

Luca Scannapieco - 877145

Andrea Pasquali - 808733

Emanuele Torelli - 876210

11/12/2016

PowerEnJoy

Design Document

Table of contents

[1. Introduction 3](#_Toc468627856)

[1.1. Purpose 3](#_Toc468627857)

[1.2. Scope 3](#_Toc468627858)

[1.3. Definitions, acronyms, abbreviations 3](#_Toc468627859)

[1.4. Document structure 3](#_Toc468627860)

[2. Architectural design 3](#_Toc468627861)

[2.1. Overview 3](#_Toc468627862)

[2.2. High level components and their interaction 3](#_Toc468627863)

[2.3. Component view for server 4](#_Toc468627864)

[2.4. Database structure 5](#_Toc468627865)

[2.5. Deployment view 6](#_Toc468627866)

[2.6. Runtime view 7](#_Toc468627867)

[2.6.1. See available cars 7](#_Toc468627868)

[2.6.2. Make a reservation 8](#_Toc468627869)

[2.6.3. Start a ride 9](#_Toc468627870)

[2.6.4. Finish a ride 10](#_Toc468627871)

[2.7. Component interfaces 11](#_Toc468627872)

[2.8. Selected architectural styles, patterns and design decisions 11](#_Toc468627873)

[3. Algorithm design 11](#_Toc468627874)

[4. User interface design 11](#_Toc468627875)

[5. Requirements traceability 11](#_Toc468627876)

[6. Other info 11](#_Toc468627877)

[6.1. Reference documents 11](#_Toc468627878)

[6.2. Used tools 11](#_Toc468627879)

[6.3. Hours of work 11](#_Toc468627880)

[6.4. Changelog 11](#_Toc468627881)

# 1. Introduction

## 1.1. Purpose

## 1.2. Scope

## 1.3. Definitions, acronyms, abbreviations

## 1.4. Document structure

# 2. Architectural design

## 2.1. Overview

The PowerEnJoy service is implemented as a common client-server application, in which the offered services are essentially three:

* User interface
* Application logic
* Database

Each of these logic services is placed in the corresponding physic layer, the result is that we adopted a three-tier architecture.

We provide a mockup to better understand the structure of the PowerEnJoy service:

MOCKUP

The user interface has two different implementations, one is constituted by a web app that can be executed on a modern browser, and the other one is the PowerEnJoy mobile application. Furthermore, during a ride, there is another component with which the user interacts: the screen of the car. On the other hand, the assistance coordinator has an interface built ad hoc to performs his work, in fact the tasks that the coordinator must do are forbidden to the common users.

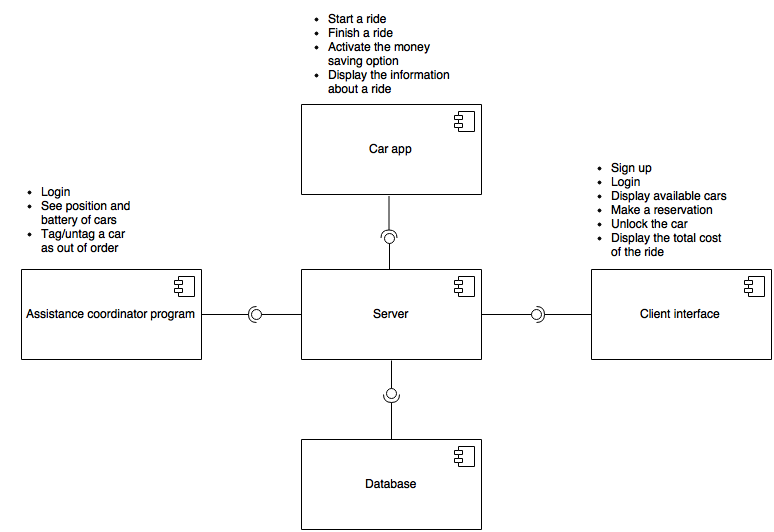
The core of our architecture is the main central server, where is provided the logic necessary to the system to run properly. In addition to the central server we need an external web server to run the PowerEnJoy web app properly. In fact, due to ensure the highest level of security and to maintain the client apps light, we decided to maintain the clients as thin as possible.

Separating the layer for the application logic and the database we can ensure a high modularity of the system, and for example the company can decide to move one of the layers (or both) to a cloud service, for example to amazon AWS where it would have dedicated cloud servers with load balance for database and other for application logic on demand.

## 2.2. High level components and their interaction

In the following diagram there are shown the main components of the PowerEnJoy system, presented in this section at the higher level possible.

For each of the three high level components of the tier 1 (the client app, the car app and the assistance coordinator app) we list the functionalities that the system must ensure; we derived those functionalities from the use cases diagram presented in the RASD.



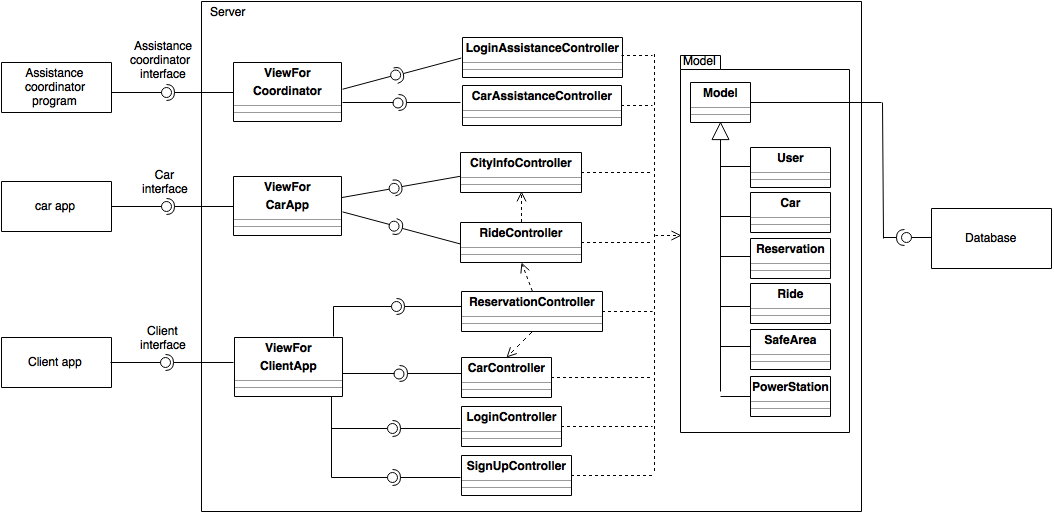
## 2.3. Component view for server

Now we can proceed with a further description of the central server, which is the core of the whole system, by analysing the structure of the software and how it works.

The various functionalities are exploited by multiple controllers, each of them implements the methods provided by the corresponding interface.

Each controller provides an interface to a view, in fact there are three views one for high level component of the tier 1. The views are important to guarantee the correct forwarding of messages from the client, so that the correct method of the correct controller is invoked.

In the server there is also a model that represents the structure of the data in the database, the structure of the database is described in the next section.



Here is a short explanation of the tasks performed by each controller:

* LoginAssistanceController: checks the info provided by the assistance coordinator during the login phase
* CarAssistenceController: collects all the information of the cars in order to display them to the assistance coordinator
* CityInfoController: collects all the information of the city (e.g. safe areas, power stations)
* RideController: collects the information about a ride
* ReservationController: collects the information about a registration
* CarController: collect the information about the car, changes the availability tag of the cars
* LoginController: checks the info provided by the users during the login procedure
* SignUpController: checks the info provided by the users during the sign up procedure

## 2.4. Database structure

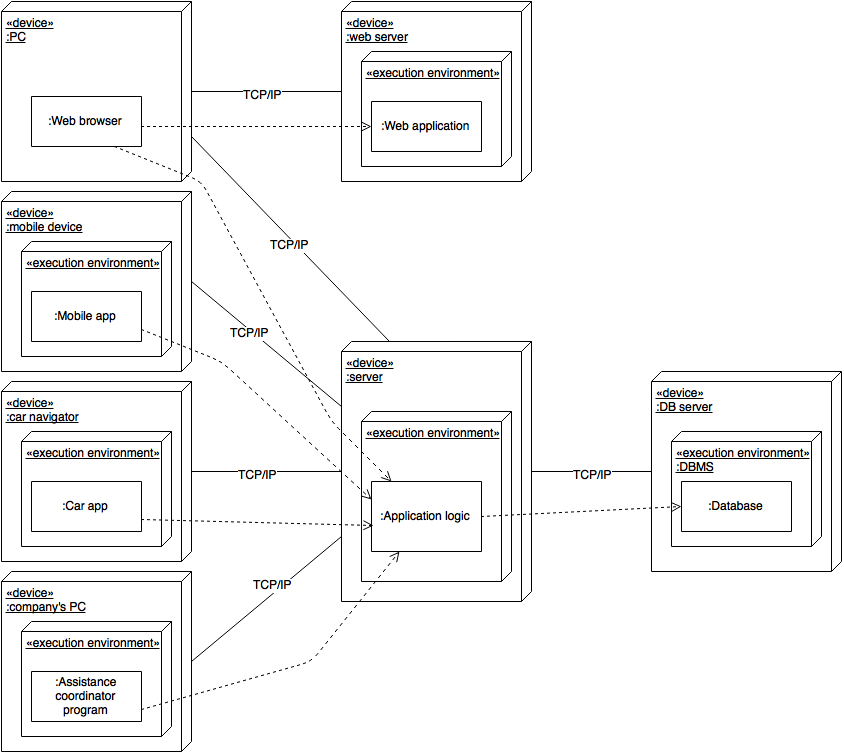
* Spiega il database (class diagram)

CLASS DIAGRAM DB

## 2.5. Deployment view

In this section there is a deployment diagram to explain the physical structure of the system. As mentioned before, for the tier 1 we have multiple devices in which the web app can be run, as well as multiple mobile apps for PowerEnJoy usage. There are also the car screens which are present in every PowerEnJoy car. The company features also a PC on which there is installed the program used by the assistance coordinator to do his tasks. All the components of the tier 1 communicate with the central server, and in addition to that the web application needs another server to communicate with to run the logic in the browser. The central server communicates with a database to collect all the data required by the system.

The components of our system communicate among them by using the TCP/IP protocol.



## 2.6. Runtime view

### 2.6.1. See available cars

C:\Users\Emanuele\AppData\Local\Microsoft\Windows\INetCacheContent.Word\DD sequence diagram AvailableCarVisulization.png

The request is handled by the car manager who extracts all the cars from the database and sends them back to the clientApp. The ClientApp has an internal render that is able to filter the cars both on the user position and on a given position inserted by the user.

### 2.6.2. Make a reservation



The client sends the reservation request to the reservation manager through one of the method offered by its view. The resource manager needs to ask to the car manager whether the reserved car is still available: if it is, a new reservation will be instantiated by the reservation manager that will also set the car and the user of the new reservation and start the reservation timer. If the car is no longer available an error will be notified to the clientApp.

### 2.6.3. Start a ride

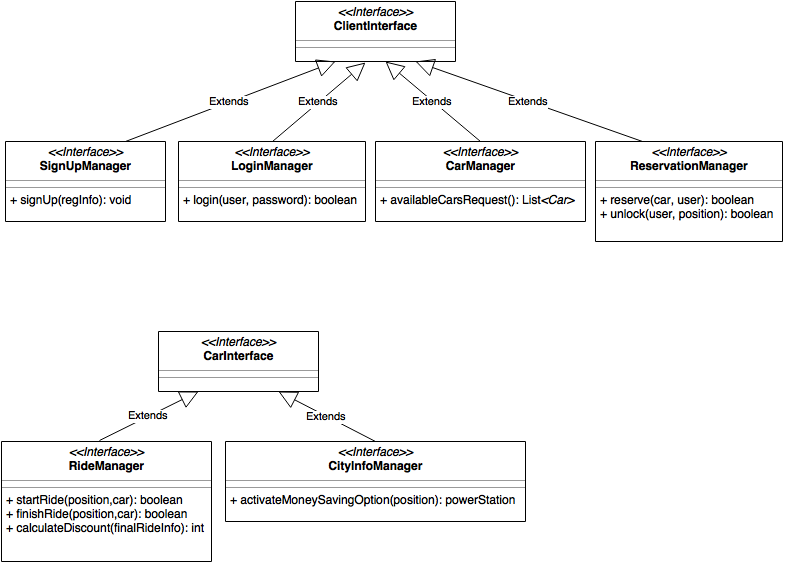


Once the client is at most five meters distant from the car he is reserved, he can send the request to unlock the car to the sever. This request is handled by the reservation manager who checks whether the client is actually close to the car. If this check goes well the reservation manager declares expired the reservation and unlocks the car. Then informs the ride manager that a reservation has just turned into a Ride. The ride manager is in charge of instantiating the new Ride and “waking-up” the CarApp on board the interested car. The new Ride will be instantiated only once the user will ignite the car. If the reservation manager detects that the user is more than five meters far from the car an error will be notified to the ClientApp.

### 2.6.4. Finish a ride

## 2.7. Component interfaces

In this section are explained the various relations among the interfaces and there is the list of methods provided by each of them. The various methods are actually implemented by the controllers described in section 2.3.



## 2.8. Selected architectural styles, patterns and design decisions

In this section there is a recap of our decisions in the choice of the architecture and the pattern we used.

* Client-server: The application is strongly based on a client-server communication model. This approach has been chosen for different reasons:
  + Data synchronization: there is only one application that manages the data.
  + Having one unique server application improves the maintainability of our system.
  + The application is independent from the number of clients connected (it can be scaled up).
  + Improves the security between clients
* Three tiers: our application will be divided into 3 tiers, each one referring to the corresponding layer:

1. Database (DAL: data access layer)
2. Application logic (BLL: business logic layer)
3. Thin client (a simple and easy interface to BLL)

* Thin client: we decided to spoil the client from as much logic as possible, in order to let the PowerEnJoy application to run efficiently also in low-resources devices. Reducing the logic in the clients also ensures a more security in the system, because hacker have more difficulty to attack the logic if it is on a server instead of on applications.
* MVC: as mentioned before, the server is structured using the MVC pattern as mentioned before. There are multiple controllers which implement the interfaces described in the previous section, three views, one for client interface, whose purpose is to dispatch every request to the corresponding controller. There is also a model which represents the structure of the data in the database

# 3. Algorithm design

# 4. User interface design

# 5. Requirements traceability

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

1. Allow guests to sign up.
   * The LoginController
   * The ViewForClient and the client app
2. Allow users to sign in.
   * The SignUpController
   * The ViewForClient and the client app
3. Allow users to see the available cars (and their battery level) near them or near to a given address.
   * The CarController
   * The ViewForClient and the client app
4. Allow users to reserve an available car for up to one hour and to know if their reservation went successfully and eventually fine them if the hour expires.
   * The CarController
   * The ViewForClient and the client app
5. Allow users to unlock and have access to a car if and only if they are close to that car and the car is reserved by them.
   * The CarController
   * The ReservationController
   * The ViewForClient and the client app
6. Allow users to end a ride if and only if the car is in a safe area or the car has run totally out of battery or an accident happens.
   * The RideController
   * The CarController
   * The ViewForCar and the car app
7. Allow users to receive a 10% discount from the total fee if they carry more than two people.
   * The RideController
   * The CarController
   * The CityInfoController
   * The ViewForCar and the car app
8. Apply a fine of 30% of the total cost to users if the car has been parked more than 3 km from the nearest power station or with less than 20% of battery.
   * The RideController
   * The CarController
   * The CityInfoController
   * The ViewForCar and the car app
9. Reward users with a 20% of discount if they leave the car with more than 50% of the battery.
   * The RideController
   * The CarController
   * The CityInfoController
   * The ViewForCar and the car app
10. Reward users with a 30% of discount if they leave the car charging into a power station.
    * The RideController
    * The CarController
    * The CityInfoController
    * The ViewForCar and the car app
11. Allow users to use the money saving option (see glossary)
    * The RideController
    * The CityInfoController
    * The ViewForCar and the car app
12. Allow users to know in real time all the information (cost, car’s battery level, safe areas’ location) about their ride.
13. Allow assistance coordinator to login.
    * The LoginAssistanceController
    * The ViewForAssistanceCoordinator
14. Allow assistance coordinator to see the GPS position of all the available cars and their battery level in order to identify the cars in need of battery replacement.
    * The LoginAssistanceController
    * The ViewForAssistanceCoordinator
15. Allow the assistance coordinator tag a car/untag as out of order following an accident or damage report by a user.
    * The LoginAssistanceController
    * The ViewForAssistanceCoordinator

# 6. Other info

## 6.1. Reference documents

## 6.2. Used tools

## 6.3. Hours of work

## 6.4. Changelog