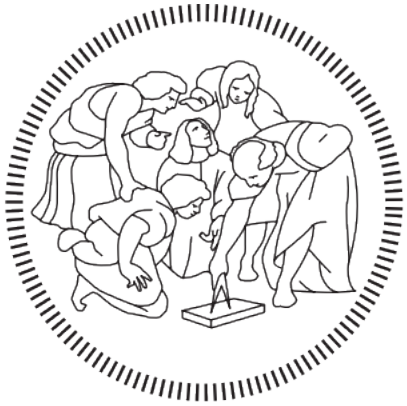
PowerEnJoy project: Design Document

Boriero Stefano 876106 Brunitti Simone 875039

January 15, 2017



POLITECNICO
MILANO 1863

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to present the architectural structure of the system that is going to be developed. First an overview of high level components is going to be provided, then more detailed views are going to descibe the low level interaction between thi components. Runtime views will describe how this interaction take place in real time situation. A brief summary of the architectural styles and decisions that have been made will end the first part of the document, followed by the design of the main algorithms. User interface is presented using both a formal notation with UX diagrams and visual mock-ups. Requirements traceability will close the document explaining how the requirements highlited in the RASD are going to be fulfilled by the designed architecture.

1.2 Scope

Our design architecture will focus mostly on the server side of the system, as we try to keep the client as thin as possible. The part of the system that will run on the car will be analysed as well, as the interaction between the server and the car will define business logic boundaries.

1.3 Glossary

Table 1: Glossary

TERM	MEANING
API	Application Programming Interface
REST	Representational state transfer: an architectural style providing interoperability between computer systems on the Internet
UX	Diagrams User experience diagrams
Payment Gateway	Service that alllows the user to pay through the Internet
Push Notification	A system to notify the user of some changes
DB	Database
Google Maps	A map service offered by Google
Exception	A programming method used to deal with errors
JEE	Java Enterprise Edition
JPA	Java Persistence API

1.4 Reference Document

-http://www.w3schools.com/graphics/google-maps-intro.asp For references on Google Maps API

-MyTaxiService Design Document Sample
-Our PowerEnJoy RASD https://github.com/StefanoBoriero/PowerEnjoy-Boriero-Brunitti/blob/master/releases/RASD-v1.md

1.5 Changelog

- V 1.0: Release Version.
- V 1.1: Added front page, updated fig. 3, removed AssistanceList and BillApplicator components, changed interfaces to comply with ITPD

2 Architectural design

2.1 Overview

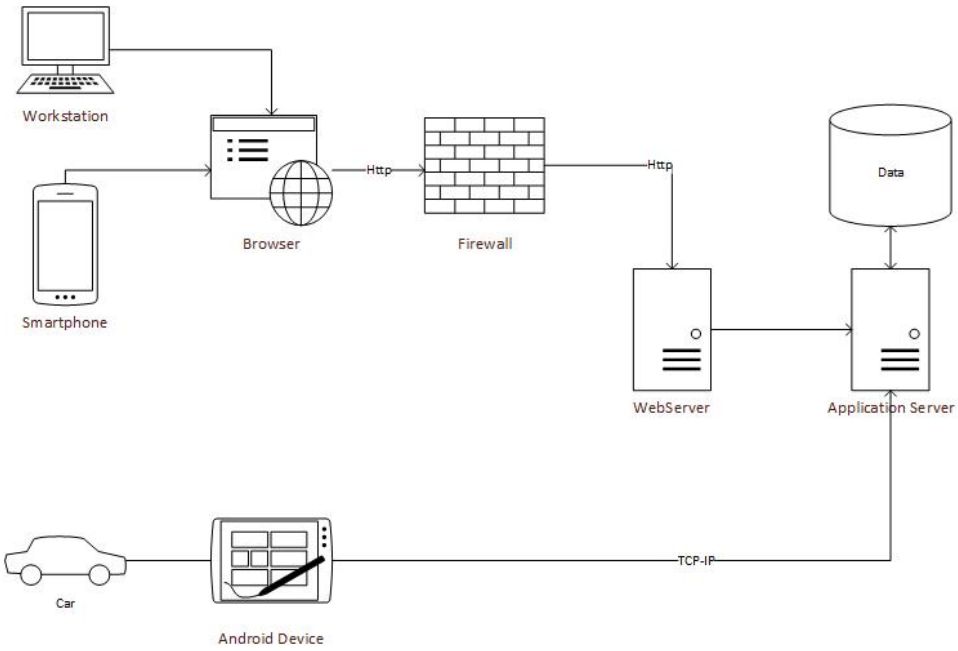


Figure 1: Overview of the system

Both client's and employee's application will communicate with the WebServer throught an HTTP channel, using a Web Browser. The WebServer will translate Http requests into API method call on the API Server, that will interact with the DataBase Server. The on-board Car Application will communicate directly with API Server throught a TCP-IP channel to have better performance while monitoring the real-time ride.

2.2 Component view

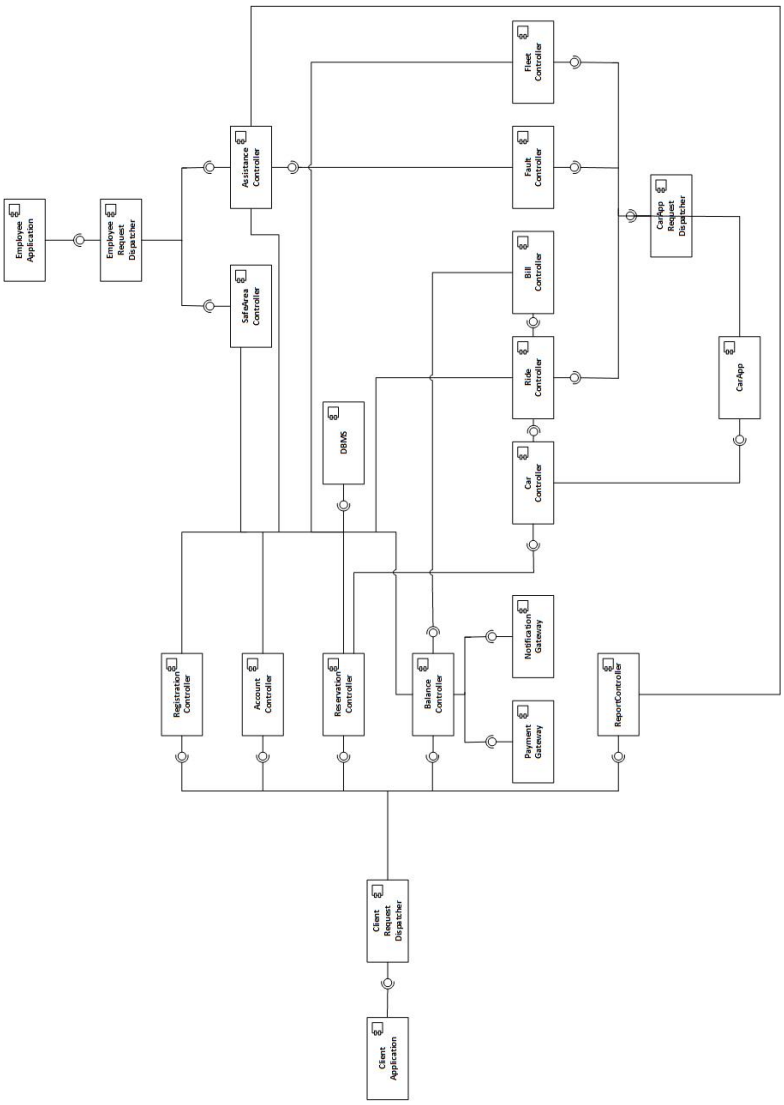


Figure 2: Middle level component view

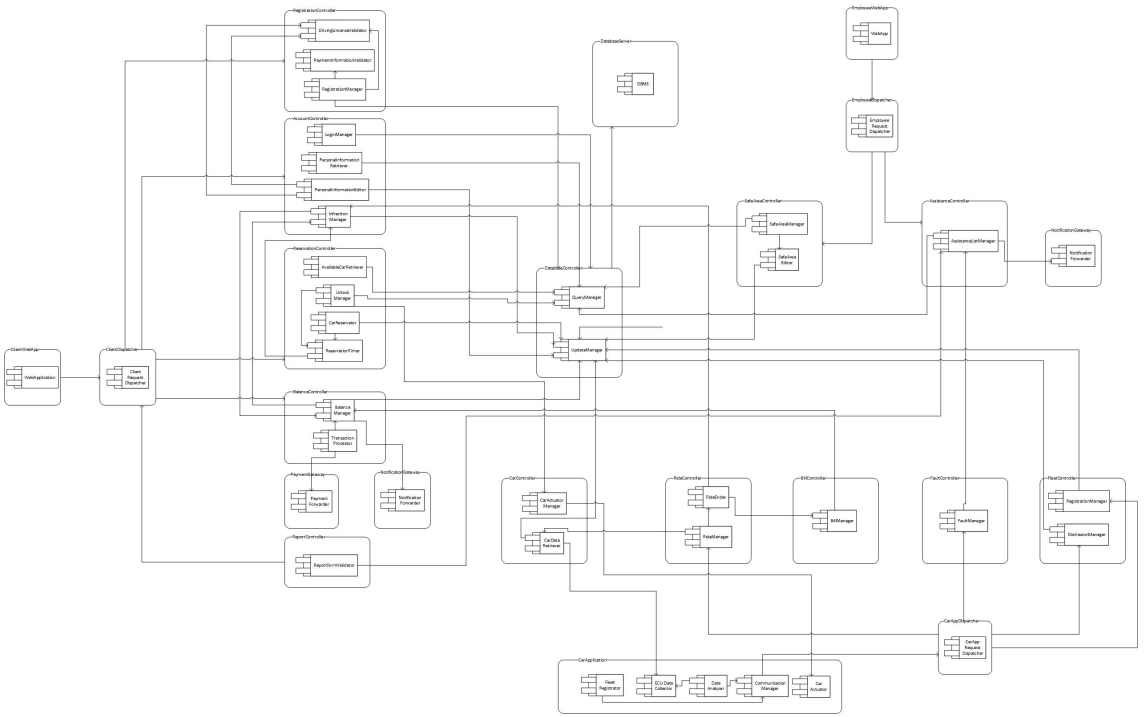


Figure 3: Component View

2.3 Deployment view

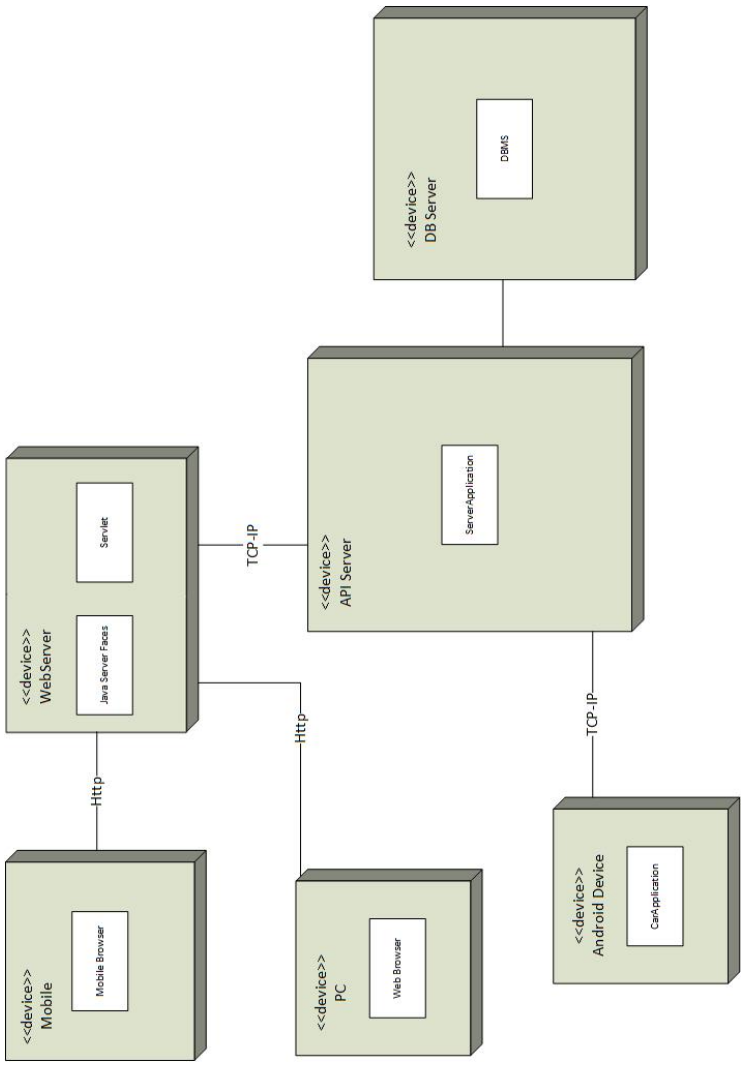


Figure 4: Deployment view

2.4 Runtime view

2.4.1 Sequence diagrams

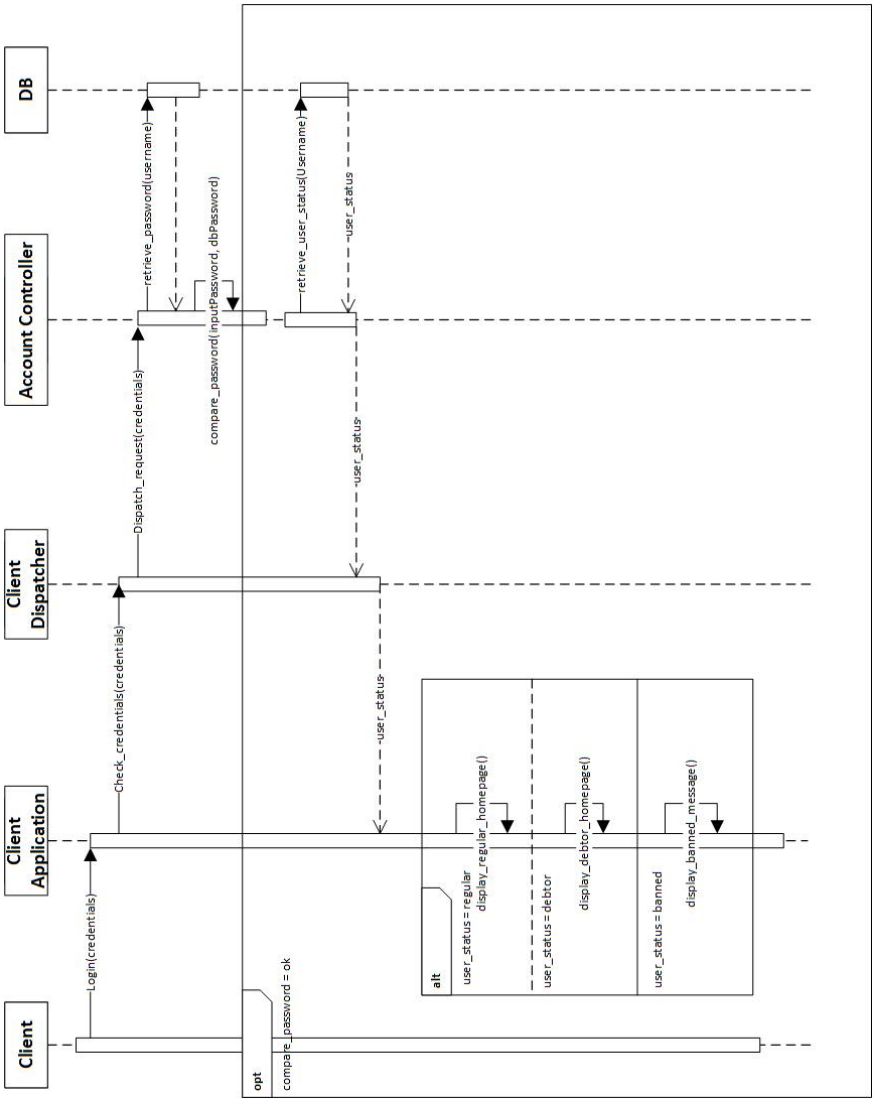


Figure 5: Customer Login

When the user inputs his login credentials in the application, the application must compare this information with the one contained in the database. To do so, the application sends a request to the Client Dispatcher, which transfers the request to the Account Controller. The Account Controller queries the database to retrieve the password corresponding to the input username and subsequently compares the two passwords. If the comparison is positive, the account con-

troller retrieves the user status and sends it to the application. Finallly, the application shows the client a different screen depending on the user status (if the client is a debtor he can't reserve, so that button will be disabled).

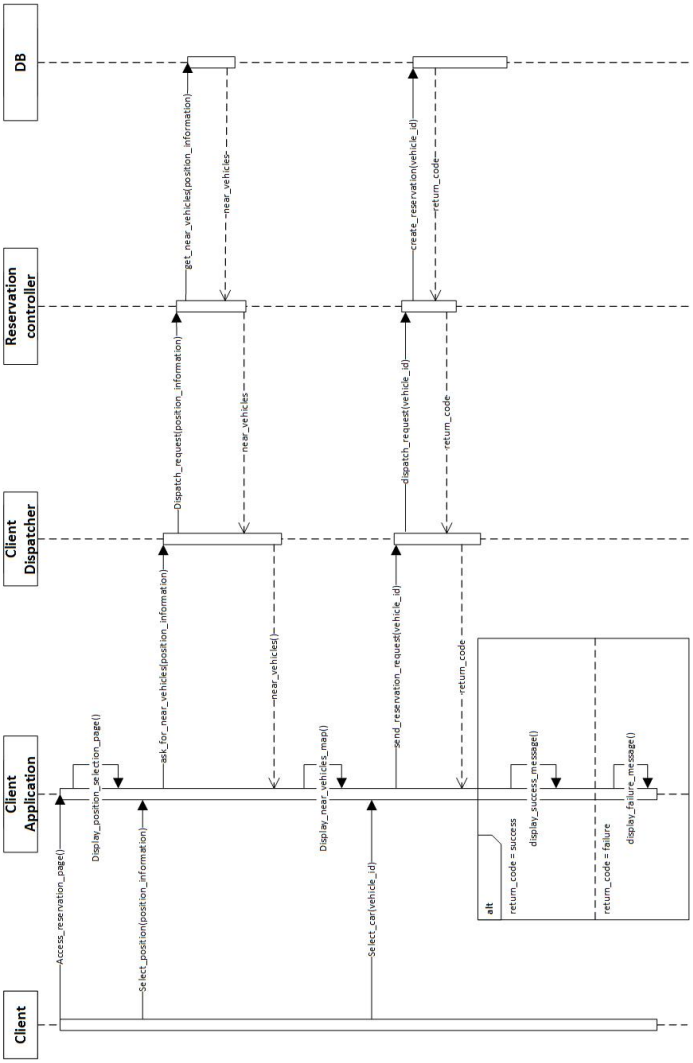


Figure 6: Customer reservation

When the client selects to navigate to the reservation page from the menu, the app will ask the client to select a search position and a search radius. When the client picks a position, the app will contact the Reservation Controller running on the server by means of the Client Dispatcher. The reservation Controller will then query the DB in order to obtain all the positions of the vehicles which are in the selcted search area. This information is passed to the client app, which uses it to create a map using Google Maps API. Finally, when the user selects a vehicle, a reservation request is sent to the Reservation Controller that will create a new reservation and will change the vehicle status to reserved (starting a 1 hour timer if the reservartion is successfull). Finallly, the client app will show a message indicating the outcome of the operation.

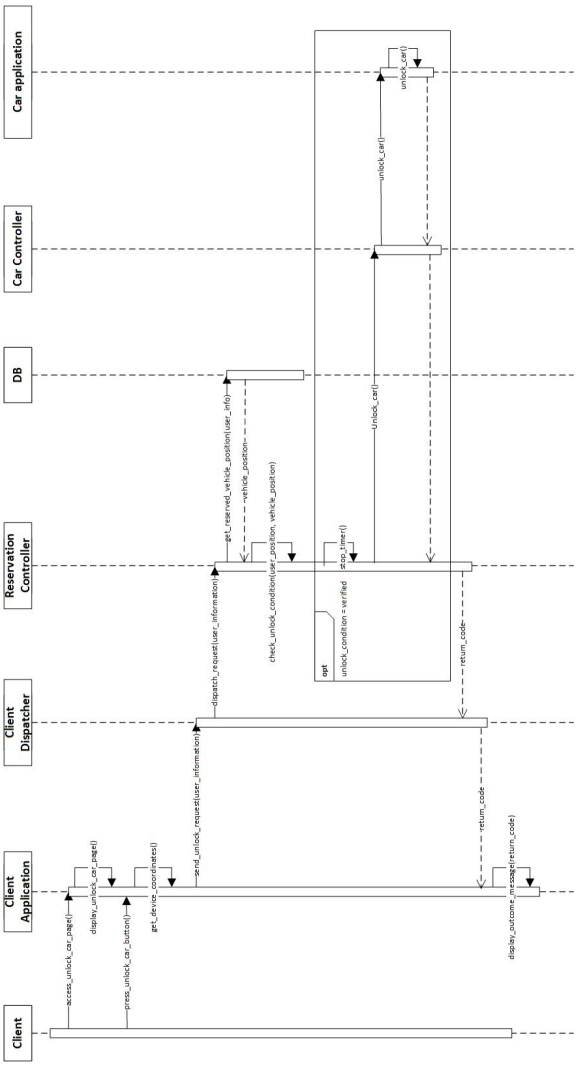


Figure 7: Unlock car

When the client selects to navigate to the unlock car page, the Client Application will display a page containing an unlock car button. If the user presses this button, the client will get the current device position and send a request to unlock the reserved car to the Car Controller using the Client Dispatcher. When the Car Controller receives this request, it will query the DB to ask for the car that has been reserved by that particular user (if a reservation exists). If a reservation has been made, the Car Controller will compare the two sets of coordinates and check if the car can be unlocked. If this is the case, the Car Controller will update the status off the car in the DB and wiil finally unlock the car, sending a message to the Car Application. Finally, the Client Application will display a message outlining the outcome of the operation.

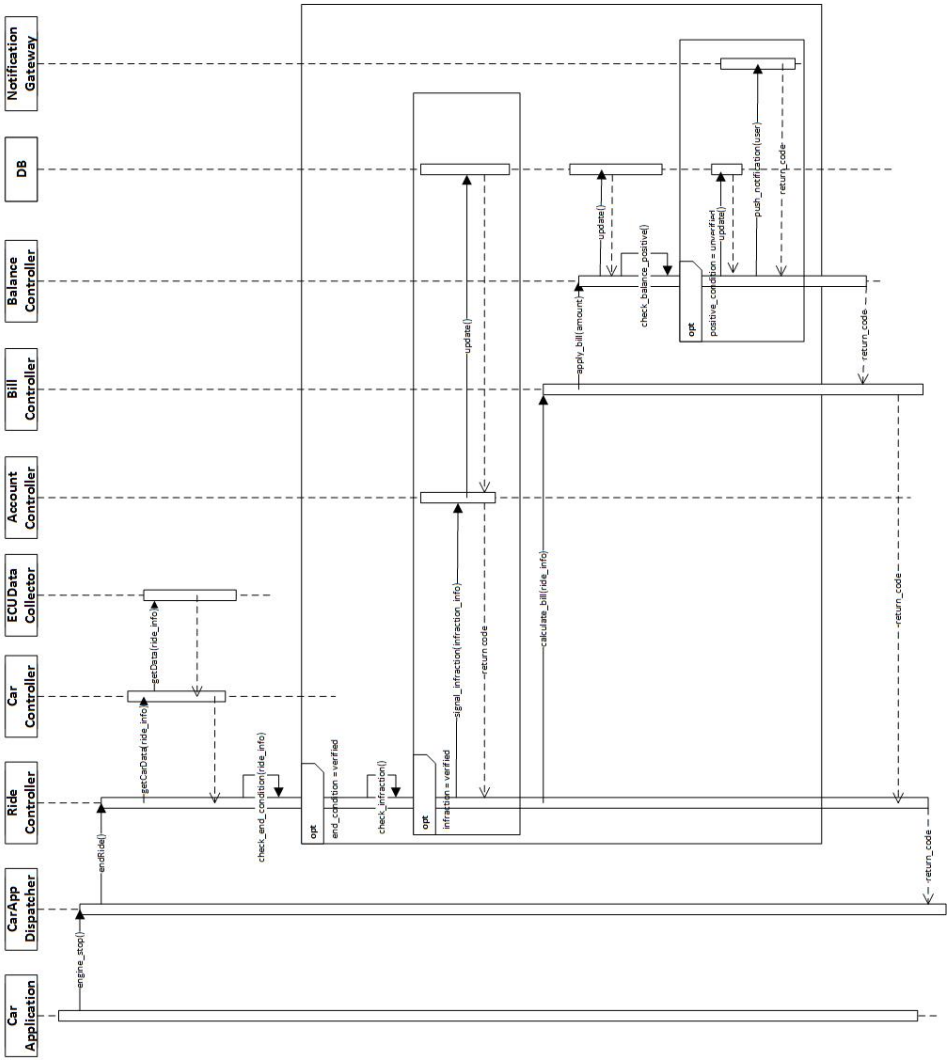


Figure 8: End of the ride

When engine stops, the on-board car application sends a notification to the server throught the CarAppDispatcher. The RideController will collect all the information regarding the ride using the CarController. If the ride has actually ended (i.e. the car has been parked in a safe area or has been abandoned) the RideController will delegate the calculation of the bill to BillController. When the BillController receives the request, it will calculate the import of the bill according ti company policy, check if any discount is applicable than delegate the payment to BalanceController. This component will query the database to get user's current balance and update it. If after the update the balance is negative, user's status is updated to debtor and a notification is sent to her.

2.4.2 State diagrams

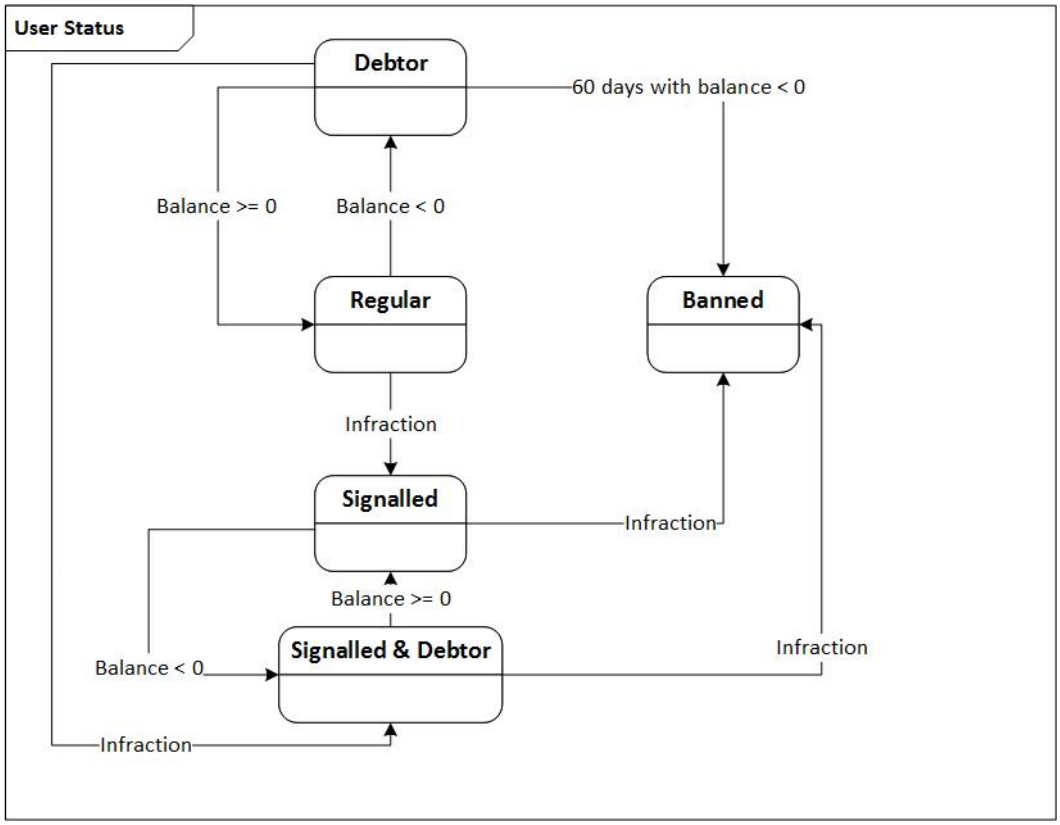


Figure 9: User status

2.5 Component interfaces

Registration Controller

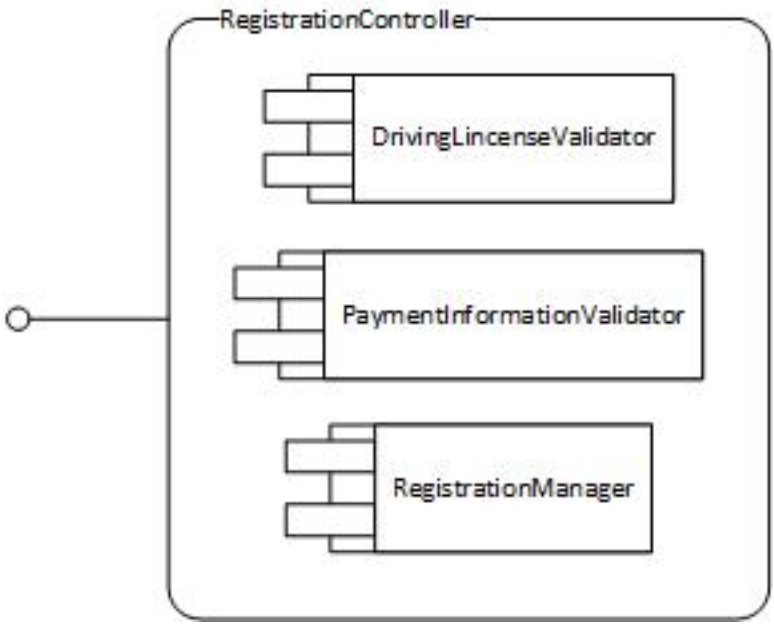


Figure 10: Registration Controller

INTERFACE:
-**register(user-info)** checks if the informations are valid, then the user is added to the database
-**InvalidArgumentException** this exception is raised if the information are incorrect
USES:
-**databaseController**

Account Controller

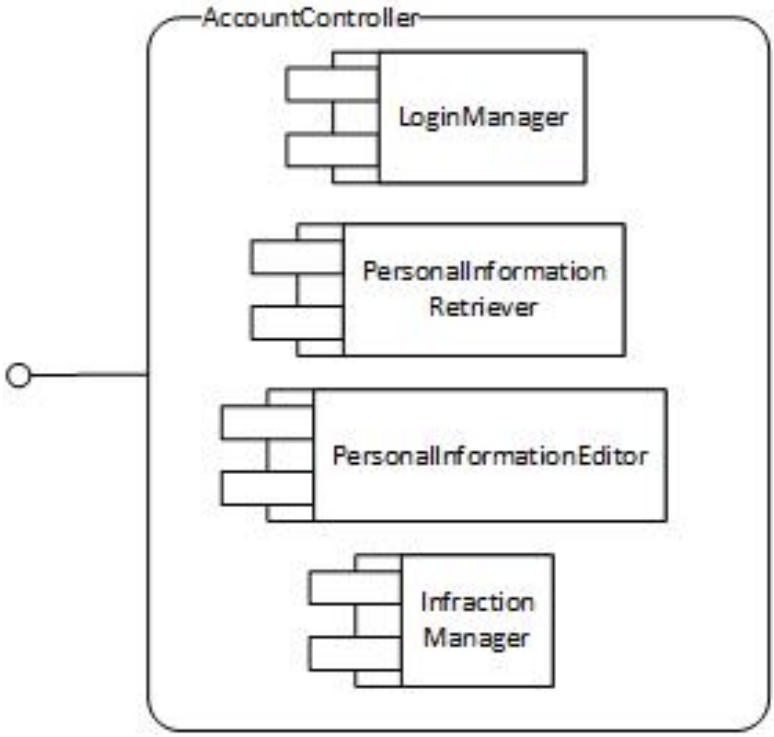


Figure 11: Account Controller

INTERFACE:
-**login(username, password):** enables users to log in
-**get-personal-info(user-id):** gets user's personal information from the database
-**edit-personal-info(user-info, user-id):** updates user's personal information in the database
-**signal-infraction(user-id, infraction-info):** changes user's status to signalled in the database
USES:
-**databaseController**

Reservation Controller

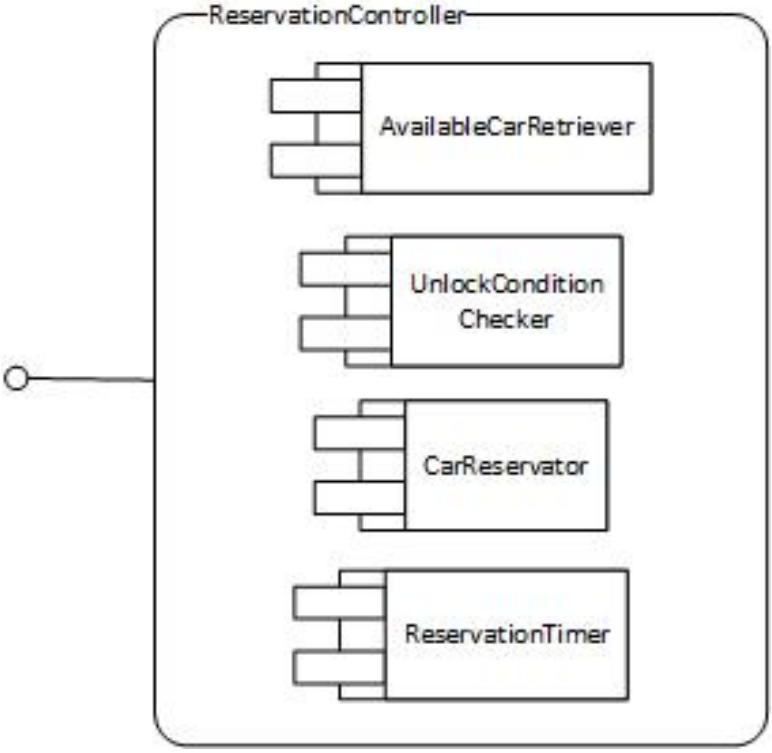


Figure 12: Reservation Controller

INTERFACE:
-**get-near-vehicles(position-information):** gets from the database the cars available near the selected position
-**create-reservation(vehicle-id, user-info):** creates a new reservation in the database; the selected car won't be available anymore
-**UnavailableCarException:** this exception is raised if the selected car is no longer available (another user has reserved it while he was looking at the map)
- **unlock(vehicle-id, user-info):** checks if the user is near the car he reserved: if so, interrupts the resertvation timer then delegates the opening to CarController
USES:
-**databaseController**

Balance Controller

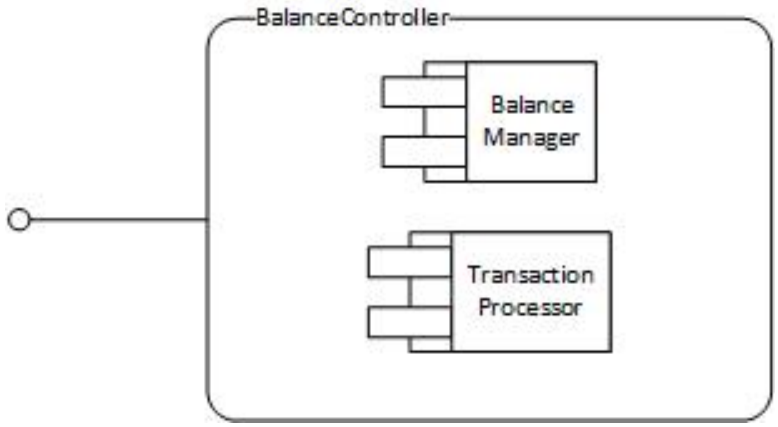


Figure 13: Balance Controller

INTERFACE:
-**deposit(payment-method, amount, user):** adds to user's balance the indicated amount of money
-**TransactionException:** this exception is raised if the transaction fails
-**pay(amount, user):** subtracts the amount of a bill: if the balance becomes negative, a notification is sent to the user
USES:
-**databaseController**
-**paymentGateway**
-**notificationGateway**

Report Controller

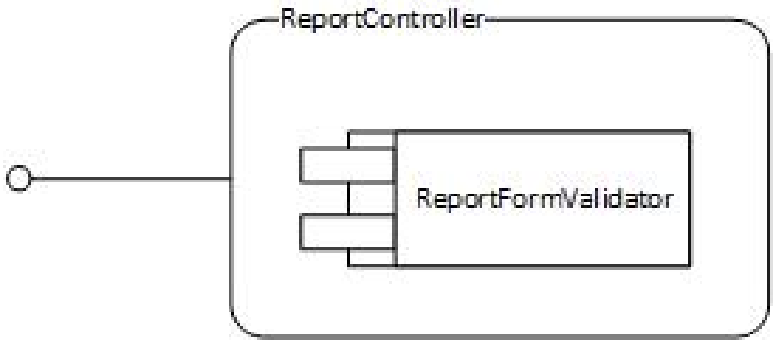


Figure 14: Report Controller

INTERFACE:
-report-damage(vehicle-plate, report): adds the car which the report is about in the assistance list
USES:
-assistanceController

Safe Area Controller

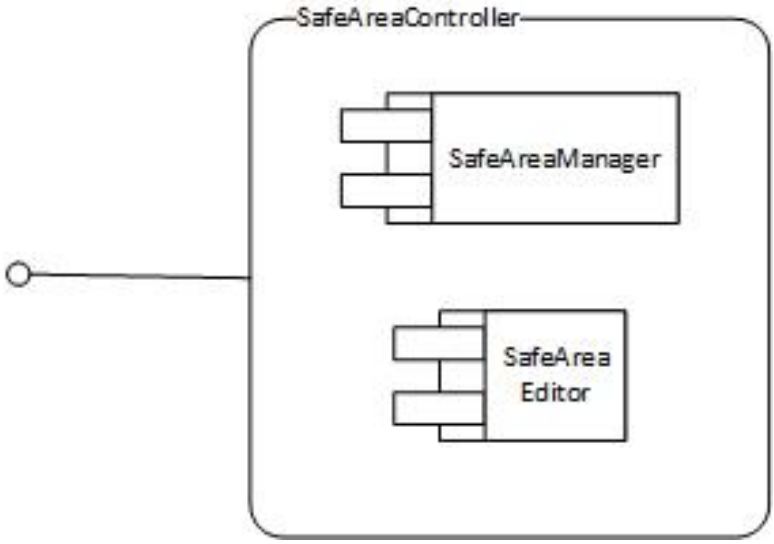


Figure 15: Safe Area Controller

INTERFACE:
-**get-safe-areas():** queries the database to get the set of safe areas already defined
-**create-safe-area(boundaries):** updates the database by adding a new safe area
-**OverlapException:** this exception is raised if a newly defined safe area overlaps an already defined one
USES:
-**databaseController**

Assistance Controller

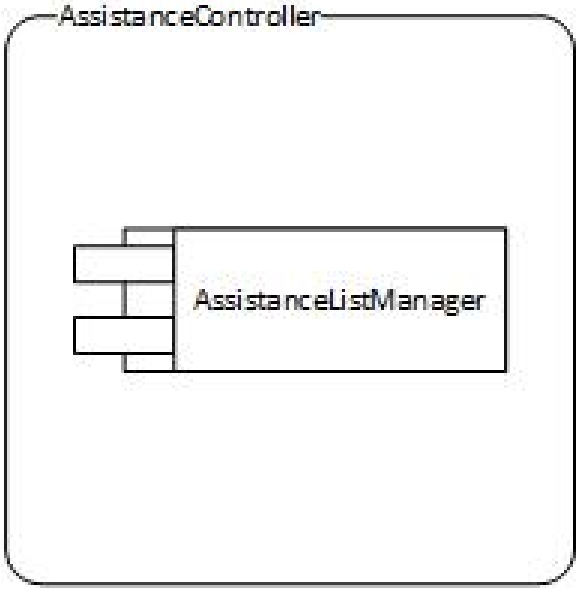


Figure 16: Assistance Controller

INTERFACE:
-**add(vehicle-id):** adds the vehicle to the assistance list; it will no longer be available to reservation
-**take-in-charge(vehicle-id, employee-id):** removes the vehicle from the assitance list as an employee will take care of the car
-**NotNeedyVehicleException:** this exception is raised if the vehicle specified is not in the list (i.e. some other employee has taken it in charge while he was choosing what to do)
-**solved(vehicle-id):** updates the database to set the vehicle as available as the issue was solved
USES:
-**databaseController**

Car Controller

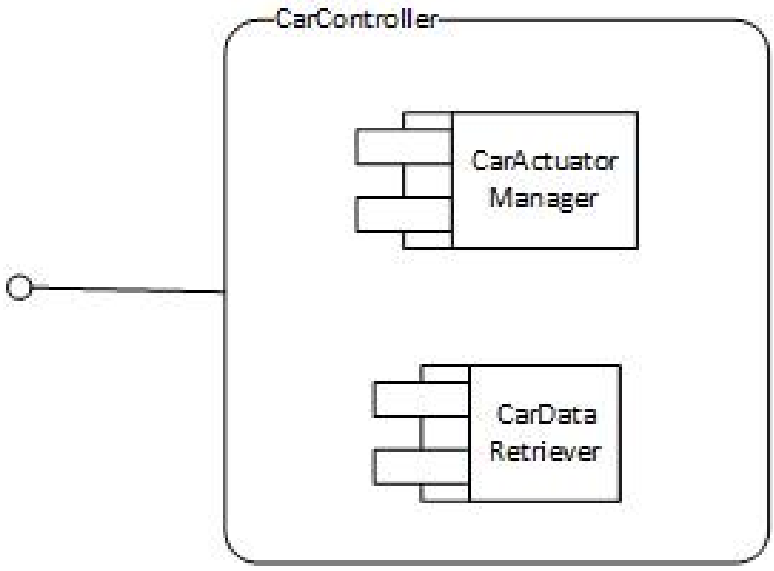


Figure 17: Car Controller

INTERFACE:
-**unlock-car(vehicle-id)**: forwards the unlock command to the car application
-**TooFarException:** this exception is raised if the user is not close to the car
-**CannotUnlockException:** this exception is raised if the opening fails
-**get-car-data(data-type):** dialogs with on-board car application to get the specified data, and updates the database
USES:
-**ECUDataCollector**
-**carActuator**
-**databaseController**

Ride Controller

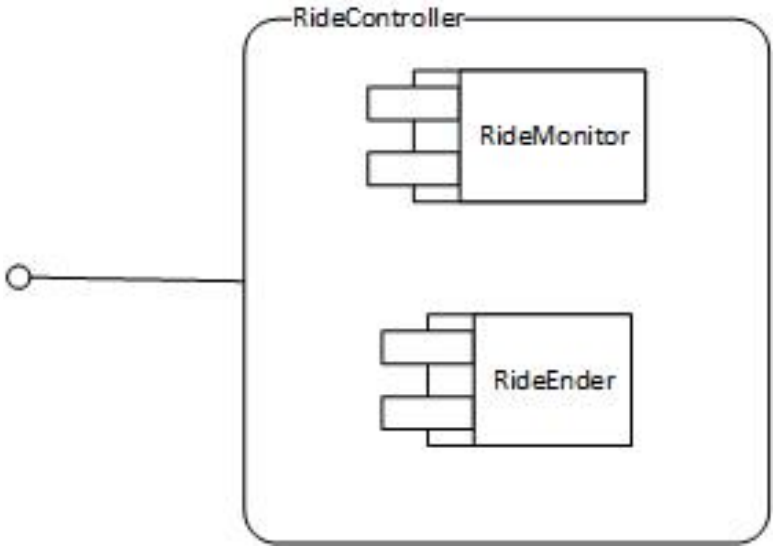


Figure 18: Ride Controller

INTERFACE:
-**start-ride():** starts the monitoring of car during the ride
-**end-ride(ride-info):** ends the ride, checks if any infraction has been committed (if any signal it to AccountController), collects all the data to calculate the bill and forwards them to the dedicated component
USES:
-**carController**
-**AccountController**
-**billController**

Bill Controller

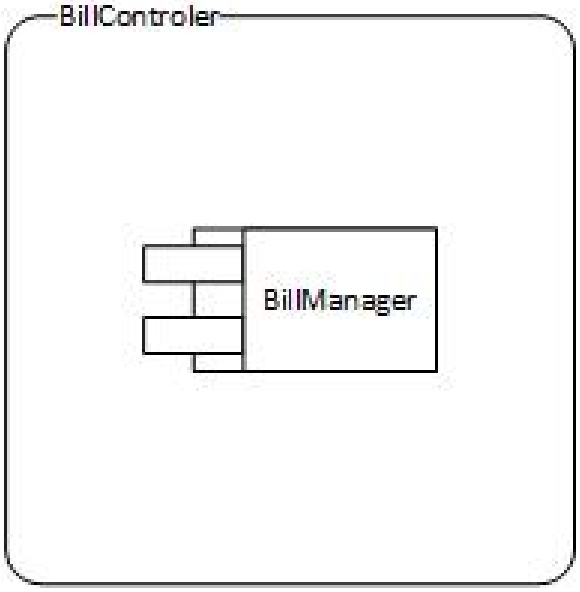


Figure 19: Bill Controller

INTERFACE:
-**calculate-bill(ride-info):** calculates the amount of the bill, than forwards it to the component dedicated to its application
USES:
-**balanceController**

Fault Controller

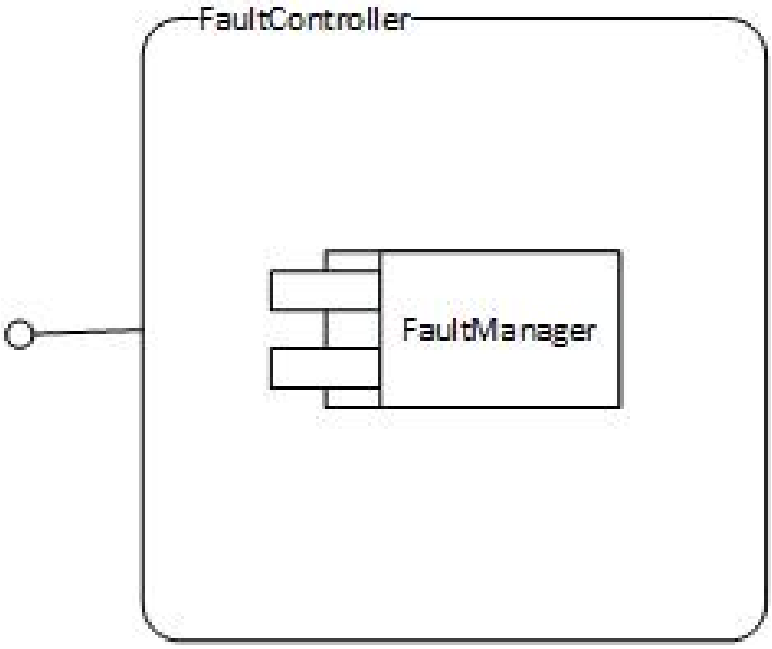


Figure 20: Fault Controller

INTERFACE:
-**notify-fault(fault-info):** receives the information regarding some issues detected by on-board application and adds the car to the assistance list
USES:
-**assistanceController**

Fleet Controller

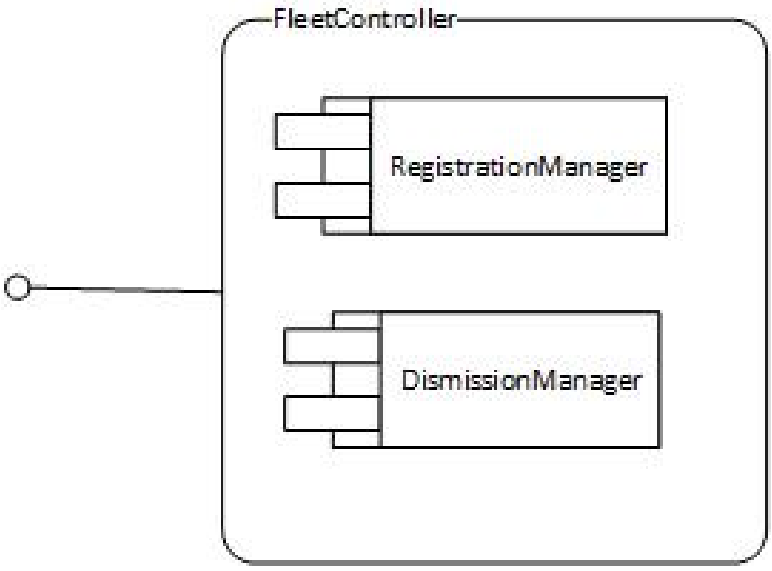


Figure 21: Fleet Controller

INTERFACE:
-**register-car(car-info):** adds a new car to the fleet
-**AlreadyRegisteredException:** this exception is raised if the car is already registered in the fleet
-**dismiss-car(car-info):** removes a car from the fleet
USES:
-**databaseController**

Car Application

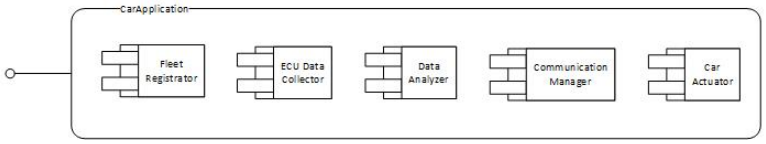


Figure 22: Car Application

INTERFACE:
-**get-data(data-type)** gets the required data from the ECU
-**unlock()** unlocks the car via its actuators
USES:
-**carAppDispatcher**

Payment Gateway

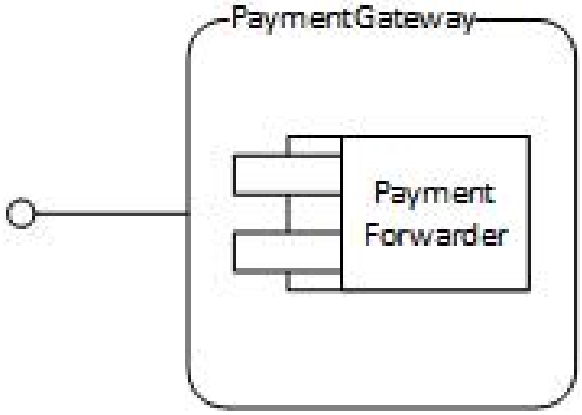


Figure 23: Payment Gateway

INTERFACE:
-**forward-payment(payment-info)** forwards the payment request to the selected payment provider after a deposit has been requested
USES:
-**externalPaymentProvider**

Notification Gateway

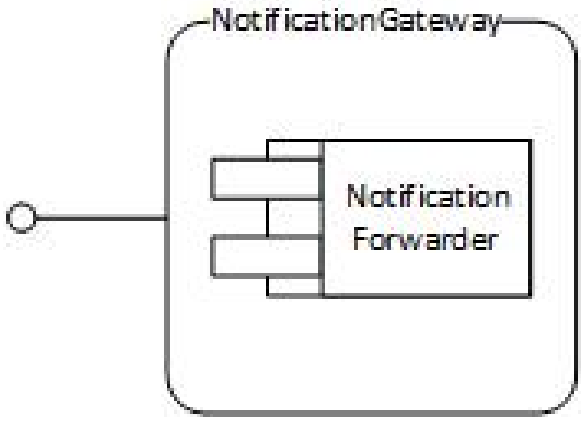


Figure 24: Notification Gateway

INTERFACE:
-**notify(user, info):** notifies the user after a relevant event has occurred
USES:
-**none**

Client Dispatcher

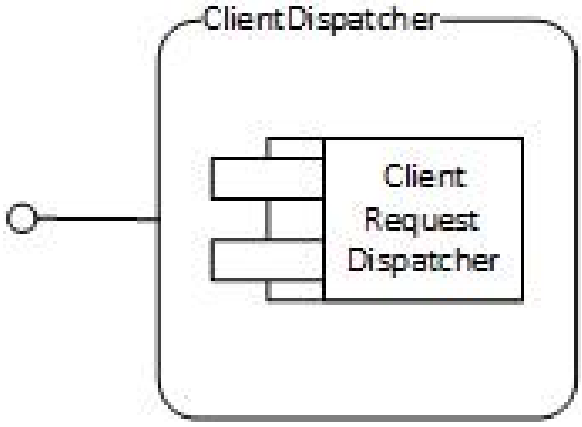


Figure 25: Client Dispatcher

INTERFACE:
-**dispatch-request(request-info):** forwards client's request to the dedicated component
USES:
-**registrationController**
-**accountController**
-**reservationController**
-**balanceController**
-**reportController**

Employee Dispatcher

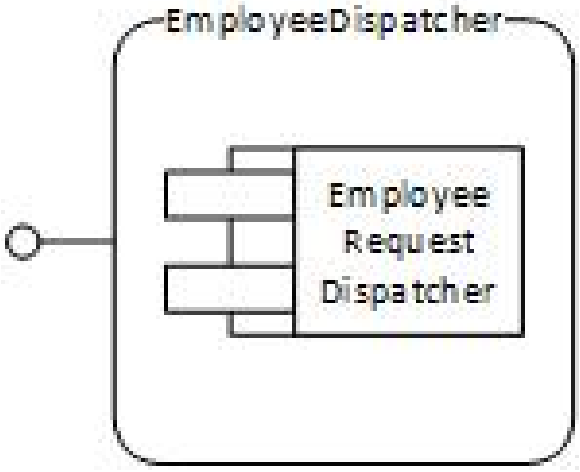


Figure 26: Employee Dispatcher

INTERFACE:
-dispatch-request(request-info): forwards employee's request to the dedicated component
USES:
-assistanceController
-safeAreaController

CarApp Dispatcher

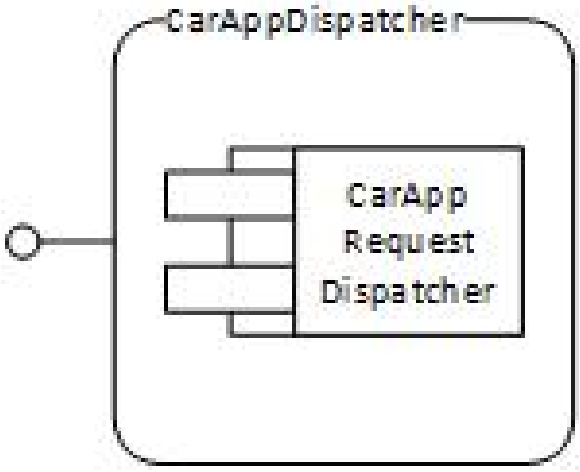


Figure 27: CarApp Dispatcher

INTERFACE:
-**dispatch-request(request-info):** forwards request of car application to the dedicated component
USES:
-**faultController**
-**rideController**
-**fleetController**

2.6 Selected architectural styles and patterns

2.6.1 General architecture

The overall architecture we will adopt to develop the PowerEnJoy platform will be a **three-tier** architecture with distributed logic. Using this architecture, we will be able to decouple the business logic layer from the presentation and the data layers. Our central system (where we can find the data layer and most of the business logic) needs to communicate both with the user and the vehicles. To achieve this communication, we will use:
-A **client-server** architectural style for the communication with the user. In this case, the client is the user device (pc or mobile device). The client side takes care of the presentation logic and will also contain some business logic (mainly the one used to communicate with the Google Maps service). We will however try to keep the client as thin as possible.
-An **event based paradigm** for the communication between the central system and the vehicles. Since the vehicles will need to notify state changes, the central system will register itself as a listener to all the vehicles and will act accordingly to these state changes. Moreover, the car must expose some methods to the central system (such as the one to unlock the car).

2.6.2 Design decisions

JEE: we will develop the system using JEE to develop a reliable system in a faster time. Security requirements will be more easily fulfilled using consolidated practices, as well as performance ones. The component-based style will also be reflected in the JEE Bean structure.
Google Maps API: the user device will interact directly with Google Maps services through Google Maps APIs. This will allow the device to display useful maps used by the user to access PowerEnjoy services.
Java Persistence API: we will use JPA for accessing, persisting, and managing our database, since JPA is already implemented in JEE.

2.6.3 Design patterns

Client-server: as aforementioned in the introduction of this chapter, we will make large use of the Client-Server design pattern, using it in the communication between the user device and the central system. To achieve this, the server will expose some methods (using a REST architecture) that will be used by the client to access the PowerEnJoy platform services.
Component-Based: the server side of the system is structured following a component-based style. This decision gives the opportunity to divide the business logic into different parts, enabling developers to focus their work on single components. The product will also be scalable, as adding new functionalities

would result in developing new components, leaving the existing structure unaltered.

Adapter: we will be using many interfaces between the components of our central system. In order to implement these interfaces, we will need to adopt the adapter design pattern.

3 Algorithm design

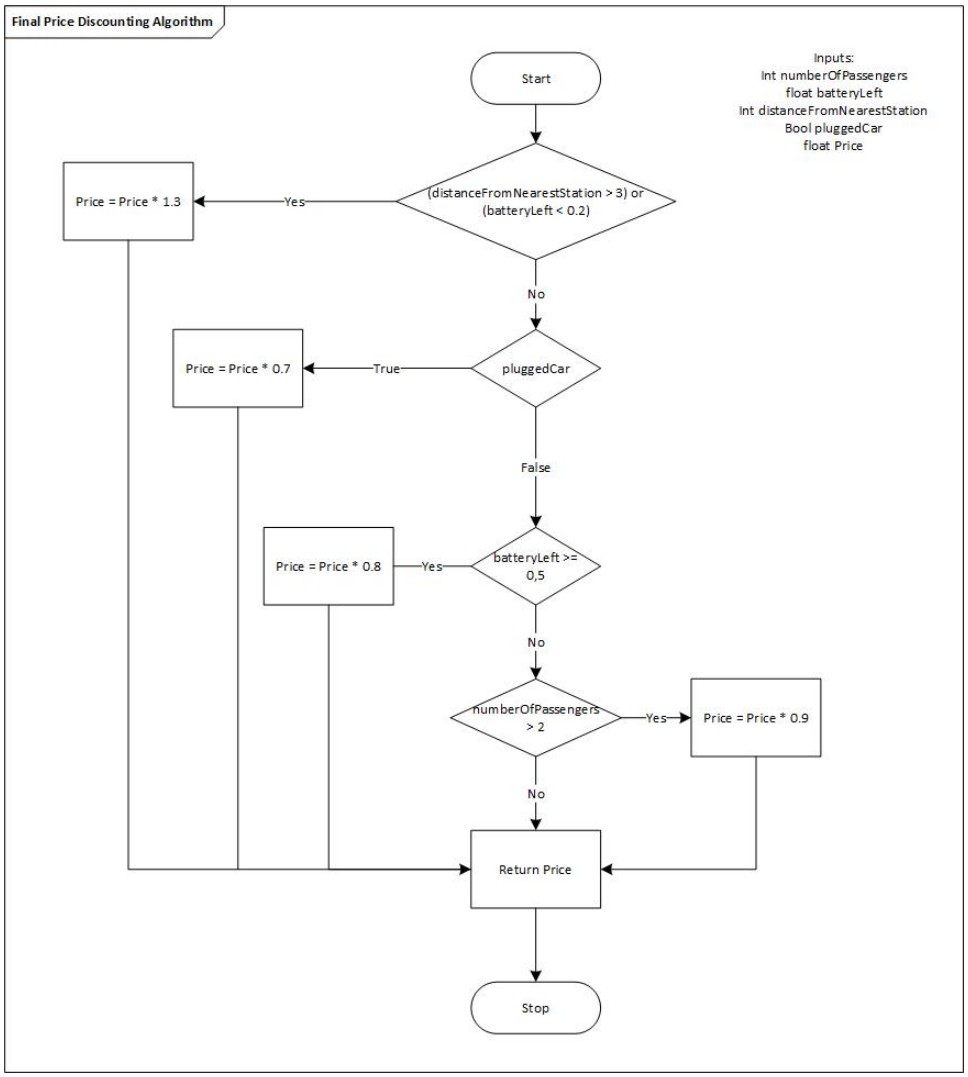


Figure 28: Pricing Algorithm

4 User interface design

4.1 UX Diagrams

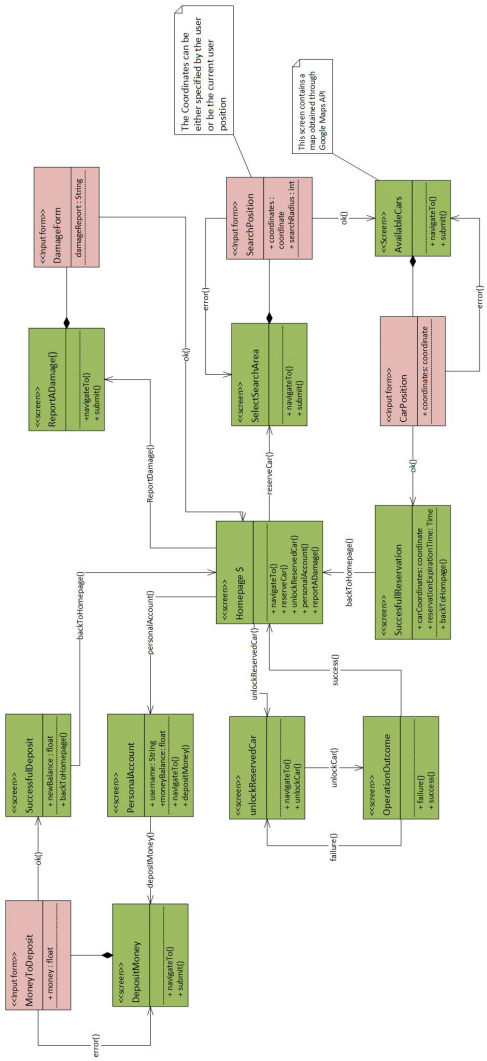


Figure 29: Client UX Diagram

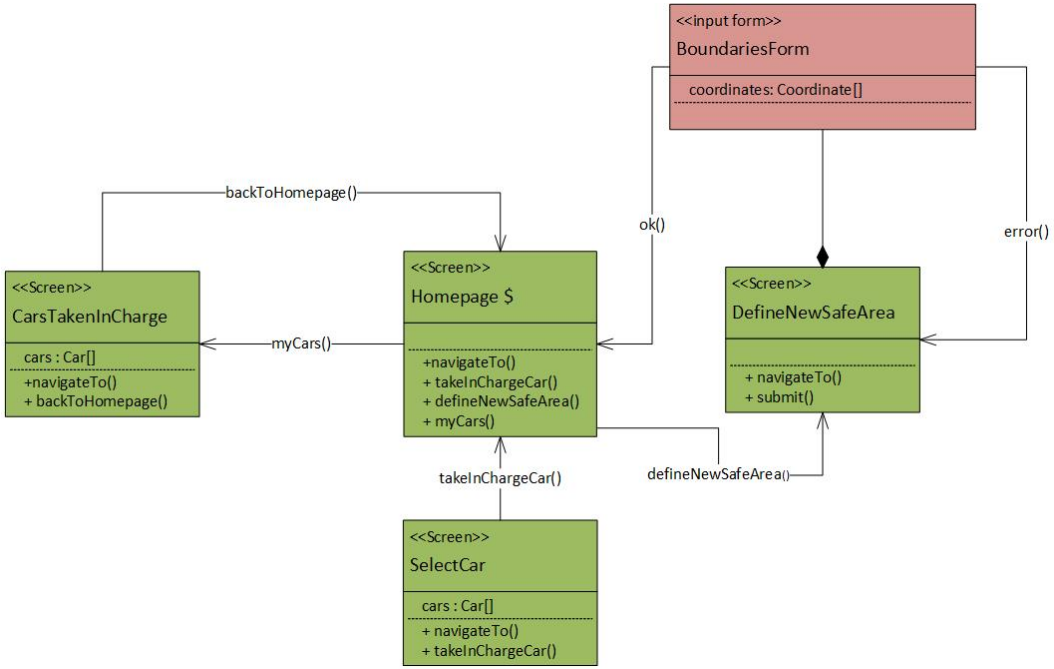


Figure 30: Employee UX Diagram

4.2 Mock-Ups

4.2.1 Employee mock-ups

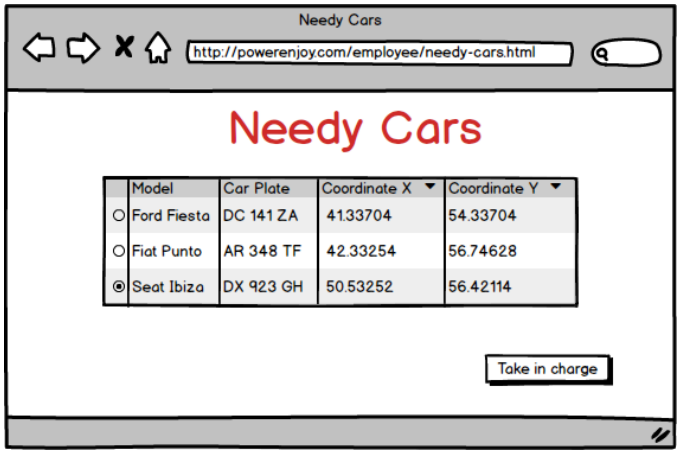


Figure 31: On this page the employee can check cars thhat need assistance and take in charge one of them

Figure 32: The employee can define new safe area by drawing them on the map

4.2.2 Customer mock-ups

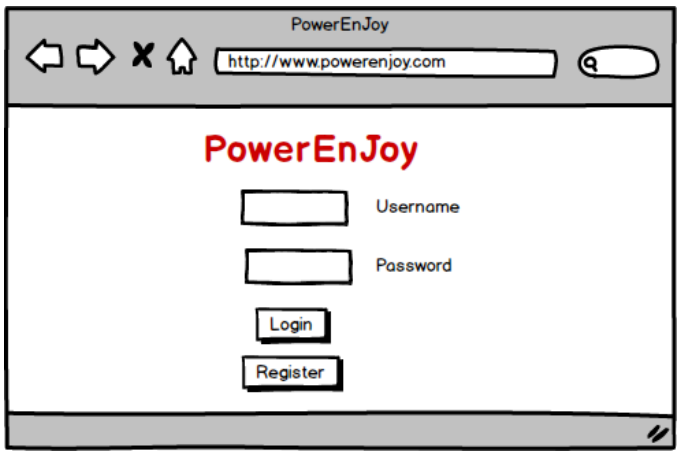


Figure 33: PowerEnjoy login screen

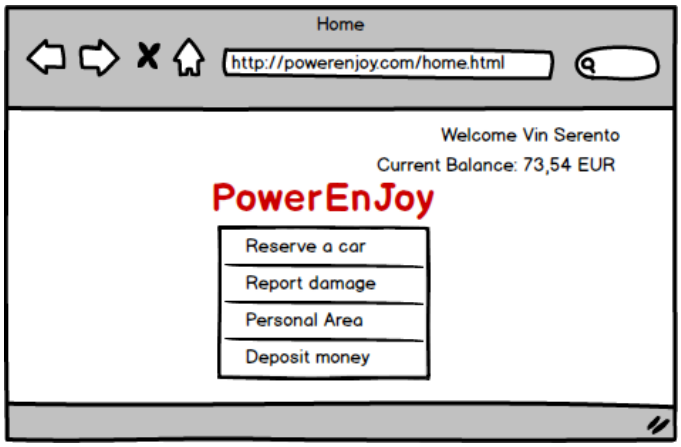


Figure 34: PowerEnjoy homepage

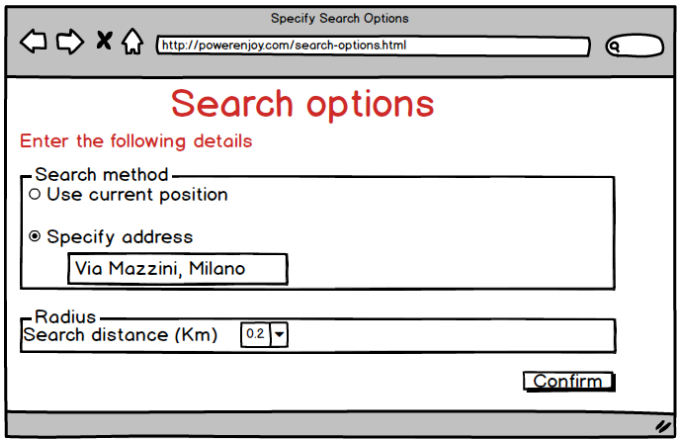


Figure 35: Here the customer can specify the search options for his reservation

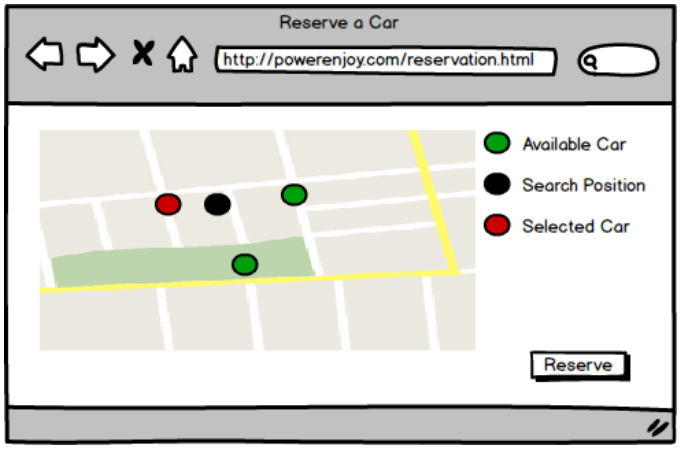


Figure 36: This map shows available cars and lets the customer choose one to reserve

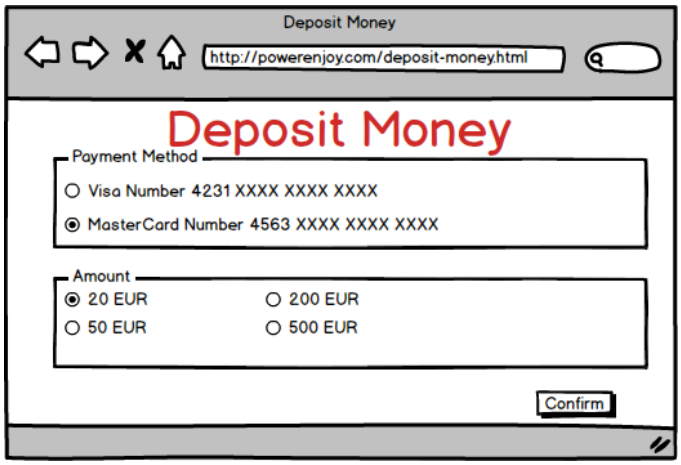


Figure 37: Here, the customer can deposit money to his balance

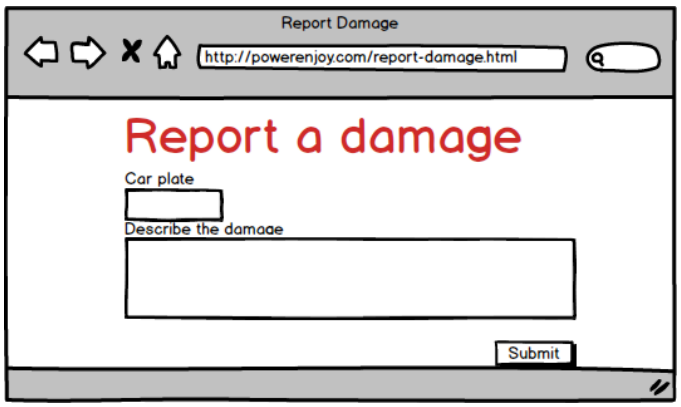


Figure 38: On this screen, the user can report a damage that he found on his reserved car

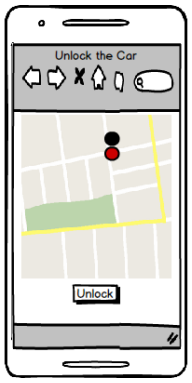


Figure 39: Unlock screen. The map shows the customer and the reserved car positions

5 Requirements traceability

Table 2: Requirement Traceability Matrix

GOAL	REQUIREMENT	COMPONENT
G1	R.1.1	Account Controller
G1	R.1.2	Client Application
G1	R.1.3	Client Application
G1	R.1.4	Reservation Controller
G1	R.1.5	Reservation Controller
G2	R.2.1	Car Controller
G2	R.2.2	Car Controller
G2	R.2.3	Car Controller
G3	R.3.1	Balance Controller
G4	R.4.1	Data Analyzer
G4	R.4.2	Data Analyzer
G4	R.4.3	Assistance Controller
G4	R.4.4	Assistance Controller
G4	R.4.5	Assistance Controller
G4	R.4.7	Assistance Controller
G4	R.4.7	Fleet Controller
G4	R.4.8	Fleet Controller
G5	R.5.1	SafeArea Controller
G6	R.6.1	Balance Controller
G6	R.6.2	Account Controller
G6	R.6.3	Balance Controller
G6	R.6.4	Bill Calculator
G6	R.6.5	Bill Calculator
G6	R.6.6	Account Calculator
G6	R.6.7	Balance Controller

6 Effort spent

Total Work Time
Simone Brunitti:
28/11/2016 2 hours
30/11/2016 2 hours
03/12/2016 3 hours
04/12/2016 2 hours
05/12/2016 1.30 hours
07/12/2016 1.30 hours
08/12/2016 3 hours
10/12/2016 6 hours
11/12/2016 2 hours
Total time: 23 hours

Stefano Boriero:
28/11/2016 2 hours
30/11/2016 2 hours
03/12/2016 4 hours
04/12/2016 1.30 hours
05/12/2016 2 hours
07/12/2016 3 hours
08/12/2016 3 hours
10/12/2016 4 hours
11/12/2016 1.30 hours
Total time: 23 hours