Integration Test Plan Document

January 15, 2017

Version 1.0

- Davide Alessandro Cottica (mat. 806868)
- Stefano Antonino Badalucco (mat. 809322)

Contents

1 Introduction

1.1 Purpose and Scope

1.1.1 Purpose

The purpose of the integration test plan is to describe the necessary tests to verify that all of the components of PowerEnjoy are properly assembled. Integration testing ensures that the unit-tested modules interact correctly.
It also includes the description of the tools that are needed due to fullfill the testing.
In this document are described too the required stubs, data structures and drivers that we will use during this process.

1.1.2 Scope

The project PowerEnjoy will provide a service based on mobile application and web application, and the only target of this service is the clients.
All the clients of the service must be registered into the system.
The system allows the client to reserve a car, not already reserved, among the ones available. The client could make a reservation via the mobile application or web application selecting the car from a map that shows all the car of the company. After the client makes a reservation, he has a limited time to reach the selected car. If the reservation expires a penalty is subtracted from the user bill.
When the client turns on the engine of the car, the ride starts. At the end of the ride the system subtract the cost of the ride from the user's bill.
This system could be very useful for all the people that would like to reach places not covered by public transportation. This service is attainable everyday and at any time of the day. This project has also as purpose the decrease of the pollution in the city using electric car instead petrol-based car.
The system will track in each moment car's position with a GPS system located inside the car.
So the main purpose of the system is offering a car rent service simple to use that takes care about the environment.

1.2 Definition, Acronyms, Abbreviation

1.2.1 Definitions

- Subcomponent: each of the low level components realizing the func- tionalities of a subsystem.

1.2.2 Acronyms

- RASD: Requirements analysis and specifications documents.
- DD: design document.

- ITPD: Integration Test Plan Document.
- UI: User Interface.
- API: application programming interface; it is a common way to communicate with another system.
- MVC: model view control.
- ServiceNow: offers everything-as-a-service cloud. computing that offers everything-as-a-service cloud computing.
- SOA: Service-oriented Architecture.
- MEP: message exchange pattern.
- SOAP: Simple Object Access Protocol.
- REST:Representational State Transfer.

1.2.3 Abbreviations

- WebApp as for Web Application.

1.3 Reference Documents

This ITPD document is based on the previous DD, RASD documents.

2 Integration Strategy

2.1 Entry Criteria

There are some important criteria that make the ITPD's results consistent:

- RASD and DD documents must be finished. This is a required step in order to have a complete picture of the interactions between the different components of the system and of the functionalities they offer. It is very important that the ITPD is coherent with these documents.
- INTEGRATION TEST. All the single unit of our system must be tested with success.
- UNIT TEST. Integration testing must be done after unit testing. All the methods and classes must be tested using JUnit. In this way we can isolate each part of the program and show that the individual parts are correct. Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit, and how to use it, can look at the unit tests to gain a basic understanding of the unit's interface (API).

2.2 Elements to be Integrated

As described in the Design Document, we developed the system following an MVC logic pattern. In particular we need to test the controller and the view part and how they communicate among themselves. It is mandatory that the database has been already implemented and we assume that both DBMS and ServiceNow platform (software developed from third parties) have been already tested and they are perfectly working. We will start showing the interaction between low-level components and how they form logic subsystems in order to easily communicate with higher-level components. The components to be tested are as follows:

- TechnicianController
- ReservationController
- IncidentController
- IssueController
- ClientController
- SignOnController
- CarController
- STMPGateway
- PushGateway

- DBHandler
- ClientView
- CarView
- EmployeeView

For convenience we will group some of the above components into one:

- ClientView, CarView, EmployeeView → **UserInterfaces**
- STMPGateway, PushGateway → **NotificationController**
- IncidentController, ProblemController → **IssueController**

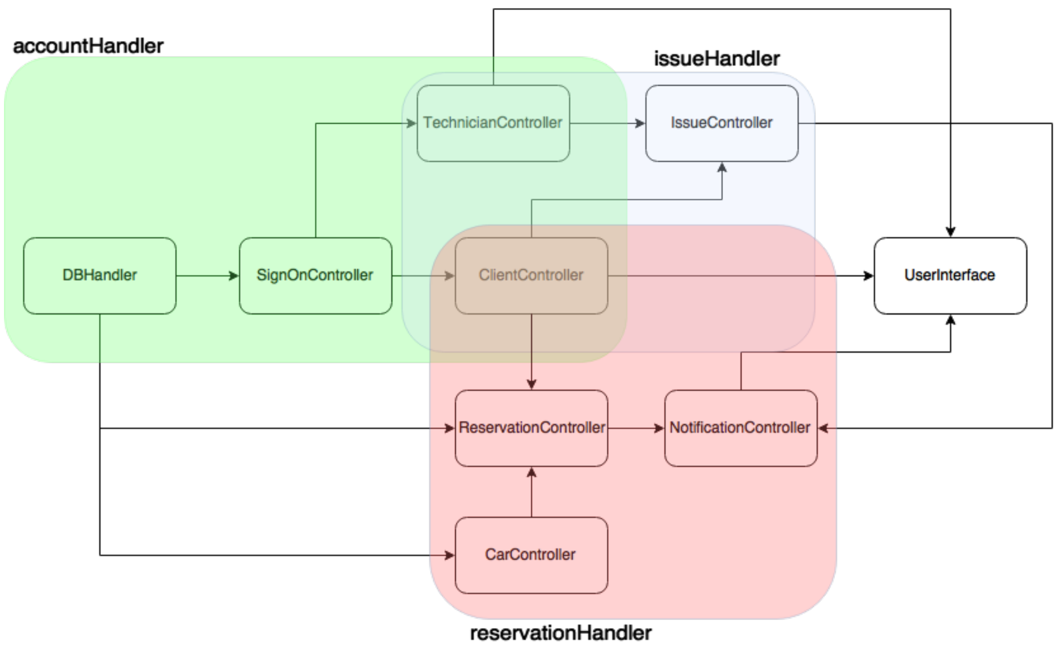
We will further refer to groups and no more to each contained single component.

2.3 Integration Testing Strategy

Our approach to integration testing is based on bottom-up integration strategy. Using this strategy means that the critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are find out early in the process. Another important consideration is that there is no urgency for any kind of program stubs as we start developing with the definite or absolute modules. From this approach derives a number of important advantages.
For example we can do integration test on component that already exists and are fully implemented, in this way we obtain accurate information about how the system works, how it interacts with the other components and how it reacts and fail in real usage world.
Bottom-up approach let to follow closer the development process, which in our case is also proceeding using the bottom-up approach; by doing this 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.

2.4 Sequence of Component

As we are following a bottom-up strategy, we will start testing from the components closer to database.



As can be clearly visible from the image displayed above, the interaction between various components generate different subsystems (we have found 3 subsystems in particular).
For completeness of information, note that the ClientController component isn't shared between subsystem, instead every subsystem is built on a different part of this component.

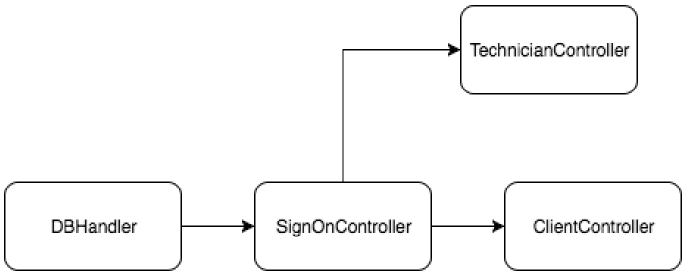


Figure 1: Client subsystem

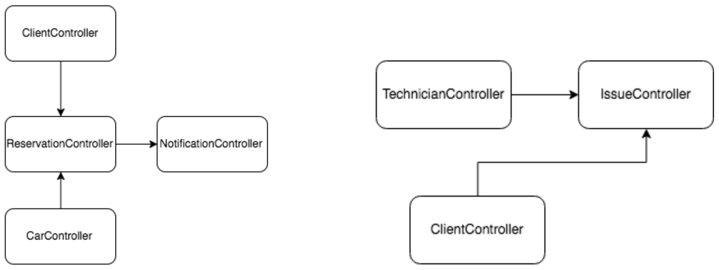


Figure 2: Reservation and Issue subsystems

3 Individual Steps and Test Description

3.1 Integration test case "Sign On"

Test Case ID	TD1
Test Item	DBHandler, SignOnController
Input Specification	SQL interaction with database
Output Specification	Check response
Environmental Needs	-

3.2 Integration test case “Reservation query”

Test Case ID	TD2
Test Item	DBHandler, ReservationController
Input Specification	SQL interaction with database
Output Specification	Check response
Environmental Needs	-

3.3 Integration test case “Car query”

Test Case ID	TD3
Test Item	DBHandler, CarController
Input Specification	SQL interaction with database
Output Specification	Check response
Environmental Needs	-

3.4 Integration test case “Technician authentication”

Test Case ID	TD4.1
Test Item	SignOnController, TechnicianController
Input Specification	TechnicianController well-formed input information
Output Specification	Access granted and session saved
Environmental Needs	TD1

Test Case ID	TD4.2
Test Item	SignOnController, TechnicianController
Input Specification	TechnicianController wrong input information
Output Specification	Access denied and not created
Environmental Needs	TD1

3.5 Integration test case “Client authentication”

Test Case ID	TD5.1
Test Item	SignOnController, ClientController
Input Specification	ClientController well-formed input information
Output Specification	Access granted and session saved
Environmental Needs	TD1

Test Case ID	TD5.2
Test Item	SignOnController, ClientController
Input Specification	ClientController wrong input information
Output Specification	Access denied and not created
Environmental Needs	TD1

3.6 Integration test case “Client Reservation”

Test Case ID	TD6
Test Item	ClientController, ReservationController
Input Specification	A ClientController instance is passed as argument for a ReservationController instance
Output Specification	Check if a correct ReservationController instance is returned
Environmental Needs	TD1, TD2, TD5.1

3.7 Integration test case “Car Reservation”

Test Case ID	TD7
Test Item	CarController, ReservationController
Input Specification	A CarController instance is passed as argument for a ReservationController instance
Output Specification	Check if a correct ReservationController instance is returned
Environmental Needs	TD2, TD3

3.8 Integration test case “Technician issues query”

Test Case ID	TD8
Test Item	TechnicianController, IssueController
Input Specification	A IssueController instance is passed as argument for a TechnicianController instance
Output Specification	Check if a correct TechnicianController instance is returned
Environmental Needs	TD1, TD4.1

3.9 Integration test case “Client Issues Creation”

Test Case ID	TD9
Test Item	ClientController, IssueController
Input Specification	A ClientController instance is create and passed as argument for an IssueController instance
Output Specification	Check if a correct IssueController instance is shown in the database
Environmental Needs	TD1, TD5.1

3.10 Integration test case “Issue Notification”

Test Case ID	TD10
Test Item	IssueController, NotificationController
Input Specification	A IssueController instance is created with a consistent Client message and passed as argument for a NotificationController instance to be sent
Output Specification	Check if a correct NotificationController instance is created and correctly sent to a target Client or group of Clients
Environmental Needs	TD1, TD4.1, TD5.1, TD8

3.11 Integration test case “Reservation Notification”

Test Case ID	TD11
Test Item	ReservationController, Notification-Controller
Input Specification	A ReservationController instance is created with a consistent Client message and passed as argument for a NotificationController instance to be sent
Output Specification	Check if a correct NotificationController instance is created and correctly sent to a target Client or group of Clients
Environmental Needs	TD1, TD5.1, TD6

3.12 Integration test case “Client Session”

Test Case ID	TD12
Test Item	ClientController, UserInterface
Input Specification	A ClientController instance is created and a session is passed to the UserInterface instance
Output Specification	Check if a correct NotificationController instance is created and correctly sent to target Client
Environmental Needs	TD1, TD5.1

3.13 Integration test case “Technician Session”

Test Case ID	TD13
Test Item	TechnicianController, UserInterface
Input Specification	A TechnicianController instance is created and a session is passed to the User-Interface instance
Output Specification	Check if a correct NotificationController instance is created and correctly sent to target Technician
Environmental Needs	TD1, TD4.1

3.14 Integration test case "Notification Sending From Reservation To Client"

Test Case ID	TD14
Test Item	NotificationController, UserInterface
Input Specification	A NotificationController delivers a notification message to target Client
Output Specification	Check if the notification is correctly received from the target UserInterface instance
Environmental Needs	TD1, TD2, TD4.1, TD5.1, TD11, TD12

4 Tools

This section regards the tools we used during the testing process and the reason why we have chosen to use those.

4.1 Tools

Testing tools serve the purpose of automating the testing process of software in some case also with graphical user interfaces.

4.1.1 JUnit Framework

We chose to use JUnit Framework for unit testing. JUnit is an open source framework designed for the purpose of writing and running tests in the Java programming language. The reason of this choice are:

- Every single tits and bits in your software is tested even before module or System level testing is performed.
- Every time you make a small or a big modification in the code (in any function), you can make sure that the function is performing well and has not broken any older functionality by executing all JUnit test cases in one go written for that function.
- JUnit is a standard for testing in java programming language and the majority of the IDE supports it.

4.1.2 Arquillan

Arquillian is an highly extensible testing platform for the JVM that enables developers to easily create automated integration, functional and acceptance tests for Java middleware.
This framework is very useful to test the integration the integration of a component with its container.

4.1.3 Mockito

Mockito is an open source testing framework for Java. This is used to mock interfaces so that a functionality can be added to a mock interface that can be used in unit testing.
Mocking is a way to test functionality of a class in isolation and Mockito facilitates creating this mock object seamlessly.
The benefits of this framework are:

- No need to write mock object on your own.
- Renaming interface method names or reordering parameters will not break the test code as Mock are created at runtime.

- Mockito provides support exception and return values.
- Mockito provides supports check on order of method calls.
- Mockito provides supports creating mocks using annotation.

4.1.4 SoapUI

SoapUI is an application and framework to simplify the testing of web applications and web services. In particular this web service is used for SOA and REST testing.
Its functionality covers web service inspection, invoking, development, simulation and mocking, functional testing, load and compliance testing too.

5 Required Program Stubs and Test Data

5.1 Program Stubs and Drivers

As we are following a bottom-up approach, there is no need to create any stub because DBMS is fully working and operational and data is correctly retrieved from database.
Here follows a list of all the drivers that will be developed as part of the integration testing phase, together with their specific role.

- **SignOnDriver**: this driver will test the SignOnController methods together with their interaction with the DBHandler.
- **ClientDriver**: this driver will test the ClientController methods together with their interaction with the SignOnController and UserInterface.
- **CarDriver**: this driver will test the CarController methods together with their interaction with the DBHandler.
- **ReservationHandlingDriver**: this driver will test the ReservationController methods together with their interaction with the DBHandler, CarController, ClientController and NotificationController in order to test reliability of the **ReservationHandler** subsystem.
- **IncidentHandlingDriver**:this driver will test the IssueController methods together with their interaction with the TechnicianController, ClientController and NotificationController in order to test reliability of the **IssueHandler** subsystem.
- **SOAPDriver, NotificationDriver**: this drivers will test messages sent and received from internet in respect of the standards adopted (WSDL envelopes respecting correct format with mandatory fields filled, SMTP communication correctly set), generated by the NotificationController methods and test their interaction with the ReservationHandler, IssueHandler subsystem and UserInterface instance.
- **UIDriver**: this driver will test the UserInterface methods together with their interaction with the NotificationController, TechnicianController and ClientController instances.
- **SecurityDriver**: this driver is thought to test the most common security vulnerabilities of our system, in particular will test the interaction between client side and server side instances. It will be provided for UserInterface methods sending data to ClientController and TechnicianController instances.

5.2 Test Data

In this section we are going to specify data needed to perform driver test as explained in the previous section.

- SignONDriver
  - Correct and fault session credentials.
  - Null fields.
- ClientDriver
  - Null fields.
  - Email field not compliant with standard.
  - Phone field not compliant with standard.
  - Bill field not compliant with standard.
- CarDriver
  - Null fields.
  - Position updating correctly.
  - Corrupted/non-compliant informations versus correct informations.
- ReservationHandlingDriver
  - Corrupted/non-compliant informations versus correct informations.
  - Null fields.
  - Requests without mandatory fields filled.
  - Multiple requests on same car.
  - Multiple requests by same user.
  - Request made without needed money on bill.
  - Discount calculation with different passengers/location/battery level inserted.
- IncidentHandlingDriver
  - Corrupted/non-compliant informations versus correct informations.
  - Null fields.
- SOAPDriver
  - Non-compliant XML tag / document structure versus correctly filled envelope.
  - Mandatory fields not filled.
  - Not-authorized user sending information.
- NotificationDriver
  - Different target user.
  - Notification triggered at correct time.

    - Null fields.

- UIDriver
    - Credential consistent with displayed informations.
    - User data settings changed correctly client-side.
-
    - Cross site scripting.
    - SQL Injection.
    - DOS attack.
    - Firewall and IDS response on probing/scanning.

6 Hours of work

The amount of working hours:

- Davide Cottica: 15 hours
- Stefano Badalucco: 15 hours