

SOFTWARE ENGINEERING 2 PROJECT  
INTEGRATION TEST PLAN DOCUMENT

# PowerEnJoy



**POLITECNICO  
DI MILANO**

TEAM: NICO MONTALI, ENRICO FINI

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>4</b>  |
| 1.1      | Purpose and Scope . . . . .  | 4         |
| 1.2      | Definitions, Acronyms and Abbreviations . . . . .                              | 4         |
| 1.3      | Reference Documents . . . . .  | 5         |
| <b>2</b> | <b>Integration strategy</b>  | <b>6</b>  |
| 2.1      | Entry criteria . . . . .   | 6         |
| 2.2      | Elements to be integrated . . . . .  | 6         |
| 2.3      | Integration Testing Strategy . . . . .   | 7         |
| 2.4      | Sequence of Integration . . . . .  | 7         |
| 2.4.1    | Model integration . . . . .  | 7         |
| 2.4.2    | Software integration . . . . .   | 8         |
| 2.4.3    | Subsystem integration . . . . .  | 9         |
| <b>3</b> | <b>Individual Steps and Test Description</b>                                   | <b>10</b> |
| 3.1      | STEP I1: VehicleManager → Vehicles . . . . .                                   | 10        |
| 3.2      | STEP I1: DriveManager → VehicleManager, PaymentManager                         | 11        |
| 3.3      | STEP I1: TaskManager → VehicleManager . . . . .                                | 12        |
| 3.4      | STEP I1: Router → AuthorizationManager, DriveManager,<br>TaskManager . . . . . | 13        |
| <b>4</b> | <b>Tools and Test Equipment</b>  | <b>16</b> |
| <b>5</b> | <b>Program Stubs and Test Data Required</b>                                    | <b>17</b> |
| <b>6</b> | <b>Work review</b>   | <b>18</b> |

# Document Status

|                       |                                |
|-----------------------|--------------------------------|
| <b>Document Title</b> | Integration Test Plan Document |
| <b>Document ID</b>    | PowerEnjoy/ITPD/1.0            |
| <b>Authors</b>        | N. Montali, E. Fini            |
| <b>Version</b>        | 1.0                            |

## Changelog

| <b>Version</b> | <b>Date</b> | <b>Changes</b>  |
|----------------|-------------|-----------------|
| 1.0            | 9/01/2017   | Initial version |

# Introduction

## 1.1 Purpose and Scope

This document describes the plans for testing the integration of the components of the PowerEnjoy system. The purpose of this document is to describe in a detailed way how to test the interfaces described in the Design Document [DD, Interfaces, Paragraph 2.7]. The integration tests will follow the sequence provided in this document, described later in Integration Strategy (Chapter 2). Chapter 3 will instead give a more detailed insight of the single tests to be run.

In this document we only take into account the integration testing of the components, i.e. test if the connections between components are functional, while Unit Testing, i.e. testing a single component in a stand-alone manner, is not described here and is assumed as already done when the component is considered ready for integration testing.

## 1.2 Definitions, Acronyms and Abbreviations

**Driver** Client of the system, i.e. the one that uses the service, reserves and drives. Every Driver must provide personal informations (name, surname, email, birthdate, a valid license and payment information;

**Drive** A single usage of the service, starts when the user turn on the engine, stops when the car is locked inside a Safe Area;

**Worker** Employee of the company, perform physical actions (moving, charging etc) on cars;

**Report** Car issue reported by the Driver during a Drive. Will later be assigned to a Worker;

**Task** Piece of work assigned to a worker. Different type of tasks are described later. Every task assigns a single car to a single worker;

**Administrator** Employee of PowerEni that is in charge of the administration of the system through the administration console;

**RASD** Requirements Analysis and Specifications Document;

**DD** Design Document;

**API** Application Programming Interface;

**DBMS** DataBase Management System;

**REST** REpresantional State Transfer.

## 1.3 Reference Documents

- Previously released RASD v1.1
- Previously released DD v1.2
- Integration testing samples (SpinGrid, myTaxiService)

## Integration strategy

### 2.1 Entry criteria

Integration Testing can start as long as the entry criteria stated below are met. First of all, the RASD and the DD documents must have been completed and accepted, since we need a complete view of the problem and the design of the system.

Also, integration should start only when the estimated percentage of completion of the various components met this requirement

- 95% of the Core functionalities
- 50% of the Client functionalities

This percentage describe only the entry criteria for the integration testing phase, not the actual integration test of the component (obviously possible only when the component is almost complete). The relatively high percentage of the Core components is due to the high correlation between components, while the relatively low percentage regarding the clients is due to the relative simplicity of them w.r.t the Core.

### 2.2 Elements to be integrated

In the DD, the structure of the system is clearly divided into high-level components, e.g. the Core and Clients, and lower-level component, i.e. the subcomponents of the Core. So, the integration phase will be performed at different level of abstraction. Given that the lower-level components compose the essential high-level component of the system (the Core), we will first integrate the lower-level and then proceed to higher levels.

The first critical component of the system is the Data Access Layer, that is implemented through an external Node.JS library (Sequelize, DD). For this reason, all the CRUD operations (Create, Read, Update, Delete) on the DB

are considered as already tested. The second critical component is the Vehicle Management Layer, that is the network of vehicles connected to the system. We need this component for the correct behaviour of internal subcomponents of the Core. The lower-level components to be tested in the first phase are: **Vehicle Manager, Drive Manager, Payment Manager, Router, Authorization Manager and Task Manager**. The high-level components of the system are all on the same level w.r.t. the Core. We will integrate **Android driver app, iOS driver app, driver web portal, Android worker app, iOS worker app, administrator web portal**.

## 2.3 Integration Testing Strategy

We are going to use mainly a bottom-up approach during the integration testing of lower-level components. So, we will start integrating the components that does not depend on other components or depend on already developed components. Since we have many simple components that are very independent (Vehicle Manager, Payment Manager, Authorization Manager), this approach gives us the advantage to begin the testing phase earlier and start to integrate as soon as components are ready and functional. The second phase will follow a critical-first approach, since the components here are only dependent to the Core. So, the order will reflect the risk represented by the incorrect behaviour of the component.

## 2.4 Sequence of Integration

This section contains the detailed integration sequence, starting from the Core subsystem in paragraph 2.4.1 to the entire system integration in paragraph 2.4.2. The first integration tests, labeled IM, are the ones regarding the model. This steps will be described only briefly, since we need to test all the CRUD operations on the DB. Instead of enumerating inputs and relative output, we state what tables are used by a specific component.

### 2.4.1 Model integration

**STEP IM1: VehicleManager → Model**

- READ, UPDATE on **Vehicles**

#### **STEP IM2: DriveManager → Model**

- CREATE, READ, UPDATE, DELETE on **Reservations**
- CREATE, READ, UPDATE on **Drive**

#### **STEP IM3: TaskManager → Model**

- CREATE, READ, UPDATE, DELETE on **Reservations**
- CREATE, READ, UPDATE, DELETE on **Tasks**
- CREATE, READ, UPDATE, DELETE on **Tasks**

#### **STEP IM4: AuthorizationManager → Model**

- CREATE, READ, UPDATE, DELETE on **Users**

### **2.4.2 Software integration**

#### **STEP I1: VehicleManager → Vehicles**

The VehicleManager component is the only one that interacts directly with the vehicles. We will avoid the description of the polling tracking system since it should be very straight-forward: every 30 seconds the VehicleManager requests the state of all the vehicles, and update the record in the DB with the new state. Instead, we will test the performAction and getLastDrive functions. We will not test the automatic report here but in step I3 (since we need TaskManager).

#### **STEP I2: DriveManager → VehicleManager, PaymentManager**

We then proceed to integrate DriveManager and their subcomponents. All the DriveManager's requests includes a call to VehicleManager, while PaymentManager is only called on reserve and stop. We will need a driver to call DriveManager interface.

#### **STEP I3: TaskManager → VehicleManager**

TaskManager calls VehicleManager to change vehicle's state to maintenance. We will need a driver to call all the TaskManager interface. Also, during polling, VehicleManager can make automatic reports (with the data provided by the vehicle itself), so, we will need to simulate a report from a vehicle.



#### **STEP I4: Router → DriverManager, TaskManager, Authorization-Manager**

We then proceed with the integration of the central component of the Core, the router. We will need an HTTP client to act as a driver, calling all the possible endpoints on the router.

### **2.4.3 Subsystem integration**

#### **STEP IC1: Android Driver App → Core**

The first high-level integration we will perform is the Android driver app. We have chosen to integrate first the driver app because is the only component that is publicly available and will be the most used. We start from Android because we predict that the majority of client will be Android ones (by mobile market distribution). The app testing requires the usage of all the functionalities of the app, verifying at every step on the server-side if the interaction raise some errors. We will not cover this testing in details. The devices used for testing are listed below.

#### **STEP IC2: iOS Driver App → Core**

The same consideration stands for the iOS driver app, tested like the Android one. The devices used for testing are listed below.

#### **STEP IC3: Driver Web Portal → Core**

As the mobile apps, the driver web portal is publicly available, but will be probably used less than the apps. Also this components is tested in every function it has. The devices and browsers used for testing are listed below.

#### **STEP IC4: Android and iOS Worker App → Core**

We then proceed to test workers apps, as before we test all the functionalities provided here. The devices and browsers used for testing are listed below.

#### **STEP IC5: Admin Web Portal → Core**

Test the admin web portal as before, test all the functionalities provided. The devices and browsers used for testing are listed below.

## Individual Steps and Test Description

### 3.1 STEP I1: VehicleManager → Vehicles

| performAction(vehicleId, action) |  |
|----------------------------------|--|
| <i>Input</i>                     | <i>Effect</i>  |
| A null vehicleId or action       | A NullArgumentException is raised  |
| Inexistent VehicleId             | An InvalidArgumentException is raised  |
| Unhandled action                 | An InvalidArgumentException is raised  |
| Valid inputs                     | An HTTP-POST is sent to the vehicle to perform the action. Action confirmation is sent when ready. |

| getLastDrive(vehicleId) |   |
|-------------------------|---|
| <i>Input</i>            | <i>Effect</i>   |
| A null vehicleId        | A NullArgumentException is raised   |
| Inexistent VehicleId    | An InvalidArgumentException is raised   |
| Valid inputs            | An HTTP-GET is sent to the vehicle to get the last Drive log. Log is returned when ready. |

### 3.2 STEP I1: DriveManager → VehicleManager, PaymentManager

| reserve(user, vehicle)  |   |
|---|---|
| <i>Input</i>  | <i>Effect</i>                             |
| A null parameter  | A NullPointerException is raised          |
| An inexistent Vehicle ID  | An InvalidArgumentException is raised     |
| A "busy" Vehicle ID   | A StateException is raised                |
| The ID of a user who has another active reservation                       | An invalidUserException is raised         |
| The ID of a user who does not have the minimum amount of credit available | An invalidUserException is raised         |
| All the inputs are correct  | The state of the vehicle is set to "busy" |

| cancel(reservation)          |  |
|------------------------------|--|
| <i>Input</i>                 | <i>Effect</i>                                |
| A null parameter             | A NullPointerException is raised             |
| An inexistent Reservation ID | An InvalidArgumentException is raised        |
| Formally valid argument      | The Reservation is removed from the database |

| start(reservation)           |  |
|------------------------------|--|
| <i>Input</i>                 | <i>Effect</i>  |
| A null parameter             | A NullPointerException is raised   |
| An inexistent Reservation ID | An InvalidArgumentException is raised                                    |
| Formally valid argument      | The state of the Reservation is updated and a new Drive entry is created |

| stop(drive)             |   |
|-------------------------|---|
| <i>Input</i>            | <i>Effect</i>   |
| A null parameter        | A NullPointerException is raised  |
| An inexistent Drive ID  | An InvalidArgumentException is raised   |
| Formally valid argument | A request for the lastDrive log is sent to the Vehicle. The state of the Drive and the respective Reservation are then updated. The user is charged for the computed amount based on the Drive log. |

### 3.3 STEP I1: TaskManager $\rightarrow$ VehicleManager

| makeTask(data)                                   |   |
|--|---|
| <i>Input</i>                                     | <i>Effect</i>   |
| A null parameter                                 | A NullPointerException is raised  |
| Data parameter contains an inexistent vehicle ID | An InvalidArgumentException is raised   |
| Formally valid argument                          | A new task is created and assigned to a worker. The state of the respective vehicle is updated. |

| updateTask(task, state)                       |   |
|---|---|
| <i>Input</i>                                  | <i>Effect</i>   |
| A null parameter                              | A NullPointerException is raised  |
| State parameter contains an inexistent state  | An InvalidArgumentException is raised   |
| Task parameter contains an inexistent task ID | An InvalidArgumentException is raised   |
| Formally valid argument                       | The state of the task is updated, and eventually the state of the respective vehicle is updated |

### 3.4 STEP I1: Router → AuthorizationManager, DriveManager, TaskManager

| login(data)   |                                       |
|---|---------------------------------------|
| <i>Input</i>  | <i>Effect</i>                         |
| A null parameter  | A NullArgumentException is raised     |
| Data parameter contains an inexistent username                            | An InvalidArgumentException is raised |
| Data parameter contains empty username or password                        | An InvalidArgumentException is raised |
| Data parameter contains a valid username but password does not correspond | Returns False                         |
| Data parameter contains valid username and password corresponds           | Returns True                          |

| logout(data)                                   |                                       |
|--|---------------------------------------|
| <i>Input</i>                                   | <i>Effect</i>                         |
| A null parameter                               | A NullArgumentException is raised     |
| Data parameter contains an inexistent username | An InvalidArgumentException is raised |
| Data parameter contains empty username         | An InvalidArgumentException is raised |
| Data parameter contains valid username         | Current session is deleted            |

| signup(data)   |                                       |
|--|---------------------------------------|
| <i>Input</i>   | <i>Effect</i>                         |
| A null parameter   | A NullArgumentException is raised     |
| Data parameter contains empty username or password                                   | An InvalidArgumentException is raised |
| Data parameter contains a username which does not comply with the regular expression | An InvalidArgumentException is raised |
| Data parameter contains valid username and password                                  | Returns True                          |

| manageWorker(worker, data) (for Admins only)                          |  |
|---|--|
| <i>Input</i>  | <i>Effect</i>  |
| A null parameter  | A NullPointerException is raised                                 |
| The calling user is not an authorized admin                           | An unauthorized error (401) is thrown.                           |
| Data parameter contains an inexistent Worker ID                       | An InvalidArgumentException is raised                            |
| Data parameter contains an invalid operation and specification (data) | An InvalidArgumentException is raised                            |
| Formally valid argument   | The specified operation is performed (e.g. register, unregister) |

| makeTask(data) (for Admins only)            |  |
|---|--|
| <i>Input</i>                                | <i>Effect</i>                          |
| A null parameter                            | A NullPointerException is raised       |
| The calling user is not an authorized admin | An unauthorized error (401) is thrown. |
| Formally valid argument                     | A new Task is created                  |

| changeMyState(data) (for Workers only)       |  |
|--|--|
| <i>Input</i>                                 | <i>Effect</i>                          |
| A null parameter                             | A NullPointerException is raised       |
| The calling user is not an authorized worker | An unauthorized error (401) is thrown. |
| Data parameter contains unhandled data       | an InvalidArgumentException is raised  |
| Formally valid argument                      | The state of the Worker is updated     |

| updateTask(data) (for Workers only)   |  |
|---|--|
| <i>Input</i>  | <i>Effect</i>                          |
| A null parameter  | A NullPointerException is raised       |
| The calling user is not an authorized worker (must also be the owner of the Task) | An unauthorized error (401) is thrown. |
| Data parameter contains an inexistent Task ID                                     | an InvalidArgumentException is raised  |
| Formally valid argument   | The state of the Task is updated       |

| reserve(data)                                    |  |
|--|--|
| <i>Input</i>                                     | <i>Effect</i>                          |
| A null parameter                                 | A NullPointerException is raised       |
| The calling user is not an authorized Driver     | An unauthorized error (401) is thrown. |
| Data parameter contains an inexistent User ID    | An InvalidArgumentException is raised  |
| Data parameter contains an inexistent Vehicle ID | An InvalidArgumentException is raised  |
| Formally valid argument                          | A new Reservation is created           |

| unlock(data)  |  |
|---|--|
| <i>Input</i>  | <i>Effect</i>                            |
| A null parameter  | A NullPointerException is raised         |
| The calling user is not an authorized Driver (must also have an active reservation for the vehicle) | An unauthorized error (401) is thrown.   |
| Data parameter contains an inexistent Vehicle ID  | An InvalidArgumentException is raised    |
| Formally valid argument   | An unlock command is sent to the vehicle |

| cancelReservation(data)  |  |
|--|--|
| <i>Input</i>   | <i>Effect</i>                          |
| A null parameter   | A NullPointerException is raised       |
| The calling user is not an authorized Driver (must also be the owner of the reservation) | An unauthorized error (401) is thrown. |
| Data parameter contains an inexistent Reservation ID                                     | an InvalidArgumentException is raised  |
| Formally valid argument  | The Reservation is cancelled           |

## Tools and Test Equipment

Since we are using Node.JS as our main programming language, we will use tools for Unit Testing and Integration Testing specific for it. These tools are open-source (the github link is provided below)

- **Mocha** Test Engine, i.e. run tests at different level. [GitHub](#)
- **Chai** Logic, i.e. to provide assertions [GitHub](#)
- **Sinon** Stubs and Drivers [GitHub](#)

For the high-level integration with clients, instead, we will list devices that will be used to perform tests:

**Android clients** Various Android devices, preferably by Google (Nexus), each major version from 4.0 to the latest 7.1 beta. Test also different screen sizes.

**iOS clients** Various iOS devices, from iOS 8.0 to 10.2.1 beta.

**Browser** Various browsers (Chrome, Safari, Internet Explorer, Firefox) with different screen sizes.



## Program Stubs and Test Data Required

Since we used the Bottom-Up approach, we don't any implementation of stubs. Instead, we need Drivers to perform all the tests described in Chapter 3. For every integration step a driver is required, the test data set is straight-forward from the specification of chapter 3 again. To test the Router functionalities, an HTTP-client "driver" must be used, since the router also parses HTTP requests.

## Work review

Based on our log of the work phases, the total amount of hour of work required were:

- N. Montali: 26 hours
  -
- E. Fini: 24 hours
  - Introduction