Politecnico di Milano

A.Y. 2016-2017

Software Engineering II: “PowerEnJoy”

Design **D**ocument

version 2.0

February 5, 2017

Pozzati Edoardo (mat. 809392), Stefanetti Nadia (mat. 810622)

Table of Contents

1 Introduction

1.1 Purpose

This document provides a more specific description of the architecture of the PowerEnJoy system based on the RASD presented in the previous delivery, adopting the IEEE-1016 standard for DD documentation.
The purpose of this document is to describe the technical details of the system that will be implemented for PowerEnJoy, by identifying the high-level architecture, the main components as well as their interfaces and their interactions. Through UML standards and other intuitive diagrams, we will explain the relations among different modules, in order to guide the software development team and provide a stable reference.

1.2 Scope

The system aims to support a car-sharing service that exclusively employs electric-cars and allows users to easily find a car, reserve it and use it: this service will be supported on web and mobile applications.
The architectural descriptions provided concern the functional view, module view, deployment view, data layer, business logic and the user interface of the RASD.
The architecture of our system will be thoroughly described in this document, reminding that it will be adapted for all the types of actors that will interact with the system-to-be, by generating a client-server dualism, hence a flow of requests-responses.

1.3 Document Structure

This document specifies the architecture of PowerEnJoy spreading from the general into the specific. In addition, it describes the architectural decisions and trade-offs and justifies them.
The design process follows a top-down approach and the document structure reflects this tactic: the outermost tiers are first identified and then broken into components that encapsulate the functionality. Hence, each component is responsible for certain functionalities and interacts with others.
The document is organized as follows:

- Section 1: Introduction. This section describes the purpose of the project, with a general introduction and overview of the Design Document and the covered topics.
- Section 2: Architectural Design. It specifies the general system architecture,

describes the basic structure and interactions of the main subsystems. This section is divided into different subparts, whose focus is mainly on design choices, interactions, architectural styles and patterns.

- Section 3: Algorithm Design. It provides a high-level description of some important algorithms that will be implemented in our system-to-be, using pseudo-code in order to ease the comprehension.
- Section 4: User Interface Design. It provides an overview on how the user interface will look like, discussing the main choices in UX and BCE design.
- Section 5: Requirements Traceability. It describes how the goals defined in the RASD are mapped to the design components defined in this document.
- Section 6: Appendixes. It provides supporting information and additional material, such as the list of used tools, hours of work, acronyms and reference documents.

2 Architectural Design

In this section, we provide a detailed view of the physical and logical infrastructure of the system-to-be, as well as the description of the main components and their interactions.

2.1 Overview

The PowerEnJoy's system will be design considering the client-server 4-tier JEE application, distributed between client machines, Java EE server machines and databases.
This choice of layers can have many beneficial effects on your application. First, since the architecture is so simple, it is easy to explain to team members and so demonstrate where each object's role fits into the "big picture".
Another benefit of this layering is that it makes it easy to divide work along layer boundaries.
Finally, a four-layer architecture makes it possible to code the bulk of your system (in the domain model and application model layers) to be independent of the choice of persistence mechanism and windowing system.

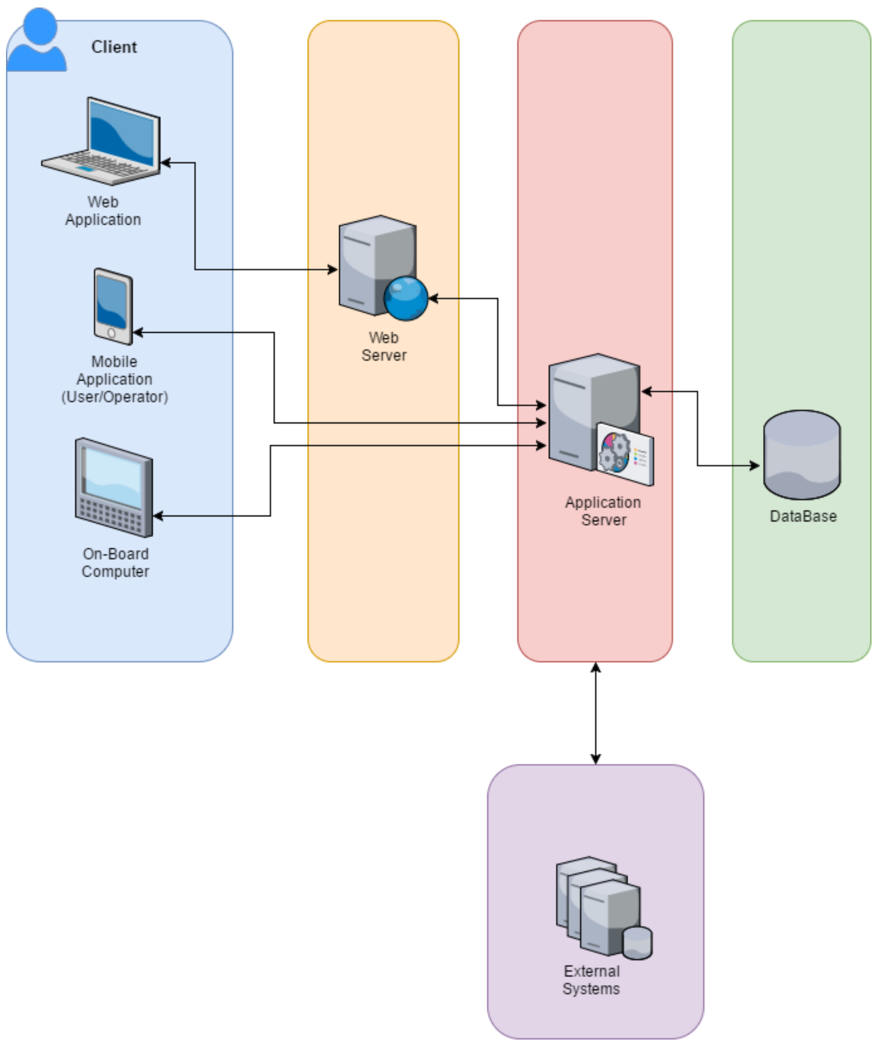


Figure 2.1: Layered structure of the system.

2.2 High level components and their interaction

This section provides a general description of the software system including its functionality and concerns related to the overall system and its design.
The main high-level components of the system are described in the following way (see Figure 2.1: FourTierArchitecture):

• Client Tier: This is the presentation layer where the physical window and widget objects live. It manages the interaction with the operator and his/her mobile application as well as the interaction with the user and therefore contains the mobile application, the web application and the on-board computer.

• Web Server: This layer is in charge of providing web pages for the web-based application. It deals with presentation logic and pages rendering.

- Application Server: This layer deals with the core functionalities of the system. It mediates between the various user interface components and coordinates the application, processes commands, makes logical decisions and evaluations and performs calculations. It is responsible for the flow of the application, controls navigation from window to window and moves and processes data between the client and databases, and therefore encloses all the logic. In addition, it includes the interaction with external systems and encapsulates the domain model.

- Database: This is where the objects that represent our DB reside, without any application logic. This system data layer includes SQLTables and DBMS, that must guarantee the correct functioning of the data structure. This tier holds the information of the system data model and it includes all structures and entities in charge of storing, retrieving and management, while assuring the ACID properties of transactional databases.

The layering approach is the mostly accepted solution for enterprise applications, which require modularity and easy maintenance. Notice that the choice of separating the Application and Web Server layers allows greater scalability, since it allows the deployment on distinct physical tiers that can be individually optimized to perform their respective task.

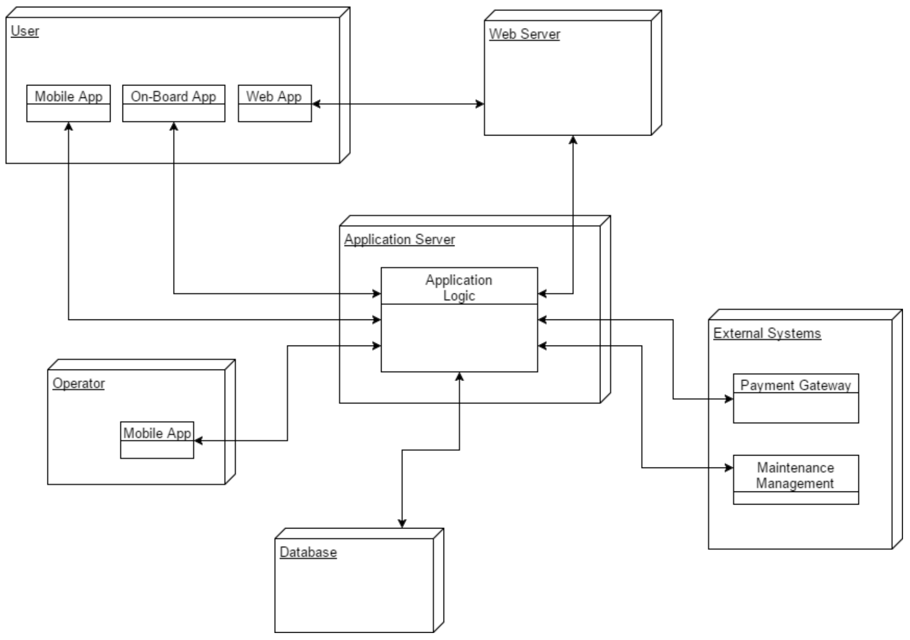


Figure 2.2: The high-level components of the system.

2.3 Component view

In this section, it is provided a detailed insight of the components described in the previous section. Therefore, we will discuss the high-level sub-parts of our system and their functioning, as well as how those sub elements interface with one another.

2.3.1 Client Tier

2.3.1.1 Mobile Application
This represent the front-end for mobile devices such as smartphones and tablets. It does not include any application logic and communicates directly with the Application Server.

2.3.1.2 Web Application
It represents the front-end for Web Browser devices, and does not include any application logic and communicates with the Application Server through a Web Server.

2.3.1.3 On-Board Application
The presentation layer dedicated to the on-board computer applications. Here only objects found in an OO analysis and design reside, such as any type of data that will be retrieved from sensors and devices on the on-board computer that will be than processed by the Application Server.

2.3.2 Web Server
The Web Server layer includes components that manage the interaction between Web Application Client and Application Server. There will not be any logic implemented within the Web Server besides the presentation of pages, which must be structured in an intuitive and clean way.

2.3.3 Application Server
This layer must handle the business logic in its completeness and the connections with the data layer and the different ways of accessing the service. The interface with the data layer must be handled by a dedicated persistence unit, that will be in charge of the dynamic data access and management; this to ensure that only the Application Server can access the database.

2.3.4 Database
The data layer must be accessible only through the Application Server and it must ensure security by granting data access according to the privilege level of the requester. Sensible data such as personal information and passwords must be encrypted properly before being stored.

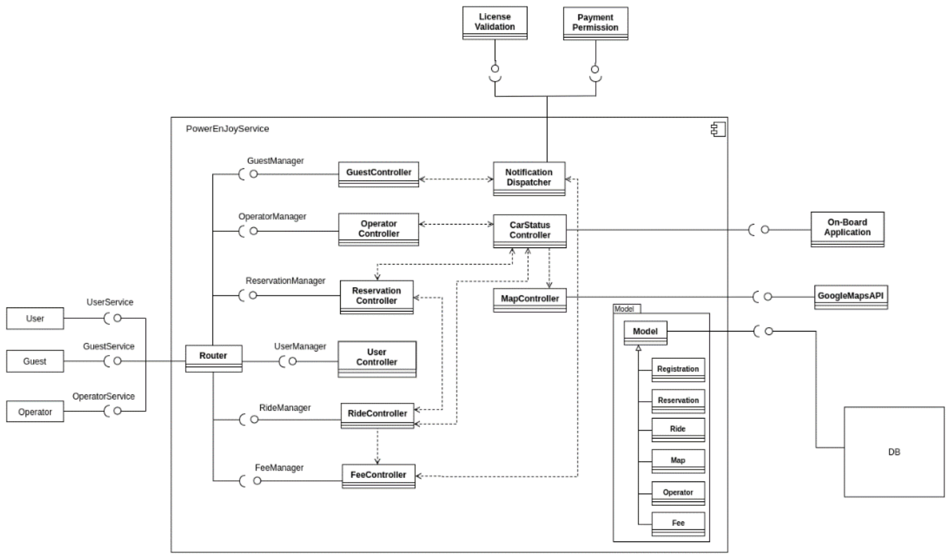


Figure 2.3: The global component view.

2.4 Deployment view

In this section, we describe a set of indications on how to deploy the illustrated components on physical tiers.

2.4.1 Mobile Application Implementation

The mobile application UI must be designed following the design guidelines provided by the Android, iOS and Windows Phone producers. The application must be able to manage the GPS modules connection of the device and must keep track of locations data, in order to provide said information and operator or user inputs to the Application Server via RESTful APIs.

2.4.2 On-Board Application Implementation

Since the On-Board Application is designed to run on every car on pre-existing embedded devices, which come together with the rest of the car equipment directly from the manufacturer, the application must be designed following the guidelines provided by the manufacturer and must be based on given APIs.
The communication with the Application Server must be performed by a connectivity unit via an adequate RESTful API.

2.4.3 Web Browser Application Implementation

The Web Application sends requests to the Web Server via a web browser. The provided html pages include Javascript to manage all components and the screen rendering on the client.
The interactions between the client's browsers and the Web Server are based on the HTTP protocol.

2.4.4 Web Server Implementation

The core components of the Web Server will be implemented with JEE web components. In particular, the application will use JavaServer Pages (JSP) as

the technology for developing the web presentation logic based on the MVC pattern. The choice of JEE also allows the Web Server to make use of servlets, based on the Glassfish requests, dynamically generating the necessary data to the client. In addition, they collect the inputs provided by the user, by sending the messages to the appropriate components in the business tier. Servlets are also used to keep users' data sessions.
The interface with the Application Server will be provided by the same RESTful API.

2.4.5 Application Server Implementation

The main choice for this layer is the use of Java Enterprise Edition 7 (JEE).
The Application Server implementation consists of the Enterprise JavaBeans that manage the logic of system. We decided to implement a Session Bean for each functionality, splitting inside which operations can be invoked by the various types of actors. In addition, it contains inside Entity Beans, derived from an analysis of our database.
We have chosen this implementation because the final product needs distribution to great numbers of clients simultaneously and it needs to satisfy continuously evolving functional requirements and customer demands. Also, JEE allows can reduce the complexity of the application by using mechanisms and models that easily adapt to a large-scale project.
The implementation choices in details are:

- GlassFish Server as the Application Server implementation;
- Java Persistence Entities (JPE) as the persistence unit to perform the object-relation mapping and the Database access;
- Session Beans and Message Driven Beans to implement the single business logic module for each functionality;
- appropriate RESTful APIs to interface with the Client Tiers and external systems partner (payment handlers, license validation, customer service assistance and maintenance system).

2.4.6 Database Implementation

The database implementation must include a relational DBMS component that will handle the insertion, update, deletion and logging of transactions on data inside the storage memory.
The implementation choices for the Database layer are:

- MySQL as the relational DBMS;
- Java DataBase Connectivity (JDBC) with the Application Server, which will define how the Application Server may access the database.

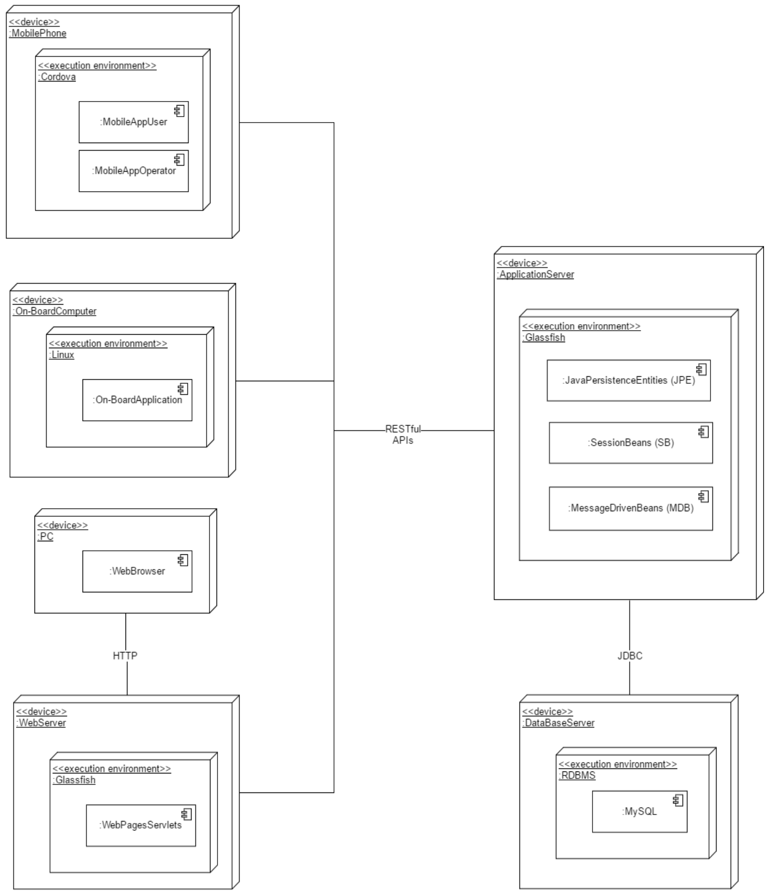


Figure 2.4: The global deployment view diagram for the system.

2.5 Runtime view

This section gives a thorough description of the dynamic behaviour of the system, complete with diagrams for the key functionalities.
The following sequence diagrams highlight the runtime interactions between clients, servers and the database among sub-components of the Application Server.

2.5.1 RegistrationSD

In this sequence diagram, it can be seen that the Guest (a non-registered User) has to input, on the Guest's Mobile Application:
- First, his credential information: the System checks the uniqueness of the email on the database.
- Second, his license information: the System checks the validity of license through an external component and the uniqueness on the database
- Third, his payment information: the System checks the validity of the payment through an external component.
After these controls, the System generates a unique password connected to the email, the Guest become a User.

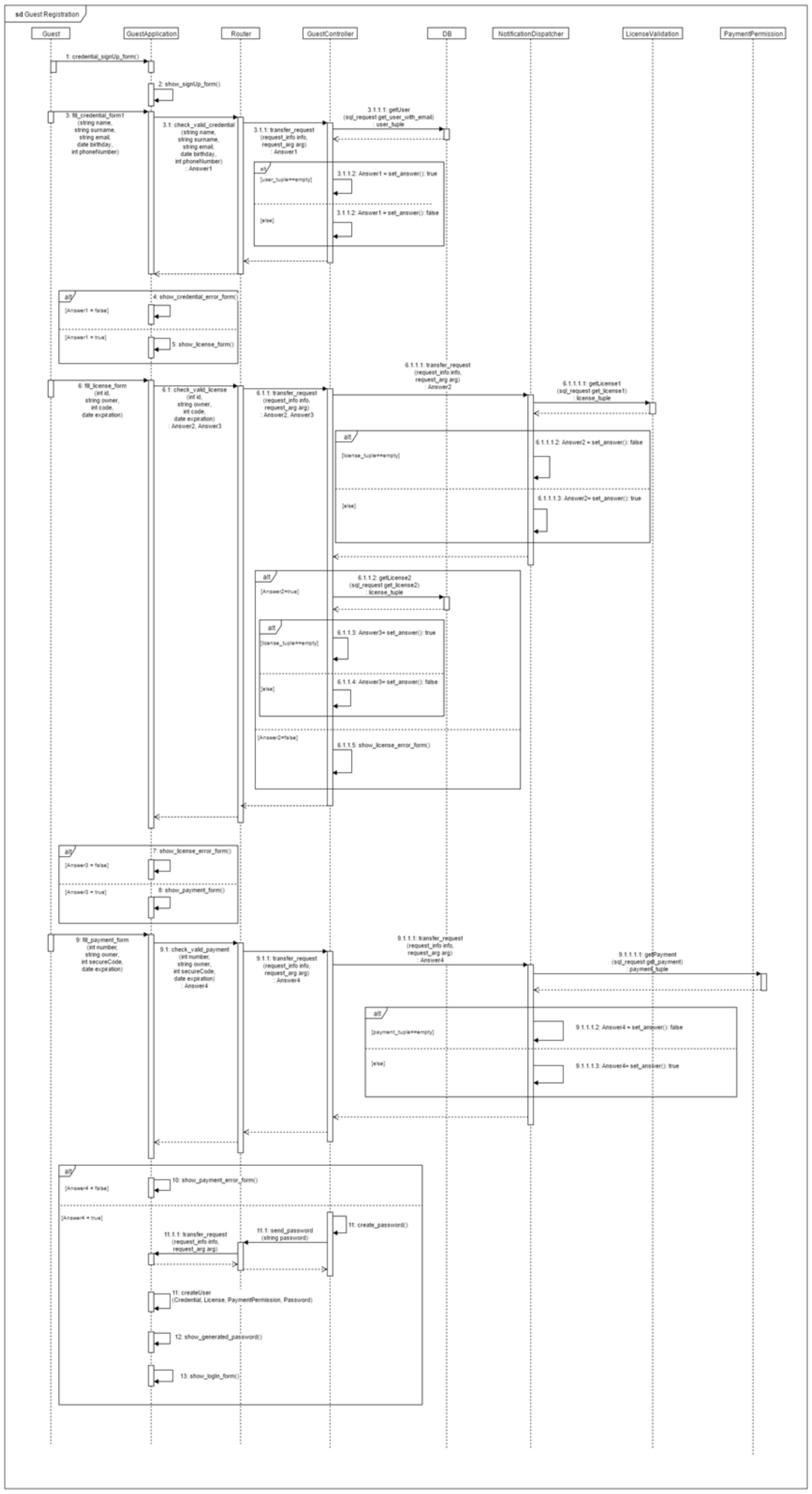

sd Guest Registration
Guest
GuestApplication
Router
GuestController
DB
NotificationDispatcher
LicenseValidation
PaymentPermission
1: credential_signUp_form()
2: show_signUp_form()
3: fill_credential_form1 (string name, string surname, string email, date birthday, int phoneNumber)
3.1: check_valid_credential (string name, string surname, string email, date birthday, int phoneNumber) : Answer1
3.1.1: transfer_request (request_info info, request_arg arg) : Answer1
3.1.1.1: getUser (sql_request get_user_with_email) : user_tuple
alt
[user_tuple==empty]
3.1.1.2: Answer1 = set_answer(): true
[else]
3.1.1.2: Answer1 = set_answer(): false
alt
[Answer1 = false]
4: show_credential_error_form()
[Answer1 = true]
5: show_license_form()
6: fill_license_form (int id, string owner, int code, date expiration)
6.1: check_valid_license (int id, string owner, int code, date expiration) : Answer2, Answer3
6.1.1: transfer_request (request_info info, request_arg arg) : Answer2, Answer3
6.1.1.1: transfer_request (request_info info, request_arg arg) : Answer2
6.1.1.1.1: getLicense1 (sql_request get_license1) : license_tuple
alt
[license_tuple==empty]
6.1.1.1.2: Answer2 = set_answer(): false
[else]
6.1.1.1.3: Answer2= set_answer(): true
alt
[Answer2=true]
6.1.1.2: getLicense2 (sql_request get_license2) : license_tuple
alt
[license_tuple==empty]
6.1.1.3: Answer3= set_answer(): true
[else]
6.1.1.4: Answer3= set_answer(): false
[Answer2=false]
6.1.1.5: show_license_error_form()
alt
[Answer3 = false]
7: show_license_error_form()
[Answer3 = true]
8: show_payment_form()
9: fill_payment_form (int number, string owner, int secureCode, date expiration)
9.1: check_valid_payment (int number, string owner, int secureCode, date expiration) : Answer4
9.1.1: transfer_request (request_info info, request_arg arg) : Answer4
9.1.1.1: transfer_request (request_info info, request_arg arg) : Answer4
9.1.1.1.1: getPayment (sql_request get_payment) : payment_tuple
alt
[payment_tuple==empty]
9.1.1.1.2: Answer4 = set_answer(): false
[else]
9.1.1.1.3: Answer4= set_answer(): true
alt
[Answer4 = false]
10: show_payment_error_form()
[Answer4 = true]
11: create_password()
11.1: send_password (string password)
11.1.1: transfer_request (request_info info, request_arg arg)
11: createUser (Credential, License, PaymentPermission, Password)
12: show_generated_password()
13: show_login_form()

Figure 2.5: *Registration* sequence diagram.

2.5.2 ReservationSD

In this sequence diagram, it can be seen that when a User should perform a reservation he must choose the position option: if he wishes to search a car around a GPS signal or around an address. User's Mobile Application shows him a map with the available cars; he can check the details' car (e.x. Battery State) and also confirm the reservation. After that a timer starts, User's Mobile Application has two new button: "Unlock Car", he can use it to tell the System that he is near, and "Deletion", he can use it to cancel the reservation up to one hour.
Otherwise he must pay a fee if he unlocks the car and leaves it or if the timer expires without a deletion.

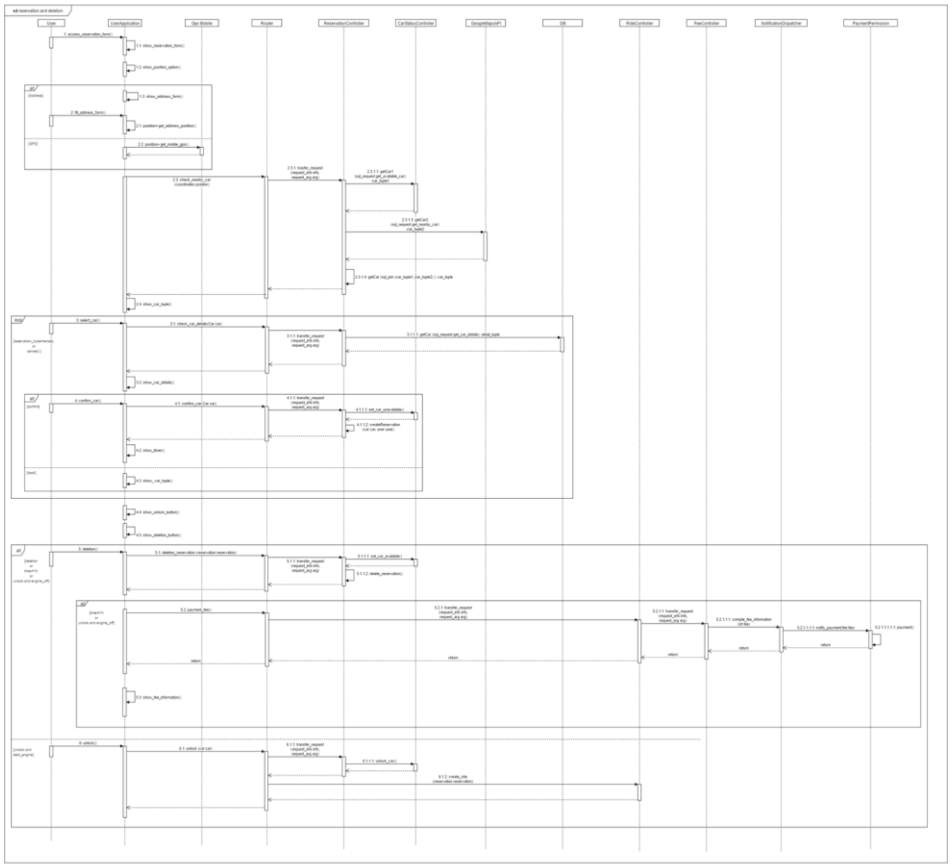


Figure 2.6: *Reservation* sequence diagram.

2.5.3 PaymentProcessSD

In this sequence diagram, it can be how the System calculates the final Fare, by On-Board Application it detects how many passengers there are, how much battery the car has and how far the car is from the nearest Special Safe Area. After that he increase the price or decrease the price according to these rules, they are not cumulative and they are in order of importance:
1 - Charge 30% more in the last ride if a car is left at more than 3 Km from the nearest power grid station or with more than 80% of the battery empty.
2 - Discount of 30% on the last ride if the car is left at a special parking area where it can be recharged and the user takes care of plugging the car into the power grid.
3 - Discount of 20% on the ride if the car is left with no more than 50% of the battery empty.
4 - Discount of 10% on the ride if it detects that the user took at least two other passengers onto the car.
The FeeController calculates the new Fare and automatically charges on User' Payment Information.

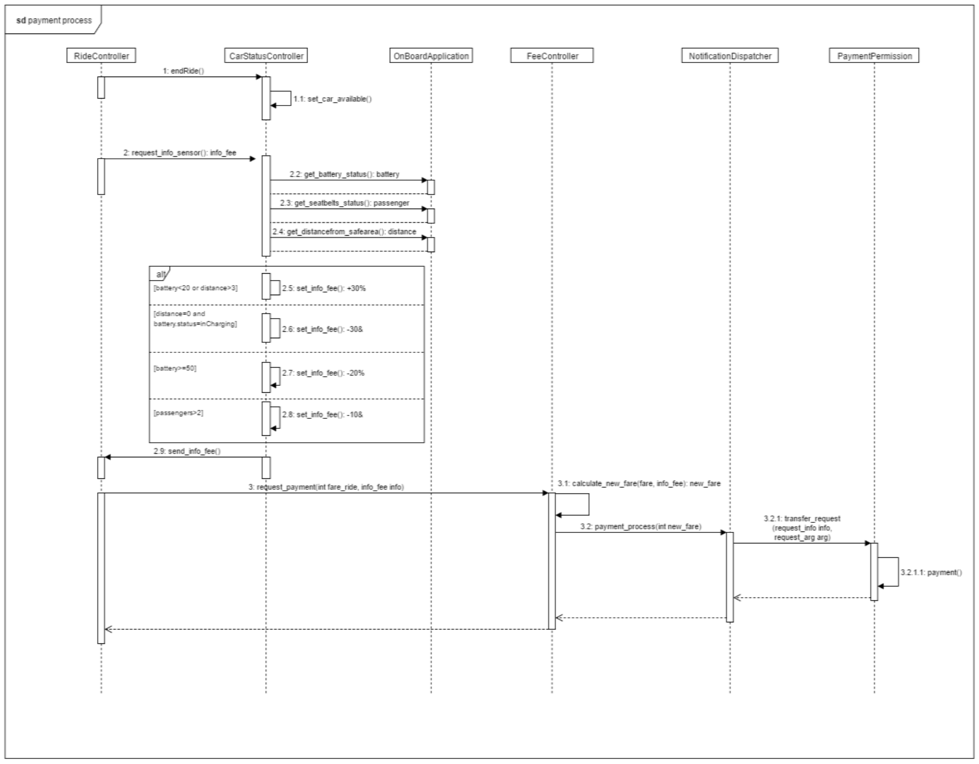
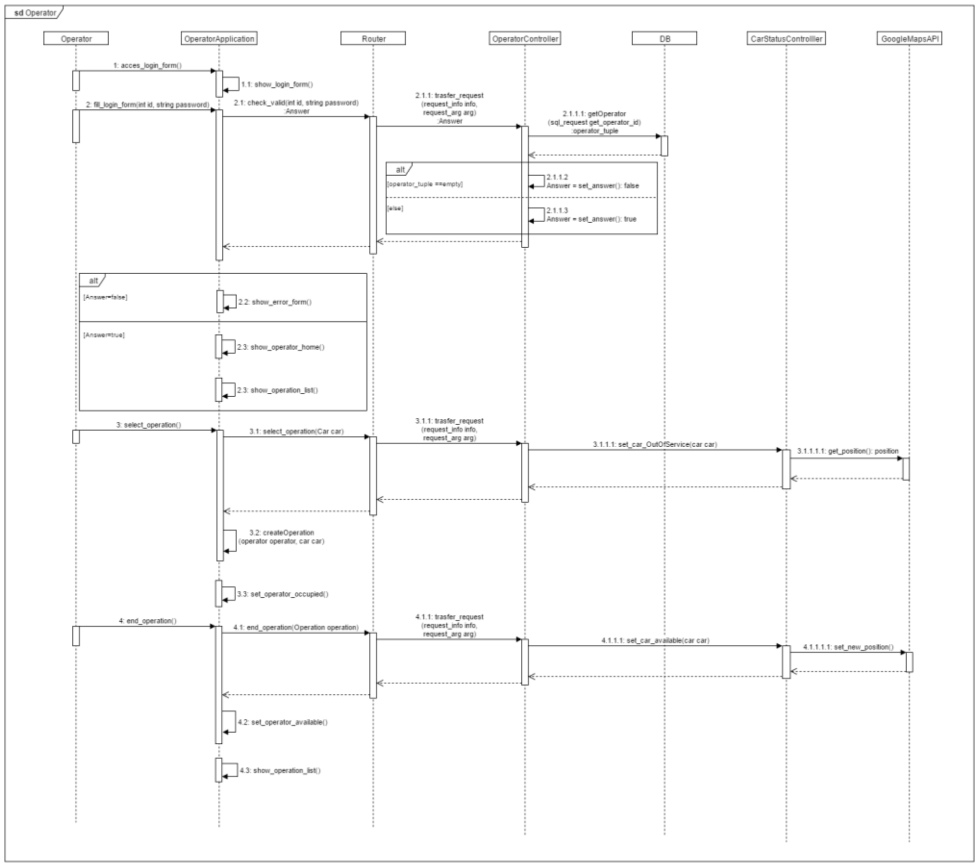

sd payment process
RideController
CarStatusController
OnBoardApplication
FeeController
NotificationDispatcher
PaymentPermission
1: endRide()
1.1: set_car_available()
2: request_info_sensor(): info_fee
2.2: get_battery_status(): battery
2.3: get_seatbelts_status(): passenger
2.4: get_distancefrom_safearea(): distance
alt
[battery<20 or distance>3]
2.5: set_info_fee(): +30%
[distance=0 and battery.status=inCharging]
2.6: set_info_fee(): -30&
[battery>=50]
2.7: set_info_fee(): -20%
[passengers>2]
2.8: set_info_fee(): -10&
2.9: send_info_fee()
3: request_payment(int fare_ride, info_fee info)
3.1: calculate_new_fare(fare, info_fee): new_fare
3.2: payment_process(int new_fare)
3.2.1: transfer_request (request_info info, request_arg arg)
3.2.1.1: payment()

Figure 2.7: *Payment process* sequence diagram.

2.5.4 OperatorSD

In this sequence diagram, it can be seen that the Operator has to input his login information on the Operator's Mobile Application. The login request is then sent with this information as parameter to the DataBase to check the correctness. After that he can see the list of operation to do with their information (position, messages,...), he can select one of these and perform the relocation/operation. Then he send an endOperation message to notify that his availability changes.



sd Operator
Operator
OperatorApplication
Router
OperatorController
DB
CarStatusControlller
GoogleMapsAPI
1: acces_login_form()
1.1: show_login_form()
2: fill_login_form(int id, string password)
2.1: check_valid(int id, string password) :Answer
2.1.1: trasfer_request (request_info info, request_arg arg) :Answer
2.1.1.1: getOperator (sql_request get_operator_id) :operator_tuple
alt
[operator_tuple ==empty]
2.1.1.2 Answer = set_answer(): false
[else]
2.1.1.3 Answer = set_answer(): true
alt
[Answer=false]
2.2: show_error_form()
[Answer=true]
2.3: show_operator_home()
2.3: show_operation_list()
3: select_operation()
3.1: select_operation(Car car)
3.1.1: trasfer_request (request_info info, request_arg arg)
3.1.1.1: set_car_OutOfService(car car)
3.1.1.1.1: get_position(): position
3.2: createOperation (operator operator, car car)
3.3: set_operator_occupied()
4: end_operation()
4.1: end_operation(Operation operation)
4.1.1: trasfer_request (request_info info, request_arg arg)
4.1.1.1: set_car_available(car car)
4.1.1.1.1: set_new_position()
4.2: set_operator_available()
4.3: show_operation_list()

Figure 2.8: *Operator* sequence diagram.

2.6 Component interfaces

This section includes a description of the different type of interfaces among the various described components.
Here there will be described the main business logic modules of the Application Server:

2.6.1 Guest Controller

This module provides the logic behind the guest registration, which requires validation through a Notification Dispatcher that receives information from external provider such as License Validation and Payment Permission.
Here we list the procedures implemented by the GuestController:
newRegistration This procedure is used to request the registration of a guest to the PowerEnJoy services, by taking as parameter first the personal guest's credential, then the license credential and finally the payment information. Each step verifies the validity of the credentials submitted: the return value for this procedure is the new user's password in case of success or an error in case of invalid data submission.

2.6.2 User Controller

This module will manage all the logic involved with login, user profile customization and management, as well as all the functionalities available for logged users.
Here we list the procedures implemented by the UserController:
submitUserLoginCredentials This procedure is used to perform the login action, in order to access the application, by submitting as parameter the user's email and password. Those parameters are processed by the UserController to verify their validity and grant the access: the return value for this function is a user-identifying token or an error if the provided credentials are wrong or non-existing.
viewHistoryRides This procedure is called to fetch the user's history from the database; the taken parameters include the user's primary key and some filters inputs. It returns a (list of) item(s) representing the filtered ride(s).
editProfile This procedure is used when a user wants to modify his/her personal data. It takes as parameters a set of modified objects, and return an error if the modified data are invalid, a confirmation message otherwise.

2.6.3 Reservation Controller

It handles the logic involved in the reservation process in its completeness, (making, deletion and expiration) and therefore it needs to update the information about the car status. It must also guarantee to avoid multiple reservations by a user.
Here we list the procedures implemented by the ReservationController:
getNearbyAvailableCars This procedure is used to indicate the address where a user wants to search nearby available cars. It takes as parameters the submitted address and an integer that represent the range in which to perform the searching process, and returns a subset of available cars based on the position parameter and range.
createReservation It selects the car chosen by a user to be reserved, with the possibility of checking first its details; the procedure takes as parameters a set of cars and a user-identifying token to associated the reservation with; the

return value is a confirmation message and the update in the database.
deleteReservation This procedure is called whenever a user decides to delete his/her pending reservation; the taken parameter is only a user-identifying token; the return value is a confirmation message.

2.6.4 Map Controller

The logic included here is used to retrieve information about the map implemented with all the safe areas within the Coverage Area. It must provide data to the controller modules that need the localization information, which are the Reservation Controller and the Ride Controller.
A complete vision of the map in general is guaranteed through the external GoogleMapsAPIs.

2.6.5 Car Status Controller

It contains the logic to update the status of the cars, which changes through the Reservation Controller, Ride Manager and Operator Controller.
Here we list the procedures implemented by the CarStatusController:
setCarStatus The procedure is called whenever a component needs to update the status of a car; this takes as parameters a car and the status to set for said car; the return value for this procedure is a confirmation message.
unlockCar This procedure is used to state the proximity of the user's position to the reserved car and therefore request the car's doors unlocking. It takes as parameter the user's reservation token and returns nothing, since its call is asynchronous.
isCarInSafeArea This procedure is used to check if a car is parked in a Safe Area once the user has ended his/her ride. The required parameters are the car and its position; the return value is a Boolean of the result of the performed check.

2.6.6 Ride Controller

This module provides the logic to retrieve and process all information about the user's ride, such as departure and arrival time, location destination and discounts/additional charges.
Here we list the procedures implemented by the RideController:
startRide This procedure is used to create a new ride entity after the user has unlocked the car and ignites the engine. It takes as parameters a car and a user-identifying token to associate with the ride; the return value is undefined since the call is asynchronous.
endRide This procedure is called when the user has ended his/her ride. The only needed parameter is a car from which recover all necessary data; the return value is undefined, since the procedure is a void.

2.6.7 Fee Controller

It retrieves information about all the possible discounts or additional charges that must be applied to the user from the Ride Controller, and then communicate the computation of the total amount to the Notification Dispatcher, that has a direct interface to the payment handlers and allows automatic payments.
Here we list the procedures implemented by the FeeController:
computeNewFare This procedure is called at the end of a ride to get all the discounts/additional charges percentages in order to compute the total amount

associated to the concluded ride; it takes as parameter such ride and the car data from the on-board application. The return value is the computation of the total amount embedded in a new fee entity.

2.6.8 Notification Dispatcher

It serves as a gateway from all the modules that need communicate and retrieve information from external modules.
Here we list the procedures implemented by the NotificationDispatcher:
sendPaymentCharges This procedure is called to send the corresponding user’s fare amount to his/her payment gateway; it takes as parameters the user and the computed amount. It returns an undefined value since the procedure includes a call to the NotificationDispatcher that manages the response to the user directly. This procedure is also used to apply a fine after the user’s reservation expires; this takes as parameters a reservation; the return value is undefined since the procedure includes a call to the NotificationDispatcher that manages the response to the user directly.
askHelp This procedure is call any time the user wants to request assistance for any genre. It takes as parameter the user and returns an undefined value since the procedure includes a call to the NotificationDispatcher that will allow communication with an external costumer service assistance.

2.6.9 OperatorController

This module will manage the logic involved with the operator’s login.
Here we list the only procedure implemented by the OperatorController:
submitOperatorLoginCredentials This procedure is used to perform the login action by submitting as parameter the operator’s ID number and password. Those parameters are processed by the OperatorController to verify their validity and grant the access: the return value for this function is a operator-identifying token or an error if the provided credentials are wrong.

2.7 Selected architectural styles and patterns

In this section, we provide a list of the architectural styles, design patterns and paradigms adopted in the design phase.

2.7.1 N-Tier data applications

These are data applications that are separated into multiple tiers. They are also called "distributed applications" and "multi-tier applications". N-Tier applications separate processing into discrete tiers that are distributed between the client and the server. A typical N-Tier application includes a presentation tier, a middle tier, and a data tier. In addition to the advantages of distributing programming and data throughout a network, N-Tier applications have the advantages that any tier can run on an appropriate processor or operating system platform and can be updated independently of the other tiers.

- Database (Data) Tier: At this tier, the database resides along with its query processing languages.

- Application (Middle) Tier: At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. FrontEnd-Clients are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the FrontEnd-Clients and the database.

- Client (Presentation) Tier: FrontEnd-Clients operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside on the application tier.

Multiple-Tier architecture is highly modifiable, as almost all its components are independent and can be changed independently.

2.7.2 Client-Server

Our system design is strongly based on a Client-Server communication model. The mobile and the on-board applications are clients with regard to the Application Server that receives and elaborate requests. In the communication between the web browser and the Web Server, the first one is the client while the second one provides the requested web page. Moreover, the Web Server, in the role of the client, communicates with the Application Server, that is in charge of processing every user's requests. The Application Server, as the client, queries the Database, which, as a server, will get results.

2.7.3 Model-View-Controller

MVC Model-View-Controller pattern has been widely in our application.
The model-view-controller pattern proposes three main components or objects to be used in software development. The MVC allows to separate the application into three communicating and interconnected parts fulfilling the design principle of the separation of concerns.
- Model, which represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface.
- View, which is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth).
- Controller, which represents the classes connecting the model and the view.

2.7.4 Thin Client

The thin client approach has been used with respect to the interaction among user's machines and the system itself. This approach will let the application run on low-resources devices, as the mobile application and the user's browser are in charge of presentation only and they do not involve decision logic.
This approach has been chosen for different reasons: first, it is practical because it allows easy data synchronization, as there is only one application that mange the data. Second, having one unique server application, the maintainability of our system improves. Third, the application is independent from the number of clients connected, as it can be scaled up. Finally, the security between clients also improves, because there are known only the server endpoint but no other clients.

2.8 Other design decisions

In this section, we present a list of all other relevant design decisions not mentioned before.

2.8.1 User’s password storage

For security reasons, the user’s password is stored using cryptographic hash functions.

2.8.2 Maps

We also have to integrate our web application and our mobile applications with a map service, and to do so we have chosen to use an external maps service: Google Maps. Its services will be used to translate addresses to coordinates and vice-versa during the process of searching a car.

3 Algorithm design

Here we give just an idea of the function used to calculate the fee, we will not write complete code.

```
//This function is called when a ride ends.
//battery state is a percentage
//distanceFrom is in km
//inCharging is a boolean (if it is True->distanceFrom=0)
//fare is in Euro


function calculateFee(Car $car, Int $fare, Client $client)
{
   if ($car.batteryState < 20 || $car.distanceFromSpecialSafeArea > 3){

      $newFare = 1.3 * $fare;

      break; //go out from if-state without check other condition
   }

   else if ($car.inCharging == " True "){

      $newFare = 0.7 * $fare;

      break; //go out from if-state without check other condition
   }
else if ($car.batteryState >= 50){

      $newFare = 0.8 * $ride.fare;

       break; //go out from if-state without check other condition
   }

   else if ($car.seatbeltsUsed > 2){
        $newFare = 0.9 * $ride.fare;

        break;
    }

    $client.pay($newFare);

}
```

4 User interface design

4.1 Mockups

We have already done mock-ups in RASD in section 3.1.1.

4.2 UX Diagrams

The purpose of UX (User eXperience) diagrams is to show how guests, users and operators perform main actions. We will provide different diagrams that model our actors interface with the application and their dynamic links.
Furthermore, we will address our analysis to the interactions among the screens themselves and the presence of input forms and required data in a specific screen.
It is our interest to point out that the mobile and web applications implement the same functionalities, so we will attach to this document a single UX diagram for both.

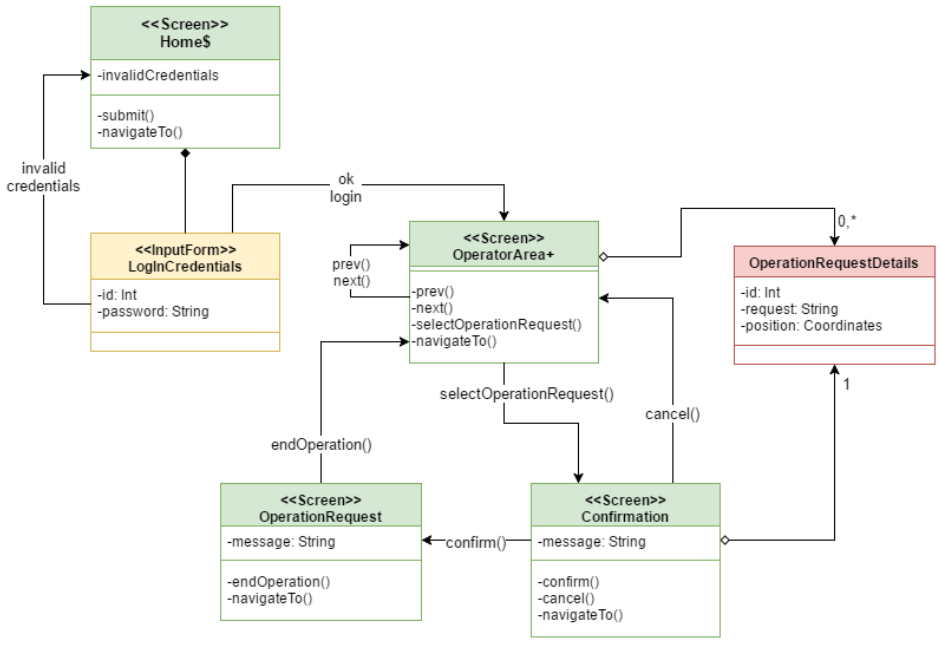


Figure 4.1: Operator UX.

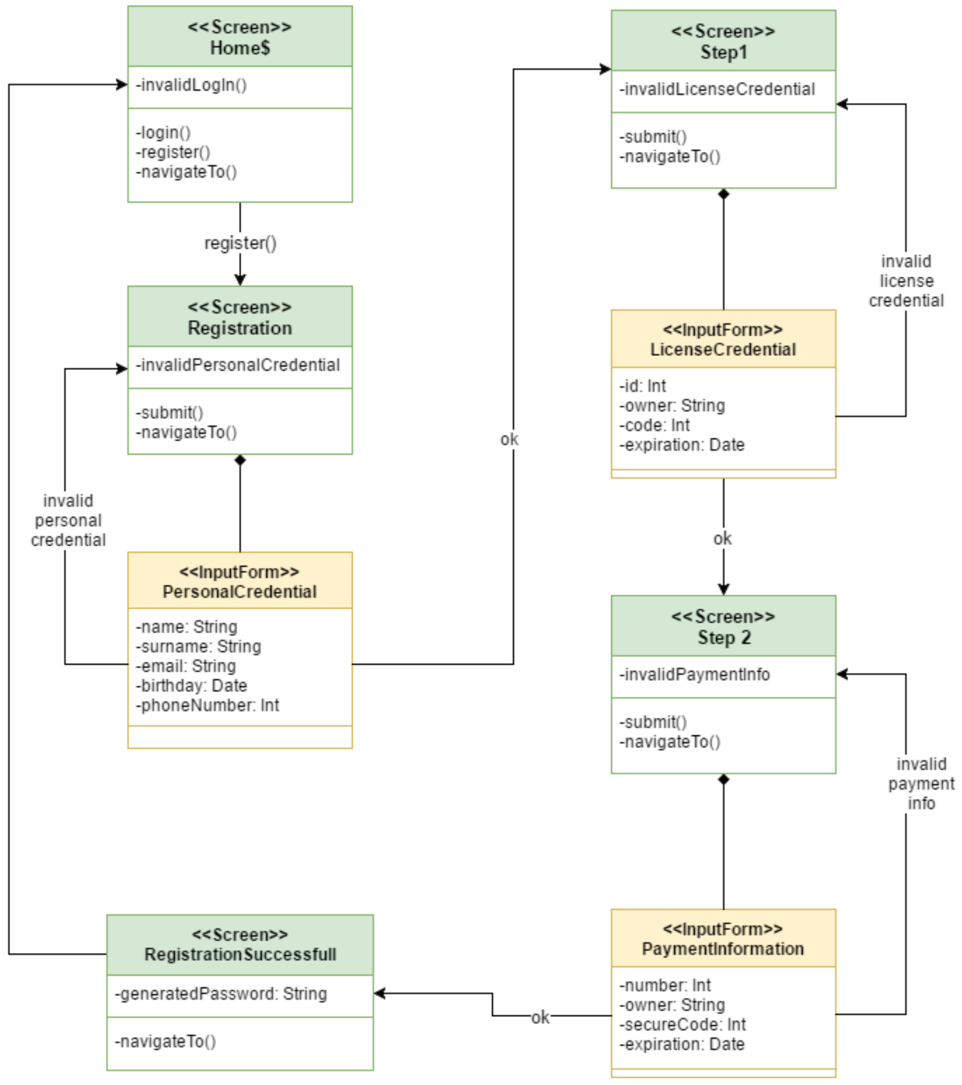


Figure 4.2: Guest’s Registration UX.

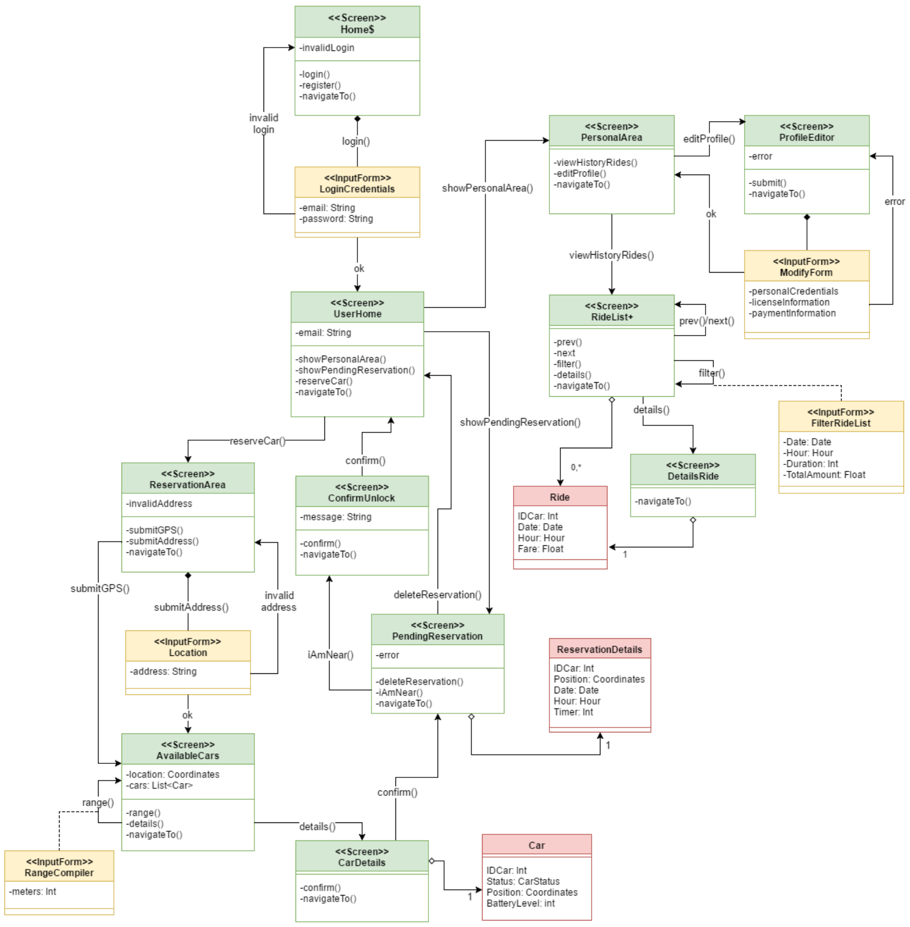


Figure 4.3: User UX.

4.3 BCE Diagrams

We insert BCE (business controller entity) diagrams to show how each user action is managed internally and how it's linked with our model. This diagram is very useful since we use MVC.

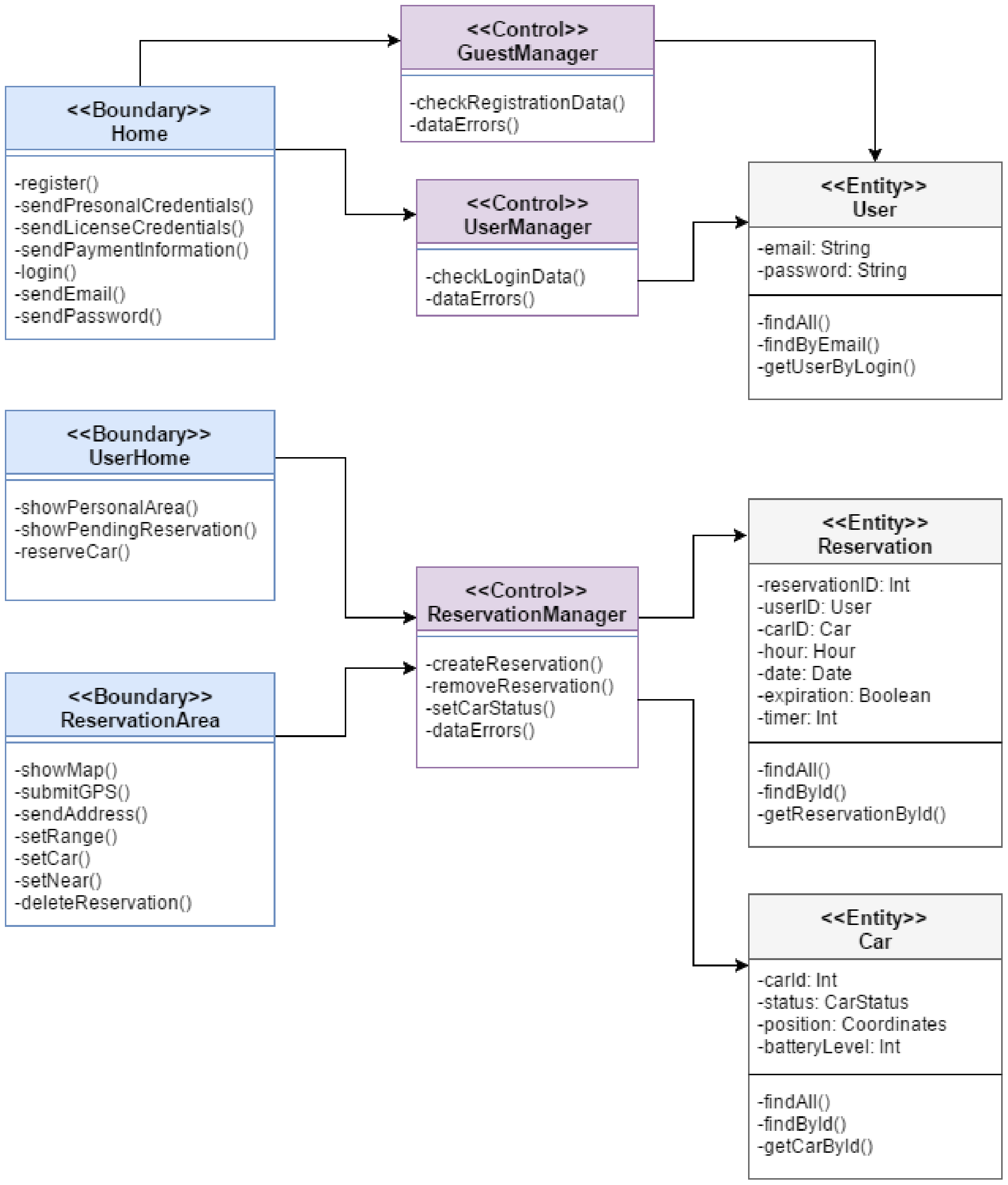


Figure 4.4: User's BCE.

5 Requirement traceability

The design of this project was made aiming to fulfil optimally the requirements and goals specified in the RASD. The reader can find here under the list of the goals the designed component of the application that will assure its fulfilment.

- [G1] Allow guests to register into the system.
  - The GuestController
  - The Router
  - The NotificationDispatcher
  - The LicenseValidation
  - The PaymentPermission

- [G2] Allow users to log into the system and manage their accounts.
  - The UserController
  - The Router

- [G3] Allow user to search the map for the closest available cars to where he is currently located or another indicated address.
  - The MapController
  - The GoogleMapsAPI

- [G4] Allow users to reserve a single car for up to one hour before they pick it up.
  - The ReservationController
  - The CarStatusController
  - The Router

- [G5] Allow users to delete an existing reservation within one hour.
  - The ReservationController
  - The CarStatusController
  - The UserController

- [G6] Allow users to unlock the doors and start the rental.
  - The UserMobile App component
  - The Router
  - The RideController
  - The CarStatusController
  - The MapController

- [G7] Allow the system to charge the user a fee for the service required.

- The RideController
- The PaymentPermission
- The FeeController
- The NotificationDispatcher

- [G8] Allow cars to automatically lock themselves.
  - The RideController
  - The CarStatusController
  - The MapController
  - The OnBoardApplication

- [G9] Allow users to contact the Customer Service Assistance directly for assistance and any other notifications.
  The UserController
  The CallGateway
  The NotificationDispatcher

- [G10] Allow the system to notify an operator to request an operation.
  - The OperatorController
  - The Router
  - The OperatorMobileApp component
  - The CarStatusController
  - The MapController

- [G11] Allow operators to login to the system.
  - The OperatorController and its interface to the database

Appendix A: Used Tools

A.1 Microsoft Word 2013

To redact and format this document.

A.2 *g*it

To submit this document in the online repository.

A.3 Dropbox

Used as version control system in order to lead development.

A.4 Draw.io

To create and design the provided diagrams.

Appendix B: Hours of work

This is the time spent by each group member in order to redact this document:

- Pozzati Edoardo: ~20 hours
- Stefanetti Nadia: ~20 hours
- Total work time: ~40 hours

Appendix C: Changelog

- **v1.0**
- **v2.0**
  - corrected some typos
  - removed Costumer Service Assistance from the external systems in figure 2.2
  - removed “Call Gateway” from the component view, figure 2.3. The Costumer Service will be reached through a call from the user smartphone, so we will not provide any direct interface.
  - Fixed functions order call in subsection 2.6 to completely match the modified component view

Acronyms

- ACID: Atomicity, Consistency, Isolation and Durability. This is the set of properties of database transactions.
- API: Application Programming Interface.
- AS: Application server.
- BCE: Business Controller Entity.
- DB: Data Base.
- DBMS: Data Base Management System.
- DD: Design Document.
- EJB: Enterprise Java Bean.
- GPS: Global Positioning System.
- JDBC: Java DataBase Connectivity.
- JEE: Java Enterprise Edition.
- JPE: Java Persistence Entities.
- MDB: Message Driven Beans
- MVC: Model-View-Controller.
- OS: Operating System.
- RASD: Requirements Analysis and Specification Document.
- REST: REpresentational State Transfer.
  It is an architectural style and an approach to communications often used in the development of Web services.
- RESTful: a REST-compliant system.
- SB: Session Beans.
- UX: User eXperience.

Bibliography

[1] A.Y. 2016/2017 Software Engineering 2 - Requirements Analysis and Specification Document - Pozzati Edoardo, Stefanetti Nadia.

[2] *Luca Mottola and Elisabetta Di Nitto, Software Engineering 2: Project goal, schedule and rules*, 2016.

[3] IEEE Standard 1016:2009 for Information Technology *Systems Design - Software Design Descriptions*.