Politecnico di Milano

Software Engineering II project

PowerEnJoy

Integration Test Plan Document

Authors:
Davide Piantella
Mario Scrocca
Moreno R. Vendra

Professor:
Luca Mottola

January 22, 2017

version 1.1

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose of this document

The purpose of the *PowerEnJoy Integration Test Plan Document* is to describe how the integration testing is going to be accomplished for the PowerEnJoy system; it describes the entry criteria of each component for the integration, the integration strategy of the software and the subsystems, the actual integration sequence and how to verify that the components integrated perform as expected in each step of the sequence. The document also describes the tools and equipment required for testing and highlights the stubs/drivers and data required for the integration process.

1.2 Scope

PowerEnJoy is a car-sharing service that exclusively employs electric cars; we are going to develop a web-based software system that will provide the functionalities normally provided by car-sharing services, such as allowing the user to register to the system in order to access it, showing the cars available near a given location and allowing a user to reserve a car before picking it up. A screen located inside the car will show in real time the ride amount of money to the user. When the user reaches a predefined safe area and exits the car, the system will stop charging the user and will lock the car. The system will provide information about charging station location where the car can be plugged after the ride and incentivize virtuous behaviours of the users with discounts [1].

1.3 Glossary

The *PowerEnJoy: Requirements Analysis and Specification Document* [1] and the *PowerEnJoy: Design Document* [2] should be referenced for terms not defined in this section.

1.3.1 Acronyms

RASD: Requirements Analysis and Specification Document

DD: Design Document

ITPD: Integration Test Plan Document

API: Application Programming Interface

GPS: Global Position System

DB: DataBase

DBMS: DataBase Management System

GIS: Geographic Information System

1.3.2 Abbreviations

w.r.t.: with respect to

i.d.: id est

i.f.f.: if and only if

e.g.: exempli gratia

etc.: et cetera

1.4 Reference documents

- Context, domain assumptions, goals, requirements and system interfaces are all described in the *PowerEnJoy: Requirements Analysis and Specification Document* [1]
- Software design and architecture of the system are all described in the *PowerEnJoy: Design Document* [2]
- *Junit* unit testing framework documentation
  `http://junit.org/junit4/javadoc/latest/index.html`
- *Mockito* mocking framework documentation
  `http://static.javadoc.io/org.mockito/mockito-core/2.6.2/org/mockito/Mockito.html`
- *Arquillian* integration testing framework documentation
  `https://docs.jboss.org/author/display/ARQ/Reference+Guide`
- *JMeter* performance testing tool documentation
  `http://jmeter.apache.org/api/index.html`

1.5 Document overview

This document is structured as

1. **Introduction**: contains refereces, glossary, definitions, acronyms and abbreviations; it also explains the purpose and scope of this document
2. **Integration Strategy**: this section explains the integration strategy for the PoweEnJoy car sharing system, the reasons that brought us to choose such strategy and the integration sequence chosen.
3. **Individual Steps and Test Description**: for each step of the sequence describes how to verify that the components integrated perform as expected.
4. **Tools and Test Equipment Required**
5. **Program Stubs and Test Data Required**
6. **Appendices**: it contains references, software and tools used and hours of work per each team member

2 Integration Strategy

2.1 Entry Criteria

The following conditions have to be verified before entering the integration testing phase in order for it to produce meaningful results.

A fundamental initial criterion is that the development of the components and their functionalities proceeds together with the unit testing on such components, so that even when the components are not fully developed, they have already been tested at unit level w.r.t. the fully developed functionalities. Given the aforementioned criterion, before entering the integration phase, the development percentage of a component must be at least 90% and the main functionalities required to test its integration w.r.t. another component of the system must be fully developed.

The integration process can start when these conditions are met by the system development

- 100% of development on the *Data Provider* component
- 90% of development on the *Event Broker* and *Car Handler* components
- 80% of development on the *Rent Manager* component
- 60% of development on the *Maintenance Manager* component
- 40% of development on the *User Information Manager* and *Access Manager* components
- 30% of development on the *User Application Server*, *User Application*, *Customer Care Server* and *Customer Care Application* components

2.2 Elements to be integrated

In this section we identify the components and subcomponents to be integrated and add some details about their specific integration.

As specified in the design document, some component of the PowerEnJoy system such as the *Rent Manager* and the *Maintenance Manager* are rather complex and thus its internal subcomponents need to be integrated before proceeding with the integration of entire components with other components of the system.

The following components will be integrated to obtain the subsystem related to the Maintenance functionalities:

- *Maintenance Manager*
- *Event Broker*
- *Data Provider*
- *Car Handler*

The following components will be integrated to obtain the subsystem related to the User Application functionalities:

- *User Application*
- *UserApp Server*
- *Access Manager*
- *Rent Manager*
- *User Information Manager*
- *Data Provider*
- *Car Handler*
- *Event Broker*

The following components will be integrated to obtain the subsystem related to the Customer Care functionalities:

- *Customer Care Application*
- *Customer Care Server*
- *User Information Manager*
- *Data Provider*

Notes The external components like the *DBMS* and the external APIs are products that are not developed by us and that we suppose already completely tested; however we decided to include them in the integration testing by firstly testing the external component's interfaces on their own, and the by proceeding to test them when integrated with PowerEnJoy components.

In the same way the Maintenance Manger will be tested as a component integrated with our system and then it will be tested through the Maintenance API it provides to the maintenance company.

2.3 Integration Testing Strategy

Both bottom-up and top-down were taken into consideration when thinking of the integration strategy for the PowerEnJoy system; looking at the structure and composition of the system it is clear the most complex and relevant functionalities are located in the lower end tiers, which are also the most independent ones.

These considerations brought us to lean towards a bottom-up approach that will allow the lower end components to be tested and integrated before any other component; doing so any issue or problem in the integration phase of these components will be found at an early stage in the integration process, and so it will be easier to tackle it and solve it, while maintaining as much parallelism and decoupling as possible.

Given that some functionalities of the system are decoupled, during the integration process some steps of the integration sequence will have the same priority. In those cases the critical-module-first approach will be used in order to integrate and test the riskiest components first and solve their issues before they create worse problems in the integration process.

Given that we chose a bottom-up approach as stated in Section 2.1 the most independent components are the only ones that needs to be fully developed at the time of starting the integration process, so if the software development of the PowerEnJoy system starts from the most independent components, it would allow to start and proceed with the integration phase before the development process is completed.

2.4 Sequence of Component Integration

This section describes the proposed plan for the integration test phase of the PowerEnJoy system.

2.4.1 Overall Component Integration Diagram

The Figure 1 shows the needed precedences in the integration phase between the main components of the system taking into account the integration testing strategy chosen.

NOTES In the following diagrams we represent with arrows the precedences needed between the components, the purpose is to indicate that following the strategy chosen a component can be integrated with another one only if that component has already been integrated with all the components it needs to be integrated with (arrows point from the *need integration* component to the *needed integration* one; numbers are shown to clearify order of the integration and steps that are possibly made in parallel).

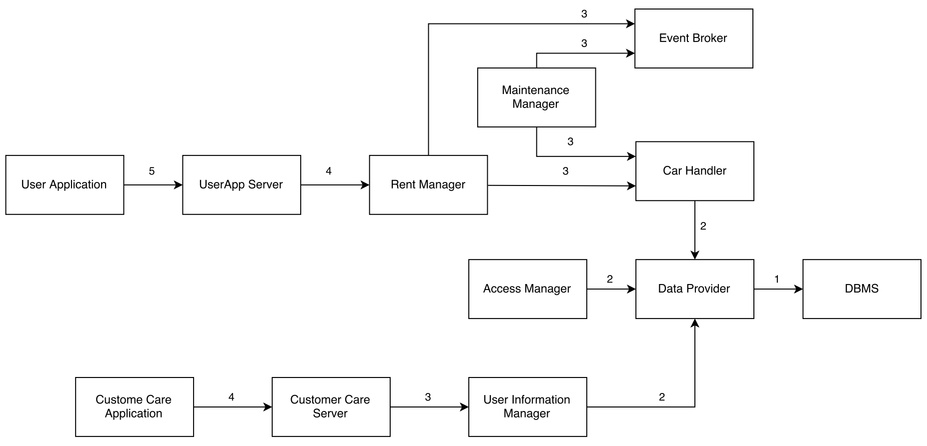

Figure 1: *Overall Component Integration Diagram*

2.4.2 Subcomponent Integration

In the *PowerEnJoy: Design Document* [2] some high-level components are described deeply through the definition of the subcomponents composing them. In this section are shown the diagrams of the precedences between the subcomponents needed to integrate them. We assume that when the high-level component enters the integration sequence presented they have been fully tested inside considering them as a single component unit-level tested to be integrated.

RentManager The diagram in Figure 2 shows the needed precedences in the integration phase between the *RentManager* subcomponents.

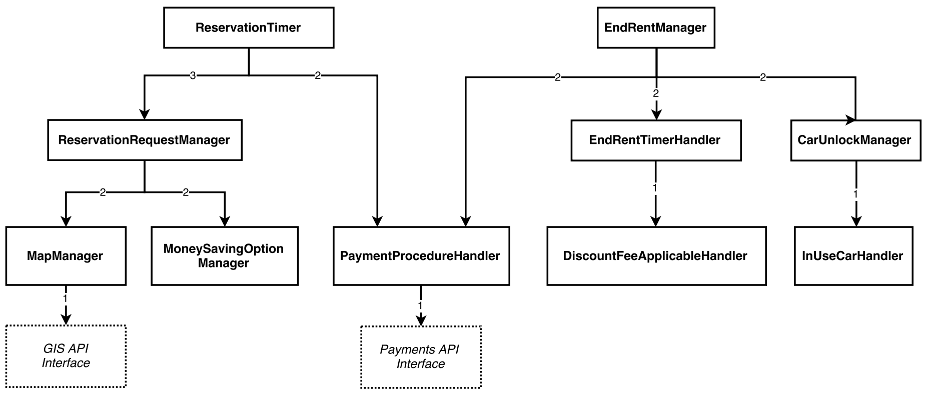

Figure 2: *RentManager subcomponents integration*

MaintenanceManager The diagram in Figure 3 shows the needed precedences in the integration phase between the *MaintenanceManager* subcomponents.

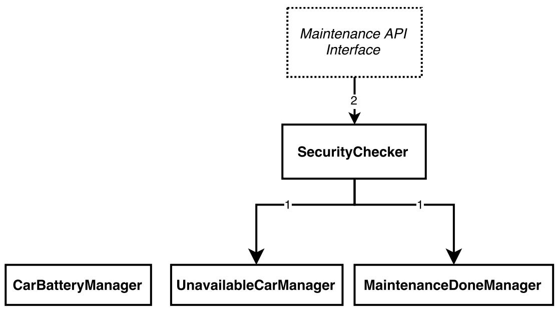

Figure 3: *MaintenanceManager subcomponents integration*

UserInformationManager The diagram in Figure 4 shows the needed precedences in the integration phase inside the *UserInformationManager* subcomponents.

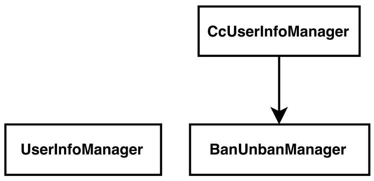

Figure 4: *UserInformationManager subcomponents integration*

2.4.3 Component Integration

This section describes the sequence proposed to plan the integration test of all the components of the PowerEnJoy system. As specified in the Integration Testing Strategy section we decide to start the bottom-up integration starting from the components related to the critical functionalities of the system (such as the communication with cars).

Even if the following steps are proposed sequentially, it is clear that the sequence proposed leaves the possibility to make some integration steps in parallel, always taking into account the needed precedences between the components shown in the Overall Component Integration Diagram.

Notes for diagrams In each diagram we assume that represented components have been already integrated as specified in precedent steps.

1. DataProvider The first step of the integration sequence is to integrate the *DataProvider* component with the DBMS component.

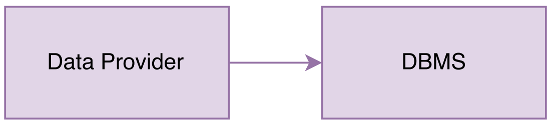

Figure 5: *DataProvider integration*

2a. EventBroker and CarHandler Integrate the *EventBroker* and the *CarHandler* with the external API provided by cars, testing the interfaces between our system and them. Integrate the *CarHandler* with the *DataProvider* component.

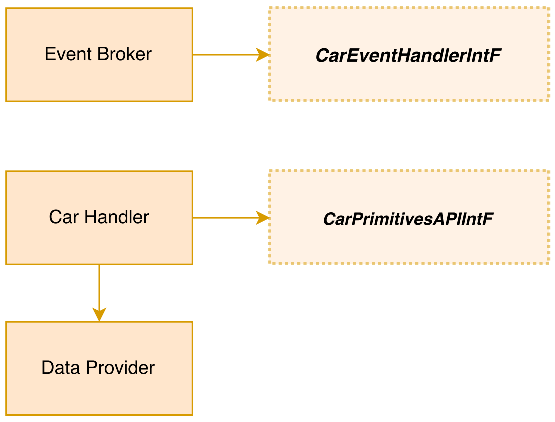

Figure 6: *EventBroker and CarHandler integration*

2b. UserInformationManager and AccessManager The integration between the *UserInformationManager* and the *DataProvider* components and between the *AccessManager* and the *DataProvider* components are here reported because they can be made at this point of the sequence (given the integration tests already completed), however it is suggested to first complete the critical integration of the *RentManager* component.

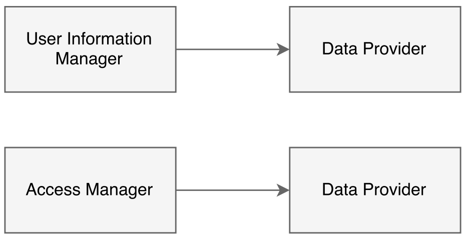

Figure 7: *UserInformationManager and AccessManager integration*

3a. RentManager Integrate the *EventBroker*, the *CarHandler* and the *DataProvider* with the *RentManager*. We also integrate the *RentManager* component with the external APIs related to the GIS and the Payment System, testing the interfaces between our system component and them.

This integration step is really critical to the functionality provided by our system and a lot of care must be given in the integration tests of the *RentManager* on the communication with cars and the consistency between information provided by cars and information in our database.

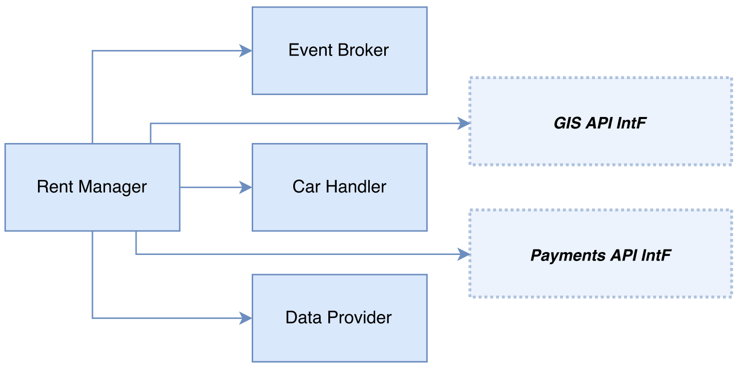

Figure 8: *RentManager integration*

3b. MaintenanceManager Integrate the *EventBroker*, the *CarHandler* and the *DataProvider* with the *MaintenanceManager* obtaining the complete test of the subsystem related to the functionalities to manage the maintenance API.

Also the integration between the maintenance subsystem and the API interface exposed externally has to be tested, checking the token mechanism and its correctly behaviour for an API-user external to the system (see the *PowerEnJoy: Design Document* [2] for more details).

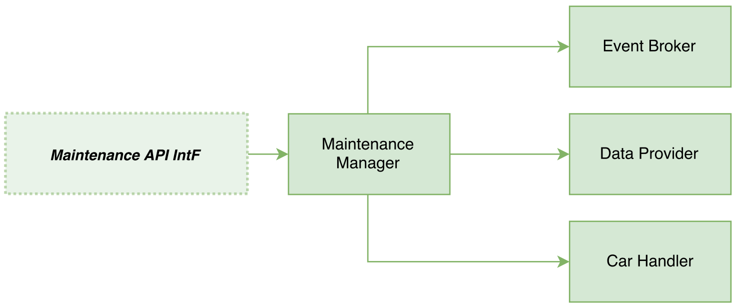

Figure 9: *MaintenanceManager integration*

3c. CustomerCare Application and Server Integrate first the *CustomerCareServer* with the *UserInformationManager* and the *DataProvider* components and secondly test the integration between the *CustomerCareApplication* and the *CustomerCareServer* obtaining the complete test of the subsystem related to the functionalities provided to a customer care operator.

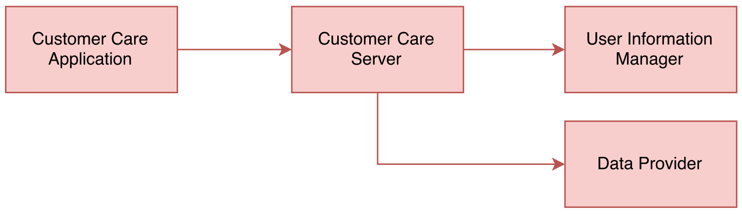

Figure 10: *CustomerCare Application and Server integration*

4. User Application and Server Integrate first the *UserAppServer* with the *AccessManager*, *RentManager* and *UserInformationManager* components and secondly test the integration between the *UserAppServer* and the *UserApplication* obtaining the complete test of the subsystem related to the functionalities provided to a user of the *PowerEnJoy* system.

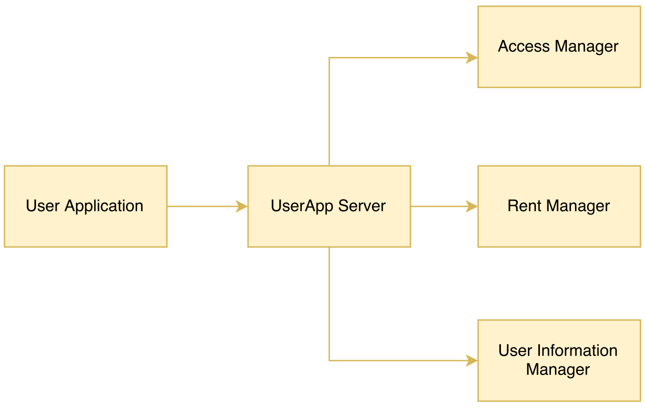

Figure 11: *User Application and Server integration*

2.4.4 Subsystem Integration Sequence

Once the three major subsystem have been fully integrated and tested, integrate them together to test the whole system and its functionalities.

Test the integration of the maintenance-related subsystem (Figure 9), the user-related subsystem (Figure 11) and the customer-care-related subsystem (Figure 10).

This step may be split in different kind of tests on local machines but it also should be done on the real architecture that we will use for our system, in order to provide reliable test data also about performances and network issues.

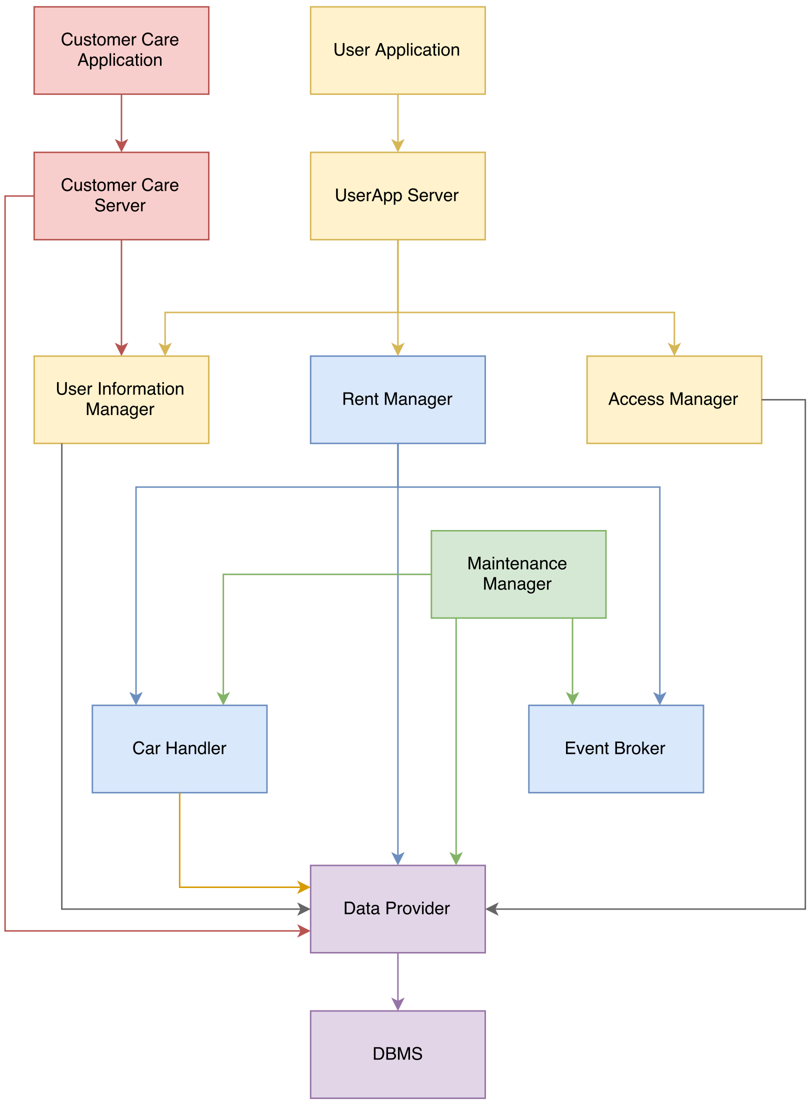

Figure 12: *Subsystems Integration*

3 Individual Steps and Test Description

This section describes the main tests that will be used to verify that the elements integrated, as specified in the integration sequence, behave as expected.

3.1 Test Description

This section describes the integration tests for most pair of components to be integrated; the integration test are described through the expected behaviour of the methods of the second component against the calls of the first in terms of relations between *input* and *expected output*.

3.1.1 RentManager, DataProvider

getFromFile(searchFor)

Input	*Expected output*
A null searchFor	A NullArgumentException is raised.
A searchFor that correspond to a non existing file	A FileNotFoundException is raised
A searchFor value that does not correspond to any known file	An InvalidArgumentValueException is raised
A searchFor value corresponds to a known existing file	The pointer to the correct file is returned

Table 1: *getFromFile* test description

createCarJPA(carID)

Input	*Expected output*
A carID which does not correspond with any car in the DB	An CarNotFoundException is raised
A carID which identifies a car in the DB	A JPA class mapped to the car identified by the carID in the DB is returned

Table 2: *createCarJPA* test description

createReservationJPA(carID, userID)

Input	*Expected output*
A carID which does not correspond to any car in the DB	A CarNotFoundException is raised
A userID does not correspond to any user in the DB	A UserNotFoundException is raised
A userID which identifies a user that has already more than zero reservation	An AtMostOneReservationException is raised
A userID which identifies a user that is performing a rent	A NoReservationDuringARentException is raised
A carID which identifies a car in the DB with the carStatus attribute that is not *Available*	A NotReservableCarException is raised
A carID which identifies a car in the DB with the carStatus attribute that is *Available* and a userID which identifies a user in the DB that has no active reservations or rents	A new reservation for the specified car and user and with the current timestamp is created in the DB

Table 3: *createReservationJPA* test description

banUser(userID, reason)

Input	*Expected output*
A null reason	A NullArgumentException is raised
An empty reason	An InvalidArgumentValueException is raised
A userID which does not identify any user in the DB	A UserNotFoundException is raised
A userID which identifies a user in the DB already banned	A UserAlreadyBannedException is raised
A not empty reason and userID which identifies a not banned user in the DB	The banned attribute is set to true and the reason attribute is updated as specified for the user identified by the userID parameter

Table 4: *banUser* test description

createPaymentJPA(rent, discounts, fees, paymentStatus)

Input	*Expected output*
A null rent and/or a null list of discounts and/or a null list of fees and/or a null paymentStatus	A NullArgumentException is raised
A rent with endTimestamp and/or endLocation null attributes	An InvalidArgumentValueException is raised
An unknown paymentStatus	An InvalidArgumentValueException is raised
A not null rent, a known payment status and • An empty list of discounts and/or an empty list of fees **or** • A not empty list of discounts and/or a not empty list of fees	The fee and discount to apply are computed, a new payment is inserted properly in the DB with the paymentStatus attribute as *Pending*

Table 5: *createPaymentJPA* test description

findReservedCar(userID)

Input	*Expected output*
A userID which does not identify any user in the DB	A UserNotFoundException is raised
A userID which identifies a user in the DB without an active reservation	A NoReservedCarException is raised
A userID which identifies a user in the DB with an active reservation	A CarJPA related to the car currently reserved by the user associated with the userID argument is returned

Table 6: *findReservedCar* test description

createRentJPA(startLocation, startTimestamp, car, msoStation)

Input	*Expected output*
A null startLocation and/or a null startTimer and/or a null car	A NullArgumentException is raised
A startTimestamp in the future	An InvalidArgumentValueException is raised
A wrong format of startTimestamp	An InvalidArgumentValueException is raised
A wrong format of startLocation	An InvalidArgumentValueException is raised
A car JPA class mapped to a car in the DB with its carStatus attribute that is not *Reserved*	A CarNotReservedException is raised
A msoStation parameter which does not identify any charging station in the DB	A ChargingStationNotFoundException is raised
Proper startLocation, startTimestamp and car parameters, null msoStation	The reservation associated to the car is removed, the car status is set to *InUse*, a new rent (paired with the user that has reserved the car) is inserted properly in the DB with endTimestamp, endLocation and moneySavingOption attributes set to null.
Proper startLocation, startTimestamp and car parameters, msoStation parameter valid and not null	The reservation associated to the car is removed, the car status is set to *InUse*, a new rent (paired with the user that has reserved the car) is inserted properly in the DB with endTimestamp, endLocation attributes set to null.

Table 7: *createRentJPA* test description

3.1.2 RentManager, CarHandler

getBattery(listOfCars)

Input	*Expected output*
A null listOfCars	A NullArgumentException is raised
A listOfCars with null element(s)	A InvalidArgumentValueException is raised
An empty listOfCars	An InvalidArgumentValueException is raised
A not empty listOfCars without null elements	The input list of cars is returned with the battery level updated for each car

Table 8: *getBattery* test description

unlock(carJPA)

Input	*Expected output*
A null carJPA	A NullArgumentException is raised
A carJPA which maps to a car whose carStatus attribute is not *Reserved*	A CarNotReservedException is raised
A carJPA which maps to a car whose carStatus attribute is *Reserved*	The car is unlocked and TRUE is returned

Table 9: *unlock* test description

trigger(carJPA, listOfTriggers)

Input	*Expected output*
A null carJPA and/or a null listOfTriggers	A NullArgumentException is raised
An empty listOfTriggers	An InvalidArgumentValueException is raised
A listOfTriggers with null element(s)	A InvalidArgumentValueException is raised
A valid carJPA and listOfTriggers	The specified triggers are enabled in the specified car and TRUE is returned

Table 10: *trigger* test description

removeTrigger(carJPA, listOfTriggers)

Input	*Expected output*
A null carJPA and/or a null listOfTriggers	A NullArgumentException is raised
An empty listOfTriggers	An InvalidArgumentValueException is raised
A listOfTriggers with null element(s)	A InvalidArgumentValueException is raised
A valid carJPA and listOfTriggers	The specified triggers are disabled in the specified car and TRUE is returned

Table 11: *removeTrigger* test description

getParameters(carJPA, listOfParameters)

Input	*Expected output*
A null carJPA and/or a null listOfParameters	A NullArgumentException is raised
An empty listOfParameters	An InvalidArgumentValueException is raised
A listOfParameters with null element(s)	A InvalidArgumentValueException is raised
A valid carJPA and listOf-Parameters	The values of parameters requested for the specified car are returned

Table 12: *getParameters* test description

3.1.3 MaintenanceManager, DataProvider

fixFailureTagAvailable(failureID)

Input	*Expected output*
A failureID which does not identify any failure in the DB	A FailureNotFoundException is raised
A failureID which identifies an already fixed failure in the DB	A FailureAlreadyFixedException is raised
A failureID which identifies a pending failure in the DB	The failure's fixedTimestamp DB attribute is updated with the current timestamp, the carStatus attribute of the car related to the failure is set to *Available*

Table 13: *fixFailureTagAvailable* test description

createFailure(carID, reason, timestamp)

Input	*Expected output*
A null listOfParameters and/or a null timestamp	A NullArgumentException is raised
An empty reason	An InvalidArgumentValueException is raised
A timestamp in the future	An InvalidArgumentValueException is raised
A carID which does not identify any car in the DB	A CarNotFoundException is raised
A carID which identifies a car in the DB whose carStatus attribute is not *NotAvailable*	A CarNotUnavailableException is raised
A carID that identifies a car in the DB which is already paired with a pending failure	A AtMostOnePendingFailureException is raised
A carID that identifies a car in the DB whose carStatus attribute is *NotAvailable* and which is not already paired with a pending failure	A new failure for the specified car and reason and with the current timestamp is created in the DB

Table 14: *createFailure* test description

3.1.4 MaintenanceManager, CarHandler

getSoftwareKeys(carsList)

Input	*Expected output*
A null carsList	A NullArgumentException is raised
An empty carsList	An InvalidArgumentValueException is raised
A carsList with null element(s)	An InvalidArgumentValueException is raised
A carsList which contains a carID that does not identify any car in the DB with pending failures	A FailureNotFoundException is raised
A carsList which contains only carIDs that identify cars in the DB with pending failures	Each specified car is asked for the unlocking software key, a list of those cars paired with their related software key is returned

Table 15: *getSoftwareKeys* test description

3.1.5 UserAppServer, RentManager

getMapAvailableCars(position)

Input	*Expected output*
A null position	A NullArgumentException is raised
A wrong format of position	An InvalidArgumentValueException is raised
A well-formatted position	A reference to a map is returned; the map shows the position of the safe areas, charging stations, input position, locations and details (see [2]) of available cars within a radius of 2 Km. The batteryLevel and lastSeenTime attributes of these cars are updated if elder than 2 hours

Table 16: *getMapAvailableCars* test description

reserveCar(userID, carID, MSODestination)

Input	*Expected output*
A null MSODestination	A NullArgumentException is raised
A carID which does not identify any car in the DB	An CarNotFoundException is raised
A userID which does not identify any user in the DB	An UserNotFoundException is raised
A userID which identifies a user in the DB that has already more than zero reservation	An AtMostOneReservationException is raised
A userID which identifies a user in the DB that is performing a rent	An NoReservationDuringARentException is raised
The carStatus DB attribute of the car identified by the carID is not *Available*	A NotReservableCarException is raised
The carStatus DB attribute of the car identified by the carID is *Available* and the user identified by the userID has no active reservations or rents, MSODestination is empty	• a new reservation for the specified car and user and with the current timestamp is created in the DB • the carStatus attribute of the specified car changes to *Reserved*
The carStatus DB attribute of the car identified by the carID is *Available* and the user identified by the userID has no active reservations or rents, MSODestination is not empty	• a new reservation for the specified car and user and with the current timestamp is created in the DB • the carStatus attribute of the specified car changes to *Reserved* • a charging station is computed according to the proper algorithm and set as destinationChargingStation in the reservation table

Table 17: *reserveCar* test description

The **carUnlock(userID)** method can throw any exception specified in
Table 6 findReservedCar(userID),
Table 2 createCarJPA(carID),
Table 9 unlock(carJPA),
Table 10 trigger(carJPA, listOfTriggers),
Table 7 createRentJPA(startLocation, startTimestamp, car, msoStation).

carUnlock(userID)

Input	*Expected output*
A null position	A NullArgumentException is raised
A userID which does not identify any user in the DB	A UserNotFoundException is raised
A userID which identifies a user in the DB without an active reservation	A NoReservedCarException is raised
A position that correspond to a location further than 5 Km away from the reserved car	A NotEnoughCloseException is raised
A userID which identifies a user in the DB with an active reservation and a position that correspond to a location closer than 5 Km to the reserved car	*union of the aforementioned methods' outputs in case of standard flow* (when the system is notified of the reserved car's engine ignition the createRentJPA method is called)

Table 18: *carUnlock* test description

3.1.6 CustomerCareServer, DataProvider

See Table 4 for the test description of **banUser(userID, reason)** method.

unbanUser(userID)

Input	*Expected output*
A userID which does not identify any user in the DB	A UserNotFoundException is raised
A userID which identifies a user in the DB that is not banned	A UserNotBannedException is raised
A userID which identifies a banned user in the DB	The banned attribute is set to false and the reason attribute is set to null

Table 19: *unbanUser* test description

3.1.7 Other Tests

In this section we describe integration tests related to components whose methods were not detailed in the *Design Document* [2] since they are mostly basic functions related to user and data management.

AccessManager, DataProvider

- **Login functionality**: test if the *DataProvider* allows *AccessManager* to check the username and password given by a user, corresponds to a user correctly registered, with a null token associated(email confirmed) and a *not banned* state.
- **Registration functionality**: test if the *DataProvider* allows *AccessManager* correctly creates a new user record in the database when a registration request is received.

UserInformationManager, DataProvider

- **Query functionality**: test if the *DataProvider* allows the *UserInformationManager* to retrieve rent and payment history of a user and to retrieve or edit personal info of a user; moreover test if the behaviour of the integrated components is correct both in the case the *UserInformationManagerDriver* behaves as a user and in the case it behaves as a customer care operator.

UserAppServer, AccessManager

- **Login functionality**: test if the *AccessManager* confirms that a user can log into the system only if username and password given by *UserAppServer* corresponds to a user correctly registered, with a null token associated(email confirmed) and a *not banned* state.
- **Registration functionality**: test if the *AccessManager* allows *UserAppServer* to make requests about the registration of a user checking the information provided as defined in the *RASD document*[1] and correctly stores the user information.

UserAppServer, UserInformationManager

- **Information retrieval functionality**: test if the *UserInformationManager* correctly provides the *UserAppServer* with user's personal information, his rent history and his payment history.
- **Information edit functionality**: test if the *UserInformationManager* correctly allows *UserAppServer* to modify user's editable personal information.

CustomerCareServer, UserInformationManager

- **Information retrieval functionality**: test if the *UserInformationManager* correctly provides the *CustomerCareServer* with a list of users, the Customer Care selection of a user personal information, his rent history and his payment history.

4 Tools and Test Equipment Required

In this section the tools needed for testing and the test equipment will be described together with the reasons why those are needed.

4.1 Tools Required

4.1.1 Unit Testing

The tools chosen for unit testing are JUnit for the testing framework in combination with Mockito as a mocking framework.

JUnit was chosen because the system will be developed using the Java language and because it will allow the developers to incrementally build test suites, to measure progress and detect unintended side effects.

Mockito instead will allow the developers to test small functionalities even when these have many dependencies. It will also allow them to ensure that the interaction between the components to be tested and the mocked ones corresponds to the expected behaviour of the system.

4.1.2 Integration Testing

As for integration testing the chosen tools are again JUnit and Mockito, but in this case in combination with the Arquillian integration testing framework. Arquillian adds to the convenience of JUnit and Mockito the capability to test the functionalities when run in the context of a container, an instance of the GlassFish server in our case.

This will allow our developer to really ensure that the components that are being integrated interact in the correct way among them and w.r.t. their container.

4.1.3 Performance Testing

For performance testing the choice is the load testing tool JMeter. It will allow the developers to see how much load the system is able to handle, and how it performs while the load increases. It is also highly configurable since, by creating different test plans, different functionalities can be tested.

4.2 Test Equipment

The different tiers of our system will run on different machines, as represented in the Deployment Diagram of the *PowerEnJoy: Design Document* [2], therefore also for the integration testing it will be fundamental to ensure that the system works properly when running in that configuration.

Backend The machines needed to test the backend will be:

- Three computers capable of running an instance of the GlassFish Server each
- One computer capable of running the DBMS

Different configurations of increasing complexity may be used in order to find and solve issues related to the deployment of the system during testing and to test the performance increase in different configurations.

Frontend The machines needed to test the frontend will be:

- One computers capable of running the latest versions of the most used web browsers
- Smartphones of different screen sizes in order to ensure that the website is always correctly rendered and shown on such devices.

Since our application is fully web based, these will be the only machines needed to access and test the frontend functionalities of our system, both for the customer care and the regular user.

5 Program Drivers and Test Data Required

Based on the testing strategy chosen and tests description presented, this section identifies program drivers and tests data required for the integration phase.

5.1 Drivers Required

As specified in the Integration Testing Strategy section we decided to use a bottom-up strategy to test the integration of the *Power EnJoy* system components. This type of strategy implies the necessity to build drivers (pieces of software) to simulate components, not already integrated, invoking methods on the integrated components we are currently testing.

The main drivers needed to complete the integration test plan are:

- **DataProviderDriver**: this driver is used to invoke methods exposed by the *DataProvider* component in order to test its integration with the DBMS component
- **CarHandlerDriver**: this driver is used to invoke methods exposed by the *CarHandler* component in order to test its integration with the *DataProvider* component and the external API provided by cars
- **EventBrokerDriver**: this driver is used to act like a subscriber of the *EventBroker* component in order to test its integration with its related interface (**note** so it must also allows a stub-like behaviour in order to verify subscribers are correctly notified of events published on the *CarEventHandlerIntF*)
- **UserInformationManagerDriver**: this driver is used to invoke methods exposed by the *UserInformationManager* component in order to test its integration with the *DataProvider* component
- **AccessManagerDriver**: this driver is used to invoke methods exposed by the *AccessManager* component in order to test its integration with the *DataProvider* component
- **RentManagerDriver**: this driver is used to invoke methods exposed by the *RentManager* component in order to test its integration with the *EventBroker*, *CarHandler*, *DataProvider* components and the GIS and Payments API interfaces
- **MaintenanceManagerDriver**: this driver is used to invoke methods exposed by the *MaintenanceManager* component in order to test its integration with the *EventBroker*, *CarHandler* and *DataProvider* components (**note** it may also be useful to develop a MaintenanceAPIIntfDriver in order to test also invocation of interface methods from the point of view of an external user of the API)
- **UserAppServerDriver**: this driver is used to invoke methods exposed by the *UserAppServer* component in order to test its integration with the *RentManager*, *UserInformationManager* and *AccessManager* components

- **CustomerCareServerDriver**: this driver is used to invoke methods exposed by the *CustomerCareServer* component in order to test its integration *UserInformationManager* and *DataProvider* components

5.2 Data Required

As specified before in this document, the DataProvider component must be integrated with the DBMS at the beginning of the integration sequence, furthermore referential integrity is always maintained because *InnoDB* is used as DB engine with the usage of foreign keys constrains, therefore we can consider the data retrieved from the DataProvider component always reliable and well-formed.

The database used during the tests must have the same tables and structure as the production one.

Well formed XML files with the location of charging stations (at least 10) and safe areas (at least 2) are also required.

In order to cover the wide variety of possible conditions, the data inserted in the test DB must include *at least*:

- 40 registered users, whereof
  - 5 banned users
  - 30 not banned users
  - 5 users with email confirmation pending
- 35 cars, whereof
  - 5 reserved
  - 5 not available
  - 10 in use
  - 15 available, whereof
    - 2 with lastSeenTime = 2 hours
    - 5 with lastSeenTime < 2 hours
    - 8 with lastSeenTime > 2 hours
- 10 failures, whereof
  - 5 pending (whereof 2 of the same car)
  - 5 fixed (whereof 2 of the same car)
- 5 active reservation, whereof
  - 2 with the money saving option
  - 3 without the money saving option
- 40 rents, whereof
  - 10 active rents
  - 30 concluded rents (whereof 5 of the same car, 3 of the same user, 5 of banned users)
- 5 fees

- 5 discounts
- 35 payments (whereof 5 of the same user), whereof
  - 2 with status = pending
  - 5 with status = rejected
  - 10 without the money saving option
  - 10 with the money saving option
  - 5 related to expired reservations
  - 30 related to concluded rents, whereof
    - 5 without fees or discounts related
    - 5 with 1 fee and 0 discounts related
    - 5 with 0 fees and 1 discount related
    - 5 with 3 fees and 0 discount related
    - 5 with 0 fees and 3 discount related
    - 5 with more than 3 fees and more than 3 discounts related

Appendices

A Software and tools used

For the development of this document we used

- LaTeX as document preparation system
- Git & GitHub as version control system
- Draw.io for graphs

B Hours of work

This is the amount of time spent to redact this document:

- Davide Piantella: ∼ 15 hours
- Mario Scrocca: ∼ 20 hours
- Moreno R. Vendra: ∼ 18 hours

C Changelog

- **v1.0** January 15, 2016
- **v1.1** January 22, 2016
  - Fixed typos and readability in Table 7 and Table 18
  - Added reservation deletion in Table 7

References

[1] D. Piantella, M. Scrocca, M.R. Vendra, *PowerEnJoy: Requirements Analysis and Specification Document*, Politecnico di Milano - Software Engineering II Project, 2016

[2] D. Piantella, M. Scrocca, M.R. Vendra, *PowerEnJoy: Design Document*, Politecnico di Milano - Software Engineering II Project, 2016