Power EnJoy
Integration Test Plan Document

Niccolo' Raspa, Matteo Marinelli

January 12, 2017

Software Engineering 2 Course Project

Contents

1 Introduction

1.1 Revision History

Version	Date	Author(s)	Summary
1.0	06/01/2016	Niccolo' Raspa, Matteo Marinelli	Initial Release

1.2 Purpose and Scope

This document represents the Integration Testing Plan Document for PowerEnJoy. The main 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 up the system.

This process is essential because it not only guarantees that every component behaves as expected but also that all the components interoperate correctly together to fullfil all the functionalities expected from the system.

We will focus deeply on the Application Layer since it implements all the core of our business. This component is divided in different services which provides great benefits but the implicit granularity of a service oriented approach must be fully tested to guarantee that all the subcomponents behave as one cohesive layer.

This document is structured as follows:

Chapter 1 Provides general information about the ITPD document.

Chapter 2 Explains in details the chosen integration strategy. In more details:

- Lists of the subsystems and their subcomponents involved in this process
- Specifies the criteria that must be met before integration testing begins
- Describes the integration testing approach and the rationale behind it
- Outlines the order in which components and subcomponents will be integrated

Chapter 3 Describes the type of tests that will be used to verify that every step of the integration process above perform as expected.

Chapter 4 Identifies all tools and test equipment needed to accomplish the integration.

Chapter 5 Identifies any program stubs or special test data required for each integration step

1.3 List of Definitions and Abbreviations

DD: Design Document

RASD: Requirement Analysis and Specification Document

ITPD: Integration Test Plan Document

EJB: Enterprise JavaBeans

SOA: Service Oriented Architecture

Component: One of the four tier of the system (Client, Web, Application, Database)

Subcomponent: Usually refers to the Application Layer, and refers to a EJB that encapsulates a specific part of the business logic of the module

Layer: synonim of *Component*

Service: synonim of *Subcomponent*

Power User: Registered user of the application

External System: Refers to third party systems used in the Power EnJoy application (Payment System, Assistance System, Car-On-Board System)

1.4 List of Reference Documents

Please refer to the following documents, for additional informations on the Power Enjoy System:

- Project rules of the Software Engineering 2 project
- Power Enjoy - Requirement Analysis and Specification Document
- Power Enjoy - Design Document

2 Integration Strategy

2.1 Entry Criteria

This section describes the prerequisites that need to be met before integration testing can be started.

Stakeholder Approval First of all, the Requirements Analysis and Specification Document and the Design Document must have been presented to the stakeholders for approval even before the coding phase can begin, this will ensure that they're satisfied with the development.

Website and Mobile App The presentation layer to the user might not be completed but communication between the Application Server and Clients, both via the Mobile App and via the Web Server, must have clearly structured and coded via RESTful APIs using JAX-RS.

Coding and Testing Application Layer

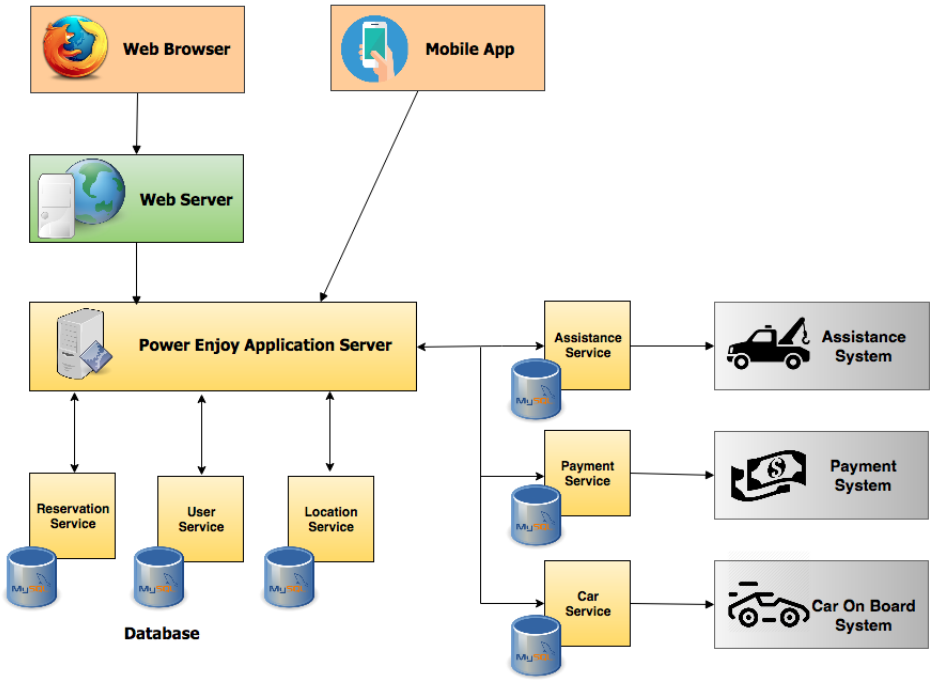
- All the classes and methods of the main Entity Beans must be coded and must pass thorough **Unit tests**. Unit tests should have a minimum coverage of 90% of the lines of code and should be run automatically at each build using JUnit. (non necessariamente tutte, ma quelle piu importanti prima ed eventualmente simulare le altre durante il testing)
- **Code inspection** has to be performed on all the code in order to ensure maintainability, respect of conventions and find possible issues which could increase the testers' effort in next testing phases. Code inspection should be performed using automated tools when possible.
- **Documentation** of all classes and functions should be well written using JavaDoc. The public interfaces of each Entity Bean shoud be clearly stated.
- **Object relational mapping** of each service to its corresponding database should be implemented with automated tools to avoid errors (such as Hibernate) and fully tested.

2.2 Elements to be Integrated

In this section we're going to provide a list of all the components that need to be integrated together. The figure below corrispond to the system architecture already discussed in the Design Document.

The system is built upon the interactions of many tiers, each one implementing a specific set of functionalities. Every tier is also obtained by the combination

of several lower-level components. This modularity causes that the integration phase will involve the integration of components at different levels of abstraction. In addition our system is in relation with External Systems, which are crucial for our application to work. It's important that they're correctly integrated to the system in a way in which everything is trasparent to the user.



In summary, the elements to integrate are:

1. Integration of the different services inside the Application Layer.
2. Integration of different tiers (Client - Web - Application)
3. Integration and configuration with third party systems (Payment System, Assistance System)

2.3 Integration Testing Strategy

The approach we're going to use to perform integration testing is based on a mixture of the bottom-up and functional-grouping integration strategies. This choice is due to the fact that if the entry criteria is met, it's reasonable to assume that we have different small services, indipendent from one other, that implement correctly a small part of the application logic and by integrating them we're able to create a more and more complex system that will eventually satisfy all the requirements. In a pure bottom-up strategy we would start from the lowest layers of the system, testing the basic functionalities, then moving forward the most abstract layers but in our case this approach would be unefficient. Since every service is dedicated to one part of the business logic we can parallelize this testing, focusing on different logic groups at the same time, giving more priority to the critical components first and then integrating secondary functionalities. Moreover, we also need to keep in mind that we're dealing with external systems and if we discover some bugs or problems on their side, fixes might take time and this would create a time gap in which we're not able to move the integration forward.

For all these reasons we believe that the best integration strategy is the following:

PHASE 1: Assure that services in relations with external systems works as expected.

As stated earlier, in order to avoid wasting time we start from the boundary of the system. This process should be fairly quick if everything was implemented as mutually agreed and should immediatly discover issues that we can notify early on to external parties. In this phase we'll test the communication between *Assistance Service - Assistance System* and *Payment Service - Payment System.*

PHASE 2: Assure that we have control over the Car

This phase is similar to the previous, and can be carried in parallel. In this phase we're also in relation with an external system which is the *Car On Board System* but due to the relevance of this process we've decided to outline it and dedicated a whole phase. We can't move the integration forward if we're lacking the foundations. In a digital management system for a car-sharing service the control over the car must be treated as a first class citizen. This will avoid a big bang scenario, where we have implemented high level functionalities that not reflect the concrete situation of the car in the real world. These two inital phases will also allow us to "forget" of external systems in the next phases, and only focus on the relation among different services.

PHASE 3: Integration of Services

In this part the bottom-up approach would be used to build complex functionalities integrating different services. Since subsystems are fairly independent from one another, the order in which they're integrated together to obtain the full system follows the critical-module-first approach. This strategy allows us to concentrate our testing efforts on the riskiest components first that represent the core functionalities of the whole system. By proceeding this way, we are able to discover bugs earlier in the integration progress and take the necessary measures to correct them on time.

The most critical service to integrate is the *Reservation Component* which manages all active reservation made by Power Users, we will ensure that it integrates correctly both with the *Location Component* and the *Car Service*. This will ensure that we have a stable prototype of the actual software that implements the core functionalities. From this prototype we will spread like wildfire, integrating other Components that implements all the other functionalities.

In this phase, it is only necessary to use drivers to simulate the top layers during the testing, which are a lot easier to produce than stubs.

PHASE 4: Integration with top layers

In this phase we will remove the drivers and connect the Application Side to the top tiers. We must ensure that the Application Layer works with real inputs from the "external word" and not only in a simulated and controlled environment. The integration should proceed smoothly since the communication via RESTful APIs was clearly structured at the beginning of the integration but we should focus deeply on errors and expection handling.

PHASE 5: Alpha Test

In this phase we'll test the Power Enjoy System as a whole. This phase will provide a confirmation of the correctness of the integration process and will ensure that we haven't overlooked possible error scenarios.

2.4 Sequence of Component/Function Integration

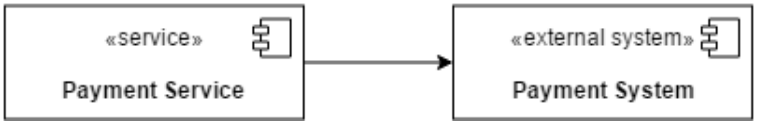
2.4.1 Software Integration Sequence

In this section we're going to formalize the order of integration (and integration testing) of the various components and subsystems presented in the previous section. Before describing this process, we will clarify the notation used. In the figures we will refer to an Entity Java Bean contained in the application layer as a *<<service>>* , we will use *<<component>>* to refer to a layer (subsystem) of the Power EnJoy Application and *<<external system>>*to refer to

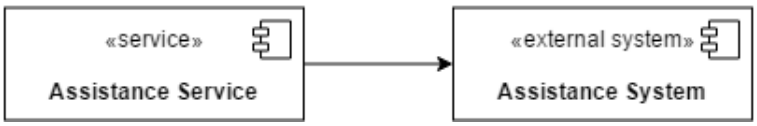
External Systems (Payment System, Assistance System, Car-on-Board System). Elements that will be simulated during the integration process will be placed inside a dashed line box.

External Systems

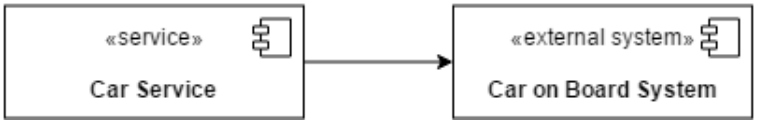
Payment The first two elements to be integrated are the *Payment Service* and the *Payment System*. As explained above, it is important to start from the external systems in order to signal any problems that involve third party software or APIs. In this step we test if we're able to check if a user has a registered account in the third payment system, regular payment functionalities (sending a payment request, receiving payment confirmation) and the correct management of payment logs.



Assistance The second step in the integration process is to integrate the other external system, the *Assistance System*, for the same reasons mentioned above. The latter has less priority because it's reasonable to assume that we have more control over the Assistance System, since it's part of the company. In this step we test if we're able to request assistance and receive the confirmation when services are rendered.



Car On Board System The last system to integrate is the Car on Board System. In this step will make sure that the system itself works (correctly detects car events such as door opening and closing, engine starting, ecc...) but also that we're able to detect this events from inside our software and controlling remotly the vehicle. In this way, all the other services will only use the interface provided by the *Car Service* forgetting about the actual Car on Board System.



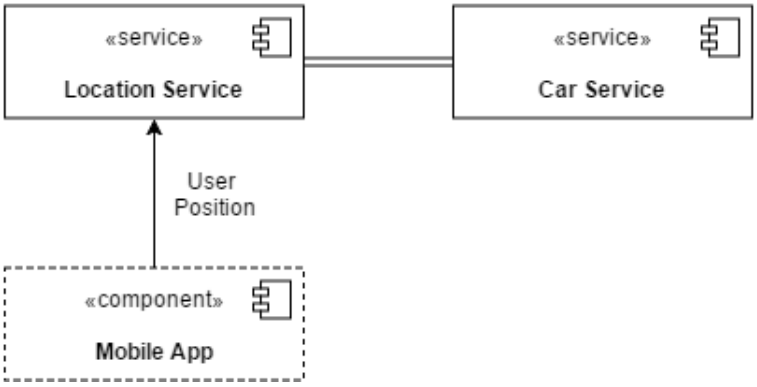
Reservation

After making sure the boundaries of our system work correctly, the next step in the integration process is to appropriately connect the services that implement the reservation functionality. This choice comes from the critical-module-first approach, because in a car sharing service being able to reserve a car is the most important functionality. As said before, it was not implemented first to avoid a big bang scenario, where we have implemented high level functionalities that not reflect the concrete situation of the car in the real world.

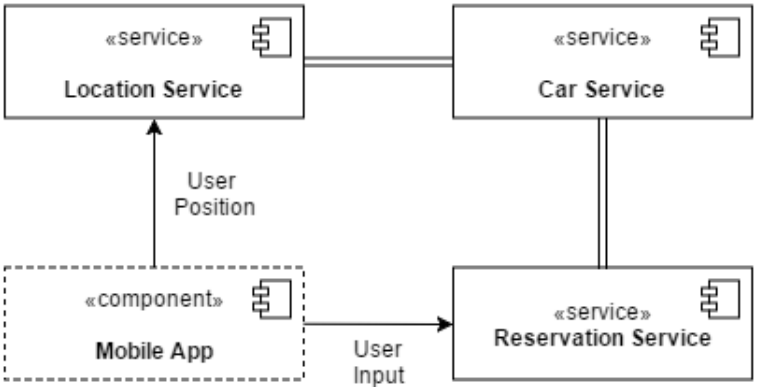
Using a bottom-up approach, we are going to show which services must be integrated together in order to implement this functionality.

Location of Cars First, we proceed by integrating the *Car Service* with the *Location Service*. We test if the location is correctly detected, if we're able to monitor the car during a ride and if we're able to correctly detect a safe or an unsafe park.

Location of Power Users Second, we integrate the location of user by simulating reservation request from a user via Mobile App. As a remainder, our system is mobile-first and the reservation of a vehicle is only possibile via a mobile device, this allows us to reduce the scope of the testing and forget about the web services which we'll be integrated later. In this step we test the "Select Car" use case (please refer to the RASD for more details) in which the user selects a car on the map given an address or a (simulated) user position and a range, the system performs the GridSearch Algorithm (please refer to DD for more details on the implementation) and returns the list of available cars.



Start/Cancel Reservation Finally, we have built all the necessary functionalities and we're able to test the registration functionality. We integrate the *Reservation Service* with the *Location Service* and the *Car Service*. In this step we test if a Power User can successfully reserve a car, unreserve a car (without the 1hour limit for now) and can request to unlock the car (with proximity check). The user input is still simulated.



Driving and Ending Reservation To complete the reservation functionality we don't need to integrate any but we need to make sure that the prototype of the system performs all the actions required to stop the reservation. In this step we test that the engine ignition triggers an active reservation and that a reservation immediately stops as soon as a car is parked and locked (both in a Safe/Unsafe Area for now).

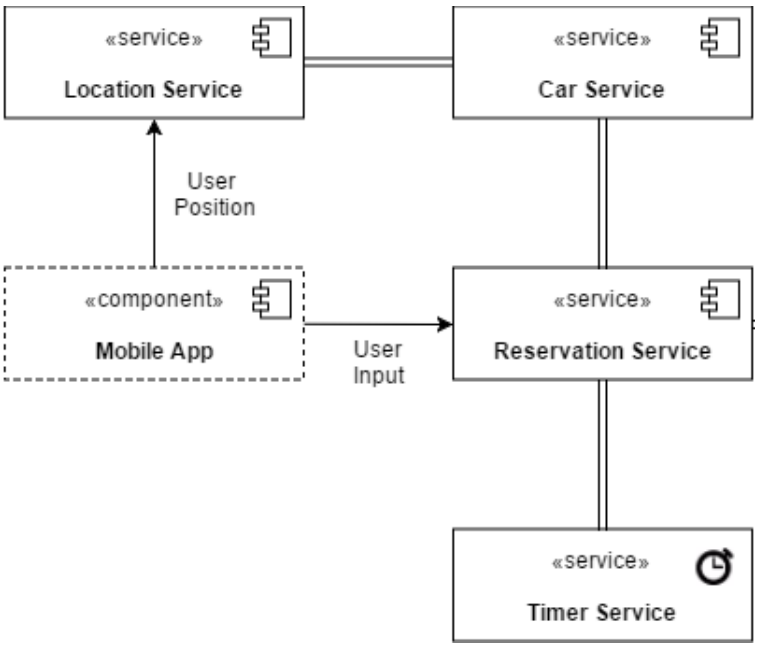
Other functionalities

Now that we have a stable prototype of the Power Enjoy Application we will spread like wildfire, integrating other services that implements all the other functionalities.

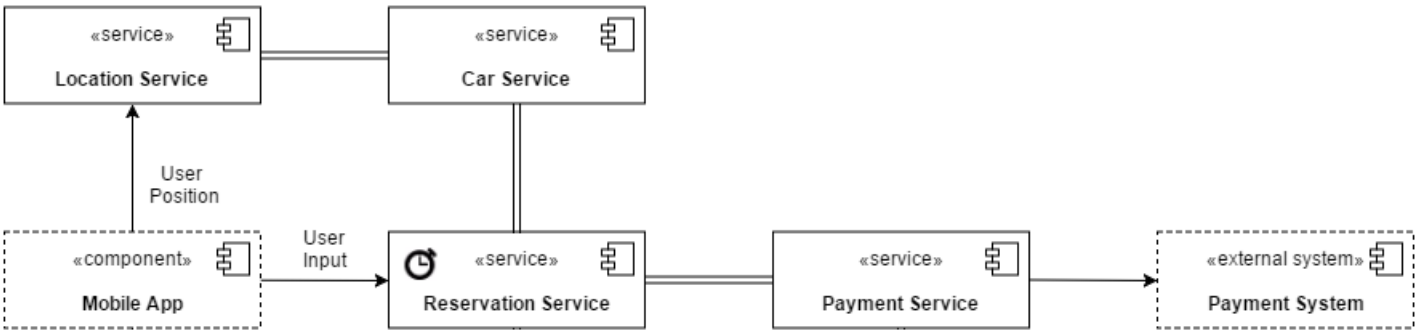
Timing Our prototype doesn't consider the timing constraint that the reservation imposes. Moreover, we need to incorporate the following constrains:

1. A Power User has 1 hour to request to unlock a vehicle once he/she has reserved it
2. A reserved car parked in an Unsafe Area, triggers a one hour countdown in which the user needs to go back to the car to move it on a Safe Area. As soon as the countdown expires the reservation expires as well. The car is set unavailable and the Power User is charged for the extra time as a refund for moving the car back in a Safe Area.

In this step we integrate these functionalities, adding the *Timing Service* to the prototype. In later figures this component we will be hidden for clarity, but a timer icon will be added to the Reservation Service as a remainder. We will test that countdown expiration causes the reservation to end and that a user requesting to unlock a vehicle stops the timer (both for case 1 and case 2)

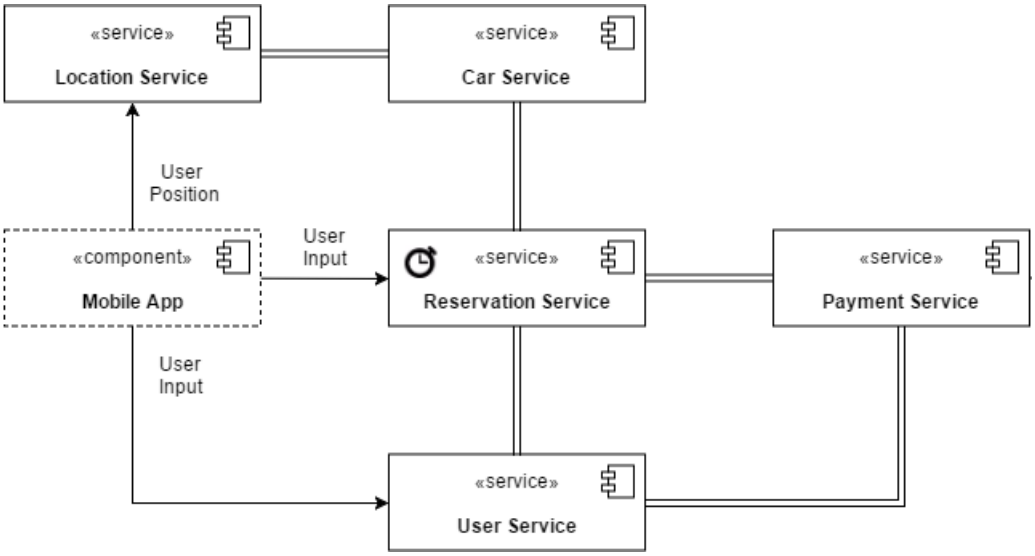


Payment In this step we start moving towards the boundaries of our system. We include the payment functionalities adding the *Payment Service* to the prototype. We test that the end of a Reservation (in any case) triggers a payment request.



User & Profile Management So far in the integration progress we have assumed that a simulated User was already registered to the Power Enjoy System, logged and always allowed to reserve a car. Now it's time to introduce this additional checks and functionalities integrating the *User Service*. The new features to test involve:

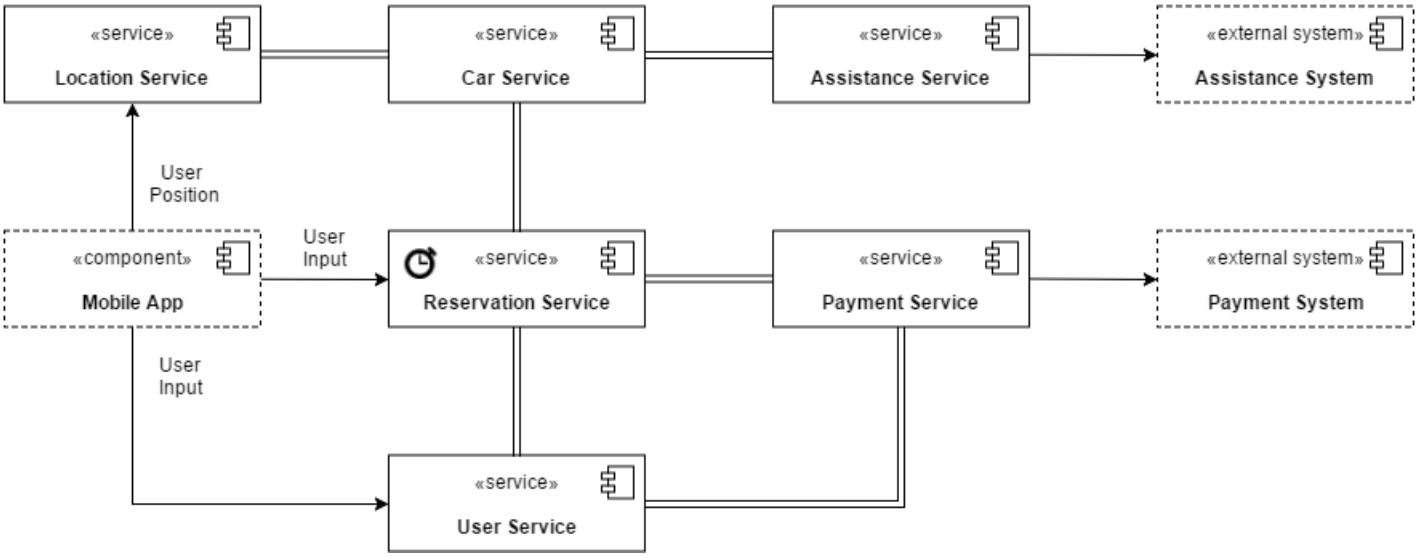
1. Login and Registering
2. Temporary Ban a Power User if it has a payment pending
3. Unban as soon as a payment request is recieved
4. Editing Profile Info
5. Request Logs (Reservations, Payments)



Assistance In this step we integrate the last componen of the Business tier, the *Assistance Service* . We test that a malfunctioning event detected from the *Car Service* triggers and assistance request and the termination of the reservation.

2.4.2 Subsystem Integration Sequence

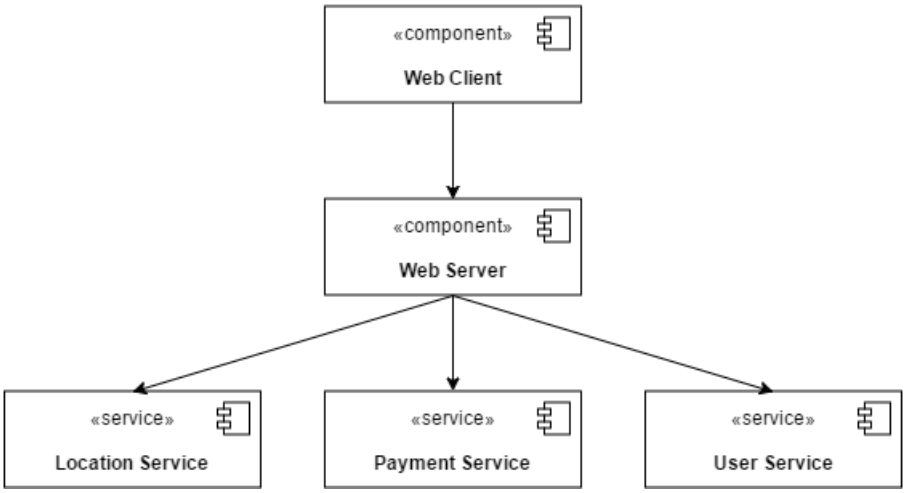
Our prototype is working with the top layers simulated. Now it's ready to integrate the real tiers into the system. We can split the integration between



the two parts: the web part and the mobile part. We first integrate the Web Part because it uses all parts of the prototype already tested for the mobile and then the mobile should be straightforward if the simulated mobile app was implemented mirroring the real app.

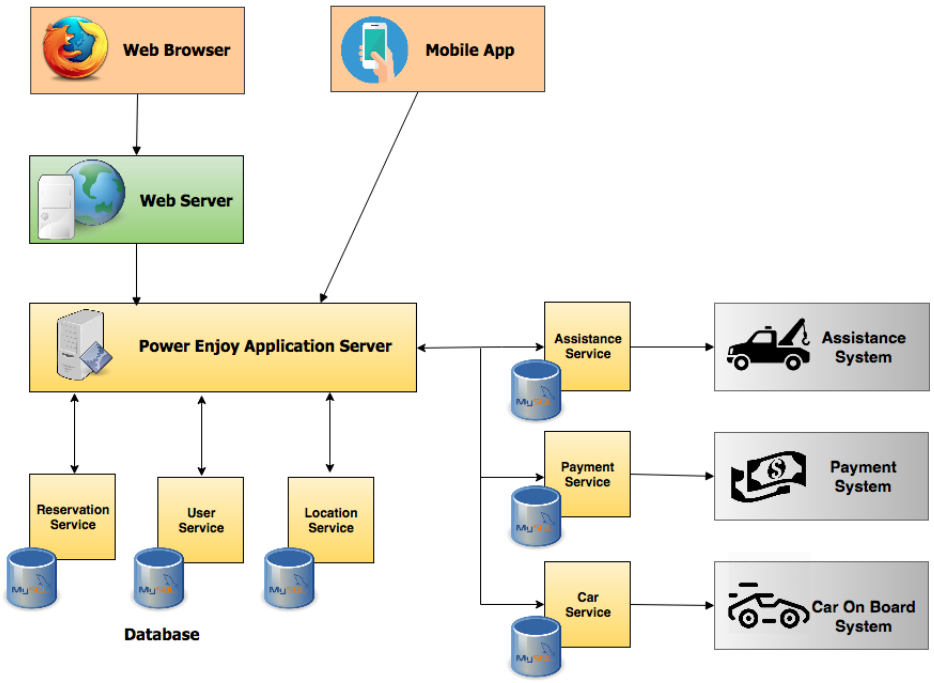
Web Subsystem

In this step we need to make sure that all the functionalities expected from the Website (Profile Management, Car Lookup and Payments) works accordingly.



Complete System

Integrating the mobile application we're ready to deploy and test the entire system. In this phase we should perform an alpha test and check that a client can successfully make a reservation from start to finish. Different scenario might occur in a real non-simulated environment so it's important to make sure that we haven't overlooked any plausible situation that might cause unexpected behaviour of the system.



3 Individual Steps and Test Description

This chapter describes the individual test cases to be executed. Test cases are identified by a code, test cases whose code starts with SI are integration tests between subsystems; test cases whose code starts with I are integration tests between components.

The Database integration test and all related tests are not present because our application is composed of microservices, each one with its own DBMS, which is strictly related to the functionality expected from that component . The tests of the single databases and of the object relational mapping were performed during the individual componenet testing stage (which is not part of this document).

3.1 I1

Test Items	Payment Service –> Payment System
Input Specification	Messages who identify the user (existing in the Payment System Database) to charge. Informations about the amount to pay. Tipical values could be existing/unexisting users, random amount to pay, also over the user's disponibility.
Output Specification	User to charge is find or not. Payment has happend or not.
Enviromental Needs	Payment Service Driver and Payment System Mock
Test Description	A invoice is sended to Payment Service. It will contact the Payment System in order to charge the user. Payment System will respond with the result (positive or negative) of the charging operaction. The test must check the behaviour of the Paiment System according to the informations received. Crucial points are the existance of the user, the possibility to pay and the return of the result. Is also to test the correct ban of the user with a payment pending and the relative unban when a payment notification arrive from the Payment System
Testing Method	Automated with JUnit and Mockito

3.2 I2

Test Items	Assistance Service –> Assistance System
Input Specification	Information about malfuncioning cars and malfunctioning type
Output Specification	Notification of completed assistance
Enviromental Needs	Assistance Service Driver and Assistance System Mock
Test Description	Is simulated a malfunciotioning car status. The test wants to check the correct exchange of messages between Assistance Service and Assistance System. Assistance Service must also notify the solution of malfunctioning to the Car Service.
Testing Method	Automated with JUnit and Mockito

3.3 I3

Test Items	Car Service –> Car on Board System
Input Specification	All car status change provided by the Application Server
Output Specification	All car status provided by the car interacting with real world (user). Cmmunication of information to produce an invoice and the position.
Enviromental Needs	Car Service Driver and Car on Board System Mock
Test Description	A complete Reservation is simulated. The test must ceck that the interaction between Car Service and Car on Board System is correct. The Remote Control must correctly lock and unlock the car when requested. When an ignition is simulated, it will be detected by the Car on Board System and notified to Car service; the same for malfunctioning and Charging state. If is simulated the presence of passengers, the Car on Board System must correctly calculate the Shared Ride time and send it to Car Service. The Car on Board System must also be able to calculate the Ride Length when the reservation simulation ends and send it to Car Service.
Testing Method	Automated with JUnit and Mockito

3.4 I4

Test Items	Location Service –> Car Service
Input Specification	Position of the cars. Safe/Unsafe Park.
Output Specification	Position Updates
Enviromental Needs	Location Service Driver and Car Service Driver
Test Description	Test must check that Location Service can give an updated postition to all cars and can register all update communicated by Car Service. The location grind must be tested as well, so a car selection must be simulated in order to check if the location grid correctly shows the most convnient available cars.
Testing Method	Automated with JUnit

3.5 I5

Test Items	Mobile App –> Location Service, Car Service
Input Specification	Request of selection of a car. User position (GPS) or an address
Output Specification	List of suggested available cars, if no cars are available the output is a notification.
Enviromental Needs	Mobile Application mock and I4
Test Description	A request of a car is performed on the app. The test must check that a the output is correct according to the input and the cars disposition
Testing Method	Automated with JUnit and Mockito

3.6 I6

Test Items	Reservation Service –> Mobile App, Location Service, Car Service
Input Specification	Selection of a car (start of the reservation). Reservation canceled. Unlock request. Car Ignition notification. Position notification at the end of the reservation.
Output Specification	Unlock successfull or denied. Start ride notification. Ending sequence.
Enviromental Needs	Reservation Service Driver and I5
Test Description	Is simulated a Reservation from the very beginning. Test must perform the reservation starting with the car selection and proceed with every step till the end (payment excluded). All possible situations must be simulated in order to verify the correct behaviour and return of the system. Remember to verify the reservation cancellation (not the expiration, it will be tested in the next test), park in a non safe area is not considered.
Testing Method	Automated with JUnit and Mockito

3.7 I7

Test Items	Payment Service, Timer Service –> Mobile App, Location Service, Car Service, Reservation Service
Input Specification	Timer Service: Selection of a car (start of the reservation). Reservation canceled. Car Ignition notification Payment Service: Invoice.
Output Specification	Timer Service: Time to expiration. Payment Service: Ban/Unban user with pending payment.
Enviromental Needs	Car Service Driver Timer Driver, I6 and I1.
Test Description	A reservation is simulated. In this test is important to check the correct management of the expiration time by the Timer Service in case of waiting of an ignition and in case of park in a non safe area. The System must also corretly send the invoices when the time expire or the reservation end.
Testing Method	Automated with JUnit and Mockito

3.8 I8

Test Items	Payment Service, User Service, Location Service
Input Specification	Informations about the user, ban/unban updates, Location Updates.
Output Specification	Result of profile modification.
Enviromental Needs	Payment Service Driver, User Service Driver, Location Driver and Web Application and Server mock
Test Description	In this test must be performed the profile managemet. Check the behaviour of the system inserting both possible and unpossible values. Both Login and Profile Modification must be tested in this section.
Testing Method	Automated with JUnit and Mockito

3.9 I9

Test Items	Web Application, Web Server –>Payment Service, User Service, Location Service
Input Specification	Informations about the user, ban/unban updates, Location Updates.
Output Specification	Result of profile modification.
Enviromental Needs	SI1 and Web Application and Server completed
Test Description	In this test must be performed the profile managemet. Check the behaviour of the system inserting both possible and unpossible values. Both Login and Profile Modification must be tested in this section. This time no components are simulated. Since the Web Application is not simulated, must be tested also the web interface and its correct behaviour.
Testing Method	Automated with JUnit and manual testing (all combination of OS-Browser)

3.10 I10

Test Items	All Services (Top Layer Simulated)
Input Specification	All methods invoked by a simulated mobile application
Output Specification	Visual notification to inform the user about any usefull state or modification in the system.
Enviromental Needs	I1, I2, I3, I7 and Mobile Application Mock
Test Description	Complete simulation of all functionalities of the Application. Perform test on every kind of users, registered or not, banned or unbanned.
Testing Method	Automated with JUnit and Mockito

3.11 I11

Test Items	All Services (no simulated layers)
Input Specification	All methods invoked by a simulated mobile application
Output Specification	Visual notification to inform the user about any usefull state or modification in the system.
Enviromental Needs	SI3 and Mobile Application
Test Description	Complete simulation of all functionalities of the Application. Perform test on every kind of users, registered or not, banned or unbanned. Since no layer are simulated, chec the correc behaviour of the application on every device (IOs and Android), check that the visual presentation is also correct.
Testing Method	Automated with JUnit and manual testing (IOs and Android)

4 Tools and Test Equipment Required

Manual Testing On Board Computer: The most frequent interaction is between the Application server and the On Board Computer in cars. Is fundamental to simulate reservations in order to control the absence of malfunctioning or errors in the interaction. The test will cover all the possible situations that may happen during a rent: reservation, cancel reservation, start the ride, end the ride, park in a non safe area, ecc...

JUnit Before integration testing, tests on single components is necessary. This first part of testing is not covered in this document. Anyway, the main issue is to check the absence of bugs and problem in each part of the system, from the application to the all the Application Server components. We choose JUnit to perform the components test basically because is the most used framework for this specific; totally java dedicated, it combines perfectly itself with the implementation choices described in the Design Document and with the integration testing framework Arquillian. In particular, we are going to use it in order to verify that the correct objects are returned after a method invocation, that appropriate exceptions are raised when invalid parameters are passed to a method and other issues that may arise when components interact with each other.

Mockito Mockito is an open-source test framework useful to generate mock objects, stubs and drivers. Since the entire system interacts with external and real objects, is necessary to use a framework to reproduce this kind of entities. In unit testing, mock objects can simulate the behavior of complex, real objects: they are useful when a real object is impractical or impossible to incorporate into a unit test. They are also useful for the

developers, who have to focus their tests on the behavior of the system without worrying about its dependencies and having predictable results.

Arquillian Arquillian is an integration testing framework for business objects that are executed inside a container or that interact with the container as a client. Our choice falls on it because is widely used, in particular makes simple the kind of testing we need, since we have different components grouped inside one big application server. Arquillian also integrates perfectly with JUnit, used in the single components testing phase. It combines a unit testing framework (JUnit), and one or more sup- ported target containers (Java EE container, etc) to provide a simple, flexible and pluggable integration testing environment. Arquillian makes integration testing no more difficult than the beans testing. Specifically, we are going to use Arquillian to verify that the right components are injected when dependency injection is specified, that the connections with the database are properly managed and similar containerlevel tests.

Devices The application run on two types of operative systems (Android and IOs). This fact make necessary testing on the direct tools that allow users to exploit power enjoy system. Power Enjoy service was tested on two groups of mobile devices (non necessarly phones), one for each type of operative system. Web page is available on the web and a group of computers were used to test the page. Test were made on every combination of Operative system and browser.

5 Program Stubs and Test Data Required

In order to perform integration testing without having developed the entire system first, we need to use stubs and drivers to simulate the software components that still donít exist and test the others.

5.1 Drivers

Data Access Driver This testing module will invoke methods in order to test interaction between DBMSs and Application Server's beans. Since the Power Enjoy is structured in microservices, bin has access at most to on specific DBMS. This driver manages all the interaction of this kind, for all bins.

Reservation Service Driver this testing module will invoke the methods visible to the Reservation Service subcomponent, in order to test its interaction with the Reservation database, the User service, the Car Service and the Payment Service components.

User Service Driver this testing module will invoke the methods visible to the User Service subcomponent, in order to test its interaction with the User database, the Reservation service, the Location Service and the Payment Service components.

Car Service Driver this testing module will invoke the methods visible to the Car Service subcomponent, in order to test its interaction with the Car database, the Reservation service, the Location Service and the Assistaance Service components.

Location Service Driver this testing module will invoke the methods visible to the Location Service subcomponent, in order to test its interaction with the Location database, the User service and the Car Service.

Assistance Service Driver this testing module will invoke the methods visible to the Assistance Service subcomponent, in order to test its interaction with the Car Service component.

Payment Service Driver this testing module will invoke the methods visible to the Payment Service subcomponent, in order to test its interaction with the Payment database, the Reservation service and the User Service components.

5.2 Stubs

Since we used a mixed approach, not purely bottom-up, stubs are necessary to emulate the presence of not yet completed components. A test DBMS must be filled with random values in order to test properly the Software. Each entity must appear in the test DBMS following the same ER diagram designed in the Design Document. Could be usefull to insert values may cause exceptions, to check the behaviour of the system on limit situations. Other stubs must reproduce the behaviour of external systems that interact with the Power Enjoy software. Must be emulated the communication between Payment System, Assistance System and Car On Board System with the respective bins. The test must evaluate also the efficiency of the user actions. Will be usefull a Mock sender and receiver in order to collect all data of an immaginary user. A set of datas choosen ad hoc (also not real ones) is necessary to verify the validity of the credential.

6 Effort Spent

The approximate number of hours of work for each member of the group is the following:

Niccolo' Raspa 5 Hours

Matteo Marinelli y Hours