POLITECNICO DI MILANO

SOFTWARE ENGINEERING 2 PROJECT

A.Y. 2016-17

PowerEnJoy

Integration Test Plan Document

Enrico Migliorini, Alessandro Paglialonga, Simone Perriello

15/01/2017

Sommario

Chapter 1

Introduction

1.1 Revision History

Version	Date	Authors	Summary
1.0	15/01/17	Alessandro Paglialonga, Enrico Migliorini, Simone Perriello	Initial Release

1.2 Purpose and Scope-

This Document represents the Integration Test Plan Document for the *PowerEnJoy* project.

This kind of testing is performed to find defects which are generally related to inter-process communication or parameter and data inputs. Two units may have passed unit testing and work well individually, but fail to communicate vital information with one another. By testing the units in aggregate, it's easier to identify the source of the issues found.

The purpose of this document is to outline, in a clear and comprehensive way, the main aspects concerning the organization of the integration testing activity for all the components that make the system up.

The aim of the *PowerEnJoy* project is to provide a *Car-Sharing Service* which implements electric-powered cars only. This system will have to interface the Cars, Charging Areas, allowing Users to reserve, unlock, drive and park Cars, finally they will be charged the cost of the ride.

In the following sections will be provided :

- A list of the subsystems and their subcomponents involved in the integration activity that will have to be tested.
- The entry criteria that must be met before the integration testing of the specified elements may begin.
- The reasoning process which led to the decision of the integration testing approach and the description of the latter. This includes the order of sequence in which components will be integrated.
- A list of all the tools employed during the testing activities.

1.3 Definitions, Acronyms and Abbreviations

DBMS = Database Management System

DB = Database

DD = Design Document

RASD = Requirements Analysis and Specification Document

SSN = Security Social Number

1.4 Reference Documents

- Project Assignment Document : Assignments AA 2016-2017.pdf
- PowerEnJoy Requirements Analysis and Specification Document : RASD.pdf
- PowerEnJoy Design Document : DD.pdf
- Integration testing example document.pdf
- Integration Test Plan Example.pdf

Chapter 2

Integration Strategy

2.1 Entry Criteria

Before this document is written, and for the steps written here to be meaningful, the **Requirements Analysis and Specification Document** and the **Design Document** of the *PowerEnJoy* project must have been fully written.

The second required step before the integration test is performed is the Unit Testing of each component.

Unit Testing verifies the correct functioning of each module and its methods according to their specification; it is used to test a specific unit (class) in the application testing all the methods in the class including all exception paths in the methods. Unit Tests only cover the testing of each module in the application, hence we can't test the functional requirements of the application or scenarios.

The communication testing between **Services** has to be done in the Unit Testing while the testing between **Services** and **External Service Interfaces** is provided in this phase.

We'll assume that Firewall has been properly configured and its functionaly have been tested in Unit Testing.

Unit testing of the DBMS, Mapping System, Banking System has already been done by their respective software producers.

As the exclusive job of the **External Service Interfaces** is to communicate with External Systems and so there's no real computation, we thought it would be more considerable to test the communication between these two in the Unit Testing.

2.2 Elements to be Integrated

In this paragraph we report the list of the components to be integrated.

The Framework Spring used for the implementation of this project, which has already been fully descripted in the DD, provides a set of native methods belonging to the Repositories that may be called and used by the Services. Obviously the use of these methods between Services and Repositories has no need to be tested as the framework natively implements it. As a result, the only Repositories methods which we're going to test in the Integration Testing are the ones implemented exclusively for the *PowerEnJoy* project.

Those below are the methods natively implemented by Spring CRUDRepository Interface.

Modifier and Type	Method and Description
`long`	`count()` Returns the number of entities available.
`void`	`delete(ID id)` Deletes the entity with the given id.
`void`	`delete(Iterable<? extends T> entities)` Deletes the given entities.
`void`	`delete(T entity)` Deletes a given entity.
`void`	`deleteAll()` Deletes all entities managed by the repository.
`boolean`	`exists(ID id)` Returns whether an entity with the given id exists.
`Iterable<T>`	`findAll()` Returns all instances of the type.
`Iterable<T>`	`findAll(Iterable<ID> ids)` Returns all instances of the type with the given IDs.
`T`	`findOne(ID id)` Retrieves an entity by its id.
`<S extends T>` `Iterable<S>`	`save(Iterable<S> entities)` Saves all given entities.
`<S extends T>` `S`	`save(S entity)` Saves a given entity.

Image from :

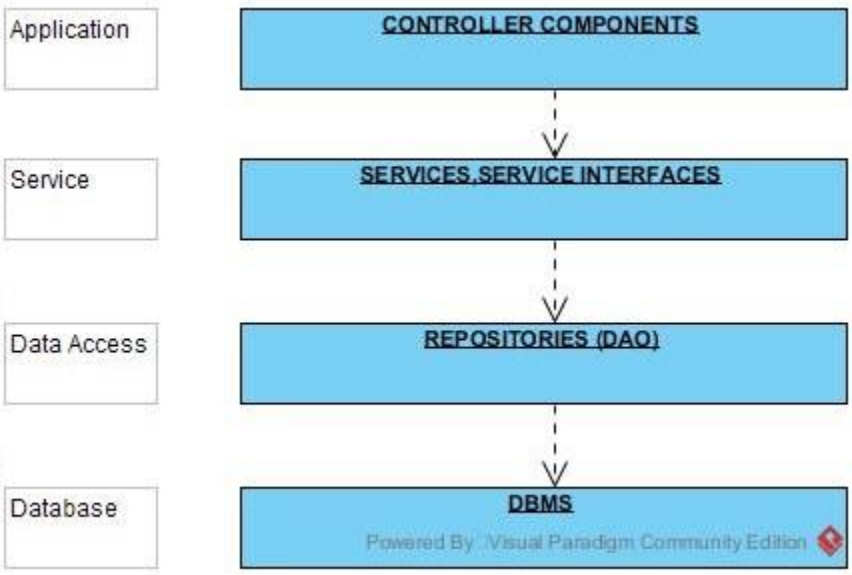
http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html?is-external=true

Starting from the bottom layer of the architecture to go along with our Integration Testing Strategy (see paragraph 2.3) we'll integrate Server-Side : **DBMS** with the **Repositories, Repositories** with **Services, Services** with **Service Interfaces, Services** with **Controllers, Controllers** and the different types of **Clients.**

We don't need to configure the communication between the **Controllers** and the **Dispatcher** as the Spring framework used for the implementation of the *PowerEnJoy* project, by using annotations on Controllers, provides an automatic mapping between an URL and a specific Controller. In this way the Spring internal Dispatcher invokes for each URL a specific class and so the only check that has to be done is if the correct class is invoked for each URL, which is done in the Unit Testing.

Below is the *PowerEnJoy* layered architecture diagram Server-Side which best shows the granularity of components to test.

PowerEnJoy Application Server Layered Architecture Diagram



2.3 Integration Testing Strategy

During the integration testing strategy selection several popular choices were evaluated by the team. Different strategies exist for the Integration Testing which all have benefits and drawbacks depending on the project to implement, Top Down, Sandwich, Bottom Up, Critical modules first and Big Bang; there's no strategy which is

clearly better than the others as each approach is perfectly suited for a specific type of architecture of components and communication between components. Consequently, the team has evaluated the different methodologies looking at the big picture of this project.

The choice which seemed to best fulfill our requirements was the Bottom Up approach. During the evaluation some of the advantages which led our choice were :

- The absence of urgency for any kind of program stubs, which is needed in top-down approach instead, as we always start developing and testing with absolute modules
- Starting from the bottom of the hierarchy means that the critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are find out early in the process.
- Fault localization is easier and no time is wasted waiting for all modules to be developed unlike Big-Bang approach.

During the evaluation some of the drawbacks which we accepted existing for our project were :

- We always have to form the test drivers which are the simulated environment in which the component being tested is to integrated. In addition, drivers have to be tested themselves and so more time and efforts are needed to accomplish this complicated task.
- The application as an entity does not exist until the last module is added.

2.4 Software Integration Sequence

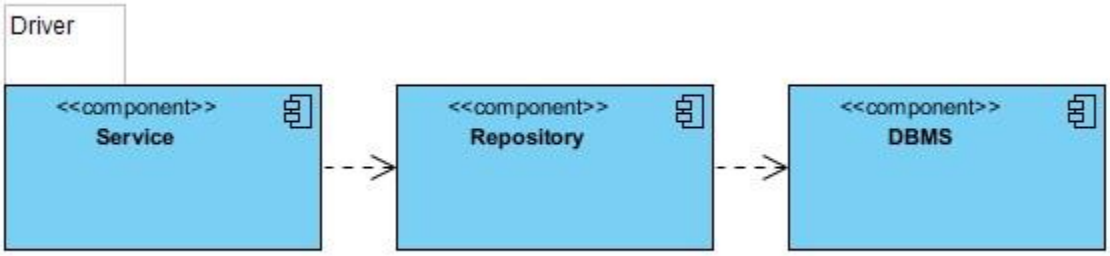
In this paragraph we describe the order in which the components are integrated and tested. The arrow arrives in the component which needs to be implemented first.

The first components to be integrated and tested are the DB and the Repositories. There's no need of drivers as the repositories are the first components to be implemented in this project and since the DB has already been implemented by its producer which is external to the team developing the *PowerEnJoy* project.

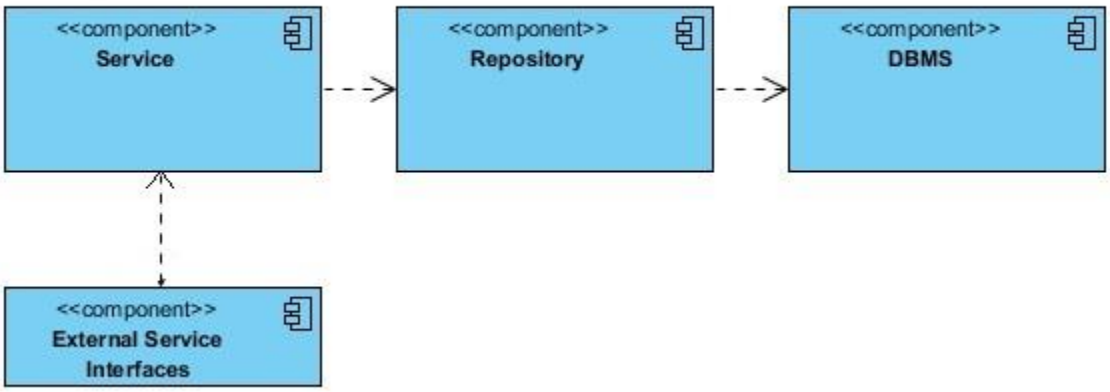
There's no actual real communication to test between the DBMS and the Repositories as the only methods called and used by the repositories are the queries made available by the DBMS according to its implementation.

For the sake of completion and to show every module from the very bottom of the project we're still putting this "fake integration" in this part. We assume that the configuration file of the communication between the DB and the Application server has already been verified in the Unit Testing.

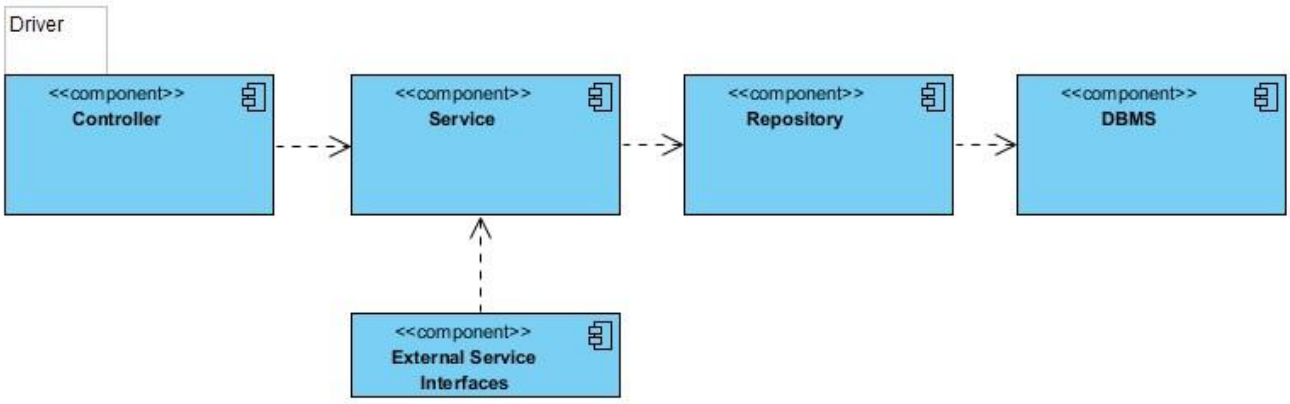
We can now test the communication and integration between Repositories and Services which have access to them. Repositories are the first components to be implemented while the Services are not implemented yet, this is why we use drivers for Services.



Now we'll be able to test the integration between Services and External Service Interfaces. The latter are the components that allow the System to connect to External Systems such as the Banking and the Mailing ones. The implementation of External Service Interfaces can be done in parallel with the implementation of the Services, hence we can test the communication between the very components without using any driver.



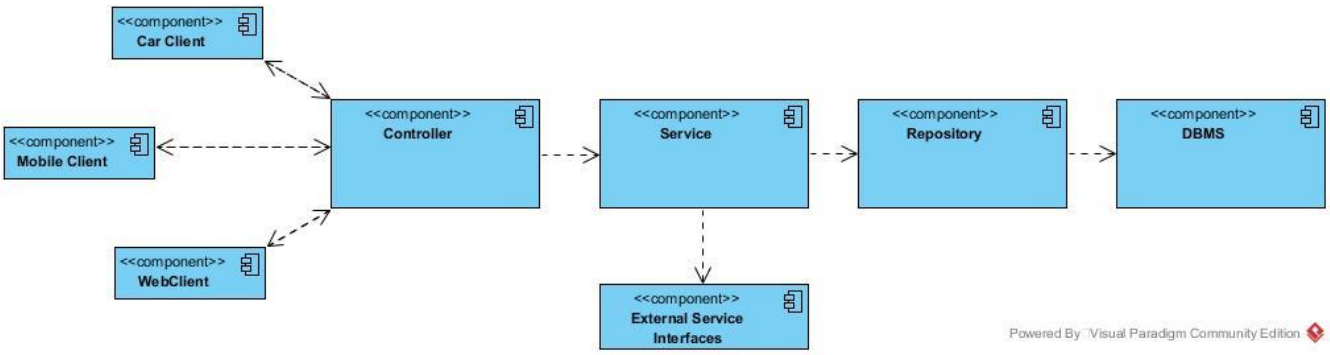
Now including the Controllers we can test the all Business Logic. Controllers will be substituted by their drivers.



Finally we integrate the System Server-Side with its Client-Side. So we integrate the Controllers with the different Clients: Mobile Client, WebClient and CarClient. We assume their development is already done and hence we don't need any driver.

Web Clients only have access to Register and Confirm Registration methods via HTPP requests and their task is to simply receive a jsp/html page as response from Controllers. Hence, as there's no logic client-side we can think of including the testing of WebClients with Controllers in the Unit Testing.

We can choose any order for picking the client to test first.



Chapter 3

Individual Steps and Test Description

3.1 Test Case Specification

In this chapter we're going to provide a detailed description of the tests to be performed on each pair of components that have to be integrated. Each subsection contains the name of the involved components, the first one is the caller component and the second one is the called component. For each method will be provided a different type of input and the corresponding expected effects on the system.

3.1.1 Services, Repositories

We start by testing the methods called by the Services belonging to the Repositories. Remember that native Repositories Spring methods have no need to be tested and consequently aren't listed in the below test cases.

3.1.1.1 Registration Service, User Repository

There are no non-native Spring methods of the User Repository called by the Registration Service because, as already described in the DD, the checks on the unicity

of SSN, Driving License Number and Email values for the insertion of an account is performed at Database level and so done by the DBMS.

The same reasoning is applied for similar contexts in all the other test cases.

3.1.1.2 Login Service, User Repository

FindByEmailAndPassword(inout email: string,inout password: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the User's data and will perform a query to check if exists a tuple with the corresponding data in the Users table in the Database. The Repository will then forward the resulting User POJO to the Service if there exists one.

3.1.1.3 Mapping Service, Area Repository

FindByCity(inout city: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the City data and will perform a query to the DB retrieve all the Areas belonging to the inserted city. The Repository will then forward the list of the resulting Area POJO to the Service if there exists one.

FindByGpsLatitudeGreaterThanAndGpsLatitudeLessThanAndGpsLongitudeGreaterThanAndGpsLongitudeLessThanAndChargingSlotsGreaterThan(inout minLatitude: float, inout maxLatitude: float, inout minLongitude: float,inout maxLongitude: float, inout minSlots: integer)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.

Valid arguments	The Repository receives the minimum and maximum GPS coordinates which represent the vertices of a square and will perform a query to the DB to retrieve all the Areas whose GPS coordinates are inside the given built square. The Repository will then forward the list of the resulting Area POJO to the Service if there exists one.

3.1.1.4 Mapping Service, Car Repository

FindByStatusAndGpsLatitudeGreaterThanAndGpsLatitudeLessThanAndGpsLongitudeGreaterThanAndGpsLongitudeLessThan(inout currentStatus: CarStatus, inout minLatitude: float, inout maxLatitude: float, inout minLongitude: float,inout maxLongitude: float, inout minSlots: integer)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the minimum and maximum GPS coordinates which represent the vertices of a square and will perform a query to the DB to retrieve all the Cars whose GPS coordinates are inside the given built square whose status is the CarStatus received by this method (will always be “Available”. The repository will then forward the list of the resulting Car POJO to the Service if there exists one.

3.1.1.5 Banking Service, Banking Repository

There are no non-native Spring methods of the Banking Repository called by the Banking Service. The unique method called by the Banking Service in this case is the native Spring method “save”.

3.1.1.6 ManageReservation Service, Driving Repository

FindByDrivingUserAndIsActiveTrue(inout user: UserPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.

Valid arguments	The Repository receives the User data and will perform a query to the DB to check if there exists a tuple in the Driving Table, whose attribute IsActive is true, associated to that User. The Repository will then forward the resulting Driving POJO if there exists one.

FindByDrivenCarAndIsActiveTrue(inout car: CarPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the Car data and will perform a query to the DB to check if there exists a tuple in the Driving Table, whose attribute IsActive is true, associated to that Car. The Repository will then forward the resulting Driving POJO if there exists one.

3.1.1.7 ManageReservation Service, Reservation Repository

FindByReservingUserAndReservedCarAndIsActiveTrue(inout user: UserPOJO,inout car: CarPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the Car data and the User data and will perform a query to the DB to check if there exists a tuple in the Reservation Table, whose attribute IsActive is true, associated to that Car and that User.The Repository will then forward the resulting Reservation POJO if there exists one.

FindByReservingUserAndIsActiveTrue(inout user: UserPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.

Valid arguments	The Repository receives the User data and will perform a query to the DB to check if there exists a tuple in the Reservation Table, whose attribute IsActive is true, associated to that User. The Repository will then forward the resulting Reservation POJO if there exists one.

FindByReservedCarAndIsActiveTrue(inout user: CarPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the Car data and will perform a query to the DB to check if there exists a tuple in the Reservation Table, whose attribute IsActive is true, associated to that Car. The Repository will then forward the resulting Reservation POJO if there exists one.

3.1.1.8 ManageDriving Service, Reservation Repository

FindByDrivingUserAndDrivenCarAndIsActiveTrue(inout user: UserPOJO,inout car: CarPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the Car data and the User data and will perform a query to the DB to check if there exists a tuple in the Driving Table, whose attribute IsActive is true, associated to that Car and that User. The Repository will then forward the resulting Driving POJO if there exists one.

FindByDrivingUserAndIsActiveTrue(inout user: UserPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.

Valid arguments	The Repository receives the User data and will perform a query to the DB to check if there exists a tuple in the Driving Table, whose attribute IsActive is true, associated to that User. The Repository will then forward the resulting Driving POJO if there exists one.

FindByDrivingCarAndIsActiveTrue(inout user: CarPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the Car data and will perform a query to the DB to check if there exists a tuple in the Driving Table, whose attribute IsActive is true, associated to that Car. The Repository will then forward the resulting Driving POJO if there exists one.

3.1.1.9 ManageDriving Service, Banking Repository

There are no non-native Spring methods of the Banking Repository called by the Banking Service. The unique method called by the Banking Service in this case is the native Spring method “save”.

3.1.1.10 ManageDriving Service, Car Repository

FindByPlateNumber(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Repository receives the Car data and will perform a query to the DB retrieve the Car with the corresponding plateNumber. The Repository will then forward the CarPOJO if there exists one.

3.1.2Services, External Service Interfaces

We now show the test cases of communication between the External Interface Services and the Services.

3.1.2.1 Banking Service, Banking Interface Service

(inout ccNumber: string, inout amount: float)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Banking Interface Service receives the credit card number and the amount of money to charge to that credit card account. It will then communicate with the Banking System to actually ask for charging the money, the Banking Interface Service will then receive the confirmation or the denial of the success of the banking transaction and will forward it to the Banking Service.

3.1.2.2 Mailing Service, Mailing Interface Service

sendMailToAddress(inout to:string, inout subject: string, inout text: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Mailing Interface Service receives the data about the email to be sent to the User. It will then communicate with the Mailing System to actually ask for sending the email, the Mailing Interface Service will then receive the confirmation or the denial of the success of the mail sending and will forward it to the Mailing Service.

3.1.2.3 Manage Driving Service, Car Interface Service

lockDoors(inout plateNumber:string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.

Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the data about the Car to be locked. It will then communicate with the Car System to actually ask for locking the doors, the Mailing Interface Service will then receive the confirmation or the denial of the success of the action and will forward it to the Manage Driving– Service.

3.1.2.4 Manage Driving Service, Company Interface Service

sendDamageNotification(inout internalId: DamagePOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Company Interface Service receives the Damage POJO about the Car which received a damage. It will then communicate with the Company System the data and information about the damage and will receive by the Company System an acknowledgment. The Company will then dispatch an employee which will perform the actions already described in the DD.

sendLowBatteryNotification(inout internalID: CarPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Company Interface Service receives the data about the Car with low battery. It will then communicate with the Company System the data and information about the Car and will receive by the Company System an acknowledgement. The company will then dispatch an employee which will perform the actions already described in the DD.

sendOutParkingAreaNotification(inout internalID: CarPOJO)	
Input	*Effect*

A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the data about the Car which has been left out of a Safe Area. It will then communicate with the Company System the data and information about the Car and will receive by the Company System an acknowledgement. The Company will then take a decision about the User according to the internal laws.

3.1.2.5 Manage Area Service, Area Interface Service

This is a background task used to poll the Areas to retrieve their actual number of Charging slots

getAvailableChargingSlots(inout internalID: long)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Area Interface Service receives the id of the Area of which we want to know the available parking slots. The Area Interface Service will then communicate with the Area System to obtain the data about that specific charging Area. The data is then forwarded to the Service.

3.1.2.6 Manage Car Service, Car Interface Service

This is a background task used to poll the Cars to retrieve their actual Status.

getStatus(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the plate number of the Car of which we want to know the actual CarStatus. The Car Interface Service will then communicate with the Car System to

	obtain the data about that specific Car. The data is then forwarded to the Service.

This is a background task used to poll the Cars to retrieve their actual GpsPosition.

getPosition(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the plate number of the Car of which we want to know the actual position. The Car Interface Service will then communicate with the Car System to obtain the data about that specific Car. The data is then forwarded to the Service.

This is a background task used to poll the Cars to retrieve their actual status of the engine.

isEngineOn(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the plate number of the Car of which we want to know the actual state of the engine. The Car Interface Service will then communicate with the Car System to obtain the data about that specific Car. The data is then forwarded to the Service.

This is a background task used to poll the Cars to retrieve if they're plugged or not.

isPlugged(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the plate number of the Car of which we want to know if

	they're plugged. The Car Interface Service will then communicate with the Car System to obtain the data about that specific Car. The data is then forwarded to the Service.

This is a background task used to poll the Cars to retrieve if they're damaged.

getDamages(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the plate number of the Car of which we want to know the damages. The Car Interface Service will then communicate with the Car System to obtain the data about that specific Car. The data is then forwarded to the Service.

This is a background task used to poll the Cars to retrieve the actual number of passengers inside the Car.

getDamages(inout plateNumber: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The Car Interface Service receives the plate number of the Car of which we want to know the actual number of passengers. The Car Interface Service will then communicate with the Car System to obtain the data about that specific Car. The data is then forwarded to the Service.

3.1.3 Controllers, Services

We now show the test cases of communication between the Controllers and the Services.

3.1.3.1 Start Ride Controller, ManageDriving Service

activateRide(inout carPlateNumber: string)

Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments and Car status is “Reserved”	The Start Ride Controller communicates the plate number of the Car to the ManageDriving Service so that the latter can activate the ride and perform all the consequent actions, including the modification the corresponding tuples of the DB. The Service sends an acknowledgment to the Controller
Valid arguments but Car status is different from “Reserved”	An invalidOperationException is raised.

3.1.3.2 End Ride Controller, ManageDriving Service

endRide(inout carPlateNumber: string,inout carDatas: string[*])	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments and Car status is not “Reserved”	The End Ride Controller communicates the Car Data to the ManageDriving Service so the latter can perform all the actions needed when a ride ends like for example changing the Car Status. The Service sends an acknowledgment to the Controller.
Valid arguments but Car status is different from “In Use”	An invalidOperationException is raised.

isEndRideinSafeArea(inout carPlateNumber: string,inout position: GpsPosition)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments and Car status is not “Reserved”	The End Ride Controller communicates the Car Data and its position to the ManageDriving Service so the latter can notify the Company and perform all the consequent actions. The Service sends an acknowledgment to the Controller.
Valid arguments but Car status is different from “In Use”	An invalidOperationException is raised.

3.1.3.3 MajorDamage Controller, ManageDriving Service

MajorDamageDetectedRide(inout carPlateNumber: string, inout damageData: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The MajorDamage Controller communicates the Car Data and the Damage Data to the ManageDriving Service so the latter can notify the Company and perform all the consequent actions. The Service sends an acknowledgment to the Controller.

3.1.3.4 UnlockCars Controller, ManageReservation Service

unlockCar(inout reservationData: ReservationPOJO)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments, the User asking the request coincides with the one that is associated with that Reservation, and the Car attached in the request coincides with the one which is associated with that Reservation.	The UnlockCars Controller sends the ReservationPOJO to the ManageDriving Service so the latter calls the method “IsCarNearUser” of the Mapping Service and with a positive answer will ask the CarInterface Service to unlock the doors of the Car and perform all the consequent actions. The Service sends an acknowledgment to the Controller.
Valid arguments but the User asking the request doesn’t coincide with the one that is associated with that Reservation and/or the Car attached in the request coincides with the one which is associated with that Reservation	An invalidOperationException is raised.

3.1.3.5 ReserveCars Controller, ManageReservation Service

reserveCar(inout user: UserPOJO, inout car: CarPOJO)	
Input	*Effect*

A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments but the User has already another active Reservation or is actually driving another reserved Car	An InvalidOperationException is raised.
Valid arguments but the Car is already reserved or in use by another User	An invalidOperationException is raised.
Valid arguments, no active reservation or active drivings of the User and the Car.	The ReserveCars Controller sends the UserPOJO and the CarPOJO to the Service which will create the ReservationPOJO and forward it to the Controller. The Service calls its method "StartReservationTimerInBackground".

3.1.3.6 LocateNearbyCars Controller, Mapping Service

getCarsNearUser(inout userPosition: GpsPosition)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The LocateNearbyCars Controller sends the Gps Position of the User to the Mapping Service. The latter call its method "BuildGpsSquare" and will then call the methods of the Repositories to obtain the Cars which are inside that Square of Coordinates. The Service will forward to the Controller the list of CarPOJO (if any).

3.1.3.7 LocateAreas Controller, Mapping Service

getAreasInsideCity(inout city: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The LocateAreas Controller sends the city to the Mapping Service. The latter will then call the methods of the Repositories to obtain the Areas belonging to the given city. The Service will forward to the Controller the list of AreasPOJO (if any).

3.1.3.8 MoneySavingOption Controller, MoneySaving Service

evaluateBestChargingArea(inout endPosition: GpsPosition)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments	The MoneySavingOption Controller sends the Gps Coordinates of the position of the User's destination to the Mapping Service. The latter will then call the method "GetAreasNearPosition" of Mapping Service receiving a list of AreaPOJO. The Service will then perform the algorithm to calculate the best Area for the User and will forward to the Controller the chosen AreaPOJO.

3.1.3.9 Login Controller, Login Service

LogUserIn(inout email: string, inout password: string)	
Input	*Effect*
A null parameter	An invalidArgumentException is raised.
Invalid arguments	An invalidArgumentException is raised.
Valid arguments and email-password pair exists in the DB	The Login Controller sends the pair email-password of the User to the Login Service. The latter will then call the methods of the Repositories to check if exists the corresponding tuple from the DB. The Service, after receiving a positive answer, receives the UserPOJO and calls its own method "storeAuthenticationToken" to create an Authentication Token which is forwarded to the Controller (and will be forward to the Client).
Valid arguments but email-password pair doesn't exists in the DB	An invalidOperationException is raised.

3.1.4 Clients, Controllers

The Mobile Client, and Car Client (as well as WebClient) always communicate with Dispatcher first, but as we already described in paragraph 2.2 the Spring Internal Dispatcher just has the role to dispatch the Client request to the Controllers according

to the bindings, so it's meaningful to directly test the communication between Clients and Controllers.

3.1.4.1 Mobile Client, Authentication Controller

This method is actually performed every time the Mobile Client sends a request with a method reserved for logged Users. The dispatcher automatically dispatches the request to the Authentication Controller to check if the User is logged and then dispatches the request to the actual Controller needed to fulfill the original request.

"Send Token" : (*/secured/*,body(token=…))	
Input	*Effect*
Invalid arguments	A 401 Unauthorized HTTP Error Code is returned by the Controller.
Valid arguments	The Client is sending a correct login token. A 200 OK HTTP Code is returned by the controller. The request is forwarded to the Controller belonging to the original request.

3.1.4.2 Mobile Client, Login Controller

"Login" : PUT("/login", body("email=…,password=…")	
Input	*Effect*
Invalid arguments	An answer with the 401 Unauthorized HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. Attached to the answer there's the login token too.

3.1.4.3 Mobile Client, ReserveCar Controller

"Reserve Car" : PUT("/secured/car/reserve/{carid}",body("reservation=…,token…))	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.

Valid arguments	The Controller sends an answer with a 200 OK HTTP code. In the Body of the HTTP answer there's JSON data containing the ReservationPOJO.
Valid arguments but User has an another active reservation or is driving another reserved Car.	An answer with the 403 Forbidden HTTP error code is forwarded from the Controller to the client.
Valid arguments but the Car is reserved or in use by another User.	An answer with the 403 Forbidden HTTP error code is forwarded from the Controller to the client.

3.1.4.4 Mobile Client, LocateNearbyCars Controller

"Locate Cars" : GET("/secured/car/locatenearby/{gpsPosition}?token=…")	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. In the Body of the HTTP answer there's JSON data containing a list of the retrieved CarPOJO.

3.1.4.5 Mobile Client, MoneySavingOption Controller

"Money Saving Option" : GET("/secured/moneysaving/{gpsPosition}?token=…")	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. In the Body of the HTTP answer there's JSON data containing the AreaPOJO chosen by the Money Saving algorithm.

3.1.4.6 Mobile Client, LocateAreas Controller

"Locate Areas" : GET("/secured/area/locate/{city}?token=…")	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.

Valid arguments	The Controller sends an answer with a 200 OK HTTP code. In the Body of the HTTP answer there's JSON data containing a list of the retrieved AreaPOJO.

3.1.4.7 Mobile Client, UnlockCar Controller

"Unlock Car" : GET("/secured/area/unlock/{GpsPosition}?token=…")	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	In the body of the request sent by the Mobile Client is attached the ReservationPOJO. The Controller sends an answer with a 200 OK HTTP code. The Server will then contact the Car Client to unlock its doors.

3.1.4.8 Car Client, EndRide Controller

"End Ride" : PUT("/unsecured-cars/{carId}/end",body(GPSCoordinates=…")	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. No further data is attached to the answer.

3.1.4.9 Car Client, MajorDamage Controller

"Report Damage" : POST("/unsecured-cars/{carId}/majordamage",body(timestamp=…,description=…")	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.

Valid arguments	The Controller sends an answer with a 200 OK HTTP code. No further data is attached to the answer.

3.1.4.10 Car Client, StartDrive Controller

“Start Drive” : POST(“/unsecured-cars/{carId}/start”,body(timestamp=…)	
Input	*Effect*
Invalid arguments	An answer with the 404 Not Found HTTP error code is forwarded from the Controller to the client.
Valid arguments	The Controller sends an answer with a 200 OK HTTP code. No further data is attached to the answer.

Chapter 4

Tools Required

4.1 Tools used for testing

For what concerns the Unit Testing part the tools used are the **JUnit framework** and to adopt the testing technique called “Mock Objects” **Mockito** is the other tool.

For the Integration Testing instead, we’ll use the **Arquillian integration testing framework.** This tool runs tests against a Java container to check the correct behavior between a component and its surrounding execution environment. More specifically Arquillian will help checking if the right component is injected in a specified dependency injection.

Chapter 5

Hours of Work

5.1 Hours

Alessandro Paglialonga was the main author of this document, he spent about 35 hours to complete it.

Simone Perriello helped planning the drawing of this document, reviewing and redacting it with an overall contribution of 2 hours.

Enrico Migliorini helped planning the drawing of this document, reviewing it with an overall contribution of 1 hour.