POLITECNICO
MILANO 1863

Politecnico di Milano
A.A. 2016–2017
Software Engineering 2: "PowerEnJoy"

Design Document

Pietro Ferretti, Nicole Gervasoni, Danilo Labanca

December 11, 2016

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to specify in a more technical way how our system is designed to be built and work. This document is intended for developers, designers and all interested stakeholders as a guideline for the implementation of the system.

This document will first of all illustrate the system architecture, the components that make it up and how each of its components is designed to interact with each other.

Furthermore this document will describe the behaviour of the system and the applications, from both the back-end and the front-end side.

Finally this document will describe the patterns we used to design the system and how each of them comes into play.

1.2 Scope

The aim of this project is to specify in detail a new digital management software for PowerEnJoy, a car-sharing service that employs electric cars only. PowerEnjoy will offer a very valuable service to its users, letting them borrow cars to drive around the city freely, as an alternative to their own vehicles and public transport.

PowerEnJoy's users, after registering, will be able to reserve, unlock and drive the cars our system will provide. Users will be charged per minute until they park the car in a safe area and end the ride. Users will be able to park their car temporarily and use it again later, or end their ride remotely.

Our system will incentivize virtuous behaviour by offering several discounts if certain conditions are met (like charging a car at a power grid station).

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- *Guest*: a person that is not registered to the system.
- *User*: a person that is registered to the system. Users can log in to the system with their email or username and their password. Their first name, last name, date of birth, driving license ID are stored in the database.
- *Safe area*: a location where the user can park and leave the car. Users can end their ride and park temporarily only in these locations. The set of safe areas is predefined by the system.
- *Power grid station* or *Charging station*: a place where cars can be parked and plugged in. While a car is plugged in a power grid station its battery will be recharged. Power grid stations are by definition safe areas.
- *Available car*: a car that is currently not being used by any user, and has not been reserved either. Available cars are in good conditions (not dirty nor damaged) and don't have dead batteries.
- *Reservation*:
  - the operation of making a car reserved for a user, i.e. giving permission to unlock and use the car only for that user, forbidding reservations by other users.
  - the time period between the moment a reservation is requested and the moment the user unlocks the car, or the reservation is canceled.
- *Ride*: the time period from the moment a reserved car is unlocked to the moment the user notifies that he wants to stop using the car and closes all the doors. A ride doesn't stop when a car is temporarily parked, but continues until the user chooses to leave the car definitely.
- *Temporary parking*: the act of parking a car in a safe area and, after notifying the system, locking it and leaving it for a finite amount of time. The user that does this retains the right to use the car and can unlock it later to use it again.
- *Bill*: a record of the money owed by the user at the end of a ride.
- *Outstanding bill*: a bill that hasn't been paid yet.
- *Suspended user*: a user that cannot reserve or use cars. Usually users are suspended because they have outstanding bills.
- *Payment method*: a way to transfer money from the user to the system. Our system will only accept credit cards and online accounts like Paypal.
- *Payment API*: an interface to carry out money transactions, offered by the external provider associated to the payment method used (e.g. a bank).

1.3.2 Acronyms

- **DD**: Design Document
- **RASD**: Requirements Analysis and Specification Document
- **DB**: Database
- **DOB**: Date of birth
- **PGS**: Power Grid Station
- **GPS**: Global Positioning System
- **API**: Application Programming Interface

1.3.3 Abbreviations

- **[Gx]**: Goal
- **[RE.x]**: Functional Requirement
- **[UC.x]**: Use Case

1.4 Reference Documents

- IEEE Std. 1016-2009, "IEEE Standard for Information Technology – Systems Design – Software Design Descriptions"
- ISO/IEC/IEEE Std. 42010:2011, "Systems and software engineering – Architecture Description"
- Specification document: "Assignments AA 2016-2017.pdf"

1.5 Document Structure

This document is structured as follows:

- **Section 1 – Introduction**
- **Section 2 – Architectural Design**
  - **Overview:** this section shows a high-level view of our system's architecture.
  - **Component View:** this section contains a more detailed and in-depth analysis of the components that comprise the architecture, and how they interact with each other.
  - **Deployment View:** this section shows how the architecture components will be deployed on physical machines.
  - **Runtime View:** in this section we use sequence diagrams to show the flow of events for the most significant functionalities of the application.
  - **Component Interfaces:** the interfaces between the various components are presented in this section.
  - **Selected Architectural Styles and Patterns:** this section explains the architectural choices taken and the patterns followed during the creation of the application.
- **Section 3 – Algorithm Design:** this section proposes algorithms to realize some of the most important functionalities. A quick complexity analysis is also included.
- **Section 4 – User Interface Design:** this section formalizes with more details the user interfaces shown and described in the RASD, using UX and BCE diagrams.
- **Section 5 – Requirements Traceability:** this section aims to explain how the goals identified in the RASD are linked to the design and architecture elements described in this document.
- **Effort Spent**
- **References**
- **Revisions**

2 Architectural Design

2.1 Overview

To satisfy our requirements we rely on a primarily client-server architecture. We identified three main types of clients: the mobile/web application, the vehicle application, and the embedded systems that keep track of the sensors in cars and power grid stations.

We decided to adopt a web-based framework (e.g. PhoneGap) to develop the mobile application, so that it would work very similarly to the web one, without needing to design a different interface.

The mobile/web application will first request the necessary web pages and scripts from the dedicated Web Application Server, then the scripts will collect data from the central server via a REST API as needed.

The vehicle application is not web based and will contact only the central server to get data and send requests.

The vehicle and the power grid stations will periodically send updates to the central servers. Cars must also be listening for commands from the servers.

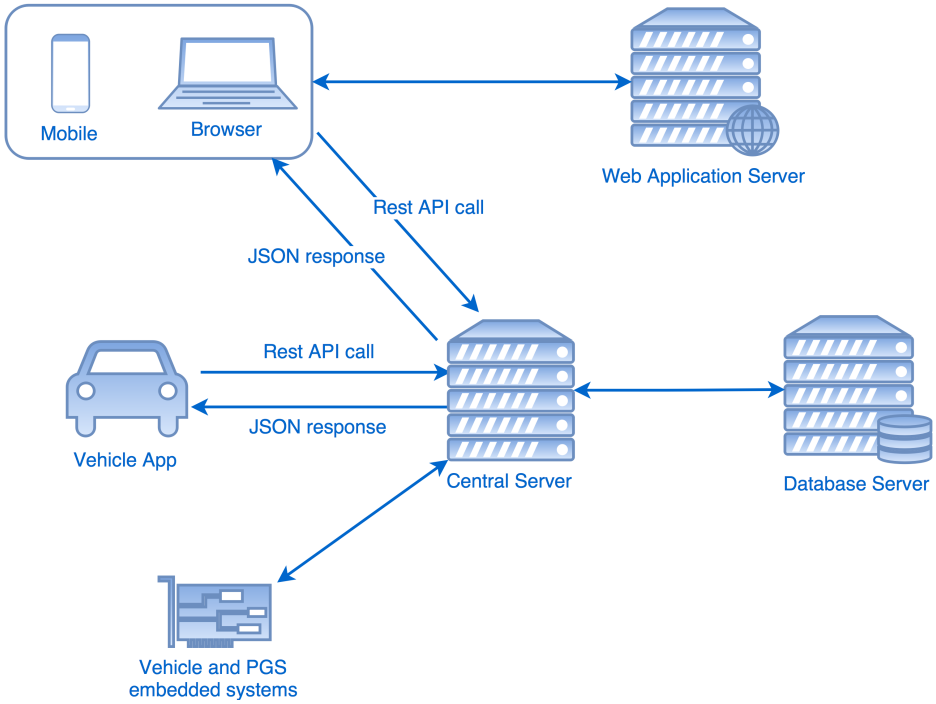

Figure 1: Architecture Overview

2.2 Component View

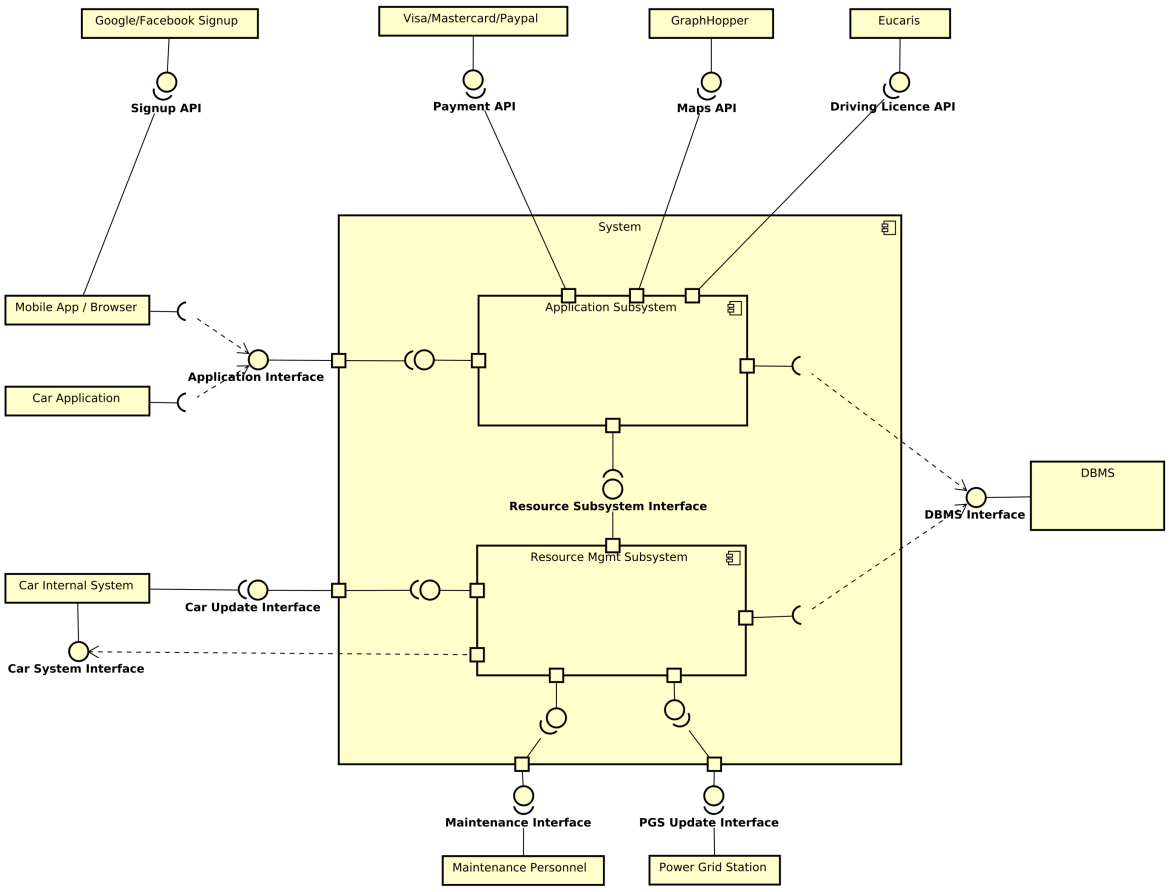

Figure 2: Component Diagram Overview

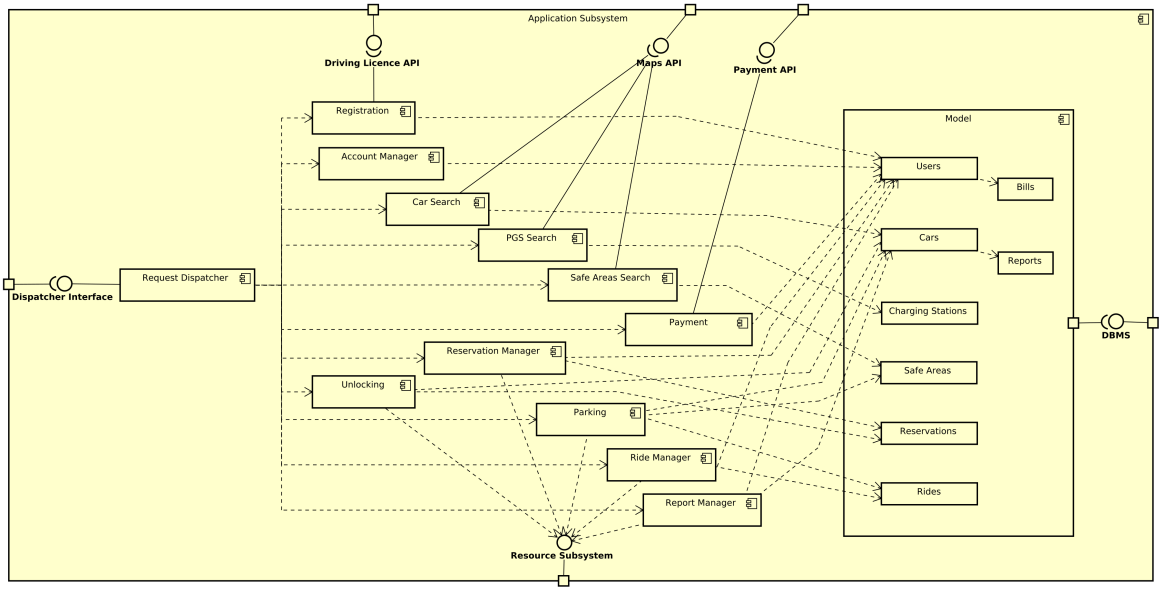

Figure 3: Component Diagram for the Application Subsystem

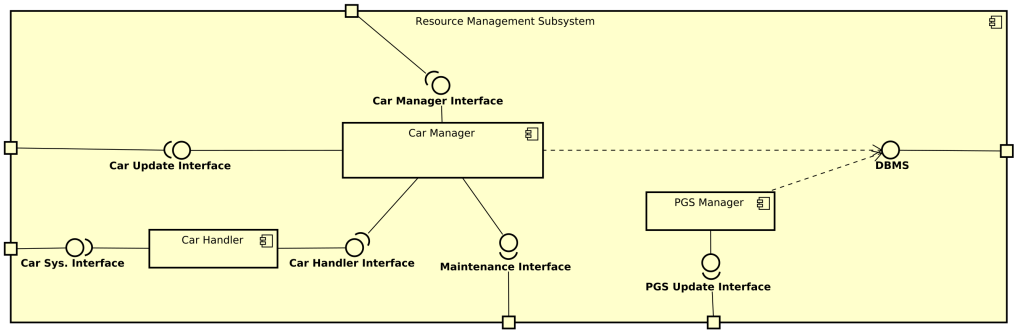

Figure 4: Component Diagram for the Resource Management Subsystem

The system offers many different interfaces, each for a different role:

- an Application Interface for the mobile, web and the car applications to connect;
- a Car Update Interface through which the cars can send updates about their state to the system;
- a Maintenance Interface to let the people in charge of this mark a car as available when it is repaired or recharged;
- a PGS Update Interface for the power grid stations' systems to send updates about the state of the plugs.

Furthermore, the car itself has an interface to receive commands from the main system (like unlocking the doors).

The main system can run queries on the DBMS through a dedicated interface.

The external APIs used by the system are also depicted.

The main system is divided into two macro-components: the Application Subsystem and the Resource Management Subsystem. The first handles all the operations related to the user applications and interfaces, while the second takes care of keeping track of all the automatic updates from the sensors in the cars and the charging stations.

The separation lets us manage the two families of functionalities in a more flexible way, by decoupling them. For example the Resource Subsystem could be considered as more critical for our system, and be run on servers that are more reliable and resistant to failure.

On the Application Subsystem the Request Dispatcher collects all types of requests (searches, reservations...) and redirects them to the specific components. The Application Subsystem can also send requests to the Resource Subsystem to send commands to the cars.

The Model is a representation of the data contained in the DBMS in a certain moment. It lets us cache information instead of having to query the database for every operation.

On the Resource Management Subsystem the Car Manager is the main component, and is in charge of handling everything that involves cars. The Car Handler keeps a list of all available cars and acts as an interface when a command needs to be sent to a car. The PGS Manager's task is to collect updates from the power grid stations about the plugs' availability and update the database accordingly.

2.3 Deployment View

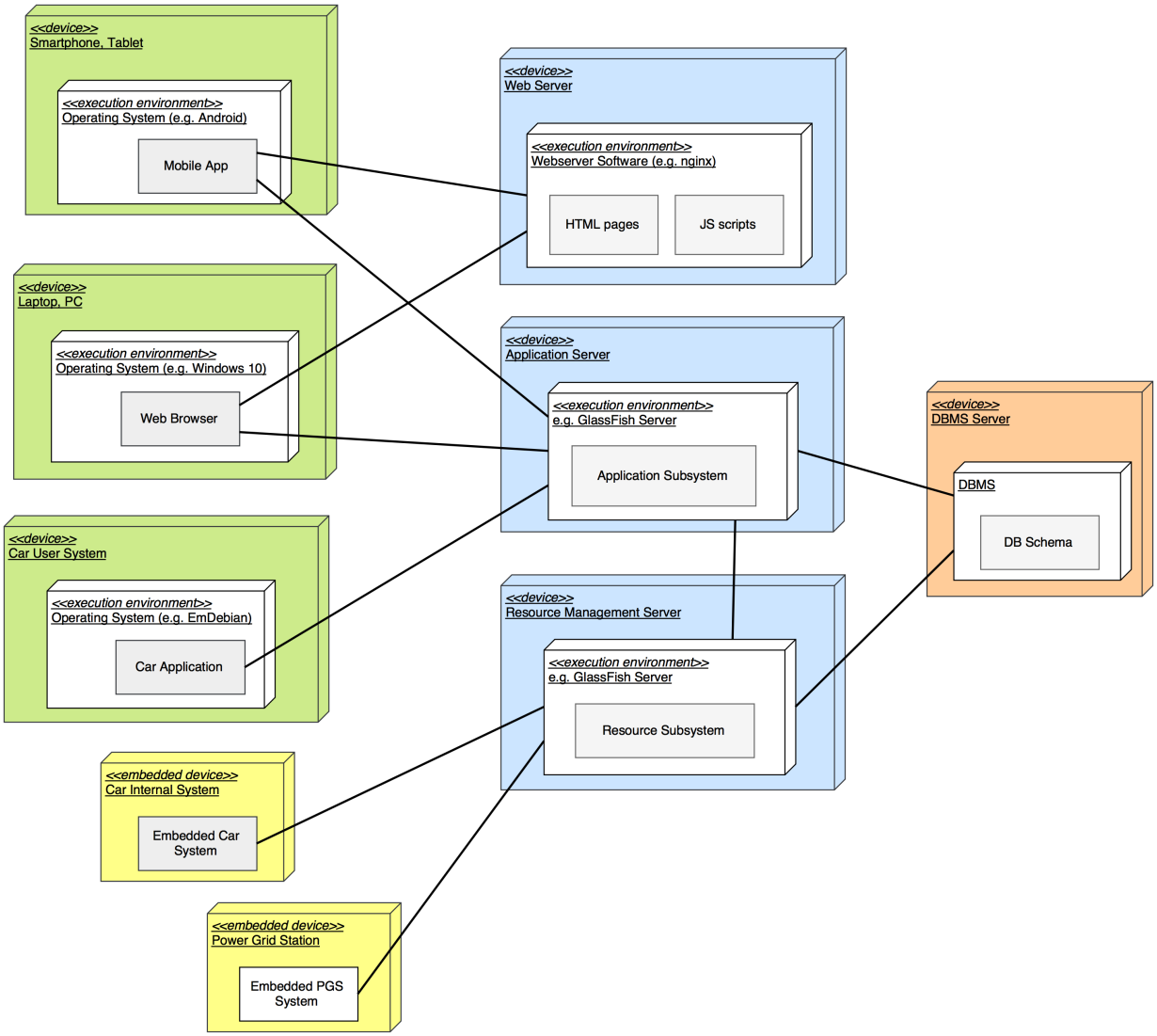

Figure 5: Deployment Diagram

2.4 Runtime View

2.4.1 Registration

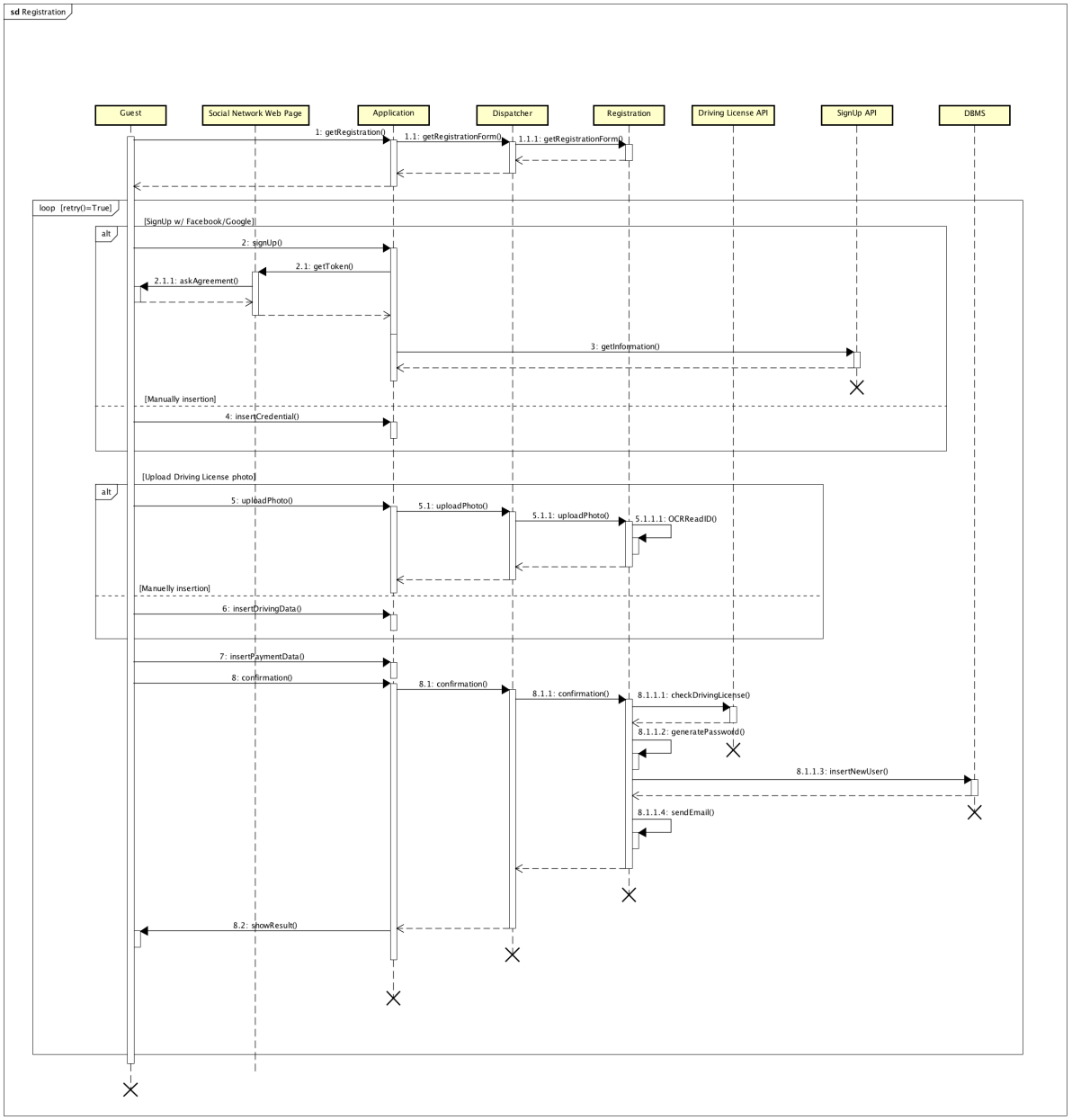

Figure 6: Registration

2.4.2 Car Search

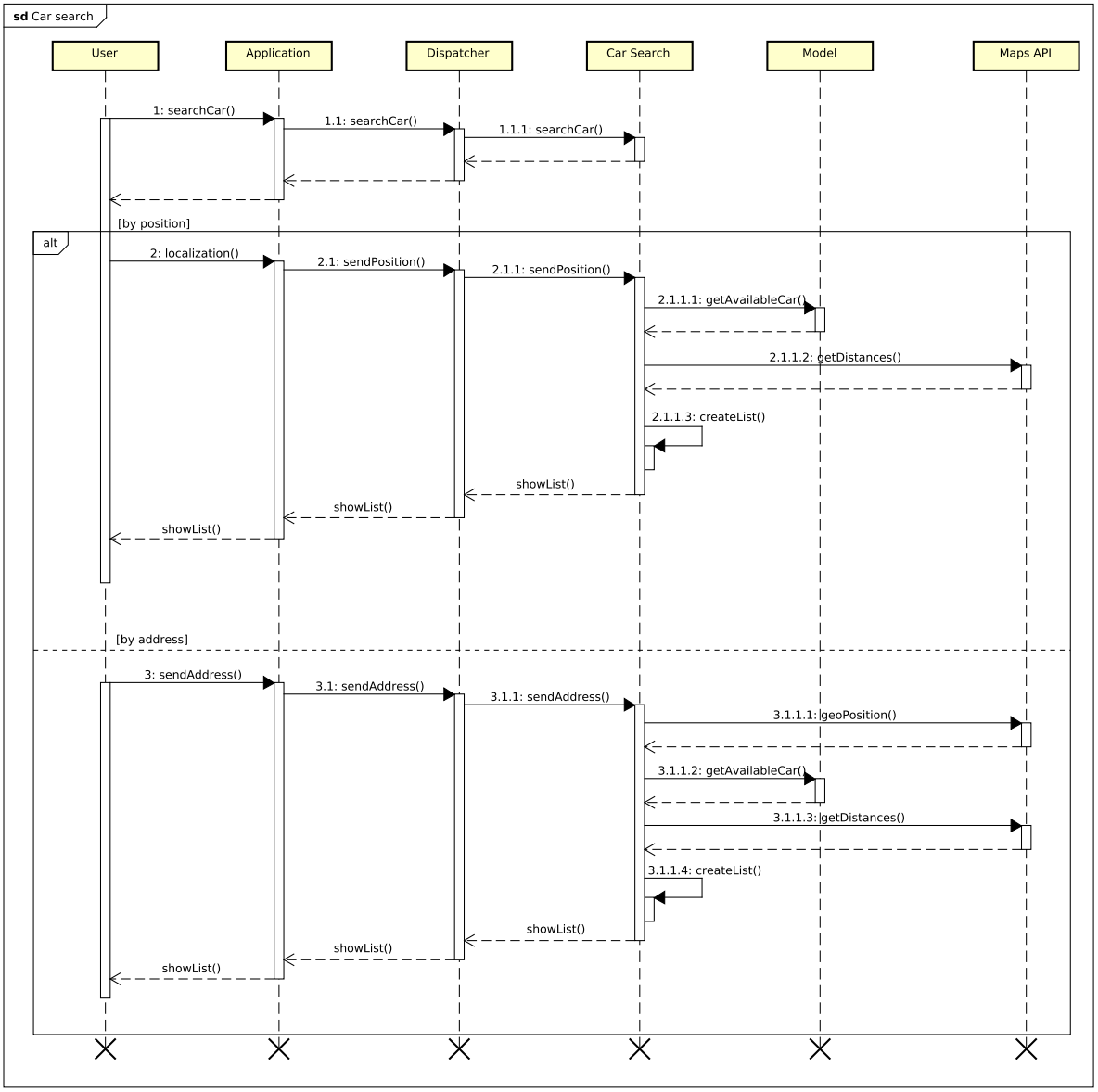

Figure 7: Car Search

2.4.3 Unlocking a car

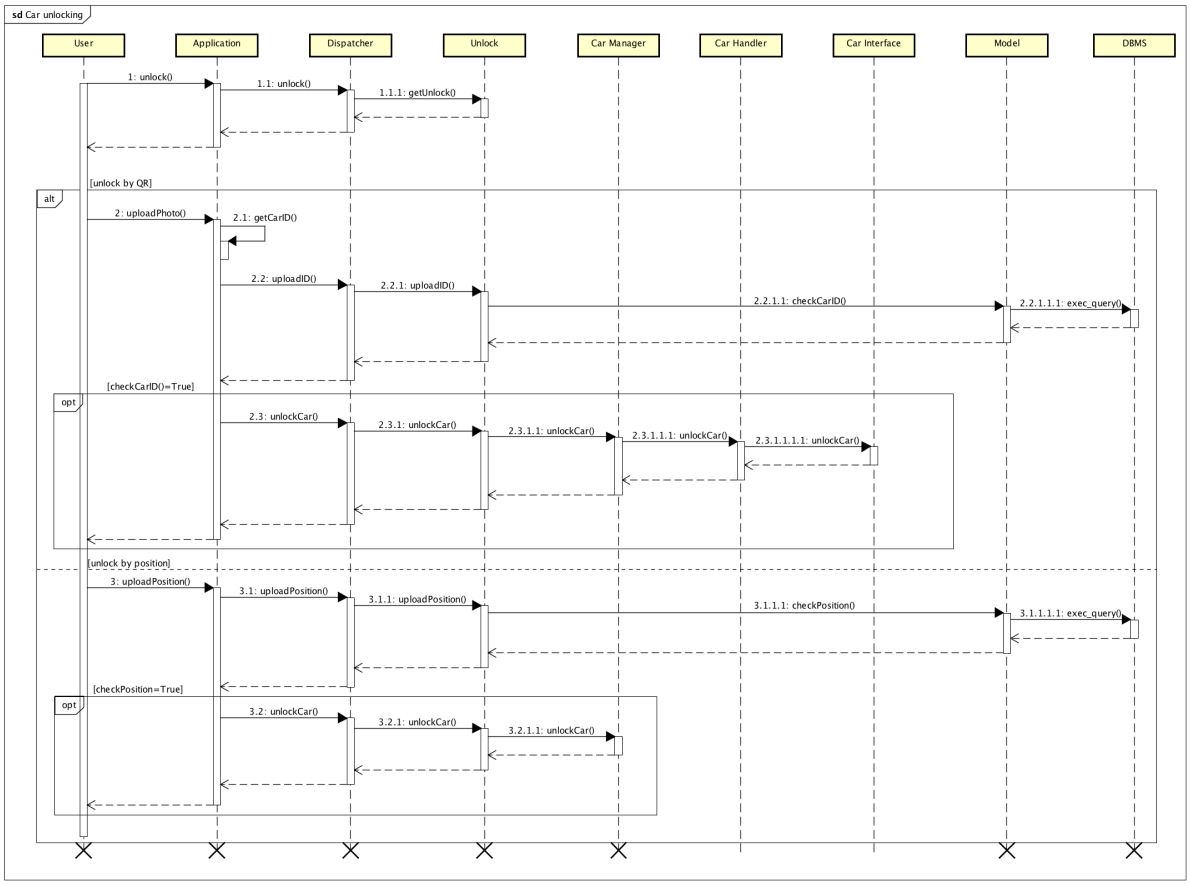

Figure 8: Car Unlocking

2.4.4 Payment

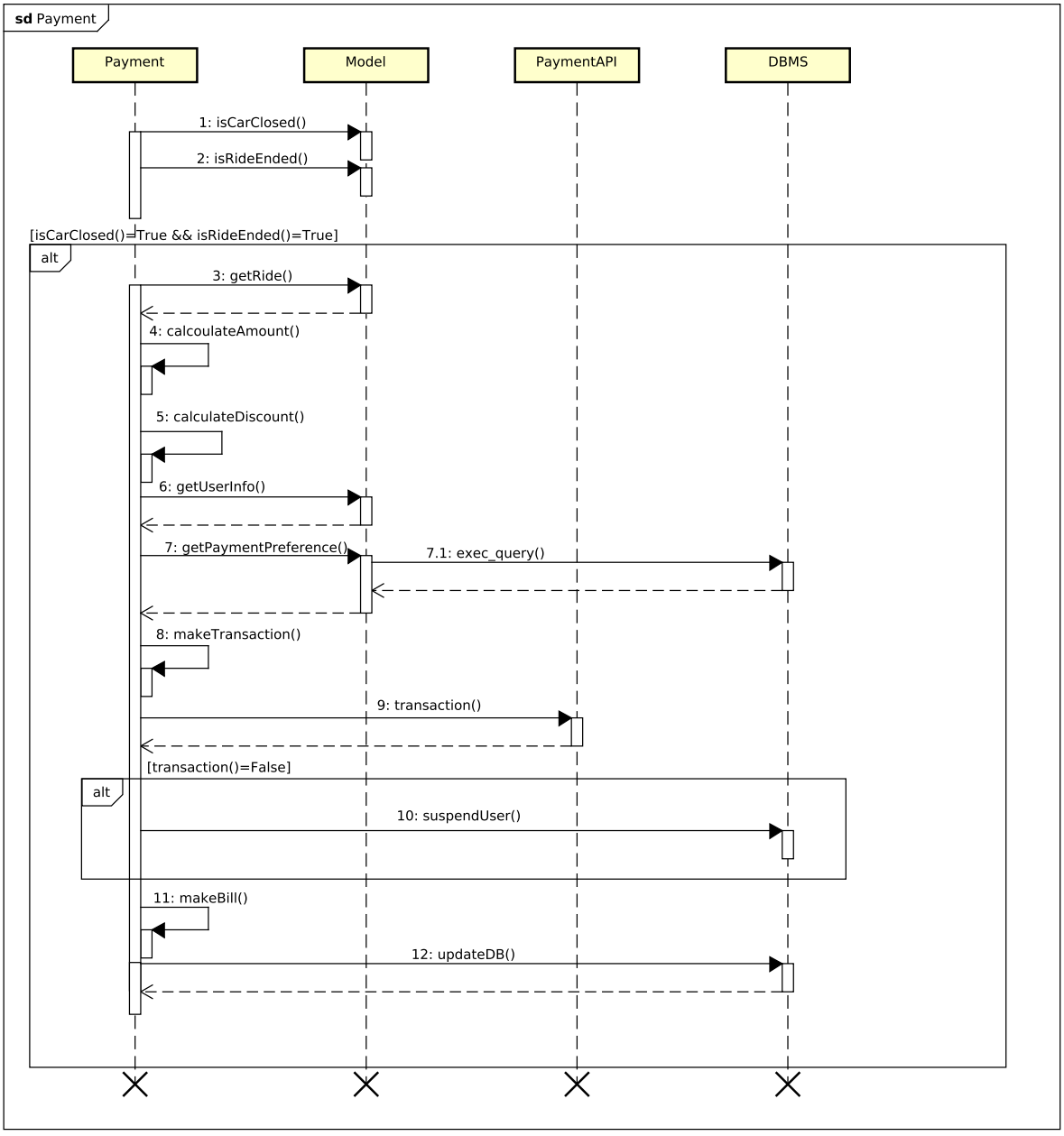

Figure 9: Payment

2.4.5 Setting a car as unavailable

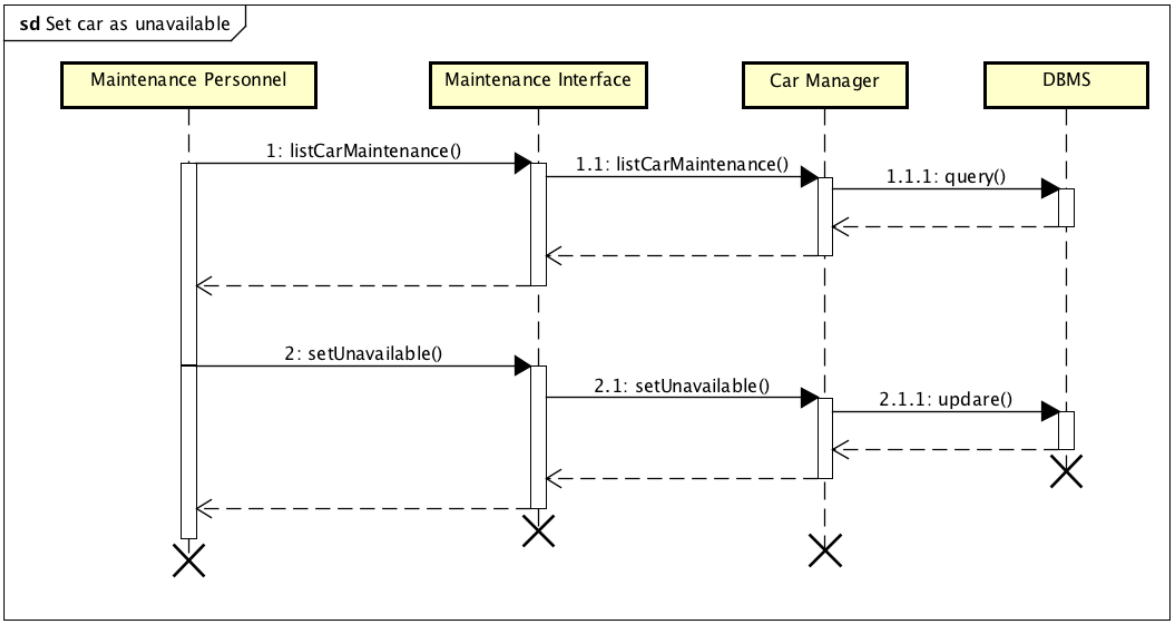

Figure 10: Setting a car as unavailable

2.5 Component Interfaces

This diagram shows how the interfaces of our system are connected, and which methods each of them must provide.

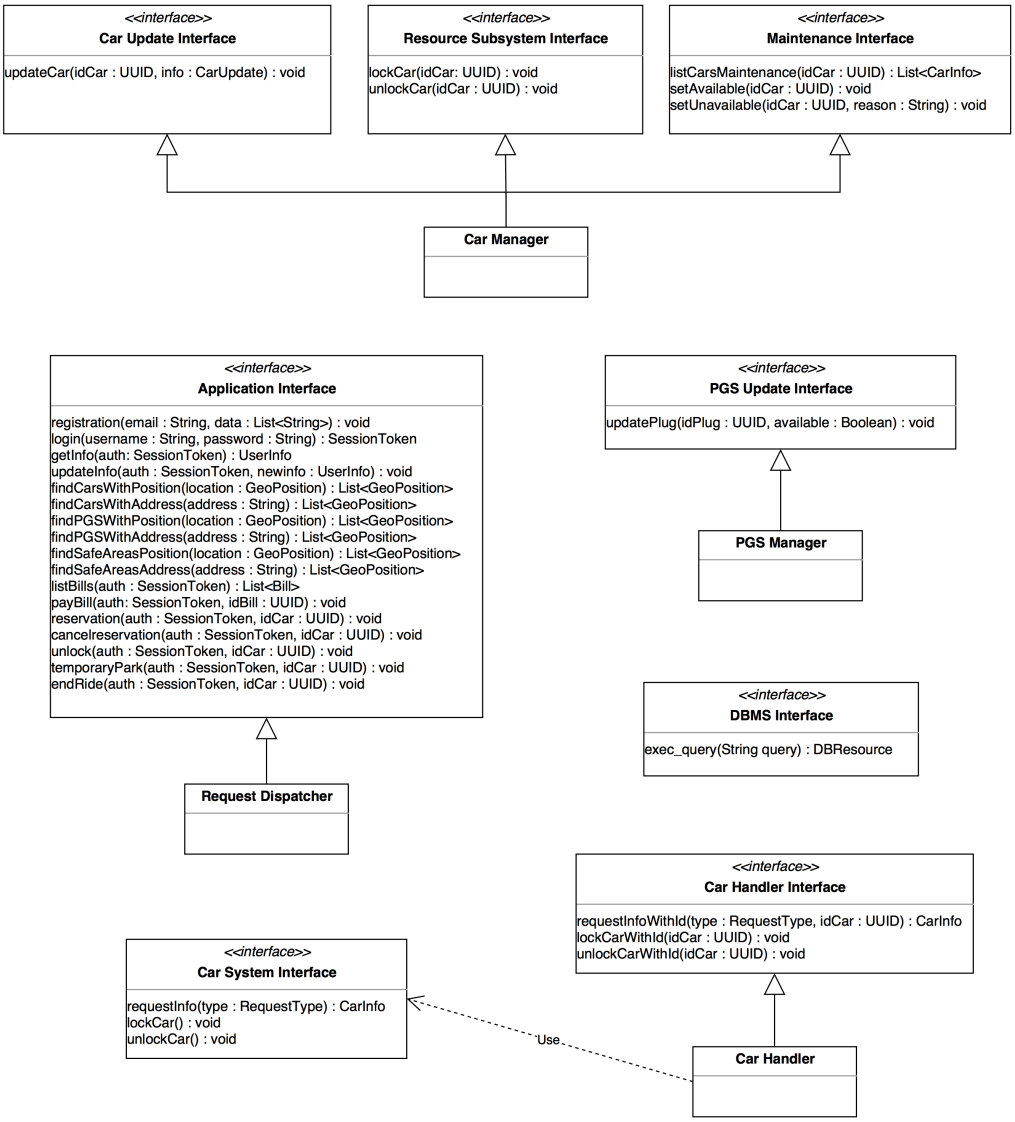

Figure 11: Component Interfaces

2.6 BCE Diagram

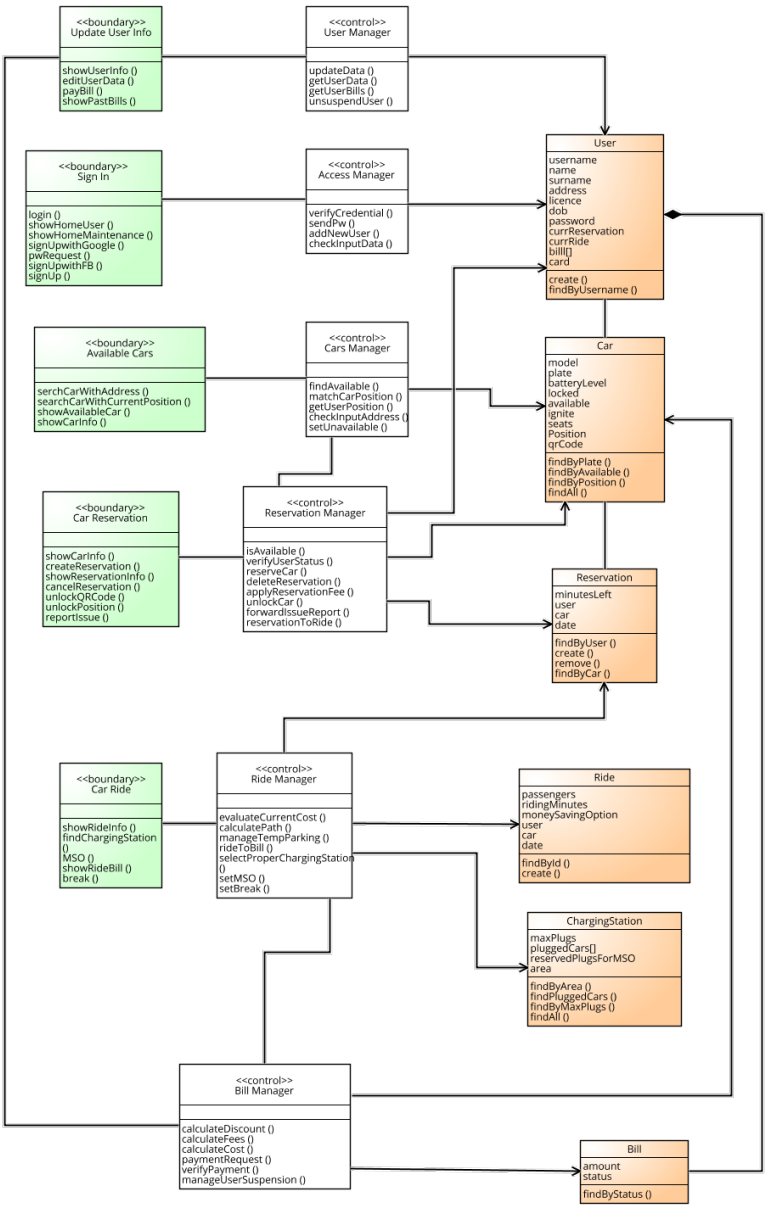

Figure 12: BCE Diagram corresponding to the UX defined in section 5

2.7 Selected Architectural Styles and Patterns

2.7.1 Architectural Style

- **Client-Server Architecture**: a network architecture where a client uses a service provided by a server. The advantages of using this type of architecture are:
  - Scalability and Upgradeability: changes can be made easily by upgrading the server alone. Also resource-intensive tasks can be left to the server without forcing clients to carry the load.
  - Accessibility: users can use the services without a need for a specialized device or client.
  - Security: servers have better control access and resources to make so that only authorized clients can manipulate or access data.
  - Maintenance: it's easy to repair, upgrade, replace or move a server without affecting clients.
- **Service Oriented Architecture**: SOA is a type of software architecture designed for web services in order to allow interoperability between different systems. The reasons for choosing this style are:
  - Flexibility and Scalability: every component is designed so that each of them is independent from the others, leaving space to develop each of them in the most appropriate way. Since every component is independent, it can be developed, tested and upgraded without affecting the others.
  - Reusability: since the components are decoupled from each other it's very easy to reuse them while developing the system.
- **Three Tiered Architecture**: the architecture structure is divided in three layers:
  1. The *presentation layer*, where data provided by the service is showed to the client. This layer communicates with the other layers to obtain the data.
  2. The *application logic layer*, where the logic of the application is actually executed through scripts and functions that elaborate the data.
  3. The *data layer*, the lowest layer of the architecture, the one that includes the database and where the data is stored and fetched.

  The reasons behind this choice are:
  - Scalability: since each tier is independent from the others it can be upgraded and expanded separately.
  - Performance: the presentation layer can for example cache the requests and results, reducing the load on lower layers.

2.7.2 Design Patterns

In this section we present some of the design patterns we followed while designing the system and its architecture.

- **Model, View, Controller (MVC)**: the MVC pattern separates the application in three parts to decouple the system's representation of data from how it is presented to the user (i.e. on a graphical interface). Our system has a Model that is updated to reflect the state of the system in any given moment. The Model is modified by the Dispatcher and the Application Subsystem components, that collectively work as a Controller to interact with and modify the Model. The clients (the web/mobile and the car applications) collect the data they need from the system and show it to the user in different ways depending on the application.
- **Facade**: following the Facade design pattern, our system provides a clear and simple interface to the external world, but each method is actually handled in a much more complex way inside the system. The interface hides the real complexity of the functions so that developers and users don't have to worry about it and can much more easily access and use them.

2.7.3 Protocols

For the interfaces that the system must offer to external components we chose to follow the REST paradigm. Clients will be able to send HTTP requests to the available endpoints, and will receive JSON objects as responses.

Clients will be authenticated through session tokens (like for example OAuth), that will be sent along with the HTTP request in the Authentication header when needed.

All the requests to the API must be protected with TLS.

The responses to requests will also be accompanied with a HTTP response status code, with the following meaning:

- `200 OK` – the request was successful.
- `400 Bad Request` – the request was not successful, or badly formed.
- `401 Authorization Required` – the authentication token is missing or not correct.
- `500 Internal Server Error` – there was a problem on the server, retrying later could be useful.

In case of error the response will provide a JSON object with an `error` field containing a description of the problem.

Each function that our system offers will have a dedicated endpoint, all of which are described in the following sections.

Application For some of these requests there is no need to pass the username or car id, as the server can infer these pieces of information from the authentication token.

- Registration:

Description	Sign up to PowerEnJoy with an email, a licence and personal data
Endpoint	`/registration`
Verb	POST
Parameters	`email`, `licence_id` or `licence_photo`, `name`, `dob`, `address`, `payment_method`
Response	`200 OK` if successful.

- Login:

Description	Login to the application
Endpoint	`/login`
Verb	POST
Parameters	`username`, `password`
Response	`auth_token`

- Get user Info:

Description	Get information related to the current user
Endpoint	`/userinfo`
Verb	GET
Parameters	`auth_token`
Response	`username`, `email`, `licence_id`, `date_of_birth`, `address`, `curr_reservation`

- Update the user's data:

Description	Update the user's personal data to new ones
Endpoint	`/userinfo`
Verb	PATCH
Parameters	`auth_token`, optional: `email`, `password`, `address`, `payment_method`
Response	`200 OK` if successful

- Find cars with position:

Description	Lists available cars close to the specified position
Endpoint	`/findcars/position/{lat}/{long}`
Verb	GET
Parameters	None
Response	Set of `(car_id, lat, long, battery)`

- Find cars with address:

Description	Lists available cars close to the specified address
Endpoint	`/findcars/address/{address}`
Verb	GET
Parameters	None
Response	Set of `(car_id, lat, long, battery)`

- Find safe areas with position:

Description	Lists safe areas close to the specified position
Endpoint	`/findsafeareas/position/{lat}/{long}`
Verb	GET
Parameters	None
Response	Set of `(lat, long)`

- Find safe areas with address:

Description	Lists safe areas close to the specified address
Endpoint	`/findsafeareas/address/{address}`
Verb	GET
Parameters	None
Response	Set of `(lat, long)`

- Find power grid stations with position:

Description	Lists power grid stations close to the specified position
Endpoint	`/findstations/position/{lat}/{long}`
Verb	GET
Parameters	None
Response	Set of `(lat, long)`

- Find safe areas with address:

Description	Lists safe areas close to the specified address
Endpoint	`/findstations/address/{address}`
Verb	GET
Parameters	None
Response	Set of `(lat, long)`

- Make a reservation:

Description	Creates a new reservation for a car
Endpoint	`/reservation`
Verb	POST
Parameters	`auth_token`, `carId`
Response	`200 OK` if successful

- Cancel a reservation:

Description	Cancels the user's current reservation if there's one
Endpoint	`/reservation`
Verb	DELETE
Parameters	`auth_token`
Response	`200 OK` if successful

- Unlock a car:

Description	Unlocks the car that is currently reserved
Endpoint	`/unlock`
Verb	POST
Parameters	`auth_token`, `unlock_id`
Response	`200 OK` if successful

- Temporary parking:

Description	Puts the car in temporary parking
Endpoint	`/parking`
Verb	POST
Parameters	`auth_token`
Response	`200 OK` if successful

- End ride:

Description	Concludes the current ride
Endpoint	`/endride`
Verb	POST
Parameters	`auth_token`
Response	`200 OK` if successful

- List bills:

Description	Shows all the user's bills
Endpoint	`/bills`
Verb	GET
Parameters	`auth_token`
Response	Set of `(bill_id, date, amount, payment_status)`

- Pay bill:

Description	Triggers a transaction to pay an outstanding bill if the user has one
Endpoint	`/paybill`
Verb	POST
Parameters	`auth_token`, `bill_id`
Response	`200 OK` if successful

Maintenance The maintenance API uses Basic HTTP Authentication to provide access. Those in charge of maintenance will be have a set of keys to use the API functionalities.

- Find cars needing maintenance:

Description	Lists all the cars in the systems with problems to solve
Endpoint	`/maintenancelist`
Verb	GET
Parameters	`maintenance_key`
Response	Set of `(car_id, lat, long, description)`

- Set car as available:

Description	Sets a car as available
Endpoint	`/setcar/{car_id}`
Verb	PUT
Parameters	`maintenance_key`, `availability`
Response	`200 OK` if successful

- Set car as unavailable:

Description	Sets a car as unavailable
Endpoint	`/setcar/{car_id}`
Verb	PUT
Parameters	`maintenance_key`, `availability`, `description`
Response	`200 OK` if successful

Sensors Sensors too use Basic HTTP Authentication. Each vehicle and power grid station will have a unique key to use the API and at the same time be identified. The TLS protocol will prevent malicious actors from sniffing the keys from the network.

- Update from car:

Description	Submits updates about the car to the server
Endpoint	`/updatecar`
Verb	POST
Parameters	`car_key`, `lat`, `long`, `speed`, `direction`, `engine_status`, `lock_status`, `passengers`
Response	`200 OK` if successful

- Update from power grid station:

Description	Submits updates about the plugs in the station to the server
Endpoint	`/updateplug`
Verb	POST
Parameters	`pgs_key`, `plugs` (set of `(plug_id, availability)`)
Response	`200 OK` if successful

Car Internal Systems Cars must receive commands from the system, from example to unlock or lock the doors. The cars' embedded system won't offer a RESTful API, but must always have an open port listening for incoming commands from the central servers. Incoming connections will be authenticated using public/private key pairs.

3 Other Design Decisions

For the development of the software for this project we have chosen to use the **Java Enterprise Edition** stack.

We have many different reasons for this choice:

- JEE is known, stable and well-established. We would have no problem finding support if needed, and we would be sure that there won't be major, code-breaking changes in the specification of the language that would require a lot of effort to fix.
- it includes several execution environments, like Glassfish and Tomcat, that will make deployment much easier and streamlined;
- it's portable: as long as the operating system supports the Java Virtual Machine our application will be able to run. The JVM is available on the great majority of modern OSs, like Windows or the most common Linux distros;
- it supports various testing frameworks, like JUnit and Arquillian;
- finally, it provides specific API and functions for concurreny and for executing large-scale, multi-tiered, scalable network applications.

4 Algorithm Design

In this section we are going to present potential algorithms to use when implementing the application.

We analyze two different functionalities that we want to implement:

- Calculating the discount to apply to a bill at the end of a ride;
- Finding and choosing a charging station to propose to the user when the Money Saving Option is activated.

We decided to use Java-like pseudo-code as it is clear and easy to understand, and at the same time a language that we feel comfortable using.

4.1 Discount calculation

```
1  package powerEnJoy;
2
3  import java.util.List;
4
5  import graphHopper.GraphHopper;
6
7  public class Discounts
8  {
9      private List<PowerGridStation> pgsList;
10     private final static int MAXDISTANCE = 3000;
11
12     public int calculatesDiscount(int amount, Car car, Ride ride)
13     {
14         int minDistance = 1000000000;
15         int distance = 0;
16         for(PowerGridStation pgs : pgsList)
17         {
18             distance = GraphHopper.distance(pgs.getPosition(), car.getPosition());
19             if(distance < minDistance)
20                 minDistance = distance;
21         }
22
23
24         int discountPercentage = 0;
25
26         if(car.getPassengers() > 1)
27             discountPercentage += 10;
28         if(car.getBattery() > 50)
29             discountPercentage += 20;
30         if(car.isCharging())
31             discountPercentage += 30;
32         if(car.getBattery() < 20 || minDistance > MAXDISTANCE)
33             discountPercentage -= 30;
34         if(ride.isMso() && (ride.getChosenPGS().getPosition().equals(car.getPosition())))
35             discountPercentage += 10;
36
37         int discount = amount * (discountPercentage/100);
38
39         return (amount - discount);
40     }
41 }
```

We sum the individual discounts before applying them, to avoid one discount lessening the effects of others like it would happen if we applied them in sequence. This makes it so every discount is always worth the same and users aren't penalized for obtaining more than one discount at a time.

NB: the discount can vary from a maximum of 70% to a minimum of -30% (a 30% overcharge).

The time complexity of this algorithm is $\theta(n)$, where n is the number of power grid stations saved in the system, because we need to cycle through every station to find the distance from the car to the closest station.

If necessary this solution can be marginally improved, for example by subdividing the stations in zones so that we can check only the ones in which the car is currently located.

4.2 Finding a station for the Money Saving Option

```
1  package powerEnJoy;
2
3  import graphHopper.*;
4
5  import java.util.ArrayList;
6  import java.util.HashMap;
7  import java.util.List;
8  import java.util.Map;
9  import java.util.Map.Entry;
10
11 public class MoneySavingOption
12 {
13     private final static int TIMEDISTANCE = 300;    //300 seconds = 5 minutes
14     private final static int SPACEDISTANCE = 500;   //500 metres
15
16     private List<PowerGridStation> pgsList;
17     private List<Car> availableCar;
18
19     public PowerGridStation moneySavingOption(String address)
20     {
21         Map<PowerGridStation, Integer> count = new HashMap<>();
22         GeoPosition destination = GraphHopper.geocoding(address);
23         List<PowerGridStation> accettablepgs = new ArrayList<>();
24
25         //looks for the power grid stations reachable in a reasonable time
26         for(PowerGridStation pgs : pgsList)
27             if(GraphHopper.isochrone(destination, pgs.getPosition()) < TIMEDISTANCE)
28                 accettablepgs.add(pgs);
29
30         //estimates the "density" of the cars near a power grid station
31         for(PowerGridStation pgs : accettablepgs)
32         {
33             int i = 0;
34
35             for(Car car : availableCar)
36                 if(GraphHopper.distance(car.getPosition(), pgs.getPosition()) <
                       SPACEDISTANCE)
37                     i++;
38
39             count.put(pgs, i);
40         }
41
42         PowerGridStation chosenPGS = null;
43         int min = 1000000000;
44
45         //finds the power grid station with the minimum "density"
46         for(Entry<PowerGridStation, Integer> entry : count.entrySet())
47             if(entry.getValue() < min)
```

```
48                      chosenPGS = entry.getKey();
49
50              return chosenPGS;
51          }
52      }
```

Because of our requirements this algorithm must take in account the distribution of cars and make it as uniform as possible.

To realize this, our algorithm:

- takes the user's final destination as input;
- looks for all the power grid stations reasonably close to the destination;
- chooses among all the acceptable stations the one that would make distribution more balanced (the one with the least cars nearby).

The time complexity of this algorithm is $\theta(n)+\theta(m \cdot c)+\theta(m)$ or $\theta(n+m(1+c))$, where n is the number of charging stations in the system, m the number of acceptable stations, and c the number of cars close to them.

This can also if needed be improved by subdividing the stations in zones. Another option could be saving the number of cars near every station in advance, and updating it every time a car is parked nearby or driven away, without having to recompute the value every time it is needed.

5 User Interface Design

5.1 Mockups

Mockups for the user interfaces can be found on the RASD at section 3.1.1.

5.2 UX Diagrams

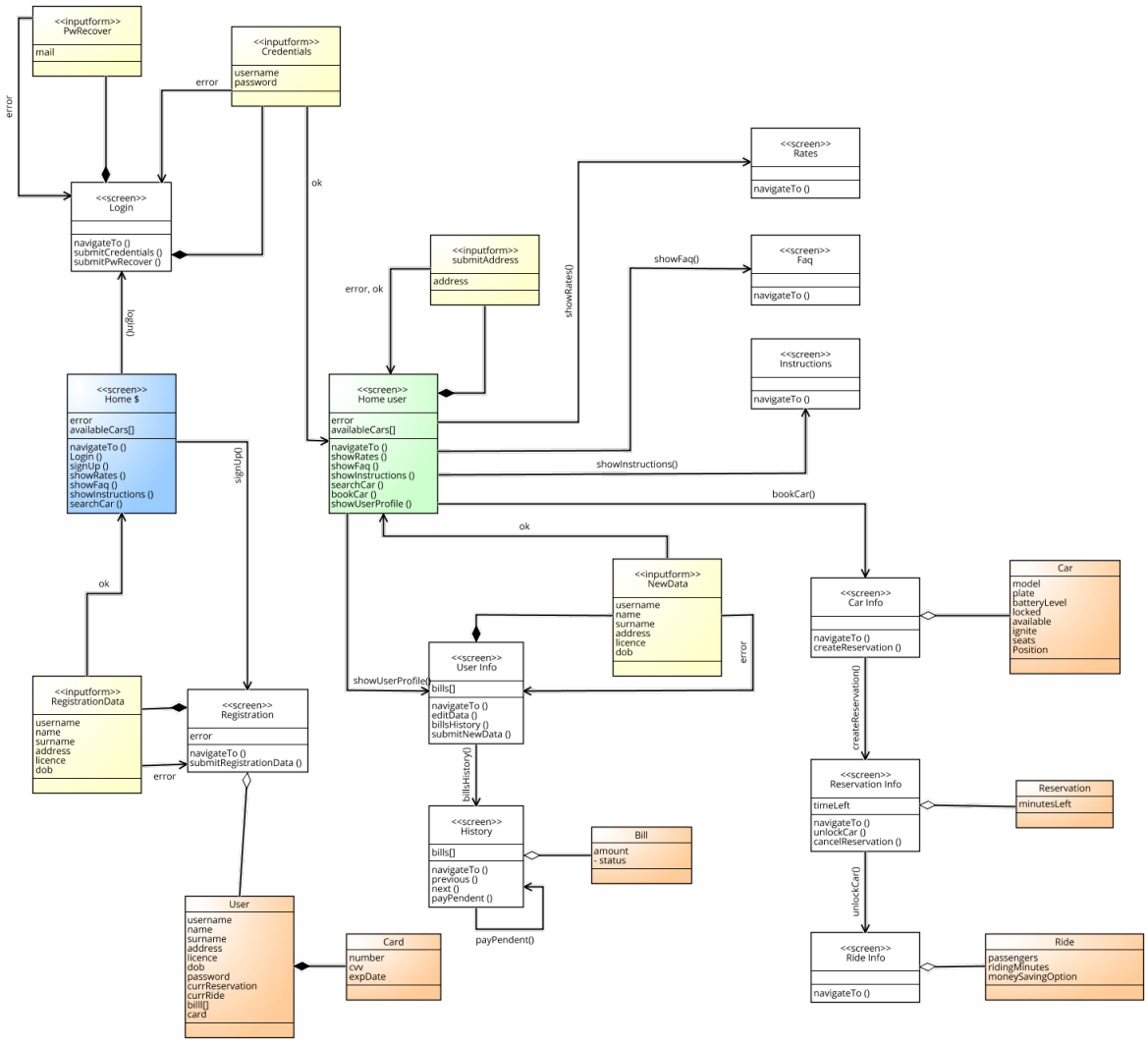

Figure 13: UX Diagram for the web application

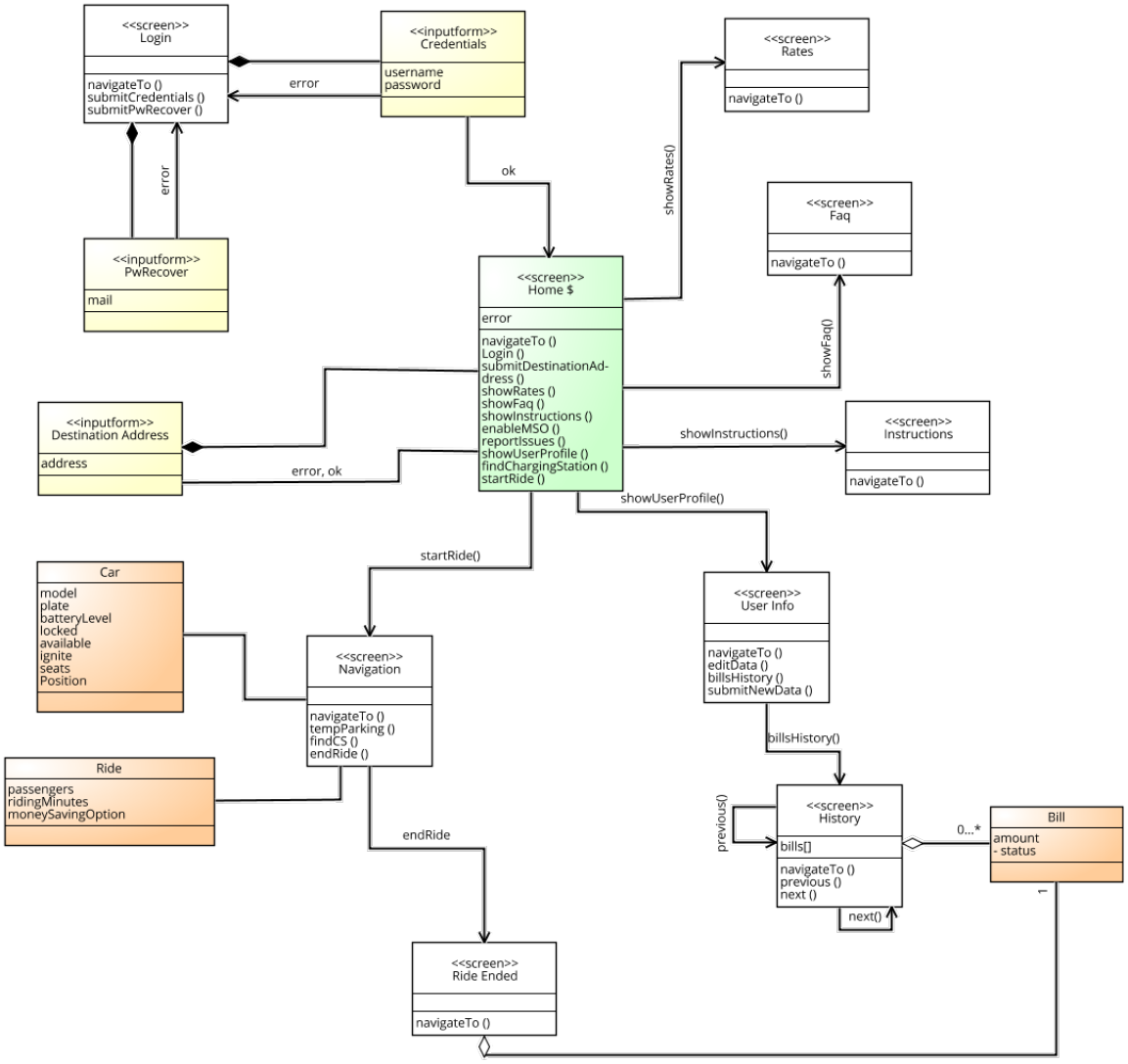

Figure 14: UX Diagram for the car application

6 Requirements Traceability

To make sure that every component of our architecture is actually needed, we now show which components are used to fulfill each goal that we identified in the Requirement Analysis and Specification Document.

[G1] Guests must be able to register as users by choosing a username and providing their personal data, driving license and payment information. They will receive a password at the email address they specified.

- Application Subsystem
- Request Dispatcher
- Registration Component
- Signup API
- Driving Licence API
- Model
- DBMS

[G2] Users must be able to login with the username/email inserted and the password received on registration.

- Application Subsystem
- Request Dispatcher
- Account Manager
- Model
- DBMS

[G3a] Users must be able to find the location and battery charge of all the available cars within a certain distance from their current location.

- Application Subsystem
- Request Dispatcher
- Car Search Component
- Maps API
- Model
- DBMS

[G3b] Users must be able to find the location and battery charge of all available cars around a specified location.

- Application Subsystem
- Request Dispatcher
- Car Search Component

- Maps API
- Model
- DBMS

[G4] Users must be able to reserve a car for up to one hour before they pick it up. If they don't take the car before the time expires they are charged a fixed fee of 1 EUR.

- Application Subsystem
- Request Dispatcher
- Reservation Manager
- Payment Component
- Model
- DBMS

[G5] Users must be able to cancel a reservation if they decide to not actually use the car.

- Application Subsystem
- Request Dispatcher
- Reservation Manager
- Model
- DBMS

[G6] A user that reaches a car reserved by them must have a way to tell the system they're nearby, so that the system unlocks the car and the user may enter.

- Application Subsystem
- Request Dispatcher
- Unlocking Component
- Resource Management Subsystem
- Car Manager
- Car Handler
- Car Internal System
- Model
- DBMS

[G7] The system must charge the user who reserved the car from the moment the engine is ignited after unlocking it. The user is charged for a given amount of money per minute.

- Car Internal System

- Resource Management Subsystem
- Car Manager
- DBMS

[G8] The system must allow the user to see the amount they're being charged through a screen on the car.

- Car Application
- Application Subsystem
- Request Dispatcher
- Ride Manager
- Model
- DBMS

[G9] The system must stop charging the user as soon as the car is parked in a safe area and the user exits the car notifying the system that they ended their ride.

- Car Internal System
- Resource Management Subsystem
- Car Manager
- Application Subsystem
- Request Dispatcher
- Ride Manager
- Payment
- Payment API
- Model
- DBMS

[G10] Users must be able to leave a car without losing the reservation by informing the system that they're only temporarily parking; in this case the system continues on charging the user. The user should be able to end the ride without coming back to the car if they so choose.

- Application Subsystem
- Request Dispatcher
- Parking Component
- Model
- DBMS

[G11] The system must lock the car automatically after the user exits the car.

- Resource Management Subsystem
- Car Manager
- Car Handler
- Car Internal System

[G12] The system must provide a money saving option to the user, proposing a suitable power grid station near the user's final destination. The user will get a discount if they park the car there and plug it in the power grid.

- Application Subsystem
- Request Dispatcher
- PGS Search Component
- Maps API
- DBMS
- PGS Internal System
- PGS Manager

[G13] The stations proposed by the system with the money saving option must be chosen in a way that ensures a uniform distribution of cars in the city.

- (None)

[G14] The system must apply a discount of 10% on the last ride if the user took at least two other passengers onto the car.

- Car Internal System
- Resource Management Subsystem
- Car Manager
- Model
- DBMS
- Application Subsystem
- Ride Manager
- Payment Component

[G15] The system must apply a discount of 20% on the last ride if the car is left with more than 50% of remaining battery charge.

- Car Internal System
- Resource Management Subsystem
- Car Manager
- Application Subsystem
- Ride Manager

- Payment Component
- Model
- DBMS

[G16] The system must apply a discount of 30% on the last ride if the car is left at special parking areas where they can be recharged, and the user takes care of plugging the car into the power grid.

- Car Internal System
- Resource Management Subsystem
- Car Manager
- Application Subsystem
- Request Dispatcher
- Ride Manager
- Payment Component
- PGS Internal System
- PGS Manager
- Model
- DBMS

[G17] The system must charge 30% more if the car is left at more than 3 km from the nearest power grid station or with less than 20% remaining battery charge.

- Car Internal System
- Resource Management Subsystem
- Car Manager
- Application Subsystem
- Request Dispatcher
- Ride Manager
- Payment Component
- Maps API
- Model
- DBMS

[G18] Users with outstanding bills cannot have access to the cars and reservations until the bills are paid.

- Application Subsystem
- Request Dispatcher
- Reservation Manager

- Model
- DBMS

[G19] Users with outstanding bills should have a way to pay them.

- Application Subsystem
- Request Dispatcher
- Account Manager
- Payment
- Model
- DBMS

[G20] Available cars should actually be available to the user and in good conditions (i.e. not dirty, damaged, and/or with low battery).

- Application Subsystem
- Request Dispatcher
- Report Manager
- Resource Management Subsystem
- Car Manager
- Model
- DBMS

[G21] Users must be able to easily find power grid stations.

- Application Subsystem
- Request Dispatcher
- PGS Search Component
- Safe Areas Search Component
- Model
- DBMS

Every component in our architecture is used by at least one goal.

6.1 Software and tools used

- Signavio for the UX and BCE diagrams
- Astah and draw.io for the rest of the diagrams
- GitHub for version control
- TEXStudio as a LATEXeditor

7 Effort Spent

- Pietro Ferretti: 40 hours of work
- Nicole Gervasoni: 15 hours of work
- Danilo Labanca: 20 hours of work

8 Revisions

8.1 Changelog

- DD v1.0, published on December 11, 2016
- DD v1.1, published on January 15, 2016
  - Added details about the Model in the Component View.
  - Changed the Login Component name to Account Manager to better describe its role in the system.
- DD v1.2, published on January 22, 2016
  - Moved BCE to the Architectural Design section
  - Added the section *Other Design Decisions*