Design Document

POLITECNICO
MILANO 1863

Version 1.2

Luca Santini	808710
Riccardo Remigio	874939

Index

1. Introduction

1.1 Purpose

The purpose of this document is to specify details about the architecture and design of the software that will be developed to meet the demands of PowerEnJoy.
This document will be useful to coordinate the work of all developers who are going to write the code, this will be their reference to follow.

1.2 Scope

The PowerEnJoy system will be developed to mainly satisfy two types of users:

- The clients of the service
- The technicians

What we're going to develop is a web application which can be accessed from PC or smartphone or other device that can connect to a browser. The user once logged in will be recognized as a customer or as a technician and as a result will have their own functionalities.
A customer can: visualize a map with the vehicles available, reserve a car and use it. During the rental, users can see the current charge. A user with good behavior will benefit from discounts.
The technicians can visualize all the vehicles and change manually their states. Then the functionalities for technicians will help them to work more efficiently and in a less stressful way.

1.3 Definition, Acronyms, Abbreviations

RASD: Requirements Analysis and Specifications Document

DD: Design Document

GPS: Global Positioning System

PC: Personal Computer

App: Software application

UX: User Experience

BCE: Boundary-Control-Entity

DB: Database

DBMS: Database Management System

API: Application Programming Interface, is a common way to communicate with other external system.

MVC: Model View Controller, it is an architectural software design pattern for implementing user interface

GUI: Graphical User Interface

1.4 Reference Documents

- RASD of PowerEnJoy produced before
- Assignments Document AA 2016-2017.pdf
- The IEEE Standard for Information Technology – System Design – Software Design Description

1.5 Document Structure

- **Introduction:** this section introduces the design document. It contains a justification of his utility and indications on which parts are covered in this document that are not covered by RASD
- **Architecture Design:** this section is divided into different parts:
    1. Overview: this sections explains the division in tiers of our application
    2. High level component view: this sections gives a global view of the components of the application and how they communicate
    3. Component view: this sections gives a more detailed view of the components of the applications
    4. Deployment view: this section shows the components that must be deployed to have the application running correctly
    5. Runtime view: sequence diagrams are represented in this section to show the course of the different tasks of our application
    6. Selected architectural styles and patterns: this section explains the architectural choices taken during the creation of the application
- **Algorithms Design:** this section describes the most critical parts via some algorithms.
- **User Interface Design:** this section presents mockups and user experience explained via UX and BCE diagrams.
- **Requirements Traceability:** this section aims to explain how the decisions taken in the RASD are linked to design elements.

2. Architectural Design

2.1 Overview

We will adopt a top down approach for the description of the architectural design of our system.
PowerEnJoy will be developed with a 3-tier architecture, using a Client-Server architectural style.

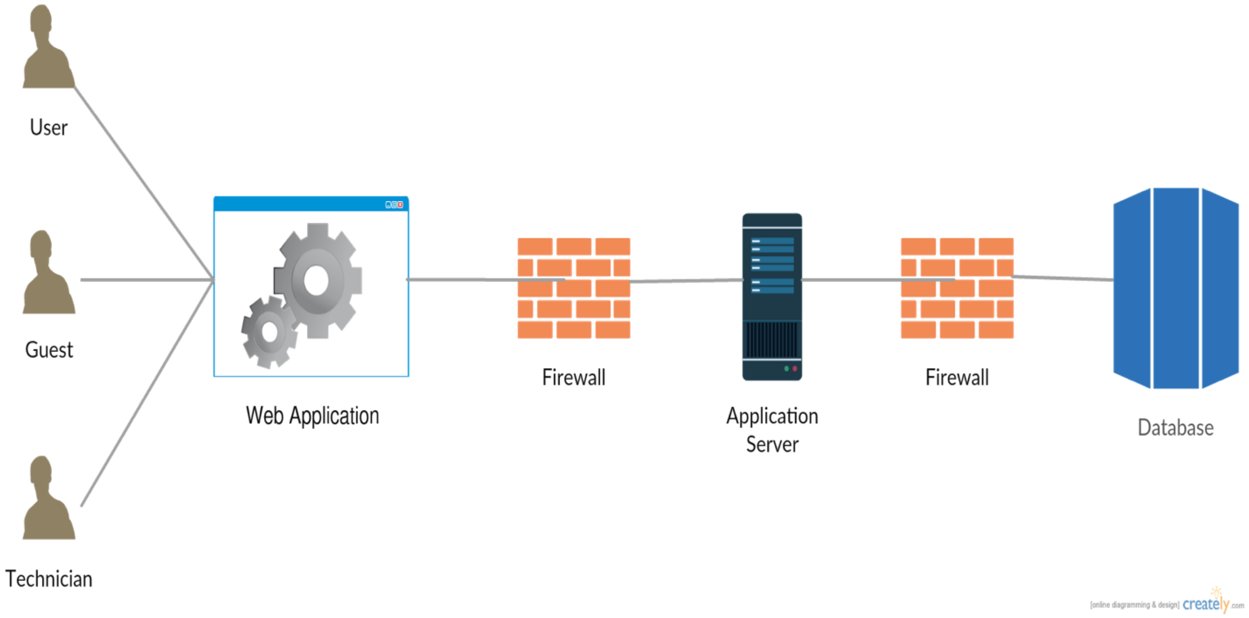


Figure 1. 3-Tier architecture

Users have access to a web site to interface with the server and take advantage of the functionalities dedicated to them. The server will receive requests from users and will have access to the database, so it can use it to save or read data if necessary.
To have a clearer outline of the high-level system we can see the functionalities divided into subsystems as shown in the figure below

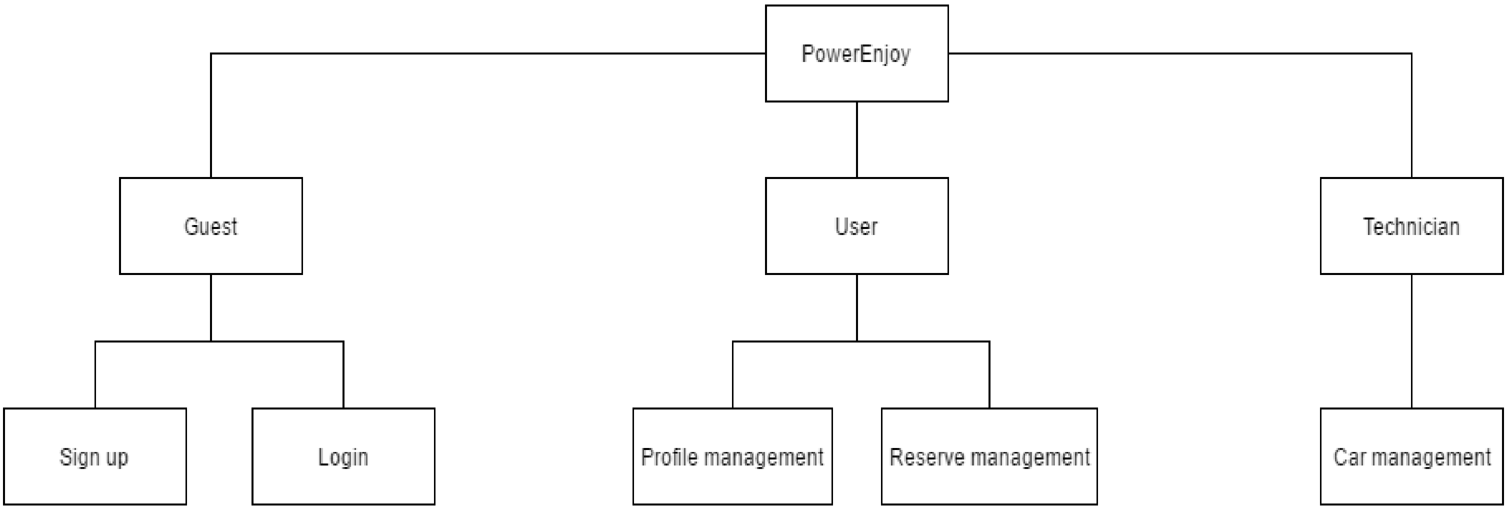


Figure 2. High-level functional system view

2.2 High-Level Component view

There are fundamentally four components in our system. The main component is the server, it is the engine of the system. Customers and technicians send their requests to the server which elaborates a response and sends it to them.
The server, to response to the requests, must be able to communicate with external systems and with the database. There are three external systems to help the server: there is a payment system that handles all requests regarding user fees. The server may request the external payment system to verify that the user data provided during registration are correct. At time of payment the server will send to the payment system the charging request and this will provide to do so, in the case where the payment system is not able to execute the charging, it will report it to the server and the user will be moved on the black list.
The vehicles are provided of an external service that monitors and communicates to the server all the necessary information that it needs, such as the location of vehicles and the battery info.
The last external service used by the server is a mapping service.

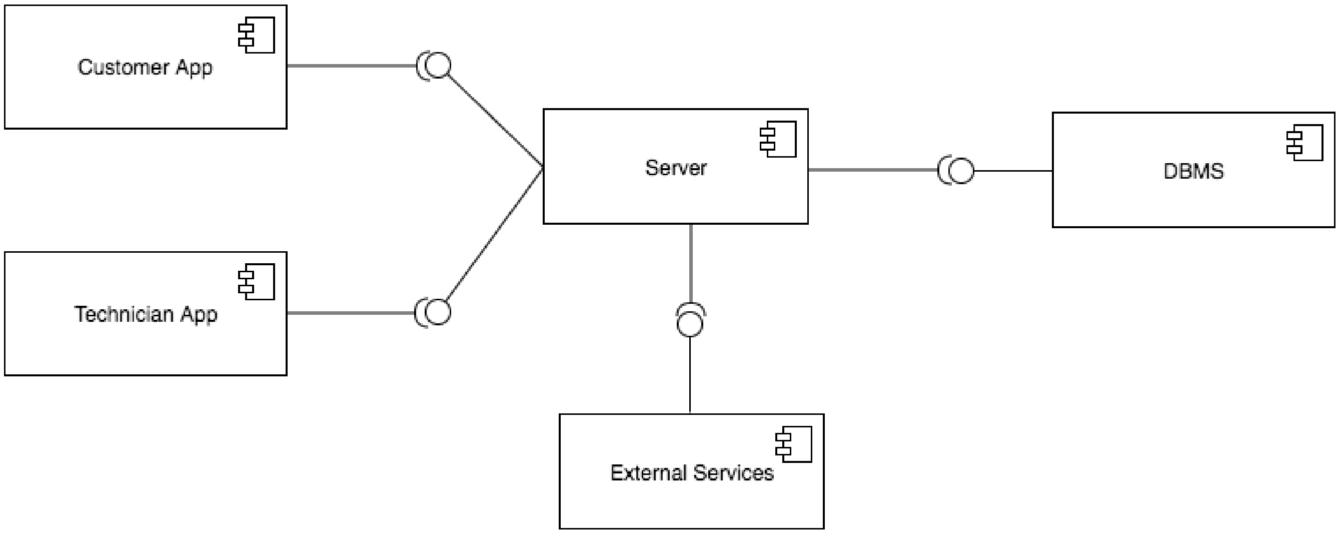


Figure 3. High-Level Component Diagram

2.3 Component View

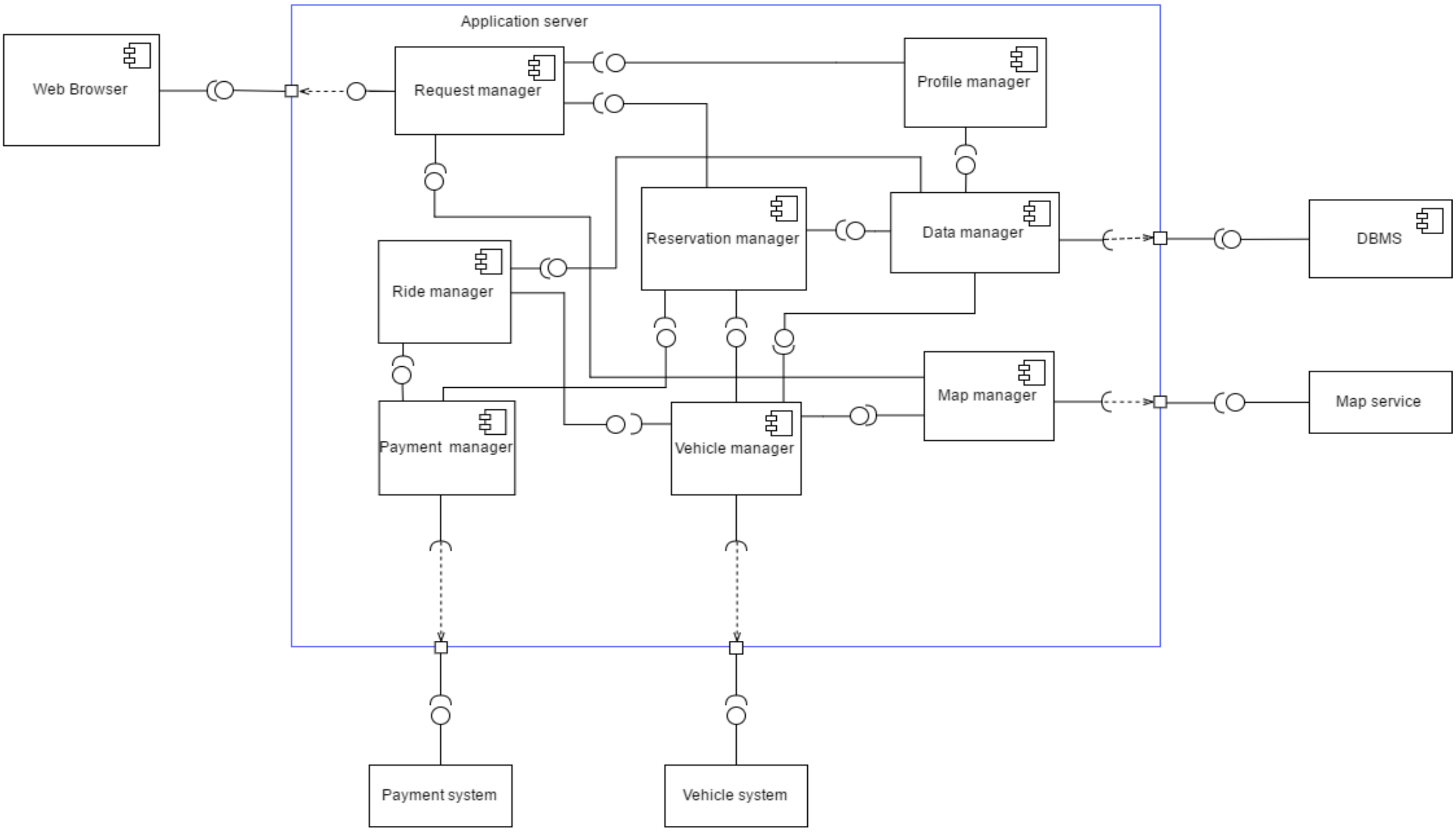


Figure 4. Component Diagram

The Component diagram summarizes in a more detailed way what we have already explained. It is also made explicit the internal division in the server of the application modules.

Request Manager is the component that handles any type of request received from the browser, and redirects them to the right component.
Profile Manager handles all types of operations on user profiles, often calling the data manager methods in the case that it must retrieve data from the database.
Data Manager is the component that interfaces directly with the DBMS. Each time another component needs access to the DBMS must pass for it.
Reservation Manager and Ride managers handle, respectively, the reservation and everything related to the user's entire ride.
Payment Manager is the component that interfaces directly with the external payment system.
Vehicle Manager is the component that communicates directly with the vehicles and is able to retrieve and to send information from/to them.
Map Manager is the component that handles the maps. With the help of the external service it can generate maps with the locations of the car and it can calculate the routes and positions.

Through this class diagram we explain how it will be organized the database to which the server will have access. In the database will be stored all the information of that we will need to keep track.

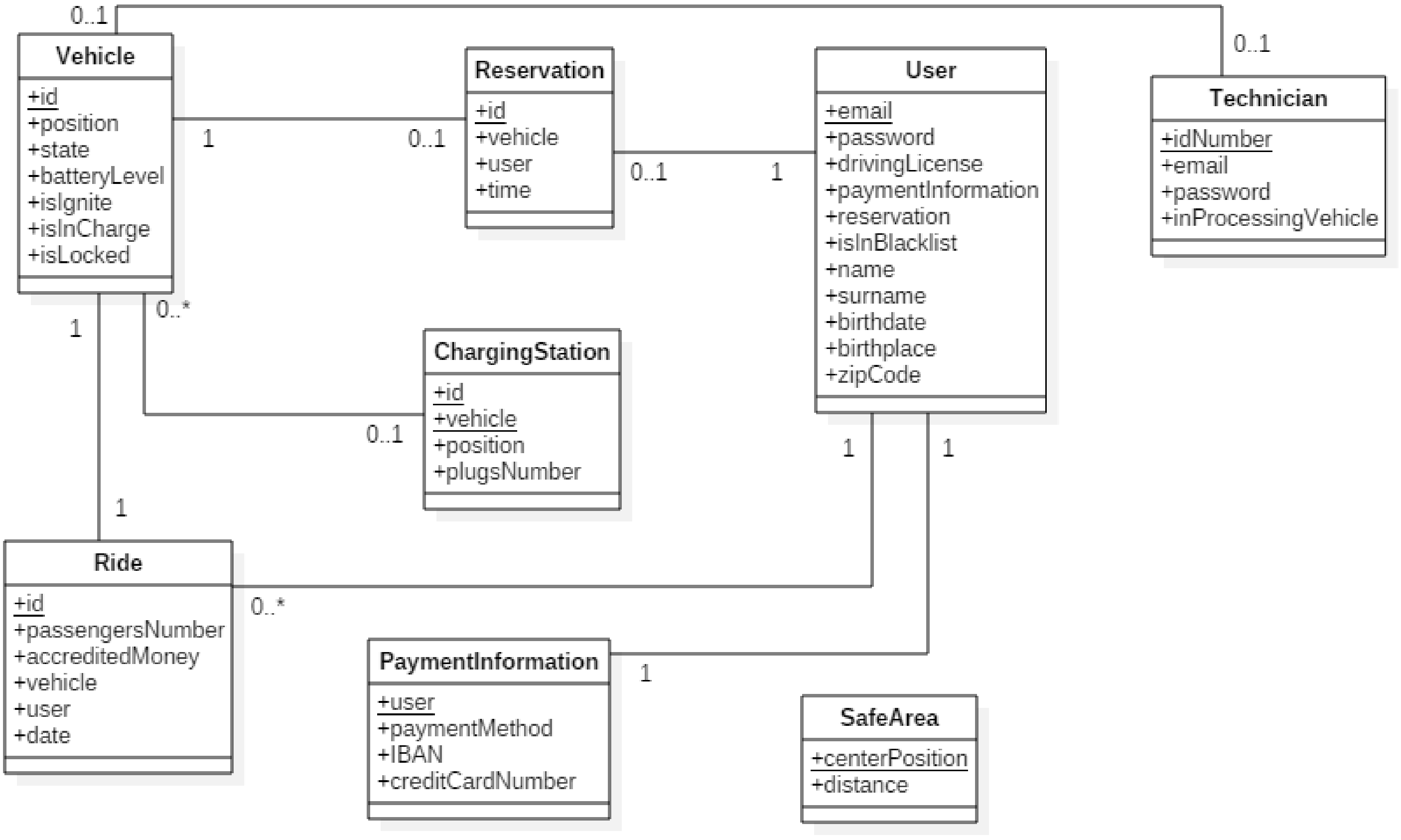


Figure 5. Class diagram representing database logic

2.4 Deployment View

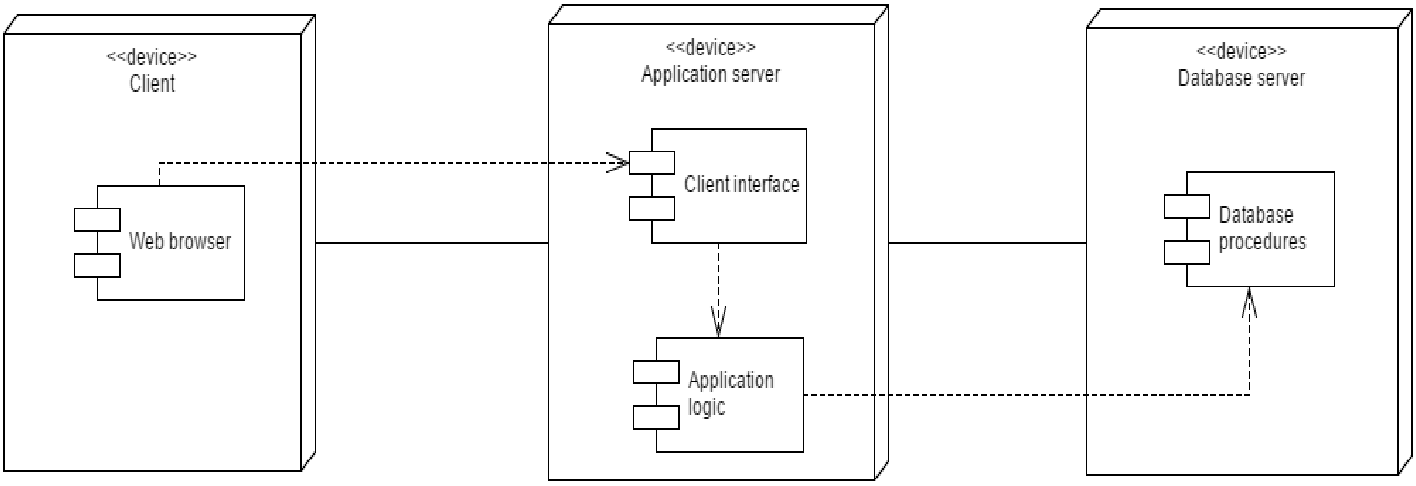
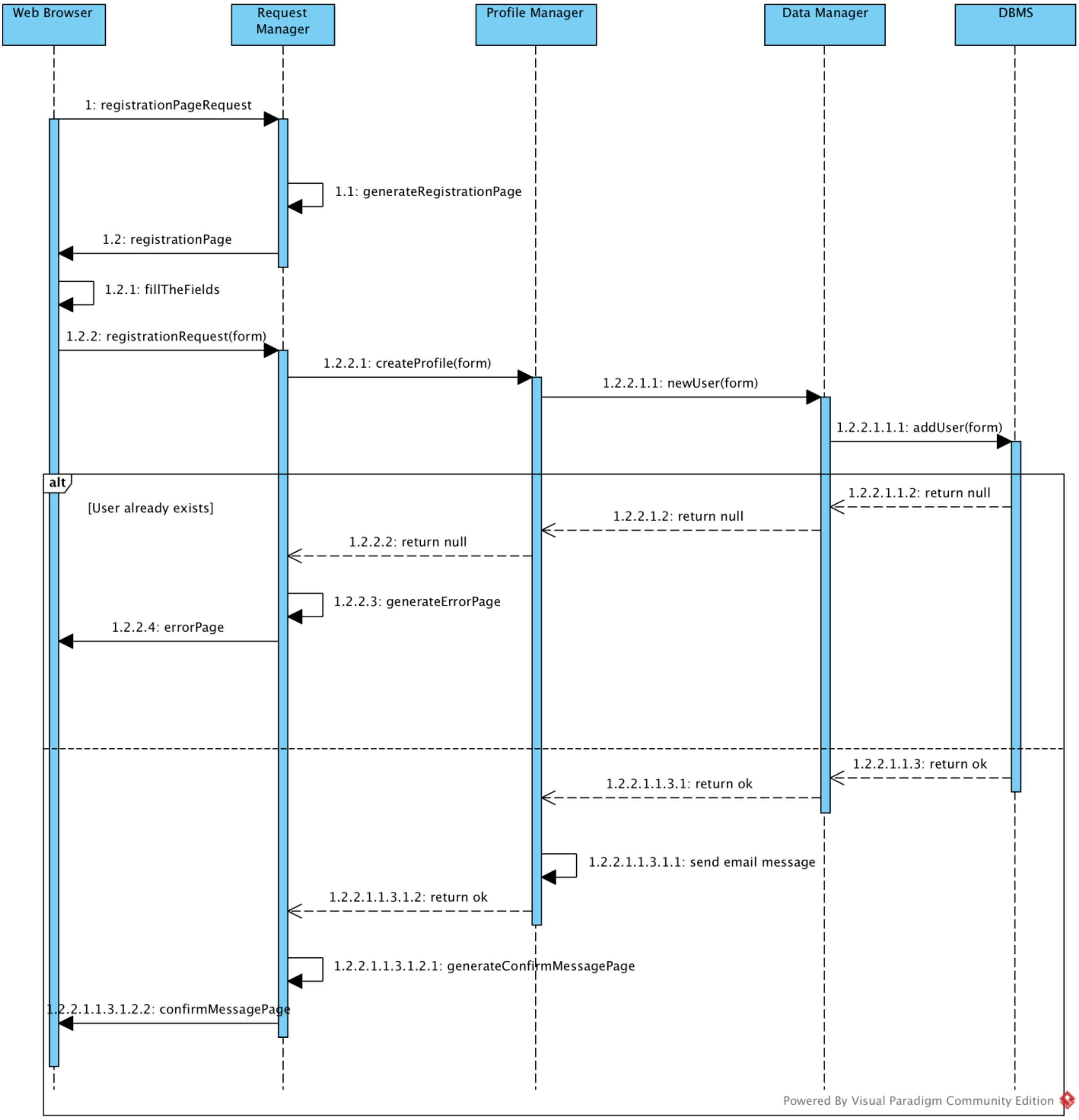


Figure 6. Deployment diagram of the system

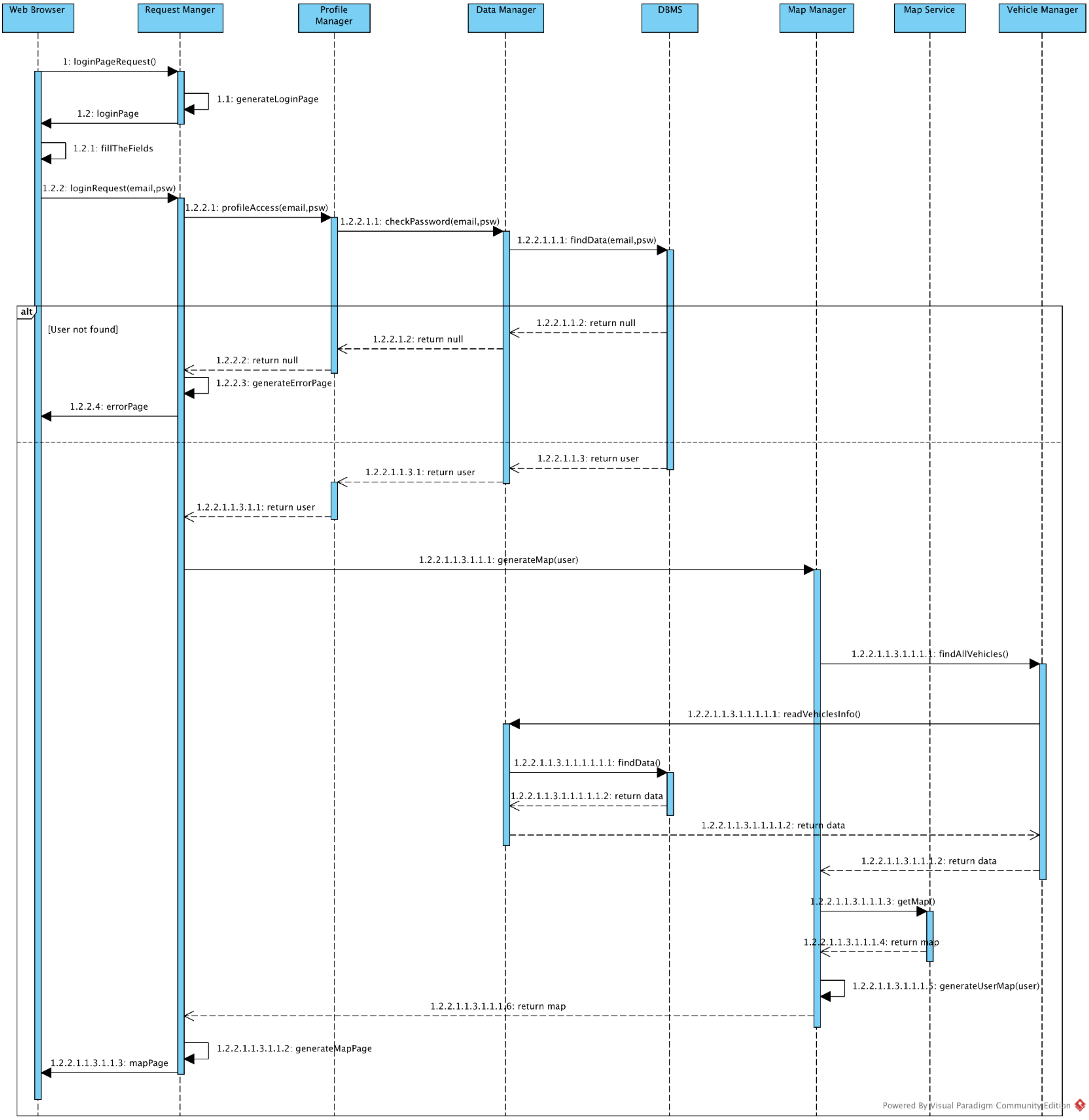
2.5 Runtime View

Now we describe the dynamic behavior of the system in most relevant case.

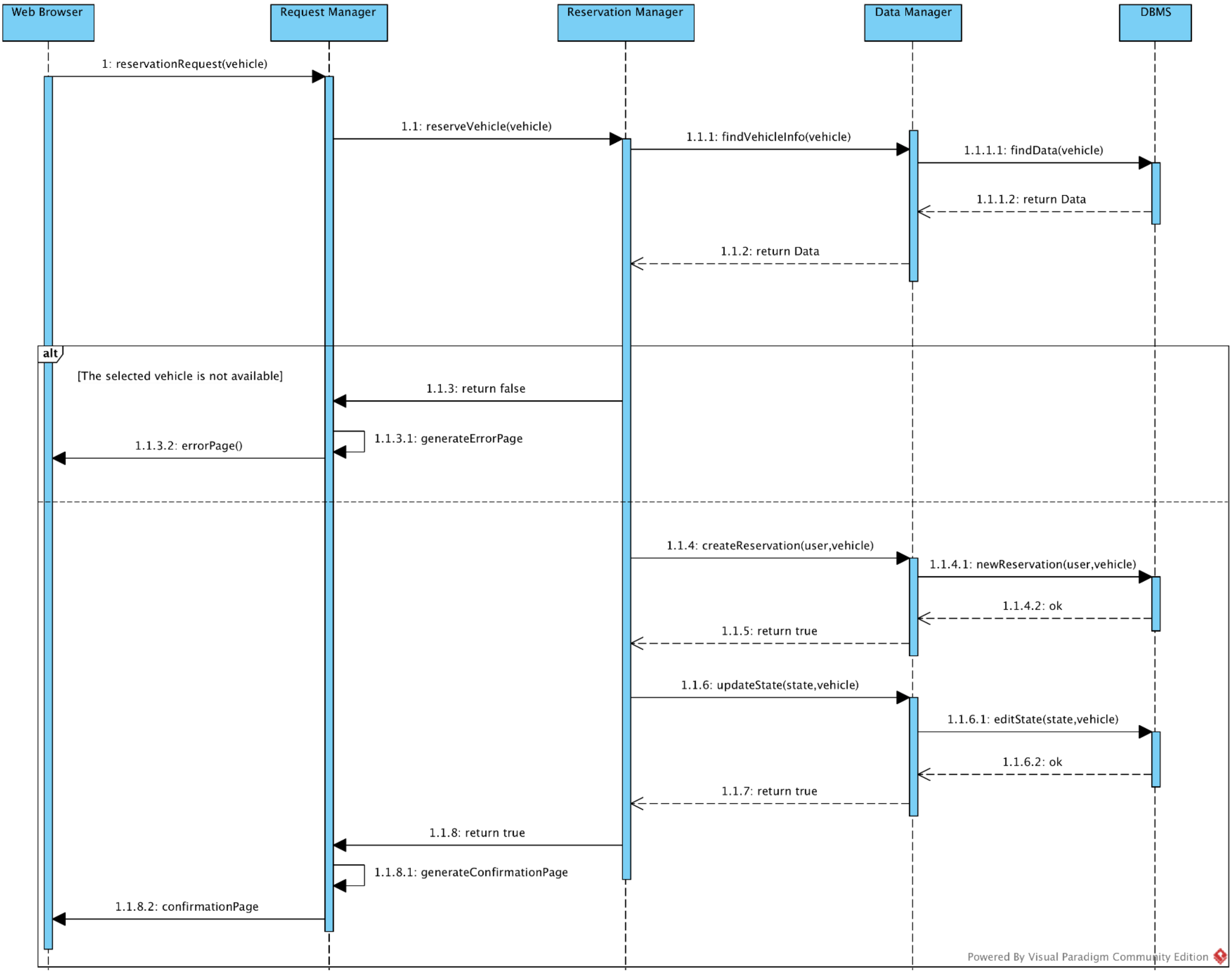
Registration


Web Browser
Request Manager
Profile Manager
Data Manager
DBMS
1: registrationPageRequest
1.1: generateRegistrationPage
1.2: registrationPage
1.2.1: fillTheFields
1.2.2: registrationRequest(form)
1.2.2.1: createProfile(form)
1.2.2.1.1: newUser(form)
1.2.2.1.1.1: addUser(form)
alt
[User already exists]
1.2.2.1.1.2: return null
1.2.2.1.2: return null
1.2.2.2: return null
1.2.2.3: generateErrorPage
1.2.2.4: errorPage
1.2.2.1.1.3: return ok
1.2.2.1.1.3.1: return ok
1.2.2.1.1.3.1.1: send email message
1.2.2.1.1.3.1.2: return ok
1.2.2.1.1.3.1.2.1: generateConfirmMessagePage
1.2.2.1.1.3.1.2.2: confirmMessagePage
Powered By Visual Paradigm Community Edition

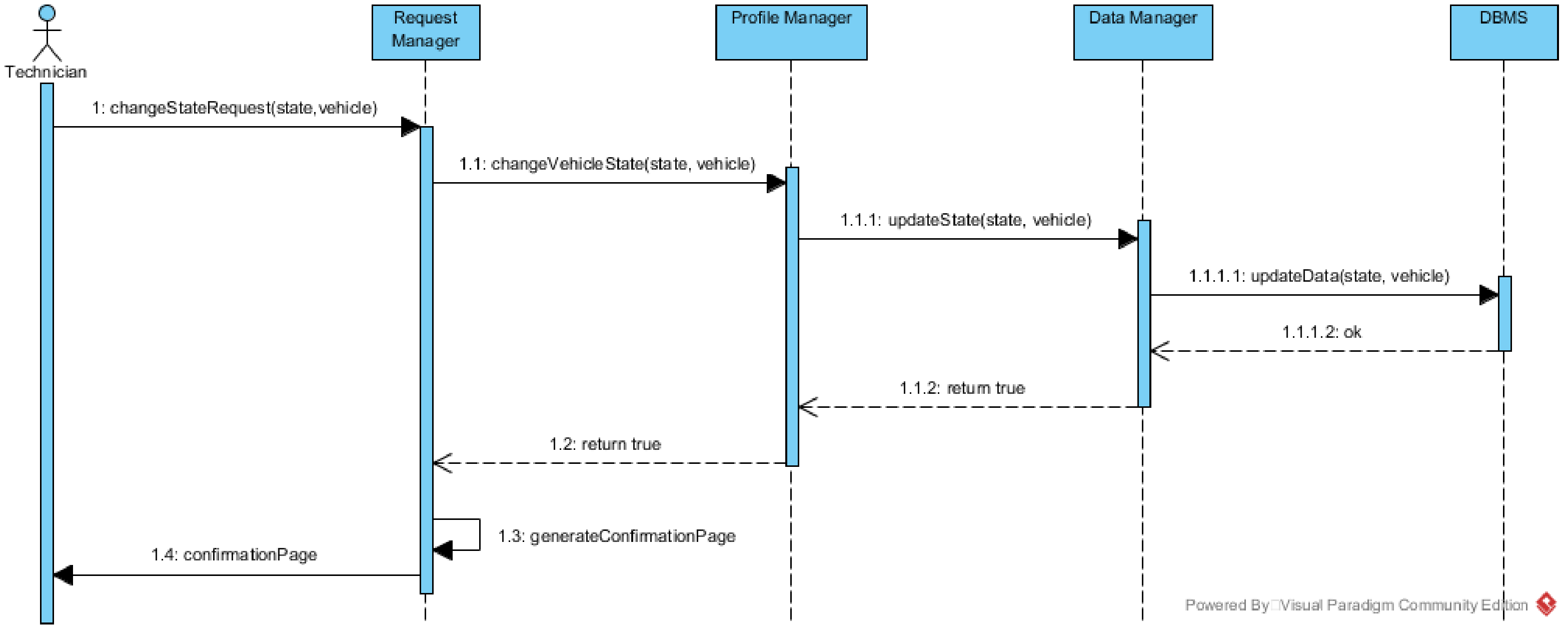
Login


Web Browser
Request Manger
Profile Manager
Data Manager
DBMS
Map Manager
Map Service
Vehicle Manager
1: loginPageRequest()
1.1: generateLoginPage
1.2: loginPage
1.2.1: fillTheFields
1.2.2: loginRequest(email,psw)
1.2.2.1: profileAccess(email,psw)
1.2.2.1.1: checkPassword(email,psw)
1.2.2.1.1.1: findData(email,psw)
alt
[User not found]
1.2.2.1.1.2: return null
1.2.2.1.2: return null
1.2.2.2: return null
1.2.2.3: generateErrorPage
1.2.2.4: errorPage
1.2.2.1.1.3: return user
1.2.2.1.1.3.1: return user
1.2.2.1.1.3.1.1: return user
1.2.2.1.1.3.1.1.1: generateMap(user)
1.2.2.1.1.3.1.1.1.1: findAllVehicles()
1.2.2.1.1.3.1.1.1.1.1: readVehiclesInfo()
1.2.2.1.1.3.1.1.1.1.1.1: findData()
1.2.2.1.1.3.1.1.1.1.1.2: return data
1.2.2.1.1.3.1.1.1.1.2: return data
1.2.2.1.1.3.1.1.1.2: return data
1.2.2.1.1.3.1.1.1.3: getMap()
1.2.2.1.1.3.1.1.1.4: return map
1.2.2.1.1.3.1.1.1.5: generateUserMap(user)
1.2.2.1.1.3.1.1.1.6: return map
1.2.2.1.1.3.1.1.2: generateMapPage
1.2.2.1.1.3.1.1.3: mapPage
Powered By Visual Paradigm Community Edition

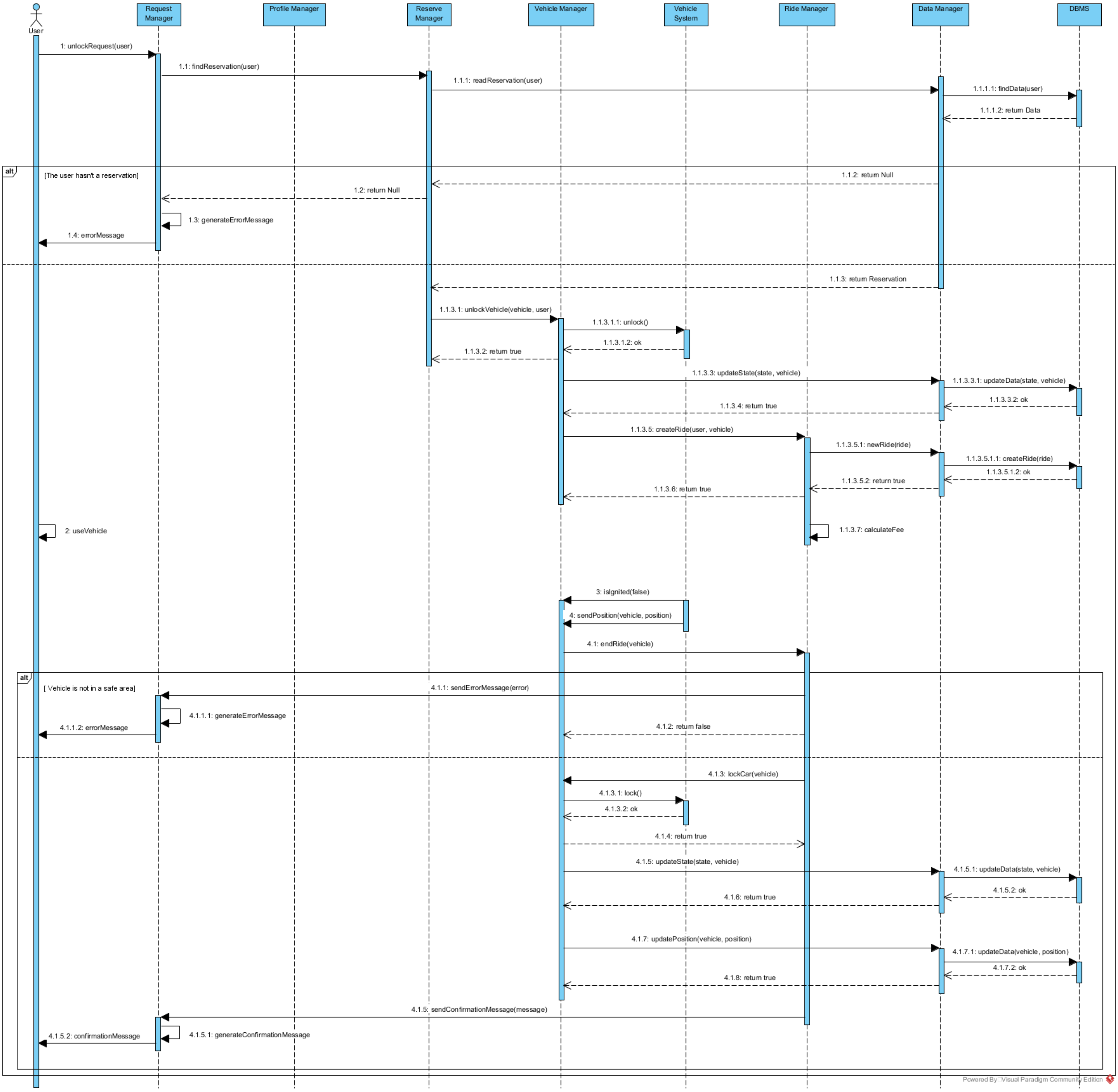
Reservation


Web Browser
Request Manager
Reservation Manager
Data Manager
DBMS
1: reservationRequest(vehicle)
1.1: reserveVehicle(vehicle)
1.1.1: findVehicleInfo(vehicle)
1.1.1.1: findData(vehicle)
1.1.1.2: return Data
1.1.2: return Data
alt
[The selected vehicle is not available]
1.1.3: return false
1.1.3.1: generateErrorPage
1.1.3.2: errorPage()
1.1.4: createReservation(user,vehicle)
1.1.4.1: newReservation(user,vehicle)
1.1.4.2: ok
1.1.5: return true
1.1.6: updateState(state,vehicle)
1.1.6.1: editState(state,vehicle)
1.1.6.2: ok
1.1.7: return true
1.1.8: return true
1.1.8.1: generateConfirmationPage
1.1.8.2: confirmationPage
Powered By Visual Paradigm Community Edition

Change vehicle state


Technician
Request Manager
Profile Manager
Data Manager
DBMS
1: changeStateRequest(state,vehicle)
1.1: changeVehicleState(state, vehicle)
1.1.1: updateState(state, vehicle)
1.1.1.1: updateData(state, vehicle)
1.1.1.2: ok
1.1.2: return true
1.2: return true
1.3: generateConfirmationPage
1.4: confirmationPage
Powered By Visual Paradigm Community Edition

Use vehicle


User
Request Manager
Profile Manager
Reserve Manager
Vehicle Manager
Vehicle System
Ride Manager
Data Manager
DBMS
1: unlockRequest(user)
1.1: findReservation(user)
1.1.1: readReservation(user)
1.1.1.1: findData(user)
1.1.1.2: return Data
alt
[The user hasn't a reservation]
1.1.2: return Null
1.2: return Null
1.3: generateErrorMessage
1.4: errorMessage
1.1.3: return Reservation
1.1.3.1: unlockVehicle(vehicle, user)
1.1.3.1.1: unlock()
1.1.3.1.2: ok
1.1.3.2: return true
1.1.3.3: updateState(state, vehicle)
1.1.3.3.1: updateData(state, vehicle)
1.1.3.3.2: ok
1.1.3.4: return true
1.1.3.5: createRide(user, vehicle)
1.1.3.5.1: newRide(ride)
1.1.3.5.1.1: createRide(ride)
1.1.3.5.1.2: ok
1.1.3.5.2: return true
1.1.3.6: return true
2: useVehicle
1.1.3.7: calculateFee
3: isIgnited(false)
4: sendPosition(vehicle, position)
4.1: endRide(vehicle)
alt
[Vehicle is not in a safe area]
4.1.1: sendErrorMessage(error)
4.1.1.1: generateErrorMessage
4.1.1.2: errorMessage
4.1.2: return false
4.1.3: lockCar(vehicle)
4.1.3.1: lock()
4.1.3.2: ok
4.1.4: return true
4.1.5: updateState(state, vehicle)
4.1.5.1: updateData(state, vehicle)
4.1.5.2: ok
4.1.6: return true
4.1.7: updatePosition(vehicle, position)
4.1.7.1: updateData(vehicle, position)
4.1.7.2: ok
4.1.8: return true
4.1.5: sendConfirmationMessage(message)
4.1.5.1: generateConfirmationMessage
4.1.5.2: confirmationMessage
Powered By Visual Paradigm Community Edition

2.6 Selected architectural styles and patterns

For the developing of our application we have done some choices about patterns and architectures used.

We decided to use the MVC pattern to decouple the interfaces, the logic and the model each other.
In this way we can develop each part separately and in case of modification or upgrade to the software the developers haven’t to touch all the components. Furthermore, this pattern works well with the client server architecture.

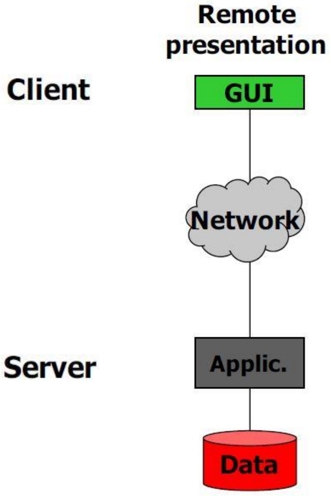
Regarding the architecture, we used a client-server architecture with three tier:

-**Client**: the user must have a web browser on his device in order to connect to the web site and access to the functionalities offered by the server.

-**Server**: in our server there are both the web server which interfaces with the browser of the client, and the application server in which there are the logic of our system and the methods to interface with the database.

-**Database**: in the database there are the persistent data of our system like vehicles, users and so on.

So our client-server architecture has a thin client, indeed the client has to manage only the presentation side, while the server manages all the logic.



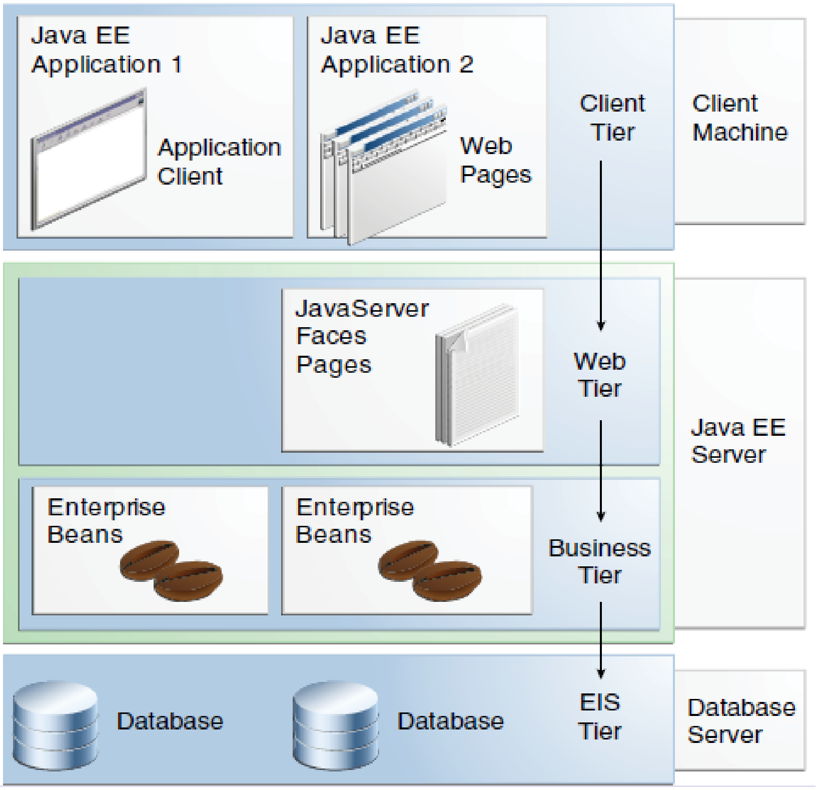
To develop our system, we will use the APIs and containers given by Java EE.

JSP: Java Sever Pages for the web server, to create dynamic web pages to send to the client.

Enterprise beans: we will use session beans to manage the communication between the web tier e the business tier in order to increase the scalability and the security.

JPA: Java Persistence API for the management of the database. With this API we can guarantee the persistence of the data, with the protection to failures and so on.

We followed the java EE architecture:



3. Algorithm design

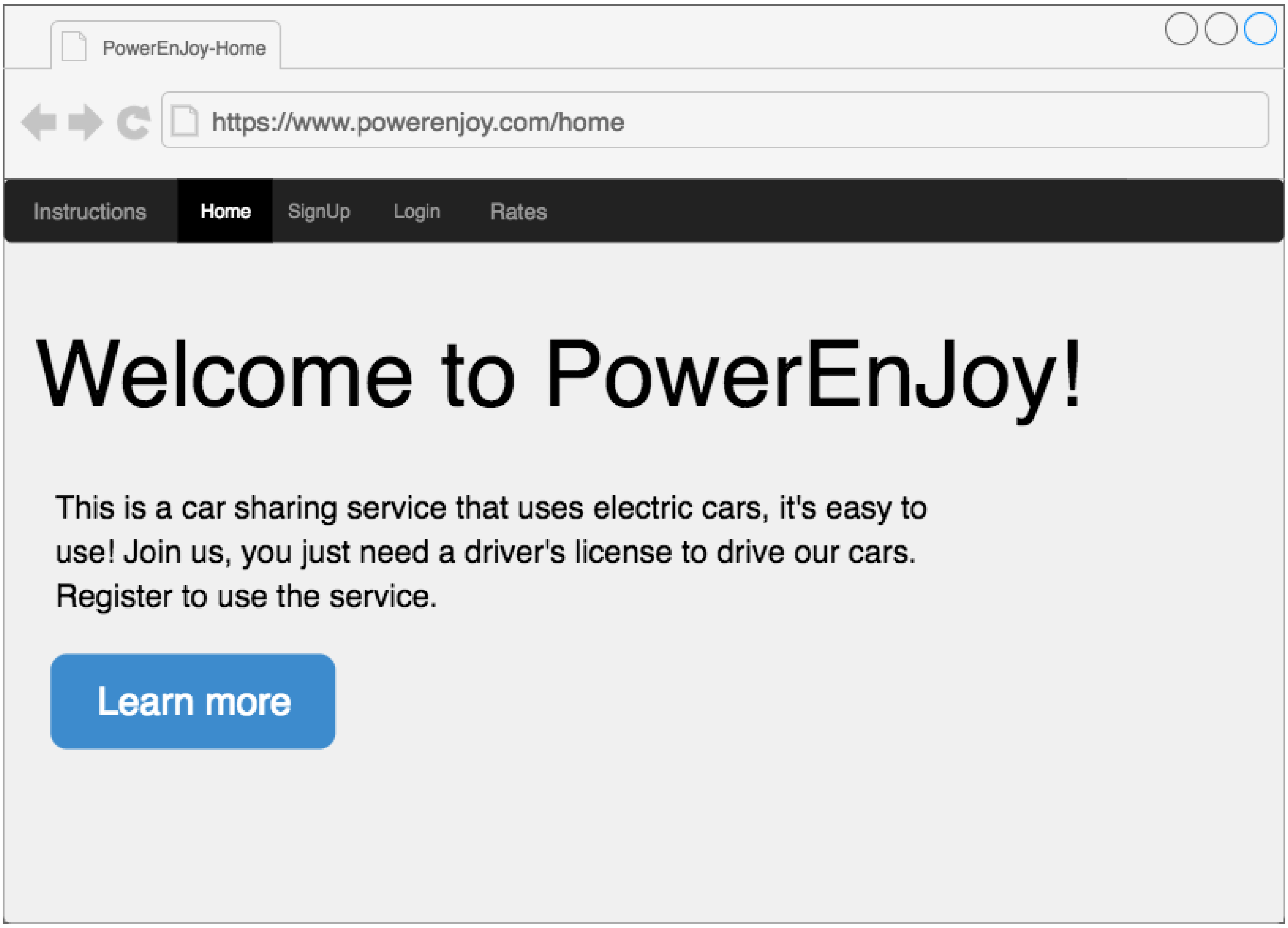
To develop this software, there are no particularly complex algorithms, and there are no particular constraints for developers. The important thing is that the code works and meets the functional requirements described in RASD. The algorithm that calculates the charge for the user must, of course, comply with all the rules that have been explained in the RASD

4. User Interface Design

4.1 Mockups

We provided some pages of our web app that are useful to understand how users will interact with the system.
So these sketches may be improved by those who will design the graphical interface.

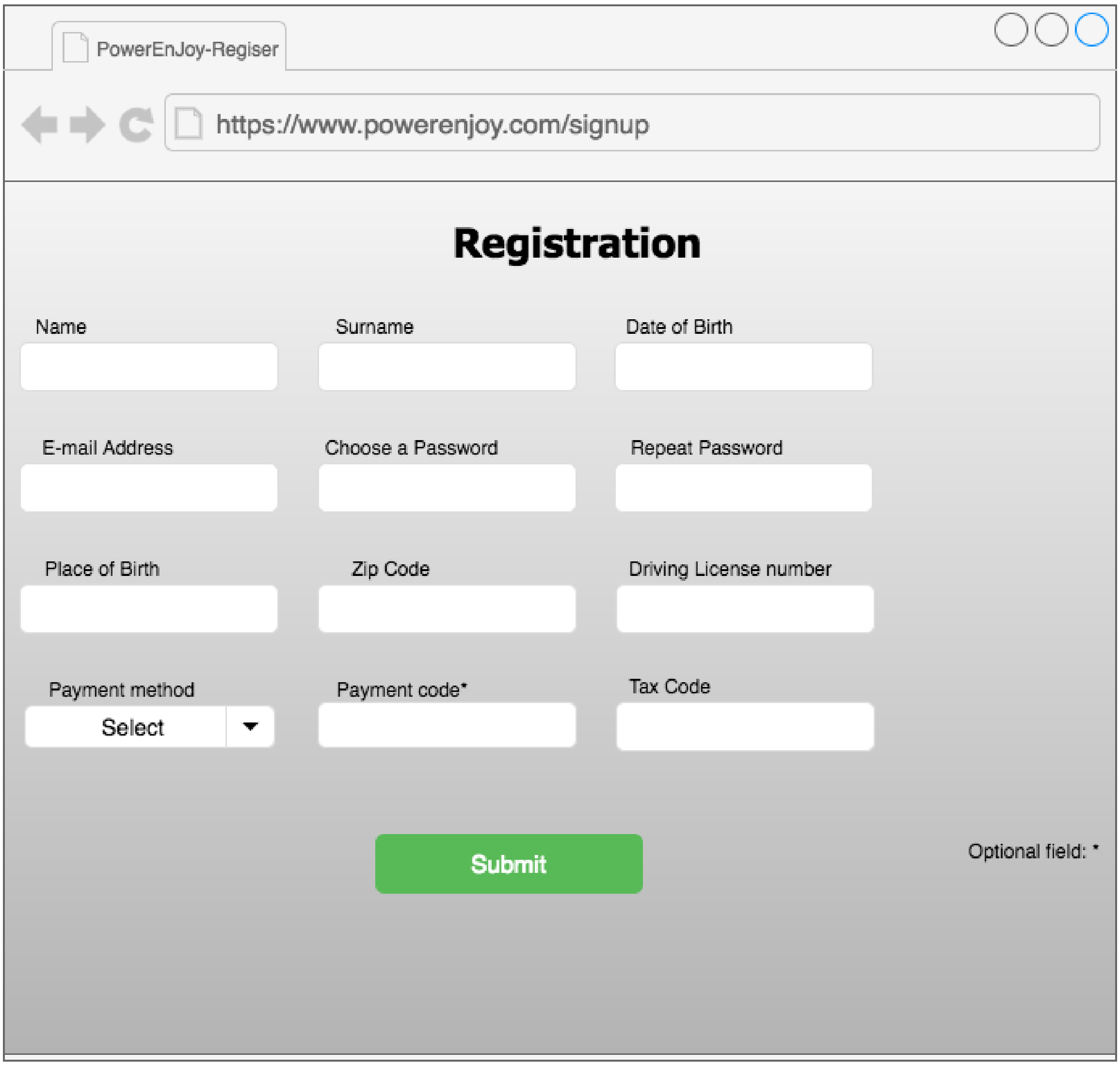
This is the PowerEnJoy homepage



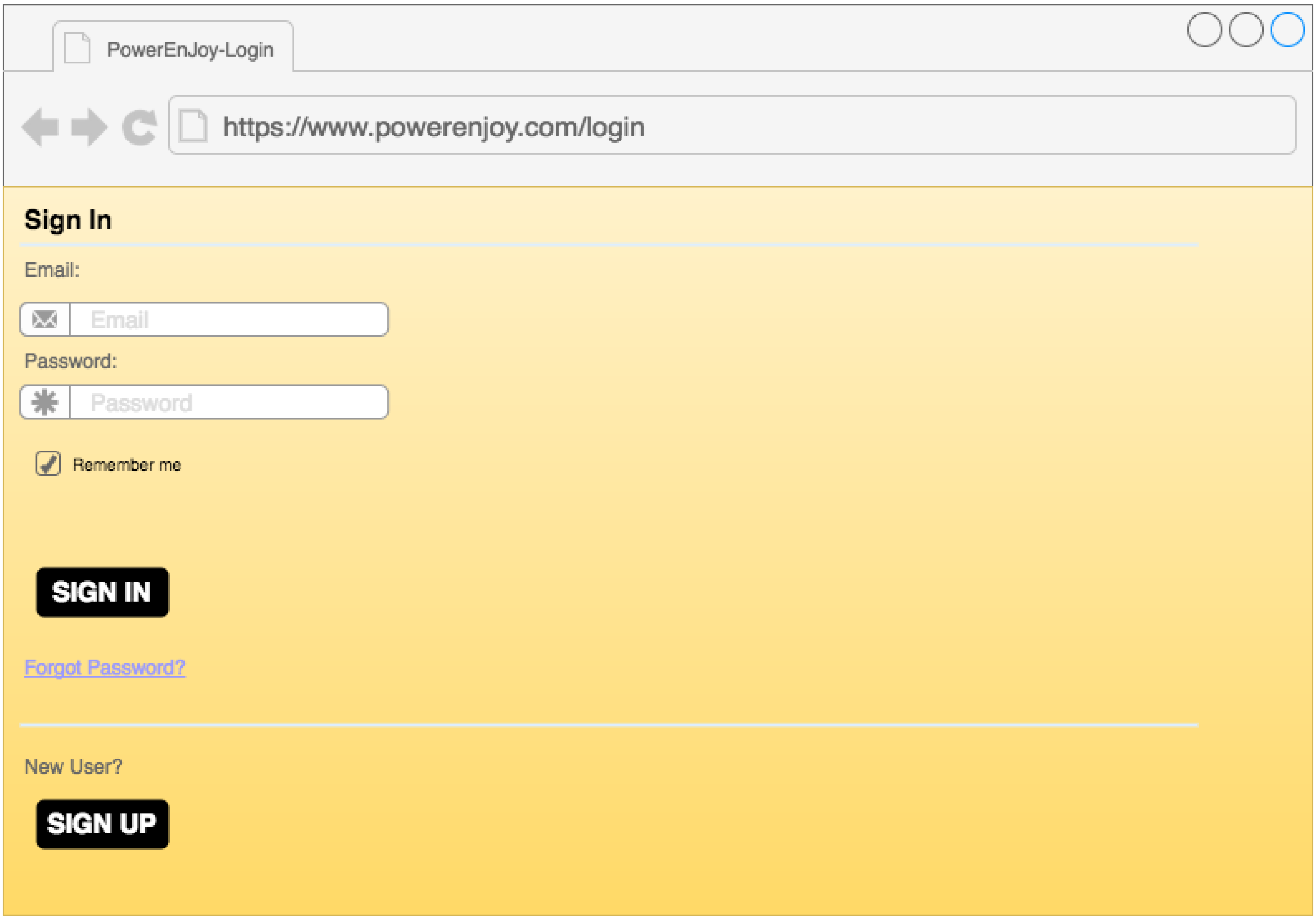
Welcome to PowerEnJoy!

This is a car sharing service that uses electric cars, it's easy to use! Join us, you just need a driver's license to drive our cars. Register to use the service.

This is the registration page



This is the login page



PowerEnJoy-Login

https://www.powerenjoy.com/login

Sign In

Email:

Email

Password:

Password

☑ Remember me

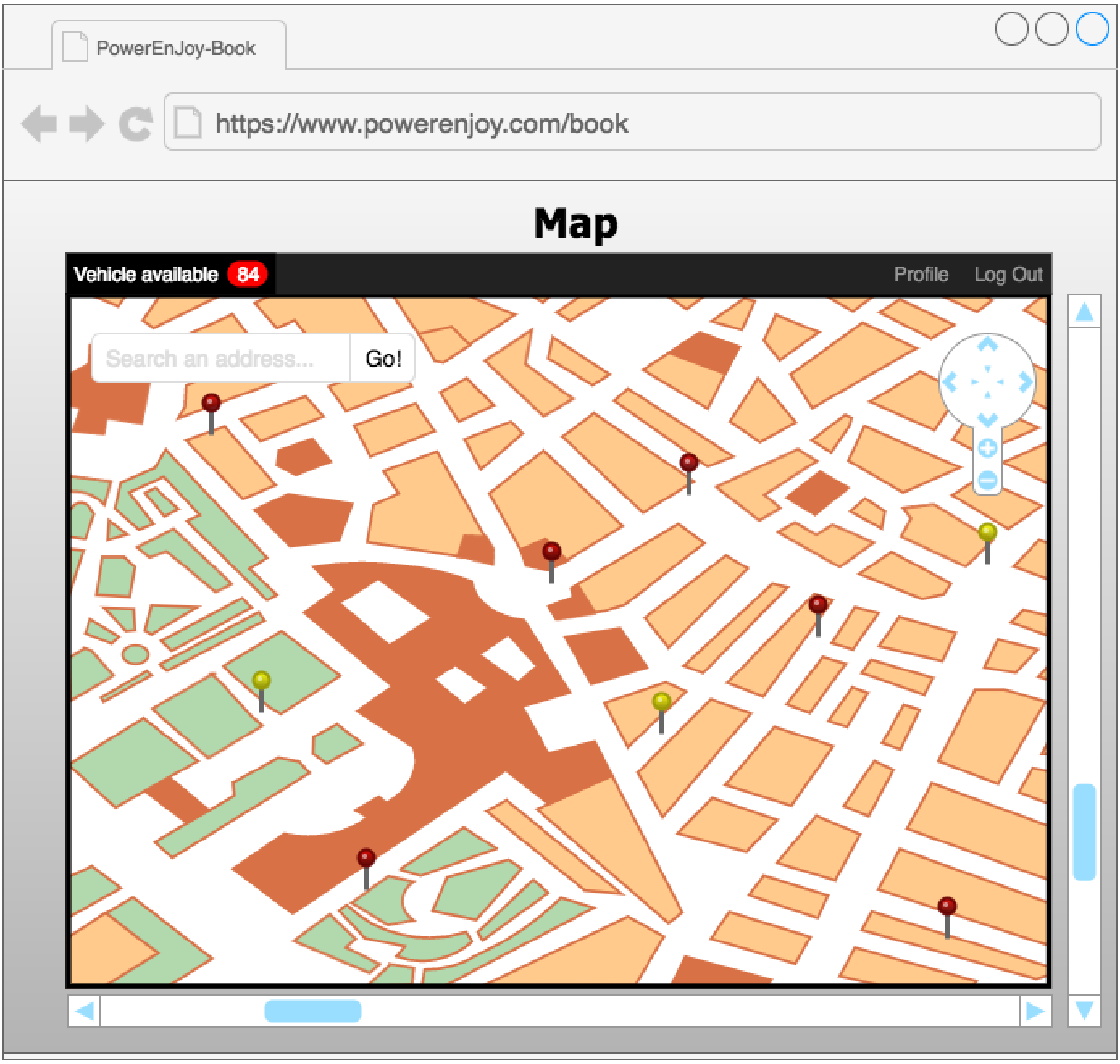
SIGN IN

Forgot Password?

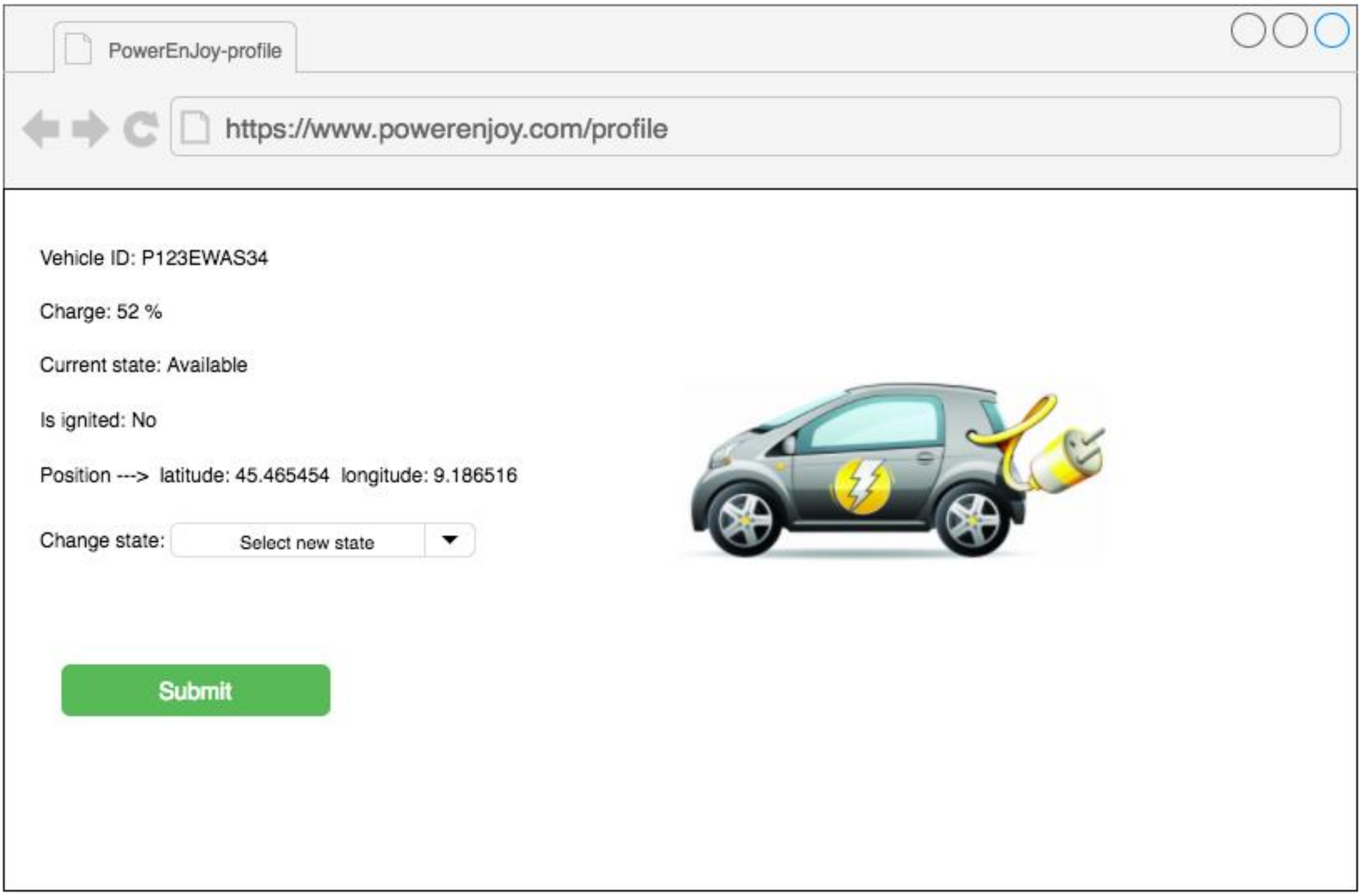
New User?

SIGN UP

This is the map page, where the users can localize the cars



This is the page for the technicians that shows car details.
There is also a page like this for users, that permits to reserve
the vehicle, but of course, is not possible to change the state
of the vehicle from that page.



Vehicle ID: P123EWAS34

Charge: 52 %

Current state: Available

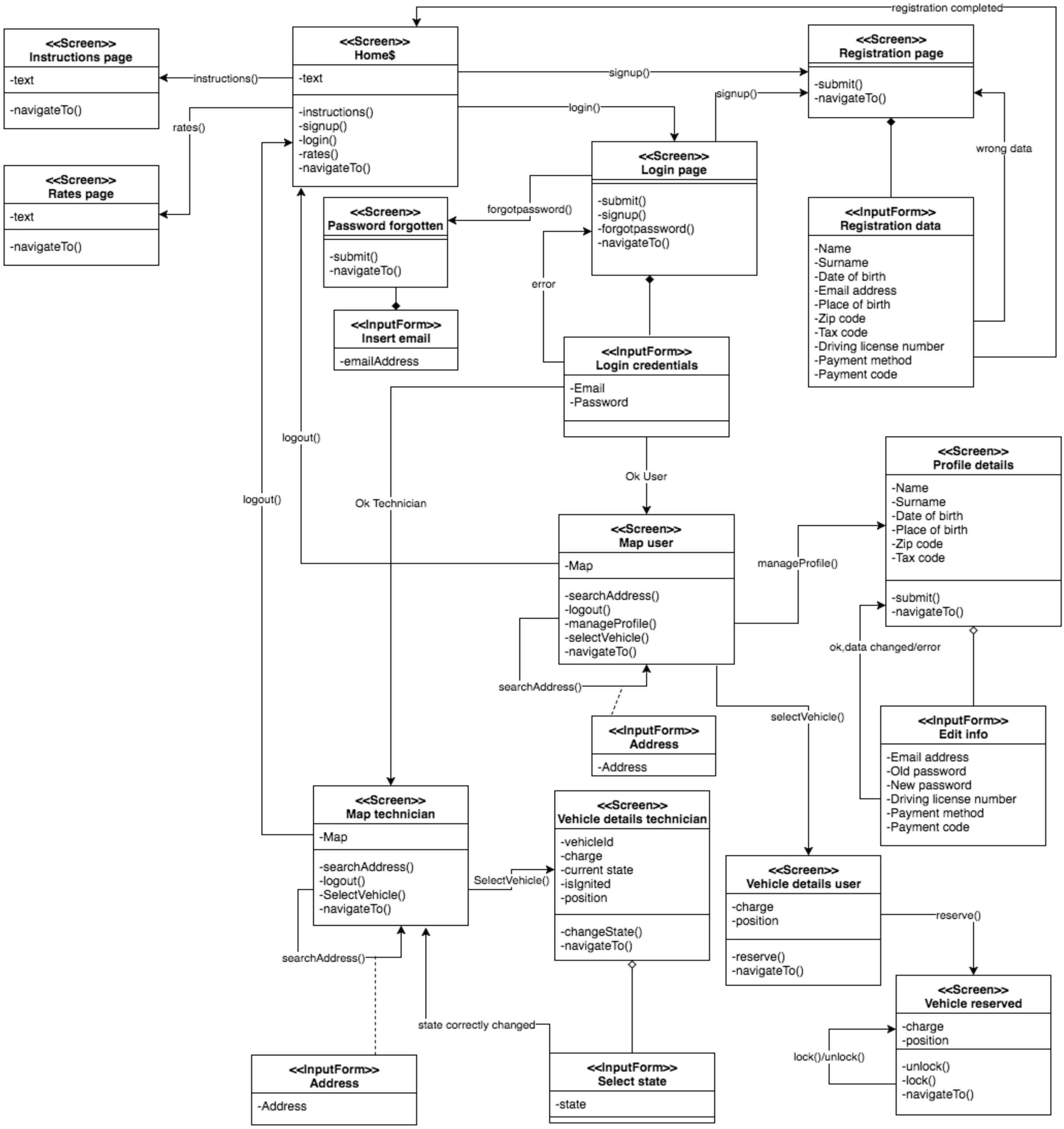
Is ignited: No

Position ---> latitude: 45.465454 longitude: 9.186516

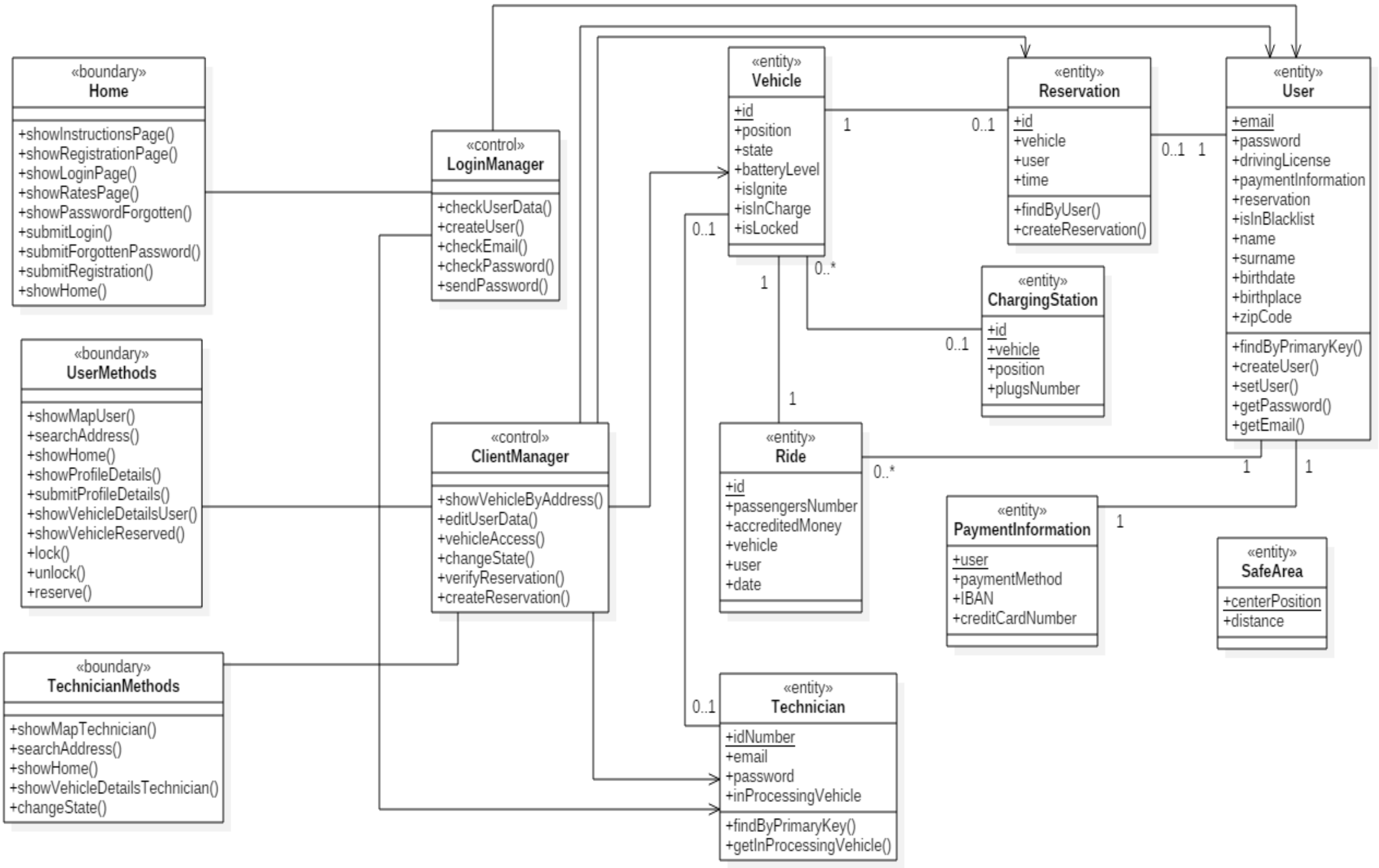
Change state: Select new state

4.2 UX Diagram

From UX diagram you can understand more precisely what are the structures of the pages and what are the possible transitions between them.

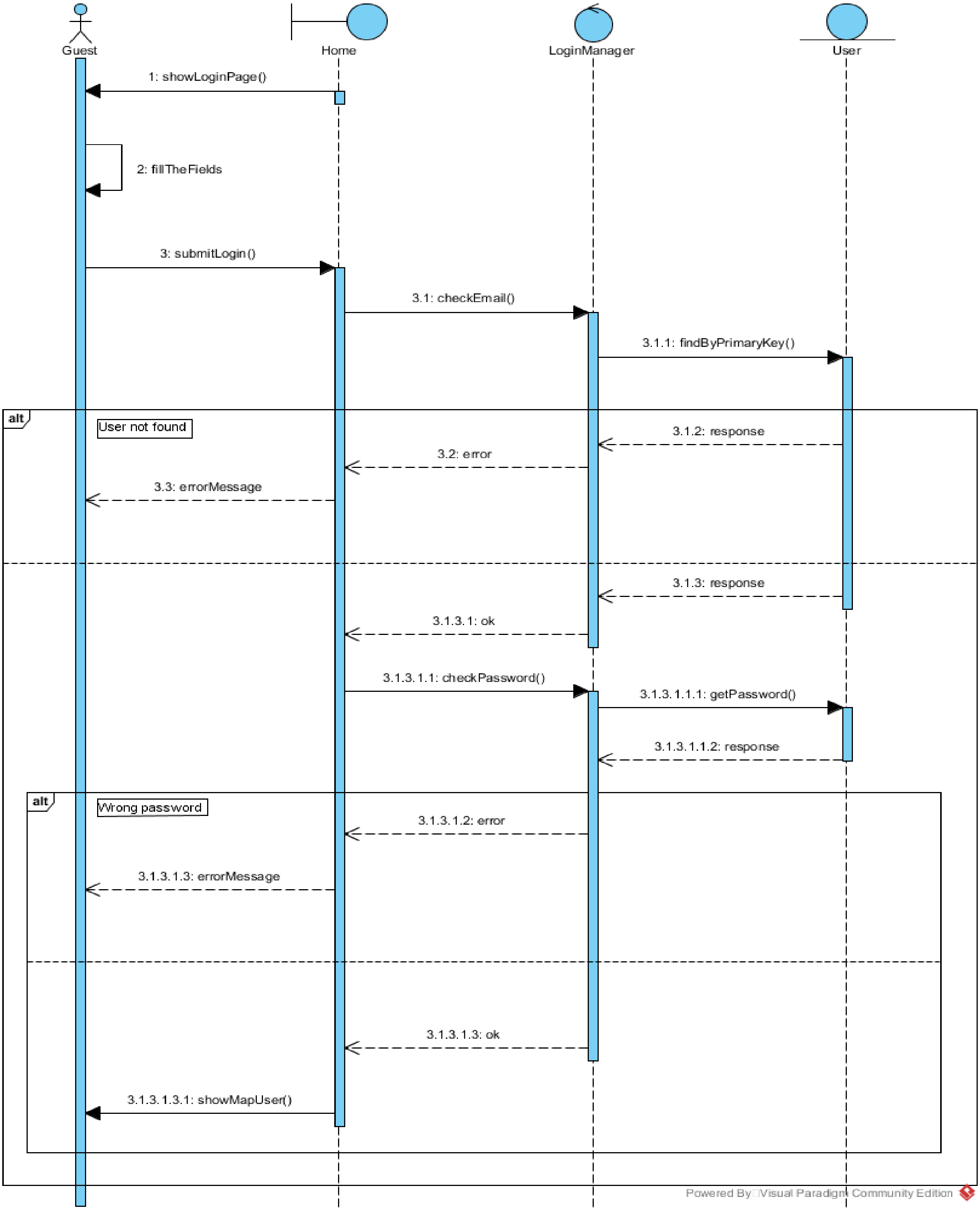


4.3 BCE Diagram

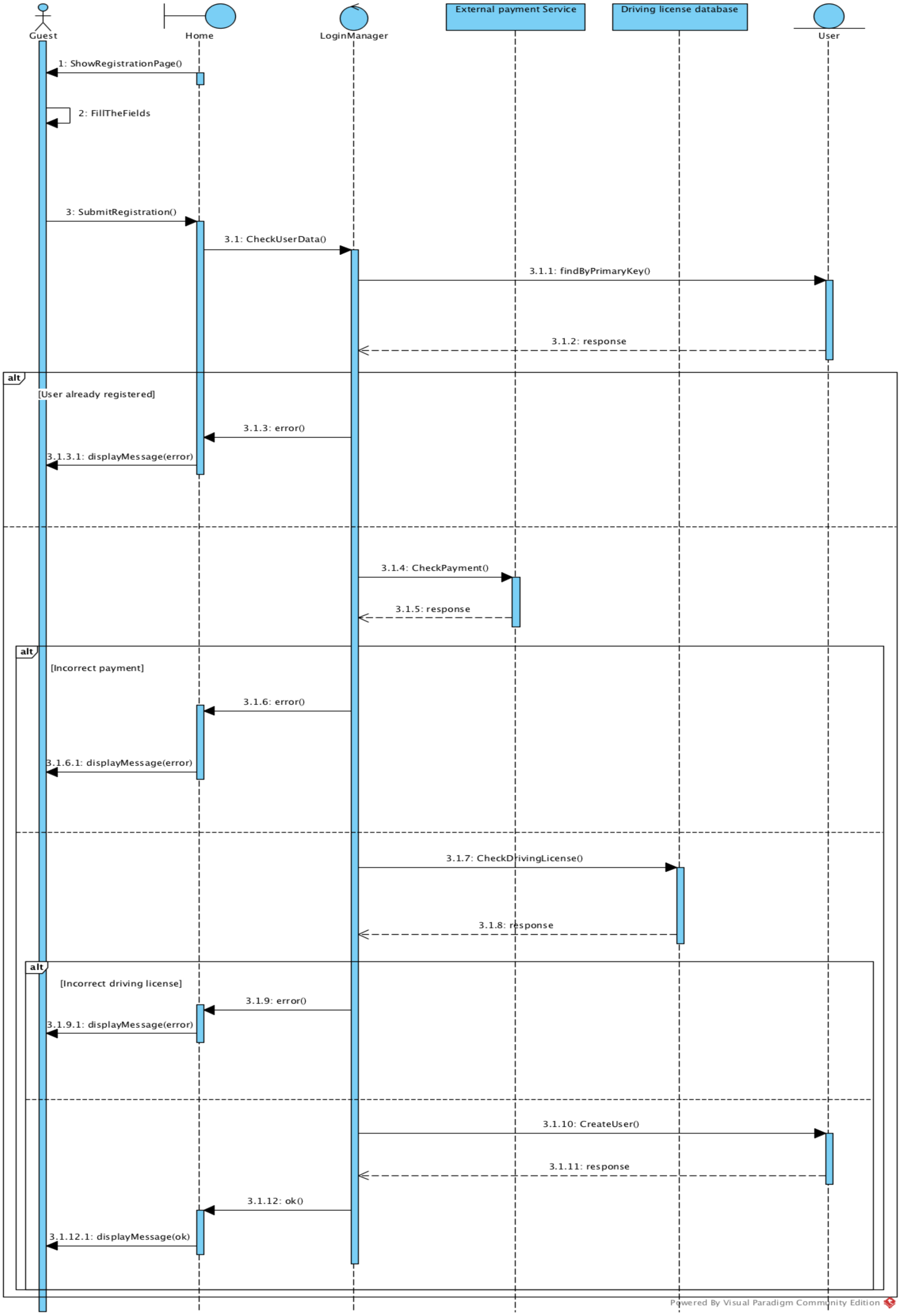


In addition to the BCE diagram we have done some sequence diagrams to show the interaction between the user and the system, through the web interface, and how the input will affect the data on the database.

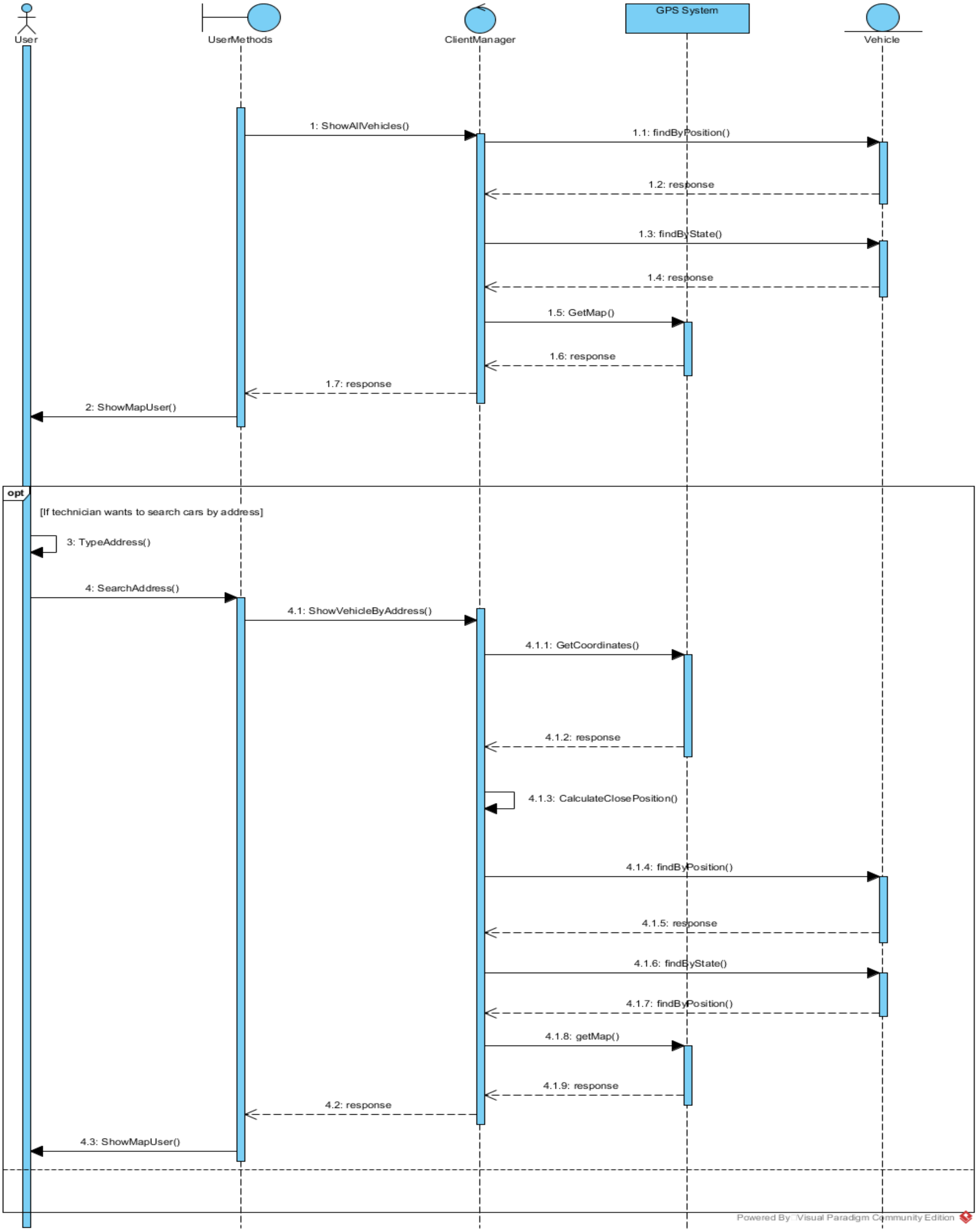
Login sequence diagram



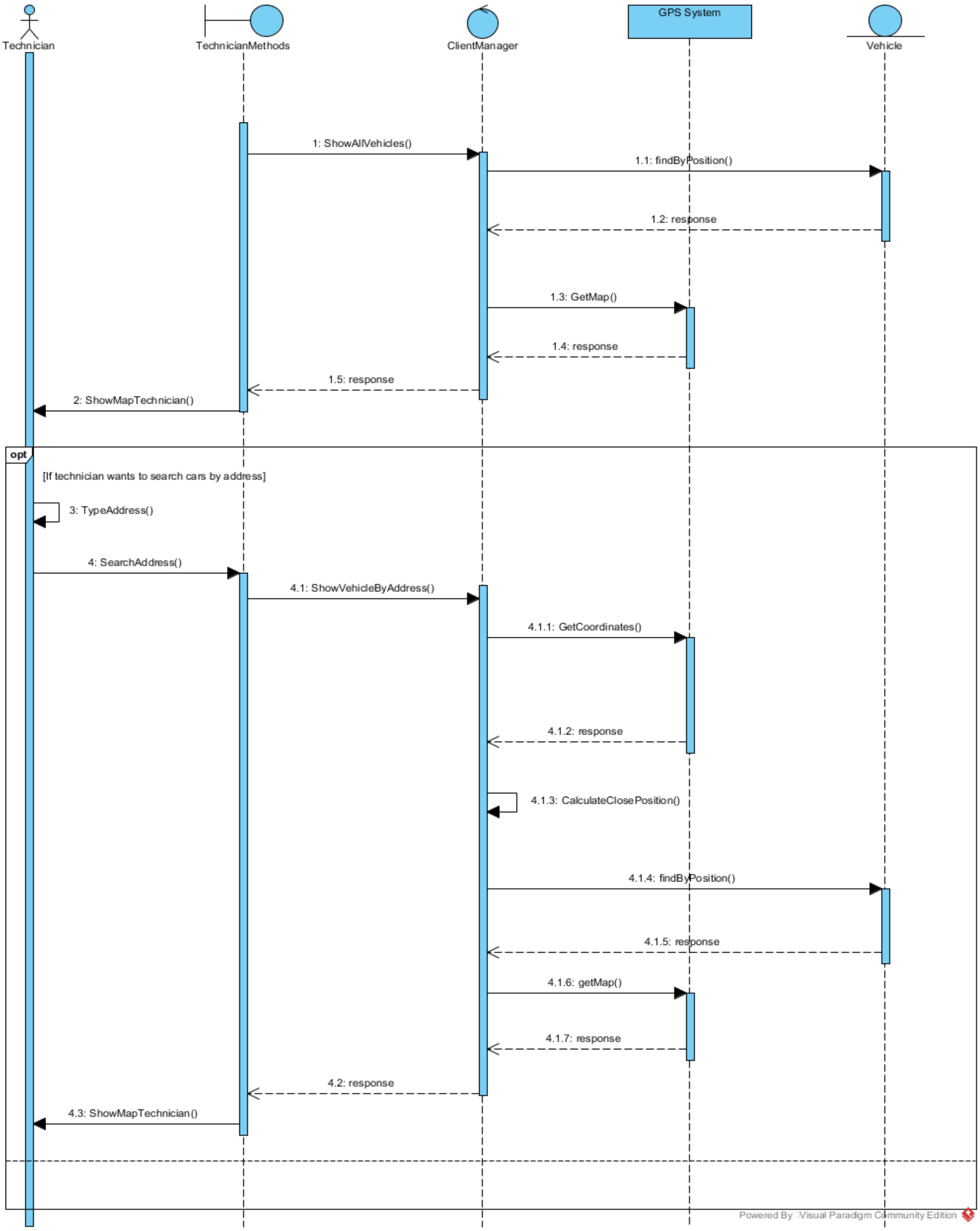
Registration sequence diagram



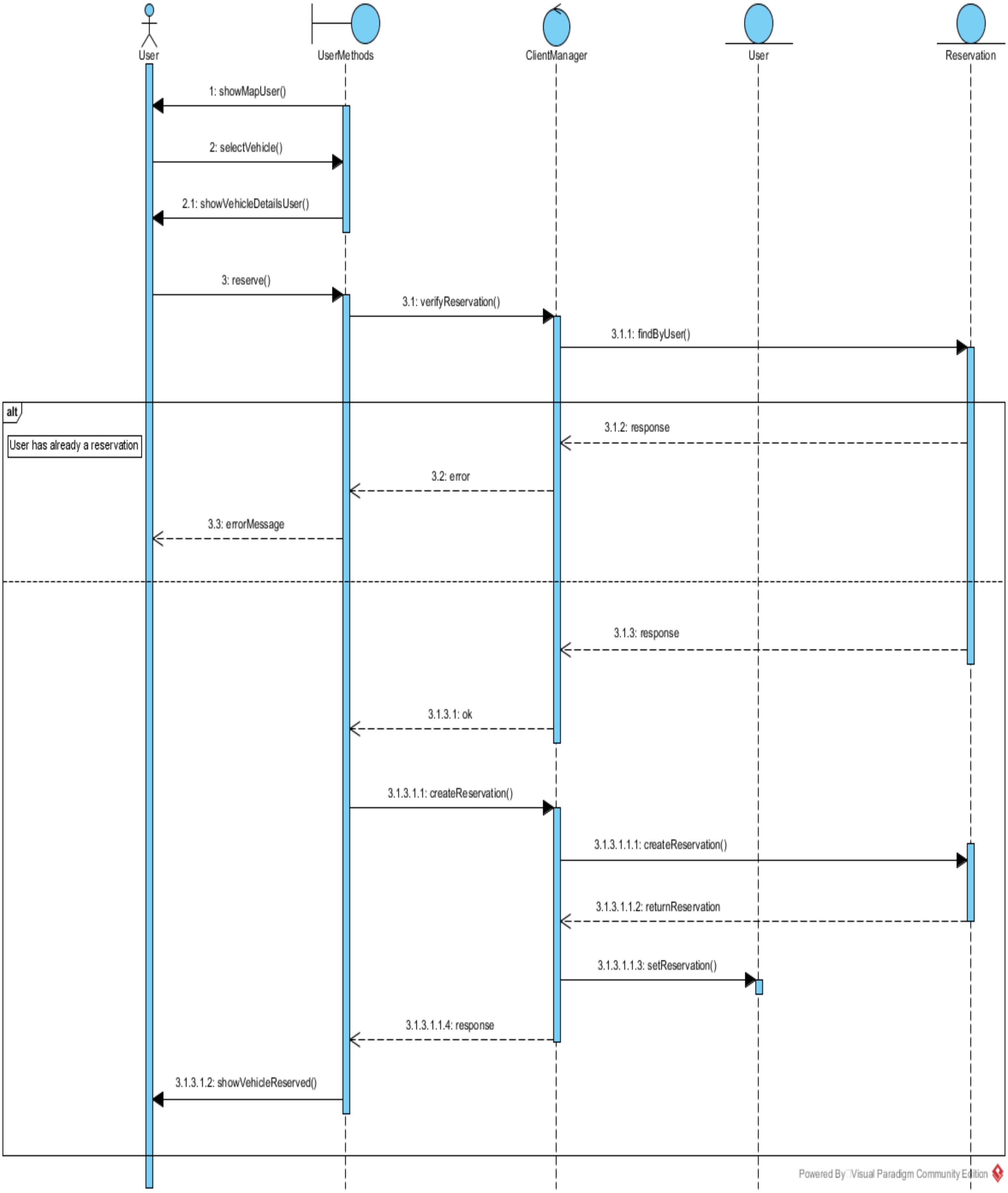
Show vehicles positions(user) sequence diagram



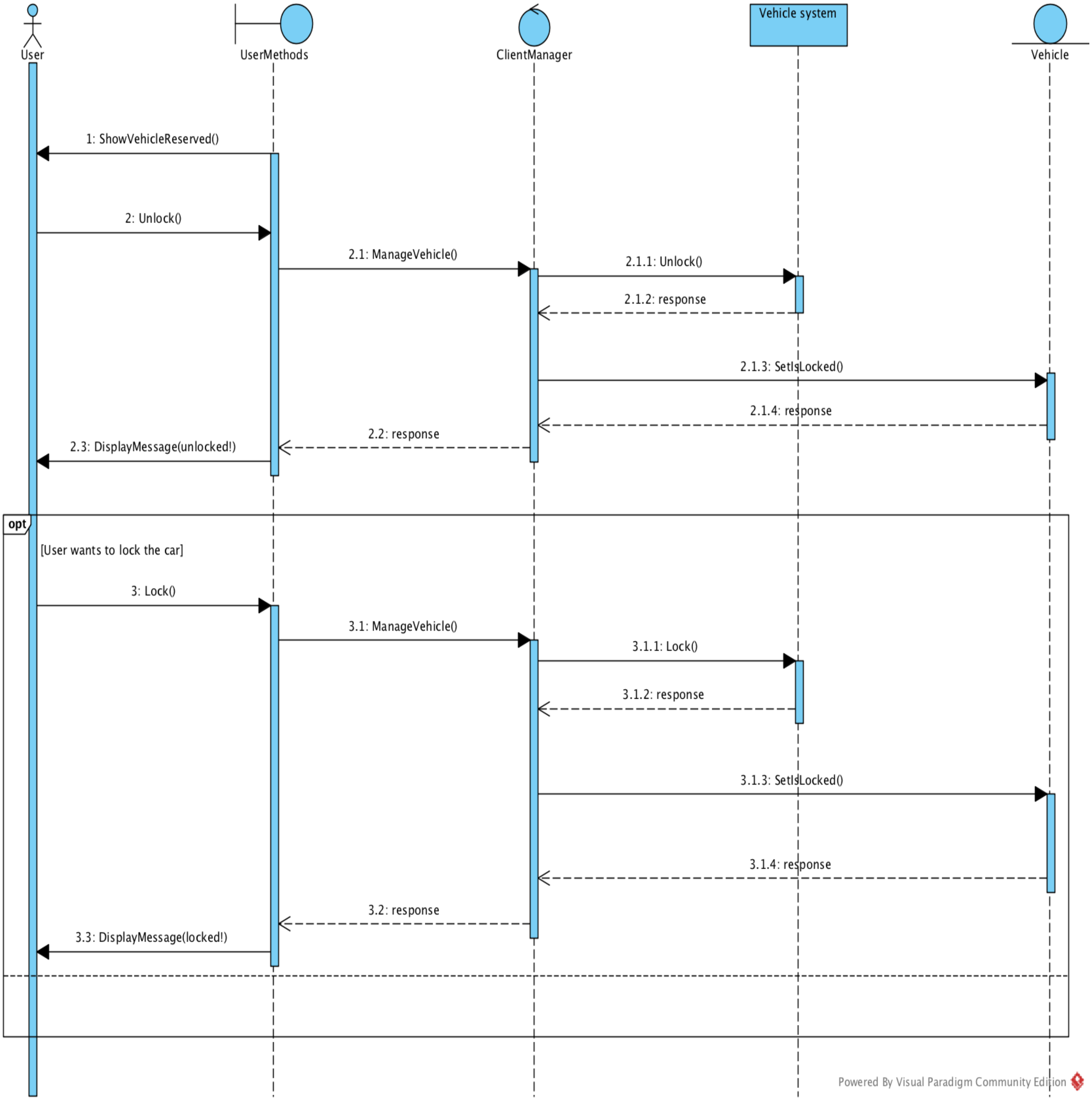
Show vehicles positions(technician) sequence diagram



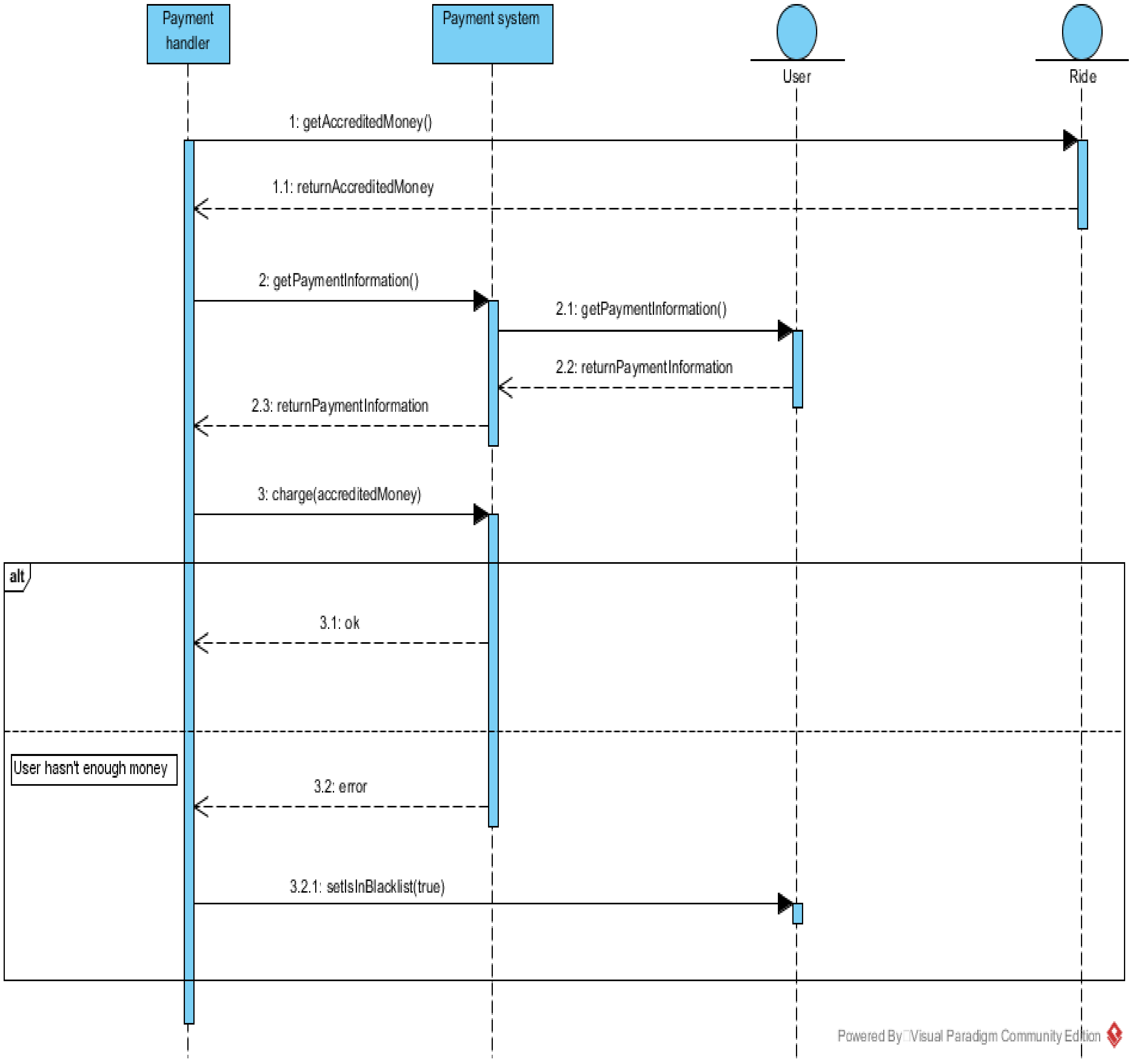
Reservation sequence diagram


User
UserMethods
ClientManager
User
Reservation
1: showMapUser()
2: selectVehicle()
2.1: showVehicleDetailsUser()
3: reserve()
3.1: verifyReservation()
3.1.1: findByUser()
alt
User has already a reservation
3.1.2: response
3.2: error
3.3: errorMessage
3.1.3: response
3.1.3.1: ok
3.1.3.1.1: createReservation()
3.1.3.1.1.1: createReservation()
3.1.3.1.1.2: returnReservation
3.1.3.1.1.3: setReservation()
3.1.3.1.1.4: response
3.1.3.1.2: showVehicleReserved()
Powered By Visual Paradigm Community Edition

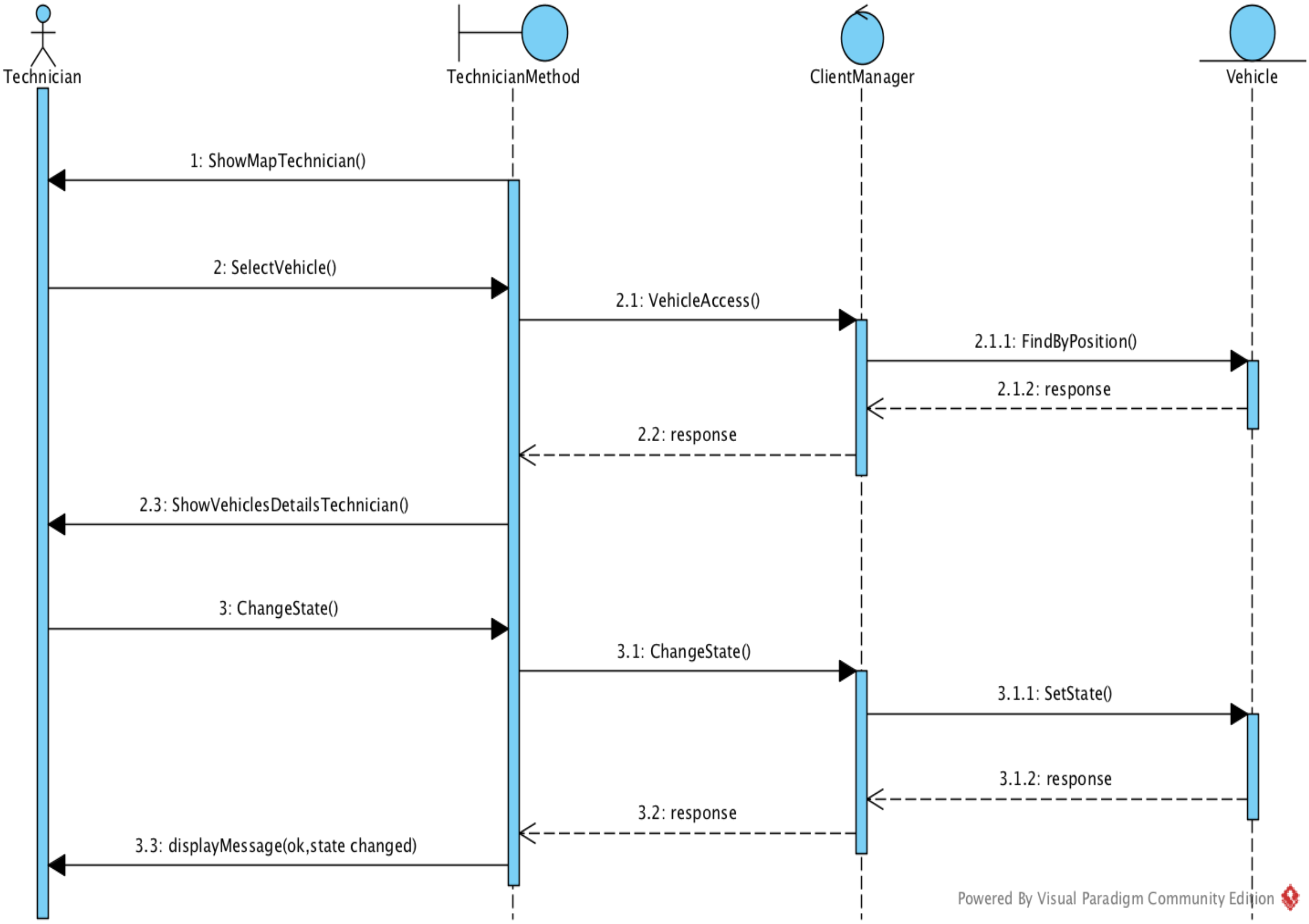
Use the vehicle sequence diagram



Payment sequence diagram


Payment handler
Payment system
User
Ride
1: getAccreditedMoney()
1.1: returnAccreditedMoney
2: getPaymentInformation()
2.1: getPaymentInformation()
2.2: returnPaymentInformation
2.3: returnPaymentInformation
3: charge(accreditedMoney)
alt
3.1: ok
User hasn't enough money
3.2: error
3.2.1: setIsInBlacklist(true)
Powered By Visual Paradigm Community Edition

Change state (technician) sequence diagram



5. Requirements traceability

G1. A person who has the right requirements must be able to register himself to the system.

- Request Manager
- Profile Manager
- Data Manager

G2. A registered person must be able to authenticate himself to the system.

- Request Manager
- Profile Manager
- Data Manager

G3. A user must be able to localize the positions of the available vehicles.

- Request Manager
- Profile Manager
- Map Manager
- Vehicle Manager

G4. A user must be able to reserve an available vehicle, for a limited time.

- Request Manager
- Profile Manager
- Reserve manager
- Vehicle Manager
- Data Manager

G5. A user who has reserved a vehicle, must be able to use it.

- Request Manager
- Vehicle Manager

G6. The system must properly charge for the user the cost of used services.

- Payment Manager
- Vehicle Manager

G7. The system must properly manage the availability of vehicles.

- Vehicle Manager

G8. The system must simplify the organization of the technicians in their work.

- Request Manager
- Profile Manager
- Data Manager
- Vehicle Manager
- Map Manager

6. References

6.1 Used Tools

- GitHub
- Microsoft Word
- Draw.io
- Visual Paradigm community edition
- Star UML

Hours of work

These are approximatively the time we spent to write this document

Luca Santini: 40 hours

Riccardo Remigio: 40 hours

Changelog

- Modified the component view
- Added sequence diagram for the internal logic
- Added description of the components