Integration Test Plan

for

Taxi Management System

Version 1.0

Prepared by Giuseppe Di Francesco & Domenico Iezzi

Sommario

1. Introduction

1.1 Purpose

This document describes the plans for testing the integration of the created components. The purpose of this document is to test the interfaces between the components. Every team member who cooperates in the integration tests should read this document.

1.2 Scope

The aim of the project is to build an application for requesting taxi in a big city. User can queue for a taxi using the web or the smartphone application, and the systems answers with the code for the incoming taxi along with the waiting time. Taxi driver uses a mobile application to inform the system of their availability and to confirm that they are going to take care of a certain call. This document must test the usability of the application. We used mainly web application software and most of them have their own Testing procedures, described in their documentation, that we will take as reference.

1.3 Reference Documents

- Latest version of our R.A.S.D. and our Design document
- AngularJS, Mocha and NodeJS documentation
- Project assignment PDFs and Integration Test Plan Example

2. Integration Strategy

2.1 Entry Criteria

All components of our software must be already unit tested, with positive results. Assuming that, the problems we will find with the next steps will surely depends on the integration between component interfaces.

For instance, we need the completed Design Document, and the Unit Testing document. Also, before the test will begin, we need all the drivers to be completed (as described in the next chapters).

2.2 Elements to be integrated

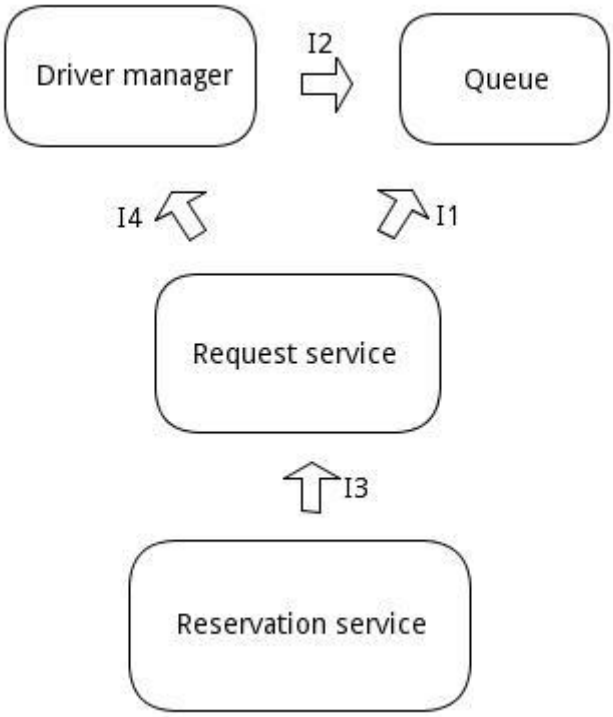
We must test all the interfaces that we can see in our Design Document, precisely in the “Component Info” scheme (See the D.D. for more details). So, in particular, the elements will be either the User application, the Database or the Driver Application communicating with the Application Core. In the bottom level, the most important specific components to test are: Queue, Request Service, Reservation Service, Driver Manager, Driver Service and Request Dispatcher. After that, we will test the notification services for the User and the Driver, and at the End we will involve also the Database, because we are using it only for logs and system recovery.

2.3 Integration Test Strategy

We are using a Bottom Up strategy to test our software. This choice is made mainly because we think it’s the more effective way to find bugs. Also, we are using NodeJS and other JS frameworks that have some powerful tools for unit and interaction testing, and that will simplify the testers work. Given the modularity of the application, building drivers and tests will be straightforward.

2.4 Sequence of Component Integration

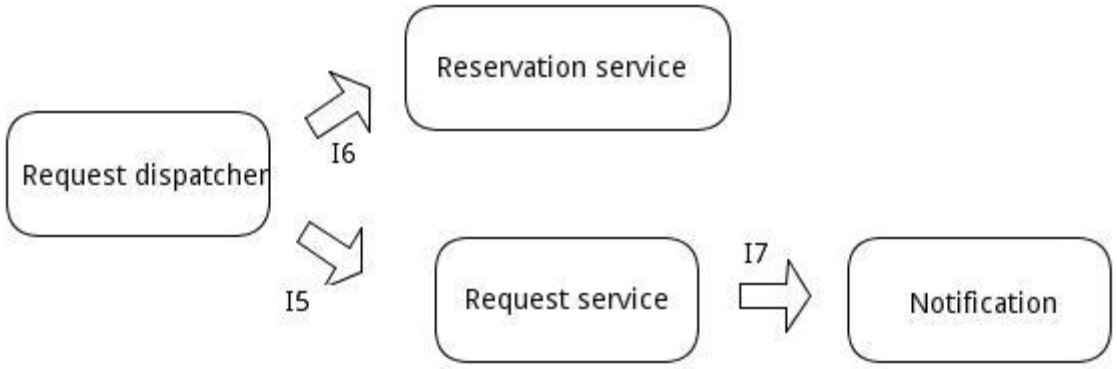
First of all we will analyze the interactions between components of the Application Core. The center of our application is the Queue Manager, because almost all of the functionalities requires a call on that component. So we will begin considering the communication between the Queue Manager and the Request Service. Then, we will test the one between Driver and Request, and, after that, the invocation of the Request by the Reservation Manager.



We have the *I2* interaction when the Driver Manager modifies the queue according to a change of the taxi status. *I3* is used when a reservation is converted into a request, and *I1*, instead, is used when a request needs to read or modify a queue (To make taxi calls or to update it when a taxi accepts the call). I4 is used when the Request Service needs to Retrieve the GPS position of the Taxies.

2.4.1 User Application -> Core

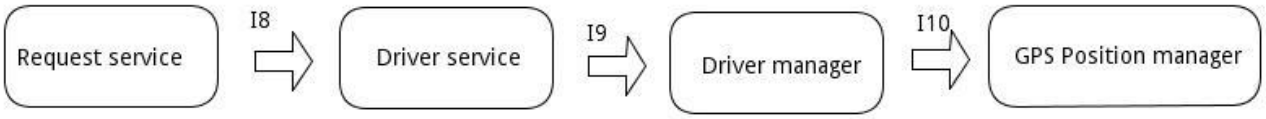
We will now analyze the interaction between the User application and the Core. User Application have one main component that communicates with the Core, and another for the notifications that will receive data from the Core. We will test the first before, because of its main importance.



I5 and I6 corresponds to the "*MakeRequest*" and "*MakeReservation*" functions respectively, which are the main functionalities of this software. I7, instead, represents the "*SendNotification*" function used by the request manager to send notification to the user application.

2.4.2 Driver Application -> Core

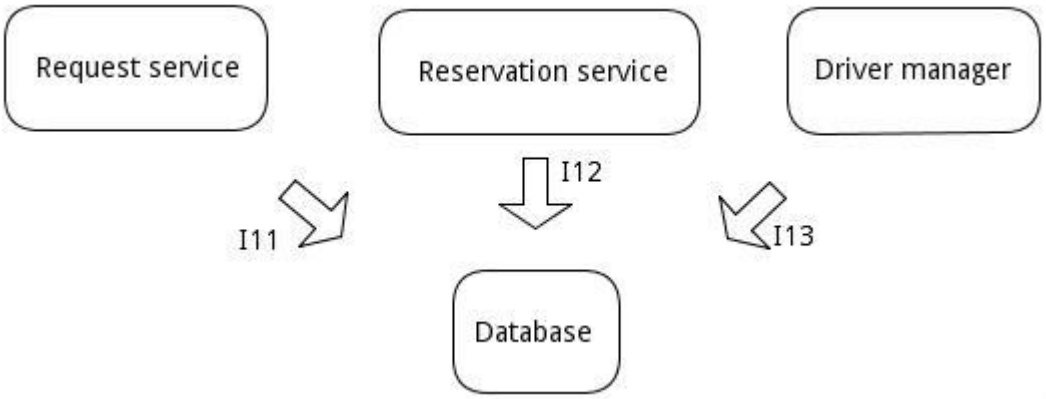
The interaction between the Driver and the Core Application has three different interfaces to do operations not directly related. We will test them in the order we can see on the diagram.



I8 is called when the Request must make a Call to the Taxi and verify its response. I9, instead, is called when a Taxi Driver must change his availability. I10 allows the driver manager to retrieve the position of the taxi.

2.4.3 Core->Database

Last thing to try is the interface for the communication with the Database. This will be very simple to test.



I10, I11 and I12 are used whenever we need to store or retrieve Requests, Reservation or Driver information on the database.

2.4.4 Final Test

Note that we left out two simple interaction: The one with the user GPS from the User Application, and the one with the “OpenStreet” API from the Request Service, but normally, those interfaces works well if they are unit tested, so they don’t need specific interaction testing.

After all of those Interfaces are verified, we can put all together and verify the system in its entirety, because there are no link missing, so we don’t need to do intermediate tests.

3 Test case specification

3.1 Integration Tests

3.1.1 Integration Test I1

Test Case Identifier	I1
Test Item(s)	Request Service -> Queue
Input Specification	Create Typical “Request” Input
Output Specification	Check if the correct functions are called in “Queue”
Environmental Needs	User Client Driver

3.1.2 Integration Test I2

Test Case Identifier	I2
Test Item(s)	Driver Manager -> Queue
Input Specification	Create Typical “Driver Manager” Input
Output Specification	Check if the correct functions are called in “Queue”
Environmental Needs	Taxi Client Driver

3.1.3 Integration Test I3

Test Case Identifier	I3
Test Item(s)	Reservation Service -> Request Service
Input Specification	Create Typical “Reservation” Input
Output Specification	Check if the correct functions are called in “Request”
Environmental Needs	User Client Driver

3.1.4 Integration Test I4

Test Case Identifier	I4
Test Item(s)	Request Service -> Driver Manager
Input Specification	Create Typical “Request Service” Input
Output Specification	Check if the correct functions are called in “Driver Manager”
Environmental Needs	User Client Driver

3.1.5 Integration Test I5

Test Case Identifier	I5
Test Item(s)	Request Dispatcher -> Request Service
Input Specification	Create Typical “Request Dispatcher” Input
Output Specification	Check if the correct functions are called in “Request Service”
Environmental Needs	N/A

3.1.6 Integration Test I6

Test Case Identifier	I6
Test Item(s)	Request Dispatcher -> Reservation Service
Input Specification	Create Typical “Request Dispatcher” Input
Output Specification	Check if the correct functions are called in “Reservation Service”
Environmental Needs	N/A

3.1.7 Integration Test I7

Test Case Identifier	I7
Test Item(s)	Request Service -> Notification
Input Specification	Create Typical “Request Service” Input
Output Specification	Check if the correct functions are called in “Notification”
Environmental Needs	I5,I1 Succeeded , Taxi Client Driver

3.1.8 Integration Test I8

Test Case Identifier	I8
Test Item(s)	Request Service -> Driver Service
Input Specification	Create Typical “Request Service” Input
Output Specification	Check if the correct functions are called in “Driver Service”
Environmental Needs	I5,I1 Succeeded

3.1.9 Integration Test I9

Test Case Identifier	I9
Test Item(s)	Driver Service -> Driver Manager
Input Specification	Create Typical “Driver Service” Input
Output Specification	Check if the correct functions are called in “Driver Manager”
Environmental Needs	N/A

3.1.10 Integration Test I10

Test Case Identifier	I10
Test Item(s)	Driver Manager -> GPS Position Manager
Input Specification	Create Typical “Driver Manager” Input
Output Specification	Check if the correct functions are called in “GPS Position Manager”
Environmental Needs	I4 Succeeded

3.1.11 Integration Test I11

Test Case Identifier	I11
Test Item(s)	Request Service -> Database
Input Specification	Create Typical “Request Service” Input
Output Specification	Check if the correct functions are called in “Database”
Environmental Needs	I5 Succeeded

3.1.12 Integration Test I12

Test Case Identifier	I12
Test Item(s)	Reservation Service -> Database
Input Specification	Create Typical “Reservation Service” Input
Output Specification	Check if the correct functions are called in “Database”
Environmental Needs	I6 Succeeded

3.1.13 Integration Test I13

Test Case Identifier	I13
Test Item(s)	Driver Manager -> Database
Input Specification	Create Typical “Driver Manager” Input
Output Specification	Check if the correct functions are called in “Database”
Environmental Needs	I9 Succeeded

3.2 Integration Test Procedures

3.2.1 Integration Test Procedure TP1

Test Procedure Identifier	TP1
Purpose	This test verifies if the main software: • Can handle request from clients • Can manage queues after new requests incoming • Can send notification to client • Can handle taxi responses • Store this data to database
Procedure Steps	I5->I11->I1->I8->I7,

3.2.2 Integration Test Procedure TP2

Test Procedure Identifier	TP2
Purpose	This test verifies if: • Client is able to send reservations • If the system is able to instantiate a request when the reservation time is coming • Store this data to database
Procedure Steps	I6->I3->I12

3.2.3 Integration Test Procedure TP3

Test Procedure Identifier	TP3
Purpose	This test verifies if: • Taxi can change its status • System can manage queues after status changes • Store this data to database
Procedure Steps	I9->I2->I13

4. Tools and Test Equipment

As we said before, we are using NodeJS for the Core Application, AngularJS in the client side and MongoDB for the Database. There are many software specifically designed to test those type of applications. We are choosing “Mocha” for the NodeJS server, which is easy to use and efficient, “Jasmine” and “Protractor” for AngularJS. The Database is a Node component, so we can test it also with “Mocha”. We will also do manual testing to verify some special cases. Those framework are used also to do Unit-Testing.

The use of those frameworks are similar to “Mockito” for Java: we will create a test file for each of the Integration Tests and then we will run those in a specific order, as described in chapter 3.

To be more specific, there will be some “assert” commands to verify the returned value by the functions, but this is used mainly to do unit-testing. More important are the “it” functions that verifies the function responses, and the “beforeEach” and “afterEach” commands that do some stuff before and after the “it” control. This allows us to test the function simulating a person that is using the various functions.

We will use manual testing, instead, to verify the connection with the GPS in the User Client Application, because it is an external and already tested module. This works also for the OpenStreet API: we will test the interface manually.

5. Test Data Required

For the integration Test Procedure 1, we will build the following drivers

- **User Client** driver
- **Taxi Client** driver

The User Client driver will contain functions to simulate *reservations* (used for TP2) and *requests* from client. So, it basically sets the initial variables and simulate function calls on the different components. Same for the Taxi Client driver, but it will be used for testing *Change Status* and *Call* operations.

We don’t need any other drivers because of the specific order in which the tests are being executed, and we also don’t have much components to simulate.