Test Plan

Figure 1: Politecnico di Milano

Version 1.0

Relasese date : 21/01/2016

- Claudio Cardinale (mat. 849760)
- Gilles Dejaegere (mat. 853950)
- Massimo Dragano (mat. 775392)

Contents

1. Introduction

1.1. Revision history

Date	Version	Description	Authors
20/01/2016	1	Original Version	C. Cardinale, G. Dejaegere and M. Dragano

1.2. Purpose and Scope

The purpose of this document is to present to the testing team the sequence of tests to be applied to the different components (and their interfaces) forming the application. These components were of course designed during the design phase of the project development and are presented in the Design Document. These test are aimed at verify weither the components behave and cooperate correctly. This is done by testing the different components throught their interfaces. The tests explained in this document will have to be done in the correct order. The document also specifies for each test the eventual testing tools to be used as well as the eventuel additional stubs or mockups to use.

1.3. List of Definitions and abbreviations

- RASD: Requirements Analysis and Specifications Document
- DD: Design Document
- ITPD: Integration Test Plan Document
- Stub: fake data used to tests application, they are useful to populate database or model objects. Stubs are used to be called by the components actually tested.
- Drivers: drivers are like stubs with the difference that they are not used to be called by the component actually tested, but are use to call themselfs specific functions of the component actually tested.
- Mocks: stubs with the possibility of verifying wheter or not a specific method of this mock has been called a specific number of times. Mocks are therefore slightly more complex stubs. [**Gilles: I developped the notions of stubs, drivers and mocks. I hope I am not wrong, do not hesitate to comment ;)**]
- Unit test: the most famous way to perform tests via assertions
- Bottom-up : Bottom-up is an strategy of information processing. It is used in many different fields such as software or scientific theories. Regarding integration testing the bottom-up strategy consists in the integration of low level modules first and the integration of higher level modules after.

- Top-down : Top-down is an strategy of information processing. Regarding integration testing the top-down strategy consists in the integration of high level modules first and the integration of low level modules after. It is the opposite of bottom-up.
- Big-bang : Big-bang is an non-incremental integration strategy where all the components are integrated at once, right after they are all unit-tested.
- jMeter: java GUI program to measure the performance of a web server, it is developed by apache
- apache: open source software company
- laravel: is an php MVC framework
- php: is a programming language designed for the web

1.4. List of Reference Documents

- The MyTaxiService project description : "Project Description And Rules.pdf"
- The Assignment document: "Assignment 4 - integration test plan"
- The MyTaxiService RASD
- The MyTaxiService DD
- The example document given: "Integration Test Plan"
- Laravel testing: https://laravel.com/docs/5.1/testing

2. Integration Strategy

2.1. Entry Criteria

The following model classes must be unit tested before our integration tests.

- `Reservation`
- `Ride`
- `Driver`
- `Request`
- `Zone`

we should test all non-trivial methods. for instance:

- `Ride#close`: mark a ride as terminated
- **TODO**

getter and setter methods can be skipped.

2.2. Elements to be Integrated

QueueManager FIX her in the pdf, with this type of titles the next title doesn't go on a new line, take a look at below titles

addRequest

Throws an exception if there is no `Zone` containing the `Request` starting position.

Add a `Request` to the starting `Zone` requests queue.

removeRequest

Remove a `Request` from a `Zone` requests queue if present.

addDriver

Throws an exception if there is no `Zone` containing the `Driver` position. Throws an exception if the `Driver` is already enqueued (even if `Zone` is different).

Add a `Driver` to it's `Zone` drivers queue.

removeDriver

Remove a `Driver` from the `Zone` drivers queue if present.

peekDriverForZone

Thows an exception if there is no `Driver` in the `Zone` drivers queue.

return the first enqueued driver for a specified `Zone`

RequestConstroller create

Throws an exception if invalid arguments has been provided.

return a new `Request`.

DriverController login

Throws an exception if credentials are wrong.

On success it return some driver informations that can be used by the clients to build their `Driver` proxy.

setAvailable

Throws an exception if a `Ride` is in progress.

It changes the `Driver` availability and put it in the correct `Zone` when changing to `true`, using `QueueManager#addDriver`.

setPosition

Throws an exception if there is no `Zone` containing the specified position.

It set the `Driver` position to the given argument. It also closes the `Ride` if:

- there is an associated `Ride`
- the driver position matches the arrive one.

RideController create

Throws an exception if the client is not allowed to perform this action. Throws an exception if the given arguments are invalid.

Create a new `Ride` from a pending `Request`. Retrun the new `Ride`.

> NOTE: allowed clients are `Driver`s in the drivers queue of the `Zone` of the request.

2.3. Integration Testing Strategy

The sequence of integrations that will have to be applied on the components of this project mainly follows a bottom-up approach. This approach has many adventages : there is no need for stubs, the errors are more easily located (compared to strategies like the big-bang strategy) and, if the conception of the components also follows a bottom-up approach, the testing of lower level modules can take place simultaneously to the conception of higher level modules. Unfortunatly, this strategy also has its drawbacks : the integration needs drivers to be done, and even worse, the high level components are tested last, which

means that conception mistakes will be spotted later. However we still think that the adventages of the bottom-up strategies are more impacting that its drawbacks. In some cases such as for exemple inter-dependencies between two components, the use of a pure bottom-up approach will not be possible, and then a mixt of top down and bottom-up strategies will be used.

2.4. Sequence of Component/function Integration

2.4.1. Software Integration Sequence

2.4.1.1 The Model To be done

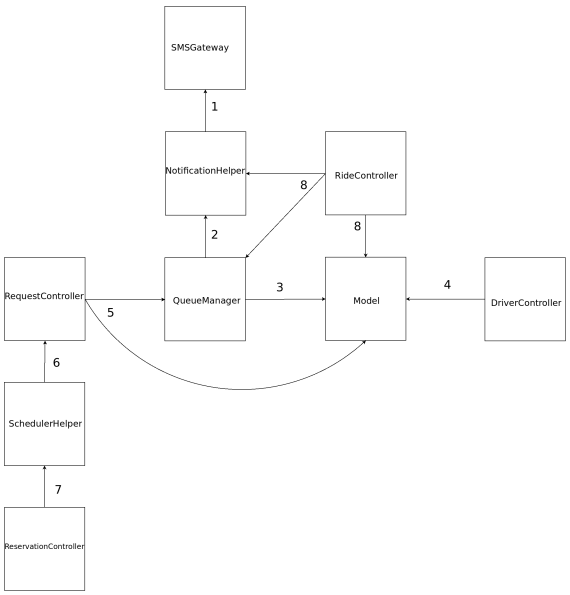

Figure 2: Controller integration sequence

2.4.1.2 The Controllers This diagramms shows that first the notification help is integrated to the sms gateway, then the queue is integrated to (smsgateway+notificationhelper) ... etc To be discussed, aproved, and then further develloped

2.4.2. Subsystem Integration Sequence

The MyTaxiService application designed is divided in different sub-systems. From the “High level components” figure (see DD pg 8) we can identify 4 subsystems :

- The central,
- The driver,
- The client,
- The database. Furthermore, the central can be devided in two sub-systems : the model and the controler (DD pg 8, Figure 5 : Component view).

The driver subsystem, the client subsystem and the database subsystem are atomic subsystems and are therefore not discussed in the section 2.4.1. In opposition, the controler and the model are composed of different subcomponents these subcomponents have to be integrated together. Concernig the order of integration of the subsystems, the model will be integrated to the controller at first. This will take place even before the subcomponents of the controller are all integrated together (see section 2.4.1**: the controller**[**to adapt**]). This is done because there are to many controller subcomponents interacting with the model. Once this integration is done, the database will be integrated, then the driver and finally the client. This can be seen on the following figure.

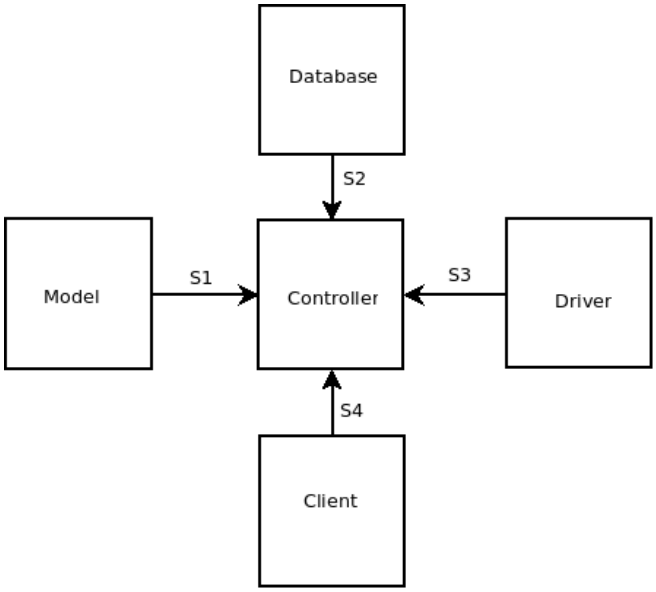

Figure 3: Subsystems integration

ID	Integration Tests	Sections
S1	Model –> Controler = Central	
S2	Database –> Central	
S3	Driver –> Central	
S4	Client –> Central	

Sections will have to be added once defined in section 3. If you guys know how to to nice arrows instead of "–>" please show me ;)

3. Individual Steps and Test Description

4. Tools and Test Equipment Required

Note: Since we said in the previous documents that we use laravel application (MVC php framework), we will use the laravel tests that extend PHPUnit tests. They are the same as Arquilan + jUnit (tests for JEE explained during the lessons)

We will create stub data to test application. Stub data are faker data used to populate the models and to have something to test.

4.1. Automatic tests

Since we want to test the entire application via integration tests, if it respects the requirements we decided to use laravel tests:

- **Laravel tests**: it is an extension to PHPUnit tests that add additional assertion and allow to emulate the entire client-server application. In fact you're able to test if a web page return the right body or the right HTTP status code, that is very useful in a pure restful application.
- **PHPUnit**: it is the standard php implementation of unit tests.
- **Unit test**: it is the most famous way to perform tests. In each test you have to make at least one assertion where you assert that two value are same, if it is false the test fails

So we create laravel tests like the following:

```
public function testApplication()
{
    $response = $this->call('POST', '/user', ['name' => 'Taylor']);

    $this->assertEquals(200, $response->status());
    $this->seeJson(['data' => 'data'])
}
```

4.2. Manual tests

We will test the entire system in a manual way to test:

- If the mobile/web applications are easy to use (user experience).
- If the localization of GPS works properly.
- **WRITE OTHERS**

Improve

Insert every word in glosary

4.3 Performance tests

We test the performance of the system like a blackbox, we test only the external API. In fact testing these we test all critical parts (in terms of performance). To do that obviously we need a lot of fake data on the server to simulate a critical situation.

We try to perform a huge amount of simultaneous requests and we measure the time needed to complete all requests (with static data like average, standard deviation and so on).

We decided to use jMeter that a powerful java program to do that (it is made by apache), but we can use also other tools like *ab* (another apache tool for server benchmark) that is very useful in some cases since it is a command line program.

5. Program Stubs and Test Data Required

We will insert fake data for taxis, clients and requests. We will add critical data tests like:

- All requests in the same zone
- All taxis in the same zone
- No data of a specif category (no taxis, no clients, ...)
- **WRITE OTHERS**

To generate fake data we will use the faker library and the seed function included with laravel, that allow us to populate easily database with fake data.

Improve

6. Used tools

- Github: for version controller
- Gedit and ReText: to write MarkDown with spell check
- Dia: to make figures

7. Hours of work

Claudio Cardinale

Gilles Dejaegere

Massimo Dragano