



PowerEnJoy  
Software Engineering II

## **Design Document**

*Giovanni Scotti, Marco Trabucchi*

Document version: 1  
November 25, 2016

# Contents

<b>Contents</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Purpose . . . . .	2
1.2 Scope . . . . .	2
1.3 Definitions, Acronyms, Abbreviations . . . . .	3
1.4 Reference Documents . . . . .	3
1.5 Document Structure . . . . .	3
<b>2 Architectural Design</b>	<b>5</b>
2.1 Overview . . . . .	5
2.2 High Level Components . . . . .	6
2.3 Component View . . . . .	8
2.3.1 Database . . . . .	8
2.3.2 Application Server . . . . .	10
2.3.3 Web Server . . . . .	11
2.3.4 Mobile Application Client . . . . .	11
2.3.5 On-Board Application Client . . . . .	12
2.3.6 Implementation Choices . . . . .	12
2.4 Deployment View . . . . .	14
2.5 Runtime View . . . . .	14
2.6 Component Interfaces . . . . .	15
2.6.1 Database - Application Server . . . . .	15
2.6.2 Application Server - Web Server . . . . .	15
2.6.3 Web Server - Web Browsers . . . . .	15
2.6.4 Application Clients - Application Server . . . . .	15
2.6.5 Application Server - External Systems . . . . .	15
2.6.6 Internal Application Server Components . . . . .	15

<b>A</b>	<b>Appendix</b>	<b>16</b>
A.1	Software and tools used . . . . .	16
A.2	Hours of work . . . . .	16
	<b>Bibliography</b>	<b>17</b>

# Section 1

## Introduction

### 1.1 Purpose

The Design Document is intended to provide a deeper functional description of the *PowerEnJoy* system-to-be by giving technical details and describing the main architectural components as well as their interfaces and their interactions. The relations among the different modules are pointed out using UML standards and other useful diagrams showing the structure of the system.

The document aims to guide the software development team to the architecture of the project providing a stable reference and a single vision of all parts of the software itself and clearly defining how they work.

### 1.2 Scope

The system aims to support a car-sharing service that exclusively employs electric cars.

The system is structured in a four-layered fashion, which will be thoroughly described in this document, that adapts to several forms of clients: various types of actors that interact with the system-to-be by generating a client-server dualism, hence a flow of requests-responses.

The architecture must be designed with the intent of being maintainable and extensible, also foreseeing future changes.

This document aims to drive the implementation phase so that cohesion and decoupling are increased as much as possible. In order to do so, individual components must not include too many unrelated functionalities and reduce interdependency between one another.

Specific architectural styles and design patterns will be followed in this document and used for future implementation, as well as common design paradigms that combine useful features of said concepts.

## 1.3 Definitions, Acronyms, Abbreviations

**ACID:** Atomicity, Consistency, Isolation and Durability. This is the set of properties of database transactions.

**DD:** Design Document.

**RASD:** Requirements Analysis and Specification Document.

## 1.4 Reference Documents

This document follows the guidelines provided by ISO/IEC/IEEE 1016:2009 [3] related to system desing and software design descriptions for complex software systems.

The indications provided in this document are also based on the ones stated in the previous deliverable for the project, the RASD document [1].

Moreover it is strictly based on the specifications concerning the RASD assignment [2] for the Software Engineering II project, part of the course held by professors Luca Mottola and Elisabetta Di Nitto at the Politecnico di Milano, A.Y. 2016/17.

## 1.5 Document Structure

This document consists of five sections:

**Section 1: Introduction.** This section provides a general introduction and overview of the Design Document and the covered topics not previously taken into account by the RASD [1].

**Section 2: Architectural Design.** It shows the main system components together with sub-components and their relationship. This section is divided into different parts whose focus is mainly on design choices, interactions, architectural styles and patterns.

**Section 3: Algorithm Design.** This section provides a high-level description and details about some of the most crucial and critical algorithms to be implemented by the system-to-be.

**Section 4: User Interface Design.** It provides an overview on how the user interface will look like and behave giving further information with respect to those contained in the RASD [1].

**Section 5: Requirements Traceability.** This section describes how the requirements defined in the RASD [1] are mapped to the design elements defined in this document.

At the end of the document are an **Appendix** and a **Bibliography**, providing additional information about the sections listed above.

## Section 2

# Architectural Design

### 2.1 Overview

In this section is a detailed view of the physical and logical infrastructure of the system-to-be, as well as the description of the main components and their interactions.

A top down approach will be adopted for the description of components:

**Section 2.2** A description of high level components and their interactions.

**Section 2.3** A detailed insight of the components described in the previous section.

**Section 2.4** A set of indications on how to deploy the illustrated components on physical tiers.

**Section 2.5** A thorough description of the dynamic behaviour of the software, complete with diagrams for the key functionalities.

**Section ??** A description of the different type of interfaces among the various described components.

**Section ??** A list of the architectural styles, design patterns and paradigms adopted in the design phase.

**Section ??** A list of all other relevant design decisions not mentioned before.

## 2.2 High Level Components

The main high level components of the system are:

**Database:** The system data layer; it includes all structures and entities responsible for data storage and management. No application logic is found at this level, apart from the DBMS one that must guarantee the correct functioning of the data structures while assuring the ACID properties of transactional databases.

**Application Server:** This layer encloses all the logic for the system applications, including the logic needed to interface with external systems and the key algorithms.

**Web Server:** This layer is in charge of providing web pages for the web-based application, and does not include any logic besides the basic request-response interaction one.

**Mobile Application:** The presentation layer dedicated to mobile devices; it communicates directly with the application server and only includes presentation logic.

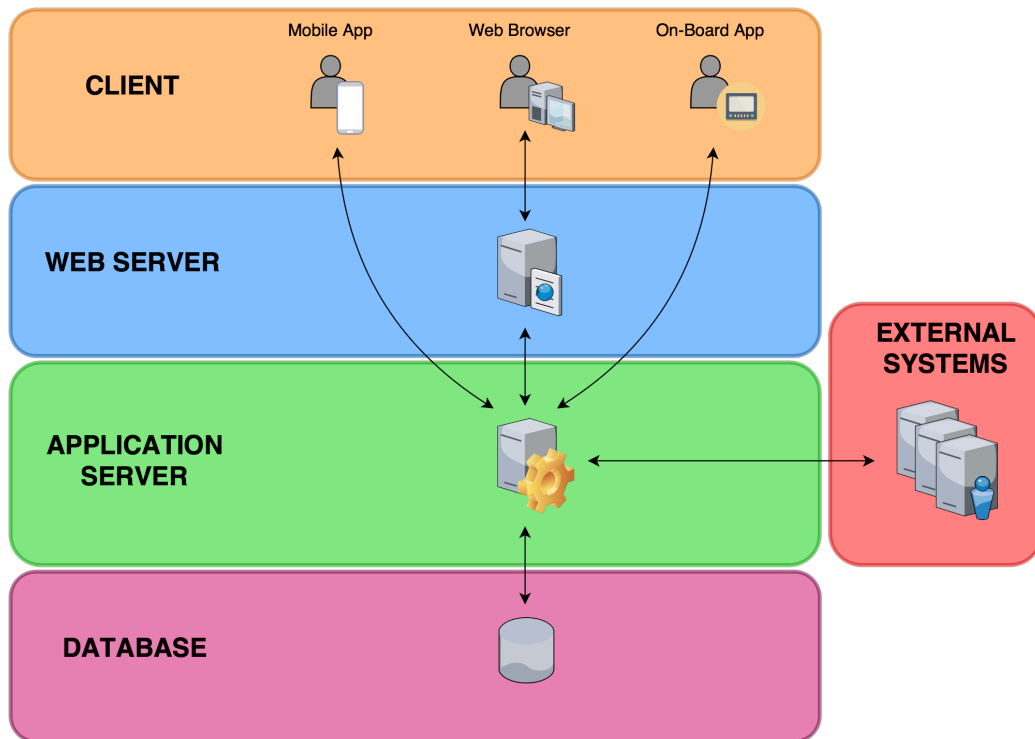
**Web Browser:** The presentation layer dedicated to web browsers; it relies on the connection with the Web Server to obtain the pages to be rendered on the client.

**On-Board Application:** The presentation layer dedicated to the on-board computers applications; it communicates with the Application Server for the most part of the logic; however it also includes the logic needed to interface with the physical car systems and to perform ride-related actions/computations.

The described components are structured in four layers, as shown in Figure 2.1. Said figure also includes the interaction with external systems, that is intended to happen at the level of the Application Server.

The choice of separating the Application and Web Server layers allows greater scalability, since it allows the deployment on distinct physical tiers that can individually be optimized to perform their respective task.





**Figure 2.1:** Layered structure of the system.

The interaction among the main system components is shown in Figure 2.2. The diagram noticeably points out the interaction with the payment handler and the maintenance systems, meant to support the *PowerEnJoy* service as stated in the RASD document [1]. Note that the web server and the application server are multi-threaded.



**Figure 2.2:** The high level components of the system.

## 2.3 Component View

In this section the individual components will be discussed in terms of the needed high level sub-parts and their functioning, as well as how those sub-elements interface with one another within the overlaying component and which of them is in charge of interfacing with other components.

### 2.3.1 Database

The database layer must include a DBMS component, in order to manage the insertion, modification, deletion and logging of transactions on data inside the storage memory.

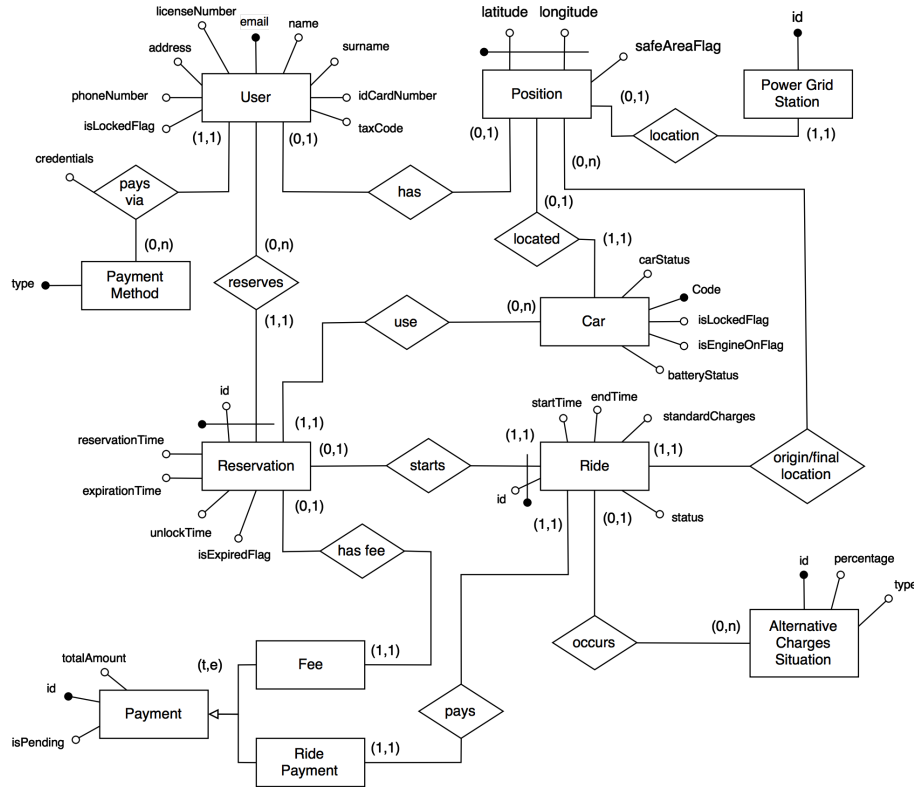
Regardless of the implementation, the DBMS must guarantee the correct functioning of concurrent transactions and the ACID properties; it also must be a relational DBMS, since the application needs in terms of data storage do not require a more complex structure than the simple one provided by

the relational data structure.

The data layer must only be accessible through the Application Server via a dedicated interface. With respect to this, the Application Server must provide a persistence unit to handle the dynamic behaviour of all of the persistent application data.

The Database must be tuned during the implementation phase in order to ensure security by granting data access according to the privilege level of the requester. Sensible data such as passwords and personal information must be encrypted properly before being stored. Users must be granted access only upon provision of correct and valid credentials.

The E-R diagram in Figure 2.3 illustrates a concept of the Database schema.



**Figure 2.3:** The E-R diagram of the database schema. Note that the relation named "origin/final location" must be considered as a short-hand notation to indicate two distinct relations: one for the starting location of the ride and one for the ending location.

### 2.3.2 Application Server

This layer must handle the business logic as a whole and the connections with the data layer and the multiple ways of accessing the application.

The main feature of the Application Server are the specific modules of business logic, which describe business rules and work-flows for each of the functionalities provided by the application itself.

The interface with the data layer must be handled, as stated in the previous section, by a dedicated persistence unit, that will be in charge of the object-relation mapping and dynamic data access and management; this ensures the fact that only the Application Server can access the Database.

The Application Server must provide a means to interface with the Web Server and the mobile and on-board clients via specific APIs in order to decouple the different layers with respect to their individual implementation. Moreover, it must provide a way to communicate with external systems by adapting the application to the existing external infrastructures.

The main business logic modules must include:

**UserManager:** This module will manage all the logic involved with user account management, login, registration, profile customization and management, as well as the generation and provision of user credentials.

**ReservationManager:** This module provides the logic behind the reservation management, with particular focus on the timing restrictions, car status updates (via the CarStatusManager) and reservation release conditions. It is also in charge of the controls to be performed in order to avoid multiple reservations by a user; lastly it must handle concurrent reservation issues such as pseudo-simultaneous requests for the same car.

**RideManager:** The logic included in this module is in charge of record all useful information about the rides as provided by the on-board application, including ride time, car battery levels and ride charges. It also manages the ride status updates.

**MapManager:** This module contains the logic used to locate cars and users, as well as defining the Safe Area boundaries and the power grids locations. It must provide useful data to the ReservationManager and the RideManager logic units, since both of them need localization information to perform their functionalities.

**CarStatusManager:** This module includes the logic needed by other components to set the car status of a car. It must also serve as an interface

with the external maintenance system by providing an automatic way to signal out-of-service cars.

**SecurityAuthenticator:** All the logic in charge of performing user-reservation-car matches is included in this module: this includes the control over cars unlocking and the logic behind the on-board authentication method.

**DiscountProvider:** This module is in charge of recording virtuous and bad situations as parameters of rides, so that the PaymentGateway can gather all the needed information to compute the corresponding net total charges.

**PaymentGateway:** The logic involved in the computation of final charges is included in this module; moreover, this unit must stand as an interface with the payment handlers upon the act of the automatic payments.

**NotificationManager:** Serves as a gateway from all the modules that need to notify the client towards the clients themselves by managing the logic behind the notifications services.

### 2.3.3 Web Server

The Web Server layer connects clients with the business logic layer in case the access to the application services is performed via web browser.

That being said, it is clear that the main functions to be implemented by this layer will essentially consist of interfaces, since - as stated in Section 2.2 - there will not be any logic implemented within the Web Server besides the presentation of pages.

The presentation must be structured in a clean and simple way, such that the components providing the client with web pages are decoupled as possible from the components that communicates with the business logic subsidizing the Application Server in order to fetch relevant data to be shown. The communication must be thought in a way that allows quick and efficient data transfer through textual data files over HTTPS, e.g. XML or JSON.

Adequate APIs must be designed in order to separate in the most efficient way the design of the Web and Application Servers.

### 2.3.4 Mobile Application Client

The Mobile Client must be designed in a way that makes communications with the Application Server easy and independent from the implementation

of both sides. In order to do so, adequate APIs must be defined and used similarly to what has been described for the interactions between the two server layers.

The mobile application UI must be designed following the guidelines provided by the Android and iOS producers.

The application must provide a software module that manages the GPS connection of the device and keeps track of locations data, providing it to the Application Server to be processed.

### **2.3.5 On-Board Application Client**

The On-Board Application consists of an application designed to run on pre-existing embedded devices on every car. The devices come together with the rest of the car equipment directly from the manufacturer. For this reason, the application must be designed following the guidelines provided by the manufacturer and based on given APIs.

The software module to be included in the application must manage the GPS locations of cars and the connection with the car sensors, as well as the transmission of all car-related data to the Application Server for the processing needed by the various business logic functions.

As far as the interface with the Application Server, the same considerations made with respect to the mobile application must be taken into account also for the on-board application.

### **2.3.6 Implementation Choices**

#### **Database Implementation**

The implementation choices for the Database layer are the following:

- MySQL 5.7 as the relational DBMS;
- InnoDB as the subsiding database engine;
- JDBC as the interface unit with the persistence unit of the Application Server.

#### **Application Server Implementation**

The main choice for this layer is the use of Java Enterprise Edition 7 (JEE). This was the most reasonable option for several reasons: the final product is a large-scope application, and thus needs distribution to great numbers of clients simultaneously; for the same reason, it needs to satisfy continuously

evolving functional requirements and customer demands; JEE also allows the developers to focus on the logic behind the main functionalities while being supported by a series of reliable APIs and tools that, among other features, can guarantee the main non-functional requirements of the case (e.g. security, reliability, availability...); lastly, it can reduce the complexity of the application by using mechanisms and models that easily adapt to a large-scale project.

The specific implementation choices are:

- GlassFish Server as the Application Server implementation;
- Enterprise JavaBeans (EJB) to implement the single business logic modules described in the sections above. These will be appropriately subdivided into EJB containers as specified by the JEE documentation;
- Java Persistence API (JPA) as the persistence unit to perform the object-relation mapping and the Database access. JPA Entities will be used to implement the object representation of the data entities;
- JAX-RS to implement proper RESTful APIs to interface with clients and the Web Server;
- To interface with external systems, existing RESTful APIs defined by the partner (payment handlers, maintenance system) will be used.

### **Web Server Implementation**

Based on the choices made for the Application Server implementation, an easy way to avoid interface issues between the two server layers is to use the GlassFish functionalities for the Web Server as well.

This layer will also heavily rely on JEE: the core components of the Web Server will be implemented with JEE web components. In particular, the application will use JavaServer Pages (JSP) as the technology for developing the web presentation logic based on the MVC pattern. The choice of JEE also allows the Web Server to make use of Servlets to manage specific interactions when necessary.

The interface with the Application Server will be provided by the same RESTful API specified in the previous subsection.

### **Mobile Application Client Implementation**

The mobile application UI must be designed following the design guidelines provided by the devices' manufacturers. Two architectures must be sup-

ported: iOS and Android. The iOS application must be written in Swift, while the Android one must be implemented in Java.

The core of both application must be a Controller that communicates user inputs (after translating them from the UI input) to the Application Server via RESTful APIs. The access to the GPS of the devices must be performed through the default frameworks of the respective systems.

### **On-Board Application Client Implementation**

The on-board application must strictly follow the car manufacturer guidelines for the development. The given APIs must be used to properly connect with the car equipment and sensors.

Since the car embedded devices have limited hardware specifications, the software must be written in C/C++. The user interface must be managed by a dedicated unit. The core of the application must include a GPS manager and a mobile connectivity unit. The communication with the Application Server must be performed by the connectivity unit via an adequate RESTful API. Data-flows from the car sensors must be handled by a separated data collection unit. The single elements will be coordinated by a Controller.

## **2.4 Deployment View**

The deployment diagram for the system is shown in the following figure:

## **2.5 Runtime View**

This section describes the dynamic behaviour of the system in the most relevant cases.

The following sequence diagrams highlight the runtime interactions between clients, servers and the database; in the case of internal interaction between sub-components of the Application Server, these interactions are expanded and detailed. This is not the case with the client applications, since the internal structure is simpler and there is no need to further detail said relations.

Direct interactions between the Application Server and the Database are not explicitly represented, since such interactions are abstracted by the persistence unit of the Application Server.



## **2.6 Component Interfaces**

This sections includes further details on the interfaces between different components of the system. Also, Subsection 2.6.6 is devoted to illustrate some relevant details about the interface among internal elements of the Application Server.

### **2.6.1 Database - Application Server**

### **2.6.2 Application Server - Web Server**

### **2.6.3 Web Server - Web Browsers**

### **2.6.4 Application Clients - Application Server**

### **2.6.5 Application Server - External Systems**

### **2.6.6 Internal Application Server Components**

# Appendix A

## Appendix

### A.1 Software and tools used

- L<sup>A</sup>T<sub>E</sub>X, used as typesetting system to build this document.
- draw.io - <https://www.draw.io> - used to draw diagrams and mockups.
- GitHub - <https://github.com> - used to manage the different versions of the document and to make the distributed work much easier.
- GitHub Desktop, the GitHub official application that offers a seamless way to contribute to projects.

### A.2 Hours of work

The absolute major part of the document was produced in group work. The approximate number of hours of work for each member of the group is the following:

- Giovanni Scotti:
- Marco Trabucchi:

NOTE: indicated hours include the time spent in group work.

# Bibliography

- [1] AA 2016/2017 Software Engineering 2 - *Requirements Analysis and Specification Document* - Giovanni Scotti, Marco Trabucchi
- [2] AA 2016/2017 Software Engineering 2 - *Project goal, schedule and rules*
- [3] IEEE Standard 1016:2009 *System design - Software design descriptions*