Integration Test Plan Document

Authors:
Patrizia Porati, Tommaso Sardelli

POLITECNICO
MILANO 1863

Contents

1 Introduction

1.1 Revision history

This is version 1.0 of our Integration Test Plan Document. In this paragraph we will list changes of each future version of the document.

Version	Date	Authors	Summary
1.0	15/01/217	Patrizia Porati, Tommaso Sardelli	First version

1.2 Purpose and Scope

This is the Integration Test Plan Document (ITPD). This document aims at describing the plan to accomplish the integration testing of the components that compose the PowerEnJoy application. Integration testing is the intermediate task between the unit testing, that tests sections of code and individual modules, and the system testing, that test the complete system. The purpose of this level of testing is to expose faults in the interaction between integrated units.

1.3 List of Definitions and Abbreviations

The following are used throughout the document:

RASD Requirement Analysis and Specification Document

DD Design Document

ITPD Integration Test Plan Document

Unit testing a software testing method used to test individual units of source code, single modules, functions and procedures

1.4 List of Reference Documents

- The project descritption document: Assignments AA 2016-2017.pdf
- PowerEnJoy Requirement Analysis and Specification Document: rasd.pdf
- PowerEnJoy Design Document: dd.pdf

2 Integration strategy

2.1 Entry criteria

There are some conditions on the project development that must be met before starting the integration testing.

- *Requirements Analysis and Specification Document* and the *Design Document* must have been written, in order to have a complete picture of the interactions between the different components of the system and their functionalities.
- All the functions and individual modules have already been tested through a unit testing process and all unit test bugs have been fixed.

2.2 Elements to be Integrated

UserHandler
UserHandler → Database
UserHandler → Map API
UserHandler → PaymentHandler
Router → UserHandler

CarHandler
CarHandler → Database
CarHandler → Map API
Router → CarHandler

PaymentHandler
PaymentHandler → Database
PaymentHandler → Payment API
Router → PaymentHandler

ReservationHandler
ReservationHandler → Database
ReservationHandler → Payment API
Router → ReservationHandler

RideHandlerHandler
RideHandler → Database
RideHandler → Payment API
RideHandler → CarHandler
Router → RideHandler

2.3 Integration Testing Strategy

The strategy that will be adopted is the so-called **bottom-up** approach. This approach comes natural with the structure of our components. They have been designed with the clear scope in mind to make them atomic and as decoupled as possible. We can take advantage of these design choices now and start our integration tests on the single component. Once we are confident that the component behaves well within its subcomponents and with the external services, we can proceed testing it's integration with another component and so on. As a short example we can consider one of the simplest use cases, the creation of a new user. The full sequence of actions that should be performed in order tu fulfill the request would be something like this:

```
HTTP Request
    |
    v
    API Gateway
        |
        v
        Dispatch Message
                 |
                 v
                 User Handler
                      |
                      v
                      Validate
                          |
                          v
                          Insert in the DB.
```

The whole operation can be seen as a sort of pipeline where each subcomponents receive the input from the previous one and passes its output to the next. Following the bottom-up approach we can start testing the integration between the database and the code whose responsibility is to interact with it. Once we are sure that everything is working as expected we move up one step and test the previous integration. In the end we should be able to test the whole "pipeline" and make sure that the goal of creating a user is achieved.

These simple test operations will be iterated over every single component and then over groups of components interacting with each other.

2.4 Sequence of Component/Function Integration

Thanks to the fact that our architecture is loosely coupled, we have a certain flexibility choosing the order in which tests should be performed. We start testing the integration of subcomponents in a depth-first manner and move upward in the stack until we find a dependency with another component. Alternatively following what we could call a breadth-first approach, we can

decide to complete the integration tests of every component and move up in the stack only when all of them are done. We will prefer the first choice because we think it provides a higher degree of flexibility. During the development process we can start testing and prototype some functionalities even if we have not yet developed the other components.

3 Individual Steps and Test Desription

3.1 Test Cases

In this section we are going to provide a description of the tests on each pair of components/subcomponents that have to be integrated. Each subsection contains tests concerning a specific component: there are tests on the relations between its subcomponents and finally the tests on the relation between the components itself and the RESTful API component.

3.1.1 UserHandler

Test Case Identifier	I1T1
Test Items	(UserHandler) SearchFilters → Database
Input Specification	Typical inputs for getting information from database: get user list, get user details, get user position
Output Specification	Check if requested information is returned correctly
Environmental Needs	PersistenceLibrary

Test Case Identifier	I1T2
Test Items	(UserHandler) CUDCommands → Database
Input Specification	Inputs for inserting, modifying and deleting database information: add administrator (email already existent, valid input), add user (email already existent, valid input), delete user, modify user information, modify user status, set user position
Output Specification	Check if information is correctly update in the database
Environmental Needs	PersistenceLibrary

Test Case Identifier	I1T3
Test Items	(UserHandler) CUDCommands → MapAdapter
Input Specification	Typical inputs for getting Map Informations: get user position, get distance
Output Specification	Check if requested information is returned correctly
Environmental Needs	

Test Case Identifier	I1T4
Test Items	RESTful API → UserHandler
Input Specification	Add an administrator (invalid email, invalid credit card number, invalid driving licence, null input(s), valid inputs),get administrators list, add a user (invalid email, invalid credit card number, invalid driving licence, null input(s), valid inputs), delete a user, update user information (valid information, invalid information), change user status, get users list, get user information, get user details, get user position, set user position
Output Specification	Check if correct functions are called in UserHandler
Environmental Needs	

3.1.2 PaymentHandler

Test Case Identifier	I2T1
Test Items	(PaymentHandler) CUDCommands → Database
Input Specification	Inputs for interting, modifying and deleting database information: update payment information (valid information, invalid information), insert payment (valid input, input already existent), change user status to 'insolvent'
Output Specification	Check if information is correctly update in the database
Environmental Needs	PersistenceLibrary

Test Case Identifier	I2T2
Test Items	(PaymentHandler) SearchFilters → Database
Input Specification	Typical inputs for getting information from database: get user payment information
Output Specification	Check if requested information is returned correctly
Environmental Needs	PersistenceLibrary

Test Case Identifier	I2T3
Test Items	CUDCommands → PaymentAdapter
Input Specification	Valid inputs, invalid credit card number, tpayment not successful (not enough money to spend)
Output Specification	Check if correct functions are called in PaymentAdapter
Environmental Needs	

Test Case Identifier	I2T4
Test Items	RESTfull API → PaymentHandler
Input Specification	Update payment information (valid inputs, invalid credit card number, payment not successful (not enough money to spend), get payment information
Output Specification	Check if correct functions are called in PaymentHandler
Environmental Needs	

3.1.3 CarHandler

Test Case Identifier	I3T1
Test Items	(CarHandler) CUDCommands → Database
Input Specification	Inputs for interting, modifying and deleting database information: add a car (already existing plate number, valid information), delete a car(car status 'in use', car status 'reserved', car status 'available', car status 'unavailable'), update car information (valid information, invalid information), change car status, set car position
Output Specification	Check if information is correctly update in the database
Environmental Needs	PersistenceLibrary

Test Case Identifier	I3T2
Test Items	(CarHandler) SearchFilters → Database
Input Specification	Typical inputs for getting information from database: get car list, get car details, get car position, get car status
Output Specification	Check if requested information is returned correctly
Environmental Needs	PersistenceLibrary

Test Case Identifier	I3T3
Test Items	(CarHandler) CUDCommands → MapAdapter
Input Specification	Typical inputs for getting Map Informations: get car position, get distance from power grid
Output Specification	Check if requested information is returned correctly
Environmental Needs	

Test Case Identifier	I3T4
Test Items	RESTfull API → CarHandler
Input Specification	Delete a car whose status is 'in use' or 'reserved', delete a car whose status is 'available', add a car (valid inputs, invalid plate number), update car information (valid inputs, invalid inputs), get cars list, get car details, set car details (valid inputs, invalid inputs), end the ride, unlock the car (valid user position, invalid user position), lock the car
Output Specification	Check if correct functions are called in CarHandler
Environmental Needs	

3.1.4 AreaHandler

Test Case Identifier	I4T1
Test Items	(AreaHandler) CUDCommands → Database
Input Specification	Inputs for interting, modifying and deleting database information: add a safe area (already existent, valid input), delete a safe area (with some cars, with no cars), update safe area details (valid inputs, invalid inputs)
Output Specification	Check if information is correctly update in the database
Environmental Needs	PersistenceLibrary

Test Case Identifier	I4T2
Test Items	(AreaHandler) SearchFilters → Database
Input Specification	Typical inputs for getting information from database: get areas list, get area details, get areas by position
Output Specification	Check if requested information is returned correctly
Environmental Needs	PersistenceLibrary

Test Case Identifier	I4T3
Test Items	(AreaHandler) CUDCommands → MapAdapter
Input Specification	Typical inputs for getting Map Informations: get distances, get area position
Output Specification	Check if requested information is returned correctly
Environmental Needs	

Test Case Identifier	I4T4
Test Items	RESTfull API → AreaHandler
Input Specification	Get all safe areas list, get safe area details, get cars parked in the safe area, get safe areas by position, add a new safe area (valid inputs, invalid inputs), delete a safe area (with cars, with no cars), update safe area information (valid input, invalid input)
Output Specification	Check if correct functions are called in AreaHandler
Environmental Needs	

3.1.5 RideHandler

Test Case Identifier	I5T1
Test Items	(RideHandler) CUDCommands → Database
Input Specification	Inputs for interting, modifying and deleting database information: set ride details (valid inputs, invalid inputs), set ride current cost
Output Specification	Check if information is correctly update in the database
Environmental Needs	PersistenceLibrary

Test Case Identifier	I5T2
Test Items	(RideHandler) SearchFilters → Database
Input Specification	Typical inputs for getting information from database: get car details, get ride details
Output Specification	Check if requested information is returned correctly
Environmental Needs	PersistenceLibrary

Test Case Identifier	I5T3
Test Items	RESTfull API → RideHandler
Input Specification	Apply possible discounts (check all different cases as input), calculate total ride cost, charge the user, calculate the current cost, set ride details, sent user email containing ride details, end of a ride
Output Specification	Check if correct functions are called in RideHandler
Environmental Needs	

3.1.6 ReservationHandler

Test Case Identifier	I6T1
Test Items	CUDCommands → Database
Input Specification	Inputs for interting, modifying and deleting database information: set reservation details
Output Specification	Check if information is correctly update in the database
Environmental Needs	PersistenceLibrary

Test Case Identifier	I6T2
Test Items	SearchFilters → Database
Input Specification	Typical inputs for getting information from database: get reservation details, get available cars, get user active reservations, get user status, get user position, get car position
Output Specification	Check if requested information is returned correctly
Environmental Needs	PersistenceLibrary

Test Case Identifier	I6T3
Test Items	RESTfull API → ReservationHandler
Input Specification	Reserve more than one car at the same time, reserve only one car, user status isn't 'active', user status is 'active', reservation expiration, get available cars (no available car, some available cars)
Output Specification	Check if correct functions are called in Reservation
Environmental Needs	

3.2 Test Procedures

3.2.1 UserHandler

Test Procedure Identifier	TP1
Purpose	This test procedure verifies whether the MapAdapter subcomponent in the UserHandler component: • can handle CUDCommands inputs • can output requested information to the CUDCommands subcomponent This test procedure verifies whether the CUDCommands subcomponent in the UserHandler component: • can handle Validator inputs • can handle MapAdapter inputs • can handle database inputs (through PersistenceLibrary) • can output requested information to the Validator subcomponent This test procedure verifies whether the Validator subcomponent in the UserHandler component: • can handle CUDCommands inputs • can handle RESTful API inputs • can output requested information to the RESTful API component This test procedure verifies whether the SearchFilters subcomponent in the UserHandler component: • can handle database inputs (through PersistenceLibrary) • can handle RESTful API inputs • can output requested information to the RESTful API component
Procedure Steps	Excecute I1

3.2.2 PaymentHandler

Test Procedure Identifier	TP2
Purpose	This test procedure verifies whether the PaymentAdapter subcomponent in the PaymentHandler component: • can handle CUDCommands inputs • can output requested information to CUDCommands subcomponent This test procedure verifies whether the CUDCommands subcomponent in the PaymentHandler component: • can handle Validator inputs • can handle PaymentHandler inputs • can handle database inputs (through PersistenceLibrary) • can output requested information to Validator subcomponent This test procedure verifies whether the Validator subcomponent in the PaymentHandler component: • can handle RESTful API inputs • can handle CUDCommands inputs • can output requested information to RESTful API component This test procedure verifies whether the SearchFilters subcomponent in the PaymentHandler component: • can handle database inputs (through PersistenceLIbrary) • can handle RESTful API inputs • can output requested information to the RESTful API component
Procedure Steps	Excecute I3

3.2.3 CarHandler

Test Procedure Identifier	TP3
Purpose	This test procedure verifies whether the MapAdapter subcomponent in the CarHandler component: • can handle CUDCommands inputs • can output requested information to the CUDCommands subcomponent This test procedure verifies whether the CUDCommands subcomponent in the CarHandler component: • can handle Validator inputs • can handle MapAdapter inputs • can handle database inputs (through PersistenceLibrary) • can output requested information to the Validator subcomponent This test procedure verifies whether the Validator subcomponent in the CarHandler component: • can handle CUDCommands inputs • can handle RESTful API inputs • can output requested information to the RESTful API component This test procedure verifies whether the SearchFilters subcomponent in the CarHandler component: • can handle database inputs (through PersistenceLibrary) • can handle RESTful API inputs • can output requested information to the RESTful API component
Procedure Steps	Excecute I3

3.2.4 AreaHandler

Test Procedure Identifier	TP4
Purpose	This test procedure verifies whether the MapAdapter subcomponent in the AreaHandler component: • can handle CUDCommands inputs • can output requested information to the CUDCommands subcomponent This test procedure verifies whether the CUDCommands subcomponent in the AreaHandler component: • can handle Validator inputs • can handle MapAdapter inputs • can handle database inputs (through PersistenceLibrary) • can output requested information to the Validator subcomponent This test procedure verifies whether the Validator subcomponent in the AreaHandler component: • can handle CUDCommands inputs • can handle RESTful API inputs • can output requested information to the RESTful API component This test procedure verifies whether the SearchFilters subcomponent in the AreaHandler component: • can handle database inputs (through PersistenceLibrary) • can handle RESTful API inputs • can output requested information to the RESTful API component
Procedure Steps	Excecute I4

3.2.5 RideHandler

Test Procedure Identifier	TP5
Purpose	This test procedure verifies whether the CUDCommands subcomponent in the RideHandler component: • can handle database inputs (through PersistenceLibrary) • can handle Validator inputs • can output requested information to Validator subcomponent This test procedure verifies whether the Validator subcomponent in the RideHandler component: • can handle RESTful API inputs • can handle CUDComponents inputs • can output requested information to RESTful API component This test procedure verifies whether the SearchFilters subcomponent in the RideHandler component: • can handle database inputs (through PersistenceLibrary) • can handle RESTful API inputs • can output requested information to RESTful API component
Procedure Steps	Excecute I5

3.2.6 ReservationHandler

Test Procedure Identifier	TP6
Purpose	This test procedure verifies whether the CUDCommands subcomponent in the ReservationHandler component: • can handle database inputs (through PersistenceLibrary) • can handle Validator inputs • can output requested information to Validator subcomponent This test procedure verifies whether the Validator subcomponent in the ReservationHandler component: • can handle RESTful API inputs • can handle CUDComponents inputs • can output requested information to RESTful API component This test procedure verifies whether the SearchFilters subcomponent in the ReservationHandler component: • can handle database inputs (through PersistenceLibrary) • can handle RESTful API inputs • can output requested information to RESTful API component
Procedure Steps	Excecute I6

4 Tools and Test Equipment Required

Since the beginning of this project we never force the adoption of any specific programming language or software solution. In this section we will describe the required tools on a point of view of the functionalities they should provide. In the end we will provide some examples for a couple of programming languages but they should be taken for what they are, examples. The first tool we need is a good integration test framework providing a full set of functionalities. This should include mocks and and tools to perform integration testing with ease. Examples in this direction for Java would be Arquillian for Integration testing and Mockito for mocks. Similarly, the same could be achieved in Ruby with Rspec. The second tool is a behaviour test framework to test web components automatically without any manual intervention. Examples in this sense would be Cucumber+Selenium for Java and Capybara for Ruby

5 Program Stubs and Test Data Required

As we have explained before, we adopted a bottom-up strategy for the integration testing: we have tested integrations starting from the low level components, so that the testing for higher component is easier. Furthermore, our architecture is buil using microservices, that by definition operate independently from each other. In that way, as it is built our architecture, we don't need any 'stubs' or 'drivers', which are temporary dummy programs that replace uncompleted functionalities or components.

6 Effort Spent

In this chapter we provide information about the number of hours each group member has worked towards the fulfillment of this document.

Section	Hours	Author
Introduction	3	Patrizia Porati, Tommaso Sardelli
Integration strategy	7	Tommaso Sardelli
Individual Steps and Test Description	10	Patrizia Porati
Tools and Test Equipment Required	4	Tommaso Sardelli
Program Stubs and Test Data Required	2	Tommaso Sardelli
Review	5	Patrizia Porati, Tommaso Sardelli
Graphic adjustments	2	Patrizia Porati, Tommaso Sardelli