Figure 1: Politecnico di Milano

Integration Testing Document
Version 1.0

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

January 14, 2017

Contents

1 Introduction

1.1 Purpose

This document represents the Integration Testing Plan Document for PowerEnJoy service. The purpose of the testing is to verify and ensure that the system operation is correct and complies with all the requirements expressed in the previous documents (RASD and DD). All components of the system will have to comply with the constraints and functionalities for which they were designed, without showing unexpected behavior. The purpose of this document resides therefore in specifying the methods and the time needed to complete a full testing, both from the point of view of the individual components, both in their interactions. In order to provide a detailed description of the future testing, in the following paragraphs it will be defined:

- A list of the subsystems and their subcomponents of the PowerEnJoy system-to-be that will be involved in the testing activity;
- The criteria that must be met by the project status before integration testing of the outlined elements may begin;
- A description of the integration testing approach and the rationale behind the specific choose;
- The sequence of component and function integration;
- A description of individual steps and test description: for each step of integration, we will define a test design with related expected results for each test;
- A list of tools, test equipment and test data required.

1.2 Definitions and Abbreviations

- RASD: Requirement Analysis and Specification Document;
- DD: Design Document;

For all other definitions, please refer to previous documents listed in the next paragraph

1.3 Reference Documents

- Specification document: Assignments AA 2016-2017.pdf
- RASD Version 1.1
- DD Version 1.1
- Integration testing example document.pdf

2 Integration strategy

2.1 Entry criteria

With reference to the past documents and in particular to the design document, it is necessary to specify how the individual components that make up the system require to achieve a given percentage of completion with respect to the features for which they were designed. Especially, with respect to the High Level Component view of the Design Document:

- 100% for the DBMS component: in order to allow a correct testing not only of the database structure itself, but also of the various components of the system that base their operation on access to the database;
- 60% of the Client components: many of the functions covered by the client involve the GUI and have no priority over interaction with the system functionality;
- 90% of the Application Server;
- 90% of the Car Sensor Subsystem: in order to ensure a correct testing of the application server, given the close interaction.

These criteria must be met before the start of the Integration testing. Different percentages want to reflect the order of integration, placed its focus on certain components, and otherwise in the background some secondary aspects such as the graphical user interface on the client.

2.2 Elements to be integrated

In order to specify the elements to be integrated, we refer not only to High Level Component view contained in the DD, but also to the Component view that allows us to identify components at a lower level and the relationships between them . As specified in the Design Document, the components of our system have been designed in such a way as to ensure a high level of modularity, by separating the different functionality into distinct components, thus also facilitating testing. For this reason, right from the Component view it is possible to note how different components are strongly dependent on one another in order to achieve more complex functionality of the system-to-be. The recognition of these complex functionality allows us to split our system into a number of subsystems, addressed respectively to the access management and registration system (*Account Subsystem*), management of payments (*Payment subsystem*), reservation management and everything that concerns the sphere of rental (*Reservation subsystem*) and finally management of all the sensors and the exchange of information between cars and system and everything related to the maintenance (*Car / Sensors subsystem*). In particular, the functions relating to the *Reservation subsystem* will involve the integration of:

- Reservation Controller;
- Ride Controller (integrated with Google Maps API);
- Notification Helper (with related gateway);

- Communication Manager;
- Request Manager;
- Data Access Manager.

With respect to the *Account subsystem*, it is concentrated only in the *Login / Registration Controller* that later, through the *Communication Manager*, will interface with the other components of the PowerEnJoy system. In detail:

- Login/Registration Controller;
- Communication Manager;
- Request Manager;
- Notification Helper (with related gateway);
- Data Access Manager.

The same reasoning can be made for the *Payment subsystem* and the respective *Payment Controller*. In detail:

- Request Manager;
- Payment Controller;
- Communication Manager;
- Data Access Manager;
- Payment Gateway;

Finally, the *Car / Sensors subsystem* in order to perform all its functionalities will need to integrate:

- Cars Controller;
- Communication Manager;
- Collector Data (with related gateway);
- Maintenance Gateway;
- Request Manager;
- Data Access Manager;

It is important to underline that the components *Notification Helper Request Manager* and *Communication Manager* are significant for all subsystems and therefore their integration. This means that if on the one hand their presence ensures greater flexibility in the management of functions, on the other hand emphasizes the importance of their proper functioning in order to ensure the fulfillment of all the functionality of the overall system. Given the above definitions of subsystems, our integration testing will proceed initially considering a single subsystem at a time, following finally with the integration of the same until the establishment of the system in its entirety.

2.3 Integration testing strategy

The approach suggested in the process of integration testing is bottom-up. This choice is based on considerations relating to the features covered by our system: in particular, as has been documented in the DD, it has been chosen to decompose the system into a series of components, each with features related to a particular field. The functionalities that the system will perform are derived from joint work of these components. From this observation it can be seen that the choice of a bottom-up reflects the needs of the testing related to our system. In this way we can favour the ability to identify problems at an early stage of development of the components and, in case, to react quickly. Later, after having tested the elementary interactions between components and subsystems, we can proceed with the integration to a higher level. It will therefore be necessary to develop a set of drivers to drive the testing, but given the simultaneous development of the overall system, this will prove to be quite natural. Once the system will be completely formed, we shall provide the implementation of a series of System Testing in order to verify not only the achievement of the functional requirements, as a whole, but also of the non-functional aspects such as the loading capacity as a function of the expected values.

2.4 Sequence of component / function integration

In this section we will provide a description of the order of integration testing of components and subsystems of PowerEnJoy system-to-be.

2.4.1 Software integration sequence

Given the previous subdivision into subsystems and starting from the *Reservation subsystem*, we now define the sequence in which the components will be integrated for each subsystem and then the sequence related to the integration of the subsystems themselves.

Reservation subsystem
Since many features covered by this system need an interaction with the database (performing queries or updating tables), the very first components to be integrated are the **Data Access Manager** and the **DBMS**. As will later be made

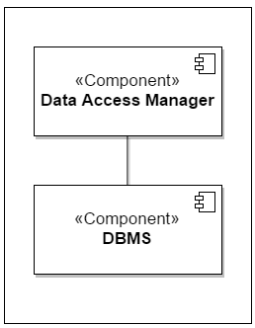

Figure 2: Integration sequence - 1 - Data Access Manager, DBMS

clear, this first integration will also be required for other subsystems, but will not be repeated if it is unnecessary; otherwise it will be repeated the possible variants of integration between these first components and the surrounding ones. With respect to the bottom-up approach, the next integration step will cover the correct communication between the database and the **Communication Manager**. We can now proceed to a higher level of integration considering the

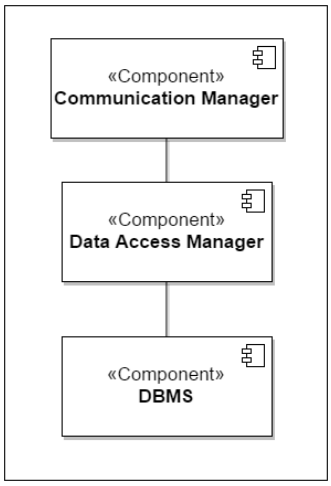

Figure 3: Integration sequence - 2 - Communication Manager & above

Ride Controller: this component will be integrated with the API provided by the services of google in order to verify the correct use within their own methods, with the "Notification Helper" in order to verify the correct sending of notifications generated to users, and finally with **Communication Manager** which, as described in the DD, is designed to allow (in a flexible and transparent way) the exchange of information between internal components of the system, to verify proper access to the database. At the same level of integration we will introduce the **Reservation Controller** integrated with **Communication Manager** and **Notification Helper** for the same reasons mentioned above.

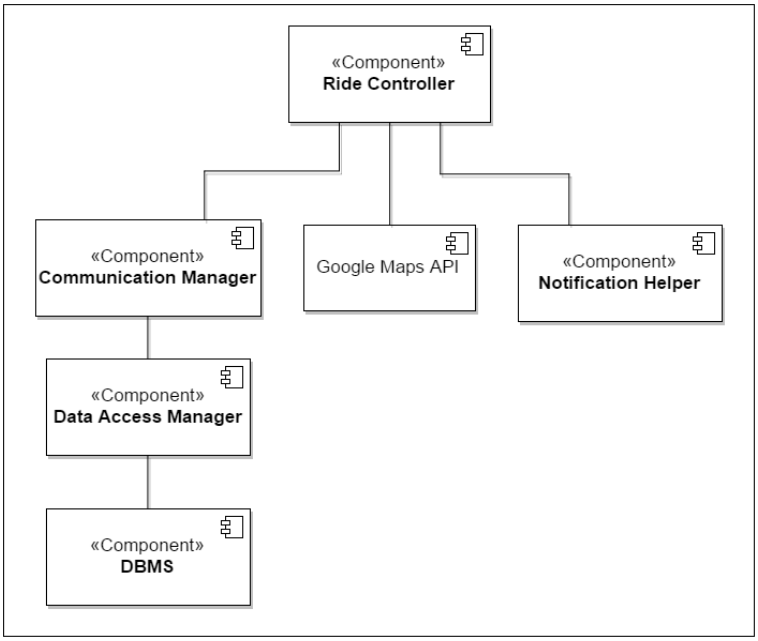

Figure 4: Integration sequence - 3 - Ride Controller, Notification Helper, Google APIs & above

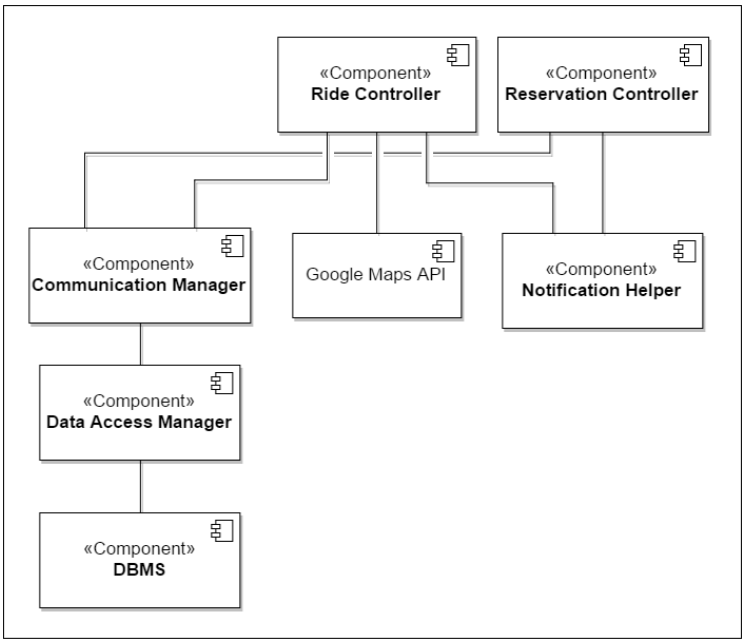

Figure 5: Integration sequence - 4 - Reservation Controller & above

Since different functions performed by the controller such as **Reservation Controller** and **Ride Controller** are performed by request from external clients, and given that the same demands are made through the component **Request Manager**, the next components to be integrated will be the following.

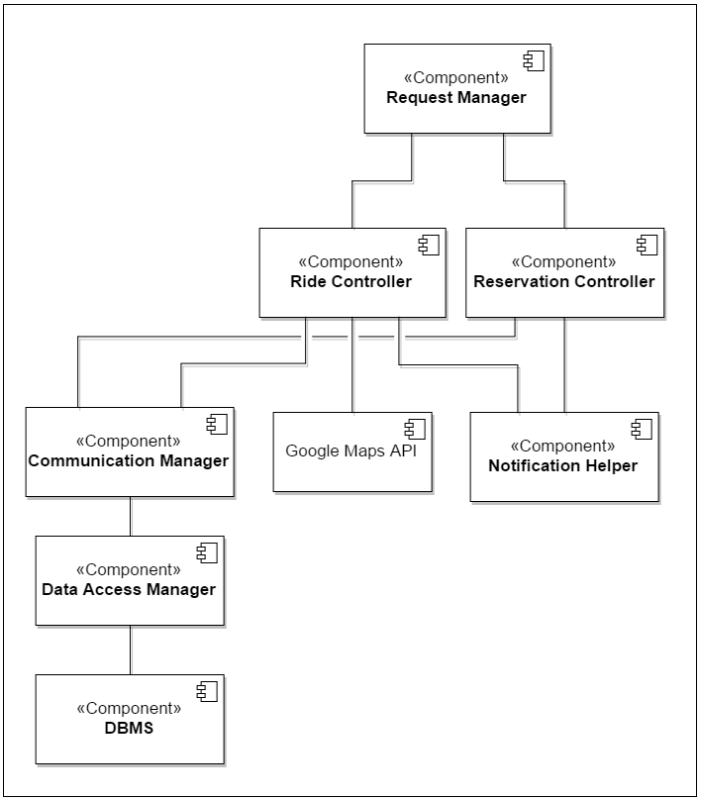

Figure 6: Integration sequence - 5 - Request Manager & above

This will make it possible to test the correct routing of requests towards the selected components and the exchange of information between them. Finally we integrate the Client, to test that all of the functionalities of this subsystem work properly.

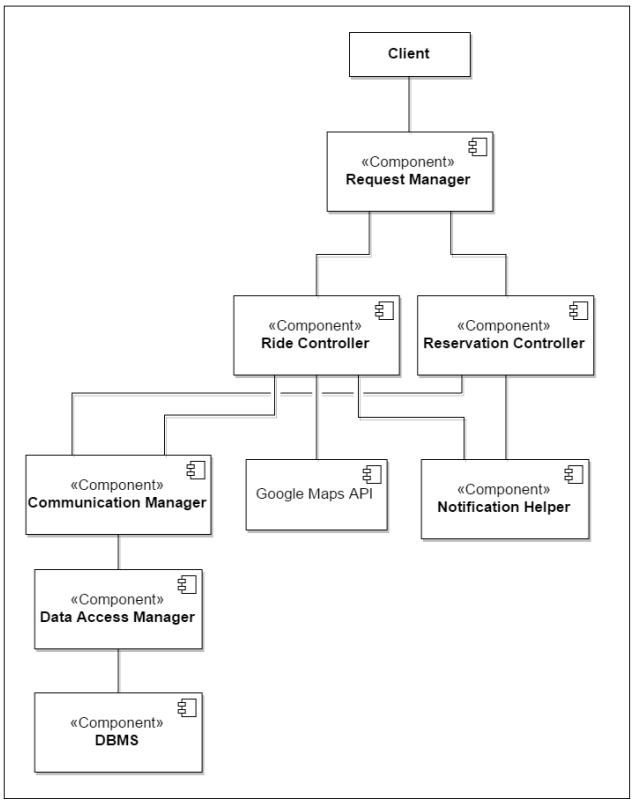

Figure 7: Integration sequence - 6 - Client & above

Account subsystem
The starting point relative to the sequence of integration of this subsystem should be regarding the access to the database, but, as previously expressed, since it was done for the previous subsystem, it is not required to do it again. It is now shown for the sake of completeness to the reader. We follow now

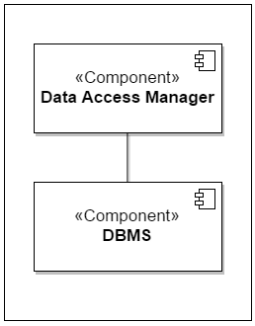

Figure 8: Integration sequence - 1 - Data Access Manager, DBMS

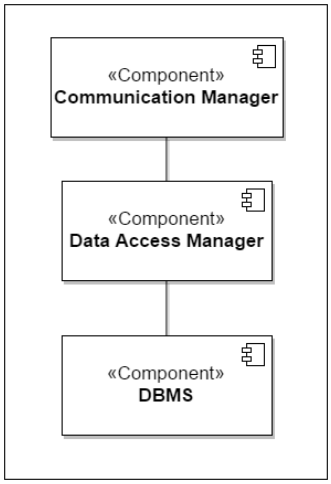

Figure 9: Integration sequence - 2 - Communication Manager & above

with the integration between the **Login / Registration Controller** and **Notification Helper** to verify, as done in the previous subsystem, the correct forwarding of notifications generated and, at the same time, integrating the **Login / Registration Controller** with **Communication Manager** and **Data Access Manager** in order to test the proper functioning of methods that require database access. Going up to a higher level of integration, we introduce

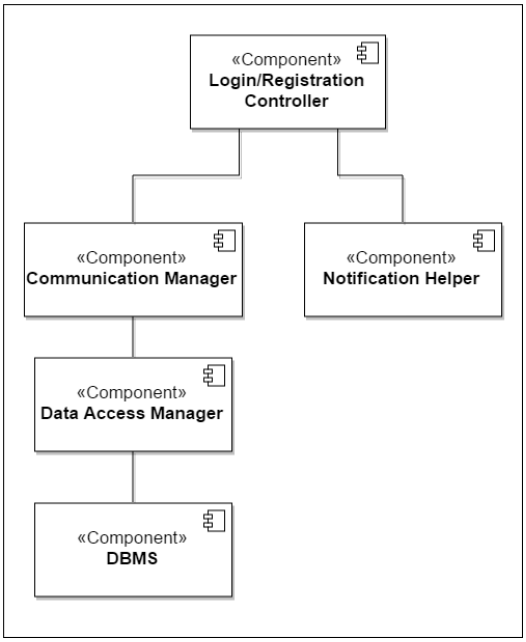

Figure 10: Integration sequence - 7 - Login/Registration Controller, Notification Helper & above

the **Request Manager** in order to test the correct routing of requests coming from the outside to the component suited to the management of the system accounts. And again we terminate this sequence of integration introducing the Clients to test that all of the functionalities of this subsystem work properly. Thus completing the integration of this subsystem.

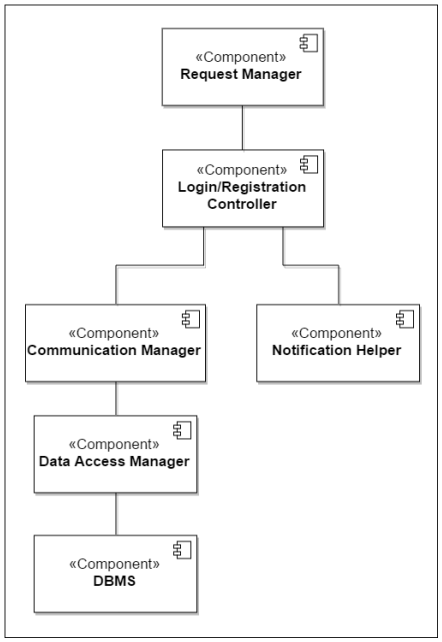

Figure 11: Integration sequence - 8 - Request Manager & above

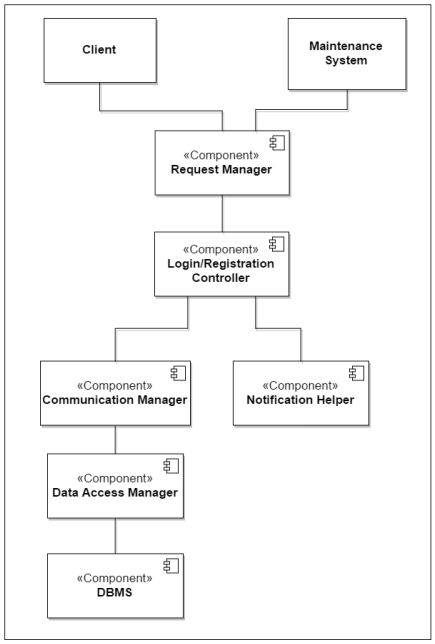

Figure 12: Integration sequence - 9 - Client & above

Payment subsystem
We now proceed in the integration process Considering the *Payment Subsystem*. As before, we start integrating the components related to the access to the database. The first components that need to be integrated are the **Payment**

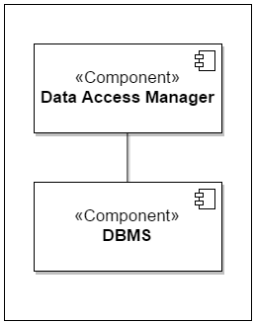

Figure 13: Integration sequence - 1 - Data Access Manager, DBMS

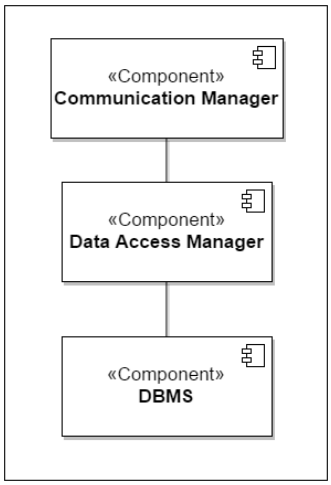

Figure 14: Integration sequence - 2 - Communication Manager & above

Controller and the **Payment Gateway**, this to verify that the results relating to payment functions are correctly sent to the external payment companies, and the integration between **Payment Controller** and **Communication Manager** to test the communication with the database. Followed by the integration

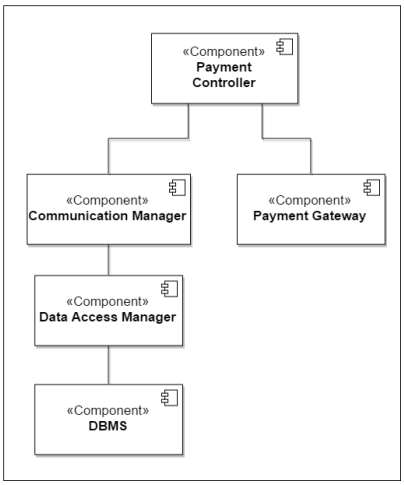

Figure 15: Integration sequence - 10 - Payment Controller, Payment Gateway & above

with the **Request Manager** to verify the correct forwarding of information requests to the subsystem. As before we terminate this sequence of integration testing the connection with the Client.

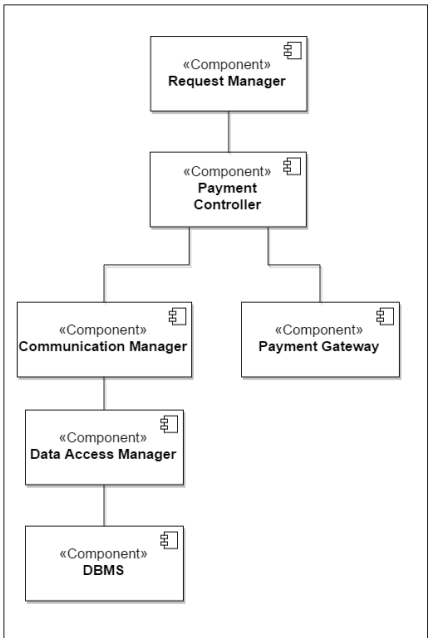

Figure 16: Integration sequence - 11 - Request Manager & above

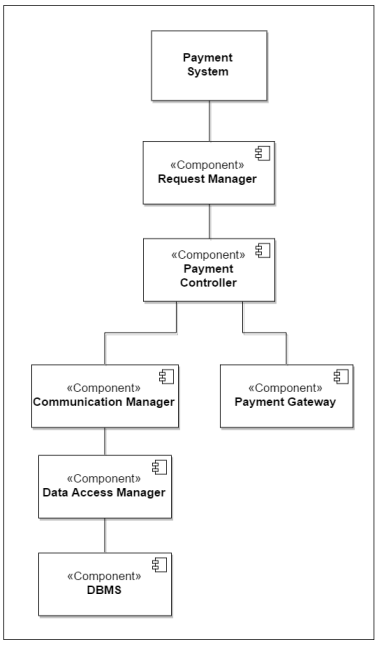

Figure 17: Integration sequence - 12 - Client Payment System & above

Car / Sensors subsystem
The last subsystem to be included in the sequence of integration is the *Car / Sensors subsystem* that will handle all exchanges of information and operations to be performed on vehicles and everything related to the maintenance. The starting point is still represented by the access to database. As first components

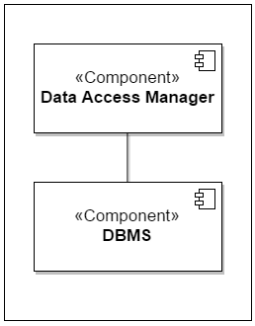

Figure 18: Integration sequence - 1 - Data Access Manager, DBMS

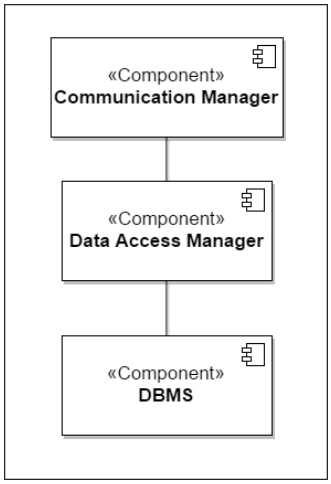

Figure 19: Integration sequence - 2 - Communication Manager & above

to be integrated, we choose **Cars Controller** and **Data Collector**: this choice derives from the importance that covers the proper exchange of information between system and machine, upon which are based a considerable number of other functions related to the **Cars Controller** itself. In addition we integrate our principal component with the **Maintenance Gateway** to test the proper sending of information from the system to the maintenance company and with the **Communication Manager** to verify the correct access with the database. Finally the sequence of integration will end integrating the **Request Manager** and so to verify the correct implementation of functionalities such as reporting

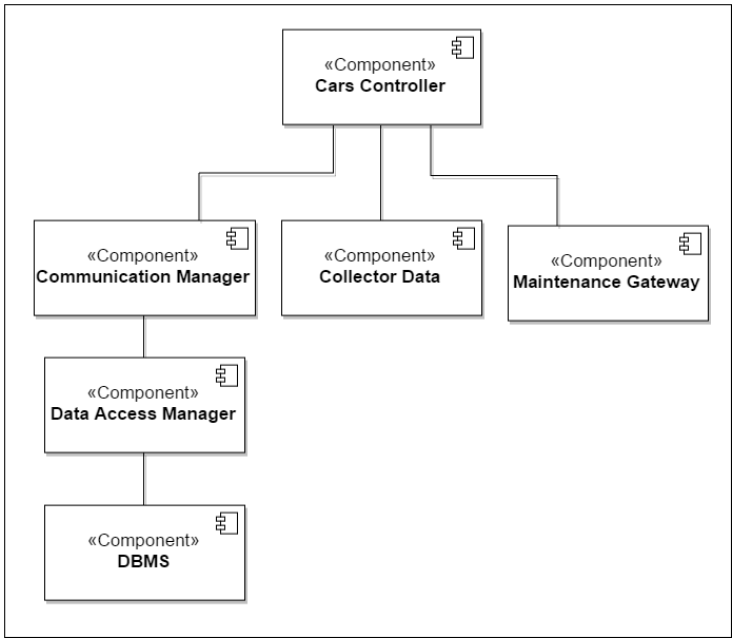

Figure 20: Integration sequence - 13 - Cars Controller, Collector Data, Maintenance Gateway & above

a fault to a machine or access by the maintenance to the list of reported cars. As last integration, like in other subsystems, we verify and test the integration with the clients.

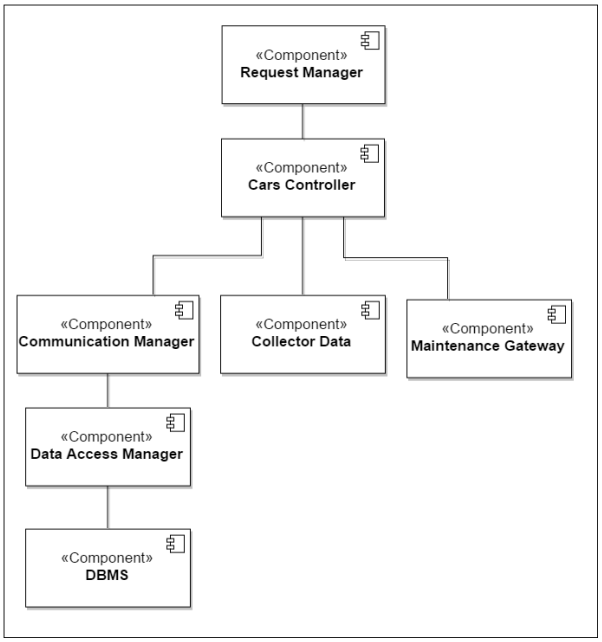

Figure 21: Integration sequence - 14 - Request Manager & above

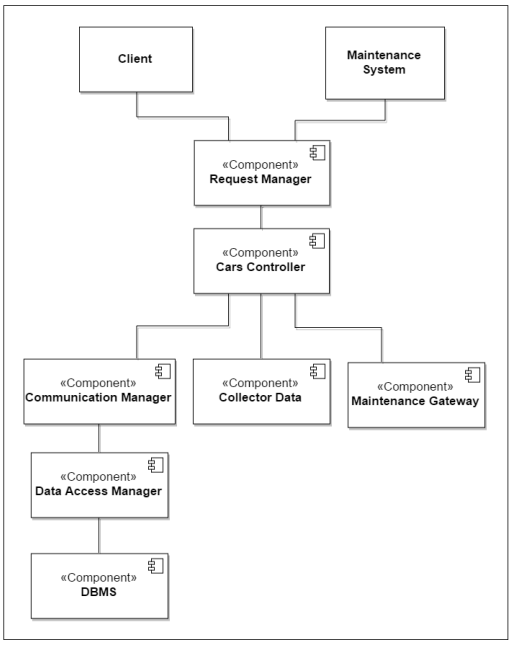

Figure 22: Integration sequence - 15 - Clients & above

Note to the reader: In previous integration sequences it was performed a choice concerning the gateway present in the component view of the DD: we decide to show explicitly only the payment and maintenance gateway. This is because the integration of the omitted gateway is to consider joint with the components to which they are directly linked: For example, the **Sensor Gateway** is considered as a joint to the component **Data Collector** and the gateway relative to the notifications joint with the **Notification Helper**.

2.4.2 Subsystem integration sequence

In order to recreate from the subsystems, considered in the previous section, the overall system, the process of integration will initially proceed by integrating the **Reservation Subsystem** with **Car / Sensor Subsystem** in order to test all of those functions of the first subsystem that require access to the functionality provided by the second. It is now integrated the **Account Subsystem***in order

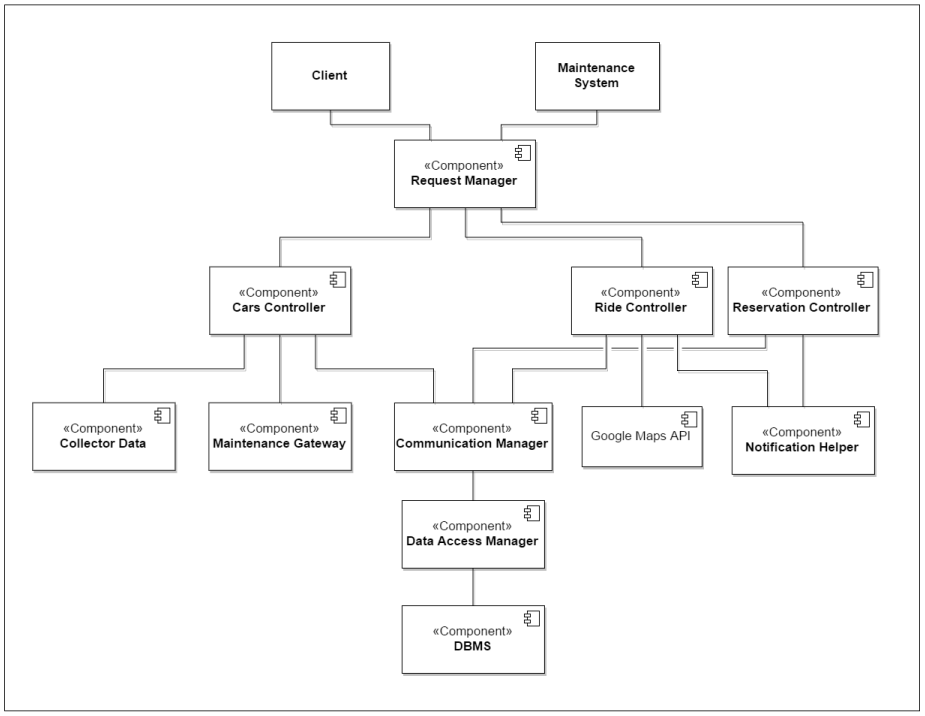

Figure 23: Integration sequence - 16 - Clients & above

to test the integration of external user and system, through all the features that will provide PowerEnjoy to it, excluding those relating to payments that are later tested with the integration of the related subsystem. ;

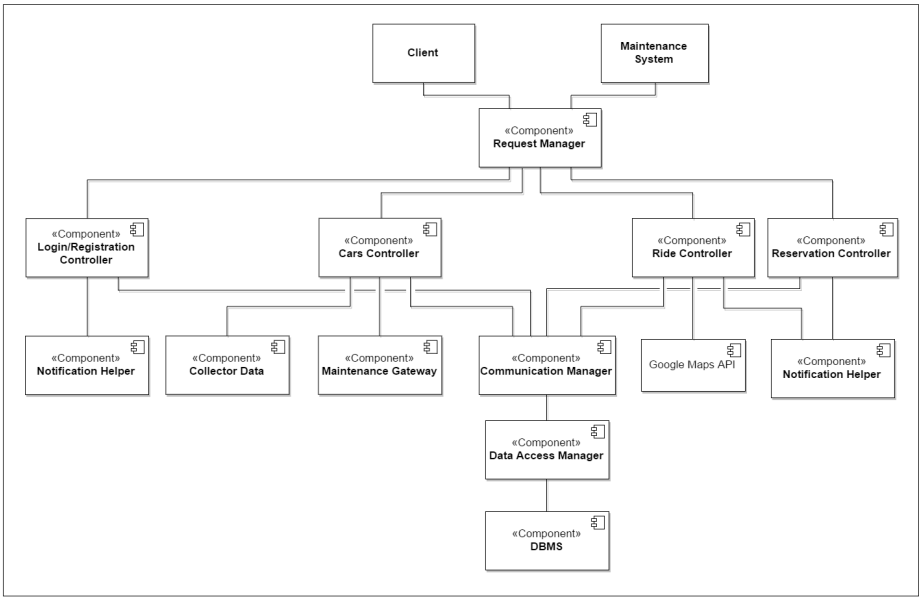

Figure 24: Integration sequence - 17 - Clients & above

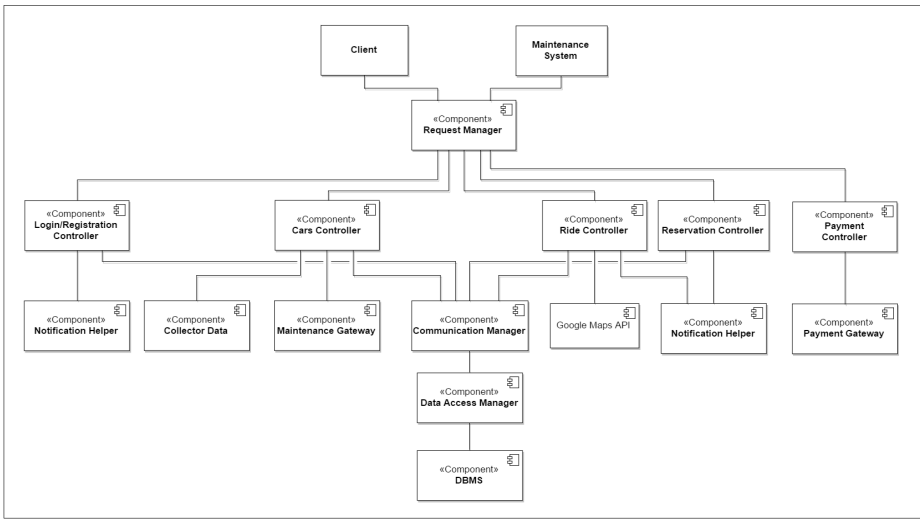

Figure 25: Integration sequence - 18 - Clients & above

3 Individual steps and test description

4 Performance analysis

The PowerEnJoy service is characterized by a complex structure that contains heterogenous operations: a simple example is the monitoring of all running cars while processing a payment and, meanwhile, evading multiple clients requests. This leads to a simulations based study, using specific stress and load tests, regarding the application server, focusing on the response latency, usage of internal resources and effective parallelism of processes. Next step, but not by importance, there should be made some investigations about the latency of DB-related operations focusing over the time that a request spends in I/O operations rather than being processed and completed. These tests are strongly advised even after the effective release of the system because their power is to highlight potential errors and generate a starting point for future improvements. Talking about time latency with respect to the client is not of interest: since the service is accessed via the Internet, the connection speed that the client is using determines the sensation of a quick and responsive service rather than a slow and unsatisfying one. However, our aim is to ensure the maximum usability of our product. Focusing on the mobile platforms, further problems arise: the CPU usage, free memory available; battery usage can be seen as a consequence of the preceding couple. Those aspects are connected together and we can look for a potentially good trade-off:

- CPU usage must be limited between bounds: an abuse of the CPU will make the usage of a mobile device quicker but at the cost of increasing the usage of the other two parameters taken into account;
- usage of available memory should be minimized by project specification (both RASD and DD): since our service is based on the interaction with a service provider, the effective data stored in the device is minimal and limited to unlock codes and few more generic ones.

5 Tools and test equipment required

5.1 Test tools

Testing for the PowerEnJoy system-to-be will be splitted in three main parts:

- Business logic components: in order to tackle testing of this section of the system, the two main tools that will be used are Arquillian integration testing framework and the JUnit framework. The first one is specifically designed for checking the execution of a component with respect to the surrounding environment. In this specific system, its functionalities will be used to verify that communication with database and how the communication flow is performed by the system with respect to what was expected. JUnit framework will be used for both the purpose of unit testing in combination with test coverage assist like Sonarqube, and its

potentiality will be adapted to integration testing with respect to the correctness of responses that the various component give back when an error and exceptions occur;

- Mobile applications: the mobile application must be tested with respect of CPU and memory usage and energy consumption. This could be done by specific tools within the SDK of the platform target of testing;
- Validation, certification and standardisation for web pages: W3C is the specific tool for validating the system-to-be web code related to the user experience. Our aim with this kind of testing is to generate a product that will target the most wide pool of potential users and therefore clients.

5.2 Test equipment

As far as equipment is concerned, the variety of products used as benchmark will be simulated and will represent a wide selection of:

- Mobile devices: their emulation is available within the SDK used to develop the mobile application;
- Personal computers: resource usage is not the main target for this specific pool of products since the effectiveness and responsiveness is mainly due to the effective Internet connection. The equipment is identified in a basic/average resources availability.

Focusing on the backend of the service, it is recommended to stipulate a contract with a resource provider: the computational power required by the entirety of the service offered to the client can setted by a pay-for-use contract. In this way the service will be improved incrementally and performances tests can be completed with the possibility of saving a certain amount of money. For the first official release will be used a completely proprietary infrastructure that, in the future can be moved into the cloud. This steps can improve the potential saving of the company because costs are setted as the computational force required is verified.

6 Required program driver and test data

6.1 Program driver and stubs

As stated before in this document, we are going to follow a bottom-up approach for integration testing. This decision leads to the necessity of creating the needed set of drivers in order to perform the identified tests pools, in particular we will construct drivers for each module, excluding those coming from third parties, as the DBMS and the API provided by the Google services. Hereafter is a list of drivers necessary to test the integration of the system-to-be:

- **Data Access Driver**: the development of this test module will be focused on invoking methods exposed by the *Data Access Manager* in order to test its interaction with the DBMS. We emphasize that the DBMS does not

need a direct driver to be tested with, being a product supplied by third parties.

- **Communication Manager Driver**: the development of this test module will be focused on invoking methods exposed by the *Communication manager* in order to test the internal communication capabilities. Especially by testing the ability to invoke methods on the *Data Access Manager* and then allow access to the database to system components.

- **Ride Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Ride Controller* and therefore the invocation of all the methods related to the ride operations, and will focus on verifying the interaction with components such as *Communication Manager*, *Notification Helper* and *Google Maps API*.

- **Reservation Controller Driver**: as for the *Ride Controller Driver*, this module will be focused on invoking methods exposed by the *Reservation Controller* and therefore the invocation of all the methods related to the reservation operations. In addition, it will be used to test the correct interaction with components such as *Notification Helper* and *Communication Manager*.

- **Notification Helper Driver**: the development of this test module will be focused on invoking methods exposed by the *Notification Helper* in order to test the proper communication between this component and the gateway forwarding notifications to users, *Push Gateway* and *Email Gateway*.

- **Request Manager Driver**: the development of this test module will be focused on invoking methods exposed by the *Request Manager* and then verify the correct forwarding of requests coming from the outside to the right components. This module, in detail, check the communication with different components: *Login / Registration Controller*, *Ride Controller*, *Reservation Controller*, *Payment Controller* and finally *Cars Controller*.

- **Login/Registration Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Login/Registration Controller* and then verify the correct operation about login and registration.This module will furthermore verify the interaction with *Notification Helper* for submitting notifications and *Communication Manager* for interacting with other components and database.

- **Payment Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Payment Controller* in order to check the proper functioning the operations relating to the management of payments. This module will verify the correct connection with the *Communication Manager* and the *Payment Gateway* to test the correct routing of payment notifications to external company.

- **Payment Gateway Driver**: the development of this test module will be focused on invoking methods exposed by the *Payment Gateway* to test che correct communication with the external company.

- **Cars Controller Driver**: the development of this test module will be focused on invoking methods exposed by the *Cars Controller* in order to check the correct functioning of the operations relating to the management of sensors and actuators on the machines, as well as functions relating to maintenance. This module will furthermore test the interaction with differents components such as *Communication Manager*, *Maintenance Gateway* to test the correct forwarding of notifications to the maintenance company, and *Collector Data* to ensure the correct communication with cars.
- **Maintenance Gateway Driver**: the development of this test module will be focused on invoking methods exposed by the *Maintenance Gateway* to verify the correct communication with the maintenance company.
- **Collector Data Driver**: the development of this test module will be focused on invoking methods exposed by the *Collector Data* to test the proper communication with sensors and actuators on cars.

6.2 Test data

Used tools

- Github: for version control
- GoogleDoc: to write the document
- Draw.io: to create the diagrams
- LaTeX: to create the pdf

Work hours

- Emanuele Ricciardelli: $\sim$ 30 hrs.
- Giorgio Tavecchia: $\sim$ 30 hrs.
- Francesco Vetró: $\sim$ 30 hrs.