Politecnico di Milano

Software Engineering II

myTaxiService Integration Test Plan Document

Authors

Giovanni Beri	852984
Biagio Festa	841988

Tuesday 19th January, 2016

Contents

1 Introduction

1.1 Purpose and Scope

The *Integration Test Plan Document* aims at describing how you plan to accomplish the integration test. This document is supposed to be written before the the integration test really happens, and provides a schedule and an high-level description of all the various integration test that will be performed. More in detail, this document will provide:

- the identification of the software components that are going to be integrated, with references to the *Design Document.*
- the description of the various subsystem in which *myTaxiService* system is split.
- the schedule and description of the expected integration tests.
- the description of the used test tool.

1.2 List of Definitions and Abbreviations

Software *or* System
The collection and the parts of programmes that compose the project. Also we will indicate with these words the components finalized at the user (*documentation* and *user interface* application).

Actor
An entity that interacts with the system. It can be either a physical user or a external component.

Taxi zones
Geographical areas of approximately $2km^2$ in which the interested city is divided. Taxi zones are not overlapping, so each point of the city is assigned to exactly one taxi zone.

Ride request
A request that is sent by a registered user to signal that he needs to be picked up by a taxi in the immediate future. The request includes the GPS coordinates of the position in which the registered user desires to be picked up.

Ride reservation
A request that an user sends to book a taxi ride in the "future". For "future" we intend an instant of time that is almost 2 hours distant from the instant in which the request is sent. It must contain origin and destination of the ride and picking up time and date.

Fake request
A Ride reservation or ride request that turns out to be not a real one, because the passenger who sent it did not show up in the indicated place (and, in the case of a ride reservation, at the indicated time and date)

ETA

The *estimated time of arrival.* It can be used to generate estimated times of arrival depending on either a static timetable or through measurements on traffic intensity. [1]

API

Application Programming Interface: is a set of routines, protocols, and tools for building software applications. [2]

[1]Wikipedia definition: `https://en.wikipedia.org/wiki/Estimated_time_of_arrival`

[2]Wikipedia definition: `https://en.wikipedia.org/wiki/Application_programming_interface`

2 Integration Strategy

2.1 Entry Criteria

We now define some entry criteria, *i.e.* some conditions that must hold when the integration test begin, in order to ensure the feasibility and correctness of the integration tests themselves.

- The *Requirement Analysis and Specification Document* and *Design Document* must be and complete, and must reflects the current state of the *myTaxiService* project.
- Every tested component must be code-complete *i.e.* all functionalities that are going to be tested, have already been completely implemented.
- Analyzed components are unit-tested (*i.e. a unit test has already been performed*) and bug free.
- Integration Test datasets have already been selected.
- Manual tester that are going to play the role of end users have already been selected.

NB: we suppose that the communication with some of the external component with which our system is going to interact (in particular, those components that *are used* by *myTaxiService*, and note vice versa) has already been tested during unit test, so it's not necessary to test their integration with our system. We assume that because the interaction with those external system is essential for the correct functioning of some of *myTaxiService*'s component, so it's not practicable to perform a unit test on those components without also testing the integration with the external component. In particular, the already tested integration are:

- *DBMS* with *Account Data Layer*, *Reservation Data Layer* and *CityZone-Data Layer*
- *Google Cloud Messaging* with *myTaxiUser App Communication Manager*, *myTaxiDriver App Communication Manager*, *myTaxiUSer App Façade* and *myTaxiDriver App Façade*
- *Google Maps* with *myTaxiUser App Presentation Layer* and *myTaxiDriver App Presentation Layer*
- *Google Places* with *myTaxiUser App Presentation Layer*

2.2 Elements to be integrated

In this paragraph we'll identify the software elements among which the integration test will be performed, by referring to the low-level components specified at paragraph 2.2 of *Design Document*. In the diagram below, all software components are listed, and they are linked by «used by» relation, represented by the dashed arcs. Note that in this situation the «used by» relationship is equivalent to the inverse of «depends from» relationship, so the nodes without incoming arcs are those that doesn't depend from any other component.

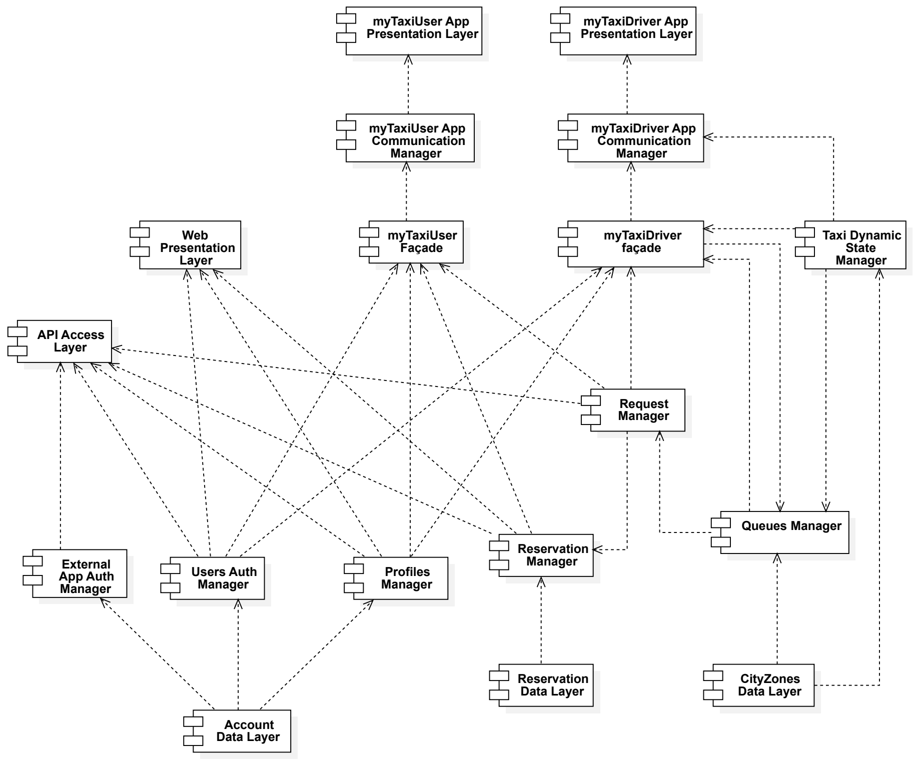


Figure 1: Elements to be integrated

The «used by» relation has been built starting from the low-level Component Diagram of *Design Document*, by replacing all links between components with an arc, with tail at the *offered interface* side, and head at the *received interface* side.

2.3 Integration Test Strategy

Server Core

For our server integration testing, we decided to use a ***bottom-up*** approach, including the creation of *Drivers*. We evaluated also the *top-down* and *sandwich* approaches, but we rejected them for the following reasons:

- ***Top-down* strategy**: this kind of strategy is suggested in case of timing problems, because it's focused on testing more the functionalities provided to the end-user than the real functioning of all low-level components of the system. We do not have any information about *myTaxiService* deadlines, so we suppose not to have any timing problem.
- ***Sandwich* strategy**: this kind of strategy is more difficult to plan and implement in case of high-interdependent systems, and the server is the most complex part of our system.

Client applications

Since the two clients are definitely strictly correlated between them, we've thought to plan a test on clients which is based on the functionalities *i.e.* thread integration test. Moreover the clients are essentially two very small sub-systems and for that reason the bottom-up approach could be a useless waste of time. The fact is that we really want to guarantee is that those functionalities work well and that two clients can correctly communicate between them through the server.

2.4 Sequence of Components

We have identified *three* different subsystems in the organization of our tests.

***myTaxiService* Server and WebApp**

It represents the entire business logic of the entire system. It's the core of the project and works in order to create a communication channel among different clients. Inside itself the taxi queues, reservations, ride request are handled. It manages the API system also.

The web application (browser based) is also embedded in that subsystem.

Mobile Client *myTaxiUser*

It's the main client for a passenger (smartphone app based). It allows to access to the functionalities of the system client side. We've decided to separate this subsystem from the principal one because the client server communication is a fundamental aspect for the operation mode of the entire project. Indeed we want to be sure (as more as possible) that the communication works properly.

Mobile Client *myTaxiDriver*

It's the main client for a taxi driver (smartphone app based). It allows to access to the functionalities of the system client side. The reasons which led us to split that subsystem are the same to the other client.

2.4.1 Server Core

In the figure 2 we can see a representation of the *bottom-up* approach. In that section we're going to analyze the testing planning about the core of the system which we've defined as *server*.

We've identified *six different independent layers* which can be seen on the figure in different colors. Every component in a low level has got a kind of dependencies with some others in a upper level. That kind of representation allows us to identify well which components can be tested before.

Thank to that representation we can easily see what kind of *parallelization* is possible. Indeed each component in a level is *independent* from the others which are in the same level.

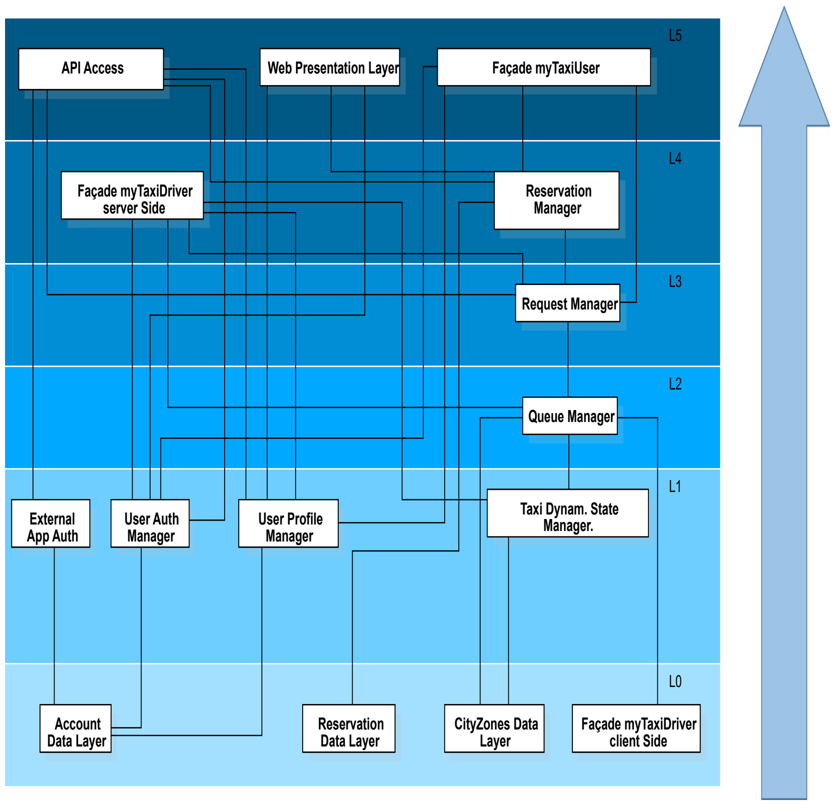


Figure 2: Bottom-Up Approach Representation.

Loop Dependency At the moment to define a sequence of testing plan we've noticed a cyclic dependency between two components. From the figure 1 we've pointed out that dependency in the figure 3.

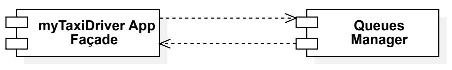
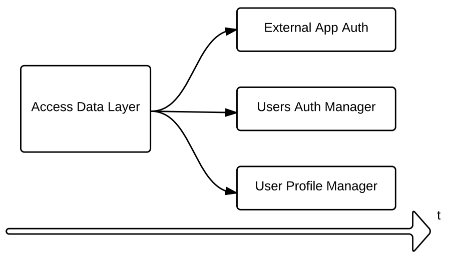


Figure 3: Loop in the component view.

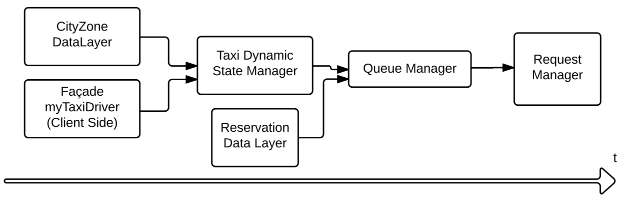
That dependency is because both components need to send asynchronous messages in order to request services to the other one. Indeed the *taxi driver client* both asks for a service but provides services also.

At testing layer, we've solved the dependency problem splitting the component «*Façade myTaxiDriver*» into two different ones. Each sub-component exports and provides functionalities according to the side with which it interfaces. In that way the component is tested following a functionalities approach.

Parallelization Approach Moreover we can identify two different *branches* which are very independent between them and so they're easily *parallelizable*. We can see them in the figure 4.



(a)



(b)

Figure 4: Testing parallel branches.

We suggest to assign two different teams in a way that they can easily work separately on those different branches and speed up the entire testing task.

Anyway there is an important aspect that has to be considered. Although it strongly *depends by the implementation*, we imagine the second branch (figure 4b) will take more time than the other one. That because the second branch is composed by more components which are more complex also (in according with the design document). For that reason when a team has completed the planning test, it can easily merge with the other one in order to complete the entire work and go on the next layer in the schema.

2.4.2 myTaxiService Clients

In the figure 5 we can see the 2 clients' components.

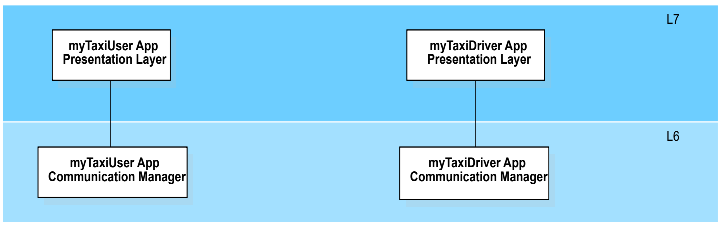

L7
myTaxiUser App Presentation Layer
myTaxiDriver App Presentation Layer
L6
myTaxiUser App Communication Manager
myTaxiDriver App Communication Manager

Figure 5: Clients Component Testing View

Once the server has been tested in each of its parts, we want to integrate the two different clients and test whether they can communicate exchanging the services the entire system is built for. As sketched in paragraph 2.3, for the client applications integration tests we are going to perform a thread-based integration test, also because *myTaxiService* clients are thin clients so, once the server core has been tested, we can assume that the business logic is correct. So we are now going to check if the functionalities that the system is expected to furnish are actually provided to the users. So, the integration test of client applications will consist in the systematic check of the presence and correctness of all the functionalities listed in the RASD [4] document.

3 Individual Step and Test Description

In this paragraph we are going to give an high-level description of the integration tests that will be performed on *myTaxiService* system.
The *bottom-up* strategy choice implies the use of drivers: every driver is associated to a specific software component, and is used to simulate the input coming from a non-tested component. The following tables are structured as:

- **Test Case ID**: an identifier for the analyzed integration test.
- **Test Items**: the couple of involved components; the arrow represent the «used by» relationship.
- **Input Specification**: the high-level description of the input coming from the right element of the couple specified at *Test Item* row.
- **Output Specification**: the high-level description of the expected output of the input above.
- **Environmental Needs**: the conditions under which the test can be executed, in terms of:
  - already performed tests
  - already implemented drivers

In our integration test, we decided to create a driver for each needed component (instead of a macro-driver that groups all the components that use the analyzed component) because, in this way, it would be easier to find the errors in case some bugs appears; furthermore, drivers associated to a single component can be reused if some components use more than 1 other component.

3.1 Server Core SubSystem

3.1.1 *Accounts Data Layer* integration test

Test Case ID	I1T1
Test Item	Account Data Layer → External App Auth Manager
Input Specification	Simulate *Account Data Layer* component typical input coming from *External App Auth Manager*, paying attention to cover the exceptional and edge cases related to the *Access Account* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *External App Auth Manager* driver must have been already implemented

Test Case ID	I1T2
Test Item	Account Data Layer → Users Auth Manager
Input Specification	Simulate *Account Data Layer* component typical input coming from *Users Auth Manager*, paying attention to cover the exceptional and edge cases related to the *Access Account* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Users Auth Manager* driver must have been already implemented

Test Case ID	I1T3
Test Item	Account Data Layer → Profiles Manager
Input Specification	Simulate *Account Data Layer* component typical input coming from *Profiles Manager*, paying attention to cover the exceptional and edge cases related to the *Access Account* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Profiles Manager* driver must have been already implemented

3.1.2 *Reservation Data Layer* integration test

Test Case ID	I2T1
Test Item	Reservation Data Layer → Reservation Manager
Input Specification	Simulate *Reservation Data Layer* component typical input coming from *Reservation Manager*, paying attention to cover the exceptional and edge cases related to the *AccessReservationData* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Reservation Manager* driver must have been already implemented

3.1.3 *CityZone Data Layer* integration test

Test Case ID	I3T1
Test Item	CityZone Data Layer → Queues Manager
Input Specification	Simulate *CityZone Data Layer* component typical input coming from *Queues Manager*, paying attention to cover the exceptional and edge cases related to the *AccessCityZone* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Queues Manager* driver must have been already implemented

Test Case ID	I3T2
Test Item	CityZone Data Layer → Taxi Dynamic State Manager
Input Specification	Simulate *CityZone Data Layer* component typical input coming from *Taxi Dynamic State Manager*, paying attention to cover the exceptional and edge cases related to the *AccessCityZone* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Taxi Dynamic State Manager* driver must have been already implemented

3.1.4 *Façade myTaxiDriver-clientSide* integration test

Test Case ID	I4T1
Test Item	Façade myTaxiDriver-clientSide → Queues Manager
Input Specification	Simulate *Façade myTaxiDriver-clientSide* component typical input coming from *Queues Manager*, paying attention to cover the exceptional and edge cases related to the *AskForRide* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Queues Manager* driver must have been already implemented

3.1.5 *External App Auth Manager* integration test

Test Case ID	I5T1
Test Item	External App Auth → API Access Layer
Input Specification	Simulate *External App Auth* component typical input coming from *API Access Layer*, paying attention to cover the exceptional and edge cases related to the *ExternalAppLogin* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *API Access Layer* driver must have been already implemented **AND** I1T1 must have already been performed

3.1.6 *User Auth Manager* integration test

Test Case ID	I6T1
Test Item	User Auth Manager → Façade myTaxiDriver-serverSide
Input Specification	Simulate *User Auth Manager* component typical input coming from *Façade myTaxiDriver-serverSide*, paying attention to cover the exceptional and edge cases related to the *UserLogin* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade myTaxiDriver-serverSide* driver must have been already implemented **AND** I1T2 must have already been performed

Test Case ID	I6T2
Test Item	User Auth Manager → Web Presentation Layer
Input Specification	Simulate *User Auth Manager* component typical input coming from *Web Presentation Layer*, paying attention to cover the exceptional and edge cases related to the *UserLogin* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Web Presentation Layer* driver must have been already implemented **AND** I1T2 must have already been performed

Test Case ID	I6T3
Test Item	User Auth Manager → Façade myTaxiUser
Input Specification	Simulate *User Auth Manager* component typical input coming from *Façade myTaxiUser*, paying attention to cover the exceptional and edge cases related to the *UserLogin* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade myTaxiUser* driver must have been already implemented **AND** I1T2 must have already been performed

Test Case ID	I6T4
Test Item	User Auth Manager → API Access Manager
Input Specification	Simulate *User Auth Manager* component typical input coming from *API Access Manager*, paying attention to cover the exceptional and edge cases related to the *UserLogin* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *API Access Manager* driver must have been already implemented **AND** I1T2 must have already been performed

3.1.7 *User Profile Manager* integration test

Test Case ID	I7T1
Test Item	User Profile Manager → API Access Manager
Input Specification	Simulate *User Auth Manager* component typical input coming from *API Access Manager*, paying attention to cover the exceptional and edge cases related to the *ModifyProfileData* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *API Access Manager* driver must have been already implemented **AND** I1T3 must have already been performed

Test Case ID	I7T2
Test Item	User Profile Manager → Web Presentation Layer
Input Specification	Simulate *User Auth Manager* component typical input coming from *Web Presentation Layer*, paying attention to cover the exceptional and edge cases related to the *ModifyProfileData* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Web Presentation Layer* driver must have been already implemented **AND** I1T3 must have already been performed

Test Case ID	I7T3
Test Item	User Profile Manager → Façade mytaxiDriver-serverSide
Input Specification	Simulate *User Auth Manager* component typical input coming from *Façade mytaxiDriver-serverSide*, paying attention to cover the exceptional and edge cases related to the *ModifyProfileData* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade mytaxiDriver-serverSide* driver must have been already implemented **AND** I1T3 must have already been performed

Test Case ID	I7T4
Test Item	User Profile Manager → Façade myTaxiUser
Input Specification	Simulate *User Auth Manager* component typical input coming from *Façade myTaxiUser*, paying attention to cover the exceptional and edge cases related to the *ModifyProfileData* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade mytaxiDriver-serverSide* driver must have been already implemented **AND** I1T3 must have already been performed

3.1.8 *Taxi Dynamic State Manager*

Test Case ID	I8T1
Test Item	Taxi Dynamic State Manager → Queues Manager
Input Specification	Simulate *Taxi Dynamic State Manager* component typical input coming from *Queues manager*, paying attention to cover the exceptional and edge cases related to the *Access State* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Queues Manager* driver must have been already implemented **AND** I3T2 must have already been performed

Test Case ID	I8T2
Test Item	Taxi Dynamic State Manager → myTaxiDriver App Communication Manager
Input Specification	Simulate *Taxi Dynamic State Manager* component typical input coming from *myTaxiDriver App Communication Manager*, paying attention to cover the exceptional and edge cases related to the *myTaxiDrivers coordinates updater* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *myTaxiDriver App Communication Manager* driver must have been already implemented **AND** I3T2 must have already been performed

Test Case ID	I8T3
Test Item	Taxi Dynamic State Manager → myTaxiDriver Façade-serverSide
Input Specification	Simulate *Taxi Dynamic State Manager* component typical input coming from *myTaxiDriver Façade-serverSide*, paying attention to cover the exceptional and edge cases related to the *Access State* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *myTaxiDriver Façade-serverSide* driver must have been already implemented **AND** I3T2 must have already been performed

3.1.9 *Queues Manager* integration test

Test Case ID	I9T1
Test Item	Queues Manager → myTaxiDriver Façade-serverSide
Input Specification	Simulate *Queues Manager* component typical input coming from *myTaxiDriver Façade-serverSide*, paying attention to cover the exceptional and edge cases related to the *Get myTaxiDrivers Distribution* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *myTaxiDriver Façade-serverSide* driver must have been already implemented **AND** I3T1 and I4T1 must have already been performed

Test Case ID	I9T2
Test Item	Queues Manager → Request Manager
Input Specification	Simulate *Queues Manager* component typical input coming from *Request Manager*, paying attention to cover the exceptional and edge cases related to the *Driver Asssigner* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Request Manager* driver must have been already implemented **AND** I3T1 and I4T1 must have already been performed

3.1.10 *Request Manager* integration test

Test Case ID	I10T1
Test Item	Request Manager → API Access
Input Specification	Simulate *Request Manager* component typical input coming from *API Access*, paying attention to cover the exceptional and edge cases related to the *Make and Cancel RideRequestes* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *API Access* driver must have been already implemented **AND** I9T2 must have already been performed

Test Case ID	I10T2
Test Item	Request Manager → Façade myTaxiDriver-server Side
Input Specification	Simulate *Request Manager* component typical input coming from *Façade myTaxiDriver-server Side*, paying attention to cover the exceptional and edge cases related to the *Update Ride State* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade myTaxiDriver-server Side* driver must have been already implemented **AND** I9T2 must have already been performed

Test Case ID	I10T3
Test Item	Request Manager → Reservation Manager
Input Specification	Simulate *Request Manager* component typical input coming from *Reservation Manager*, paying attention to cover the exceptional and edge cases related to the *Reservation Activation* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Reservation Manager* driver must have been already implemented **AND** I9T2 must have already been performed

Test Case ID	I10T4
Test Item	Request Manager → Façade myTaxiUser
Input Specification	Simulate *Request Manager* component typical input coming from *Façade myTaxiUser*, paying attention to cover the exceptional and edge cases related to the *Make and Cancel RideRequestes* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade myTaxiUser* driver must have been already implemented **AND** I9T2 must have already been performed

3.1.11 *Façade myTaxiDriver-serverSide* integration test

Test Case ID	I11T1
Test Item	Façade myTaxiDriver-serverSide → my-TaxiUSer App Communication Manager
Input Specification	Simulate *Façade myTaxiDriver-serverSide* component typical input coming from *myTaxiUSer App Communication Manager*, paying attention to cover the exceptional and edge cases related to the *RemoteInterfaceCommunicationDriver* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *myTaxiUSer App Communication Manager* driver must have been already implemented **AND** I6T1, I7T3, I10T2 must have already been performed

3.1.12 *Reservation Manager* integration test

Test Case ID	I12T1
Test Item	Reservation Manager → API Access
Input Specification	Simulate *Reservation Manager* component typical input coming from *API Access*, paying attention to cover the exceptional and edge cases related to the *AccessRideReservation* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *API Access* driver must have been already implemented **AND** I2T1 and I10T3 must have already been performed

Test Case ID	I12T2
Test Item	Reservation Manager → Web Presentation Layer
Input Specification	Simulate *Reservation Manager* component typical input coming from *Web Presentation Layer*, paying attention to cover the exceptional and edge cases related to the *AccessRideReservation* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Web Presentation Layer* driver must have been already implemented **AND** I2T1 and I10T3 must have already been performed

Test Case ID	I12T3
Test Item	Reservation Manager → Façade myTaxiUser
Input Specification	Simulate *Reservation Manager* component typical input coming from , paying attention to cover the exceptional and edge cases related to the *AccessRideReservation* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *Façade myTaxiUser* driver must have been already implemented **AND** I2T1 and I10T3 must have already been performed

3.1.13 *API Access Layer* integration test

In this particular case, the driver will simulate a generic external application, sending request through the *API Access* interface; this driver is named *External Application* driver.

Test Case ID	I13T1
Test Item	API Access Layer → Generic External Application
Input Specification	Simulate *API Access Layer* component typical input coming from , paying attention to cover the exceptional and edge cases related to the *API Access* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *External Application* driver must have been already implemented **AND** I5T1, I6T4, I7T1, I10T1, I12T1 must have already been performed

3.1.14 *Web Presentation Layer* integration test

n this particular case, a different kind of test is going to be performed. This component is going to interact with browsers, that are usually well tested programs; so, instead of building a driver that simulates the typical browser input, this component will be integrated through manual testing performed via browser (*e.g.* Google Chrome).

3.1.15 *Façade myTaxiUser* integration test

Test Case ID	I14T1
Test Item	Façade myTaxiUser → myTaxiDriver App Presentation Layer
Input Specification	Simulate *Façade myTaxiUser* component typical input coming from *myTaxiDriver App Presentation Layer*, paying attention to cover the exceptional and edge cases related to the *RemoteInterfaceCommunicationUser* interface
Output Specification	Check if the correct method are invoked, along with the correct parameter types and values
Environmental Needs	The *myTaxiDriver App Presentation Layer* driver must have been already implemented **AND** I6T3, I7T4, I12T3, I10T4 must have already been performed

4 Tools and Test Equipment Required

Since we've divided the entire system in three different sub-systems, a good starting point could be to make an individual analysis for each sub-system.

The core of the system, as we have already described in the section 2.4.1, is the server in which all the business logic is concentrated. Indeed this is the principal subsystem of the entire system. Moreover the server is the one which is composed by the major number of components and so it's definitely more complex to test.

For those reasons we think that a testing through *Arquillian* is fundamental in such sense. Indeed Arquillian represents an optimal solution in a such context where more components have to be tested and integrated among them.

The main aspect is also decided by the particular platform with which the system will be developed. In accordance with the design document [3], *Java EE* will be the main framework for the development of that system. That's why Arquillian should be the best solution in these case in which the integration among *multiple enterprise components* has to be tested.

> *"Ever since the inception of Java EE, testing enterprise applications has been a major pain point. Testing business components, in particular, can be very challenging. Often, a vanilla unit test isn't sufficient for validating such a component's behavior. Why is that? The reason is that components in an enterprise application rarely perform operations which are strictly self-contained. Instead, they interact with or provide services for the greater system. They also have declarative functionality which gets applied at runtime. You could say no business component is an island."* [2]

4.1 Operational Testing Environment

We demand that each test will be run on different operational environments. That's because we want to try to reduce the probabilities that system could give errors in deployment phase.

Obviously we expect that the integration test will be done in a development environment in way that it will be easier for the programmers fix any errors. For that reason the architecture will be much different than one which will be present in deployment phase.

A common desktop development environment we expect is one composed by:

Java EE 7
: As described in the design document [3].

GlassFish server Open Source Edition
: Version 4.0.

Eclipse (*or* Netbeans)
: The *IDE* for source development.

Database
: *Oracle RDBMS (version 12.1.0.2).* We expect a dedicated machine will

run oracle DBMS with an appropriate licence in order to execute local communication testing with the database.

Client Browser
For web-application testing. We suggest: *Google Chrome Browser (version 47.0)* and *Mozilla Firefox (version 43.0.4)*.

Mobile Client
For mobile app client testing: *Android (version 4.0 (*Ice Cream Sandwich*) or higher). iOS (version 7.1.2 or higher).*

5 Special Test Data and Configurations

A particular note on the *database configuration.* Indeed in order to perform an *operational test* is quite important that the database is configured properly. A script should be created to perform an *population operation* into the database itself.

An important aspect in our testing planning is the messages exchanged between clients and server. In the design document has been explained the usage of the service called *Google Cloud Messaging.* As the document says [3], "*this service allow us to implement a system based on push notification in a useful and practical way*".

That component is quite important in the architectural structure because it allows the remote nodes to communicate among them.

At the final phase, when we need to integrate all the three sub-systems, the Google service could be not available at this moment (for business or technical reasons). That's why in the test phase becomes very important to build an appropriate *drivers* and *stubs* which allow to emulate that service.

Something similar applies to *Google Maps API.* But in that case it's easier because the issue concerns just a single component and does not affect in the integration test.

The last important aspect is the testing about API components. In order to execute an appropriate integration with the system we need to build a proper *driver* which allows to interact with the *API interface.*

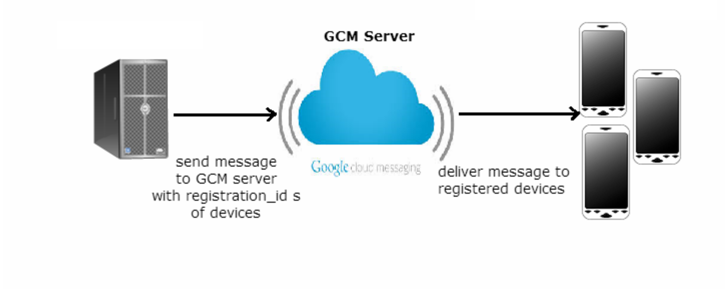

GCM Server
send message to GCM server with registration_id s of devices
Google cloud messaging
deliver message to registered devices

Figure 6: Google Cloud Messaging.

References

[1] Martin Fowler,
Mocks Aren't Stubs,
`http://martinfowler.com/articles/mocksArentStubs.html`, 2007.

[2] Dan Allen, Aslak Knutsen, Pete Muir, and Andrew Rubinger,
Arquillian: An integration testing framework for Java EE,
`https://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/`, 1-0-0.

[3] Biagio Festa, Giovanni Beri
myTaxiService, Design Document,
2015.

[4] Biagio Festa, Giovanni Beri
myTaxiService, Requirements Analysis and Specification Document,
2015.

6 Hours of Work

- Beri Giovanni: ~ 15 hours
- Festa Biagio: ~ 15 hours