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 tests are aimed at verifying whether the components behave and cooperate correctly. This is done by testing the different components throughout 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 eventual 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: some codes emulating other functionalities or data, eventually using fake data.
- Drivers: drivers are like stubs with the difference that they are not used to be called by the component actually tested, but they are used to call themselves specific functions of the component actually tested. In this document the word driver is also used to design the “driver component” of our application. The distinction between the two should easily be done by the lector thanks to the context.
- Mocks: stubs with the possibility of verifying whether or not a specific method of this mock has called a specific number of times. Mocks are therefore slightly more complex stubs.
- Unit test: the most famous way to perform tests via assertions.
- Bottom-up: Bottom-up is a 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 a strategy of information processing. Regarding integration testing, the top-down strategy consists in the integration of high level modules first and the integration of lower level modules after. It is the opposite of bottom-up.
- Big-bang: Big-bang is a 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: it is a php MVC framework.
- php: it is a programming language designed for the web.
- SMS: short message service; it is the most famous way to send text messages to a mobile phone.
- push notification: the modern way to send complex messages to a smartphone

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`
- `SchedulerHelper`
- `QueueManager`

We should test all non-trivial methods. For instance:

- `Ride#close`: mark a ride as terminated
- `SchedulerHelper#addReservation`: manage `Reservation` scheduling correctly
- `QueueManager#addRequest`: manage `Request`s according to specifications

Getter and setter methods can be skipped.

2.2. Elements to be Integrated

As already mentionned before, the elements that will be integrated are nothing else than the components represented in the component view of our design document.

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 advantages: 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. Unfortunately, this strategy has also 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 advantages of the bottom-up strategies are more impacting that their drawbacks. In some cases such as, for example, inter-dependencies between two

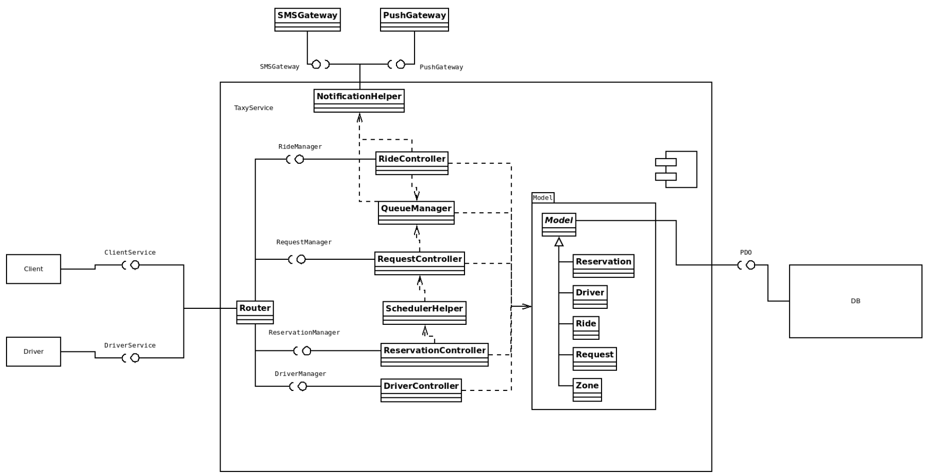

Figure 2: Components to be integrated

components, the usage of a pure bottom-up approach will not be possible, and then a mix of top down and bottom-up strategies will be used. This is the reason why some stubs are still needed.

2.4. Sequence of Component/function Integration

2.4.1. Software Integration Sequence

Here under, the integration sequence that will be applied can be seen. We can observer that the bottom-up has been respected in vast majority of the cases.

ID	Integration Test	Paragraphs
I1	NotificationHelper -> SMSGateway & PushGateway	3.1
I2	QueueManager -> Model	3.2
I3	QueueManager -> NotificationHelper	3.3
I4	DriverController -> Model	3.4
I5	RequestController -> Model & QueueManager	3.5
I6	SchedulerHelper -> RequestController	3.6
I7	ReservationController -> SchedulerHelper	3.7
I8	RideController -> Model & QueueManager & NotificationHelper	3.8

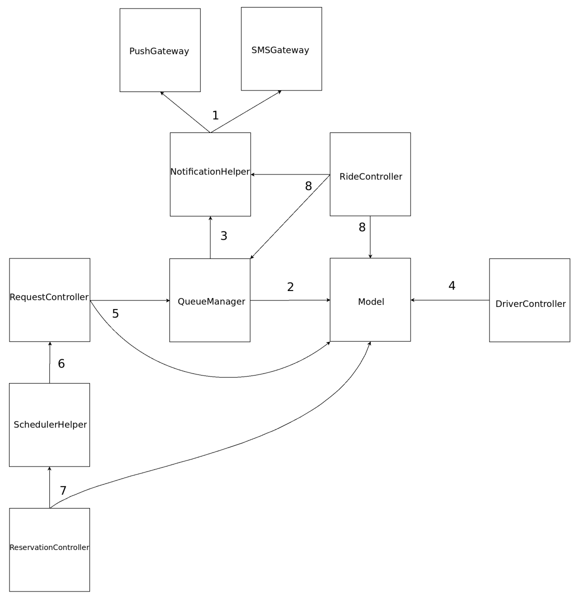

Figure 3: Controller integration sequence

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 sub-systems:

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

The driver subsystem, the client subsystem and the database subsystem are atomic sub-systems and therefore are not discussed in the section 2.4.1. In opposition, the controller and the model are composed of different subcomponents. On the other hand, their are no component interaction inside the model other than using a components "getter" or "setter". This means only that only the integration of the components of the controller subsystem will have to be tested. Concerning 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). This is done because there are too many controller subcomponents interacting with the model and therefore, the integration of a controller together will be done symultaneously with the integration of the other controllers. Once the controllers are fully integrated, the database will be integrated, then the driver and finally the client. This can be seen on the following figure.

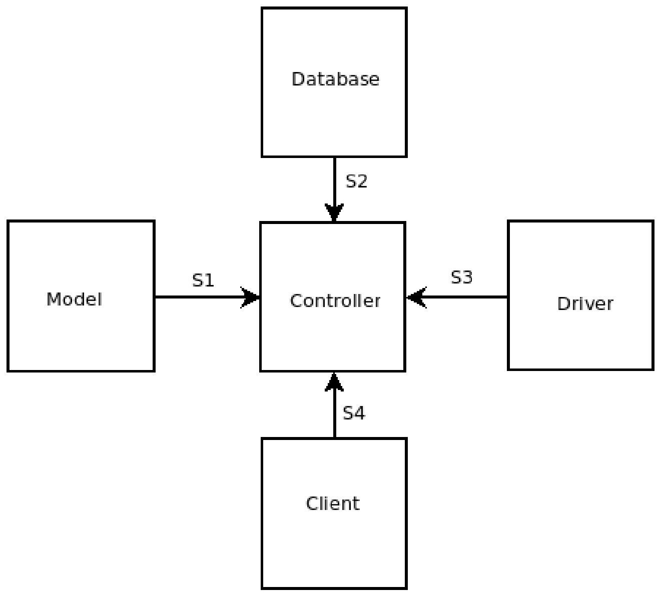

Figure 4: Subsystems integration

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

3. Individual Steps and Test Description

3.1. Integration test case I1

- **Test Case ID**: I1T1
- **Test Item(s)**: `NotificationHelper` -> `SMSGateway`
- **Input specification**: `Ride`, `Client`
- **Output specification**: A notification is sent as an SMS to the `SMSGateway`
- **Purpose**: Verify `NotificationHelper` and `SMSGateway` interaction
  - notify about a new `Ride` to the `Client`
- **Dependencies**: `SMSGateway` stub

- **Test Case ID**: I1T2
- **Test Item(s)**: `NotificationHelper` -> `PushGateway`
- **Input specification**: `Request`, `Driver`
- **Output specification**: A push notification is sent to the `PushGateway`
- **Purpose**: Verify `NotificationHelper` and `PushGateway` interaction
  - notify about a new `Request` to the `Driver`
- **Dependencies**: `PushGateway` stub

3.2. Integration test case I2

- **Test Case ID**: I2T1
- **Test Item(s)**: `QueueManager` -> `Zone`
- **Input specification**: `Request` and `Zone`
- **Output specification**: `Zone` is managed in the correct way.
- **Purpose**: Verify `QueueManager` and `Zone` interaction
  - add `Request`s to the correct `Zone`
  - remove `Request`s from the correct `Zone`
- **Dependencies**: N/A

3.3. Integration test case I3

- **Test Case ID**: I3T1
- **Test Item(s)**: `QueueManager` -> `NotificationHelper`
- **Input specification**: `Request` and `Driver`

- **Output specification**: A notification is sent to the `Driver`
- **Purpose**: Verify `QueueManager` and `NotificationHelper` interaction
  - notify about a new `Request` to the first available `Driver`
  - if no answer, or a negative answer arrives from the Driver (a driver will be used to simulate the answers, see section 5), the QueueManager should put the concerned `Driver` at the end of its queue and ask the following `Driver`
- **Dependencies**: `ClientDriver`, `PushGateway` stub

3.4. Integration test case I4

- **Test Case ID**: I4T1
- **Test Item(s)**: `DriverController` -> `Model`
- **Input specification**: `Driver`, `Ride`
- **Output specification**: `Driver` and `Ride` are managed in the correct way.
- **Purpose**: Verify `DriverController` and `Model` interaction
  - set `Driver` position
  - close a `Ride` when `Driver` reach the arrive
  - set `Driver` availability
  - get a `Driver` from it's authentication credentials
- **Dependencies**: `ClientDriver`

3.5. Integration test case I5

- **Test Case ID**: I5T1
- **Test Item(s)**: `RequestController` -> `QueueManager`
- **Input specification**: `Request`
- **Output specification**: `Request` are sent to `QueueManager`
- **Purpose**: Verify `RequestController` and `QueueManager` interaction
  - enqueue a created `Request`
- **Dependencies**: `ClientDriver`

3.6. Integration test case I6

- **Test Case ID**: I6T1
- **Test Item(s)**: `SchedulerHelper` -> `RequestController`
- **Input specification**: `Reservation`
- **Output specification**: `SchedulerHelper` built `Request`s are sent to the `RequestController`

- **Purpose**: Verify `SchedulerHelper` and `RequestController` interaction
  - send `Requests` to the `RequestController` when fired
- **Dependencies**: N/A

3.7. Integration test case I7

- **Test Case ID**: I7T1
- **Test Item(s)**: `ReservationController` -> `SchedulerHelper`, `Model`
- **Input specification**: `Reservation`
- **Output specification**: `Reservation`s are sent to the `SchedulerHelper`
- **Purpose**: Verify `ReservationController` behavior
  - create a `Reservation`
  - delete a `Reservation`
  - notify new `Reservation`s to the `SchedulerHelper`
  - notify deleted `Reservation`s to the `SchedulerHelper`
- **Dependencies**: `ClientDriver`

- **Test Case ID**: I7T2
- **Test Item(s)**: `ReservationController` -> `SchedulerHelper`, `Model`
- **Input specification**: Compatible and incompatible `Reservation`s with and without the shared-ride option activated
- **Output specification**: The right `Reservation`s are sent to the `SchedulerHelper`.
- **Purpose**: Verify `ReservationController` behavior concerning merged requests.
  - merge compatible reservations with the shared-ride option activated
  - does not merge incompatible reservations
  - does not merge reservations without the shared-ride option activated
- **Dependencies**: `ClientDriver`

3.8. Integration test case I8

- **Test Case ID**: I8T1
- **Test Item(s)**: `RideController` -> `Model`
- **Input specification**: `Ride`
- **Output specification**: `Ride`s are managed as expected
- **Purpose**: Verify `RideController` and `Model` interaction

  - create a `Ride`
  - delete created `Ride`'s parent `Request`
- **Dependencies**: `ClientDriver`

- **Test Case ID**: I8T2
- **Test Item(s)**: `RideController` -> `QueueManager`
- **Input specification**: `Ride`
- **Output specification**: parent `Request` are pulled out from queue
- **Purpose**: Verify `RideController` and `QueueManager` interaction
  - remove created `Ride` parent `Request` from queue
- **Dependencies**: `ClientDriver`

- **Test Case ID**: I8T3
- **Test Item(s)**: `RideController` -> `NotificationHelper`
- **Input specification**: `Ride`
- **Output specification**: send `Notification`s as expected
- **Purpose**: Verify `RideController` and `NotificationHelper` interaction
  - send a `Notification` to the `Client` when it's `Request` is accepted
- **Dependencies**: `ClientDriver`

4. Tools and Test Equipment Required

Note: Since in the previous documents we said 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), but for php + laravel.

We will create fake data to test application. Fake data are used to populate the models and to have something to test.

4.1. Automatic tests

Since we wanted 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 adds additional assertions and allows to emulate the entire client-server application. In fact you're able to test if a web page returns the right body or the right HTTP status code, 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 values are the same, if it is false the test fails.

So we created laravel tests like the following ones:

```php
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.

Note: we will use the devices and systems defined on RASD section 1.6.2

4.3 Performance tests

We test the performance of the system like a blackbox, we test only the external APIs. In fact testing them, we test all the critical parts (in terms of performance). To do that we obviously 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 the requests (with static data like average, standard deviation and so on).

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

Insert every word in glosary

5. Program Stubs and Test Data Required

5.1 Stubs and drivers

We only have 2 stubs since we decided to use bottom-up. **FIX THE NUMBER WHEN WE TAKE A DECISION ABOUT CLIENTDRIVER**

5.1.1. SMS gateway

Usages

- I1T1

Description This stub allows to test the SMS functionalities, emulating the external gateway. This is done for the following two reasons:

- Cost: reduce the cost of tests (it doesn't send real SMS)
- Easy to test: in this way it is an easy test functionality, in fact there are no network problems (the *send* return always OK) and the stub offers easy methods to see the text of messages sent

5.1.2. Push gateway

Usages

- I1T2
- I3T1

Description This stub allows to test the push notification functionalities, emulating the external gateway. This is done for the following two reasons:

- Cost: reduce the cost of tests (a big amount of notifications must be paid)
- Easy to test: in this way it is an easy test functionality, in fact there are no network problems (the *send* return always OK) and the stub offers easy methods to see the text of notification sent

5.1.3. ClientDriver

Usages

- I3T1

- I4T1
- I5T1
- I7T1
- I7T2
- I8T1
- I8T2
- I8T3

Description This stub allows to the driver application to emulate different things:

- it emulates the mobile application as a restful client performing requests, this is done via laravel tests
- it emulates the push notification service gateway needed to send push notifications to a specific mobile device, this stub allows to emulate it in the same way of the emulation of SMS gwateway. So there are no network problems and it is easy to assert via tests that the text of the notification sent.

TODO AGAIN Uses notifications EMULATES GPS

5.2 Data for tests

We will insert fake data for taxis, clients, requests and other entities to populate the database. To generate them we will use the faker library and the seed function included with laravel which allows us to populate easily database with fake data.

5.3 Critical data tests

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, . . .)

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