Politecnico di Milano

PowerEnjoy Service - Integration Test Plan Document

January 10, 2017

Version 1.1

Authors:

- Domenico FAVARO (Mat. 837995)
- Matheus FIM (Mat. 876069)
- Caio ZULIANI (Mat. 877266)

Prof. Elisabetta DI NITTO

Contents

1 Introduction

1.1 Revision History

This section records all revisions to the Document.

Version	Date	Authors	Summary
1.1	15/01/16	Domenico Favaro, Caio Zuliani, Matheus Fim	Initial Release

1.2 Purpose and Scope

The Integration Test Plan Document (ITPD) serves to present the integration sequence and testing for all Subsystems and Components that conform PowerEnjoy Car Sharing Service. This is a key part to guarantee the functioning and quality of the software. The Document will present the division of the System in Subsystems and Components that will endure individual testing as independent modules and then be subject to integration on the whole System.

1.3 Definitions and Abbreviations

- **RASD:** Reqirements And Specifications Document.
- **DD:** Design Document.
- **ITPD:** Integration Test Plan Document.
- **SDK:** Software Development Kit
- **App:** Application, refering to Web or Mobile App.
- **Subsystem:** Part of the system the generally encapsulates one or more features.
- **Component:** Self sustained part of the System that provides with functionalities and is part of one or more subsystems.
- **Bottom-up:** Referring to Bottom-up testing. Each component at lower hierarchy is tested individually and then the components that rely upon these components are tested.
- **Top-down:** Top-down integration testing is an integration testing technique used in order to mock or simulate the behaviour of the lower-level modules that are not yet integrated.
- **Mock:** Simulation that mimic the behavior of certain objects and fucntions in controlled ways, done to test the behavior of some other object.

For other concepts concerning the Service definition look in the **Glossary** section of the RASD and DD.

1.4 Reference Documents

- Specification Document: Assignments AA 2016-2017.pdf
- PowerEnjoy Requirements And Specifications Document (RASD)
- PowerEnjoy Design Document (DD)
- Example Document - Integration testing example document.pdf
- Testing Tools Documents:
  - Mockito
  - JMeter

2 Integration Strategy

2.1 Entry Criteria

We define the criteria that must be met before integration testing of the system components. We consider Integration a part of the production development. In order for production to start all documentation must first be written and up to date, including RASD and DD, to have a clear and full scope of the system components functionalities and importance. Once in production, the integration of a singe component can be done when the following criteria is met:

- The Component feature must be 100% complete, that is all classes and functions must have been implemented.
- No tickets must be opened for the Component, no bugs or cosidered missing features must be present.
- Individual component testing must have been performed, using JUnit to test its classes and functions.
- All the interfaces the Component has to communicate to have to be present or at least mocked to be able to test its coupling.

2.2 Elements to be Integrated

As stated in the Design Document, our system is composed by several High level Components presented in 3 tiers. Specifically these components are:

- Client Tier:
  - User Client Component
  - CRM Client Component
  - Car Component
- Server Tier:
  - User Controller
  - CRM Controller
  - Car Controller
  - Reservation Controller
  - Ride Controller
  - Payment Controller
  - User Report Controller

- Email Helper
- Location Helper
- Chat Service

- EIS Tier:
  - Database

2.3 Integration Testing Strategy

Our approach following the 3-tiered structure will follow a **Bottom-up** strategy, working on components that do not depend on others to function first.
This implies following the next tier order: EIS -> Server -> Client for development and testing.
Inside each Tier, Bottom-up strategy will be used again to integrate independent modules first and then those that depend on others.This strategy will help in contrast to Top-down to minimize the mock-up testing to be done, testing will be done on top of already deployed modules.
The order in which the 'Bottom' modules will be picked will follow a Critical-Module-First Integration Strategy, giving priority to those that will have dependencies of other modules, this will help not only to spot any error on critical modules first but also to unblock the integration of dependant modules earlier on.

2.4 Sequence of Component/Function Integration

Following the Bottom-up strategy we'll integrate the components that have no dependencies on other modules. First we'll present the component integration within each subsystem and then the subsystems integration order.

2.4.1 Software Integration Sequence

For each subsystem, we'll identify the sequence in which the software components will be integrated within the subsystem.

EIS Tier - DBMS: As shown in the DD our DBMS is not dependent on any module and even if it's a System already present for some structures (Cars, CRM) we have to add the entities that we'll serve the purpose of our System, that is Users, Reservations, Rides, Payments, User Reports. These will be entities that will be added by our System, specifically by the Controllers so this module has to be integrated first to answer the queries for the rest of the subsystems.

Server Tier: In the DD we present the High level Component structure. Based on the dependencies shown in this structure we'll select the order to test and deploy these components. The Helper Components de-

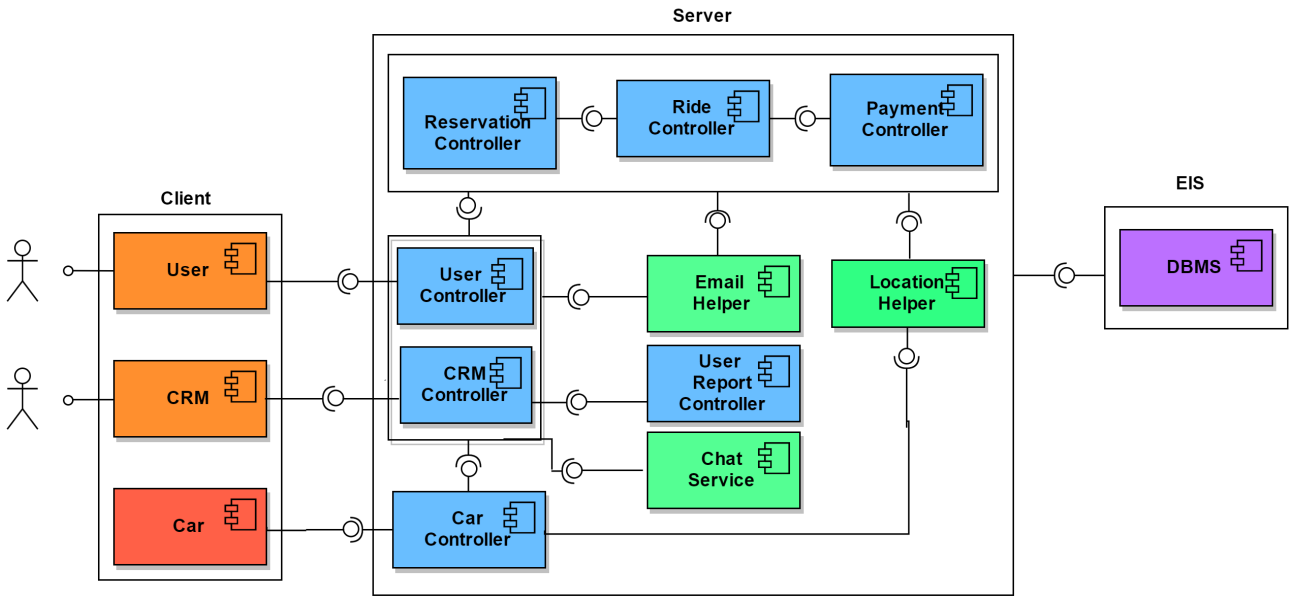

Figure 1: Component Structure

pend on no other components so according to the Bottom-up strategy they will be the first ones to integration. However not all Helpers are critical to other components so we'll prioritize the Critical Helper Components first, that is Location Helper as User, Car and Ride Controllers depend on it.
For each controller we have to implement Entity and Session Beans that will contain the methods and functions for managing the corresponding entites and logic so everyone of them will have Data Access Utilities to communicate with the DBMS. Their Integration order will then depend on the other controllers they depend on, meaning the other controllers they have to interface with.

An arrow suggest the Component on the right depends on the one on the left and will be integrated after all the previous one have been integrated.

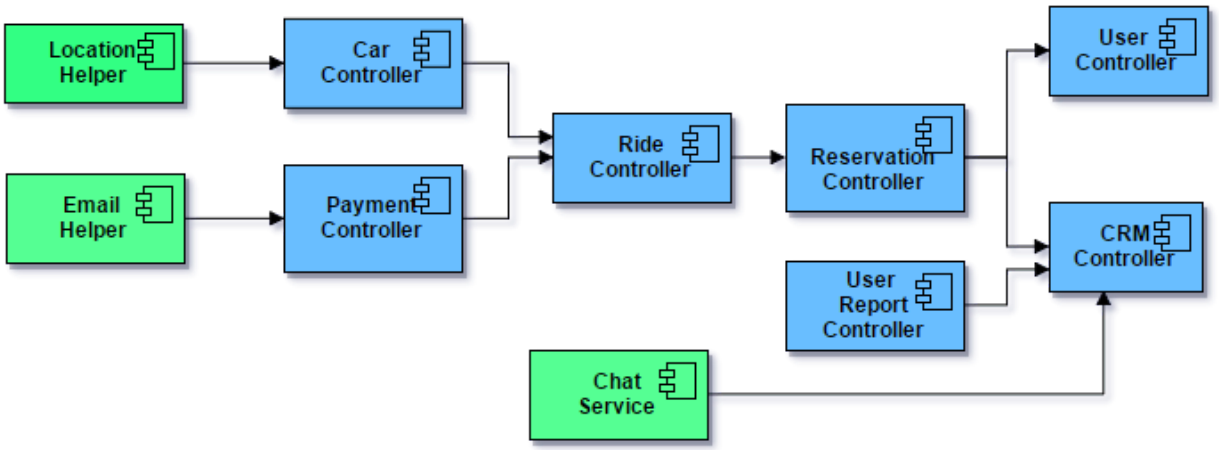

Figure 2: Server Components Integration

Client Tier: The Client Tier is based in two main components, User and CRM. Even if they are different they share many common functionalities including Login, Car Localization, Car Detail and Chat Service. They can be deployed simultaneously as they do not depend on each other and to test the logic subsystem from the Client side we can deploy one functionality at a time when all the dependencies have been met.

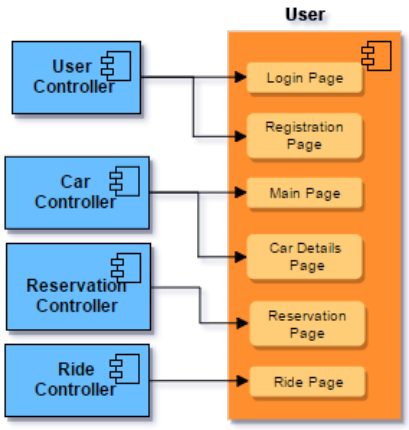

Figure 3: User Component Dependencies

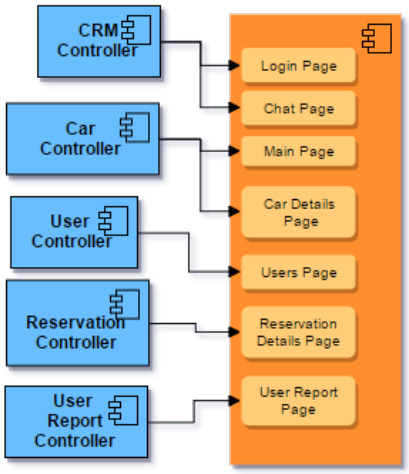

Figure 4: CRM Component Dependencies

2.4.2 Subsystem Integration Sequence

As mentioned before our subsystems corresponging to the Data, Logic and Presentation Tier will be integrated in that order. Before integrating a subsystem all the components inside it must be integrated as well.

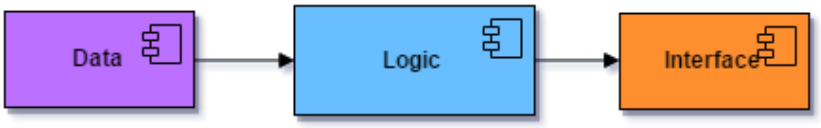

Figure 5: Subsystem Integration Sequence

3 Individual Steps and Test Description

This section will include a detailed description on the sequence and tests to be performed on each component to be integrated. For each component we'll present individual function testing to secure the correctness and robustness of the function with respect to unexpected or invalid input. For integration testing we'll test if each of the component involved works as expected or if not, where does the error occur to determine critical points in the integration. The EIS Tier is considered to be implemented for the individual function testing. To get an overall picture of the Component functions and interfaces look at the Section 2 of the **DD**.
Note: Null Input tests are considered for all applicable functions and these are considered to respond with a NullArgumentException unless expressed otherwise.

3.1 Location Helper

It uses Google Maps API which locations use 2 float parameters, as many functions use the class Location we remind that, as explained in the DD, the Class consist in two floats, lat (latitude) and lon (longitude). All functions that accept Location as parameter will also be implemented to accept 2 floats as parameters.

Function: isSafeParkingArea(Location) : bool

Input	*Response*
Invalid or Null Location	false
Location valid inside the Valid parking area. Check with DB.	true

Function: getCloseRechargingStations(Location) : List< *Location* >

Input	*Response*
Invalid or Null Location	Empty List
Location valid inside the Valid parking area.	Returns the list of available Recharging Stations closeby

Function: getBestRechargingStation(Location) : Location
//Uses getCloseRechargingStations

Input	*Response*
Invalid or Null Location	Null Location
Location valid inside the Valid parking area.	Returns the best available Recharging station found by the Standard Deviation Algorithm

Function: calculatePath(Location, Location) : List< *Location* >

Input	*Response*
Invalid or Null Location	Empty List
Locations valid inside the Valid parking area.	Returns faster path found by the Google Maps API, if path not found returns Empty List

3.2 Email Helper

Function: sendRegistrationEmail(Email, Password) : void

Input	*Response*
Invalid Email	Password Delivery failed Exception
Valid Email and Password	Registration Mail sent

Function: sendPaymentEmail(Email, Payment) : void

Input	*Response*
Invalid Email	Payment Delivery failed Exception
Valid Email and Payment	Payment Mail sent

3.3 Chat Service

Function: requestCRMContact() : Id

Input	*Response*
None	Returns a vaild CRM Id that is available for contact, else returns Null

3.4 Car Controller

Our system will not handle adding or deleting Cars in the DB so these operations will not be integrated. **Function:** reserveCar(CarId) : bool

Input	*Response*
Invalid or CarID not present	false - DB unchanged
CarID already reserved	false - DB unchanged
Valid CarID	true - Car marked as reserved in the DB

Function: enableCar(CarId) : bool

Input	*Response*
Invalid or CarID not present	false - DB unchanged
Valid CarID	true - Car changed to enabled in the DB

Function: disableCar(CarId) : bool

Input	*Response*
Invalid or CarID not present	false - DB unchanged
Valid CarID	true - Car changed to disabled in the DB

Function: getCloseCars(Location) : List$< CarId >$
Components: LocationHelper

Input	*Response*
Invalid or Null Location	Empty List
Locations valid	Returns List of Cars close in walking range to a given Location

4 Performance Analysis

Even tough the performance analysis is evaluated within the system as a whole, it was agreed that while testing the components, the isolated performance will be taken into account, as to correct unacceptable behavior, such as too slow response time, as soon as possible.
The aim of the performance analysis is to check the reliability of the application under normal usage conditions, providing benchmarks and identifying the response time, utilization and throughput of the application. So, for this test, the expected workload should be considered as in terms of the biggest city where the application will be implemented, taking into account an average usage for a long period with peaks of heavy traffic.
Both the server side and the client side can affect the performance of the application. For the client side, it is necessary to consider the performance of all the different interfaces. Particularly, the mobile application, the web application and the screen inside the car should all have satisfactory behavior, considering all kinds of users that can operate each of them.
Some important requirements to be evaluated:

- For the mobile application it is considered that the target public of the app can have any kind of Smartphone. So, the test will be made in low-range devices. This includes low ram availability, small internal space allocation and low processing power capacity.

- All the interfaces need to respond properly with slow network situations, and be reliable in situations of unstable network.

5 Required Tools and Test Equipment

5.1 Tools

In order to guarantee the most reliable system possible, when integrating components all the individual tests will be carried out once again, to make sure the integration did not cause new bugs to occur. For this reason, all the used programs provide tests automation tools, minimizing the rework.
The tools chosen for the Java EE and the EIS tiers were specifically three and each has its own scope of tests.

- First, the **JUnit Framework** will take care of the individual components testing. This is basically a way to certify that results produced by the components matches the theoretical value.
- Secondly, the **Arquillian Framework** helps keeping the integration testing simple. Testing the components of an application is challenging, so this framework focus on the interaction with the system, providing tools to check that the right components are being injected and the interactions with the database are occurring in a normal way.
- Finally,**Jmeter Framework** brings the application closer to the real world by performing load tests and performance tests. This tool provides emulations of loads on the server, network and objects providing solid data to analyze the overall performance of the systems under various conditions.

However, it is still necessary to test components withheld in the Client tier, specifically the mobile applications. In this matter, it will be used tools provided with each platform (IOS and Android have their own performance analysis tool that comes along with the SDK of the plataform). Also as stated in the previous session, particularly harsh situations as low battery, bad network coverage and low available memory need to be taken into account.
These tools ensure that every aspect of the application is tested in the proper way and provide all the necessary features to accomplish the proposed points of this document.

5.2 Test Equipment

In some previous sections, characteristics of the devices in which the application needs to operate successfully have been discussed. They were always pessimistic, assuming that low-range smarphones, with limited

functionalities were being used. However, this is only part of the testing devices, most users own better smarphones than the ones described. And even tough the older devices should be supported, the application must be better optimized for the majority of the public.
Therefore, for the Android application, for each available screen size in the market (including smarphones and tablets), it will be necessary to execute the tests for a low-range and a mid-range device.
When using a low-range device, the characteristics to keep in mind are a single core with slow processor clock (preferably less then 1 ghz), and less then 1gb of ram. As for mid-range, it can be a dual core with up to 2ghz of clock, and 2gb of ram.
In the IOS the specifications of the devices only vary within the different models, so at least one smarpthone and tablet of the IOS family need to be available for testing.
For the devices above described it will be carried out tests with the native mobile application and the mobile version of the web application. This is to ensure a broader number of devices and ways of accessing the application is covered.
Changing the scope from the mobile applications to the web browser client, a small number of notebooks and computers are also to be used, here the specifications of the devices are not important, but different browsers need to be installed in each of them.
The last category of devices that will be used are in respect to the screen inside the car. The tests will be carried out in android based devices, with different processing power and RAM availability. The results will be used to choose which model is to be deployed to all cars as to minimize the costs while keeping the necessary quality.
Finally, a identical server structure of the one used for the final deployment needs to be available. Although this is not yet completely defined, the test should recreate a realistic environment, using the same cloud infrastructure, Operating System, Java Enterprise Application server, and DBMS.

6 Required Program Stubs and Test Data

6.1 Program Stubs

6.2 Test Data

7 Effort Spent

Date	Domenico	Caio	Matheus
27/12/16	2h	2h	2h
28/12/16	-	-	-
29/12/16	1h	-	-
30/12/16	2h	-	-
31/12/16	-	-	-
01/01/17	-	-	-
02/01/17	2h	-	-
03/01/17	2h	-	-
04/01/17	2h	-	-
05/01/17	-	-	-
06/01/17	-	-	-
07/01/17	-	-	-
08/01/17	-	-	5h
09/01/17	-	-	4h
10/01/17	2h	-	-
11/01/17	-	-	-
12/01/17	-	-	-
13/01/17	-	-	-
14/01/17	-	-	-

8 Changelog

As the project and design decisions may change during the development this document is also prone to change. We'll document every version in this part.

- **Version 1.1:** 15/01/2017