POLITECNICO
MILANO 1863

Politecnico di Milano
A.A. 2016–2017
Software Engineering 2: "PowerEnJoy"

Integration Test Plan Document

Pietro Ferretti, Nicole Gervasoni, Danilo Labanca

January 15, 2017

Contents

1 Introduction

1.1 Revision History

- ITDP v1.0, published on January 15, 2017

1.2 Purpose and Scope

1.2.1 Purpose

The purpose of this document is to provide a complete description of the integration testing plans for PowerEnJoy. It will focus on testing the proper behavior of the software by checking the interoperability between its components. It is intended for developers and for any team member involved in the testing process.

In this document we will illustrate:

- the initial conditions needed to start integration testing
- which components will be integrated
- the order that the tests will follow
- a detailed description of the tests
- the equipment and data required for integration testing

1.2.2 Scope

The aim of this project is to specify and design a new digital management software for PowerEnJoy, a car-sharing service that employs electric cars only.

PowerEnJoy will offer a very valuable service to its users, letting them borrow cars to drive around the city freely, as an alternative to their own vehicles and public transport. Among the advantages of using PowerEnJoy we can note being able to find available cars in any place that is served by our system and having dedicated spots to park in (namely, PowerEnJoy's power grid stations). Furthermore, thanks to the fact that all the cars that we provide are electrically powered, PowerEnJoy is also very environmentally friendly.

1.3 List of Definitions and Abbreviations

1.3.1 Definitions

- *User*: a person that is registered to the system. Users can log in to the system with their email or username and their password. Their first name, last name, date of birth, driving license ID are stored in the database.
- *Safe area*: a location where the user can park and leave the car. Users can end their ride and park temporarily only in these locations. The set of safe areas is predefined by the system.
- *Power grid station* or *Charging station*: a place where cars can be parked and plugged in. While a car is plugged in a power grid station its battery will be recharged. Power grid stations are by definition safe areas.
- *Available car*: a car that is currently not being used by any user, and has not been reserved either. Available cars are in good conditions (not dirty nor damaged) and don't have dead batteries.
- *Reservation*:
  - the operation of making a car reserved for a user, i.e. giving permission to unlock and use the car only for that user, forbidding reservations by other users.
  - the time period between the moment a reservation is requested and the moment the user unlocks the car, or the reservation is canceled.
- *Ride*: the time period from the moment a reserved car is unlocked to the moment the user notifies that he wants to stop using the car and closes all the doors. A ride doesn't stop when a car is temporarily parked, but continues until the user chooses to leave the car definitely.
- *Temporary parking*: the act of parking a car in a safe area and, after notifying the system, locking it and leaving it for a finite amount of time. The user that does this retains the right to use the car and can unlock it later to use it again.
- *Bill*: a record of the money owed by the user at the end of a ride.
- *Suspended user*: a user that cannot reserve or use cars. Usually users are suspended because they have outstanding bills that have not been paid.
- *Payment method*: a way to transfer money from the user to the system. Our system will only accept credit cards and online accounts like Paypal.

1.3.2 Acronyms

- **ITPD**: Integration Test Plan Document
- **DD**: Design Document
- **RASD**: Requirements Analysis and Specification Document
- **DB**: Database
- **CVV**: Card Verification Value
- **DOB**: Date of birth
- **PGS**: Power Grid Station
- **GPS**: Global Positioning System
- **API**: Application Programming Interface
- **ISDTN**: International Standard Date and Time Notation

1.4 List of Reference Documents

- Requirements analysis and specification document: "RASD.pdf"
- Design document: "DD.pdf"
- Project description document: "Assignments AA 2016-2017.pdf"
- Example document: "Integration Testing Example Document.pdf"

2 Integration Strategy

2.1 Entry Criteria

Before starting the integration testing phase specific conditions concerning the whole project development must be met.

It is fundamental that the Requirements Analysis and Specification Document and the Design Document have been properly written and completed.

Regarding the code development, it is necessary to fully complete

- the Resource Subsystem
- the Car Internal System
- the Model Component in the Application Subsystem.

All the other components will be integration tested as soon as they are unit tested, following the sequence described in subsection 2.4. Furthermore, each component has to be successfully unit tested before being involved in the integration testing.

2.2 Elements to be Integrated

As stated in the Design Document (paragraph 2.2), the system is based on the cooperation of two parts, the *Application Subsystem* and the *Resource Management Subsystem*. The first handles all the operations related to the user applications and interfaces, while the second takes care of keeping track of all the automatic updates from the sensors in the cars and the charging stations.

The testing process will verify the correctness of the integration between these macro-components after checking the proper cooperation of smaller components inside each subsystem.

Finally, the interaction between components on the client/server level will be tested, in particular between

- the Web Application and the Application Subsystem
- the Car Application and the Application Subsystem

Concerning the external APIs, we expect them to work properly so we will only check, by unit testing, that the components correctly execute the API calls.

Application Subsystem Components to be tested:

1. Registration
2. Account Manager
3. Car Search
4. PGS Search

5. Safe Areas Search
6. Payment
7. Reservation Manager
8. Unlocking
9. Parking Ride Manager
10. Ride Manager
11. Report Manager
12. Request Dispatcher
13. Model

We will test that the Request Dispatcher interacts with each component from 1 to 11 in the proper way and that these components cooperate as expected with the Model. In addition, we will test the integration between the Model component and the DBMS.

Resource Management Subsystem Components to be tested:

1. Car Manager
2. PGS Manager
3. Car Handler

We will test the cooperation between the Car Manager and the Car Handler components, and between the PGS Manager and the DBMS.

2.3 Integration Testing Strategy

We believe that the most efficient way to test the integration between PowerEnJoy's software components is the *bottom up approach*. Using this strategy, we will be able to test the cooperation between different units as soon as they are fully developed without waiting for the whole subsystem to be ready. This method will also highlight all the possible integration issues while the software is still under development, allowing quicker smaller fixes. We will start the process from independent components which don't have any dependencies and we will build further integration tests on the already tested software. As specified in the Design Document, several components of the Application Subsystem depend on the Resource Management Subsystem, therefore the testing phase will start with this subsystem.

2.4 Sequence of Component / Function Integration

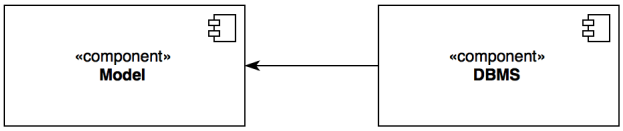
In this section we will describe in which order the components of our system will be coupled together to be tested and integrated. First we will integrate the components in each subsystem separately, then the two subsystems (the Application Subsystem and the Resource Management Subsystem) will be combined together.

We will use diagrams to make the dependencies between components clearer. Arrows from a component to another imply that the first component is necessary for the second one to work correctly.

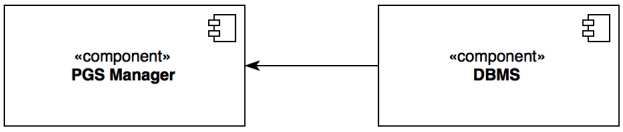
2.4.1 Software Integration Sequence

Following our choice of using a bottom-up approach, integration testing will start from two base components: the DBMS and the Car Internal System. The DBMS is free from dependencies and can be easily used as a starting point. The Car Internal System on the other hand, even if it is a prerequisite for many of the system's functions to work, requires an interface to the Resource Management Subsystem that we haven't integration tested yet. Fortunately the interface is used only for sending updates about its status, so the interface functions can be easily replaced by an appropriate stub.

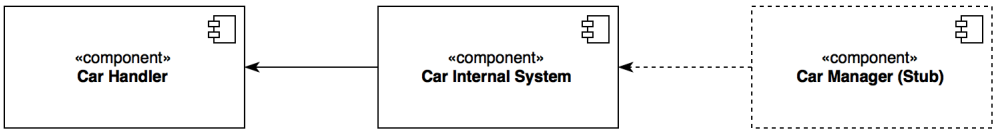
Model The first two elements to be integrated will be the Model and the DBMS. The Model is the foundation of the Application Subsystem and integrating it will let us test all the other components in the subsystem.



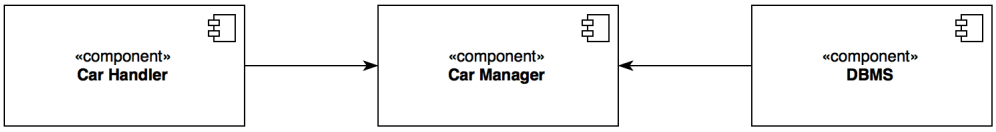
Power Grid Stations Manager On the Resource Management Subsystem we can start from integrating the Power Grid Stations Manager with the DBMS. The PGS Manager is almost fully independent from the other components and can be tested easily.



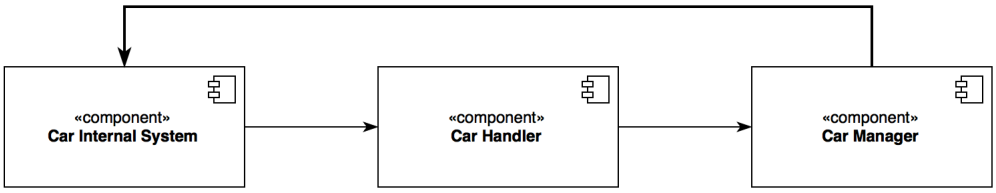
Car Handler The Car Handler is the component that is in charge of issuing commands to the car. We can start testing its behavior as soon as the Car Internal System is fully unit tested. We will make use of a stub to simulate the Resource Subsystem Interface.



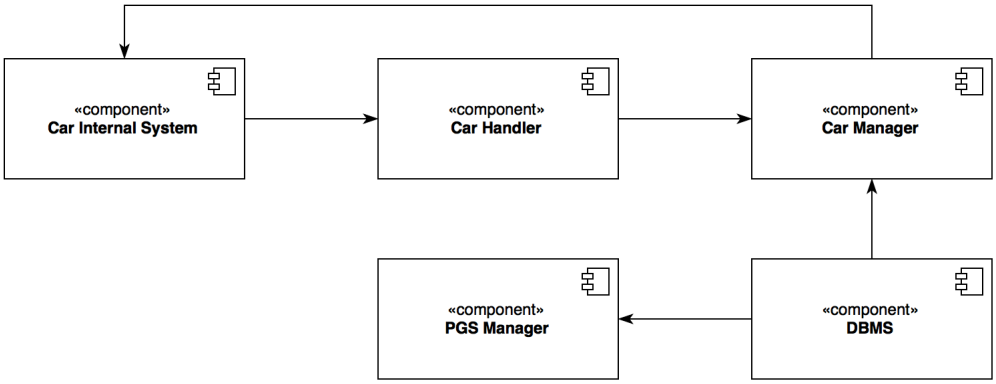
Car Manager After the Car Handler is integrated we can finally incorporate the Car Manager, the component doing the heavy lifting in the Resource Management Subsystem.



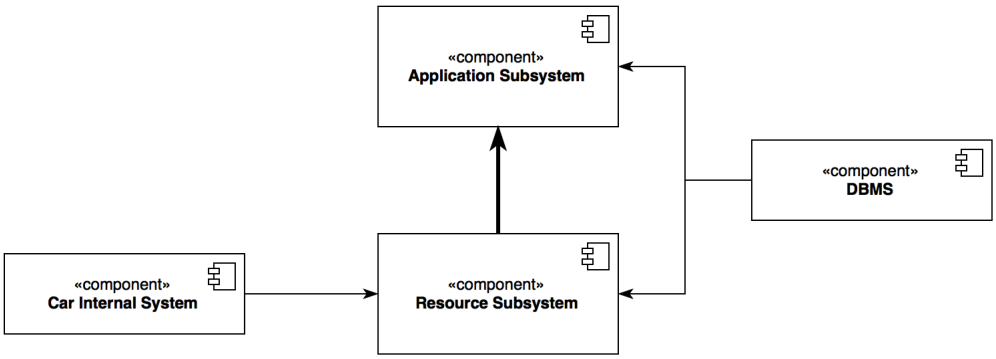
Once the Car Manager has been integration tested, we can actually test the interaction between the Car Internal System and the Resource Subsystem Interface.



At this point the Resource Management Subsystem is fully integrated.

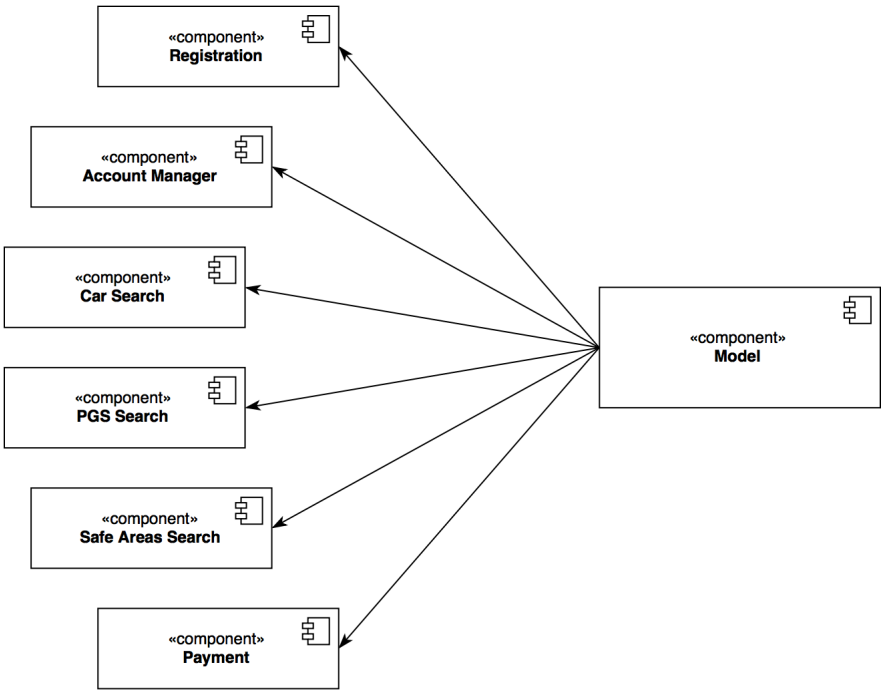


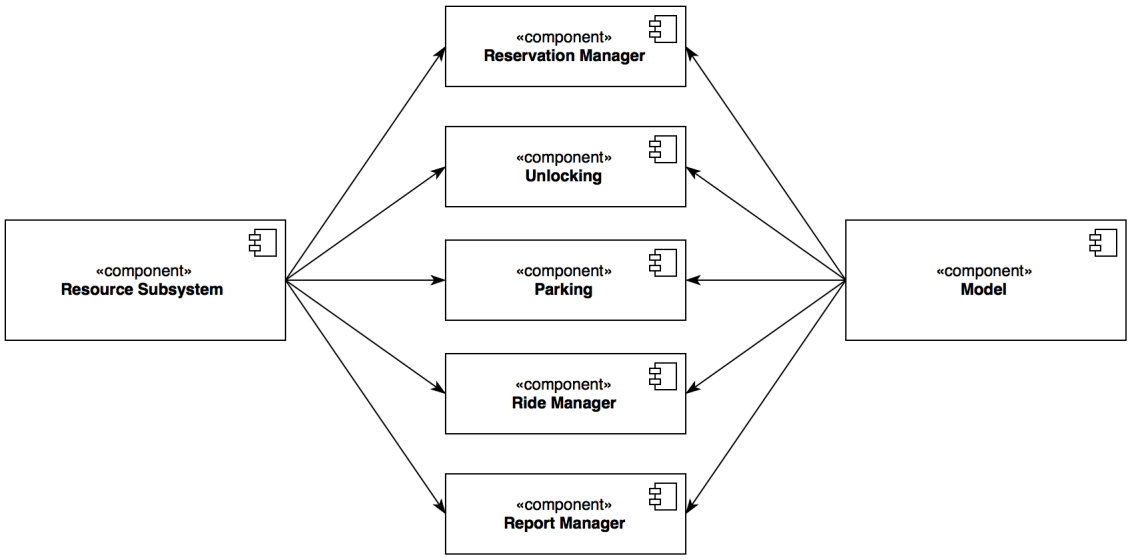
Application and Resource Subsystems Integration We can now proceed to integrating the Application Subsystem and the Resource Subsystem, again following the bottom-up approach.



Application Subsystem Components Most of the components in the Application Subsystem are independent from each other, therefore they can be freely tested in parallel.

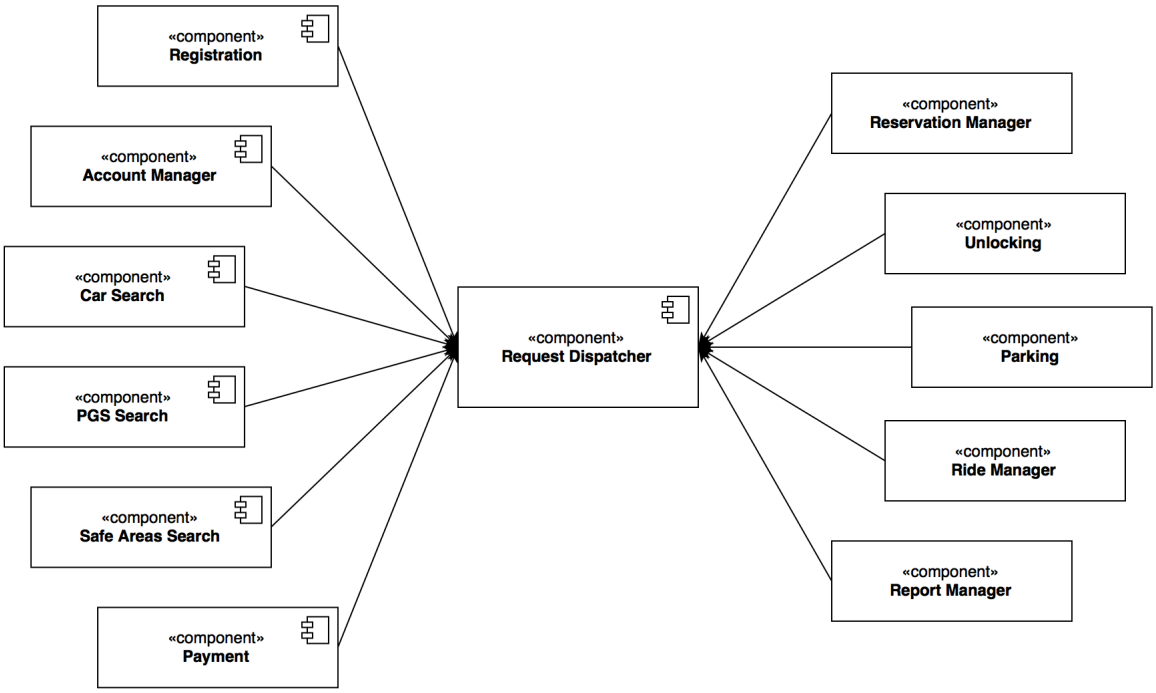
The Registration Component, the Account Manager, the Car Search Component, the PGS Search Component, the Safe Areas Search Component and the Payment Component only need the Model, while the Reservation Manager, the Unlocking Component, the Parking Component, the Ride Manager and the Report Manager also need to interface with the Resource Subsystem.



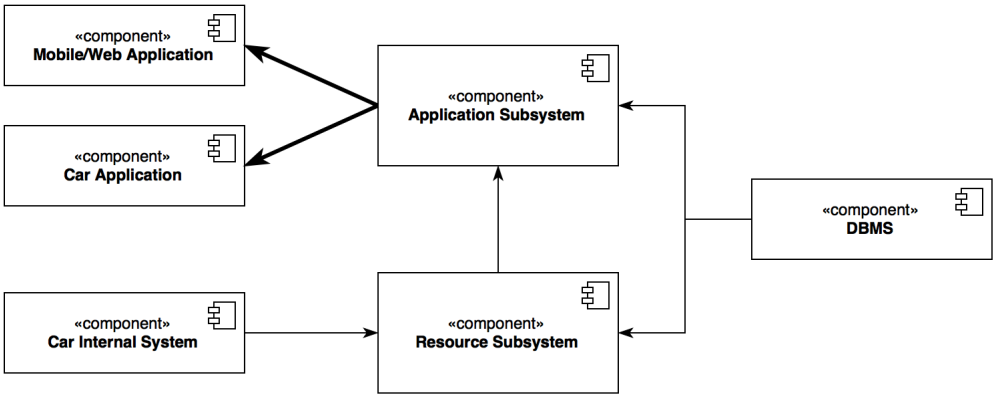


Request Dispatcher The Request Dispatcher acts as a wrapper for all of the requests to the application API, processing and distributing them to the corresponding components.

This allows us to integrate it with thread based testing: when testing for one of the intermediate components has been finished, we can test the corresponding functionality on the Request Dispatcher, that will interact with the specific component. The request dispatcher will be integrated one by one with all the other components, until the Application Subsystem is complete.



Client Applications Integration Finally, after the whole central system is tested and has been successfully integrated, we will test the interaction between the client applications and the system.



3 Individual Steps and Test Description

In this section we will describe the tests which have to be taken in each step of our integration plan. After integrating a component with another we will perform tests on their functionalities to see if they work according to the specifications.

We will outline the type of inputs and the conditions used for each test, and the corresponding expected result.

In addition to the guideline specified in this section the tests should also use various combinations of null values and parameters of the wrong type, to check if the implementation accounts for unforeseen errors and problems in the code, following the basic principles of defensive programming.

3.1 Model - DBMS

This integration step couples the Model to the DBMS. We describe the tests for the main functions that the Model will offer to the other components.

Insert User	
Input	*Result*
A not already existing username, a not already used email, the user's personal data, a not already used license id number and a payment method (e.g. a credit card number).	A new user is inserted in the database.
Like the first case but with a license id number already present in the database	An exception is raised.
Like the first case but with an email already present in the database	An exception is raised.
Like the first case but with a username already present in the database	An exception is raised.

Update User	
Input	*Result*
An existing username and the user's data to update (not an email or driving license belonging to another user)	The user's data in the database is updated.
Like the first case but with a license id number already belonging to another user	An exception is raised.
Like the first case but with an email address already belonging to another user	An exception is raised.
A non-existent username	An exception is raised.

Insert Bill	
Input	*Result*
A valid user and a well-formed bill.	The bill is added to the user's bills in the database.
A valid user and an ill-formed bill	An exception is raised.
A non-existent user and a bill	An exception is raised.

List Bills	
Input	*Result*
A valid user	Returns the user's bills.
A valid user with no bills	Returns an empty set.
A non-existent user	An exception is raised.

List Cars	
Input	*Result*
Nothing	Returns a set of all cars, inclusive of information about their position, battery level and availability.

List Power Grid Stations	
Input	*Result*
Nothing	Returns a set of all power grid stations, inclusive of information about their position and plugs availability.

List Safe Areas	
Input	*Result*
Nothing	Returns a set of all safe areas along with their position.

Insert Reservation	
Input	*Result*
A valid user, an id for an available car	A new reservation is created in the database.
A valid user, an id for a car that is not available	An exception is raised.
A valid user, a non-existent car id	An exception is raised.
A non-existent user, a car id	An exception is raised.

Update Reservation	
Input	*Result*
An existent ongoing reservation, valid data to update the reservation	The reservation is updated with the new data.
An existent ongoing reservation, invalid data to update the reservation	An exception is raised.
A canceled reservation, data to be updated	An exception is raised.
An ended reservation, data to be updated	An exception is raised.
A non-existent reservation, data to be updated	An exception is raised.

Insert Ride	
Input	*Result*
A valid user, an id for a car currently reserved by the user	A new ride is created in the database.
A valid user, an id for a car not currently reserved by the user	An exception is raised.
A valid user, a non-existent car id	An exception is raised.
A non-existent user, a car id	An exception is raised.
A suspended user, a car id	An exception is raised.

Update Ride	
Input	*Result*
A valid ride and valid data to update	The ride is updated with the new data.
A valid ride, invalid data to update	An exception is raised.
A non-existent ride, data to update	An exception is raised.

3.2 PGS Manager - DBMS

Update Plug Availability	
Input	*Result*
A station's valid authentication credentials, a valid plug id and its availability	The plug's availability is updated in the database.
A station's valid authentication credentials, an invalid plug id and an availability	An exception is raised.
Invalid authentication credentials, a plug id and an availability	An exception is raised.

3.3 Car Handler - Car Internal System

Request Update from Car	
Input	*Result*
A valid car id and a valid update request type	The request is sent to the car and the car sends an appropriate update to the Car Manager.
A valid car id and an invalid update request type	An exception is raised.
A non-existent car id and an update request type	An exception is raised.

Lock Car	
Input	*Result*
A valid car id	The car is locked.
A non-existent car id	An exception is raised.

Unlock Car	
Input	*Result*
A valid car id	The car is unlocked.
A non-existent car id	An exception is raised.

3.4 Car Manager - Car Handler - DBMS

3.4.1 Maintenance

List Cars in Need of Maintenance	
Input	*Result*
Valid authentication credentials	Returns a set with all cars in need of maintenance, along with their issues, position and battery level.
Invalid authentication credentials	An exception is raised (for bad authentication).

Set Car as Available	
Input	*Result*
Valid authentication credentials, a valid car id	The car is set as available.
Valid authentication credentials, a non-existent car id	An exception is raised.
Invalid authentication credentials, a car id	An exception is raised (for bad authentication).

Set Car as Unavailable	
Input	*Result*
Valid authentication credentials, a valid car id and a reason for making it unavailable	The car is set as unavailable and the reason is saved on the database.
Valid authentication credentials, a valid car id and no reason to for making it unavailable	An exception is raised.
Valid authentication credentials, a non-existent car id	An exception is raised.
Invalid authentication credentials, a car id	An exception is raised (bad authentication).

3.4.2 Interface to the Application Subsystem

Request to Lock Car	
Input	*Result*
A valid car id	The car is locked.
A non-existent car id	An exception is raised.

Request to Unlock Car	
Input	*Result*
A valid car id	The car is unlocked.
A non-existent car id	An exception is raised.

3.4.3 Car Interface

Car Status Update	
Input	*Result*
A car's valid authentication credentials, well-formed info about the car's status	The car's status is updated in the database.
A car's valid authentication credentials, ill-formed info about the car's status	An exception is raised.
Invalid car authentication credentials and info about a car's status	An exception is raised.

3.5 Application Subsystem Components - Model

3.5.1 Registration Component - Model

Register User	
Input	*Result*
A not-yet-taken username, an email, valid personal data, a valid payment method and a valid driving license id number	A new user is created in the database and an email containing the user's password is sent to the email address provided.
A not-yet-taken username, an email, valid personal data, a valid payment method and a picture of a valid driving license	A new user is created in the database and an email containing the user's password is sent to the email address provided.
A not-yet-taken username, an email, valid personal data, a valid payment method and an invalid driving license	An exception is raised.
A not-yet-taken username, an email, valid personal data, an invalid payment method and a driving license	An exception is raised.
A not-yet-taken username, an email, invalid personal data, a payment method and a driving license	An exception is raised.
An already taken username, an email, personal data, a payment method and a driving license	An exception is raised.
An ill-formed username, an email, personal data, a payment method and a driving license	An exception is raised.

3.5.2 Account Manager - Model

Login	
Input	*Result*
A valid username and the correct corresponding password	Creates a session for the user and returns a session token.
A valid username and a wrong password	An exception is raised.
A non-existent username and a password	An exception is raised.

Get User Info	
Input	*Result*
A valid session token	Returns all the session user's info
An invalid session token	An exception is raised (bad authentication).

Update User Info	
Input	*Result*
A valid session token and valid new data to update	The user info is updated with new data on the database.
A valid session token and invalid new data	An exception is raised.
An invalid session token and new data	An exception is raised (bad authentication).

List User Bills	
Input	*Result*
A valid session token	Returns a set with all the user's bills
An invalid session token	An exception is raised (bad authentication).

3.5.3 Car Search Component- Model

Find Cars with Position	
Input	*Result*
A valid geographic position	Returns a list of all the cars close to that position, along with their associated information.
An invalid geographic position	An exception is raised.

Find Cars with Address	
Input	*Result*
A valid well-formed address	Returns a list of all the cars close to that address, along with their associated information.
An invalid address	An exception is raised.

3.5.4 PGS Search Component- Model

Find Stations with Position	
Input	*Result*
A valid geographic position	Returns a list of all the charging stations close to that position, along with their associated information.
An invalid geographic position	An exception is raised.

Find Stations with Address	
Input	*Result*
A valid well-formed address	Returns a list of all the charging stations close to that address, along with their associated information.
An invalid address	An exception is raised.

3.5.5 Safe Areas Search Component - Model

Find Safe Areas with Position	
Input	*Result*
A valid geographic position	Returns a list of all the safe areas close to that position.
An invalid geographic position	An exception is raised.

Find Safe Areas with Address	
Input	*Result*
A valid well-formed address	Returns a list of all the safe areas close to that address.
An invalid address	An exception is raised.

3.5.6 Payment Component- Model

Pay a Bill	
Input	*Result*
A valid session token, a bill that needs to be paid and a valid payment method	The transaction is carried out; if it succeeds the bill is marked as paid, otherwise returns failure.
A valid session token, a bill that needs to be paid and an ill-formed payment method	An exception is raised.
A valid session token and a bill that needs to be paid	The system uses the payment method saved for the user to carry out the transaction; if it succeeds the bill is marked as paid, otherwise returns failure.
A valid session token and a bill that has already been paid	An exception is raised.
A valid session token and a non-existent bill	An exception is raised.
An invalid session token and a bill	An exception is raised (bad authentication).

3.5.7 Reservation Manager - Model - Resource Subsystem

Create Reservation	
Input	*Result*
A valid session token for a not suspended user that has no current reservation, an available car	A new reservation is created.
A valid session token for a not suspended user that has no current reservation, a car that is not available	An exception is raised.
A valid session token for a not suspended user that has no current reservation, a non-existent car	An exception is raised.
A valid session token for a not suspended user with an ongoing reservation, a car	An exception is raised.
A valid session token for a suspended user and a car	An exception is raised.
An invalid session token and a car	An exception is raised (bad authentication).

Cancel Reservation	
Input	*Result*
A valid session token and an ongoing reservation made by the user	The reservation is canceled.
A valid session token and an ongoing reservation made by another user	An exception is raised.
A valid session token and a reservation that has already ended	An exception is raised.
An invalid session token and a reservation	An exception is raised (bad authentication).

3.5.8 Unlocking Component - Model - Resource Subsystem

Unlock Car with Code	
Input	*Result*
A valid session token, the code of the car that is currently being reserved by the user	The car is unlocked.
A valid session token, the code of a car that is not currently being reserved by the user	An exception is raised.
A valid session token, an invalid car code	An exception is raised.
An invalid session token	An exception is raised (bad authentication).

Unlock Car with Position	
Input	*Result*
A valid session token for a user that has an ongoing reservation, a geographic position close to the car that the user has reserved	The car is unlocked.
A valid session token for a user that has an ongoing reservation, a geographic position far from the car that the user has reserved	An exception is raised.
A valid session token for a user that has no ongoing reservation, a geographic position	An exception is raised.
An invalid session token	An exception is raised (bad authentication).

3.5.9 Parking Component - Model - Resource Subsystem

Park Car	
Input	*Result*
A valid session token for a user that has an ongoing ride	If the car has been stopped and the car's position is in a safe area, then the car is locked, otherwise returns failure.
A valid session token for a user that has no ongoing ride	An exception is raised.
An invalid session token	An exception is raised (bad authentication).

3.5.10 Ride Manager - Model - Resource Subsystem

End Ride	
Input	*Result*
A valid session token for a user that has an ongoing ride	If the car has been stopped and the car's position is in a safe area, then the car is locked and the ride is marked as ended, otherwise returns failure.
A valid session token for a user that has no ongoing ride	An exception is raised.
An invalid session token	An exception is raised (bad authentication).

3.5.11 Report Manager - Model - Resource Subsystem

Report Issue	
Input	*Result*
A valid session token for a user with an ongoing reservation or ride, an explanation of the issue	The issue is recorded and the car is set as unavailable as the user leaves the car.
A valid session token for a user with no ongoing reservation or ride	An exception is raised.
An invalid session token	An exception is raised.

3.6 Request Dispatcher - Application Components

The Request Dispatcher acts as a middle man between the external Application Subsystem API and the single components in the Application Subsystem, and therefore apart from some preprocessing the functionalities handled by the Request Dispatcher are basically the same we described in section 3.5.

To test this integration step we should just make sure that the external Application Subsystem API works as intended. To do this we can follow the API specification that can be found on section 2.6.3 (Protocols) of the Design Document.

Appropriate tests must be made for each one of the API endpoints, with valid and invalid inputs (mostly following the same patterns as section 3.5) to test both positive and negative results.

3.7 Mobile/Web Application - Application Subsystem

The mobile/web client application will be tested manually to see if all the available functionalities work as expected after the system has been completely integrated.

3.8 Car Application - Application Subsystem

Like the mobile application, the car application will also be tested manually to check if all functionalities work correctly once the system has been integrated.

4 Tools and Test Equipment Required

4.1 Tools

In order to test the software in the most efficient way, we will use some tools for automated testing.

Mockito Mockito is an open source testing framework for Java released under the MIT License. It allows the *creation of mock objects* in automated unit tests. It supports *scaffolding and stubs*.
`http://site.mockito.org/`

JUnit JUnit is a simple, open source framework used *to write and run repeatable tests*. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include: assertions for testing expected results, test fixtures for sharing common test data and test runners for running tests.
`http://junit.org/junit4/`

Arquillian Arquillian is a test framework that can be used to perform testing inside a remote or embedded container, or deploy an archive to a container so the test can interact as a remote client; in addition Arquillian integrates with other testing frameworks (e.g., JUnit 4), allowing the use of IDE and Maven test plugins. It will be used mainly for *testing dependency injections* and *transaction control*.
`http://arquillian.org/`

In addition to automatic testing we will also make use of manual testing, mainly for testing the client applications.

4.2 Test Equipment Required

In order to make sure that our software works properly in the majority of cases, we will need some equipment to test our client applications on real devices.

To test the mobile/web application we will need:

- at least an Android smartphone or tablet,
- at least an Apple smartphone or tablet,
- at least a Windows Phone smartphone or tablet.

Since we are developing a responsive web application, it is enough to check the software behavior on different mobile OS and browsers. To do so we will also test the application on a personal computer, using different browsers (Chrome, Explorer, Safari, Firefox).

To test the car application we need to have access to at least one car per model that PowerEnJoy will offer to the customers.

The car(s) must be fully equipped with a working Internal System and all the sensors required for the system to function.

5 Program Stubs and Test Data Required

5.1 Program Stubs and Drivers

Due to our choice to follow the bottom-up approach for the testing and the integration, we are going to make use mostly of drivers, with the exception of one stub that will act as an interface in place of the central system.
These are the artifacts that we will be developed, ordered according the software integration sequence:

- **Model Driver**: this driver replaces the **Model** component and tests the access to the **DBMS**.
- **PGS Manager Driver**: this driver replaces the **PGS Manager** component and tests the access to the **DBMS**.
- **Car Handler Driver**: this driver replaces the **Car Handler** component and tests the interaction with the **Car Internal System** component. In this case we will take advantage of a stub to perform the behavior of the **Car Manager** component.
- **Car Manager Driver**: this driver replaces the **Car Manager** component and tests the interaction with the **DBMS**, the **Car Handler** component, **Car Internal System** component and with the **PGS Manager** component, for completeness reasons.
- **Registration Driver**, **Account Manager Driver**, **Car Search Driver**, **PGS Search Driver**, **Safe Areas Search Driver**, **Payment Driver**: these drivers replace a subpart of the **Application Subsystem**. They are independent from each other, so we can simultaneously test their interaction with the **Model**.
- **Report Manager Driver**, **Reservation Manager Driver**, **Unlocking Driver**, **Parking Driver**, **Ride Manager Driver**: these drivers replace a subpart of the **Application Subsystem**. They are independent from each other, so we can simultaneously test their interaction with the **Model** and with the **Resource Subsystem**.
- **Request Dispatcher Driver**: this driver replaces the **Request Dispatcher** component and tests the interaction with the **Application Subsystem** components.

5.2 Test Data

For the sake of completeness, the data used in the tests allows us to verify the proper behavior and the result of any kind of incorrect practice. Moreover the DB will be populated with appropriate data to be ready for the tests.

In order to perform the necessary tests, we are going to need:

- A list of both valid and invalid sign up data to test the **Registration** component. The data must have all combinations of the following:
  - Correct personal data
  - Personal data with numeric characters and notation
  - DOBs compliant with the ISDTN UNI EN 28601
  - DOBs not compliant with the ISDTN UNI EN 28601
  - Non-existent email addresses
  - Valid email addresses
  - Valid driving licenses
  - Suspended driving licenses
  - Non-existent driving licenses
  - Driving licenses not compliant with legal format
  - Valid credit card numbers
  - Credit card numbers not compliant with legal format
  - Valid credit card CVVs
  - Invalid credit card CVVs
- A list of both valid and invalid log in data to test the **Account Manager** component. The data must have all combinations of the following:
  - Valid usernames
  - Non-existent usernames
  - Valid password
  - Invalid passwords
- A list of both valid and invalid tracing data to test the **Car Search** component, the **PGS Search** component and the **Safe Areas Search** component. The data must have all combinations of the following:
  - Valid positions
  - Non-existent positions
  - Positions outside safe areas
  - Valid addresses

  - Invalid addresses
  - Non-existent addresses
  - Addresses outside safe areas

- A list of both valid and invalid banking data to test the **Payment** component. The data must have all combinations of the following:
  - Valid credit card numbers
  - Credit card numbers not compliant with legal format
  - Valid credit card CVVs
  - Invalid credit card CVVs

- A list of both valid and invalid user data to test the **Reservation Management** component. The data must have all combinations of the following:
  - Valid usernames
  - Non-existent usernames
  - Valid passwords
  - Invalid passwords
  - List of cars on DB

- A list of both valid and invalid user and tracing data to test the **Unlock** component. The data must have all combinations of the following:
  - Valid usernames
  - Non-existent usernames
  - Valid passwords
  - Invalid passwords
  - Valid positions
  - Non-existent positions
  - Positions outside safe areas
  - Valid QR Codes
  - Invalid QR Codes
  - List of cars on DB

- A list of both valid and invalid user and tracing data to test the **Ride** component. The data must have all combinations of the following:
  - Valid usernames
  - Non-existent usernames
  - Valid passwords

  - Invalid passwords
  - Valid positions
  - Non-existent positions
  - Position outside safe areas

- A list of both valid and invalid user data to test the **Report Manager** component. The data must have all combinations of the following:
  - Valid usernames
  - Non-existing usernames
  - Valid passwords
  - Invalid passwords
  - Valid input fields for the form
  - Null input field for the form
  - List of cars on DB

6 Effort Spent

- Pietro Ferretti: 20 hours of work
- Nicole Gervasoni: 10 hours of work
- Danilo Labanca: 6 hours of work