fee Manager applies the discount if it is compatible.
R.12.4	The system must be able to detect if the battery level at the end of the ride is less than 20% of the total battery level, in this case the system applies a raise of 30% on the final fee of that ride.	CMP: Car Manager, Fee Manager	The Car Manager does this check and the Fee Manager applies the discount if it is compatible.
R.12.5	The system must be able to detect if the car is left more than 3 Km away from the nearest recharging station, in this case the system applies a raise of 30% on the final fee of that ride.	CMP: Car Manager, Safe Area Manager, Fee Manager	The Car Manager communicates the position of the car to the Safe Area Manager, which checks if the distance is respected and in this case, it communicates this to the Fee Manager, that applies the discount.

R.13.1	The system must be able to acquire all the informations for the registration (name, surname, email address, username, birth date, driving license and payment information).	CMP: User Manager	The User Manager does it.
R.13.2	The system must be able to check if all the mandatory fields has been completed with valid data.	CMP: User Manager	The User Manager does it.
R.13.3	The system must be able to check the validity of the driving license through an external service.	CMP: User Manager	The User Manager does it.
R.13.4	The system must be able to check the validity of the payment information through an external service.	CMP: Payments Manager	The Payments Manager does it.
R.14.1	The system must be able to check if the username and password provided are correct.	CMP: User Manager	The User Manager is able to do it.
R.14.2	The system must allow the user login if and only if the provided username and the password are correct.	CMP: User Manager	The User Manager does this check.

Table 5.1: Requirements traceability table

Appendix A

Appendix

A.1 Used software and tools

- LaTeX [1], for typesetting this document.
- Texmaker[2], for the writing of this document.
- GitHub[3] for version control and distributed work.
- Evolus Pencil[4] for the mockups.
- StarUML[5] for the class diagram.
- GitHub desktop[6] used to collaborate in the team and to keep track of the changes.

A.2 Changelog

v1.1:

- corrected some sequence diagram;
- added external service for maintainers;
- corrected typos;

[1] `https://www.latex-project.org/`
[2] `http://www.xm1math.net/texmaker/`
[3] `https://github.com/`
[4] `http://pencil.evolus.vn/`
[5] `http://staruml.io/`
[6] `https://desktop.github.com/`

- diagram updated in order to add external service for maintainers;

v1.0:

- initial release.

A.3 Work hours

The statistics about commits and code contribution are available on the GitHub repository of the project[7].

These are our estimation of the work hours spent on this project:

- Marco Ieni: 23 hours
- Francesco Lamonaca: 18 hours
- Marco Miglionico: 24 hours

[7] `https://github.com/marcomiglionico94/Software-Engineering-2-Project`