Figure 1: Politecnico di Milano

Design Document
Version 1.0

Emanuele Ricciardelli (mat. 875221)
Giorgio Tavecchia (mat. 874716)
Francesco Vetró (mat. 877593)

December 9, 2016

Contents

1 Introduction

1.1 Purpose

This document will provide a set of architectural and design details in order to drive software developent of the PowerEnjoy system-to-be. Specifically, the following will be now identified and described:

- High level architecture;

- Main component view;

- The runtime behaviour of components;

- Design patterns.

1.2 Scope

The system should be able to register new users with their credentials, such as name, surname, address and e-mail, their valid driving license number and some payment information. Once the registration process is completed and all entered data are veried, the system will send a password to the user: this password will be used by the user to access the system and its functionalities. The system provides the following common features:

- find an available car;

- reserve an available car one hour prior the pick up;

- unlock the car when the user is detected nearby the car he/she reserved;

In addition there will be incentives for virtuous behaviors of the user that will aect their fares: if an user helps the community by being responsible with the usage of the car, his/her fare will be reduced by a specied percentage; otherwise, by not following the guidelines of the company, the user will be charged more than the standard fare

1.3 Definitions, acronysm, abbreviations

- **RASD**: Requirements Analysis and Specifications Document;

- **Gateway**: tool to set the communication and the exchange of data between two or more networks with, often, different protocols;

- **Push notification**: notification sent to a smartphone via application installed on it;

- **Email**: mail service carried over the web;

- **API**: Application Programming Interface. It is a set of subroutine definitions, protocols, and tools for building application software;

- **Google Maps API**: APIs provided by Google to compute functionalities over the Google Maps environment;

- **MVC**: Model View Controller pattern;

- **Tier**: architectural level in which the system is decomposed;

- **Client**: program or part of a program which allows to exchange data with a server to perform computations;

- **Thin Client**: client dedicated to only graphical representation, which displays the results of a computational logic contained exclusively in server;

- **Application Server**: it provides the infrastructure and support functions, development and execution of applications in a distributed environment;

- **Database Server**: software system able to store and manipulate data contained in a database;

- **Execution Environment**: it represents a particular execution platform, such as an operating system or a database management system. Execution environments are used to describe the context in which the execution of a model takes place;

- **Controller**: software system that applies changes to the model and database;

- **UX**: user experience diagram;

- **BCE**: Boundary Control Entity diagram;

- **Ticket**: the term refers to a fine that a driver has received during a ride;

- **Record Maintenance**: record saved in the Database, used to save information concerning the maintenance processes of a car.

For all terms not specified, please refer to the glossary contained in the previous document (RASD).

1.4 Reference documents

- Specication document: Assignments AA 2016-2017.pdf;

- RASD previously version 1.0;

1.5 Document structure

- Introduction: this section introduces the design document, providing a brief indication of its utility and about the covered sections;

- Architecture design:

    - Overview: this sections explains the high level components and their interaction;

    - Component view: more detailed view of the components of the application;

3

- Deployment view: this section shows all components that must be deployed to have the application running completely and correctly;

- Runtime view: it describes how components interact with each other to accomplish specific tasks;

- Component interfaces: it describes the interfaces between the components;

- Selected architectural styles and patterns: it provides details related to patterns and architectural styles selected for the development;

- Other design decisions;

- Algorithms design: this section provides some algorithms useful to describe some functionalities of the system;

- User interface design: this section presents a detailed description of user experience explained with diagrams like UX and BCE;

- Requirements traceability: this section aims to explain how the decision taken in the RASD are linked to design elements.
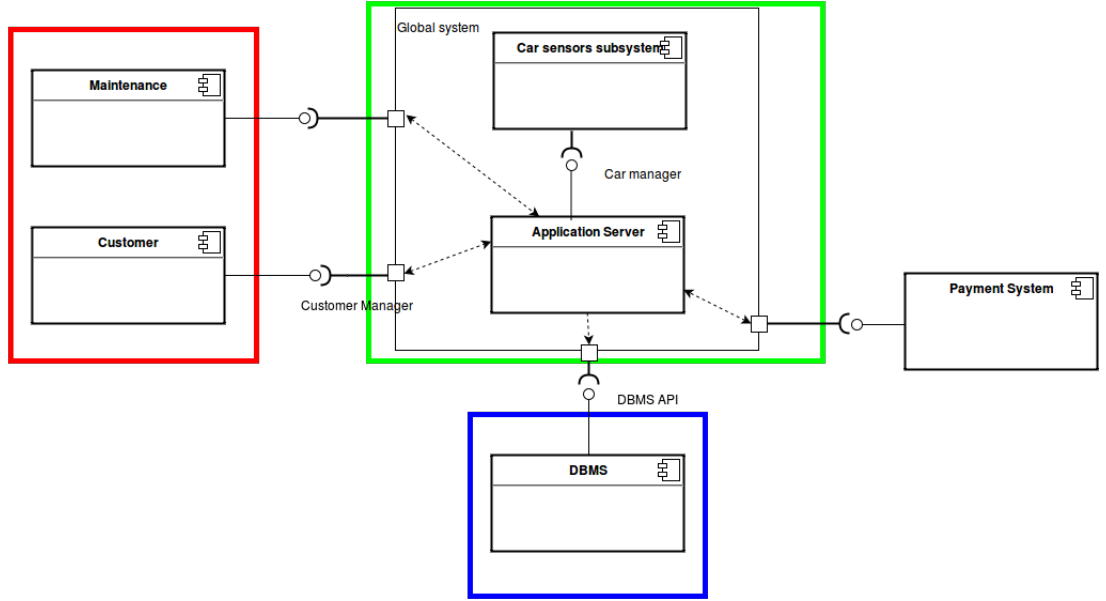
Figure 2: High level components

2 Architectural Design

2.1 Overview

The PowerEnJoy system will be shaped in a three tier architecture. This architectural style allows a further enhancement: the possibility to move the entire application in a dedicated cloud server with elastic infrastructure in order to get performances needed by the load on the application on demand and the consequent money saving. This choice will also face the coupling problem with respect either to the logic and data level and to the logic and presentation level. Nonetheless a performance evaluation might need to check whether the contract terms are satisfied. Important decision based on this architecture is the thin client characterization: the client can only send to the server a sequences of requests on what operation is intended to perform. This choice can possibly raise the expenses for the company. In the right part of Figure 2 is present the component named as Payment System: this component is intended to be a set of functionalities offered to the banks in order to ensure the correctness of all the payment operations. In Figure 3 we show the existence of firewalls in between the layers: this is due to the high security problems that this service arouse. Since our data level will store personal information such as identity, mail address, number of credit cards and even positions of a human being, a possible solution is to use different techniques to ensure that none will ever access private data.
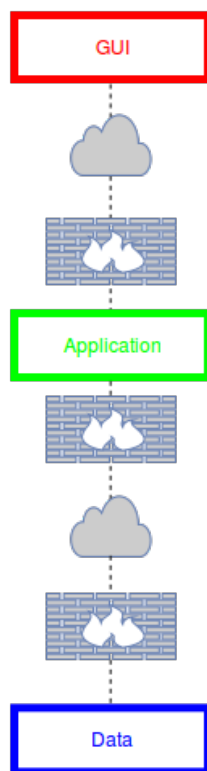
Figure 3: Tier levels

2.2 High level components and their interaction

The high level components architecture in Figure 2 is made of five parts. The main part is the *Global system*: it is a singleton that contains all the business logic. It is splitted into two sections:

- the *Car sensors subsystem*, as stated in the RASD, the cars are seen by the system as a set of sensors that only provide data. This part of the system-to-be is devoted to collect data from cars, convert them in a usable structure for the system and then send them to the business logic part named *Application server*;

- the *Application server* that will receive requests, will manage the information provided by the *Car sensors subsystem* and will be able to communicate with the *Payment system* in order to retrieve payments information.

The customer, via the mobile app or the services web page, can initiate a communication with the *Application server* that will complete the task requested and will answer back immediately with a message about the success or failure of the request and, if other information are needed, will send an email or a push notification providing the missing part (e.g. the client reserves a car, the request is successful and the system answers with a push notification/email containing the code to unlock the car). Then the maintenance service can communicate with the main application by visiting a dedicated web page. Here the maintenance personnel can look at all the stored maintenance requests and change the status of this requests. (e.g. a request of a car left with less than 5% of the battery might be changed to *performing*, and after the car is put in charge its status will be changed in *complete*). In the right part of Figure 2 there is the *Payment system*. This system is supposed to interact with banks and the Application server in order to perform checks and updates all the information regarding payments. The banks will use this interface to communicate with our system, and the *Payment system* will act as a controller and will request the *Application server* to perform the needed changes to the database in order to maintain the correct status of payments and users. This is due to a previous decision stated in the RASD *(Table 13: Provide a payment)*. The *Application server* needs to store and perform queries in order to perform all the tasks and so communicates with a database.
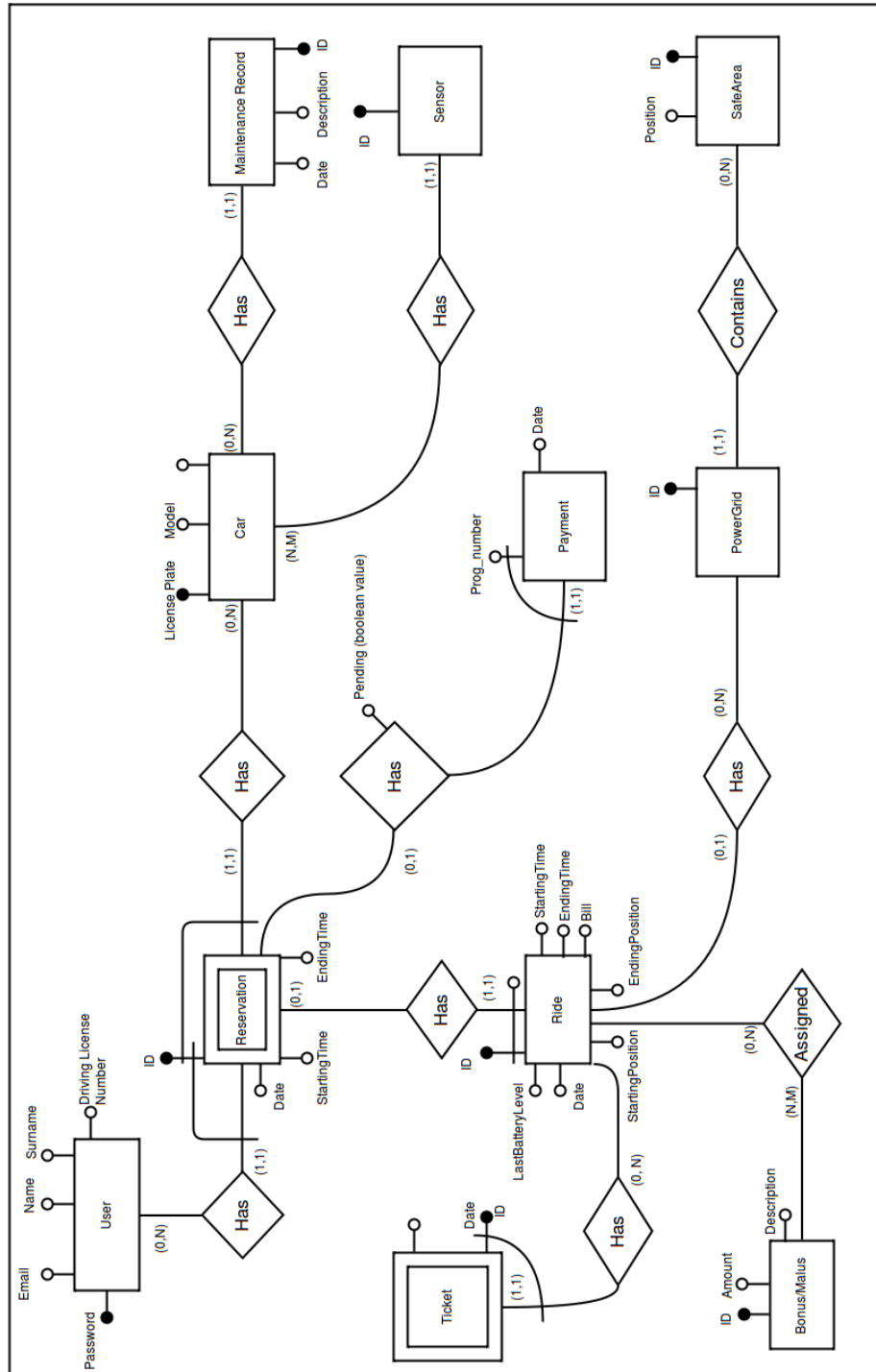
Figure 4: ER diagram

Observation: ID in *Reservation* entity will later be converted into QR Code to enable the user to access the car that he has previously reserved.

2.3 Component view

The component view consists of the following components:

- **Client**: clients device used to connect to the service

- **Maintenance**: maintenance device used to interact with the system in order to provide maintenance features.

- **Payment system**: third party device used to provide information about payments, their completion and other information relating.

- **Request manager**: manager used to receive request from the outside and to route them to the correct controller.

- **Login/Registration controller**: manage login and registration aspects

- **Reservation controller**: manage reservation aspects.

- **Ride controller**: manage ride aspects.

- **Cars controller**: manage all informations related to cars, sensors and their interaction with the users.

- **Collector data**: manage the data collection from different sensors

- **Payment controller**: manage payment aspects like register debts or mark their completion.

- **Communication manager**: component used to manage information exchange between different controllers and between them and the model.

- **Notification helper**: manage notification aspects, routing them to the correct gateway

- **Push gateway**: manage the sending of push notification to mobile app.

- **Email gateway**: manage the sending of Email messages.

- **Sensor gateway**: handle the sending and receiving of data collected by the various sensors present on cars.

- **Maintenance gateway**: manage the sending of notification to the maintenance company

- **Payment gateway**: manage the sending of payment information to the third party payment service

- **Model**: model of the world, used by the controllers.

- **Database**: database used to store persistent data.

- **Google Maps API**: API service used by the ride controller to provide navigation information.
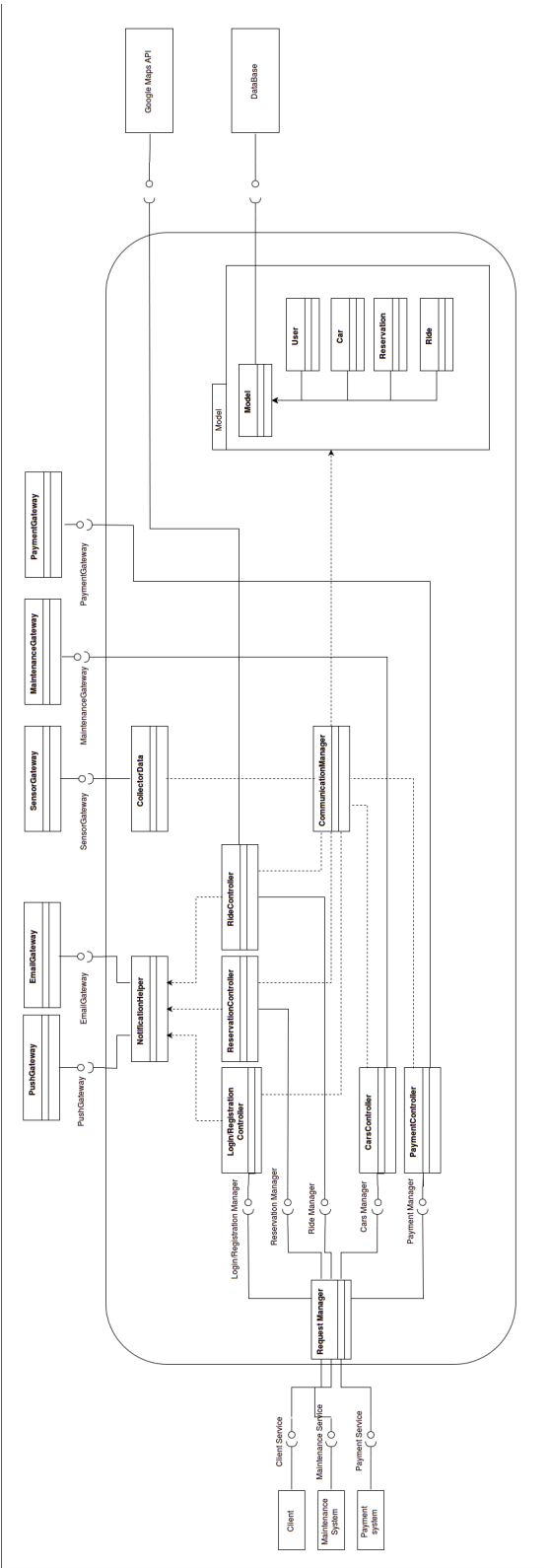
Figure 5: Component view

10

The interaction between the system and users is provided by a series of interfaces, one for each type of actor. All requests are handled by the single component *Request Manager* that will provide for their proper forwarding within the system. We have designed a system consisting of several controllers, each dedicated to a specific type of functionality. This allows us to maintain a high level of cohesion of the different components, allowing, in case of future expansions of the service, to update the individual components smoothly and with great flexibility. The division of functions in various components will also allow us to efficiently test the service provided. In order to maintain a high degree of flexibility, we decided to introduce a component *Communication manager* dedicated to the sorting of the information exchanged between the controllers themselves and the model, thus avoiding the definition of direct links between controllers that might obstruct and make more complex a future possible expansion of the service. In analogous way we introduced a component *Notification helper* dedicated to the proper addressing of the notifications to users, thus separating the functionality of sending of notifications from the logic of the underlying service. As shown in the diagram, the only access to the model and to the database it is via the *Communication Manager* that will carry out read and write functionality.
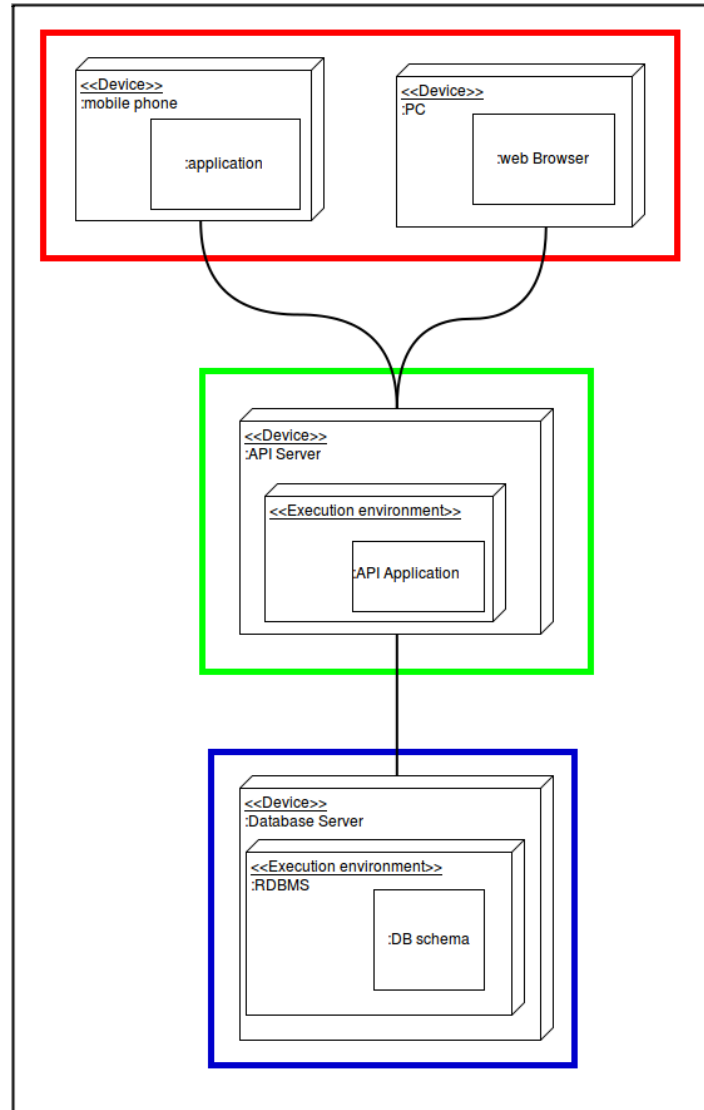
2.4 Deployment view

Figure 6: Deployment view

2.5   Runtime view