Software Engineering 2 Project
Integration Test Plan Document

PowerEnJoy

POLITECNICO
DI MILANO

Team: Nico Montali, Enrico Fini

Contents

Document Status

Document Title	Integration Test Plan Document
Document ID	PowerEnjoy/ITPD/1.0
Authors	N. Montali, E. Fini
Version	1.0

Changelog

Version	Date	Changes
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 and Abbreviations

1.3 Referenced Documents

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 v1.1). For this reason, all the CRUD operations (Create, Read, Update, Delete)

on the DB are considered as already tested. The usage of these operations inside components are consequentially already tested in Unit Testing. 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

2.4.1 Software integration

STEP I1: DriveManager → VehicleManager

This integration contains the most important components in our system. All the DriveManager functionalities are tested. Since DriveManager uses also PaymentManager, but we still want to integrate one-by-one, we will use a PaymentManager stub, that simply simulate a payment and the availability check (always replying in a correct way). We will need a driver to call the relevant DriveManager's functions.

STEP I2: DriveManager → PaymentManager

The previous PaymentManager stub is replaced by the real component, and all the functionalities that require a payment are tested. We also will need a driver to call these DriveManager's functions.

STEP I3: TaskManager → VehicleManager

The integration proceed to TaskManager, which only depends on VehicleManager. A driver to call TaskManager interface is required.

STEP I4: Router → AuthorizationManager

TODO

STEP I4: Router → DriveManager

TODO

2.4.2 Subsystem integration

2.5 Test items

The items are the integrations of the components previously described in [Design Document, DD, Paragraph 2.X].

Individual Steps and Test Description

3.0.1 Router, AuthorizationManager

login(data)	
Input	*Effect*
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)	
Input	*Effect*
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)	
Input	*Effect*
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

3.0.2 DriveManager, VehicleManager

reserve(user, vehicle)	
Input	*Effect*
A null parameter	A NullArgumentException 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
Formally valid argument	The state of the vehicle is set to "busy"

cancel(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
An inexistent Reservation ID	An InvalidArgumentException is raised
Formally valid argument	The Reservation is removed from the database

start(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException 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)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
An inexistent Drive ID	An InvalidArgumentException is raised
Formally valid argument	The state of the Drive and the respective Reservation are updated

3.0.3 DriveManager, PaymentManger

reserve(user, vehicle)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
An inexistent user ID	An InvalidArgumentException is raised
The ID of a user who has another active reservation	An invalidUserException is raised
Formally valid argument	A fixed amount is pre-authorized from the user's credit card

TODO manca check availability

stop(drive)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
An inexistent Drive ID	An InvalidArgumentException is raised
Formally valid argument	The right amount is charged on the user's credit card

3.0.4 TaskManager, VehicleManger

makeReport(data)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Data parameter contains an inexistent vehicle ID	An InvalidArgumentException is raised
Formally valid argument	A new report is created and the state of the respective vehicle is updated

makeTask(data)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Data parameter contains an inexistent vehicle ID	An InvalidArgumentException is raised
Formally valid argument	A new task is created and the state of the respective vehicle is updated

updateTask(task, state)	
Input	*Effect*
A null parameter	A NullArgumentException 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

3.0.5 Router, DriveManager

reserve(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent User ID	An InvalidArgumentException is raised
Request parameter contains an inexistent Vehicle ID	An InvalidArgumentException is raised
Formally valid argument	A new Reservation is created

unlock(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Vehicle ID	An InvalidArgumentException is raised
Formally valid argument	An unlock command is sent to the vehicle

cancel(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Reservation ID	an InvalidArgumentException is raised
Formally valid argument	The Reservation is cancelled

3.0.6 Router, VehicleManager

endDrive(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Drive ID	an InvalidArgumentException is raised
Formally valid argument	The Drive is stopped and the Vehicle status is updated

3.0.7 Router, TaskManager

report(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Vehicle ID	an InvalidArgumentException is raised
Formally valid argument	A Report is created

changeMyState(request) (for Workers only)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Worker ID	an InvalidArgumentException is raised
Formally valid argument	The state of the Worker is updated

updateTask(request) (for Workers only)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Task ID	an InvalidArgumentException is raised
Formally valid argument	The state of the Task is updated

manageWorker(worker, data) (for Admins only)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Request parameter contains an inexistent Worker ID	An InvalidArgumentException is raised
Formally valid argument	???

makeTask() (for Admins only)	
Input	*Effect*
A null parameter	A NullArgumentException is raised
Formally valid argument	A new Task is created

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

Program Stubs and Test Data Required

5.1 DRAFT OF STUBS

Payment stub
Authorization stub

5.2 DRAFT OF DRIVERS

Software drivers calls the tested objects with a collection of input from the detailed description of the test. Router uses as a driver an HTTP client

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