PowerEnJoy, ITPD

Riccardo Cattaneo 873647
Fabio Chiusano 874294

Version: 1.0.0
Release date: 14-01-2017

Table of Contents

1 Introduction

1.1 Revision History

1.1.1 Version 1.0

The first release, it has been delivered online before the deadline.

1.2 Purpose and Scope

This Integration Test Plan Document aims at pointing out how to accomplish integration tests. Developers, testers and, in general, all the people involved in the development of the PowerEnjoy System should read this document before starting testing of the integration of components.
This document aims at explaining to the development team what to test, in which sequence, which tools are needed for testing, and which stubs/drivers/oracles need to be developed.
Particularly, in the following sections we have found out what are the entry criteria for integration testing, what are components and subcomponents to be integrated and what are the dependencies between them.
Then we have described every integration test, pointing out type of tests required and functions involved. We have also provided a list of drivers or stubs, and non-common test data needed for each integration test.
Finally we have described what are the tools that we are going to use during integration testing, in order to make this process easier.

1.3 List of Definitions and Abbreviations

- BL: Business Logic
- ITPD: Integration Test Plan Document
- DD: Design document;
- ER: Entity-Relationship diagram;
- RASD: Requirements Analysis and Specification Document;
- JSE: Java Serial Edition;
- JEB: Java Enterprise Bean;
- REST: REpresentantional State Transfer;
- RESTful service: REST compliant service;
- UX: User eXperience;
- JEE: Java Enterprise Edition;
- JAX-RS: Java API for RESTful web Services;
- WildFly: JEE open source application server;

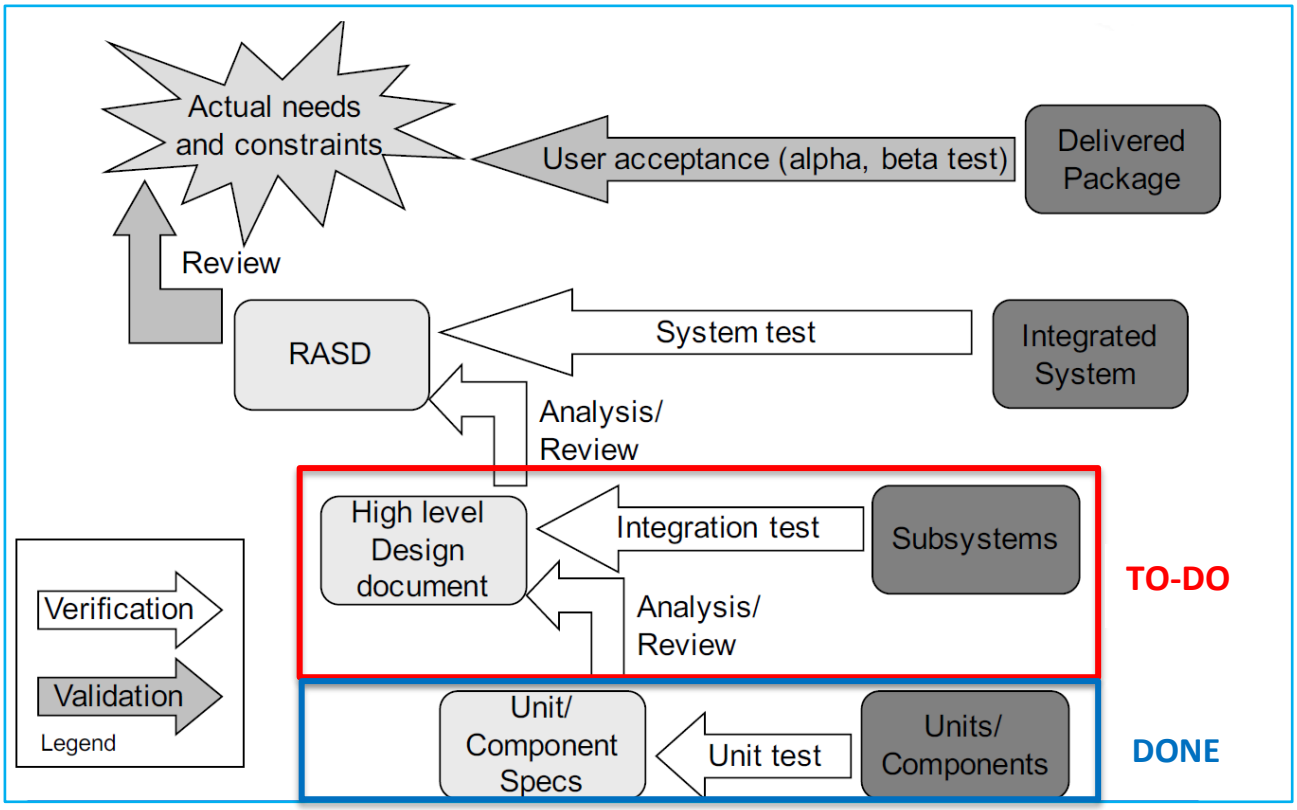
- JPA: Java Persistence API;
- Cordova: mobile cross-platform development framework;
- PhoneGap: mobile cross-platform development framework that works over Cordova;
- JSP: Java Server Pages.

1.4 List of Reference Documents

- Specification Document: Assignments AA 2016-2017.pdf;
- RASD v1.1 Document;
- Design Document v1.0;
- ITPD Examples from previous years;
- SPINGrid Integration Test Plan Document example;

2 Integration Strategy

2.1 Entry Criteria



We are to design and plan the Integration Test, which aims to verify that software component work with each other and cooperate in the right and expected way. Hence, in order to make possible the integration testing and to produce meaningful

Results, it is supposed that each component works well individually and this can be formally proved with Unit Tests.
We assume that Low-level code is already tested, functions of every component are covered with unit tests, with mainly a white-box approach.

2.2 Elements to be integrated

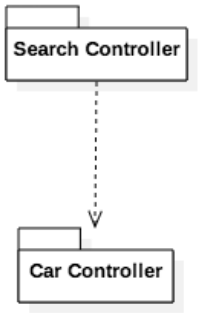
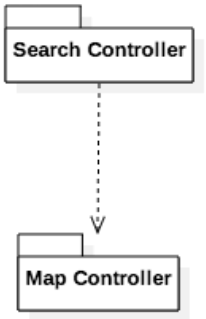
2.2.1 Elements Identification

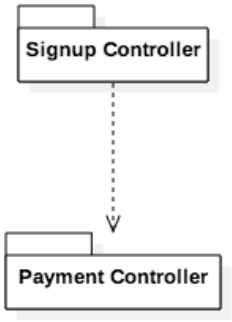
Starting the analysis from a high-level view, it is necessary to integrate and test software tiers in the main server: clients with the web tier, web tier with the business logic tier, and this one with the persistence manager and the database. It is also necessary to test interaction between external handlers and corresponding controllers. In a more low-level view, it is necessary to integrate every controller with related ones for example the Search Controller with the Map Controller.
It is also necessary to test the interaction between controllers developed in the car software, and between cars and server.

2.2.2 Dependencies Identification

In order to find precisely the components that must be integrated, we should analyse their dependencies.
We have added the main dependencies, listing major function involved.

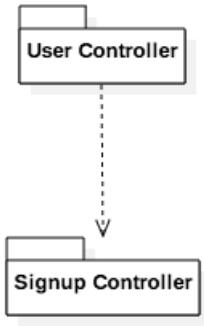
Main Server Dependencies

	SearchController.findCars() calls: • CarController.getLocation() • CarController.isInSafeArea() • CarController.getBatteryLevel() • CarController.isPlugged() • SearchController.findSafeAreas()
	SearchController.getMap() calls: • MapController.getMap()

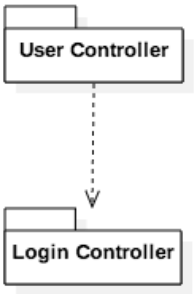


SignupController.signup() calls:

- PaymentController.checkPaymentInfo()

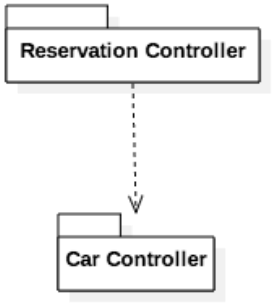


UserController.signup(cred: FullCredential, pi: PaymentInfo)



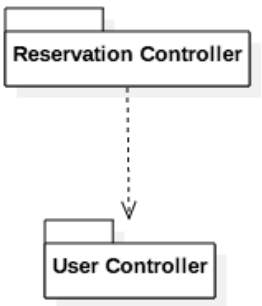
UserController.login() calls:

- LoginController.login()

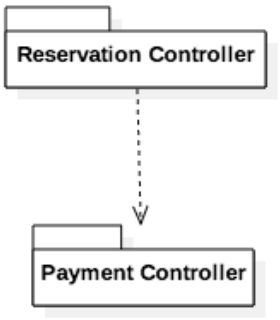


ReservationController.reserveCar() and
ReservationController.setReservationExpired()
call:

- ReservationController.applyExpirationFee()
- ReservationController.addExpirationTimer ()
- ReservationController.createReservationCode()
- ReservationController.checkReservationCode()

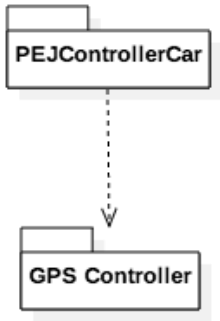


UserController.findUser()



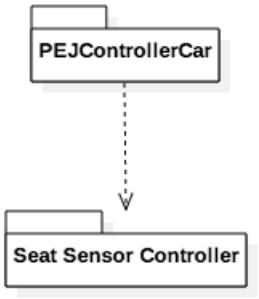
PaymentController.makePayment()

Car Server Dependencies



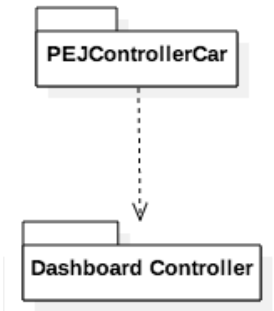
PEJControllerCar.sendLocation() calls:

- GPSController.getLocation()



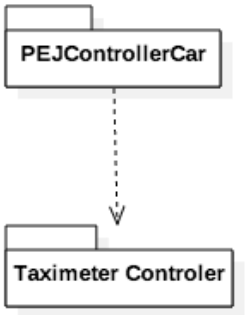
PEJControllerCar.numOfSeatsUsed() calls:

- SeatSensorManager.isSeatUsed()
- SeatSensorManager.numOfSeatsUsed()



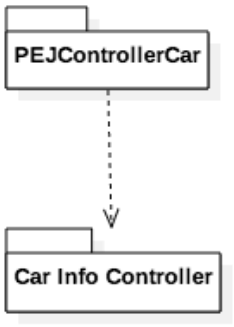
PEJControllerCar.showReservationCode() and PEJControllerCar.hideReservationCode() call:

- DashboardController.showReservationCode()
- DashboardController.hideReservationCode()



PEJControllerCar.getTaximeterValue() and
PEJControllerCar.resetTaximeter() call:

- TaximeterController.getValue():
- TaximeterController.reset()



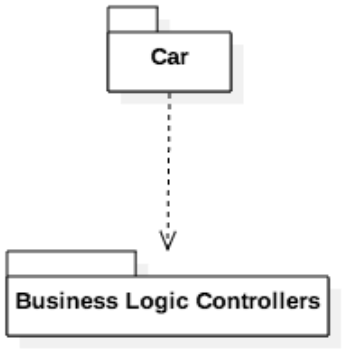
PEJControllerCar.rideFinished() calls:

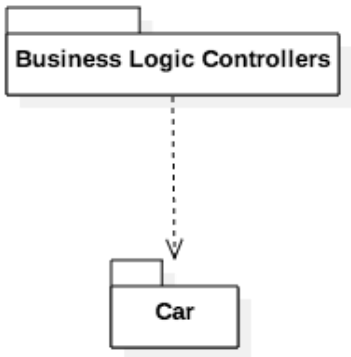
- CarInfoManager.getBatteryLevel()
- CarInfoManager.lockDoors()
- CarInfoManager.isEngineOn()
- CarInfoManager.isPlugged()

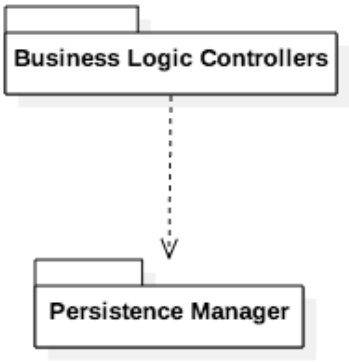
The system can be divided into subsystems:

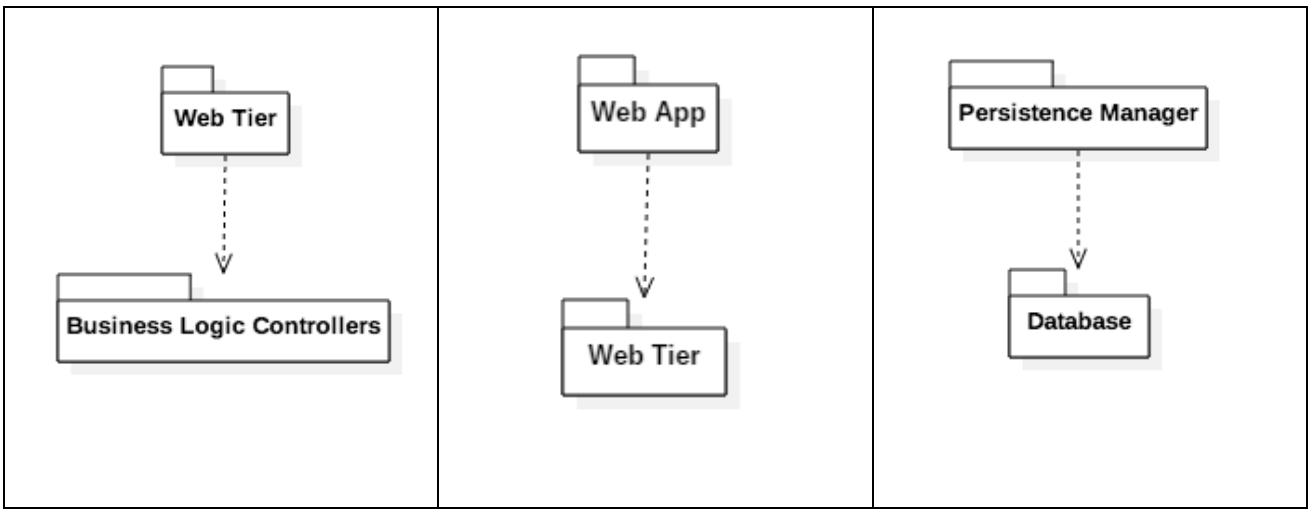
- Database;
- Persistence Manager;
- Business Logic Controllers;
- Car;
- Web Tier;
- Web App.

These are their dependences, which will be better analysed in the next chapter:









2.3 Integration Testing Strategy

For testing, we choose the bottom-up approach. Since there is not an old system to support, the project will be built up starting from the ground up.
By choosing bottom-up approach, it is possible to test integration of components as they are ready, with no further delay. Thus, we can start performing integration testing earlier in the development process as soon as the required components have been developed in order to maximize parallelism and efficiency.
By testing in an incrementally fashion, we also make it easier to track bugs in the integration progress and take the necessary measures to correct them on time.
Another consequence of the fact that we use a bottom-up approach is that no stubs are needed, unless special cases due to non-trivial components dependencies.

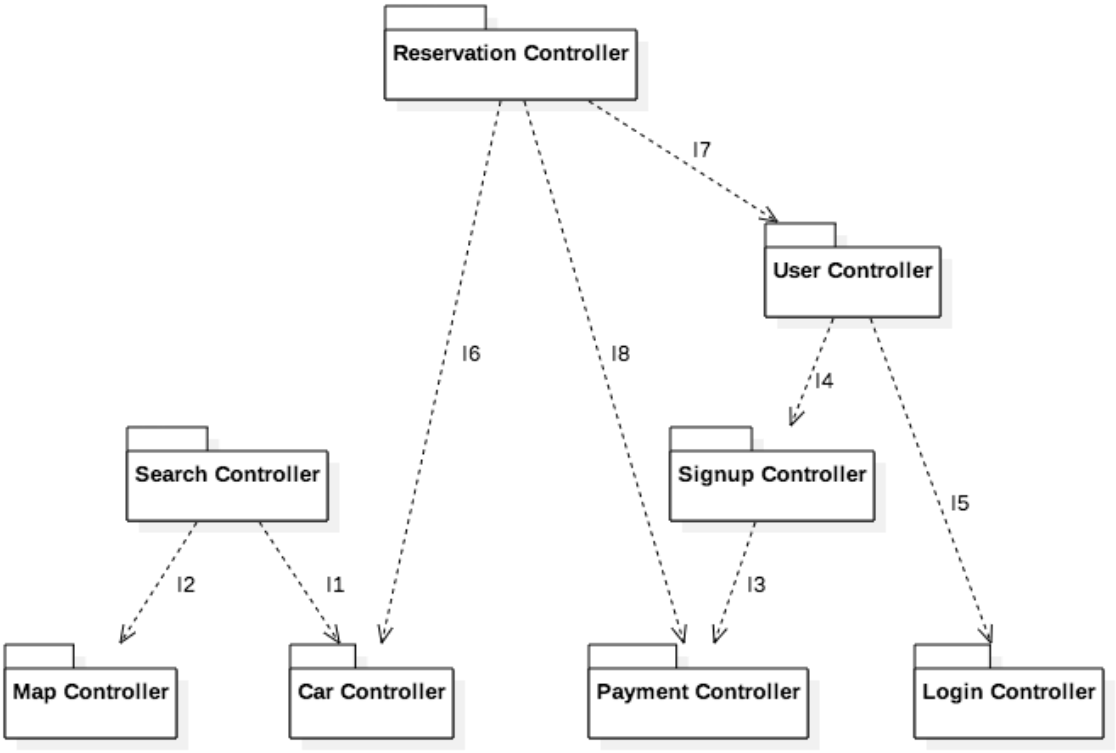
2.4 Sequence of Component – Function Integration

2.4.1 Software Integration Sequence

Components have to start to be integrated starting from low-level ones.
This process brings to different higher-level and integrated sub-systems.
Every integration test in each section can be performed in parallel with the others.
We suppose that the communication with external handlers, namely between search-on-a-map handler and the Map Controller, and between Payment Handler and Payment Controller, has been already performed while testing single components.

2.4.1.1 Main Server Business Logic Components Integration

Starting from the analysis previously done in chapter 2.2.2, we have built this is the dependency graph of the main server logic components:



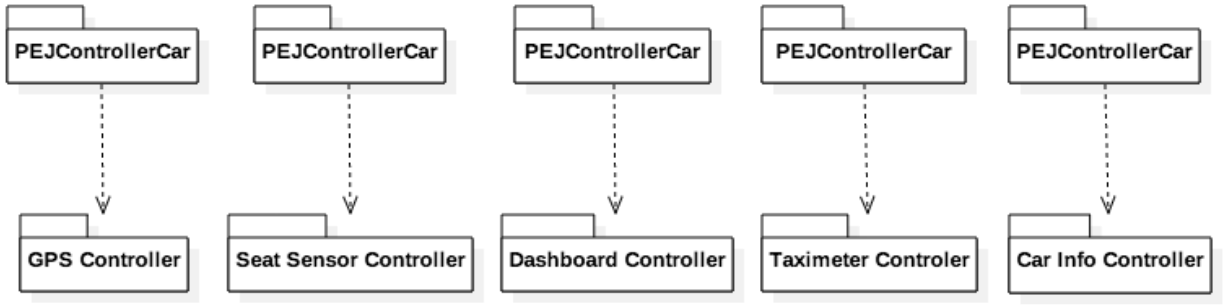
Thanks to the bottom-up approach, integration testing can be easily parallelized. Some integration tests can be performed in parallel (e.g. I1 and I3), hence the order is not mandatory. They can be carried out as components are finished to be developed and unit tested.

This is a possible order for the integration tests that is consistent with the dependencies previously found.

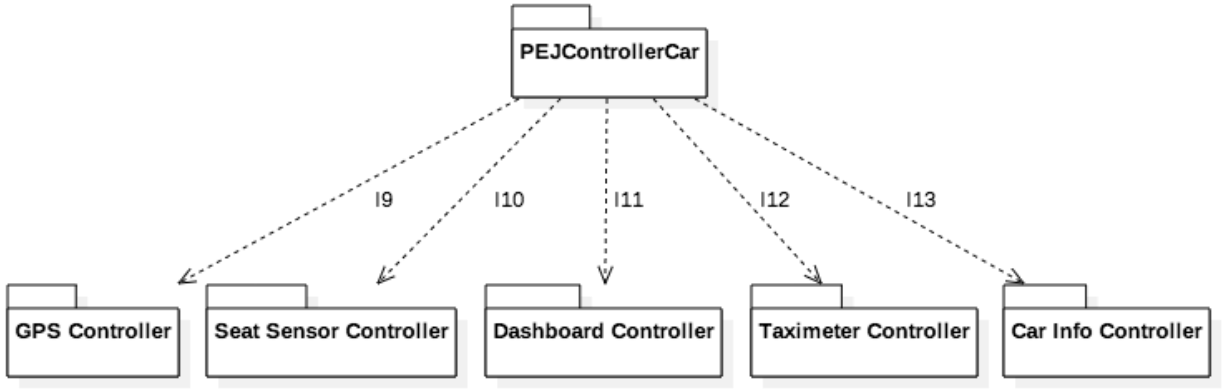
Step	Parallel Development
1	I1, I2, I3
2	I4, I5
3	I6, I7, I8

2.4.1.2 Car Components Integration

Software components that run on the car software are centred on the main controller, called PEJControllerCar. These are the main dependencies.



When the other controllers are finished, they can be integrated and tested with the main one. This is the dependences graph.



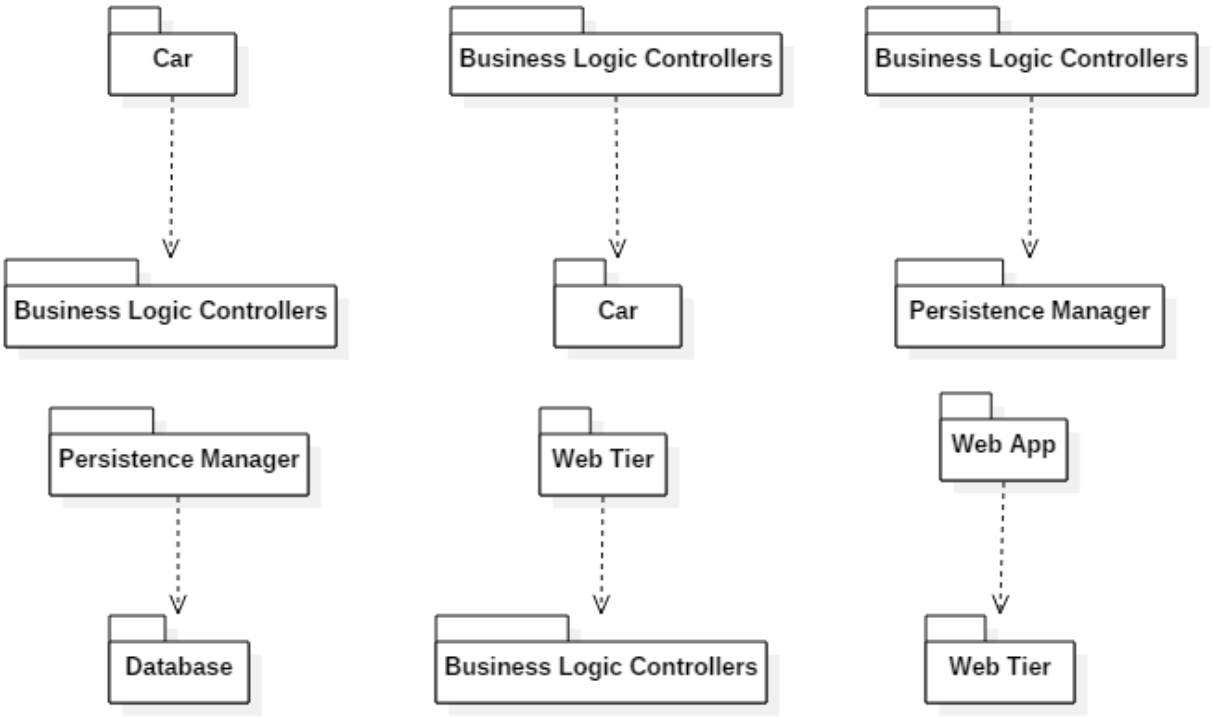
Since the dependency graph is a tree with height one, it's possible to parallelize all the integrations.

Step	Parallel Development
1	I9, I10, I11, I12, I13

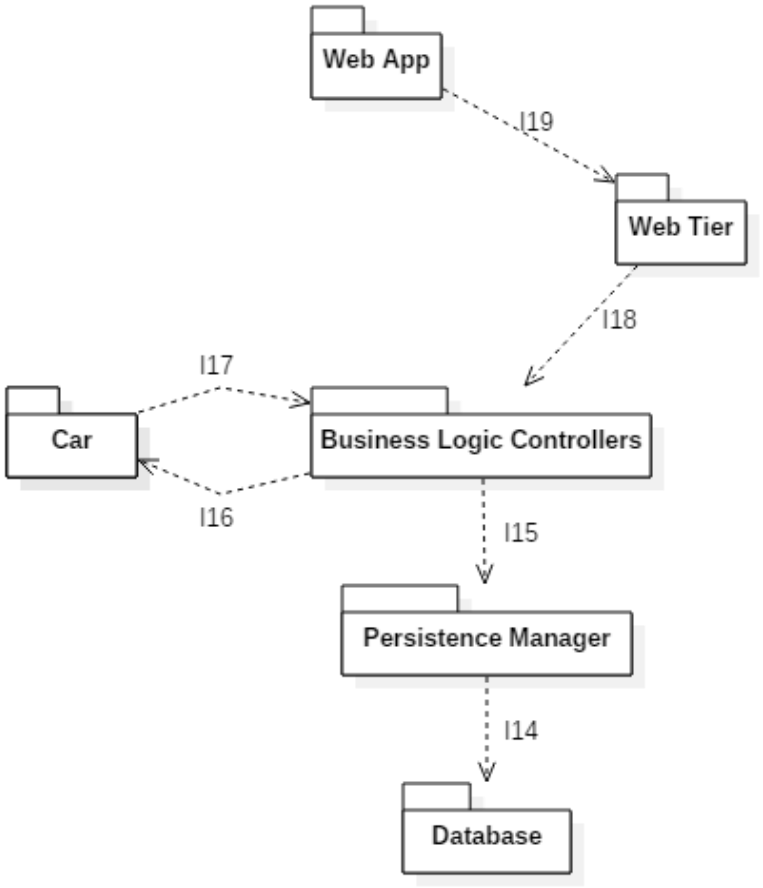
2.4.2 Subsystem Integration Sequence

After having integrated and tested components in the main server and in the car, it is possible to move on higher-level software modules. It is possible to test the integration between tiers on the main server, client and server, server and database, and between car and server.

These are the main dependencies between subsystems.



This is the dependencies graph that is consistent with the dependencies previously found.



This is a possible order for the integration tests that is consistent with the dependencies previously found.

Step	Parallel Development
1	I14
2	I15, I17 (with the help of a stub of Business Logic Controllers)
3	I16, I18
4	I19

After doing this, it will be possible to move to the next phase and test the whole system. Anyway, the complete System Test verification and validation phase has not been detailed in this document.

3 Individual Steps and Test Description

3.1 Tests overview

These are the software integration tests for the main server that we propose along with the components involved.

Integration test	Components involved
I1	Search Controller ⟶ Car Controller
I2	Search Controller ⟶ Map Controller
I3	Signup Controller ⟶ Payment Controller
I4	User Controller ⟶ Signup Controller
I5	User Controller ⟶ Login Controller
I6	Reservation Controller ⟶ Car Controller
I7	Reservation Controller ⟶ User Controller
I8	Reservation Controller ⟶ Payment Controller

These are the software integration tests for the car that we propose along with the components involved.

Integration test	*Components involved*
I9	PEJControllerCar ⟶ GPS Controller
I10	PEJControllerCar ⟶ Seat Sensor Controller
I11	Car Controller ⟶ Dashboard Controller
I12	Car Controller ⟶ Taximeter Controller
I13	Car Controller ⟶ Car Info Controller

These are the subsystem integration tests we propose along with the subsystems involved.

Integration test	*Subsystems involved*
I14	Persistence Manager ⟶ Database
I15	Business Logic Controllers ⟶ Persistence Manager
I16	Business Logic Controllers ⟶ Car
I17	Car ⟶ Business Logic Controllers
I18	Web Tier ⟶ Business Logic Controllers
I19	Web App ⟶ Web Tier

3.2 Main Server tests description

3.2.1 I1 – Integration Test 1

Test Items	Search Controller SC → Car Controller CC
Type of Tests	• SC can retrieve correctly information of a car (position, battery level, isPluggled, status…) from CC • SC and CC exchange addresses and coordinates with same format • the SC can find all and only the cars that are available in a given address within a certain range • car status and address errors are handled correctly

SearchController.findCars(loc: Location, radius: int): List<CarInfo>	
Input	Effect
A valid location *loc* (i.e. consistent coordinates) and a valid radius *radius* (i.e. positive integer)	Returns a list filled with CarInfo objects, that contain the locations of the available cars in the circle of center *loc* and radius *radius* along with other info such as the battery level
A not valid location or a not valid radius	Returns a SearchInfoException

We need to make sure that the following methods works too.

CarController.getLocation(carID: int): Location	
Input	Effect
The id of a car in the database	Returns a Location object, containing the car position as it is stored in the database
An integer that does not correspond to a car in the database or null	Returns a NoCarFoundException

CarController.isInSafeArea(carID: int): bool	
Input	Effect
The id of a car in the database	Returns true if the car is parked in a safe area, false otherwise
An integer that does not correspond to a car in the database or null	Returns a NoCarFoundException

CarController.getBatteryLevel(carID: int): int	
Input	Effect
The id of a car in the database	Returns an integer between 0 and 10 that represents the amount of energy stored in the car
An integer that does not correspond to a car in the database or null	Returns a NotCarFoundException

CarController.isPlugged(carID: int): bool	
Input	Effect
The id of a car in the database	Returns true if the car corresponding to *carID* is plugged, false otherwise
An integer that does not correspond to a car in the database or null	Returns a NoCarFoundException

SearchController.findSafeAreas(onlySpecial: bool): List<AreaInfo>	
Input	Effect
onlySpecial = false	Returns a list of AreaInfo objects that contain info of such areas for both special safe areas and normal safe areas
onlySpecial = true	Returns a list of AreaInfo objects that contain a set of special safe area info

3.2.2 I2 – Integration Test 2

Test Items	Search Controller SC → Map Controller MC
Type of Tests	• SC can retrieve correctly a map from MC • SC and MC exchange addresses and coordinates with same format • the SC shows cars over the map in the right place • map and address errors are handled correctly • SC returns to the caller a right formatted map

SearchController.getMap(center: Location, radius: int): MapImage	
Input	Effect
A valid location and a valid radius	If MapController does not return an exception, then the SearchController returns a MapImage object that is an image of the requested map. If MapController returns a MapHandler, then the SearchController returns a MapHandlerException too.
A not valid location ot a not valid radius	Returns a SearchInfoException

We need to make sure that the following methods works too.

MapController.getMap(center: Location, radius: int): MapImage	
Input	Effect
A valid location and a valid radius	If MapHandler does not return an exception, then the MapController returns a MapImage object that is an image of the requested map. If MapHandler returns an exception, then the MapController returns a MapHandlerException
A not valid location ot a not valid radius	Returns a SearchInfoException

3.2.3 I3 – Integration Test 3

Test Items	Signup Controller SC → Payment Controller PC
Type of Tests	• PC checks the payment info passed to it by SC

SignupController.signup(cred: FullCredential, pi: PaymentInfo)	
Input	Effect
All input is valid	The user is registered in the database
cred.email already present in the database	Throws an EmailAlreadyUsedException
cred.username already present in the database	Throws a UsernameAlreadyUsedException
Payment info not valid	Throws a PaymentInfoException

We need to make sure that the following methods works too.

PaymentController.checkPaymentInfo(pi: PaymentInfo)	
Input	Effect
Valid payment info	Nothing
Not valid payment info	Throws an PaymentInfoException

3.2.4 I4 – Integration Test 4

Test Items	User Controller UC → Signup Controller SC
Type of Tests	• UC correctly identifies if a user is already registered to the system • SC and UC exchange data with the same format • signup errors are handled correctly

UserController.signup(cred: FullCredential, pi: PaymentInfo)	
Input	Effect
All input is valid	The user is registered in the database
cred.email already present in the database	Throws an EmailAlreadyUsedException
cred.username already present in the database	Throws a UsernameAlreadyUsedException
Payment info not valid	Throws a PaymentInfoException

3.2.5 I5 – Integration Test 5

Test Items	User Controller UC → Login Controller LC
Type of Tests	• LC can retrieve correctly user's information from the UC • LC and UC exchange data with the same format • login errors are handled correctly

UserController.login(ui: UserInfo)	
Input	Effect
Valid user info	The user is logged into the system
Username not present in the database	Throws an NoUserFoundException
Username present in the database but wrong password	Throws a WrongPasswordException

We need to make sure that the following methods works too.

LoginController.login(ui: UserInfo)	
Input	Effect
Valid user info	The user is logged into the system
Username not present in the database	Throws an NoUserFoundException
Username present in the database but wrong password	Throws a WrongPasswordException

3.2.6 I6 – Integration Test 6

Test Items	Reservation Controller RC → Car Controller CC
Type of Tests	• RC can retrieve correctly car information from CC • RC and CC exchange data with the same format • RC can correctly set the car as reserved • RC can correctly set and hide the reservation code on the reserved car • RC and CC can correctly manage the lock and unlock of the car •RC can set a reservation expired and set the car available again • reservation and car errors are handled correctly

ReservationController.reserveCar(info: ReservationInfo)	
Input	Effect
Valid reservation info	The car is reserved: it's locked, not available, the reservation code is shown in the dashboard and it's possible to insert it in the application, the reservation timer starts, a reservation object is allocated
info.carID not in the database	Throws a NoCarFoundException
info.user not in the database	Throws a NoUserFoundExcpetion

ReservationController.setReservationExpired(resID: int)	
Input	Effect
Valid reservation identifier	The reservation is marked as expired, the car is set available and unlocked, the fee is applied
Not valid reservation identifier	Throws an NoReservationFoundException

We need to make sure that the following methods works too.

ReservationController.applyExpirationFee(resID: int)	
Input	Effect
Valid reservation identifier	The expiration fee is applied to the user associated with the reservation
Not valid reservation identifier	Throws an NoReservationFoundException
Payment info of the user are not valid	Throws a PaymentInfoException

ReservationController.addExpirationTimer (resID: int)	
Input	Effect
Valid reservation identifier	The reservation timer immediately starts
Not valid reservation identifier	Throws an NoReservationFoundException

ReservationController.createReservationCode(): int	
Input	Effect
Nothing	Returns a valid reservation code, that is a reservation code not already used in the database

ReservationController.checkReservationCode(resID: int)	
Input	Effect
Valid reservation identifier	Nothing
Not valid reservation identifier	Throws an NoReservationFoundException

3.2.7 I7 – Integration Test 7

Test Items	Reservation Controller RC → User Controller UC
Type of Tests	• RC can use UC to know if a username corresponds to a user in the database

UserController.findUser(username: String)	
Input	Effect
User with username *username* present in the database	Nothing
username not present in the database	Throws an NoUserFoundException

3.2.8 I8 – Integration Test 8

Test Items	Reservation Controller RC → Payment Controller PC
Type of Tests	• RC can correctly make a payment via PC • RC and PC exchange data with the same format • payment errors are handled correctly

PaymentController.makePayment(pi: PaymentInfo, amount: MoneyAmount)	
Input	Effect
Valid data	The payment is correctly carried out
Not valid payment info	Throws a PaymentInfoException
Not enough money to make the payment	Throws a NotEnoughMoneyException

3.3 Car tests description

3.3.1 I9 – Integration Test 9

Test Items	PEJControllerCar PJC → GPS Controller GC
Type of Tests	• PJC can retrieve correctly actual location information from GC • GC returns to the caller a right formatted coordinate • GPS position errors are handled correctly

PEJControllerCar.sendLocation(carID: int, loc: Location)	
Input	Effect
carID is an ID of a car that is present in the database	The main server updates the database with the new location of the car
carID not present in the database	Throws an NoCarFoundException
loc is not a valid location	Throws a LocationNotValidException

GPSController.getLocation(): Location	
Input	Effect
Nothing	If the GPS module works ok, it returns a valid Location object. It throws a GPSSensorNotWorkingException otherwise

3.3.2 I10 – Integration Test 10

Test Items	PEJControllerCar PJC → Seat Sensor Controller SC
Type of Tests	• PJC can retrieve correctly seat information from SC • PJC can understand how many passengers there are inside the car • PJC and SC exchange data with the same format • sensor errors are handled correctly

PEJControllerCar.numOfSeatsUsed(): int	
Input	Effect
Nothing	If the sensors work ok: it returns the number of seats used. If at least one sensor doesn't work: it throws a SeatSensorNotWorkingException

SeatSensorManager.isSeatUsed(seat: int): bool	
Input	Effect
The integer *seat* corresponds to an actual seat of the car.	If the sensors work ok: it returns true is the seat is used, it returns false if not used. If the sensors don't work: it throws a SeatSensorNotWorkingException
The integer *seat* corresponds to an actual seat of the car.	Throws a InvaildSeatException

SeatSensorManager.numOfSeatsUsed(): int	
Input	Effect
Nothing	If the sensors work ok: it returns the number of seats used. If at least one sensor doesn't work: it throws a SeatSensorNotWorkingException

3.3.3 I11 – Integration Test 11

Test Items	PEJControllerCar PJC → Dashboard Controller DC
Type of Tests	• PJC can send and show correctly information to DC • PJC can show and hide the reservation code • PJC and DC exchange data with the same format • dashboard errors are handled correctly

PEJControllerCar.showReservationCode(code: int)	
Input	Effect
An integer that corresponds to the reservation code	If the dashboard works: the reservation code is shown on the dashboard. If the dashboard does not work: it throws a DashboardNotWorkingException

PEJControllerCar.hideReservationCode()	
Input	Effect
An integer that corresponds to the reservation code	If the dashboard works: the reservation code is hidden from the dashboard. If the dashboard does not work: it throws a DashboardNotWorkingException

DashboardController.showReservationCode(code: int)	
Input	Effect
An integer that corresponds to the reservation code	If the dashboard works: the reservation code is shown on the dashboard. If the dashboard does not work: it throws a DashboardNotWorkingException

DashboardController.hideReservationCode()	
Input	Effect
An integer that corresponds to the reservation code	If the dashboard works: the reservation code is hidden from the dashboard. If the dashboard does not work: it throws a DashboardNotWorkingException

3.3.4 I12 – Integration Test 12

Test Items	PEJControllerCar PJC → Taximeter Controller TC
Type of Tests	• PJC can retrieve correctly taximeter information from TC • Calculated fees are proportional to the time elapsed • taximeter errors are handled correctly

PEJControllerCar.getTaximeterValue(): MoneyAmount	
Input	Effect
Nothing	If the taximeter works: returns the value stored in the taximeter. If the taximeter does not work: it throws a TaximeterNotWorkingException

PEJControllerCar.resetTaximeter()	
Input	Effect
Nothing	If the taximeter works: reset the taximete.r If the taximeter does not work: it throws a TaximeterNotWorkingException

TaximeterController.getValue(): MoneyAmount	
Input	Effect
Nothing	If the taximeter works: returns the value stored in the taximeter. If the taximeter does not work: it throws a TaximeterNotWorkingException

TaximeterController.reset()	
Input	Effect
Nothing	If the taximeter works: reset the taximeter. If the taximeter does not work: it throws a

3.3.5 I13 – Integration Test 13

Test Items	PEJControllerCar PJC → Car Info Controller IC
Type of Tests	• PJC can retrieve correctly car status information from IC • PJC can lock and unlock doors when is needed • PJC and IC exchange data with the same format • sensors and locking errors are handled correctly

PEJControllerCar.rideFinished(rideInfo: RideInfo)	
Input	Effect
Valid ride info	If the internet module works: it sends to the main server info about the ride. If the engine sensor does not work: it throws a InternetModuleNotWorkingException

CarInfoManager.getBatteryLevel(): int	
Input	Effect
Nothing	If the battery sensor works: returns an integer that represents the battery level of the car If the battery sensor does not work: it throws a BatterySensorNotWorkingException

CarInfoManager.lockDoors(lock: Bool)	
Input	Effect
lock = true	If the doors module works: locks the doors If the doors module does not work: it throws a DoorsModuleNotWorkingException
lock = false	If the doors module works: unlocks the doors If the doors module does not work: it throws a DoorsModuleNotWorkingException

CarInfoManager.isEngineOn(): Bool	
Input	Effect
Nothing	If the engine sensor works: returns true if the engine is on, false if it is not. If the engine sensor does not work: it throws a EngineSensorNotWorkingException

CarInfoManager.isPlugged(): bool	
Input	Effect
Nothing	If the plug sensor works: returns true if the car is plugged, false if it is not. If the plug sensor does not work: it throws a PlugSensorNotWorkingException

3.4 Subsystems tests description

3.4.1 I14 – Integration Test 14

Test Items	Persistence Manager ⟶ Database
Type of Tests	• SQL Queries are correctly formed and accepted by the DBMS • Answers from the database are consistent • The persistence manager keeps data updated and consistent with the Database • Data errors are handled correctly

A possible way to test the interaction between the Persistence Manager and the Database is to:

1. look for a particular data in the database. If that data is not present, then it's possible to continue to step 2;
2. create that data and insert it into the database;
3. search for it: the data should now be found;
4. modify it;
5. search for it: modifications should be correctly applied;
6. delete it;
7. search for it: data should not be found.

We should do these steps for all the types of data that the Persistence Manager manages, that is for all the data in the tables of the database:

- User;
- PaymentInfo;
- Reservation;
- Ride;
- Car;
- SafeArea;
- SpecialSafeArea.

3.4.2 I15 – Integration Test 15

Test Items	Business Logic Controllers ⟶ Persistence Manager PM
Type of Tests	• Controllers can retrieve correctly persistent information through PM •BLC and PM exchange data with the compatible formats • Controllers can save information correctly • not compatible information (such as out of bound values) are managed correctly

We should check that all the Business Logic Controllers use the Persistence Manager in a correct way.

A possible way to do this is to test all the methods that interacts with the persistent manager, that are the methods that needs to interact with the database. We can then analyse the tables in the database and find for each one of them some methods in the Business Logic Controller that use it.

For each table in the database we can test:

- User:
    - UserController.login;
    - UserController.signup;
    - UserController.findUser;
- PaymentInfo:
    - UserController.signup;
    - UserController.login;
- Reservation:
    - ReservationController.reserveCar;
    - UserController.getReservationHistory;
- Ride:
    - UserController.getReservationHistyory;
    - ReservationController.applyDiscount;
- Car:
    - ReservationController.reserveCar;
    - CarController.getLocation;
    - CarController.setReserved;
    - SearchController.findCars;
- SafeArea:
    - SearchController.findParkings;
- SpecialSafeArea:
    - SearchController.findParkings;

3.4.3 I16 – Integration Test 16

Test Items	Business Logic Controllers → Car
Type of Tests	• Controllers, located in the main server, can retrieve correctly information about the status of the car, its position, passengers on it, and others. •BLC and Car exchange data with the compatible formats • The main server can correctly display information on the car • The main server can understand when the ride is finished, and then perform consequent task • the communication between server and car satisfy some performance constraints

The Business Logic Controllers can directly interact with any car of the system. A possible way to test this interaction is to test the methods that interact with the cars, i.e. those methods that look for car information not stored in the database.

Some methods to be tested are the sequent:

- CarController.getBatteryLevel: interacts with the car and gets its battery level;
- CarController.unlock: interacts with the car and locks or unlocks it;
- CarController.isPlugged: interacts with the car and checks if it is plugged or not.

3.4.4 I17 – Integration Test 17

Test Items	Car → Business Logic Controllers
Type of Tests	• Sends ride info to the main server when the user finishes using the car • Sends the car location to the main server when the user finishes using the car

The Car subsystem needs to interact with the main server too.

Those interactions are:

- PEJControllerCar.rideFinished: when the user finishes using the car, the car automatically informs the main server that the rides has finished and sends its information to it;
- PEJControllerCar.sendLocation: when the user finishes using the car, the car automatically informs the main server about the car location. Since the car location is useless to other users while a user is using it (and therefore the car should not be displayed on the web app maps), the car location in the main server database is updated only at the end of each ride in order to increase performance.

3.4.5 I18 – Integration Test 18

Test Items	Web Tier → Business Logic Controllers
Type of Tests	• All possible client request are correctly understood • Controllers are called in the right order • Controller answers are consistent and are correctly managed by the web tier • Web tier answers are well formatted • possible data error are handled correctly

We need to make sure that the web tier make a correct use of the Business Logic Controllers in order to show the web pages on the web app.

All the methods that interact with the main server should be tested. All these methods are in the PEJControllerUser class, that acts as a façade of the server.
Those methods are:

- PEJControllerUser.sendLocation;
- PEJControllerUser.login;
- PEJControllerUser.signup;
- PEJControllerUser.setPaymentInfo;
- PEJControllerUser.makeReservation;
- PEJControllerUser.getReservationHistory;
- PEJControllerUser.gatMap;
- PEJControllerUser.getCarsLocations;
- PEJControllerUser.sendReservationCode;

PEJControllerUser.gegtActiveReservations.

3.4.6 I19 – Integration Test 19

Test Items	Web App → Web Tier
Type of Tests	• The main server can manage all possible HTTP requests from the Web App • The web app displays correctly the answer received • communication protocols are correctly managed • communication errors are handled gracefully • HTML forms send by the server are consistent syntactically

The communication between app and main server is tested here.
A possible (an easy) way to test it is with manual testing. Every type of HTTP request and response has to be tested, searching for data format error or request interpretation problems. Manual testing helps also to check if forms from the main server are displayed correctly.
Consistency and correctness of these forms has already tested in previous integration tests.

4 Tools and Test Equipment Required

4.1 Tools

The tools needed to accomplish the integration are the following.

4.1.1 Unit testing

JUnit for Java: a unit testing framework. It has been important for the development of test-driven development, and is one of a family of unit testing frameworks, which is collectively known as xUnit.

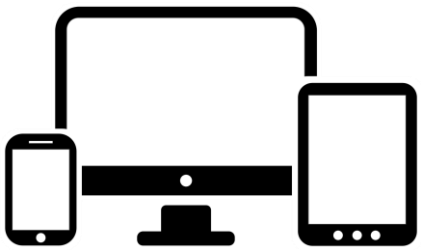
Mockito for Java: an open source testing framework released under the MIT License. The framework allows the creation of test double objects (mock objects) in automated unit tests for the purpose of test-driven development or behaviour-driven development.

QUnit for JavaScript: used heavily by the jQuery Project for testing jQuery, jQuery UI and jQuery Mobilt. It is a generic framework to test any JavaScript code. It supports server-side (e.g. node.js) and client-side environments.

4.1.2 Intergation testing

Mockito for Java components on the main server and on the car software.

Arquillian for Java components on the main server and on the car software. Arquillian is a platform that simplifies integration testing for Java middleware. It deals with all the plumbing of container management, deployment and framework initialization.



Manual testing where other tools are not usable for some reasons (e.g. to test that the Web App subsystem correctly shows the Web Tier pages).

4.1.3 Code quality evaluation

SonarQube: an open source platform for continuous inspection of code quality. It supports both Java and JavaScript. It offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, potential bugs, comments, design and architecture. It records history and provides evolution graphs and differential views.

4.1.4 Performance testing

Apache JMeter: an open source software designed to load test functional behaviour and measure performance. It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyse overall performance under different load types.

4.2 Equipment required

Both client side and server side equipment is required.

4.2.1 Server side

In order to test the software, we need of course a server to run it on.

Instead of buying a physical server, we can take advantage of modern technologies and buy a subscription to a cloud IaaS platform.

We propose to buy a subscription to the Microsoft Azure IaaS service for a virtual machine with characteristics similar to the ones of the final server. The operating system installed will be of course the same of the one on the final server, that is WildFly v10.1.

As it concerns cars, at least two cars are needed to test the functionalities of our service. Both of them must have a server that runs our software. Their operating system will be the same of the one on the actual final cars, that is WildFly v1.0 again.

4.2.2 Client side

The PowerEnJoy service will be available to both smartphones (as an app) and browsers, therefore we should get the required equipment in order to test it.
As it concerns smartphones, we will try the mobile app on different operating systems and models in order to be as certain as possible that our product works for everyone it was conceived for.

The chosen equipment is:

- iPhone 6:
  - OS: iOS 9.2.0;
  - Display: 4.7 inches;
  - Resolution: 750 x 1334 pixels;
  - Chipset: Apple A8 (Dual-core 1.4 GHz Typhoon, ARM v8-based);
  - Memory: 16GB, 1GB RAM.
- iPhone 7 Plus:
  - OS: iOS 10.1.1;
  - Display: 5.5 inches;
  - Resolution: 1080 x 1920 pixels;
  - Chipset: Apple A10 Fusion (Quad-core 2.23 GHz);
  - Memory: 128GB, 3 GB RAM;
- Samsung Galaxy S6:
  - OS: Android OS, v5.0.2 (Lollipop);
  - Display: 5.1 inches;
  - Resolution: 1440 x 2560 pixels;
  - Chipset: Exynos 7420 Octa-core (4x2.1 GHz Cortex-A57 & 4x1.5 GHz C-A53);
  - Memory: 64GB, 3 GB RAM;
- Xiaomi Mi 5:
  - OS: Android OS, v6.0 (Marshmallow);
  - Display: 5.15 inches;
  - Resolution: 1080 x 1920 pixels;
  - Chipset: Qualcomm MSM8996 Snapdragon 820;
  - CPU: Quad-core (2x1.8 GHz Kryo & 2x1.36 GHz Kryo)
  - Memory: 64GB, 3 GB RAM;

As it concerns desktop browsers, the following will be tried:

- Mozilla Firefox, v45.4.0 ESR;
- Microsoft Edge, v38.14393;
- Internet Explorer for Windows, v11.0;
- Google Chrome, v55.0;
- Opera, v41.0.

5 Program Stubs and Test Data Required

Consistently with the testing strategy and test design, we now identify any program stubs or special test data required for each integration step.

5.1 Main Server Tests

In order to repeat test more than once with predictable results, it may be necessary to store some predefined answers given by components linked with dynamic real world data. For example, a test sample of data that describes the status of a particular car in a particular time, retrieved from Car Controller in the Main Server, should be stored for test purposes.
More details about testing strategies and simple data required to perform test are present in the section 3.2 of this document.

5.1.1 I1 – Integration Test 1

Test Items	Search Controller SC ⟶ Car Controller CC
Environmental Needs	Router Driver
Special test data	Sample of Car Server data given as answer to CC requests.

5.1.2 I2 – Integration Test 2

Test Items	Search Controller SC ⟶ Map Controller MC
Environmental Needs	Router Driver
Special test data	N/A

5.1.3 I3 – Integration Test 3

Test Items	Signup Controller SC ⟶ Payment Controller PC
Environmental Needs	User Controller Driver
Special test data	Some valid and real payment info

5.1.4 I4 – Integration Test 4

Test Items	User Controller UC ⟶ Signup Controller SC
Environmental Needs	Reservation Controller Driver
Special test data	Some user already signed up and added in the Database

5.1.5 I5 – Integration Test 5

Test Items	User Controller UC ⟶ Login Controller LC
Environmental Needs	Reservation Controller Driver
Special test data	Some user already signed up and added in the Database

5.1.6 I6 – Integration Test 6

Test Items	Reservation Controller RC → Car Controller CC
Environmental Needs	Router Driver
Special test data	Sample of Car Server data given as answer to CC requests.

5.1.7 I7 – Integration Test 7

Test Items	Reservation Controller RC → User Controller UC
Environmental Needs	Router Driver
Special test data	Some user already signed up and added in the Database

5.1.8 I8 – Integration Test 8

Test Items	Reservation Controller RC → Payment Controller PC
Environmental Needs	Router Driver
Special test data	Some valid and real payment info

5.2 Car Tests

5.2.1 I9 – Integration Test 9

Test Items	Car Controller CC → GPS Controller GC
Environmental Needs	Main Server Driver
Special test data	Sample of GPS Sensot Data

5.2.2 I10 – Integration Test 10

Test Items	Car Controller CC → Seat Sensor Controller SC
Environmental Needs	Main Server Driver
Special test data	Sample of Seat Sensor Data

5.2.3 I11 – Integration Test 11

Test Items	Car Controller CC → Dashboard Controller DC
Environmental Needs	Main Server Driver
Special test data	N/A

5.2.4 I12 – Integration Test 12

Test Items	Car Controller CC → Taximeter Controller TC
Environmental Needs	Main Server Driver
Special test data	Some samples of completed rides, in order to check fee calculation

5.2.5 I13 – Integration Test 13

Test Items	Car Controller CC → Car Info Controller IC
Environmental Needs	Main Server Driver
Special test data	N/A

5.3 Subsystems Tests

5.3.1 I14 – Integration Test 14

Test Items	Persistence Manager → Database
Environmental Needs	Business Logic Controllers Drivers
Special test data	Samples are created and deleted during tests

5.3.2 I15 – Integration Test 15

Test Items	Business Logic Controllers → Persistence Manager PM
Environmental Needs	Car Stub, Router Driver
Special test data	Data samples added to the database in order to allow execution of all the tests

5.3.3 I16 – Integration Test 16

Test Items	Business Logic Controllers → Car
Environmental Needs	Router Driver
Special test data	Different consistent car status

5.3.4 I17 – Integration Test 17

Test Items	Car → Business Logic Controllers
Environmental Needs	User Driver
Special test data	Samples of just finished rides

5.3.5 I18 – Integration Test 18

Test Items	Web Tier → Business Logic Controllers
Environmental Needs	Application Client Driver
Special test data	A partially populated database in order to test every logic function

5.3.6 I19 – Integration Test 19

Test Items	Web App → Web Tier
Environmental Needs	User Driver
Special test data	valid payment info, some GPS Data sample and have some reservation already registered in the Database

6 Effort Spent

We managed to distribute the workload fairly between days and team members in a way that allowed us to finish the draft of this document a few weeks before the deadline. A few days before we have delivered the document, we have reviewed it and integrated some concepts learned during class discussions.

Therefore, the total amount of time required to build this document is about 18 hours for each team member.