POLITECNICO
MILANO 1863

Integration Test Plan

Andrea Gussoni Federico Amedeo Izzo Niccolò Izzo

January 21, 2016

Contents

Chapter 1

Introduction

1.1 Revision History.

Revision 1.0

1.2 Purpose and Scope

1.2.1 Purpose

This document describes the plans for the testing of the *myTaxiService* system. We will describe the designated work-flow for testing the integration between the components of our system. The audience of this document should be composed by all the team members that are going to take part in the testing activity. For this reason from now on we will assume that all the readers of this document have at least a basic knowledge of the technologies involved in the development and testing of the system. This is in fact a fundamental requirement for a correct comprehension of the document itself.

1.2.2 Scope

The system to be developed is **myTaxiService**. This service aims to offer a simplified and reliable access to the preexisting taxi infrastructure of the city. Both taxi drivers and customers will benefit from this product. For example the customers will be able to:

- Have access to a taxi more quickly.
- Reserve in advance a taxi ride with fixed origin and destination.
- Share the taxi fee with others passengers going in the same direction.

And also the drivers will have a number of benefits with the adoption of our system, amongst them there are:

- Fair distribution of the customers.
- Virtual waiting queues instead of physical ones.
- Customer geographical localization.
- Automatic route planning.

1.3 List of Definitions and Abbreviations

1.3.1 Definitions

- Customers: those who will request the ride through the web application or the mobile application.
- Taxi drivers: registered users of the mobile application. They will upload their position and their availability to take rides to the system.
- Standard ride: action that begins with the customer's ride request and ends with the customer's payment at the end of the ride.
- Reserved ride: a ride that has been reserved at least two hours before the starting time. It begins from the reservations and ends with the customer's arrival at his destination.
- Shared ride: a different type of ride in which the customer give his availability to share the ride. A ride is considered shared when at least two customers are traveling in the same taxi cab.
- Smart-phone: a mobile device capable of connecting to the Internet and making and receiving calls and SMS.
- Geo-localization: the act of obtaining a user's geographic coordinates, eventually uploading them to an on-line service.
- Application: mobile or web app running on the user's device.
- System: refers to the part of the application logic that runs on the remote server.
- Taxi Zone: is an area of approximately $2km^2$ for which the taxi-queue is unique.

- Developers: all the people involved in the development of the Service.
- Stakeholders: all the people that may be affected by the Service activities.
- Scalable: used in describing the capability of adapting the resource usage in accordance to an increase/decrease of number of incoming requests.

1.3.2 Acronyms

- DD: Design Document.
- RASD: Requirement Analysis and Specification Document.
- API: Application Programming Interface.
- UI: User Interface.
- OS: Operating System.
- UX: User eXperience.
- SOA: Service Oriented Architecture.
- IaaS: Infrastructure as a Service.
- SaaS: Software as a Service.
- PaaS: Platform as a Service
- MVC: Model-View- Controller.
- OO: Object Oriented.
- Java EE: Java Enterprise Edition.

1.3.3 Abbreviations

- Web app: Web-based application.

1.4 List of Reference Documents

Here a list of the documents we used as reference in the drafting of this document:

- **Project goal, schedule and rules** provided us by the professor of the course of Software Engineering 2.
- **Assignment 4: Test plan**
- **Integration Test Plan** by SpinGrid Project.
- **RASD**, first revision, available at `https://github.com/andrealinux1/se2project/blob/master/Deliveries/RASD_01.pdf`
- **Design Document**, first revision, available `https://github.com/andrealinux1/se2project/blob/master/Deliveries/DD_01.pdf`

Chapter 2

Integration Strategy

2.1 Entry Criteria

Before the *Integration Test* phase can begin, all the code regarding the single components must have passed the respective *unit testing*. This phase is devoted to highlight all the major problems in the algorithms implementation and other errors in the classes themselves. In order to pursue this goal we can use the *JUnit* test framework to have tests ran automatically at each build. An ideal coverage for the unit test can be the 80% of lines of code covered.

Another important aspect that have to be taken in high consideration is the development of a complete and updated documentation of the written code. This will be written in the form of a JavaDoc, and must be updated along with the software in order to provide a clear and stable reference to the code. This will help a lot also during the integration testing phase, making easier to build tests that will stress the most important parts of the components. In addition to this the JavaDoc will provide a quick reference of the interfaces of the various components.

Also a consistent *Code Inspection* is highly recommended. This phase must be performed on the code of the single components, in order to ensure that all the established conventions have been followed. In this phase, for example, we can find issues in the declarations of interfaces, and resolve them before the *Integration Testing* phase and consequently lower the overall weight of the Integration Testing phase.

Finally we have to make sure that before the beginning of the *Integration Testing* phase the following documents have to be delivered:

1. Requirement Analysis and Specification Document.
2. Design Document.

3. Integration Test Plan.

2.2 Elements to be Integrated

As we highlighted in the **Design Document** in our system we have an architecture composed by 4 main tiers:

- Database Tier: this tier is composed by the DBMS. The DBMS is a ready product, we have no development for this component but just a setup. Nevertheless the integration between this layer and the application layer is fundamental.
- Application Tier: in this layer will reside the core of the business logic, and together with the client side will be the tier where the most of the development will be done.
- Web Server Tier: this is the tier that will serve as a bridge between the external world and the application layer.
- Client Tier: is composed by the web application and the mobile applications.

2.3 Integration Testing Strategy

For the integration testing we will mainly follow a *bottom-up approach*. This choice seems the best considering the fact that for the previous development we already divided the whole system in a set of subsystems and components. Each component should have at this point unit tests that guarantee the correct behavior of itself, and we can start from the bottom and gradually integrate more and more components together.

In addition to this fact, we can observe that the interaction between the 4 tiers is done through standardized and well-known interfaces, that shouldn't cause particular problems if the internal development of the tiers is done in a appropriate way. The isolation between the various subsystems keeps the overall coupling very low.

In conclusion once the integration between the internal components of each tier has been done, it should not be too difficult to integrate the subsystems through the high level interfaces (that are mainly RESTful APIs, HTTP based, etc., all well-defined interfaces implemented with common interoperation means). Also this means that the number of required stubs

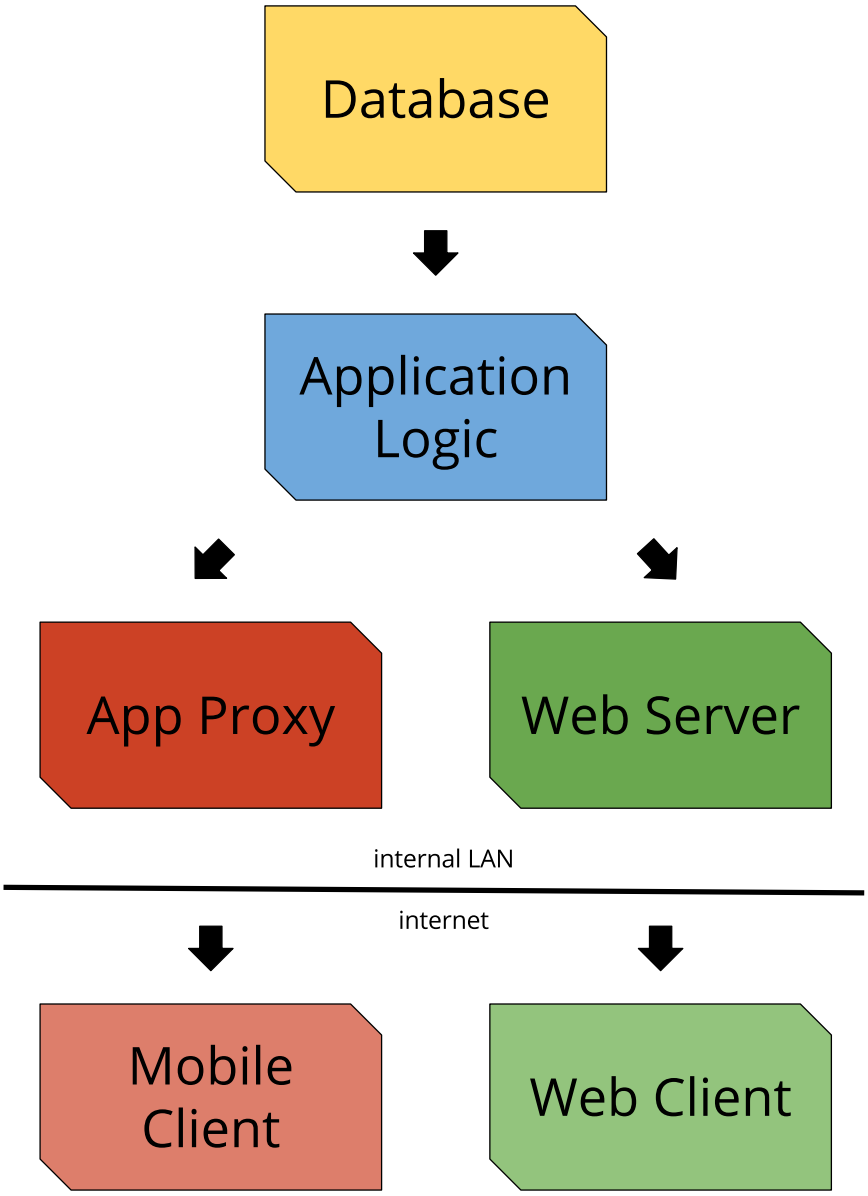

Figure 2.1: The figure shows the order of integration of the subsystems at an high level.

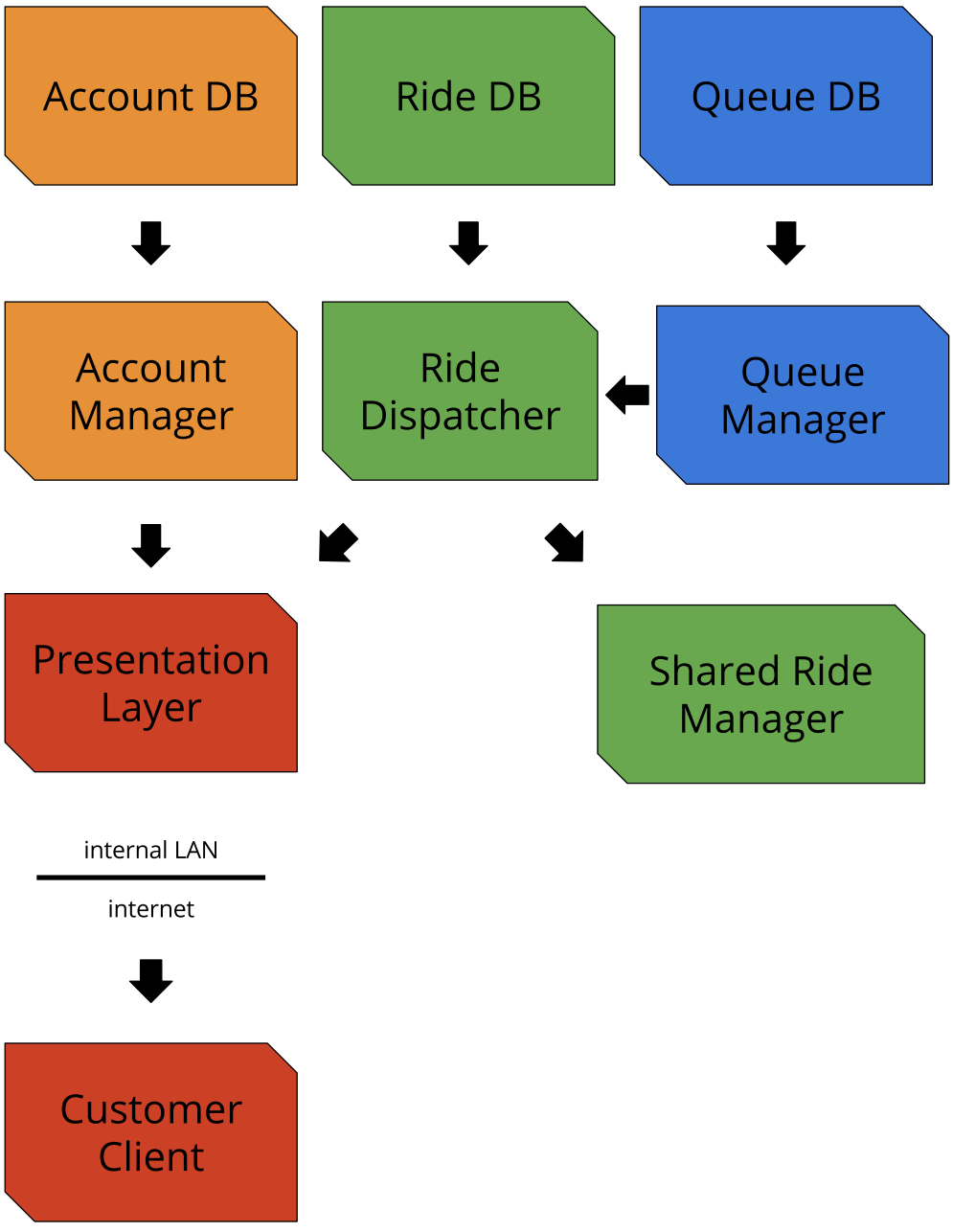

Figure 2.2: The figure shows the order of integrations of the components of the system.

needed in the integration testing can be reduced, due to the fact that we only need to build stubs for the main subsystems and not for each internal component involved in the test.

2.4 Sequence of Component/Function Integration

2.4.1 Subsystem Integration Sequence

For the integration of the subsystems we decided to integrate at first the components of the Back-end. We planned to start with the integration between the Database tier and the Application tier in order to find as early as possible possible issues and bugs in the core of our system. Detecting those bugs later would be as much as expensive as we'd go further in the integration. Then we will integrate the Web Server (and the App proxy) with the Application tier. This is an obvious choice because without a functional back-end we can't in any way begin the test of the client side of the application. Then we will focus on the integration of the client side, starting working with the mobile application that is more important and will be used more than the web interface, because it will be the only way of interaction with the system for the taxi-drivers.

ID	Subsystem	Integrated with
SI1	Application Logic	Database
SI2	App Proxy	Application Logic
SI3	Web Server	Application Logic
SI4	Mobile Client	App Proxy
SI5	Web Client	Web Server

Table 2.1: The table shows the selected order of integration of the subsystems

2.4.2 Software Integration Sequence

In this section we show the order of the integration for the components of each subsystem. We try to integrate at first the components that have less dependencies than the others, in order to implement the minimum number possible of stubs and drivers.

This task could be done by obtaining a proper tree from the dependency graph of the whole system. By giving to each node a cost, representing

the number of dependencies of that particular node we have to take care in ordering the nodes of the tree in ascending cost order. By doing so we will obtain algorithmically the minimum effort order of testing. But reality turns out to be much harder than our models; in fact not all the dependencies arcs have the same value, and if we have the integrated part by our side we can spare an extra stub.

After the integration of all the components needed in a subsystem we can proceed to the integration of different subsystems as it is shown in the following paragraph.

ID	Component	Of Subsystem	Integrates with Component	Of Subsystem
CI1	Account Manager (EJB)	Application Logic	Account DB	Database
CI2	Queue Manager (EJB)	Application Logic	Queue DB	Database
CI3	Ride Dispatcher (EJB)	Application Logic	Ride DB	Database
CI4	Ride Dispatcher (EJB)	Application Logic	Queue Manager (EJB)	Application Logic
CI5	Shared Ride Manager (EJB)	Application Logic	Ride Dispatcher (EJB)	Application Logic
CI6	Shared Ride Manager (EJB)	Application Logic	Ride DB	Database
CI7	Presentation Layer (JSF)	Web Server (App proxy)	Account Manager (EJB)	Application Logic
CI8	Presentation Layer (JSF)	Web Server (App proxy)	Taxi Dispatcher (EJB)	Application Logic
CI9	Presentation Layer (JSF)	Web Server (App proxy)	Ride Dispatcher (EJB)	Application Logic
CI10	Taxi Client	Client	App Proxy	Web Server
CI11	Customer Client	Client	Presentation Layer (JSF)	Web Server
CI12	Customer Client	Client	App Proxy	Web Server

Table 2.2: Integration components in the subsystems

Chapter 3

Individual Steps and Test Description

3.1 Integration test case SI1

Test Case Identifier	SI1T1
Test Item(s)	Application Logic → Database
Input Specification	We will perform some typical interrogations on the database from the components of the Application Logic layer. Various type of queries must be taken in consideration, also not valid ones.
Output Specification	The database layer should perform the correct interrogations on the data, returning the expected result for the queries. In the situations where invalid queries has been performed, systems should handle the exceptions in the right way. Finally attempts of unauthorized access should be denied and appropriately notified.
Environmental Needs	We need to have ready at least a test table, in addition to this, the development and integration of the needed EJBs should have been performed. We also need an adequate driver to perform the calls.
Testing procedure	This kind of test should be automated with the help of the JUnit test suite.

3.2 Integration test case SI2

Test Case Identifier	SI2T1
Test Item(s)	App Proxy → Application Logic
Input Specification	We will test some calls to the back-end by using the RESTful API. We will try both valid calls and invalid calls. The invalid calls will be semantically broken and syntactically broken.
Output Specification	The behavior of the back-end should be the one expected, returning the right values in the case of a valid call, and managing errors and exceptions in the case of invalid, malformed or corrupted requests.
Environmental Needs	We need to have the complete implementation of the Application Logic tier, including the specification of its APIs.
Testing procedure	This kind of test should be automated with the help of the JUnit test suite.

3.3 Integration test case SI3

Test Case Identifier	SI3T1
Test Item(s)	Web Server → Application Logic
Input Specification	The input will be very similar to the App proxy test input listed before. We will test some calls to the back-end by using the RESTful API. We will try both valid calls and invalid calls. The invalid calls will be semantically broken and syntactically broken.
Output Specification	The expected behavior of the back-end should be similar to the one expected in the App proxy testing, returning the right values in the case of a valid call, and managing errors and exceptions in the case of invalid, malformed or corrupted requests.
Environmental Needs	We have the same dependencies of the App proxy: we need to have the complete implementation of the Application Logic tier, including the specification of its APIs.
Testing procedure	This kind of test should be automated with the help of the JUnit test suite.

3.4 Integration test case SI4

Test Case Identifier	SI4T1
Test Item(s)	Mobile Client → App Proxy
Input Specification	We will perform some calls to the App Proxy level through simulating some operations that will take place during the normal life cycle of the system.
Output Specification	The responses obtained by the App Proxy should be the expected ones. Also in this phase we have to check that the requirements about the desired response time and performances are respected.
Environmental Needs	We need to have the complete implementation of the Application Logic tier, including the development of the HTTP interface exposed to the external world. In this phase we also need to be sure that the measures specified in the Design Document about the security concerns have been implemented.
Testing procedure	This kind of test should be automated with the help of the JUnit test suite. We plan also to use some more advanced techniques such as the use of a framework that enables us to deploy some virtual instances of the mobile client and automate operations on them.

3.5 Integration test case SI5

Test Case Identifier	SI5T1
Test Item(s)	Web Client → Web Server
Input Specification	We will perform some calls to the Web Server through simulating some operations that will take place during the normal life cycle of the system.
Output Specification	The responses obtained by the Web Server should be the expected ones. Also in this phase we have to check that the requirements about the desired response time and performances are respected.
Environmental Needs	We need to have the complete implementation of the Application Logic tier, including the development of the HTTP interface exposed to the external world. In this phase we also need to be sure that the measures specified in the Design Document about the security concerns have been implemented.
Testing procedure	This kind of test should be automated with the help of the JUnit test suite. In this phase the help of the JMeter will be very useful to do some automated tests, especially for testing the performances.

3.6 Integration test case CI1

Test Case Identifier	CI1T1
Test Item(s)	Account Manager (EJB) → Account DB
Input Specification	Perform basic operations on the account manager, such as creating a new account, modifying an account or delete one.
Output Specification	The creation of an account should result in a new entry in the Account DB, Also deletion and modifications of accounts should be reflected on relative DB entries.
Environmental Needs	SI1 test passed
Testing procedure	The test can be made using JUnit

3.7 Integration test case CI2

Test Case Identifier	CI2T1
Test Item(s)	Queue Manager (EJB) → Queue DB
Input Specification	Test the creation of queues in the Queue DB and the registration of taxi drivers in a queue. The test also covers taxi accepting a ride and deregistering from the queue.
Output Specification	The operation of the Queue Manager should reflect correctly in changes in the Queue DB.
Environmental Needs	SI1 test passed.
Testing procedure	The test can be made calling the necessary methods of Queue Manager via JUnit

3.8 Integration test case CI3

Test Case Identifier	CI3T1
Test Item(s)	Ride Dispatcher (EJB) → Ride DB
Input Specification	The Ride Dispatcher should be able to register new rides in the Ride DB and update details in a second time, such as the end of the trip or new passengers in a shared ride.
Output Specification	The entries created in the Ride DB should be correct according to the Ride Dispather requests, also entries modification should work correctly.
Environmental Needs	SI1 test passed.
Testing procedure	The test can be made using JUnit.

3.9 Integration test case CI4

Test Case Identifier	CI4T1
Test Item(s)	Ride Dispatcher (EJB) → Queue Manager (EJB)
Input Specification	Create a typical Ride dispatcher working state.
Output Specification	Check if the right methods of Queue Manager are called.
Environmental Needs	None: the test is within the Application Logic subsystem.
Testing procedure	The test can be made with JUnit.

3.10 Integration test case CI5

Test Case Identifier	CI5T1
Test Item(s)	Shared Ride Manager (EJB) → Ride Dispatcher (EJB)
Input Specification	Create a typical Shared Ride Manager working state.
Output Specification	Check if the right methods of Ride Dispatcher are called.
Environmental Needs	None: the test is withing the Application Logic subsystem.
Testing procedure	The test can be made with JUnit.

3.11 Integration test case CI6

Test Case Identifier	CI6T1
Test Item(s)	Shared Ride Manager (EJB) → Ride DB
Input Specification	The Shared Ride Manager should be able to access to rides registered in the Ride DB and mo
Output Specification	The changes from the Shared Ride manager are correctly registered in the Ride DB
Environmental Needs	SI1 test passed.
Testing procedure	The test can be performed with JUnit.

3.12 Integration test case CI7

Test Case Identifier	CI7T1
Test Item(s)	Presentation Layer (JSF) → Account Manager (EJB)
Input Specification	A registration request from the website or an account details modification request.
Output Specification	The Account creation or account changes are correctly forwarded to the Account Manager.
Environmental Needs	SI3 test passed
Testing procedure	The test can be performed with JUnit.

3.13 Integration test case CI8

Test Case Identifier	CI8T1
Test Item(s)	Presentation Layer (JSF) → Taxi Dispatcher (EJB).
Input Specification	A standard ride request from the website.
Output Specification	The request is correctly forwarded to the Taxi Dispatcher.
Environmental Needs	SI3 test passed
Testing procedure	The test can be performed with JUnit..

3.14 Integration test case CI9

Test Case Identifier	CI9T1
Test Item(s)	Presentation Layer (JSF) → Ride Dispatcher (EJB)
Input Specification	We will perform some attempts to allocate a ride.
Output Specification	The system should return the expected values, and in case that is not possible manage the error in an appropriate way
Environmental Needs	The Application Logic should be fully operative and the previous integration procedures must be completed.
Testing procedure	For this test we will make use of JUnit and Mockito for the creation of the stubs.

3.15 Integration test case CI10

Test Case Identifier	CI10T1
Test Item(s)	Taxi Client → App Proxy
Input Specification	Standard calls from the client to the system.
Output Specification	Expected values as result, also in case of invalid or malformed request.
Environmental Needs	Application Logic, Web tier and Mobile App ready.
Testing procedure	JUnit and GenyMotion for simulating virtual instances of the clients.

Test Case Identifier	CI10T2
Test Item(s)	Taxi Client → App Proxy
Input Specification	Potential flooding of requests, maybe malformed.
Output Specification	A correct response to this type of attack, resulting in the system to identify the attack and reacting to it.
Environmental Needs	Application Logic, Web tier and Mobile App ready.
Testing procedure	JUnit and GenyMotion for simulating virtual instances of the clients and JMeter for measuring the level of stress of the system.

3.16 Integration test case CI11

Test Case Identifier	CI11T1
Test Item(s)	Customer Client → Presentation Layer (JSF)
Input Specification	Standard operations on the web interface.
Output Specification	Expected behavior, especially for wanted invalid request and not allowed operations.
Environmental Needs	Application Logic and Web tier.
Testing procedure	JMeter and JUnit to automatize requests .

Test Case Identifier	CI11T2
Test Item(s)	Customer Client → Presentation Layer (JSF)
Input Specification	Flooding of requests.
Output Specification	A correct response to this type of attack, resulting in the system to identify the attack and reacting to it.
Environmental Needs	Application Logic and Web tier.
Testing procedure	JUnit to automatize requests with the help of JMeter to perform the attack and to measure the level of stress of the system .

3.17 Integration test case CI12

Test Case Identifier	CI12T1
Test Item(s)	Customer Client → App Proxy
Input Specification	Standard operations on the mobile client.
Output Specification	Expected values as result, also in case of invalid or malformed request.
Environmental Needs	Application Logic, Web tier and Mobile App ready.
Testing procedure	JUnit and GenyMotion for simulating virtual instances of the clients.

Chapter 4

Tools and Test Equipment Required

For the testing we will relay on tools that enable to automate at least a portion of the work. Follows a list of the main tools that are used (we included in particular the tools that were shown us during the laboratory lessons, and others tools found on the Internet or already known):

- JMeter [1]: is a tool that enables to perform a variety of test on the performance of a system. We will use it on the following subsystems:

  Database Tier The goal is to stress the database layer in order to check if the system is still responsive under a high load, and to adjust the number and the specifications of the dedicated machines that will compose this tier.

  Web Server Tier We will simulate an high number of requests on the web server and on the App proxy in order to simulate a moment of elevated use of the service or also a potential DoS (or DDoS) attack. We have to be sure that the requirements of simultaneous clients and response time identified during the Requirement Analysis phase can be satisfied. In this phase also we have to make sure that the solutions to prevent attacks on the back-end are correctly working.

  Application Logic Tier We can also perform stress tests directly on the Application tier (using the API) in order to identify performance issues that are not caused by the interaction with the Web Server tier but that are at a lower level.

[1] `http://jmeter.apache.org/`

- JUnit [2] is the selected tool for performing the unit testing (not covered in this document), but it is also necessary in order to use Mockito.
- Mockito [3] is a test framework that helps in the generation of stubs and mock elements. Is used in cooperation with JUnit.
- GenyMotion [4] is a virtualization framework that allow us to run multiple instance of the android operating system simultaneously and to have them run our mobile Application. This will indeed require a minimum effort compared with the test on a set of physical devices.

[2] `http://junit.org/`
[3] `http://site.mockito.org/`
[4] `https://www.genymotion.com/`

Chapter 5

Program Stubs and Test data Required

In this section we will describe the required stubs to perform integration testing. Stubs are required to test the functionalities of parts of the systems before the other systems they rely on are fully developed and tested. In this way we can simulate a basic function of some of the components or the subsystems (e.g. a simple stub that accepts method calls and returns a fixed result) in order to test the component under development. In this way we don't have to use a *big bang integration* approach and we can do the test of all subsystems only as a final step.

We plan to create and use stubs for the following entities:

- Database: we can provide a simple stub of the database by creating the structure of tables and tuples and populating them with test data in order to make possible for the application tier to begin manipulating some data. We can also try to populate the database by doing a massive fake subscription to the service using an automated tool in order to test if everything goes in the right way in this phase.

- Web Server tier and Application Logic: in order to prepare the system to be subjected to a test using some selected beta-testers (this kind of tests can be performed later on in the development phase and not necessarily in this phase) it is imperative to have ready at least a simplified version of an interface for customers and taxi drivers. For this reason we need to prepare a simplified version of the back-end subsystem that enables us to test the functionalities that the client should have. Also in this case (as seen for the Database stub) we can have a simple interface (that works on http) that simply responds to requests

with prefixed values, in order to test in an exhaustive way the web app and the mobile app.

- Client: during the testing of the back-end (that means either the Web Server and the Application Server) we should have a way to emulate the actions performed by the clients during a normal period of activity of the system. Doing this by manually performs actions would result into a high manual effort. Also the tools provided by JMeter wouldn't be enough to simulate a real interaction between the system and an user. So we plan to use a framework that can automate the deploy of virtual instances of the application in order to have an automatic way to schedule the execution of predetermined tasks.

- App proxy: to emulate a large number of mobile App clients we can use a virtualization infrastructure such as GenyMotion [1]. This software allows us to generate a set of virtual devices, each one with his android operating system and to have everyone of them running our client app and with some scripting to simulate the activity of a large number of agents. This option lets us reproduce in a testing environment a situation very similar to the real use cases. This requires the mobile App to be fully developed, in order to execute it properly on the virtualized devices.

[1] `https://www.genymotion.com/`

Chapter 6

Appendix

6.1 Used Software

For the redaction of this document we used various software, here a little list:

- LaTeX framework (MacTeX on OS X and TeX Live on GNU/Linux) to generate the document.
- Various editor to edit the source file:
  - Sublime Text 3 (Beta)
  - Atom.io
  - Vim
- Self-Hosted ShareLaTeX to collaboratively edit the document.
- Git to version the source, GitHub to host the repository.
- Gimp, Inkscape and ImageMagick to edit some images (.svg to .png).
- Draw.io for drawing some diagrams.
- Teamspeak and Hangouts used to organize conference-calls in order to work together.

6.2 Hours of Work

We tried to distribute in an equal way the workload for each team-member. In particular we tried to arrange physical meetings or conference calls for the parts where important choices had to be made. We estimated that each of

us spent on average 15 hours on the drafting of this document. Therefore this task took a total of, more or less, 40 hours of work.