Politecnico di Milano

A.A. 2015-2016

Software Engineering 2 project

Integration Test Plan Document

Alessandro Pozzi (mat. 852358), Marco Romani (mat. 852361)

21 January 2016

Version 1.0

Summary

1. Introduction

1.1 Revision History

January 21, 2016 – First Version (1.0) of this document.

1.2 Purpose and scope

The purpose of the Integration Test Plan Document (ITPD) is to describe the set of tests necessary to verify that every component of a system (i.e. *MyTaxiService*, in this context) works as expected in relation with the others. To accomplish this, an integration strategy will describe in which order and with which procedures the system's components should be assembled together during the testing phase.

MyTaxiService's application is a client-server software, which aims to facilitate taxi's requests and booking performed by registered customers. To do so, it must be able to handle remote communication over the Internet. In addition to this, the server side of the system needs to access external services such as: email service, GPS service and, of course, transactions with the database. The integration test should consider this aspects simulating the behavior of these external and network components in order to test correctly all the set of functionalities of the other components.

1.3 List of Definitions and Abbreviations

- MTS – MyTaxiService
- RASD – Requirement Analysis and Specification Document
- DD – Design Document

1.4 List of References

1.4.1 Documents

- MyTaxiService's RASD (Alessandro Pozzi, Marco Romani)
- MyTaxiService's DD (Alessandro Pozzi, Marco Romani)

1.4.2 Tools

- Documentation for Mockito: http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/Mockito.html
- Documentation for JUnit: http://junit.org/javadoc/latest/
- Documentation for Arquillian: http://arquillian.org/guides/
- Documentation for Citrus: http://www.citrusframework.org/documentation.html
- Documentation for GreenMail: http://www.icegreen.com/greenmail/javadocs/index.html

2. Integration Strategy

2.1 Entry Criteria

Most of the functionalities of the system's components relies on complex interactions between multiple physical and logical entities.

The only component that, at this level of abstraction, contains reasonably autonomous functionalities is the QueueManager. For this reason, the managing of the taxi zones/queues and the algorithms that exploit them (e.g. the depth first search of adjoining taxi zones/queues) should be exhaustively unit tested before the integration test phase.

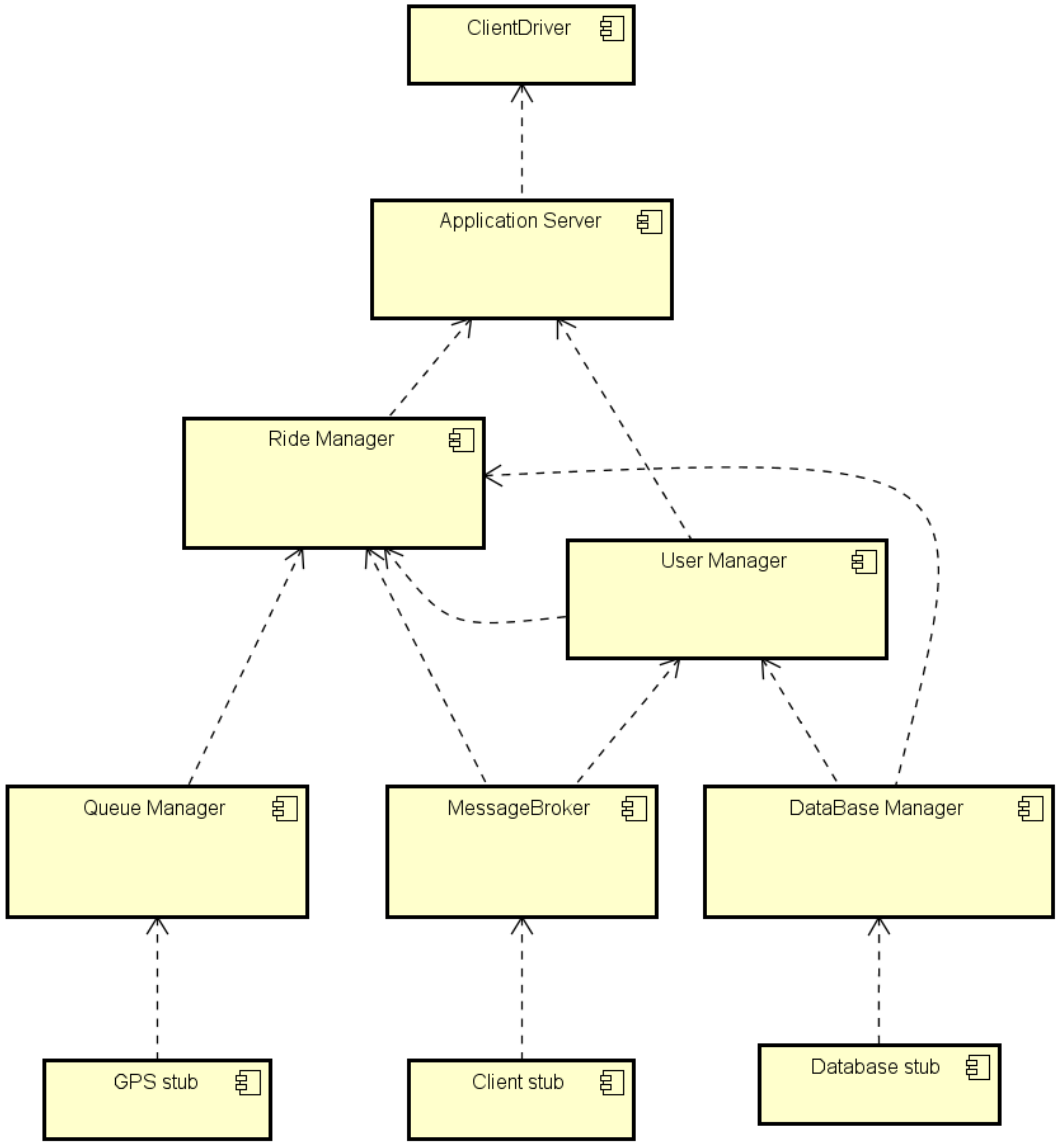
In this way, during integration test phase, testers and developers will focus only on issues related to components' interaction.

2.2 Elements to be integrated

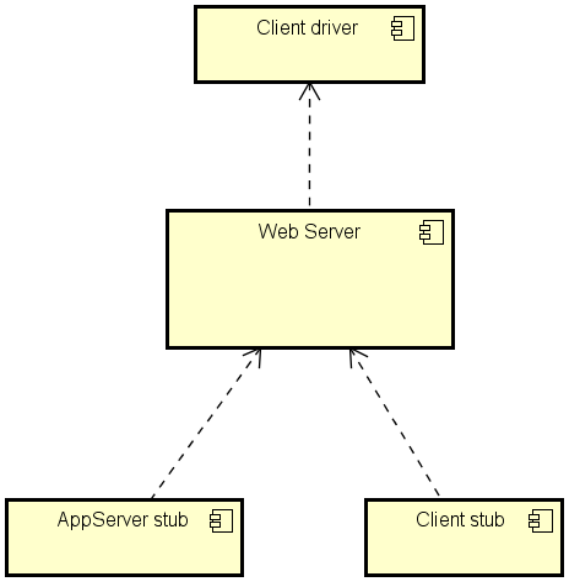
The elements to be integrated during the integration test are the components defined in the Design Document, plus some stubs that act as placeholders for remote components.

The components are divided in 3 subsystems according to their deployment: *client* subsystem, *web server* subsystem and *application server* subsystem.

The application server subsystem is the most interesting and relevant to test. Here we provide a diagram representing the components belonging to it. The arrows represent a topological order for the integration.

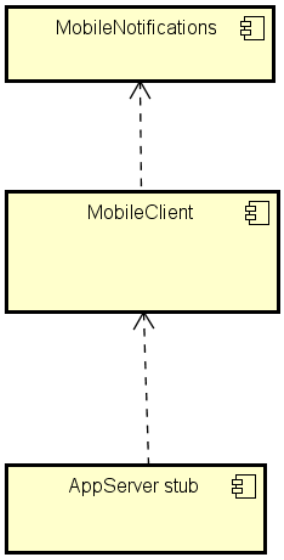


The web server subsystem is very simple and, at this level of abstraction, contains only one component which needs to be tested using a couple of drivers and stubs simulating inputs/outputs from/to the client and the application server.



The client subsystem is quite simple too. Similarly to the previous subsystem, it requires a stub representing the application server and a driver that generates fake notifications.

We only include a diagram for the integration test of the mobile client, since the web client relies on existing and trustful browsers and on the GUI – html pages stored in the web client. We think that it is more practical to test this things by hand on the whole functioning system.



2.3 Integration test strategy

The integration strategy chosen is a mixed strategy. For the application server it is a *bottom-up*-like approach, with the exception of the components external to the application that are represented by stubs. Starting by these stubs, all other components are integrated and tested in a bottom-up way.

The strategy for the web server subsystem and the client subsystem is different, since they are simple components whose functionalities all rely on remote services. They could be divided in more granular components (e.g. GUI and communication on the client) but, basically, their integration test strategy consists in providing stubs and drivers necessary for in and out communication over the network.

The strategy described is of course at component level, more specific testing strategy of single components' code is not part of this document.

2.4 Sequence of Component/Function integration

The three subsystems **Application Server**, **Web Server** and **Client** do not interact directly during the testing because of the set of stub and driver components in which they are wrapped into. This means that these subsystem can be tested in any order; however it is advise to test the Application Server before the other components.

Note that the dashed arrows in the images below have been used to symbolize the *dependency* between components. A typical example: a *driver* uses a *component* (i.e. the component depends from the driver) and a *component* uses a *stub*.

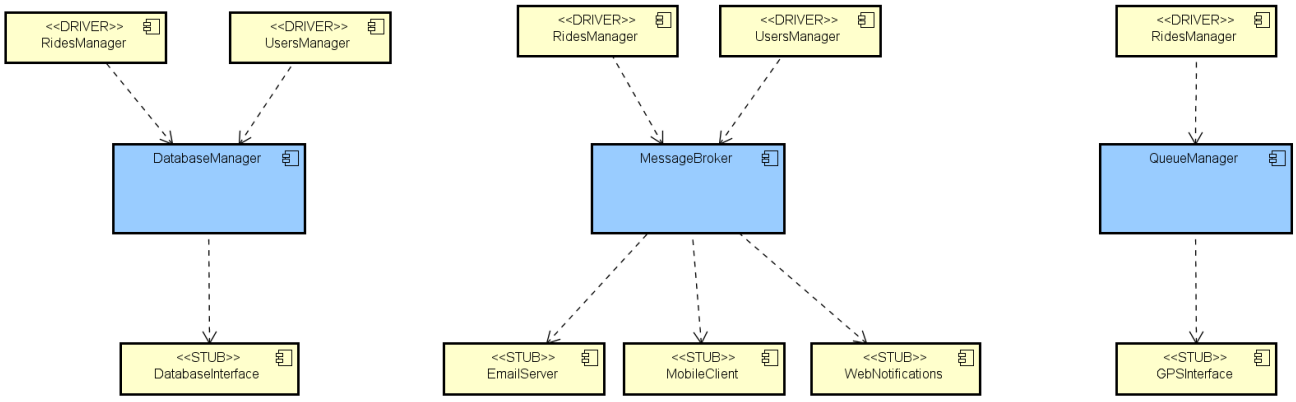
The blue components in the images represent the components that are actually tested at that level.

2.4.1 Application Server

In the following section the steps required to perform the *bottom-up* integration approach will be shown.

2.4.1.1 Level 1 (DatabaseManager, MessageBroker, QueueManager)

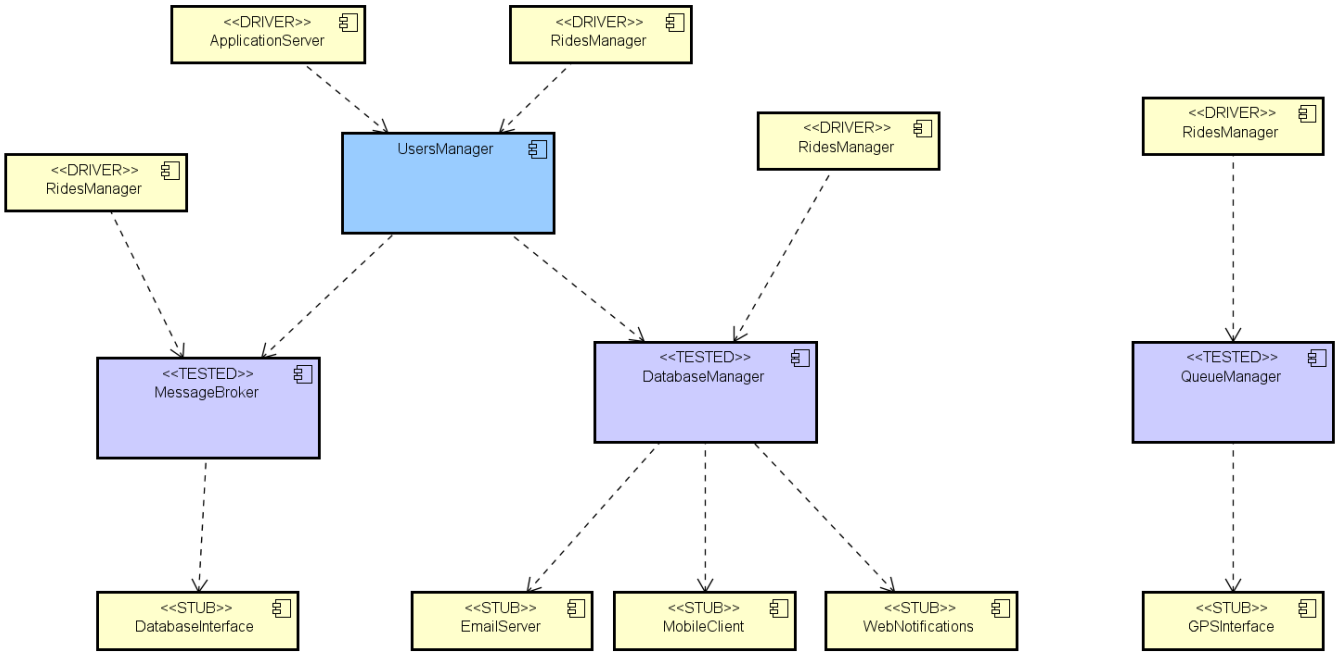
DatabaseManager, *QueueManager* and *MessageBroker* are components that can be tested independently in any order. The image shows the required interfaces' stubs and the components' drivers.



Notice that there are different drivers for the same component (like *RideManager*) because, of course, each driver is specifically bound to a single component.

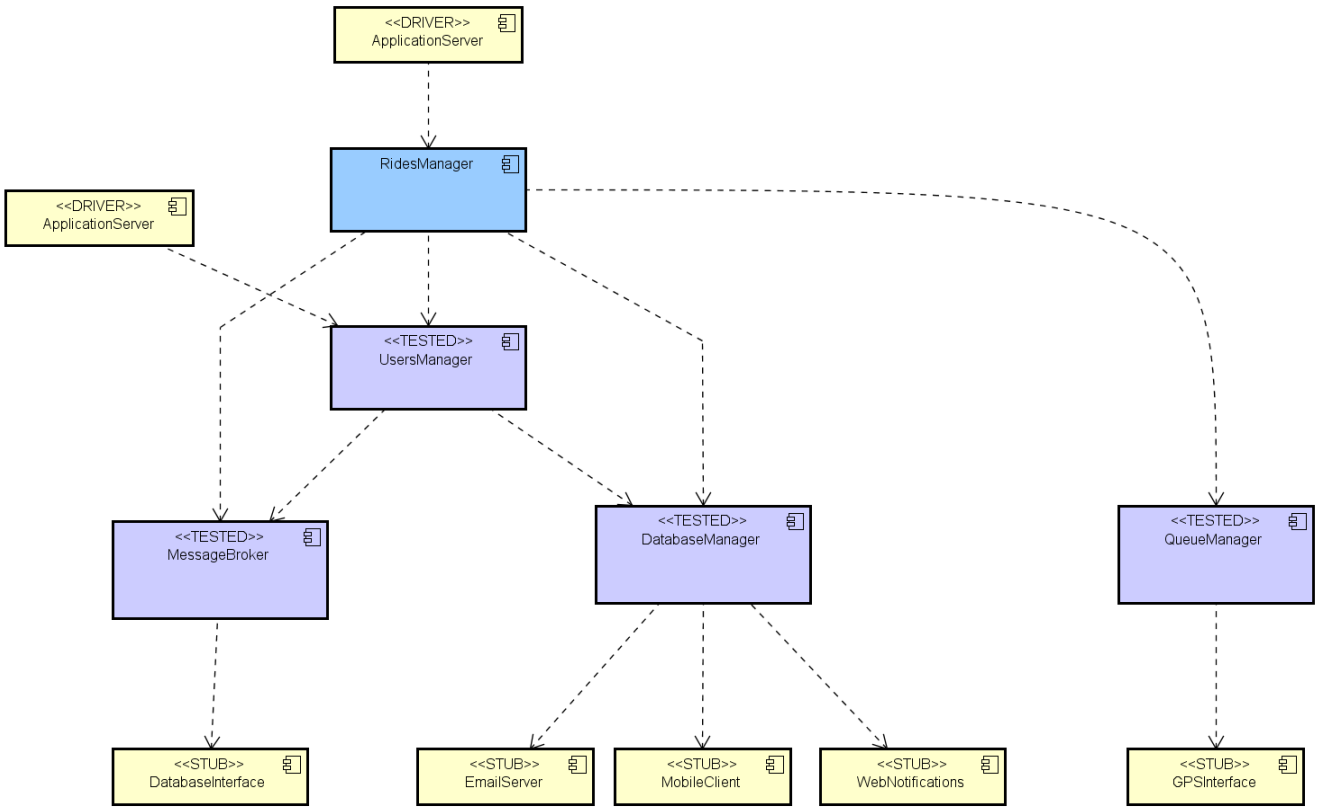
2.4.1.2 Level 2 (UsersManager)

Testing the *UsersManager* is the following step. 2 *UsersManager* driver are removed and the proper *UsersManager* component is introduced.



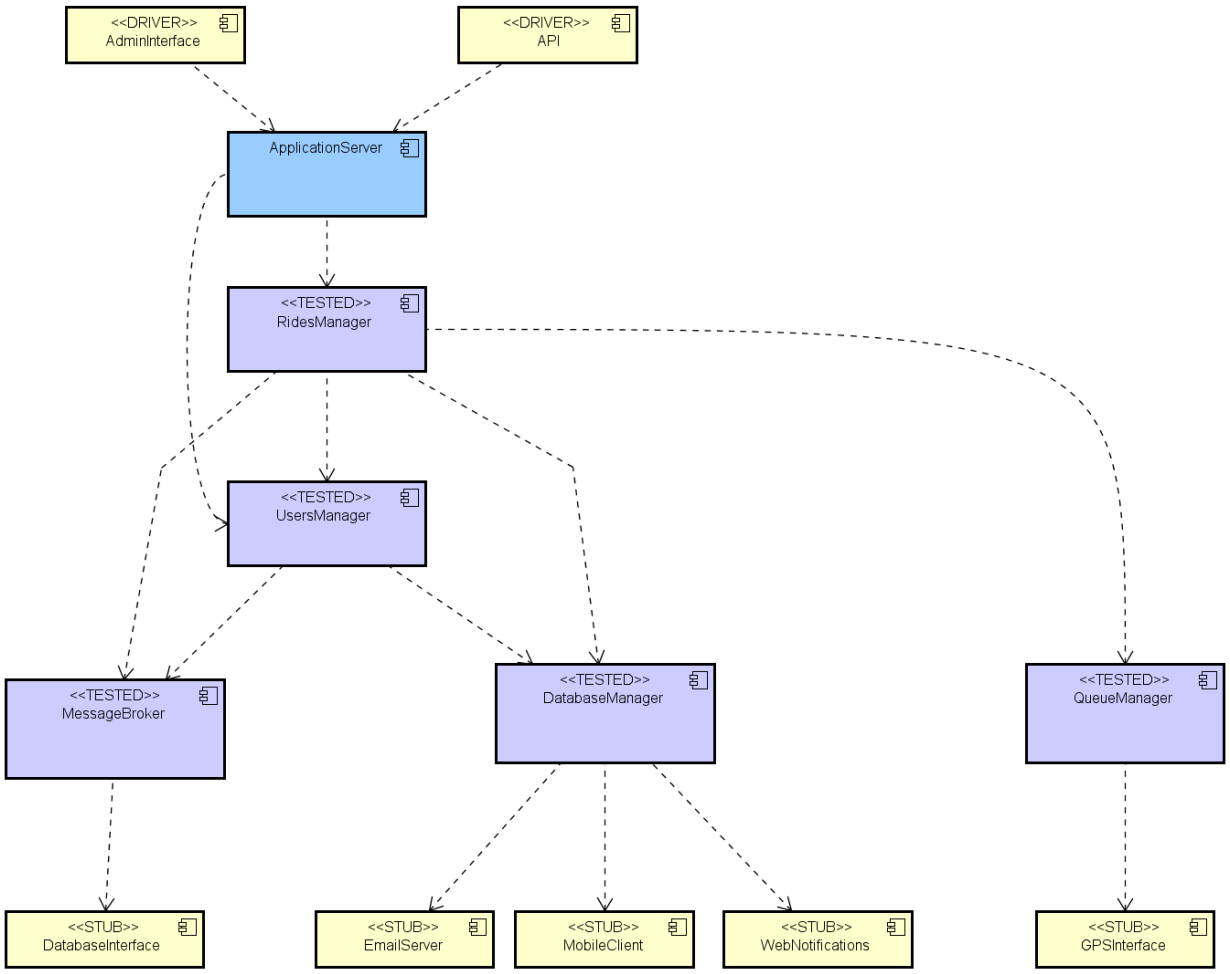
2.4.1.3 Level 3 (RideManager)

Now the RidesManager is tested.

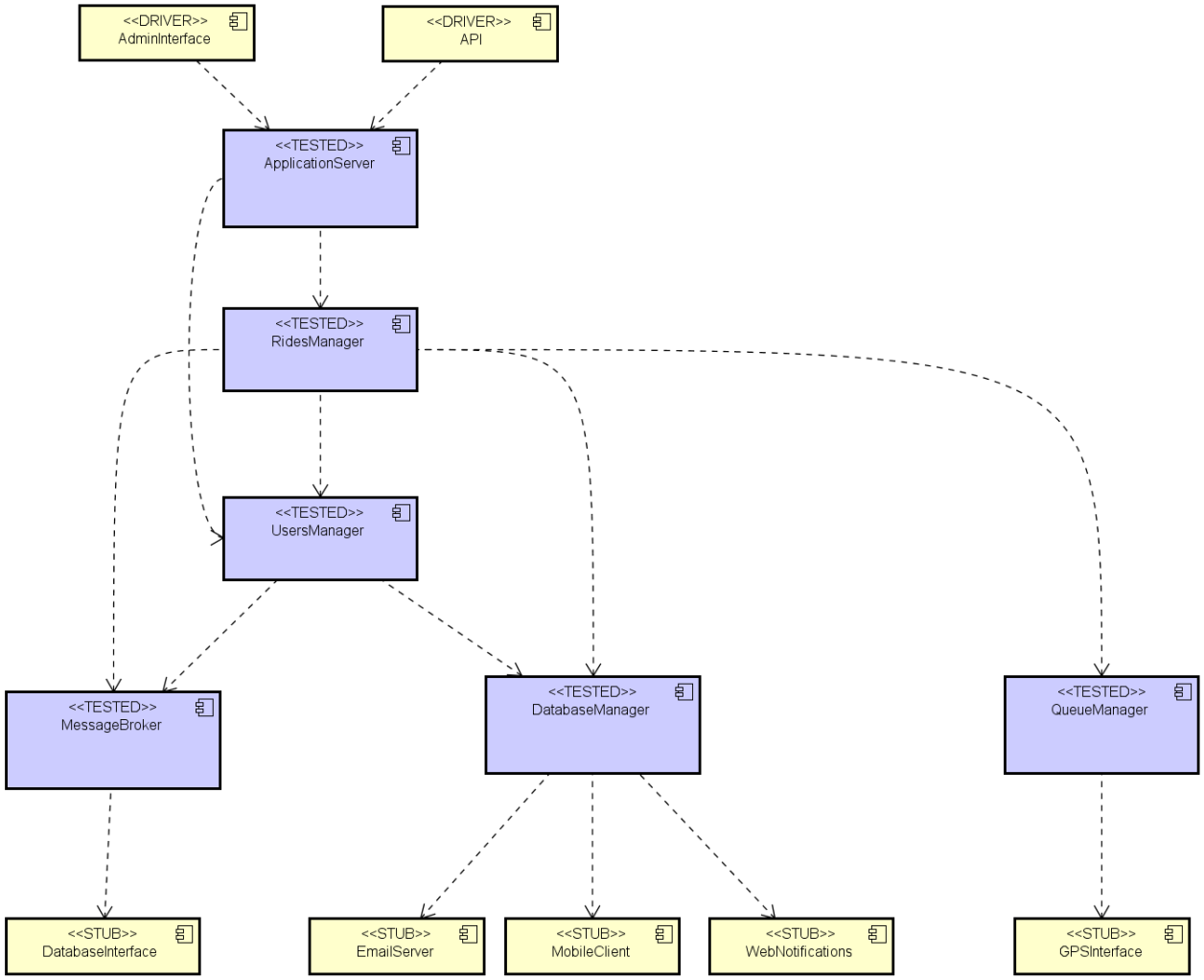


2.4.1.4 Level 4 (ApplicationServer)

The last component to be tested is the *ApplicationServer*.

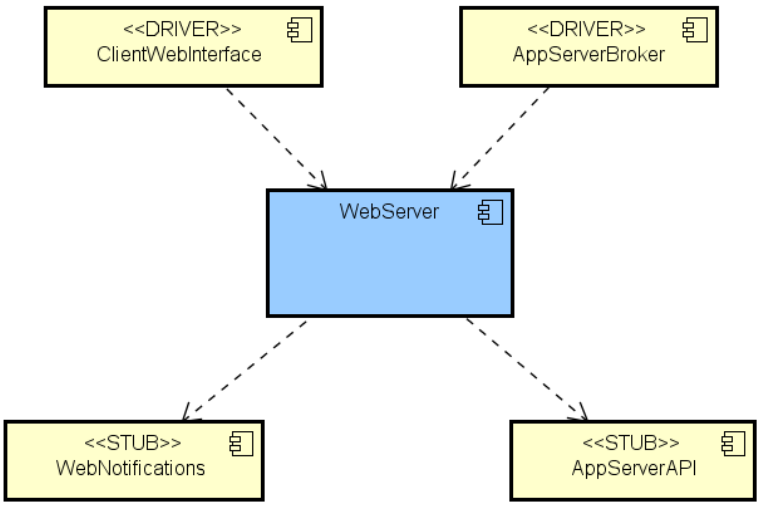


2.4.1.5 Final result

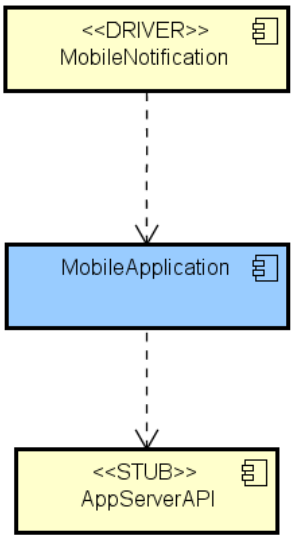


2.4.2 Web Server

The *WebServer* can be tested alone, using the appropriate set of stubs and drivers.



2.4.3 Client

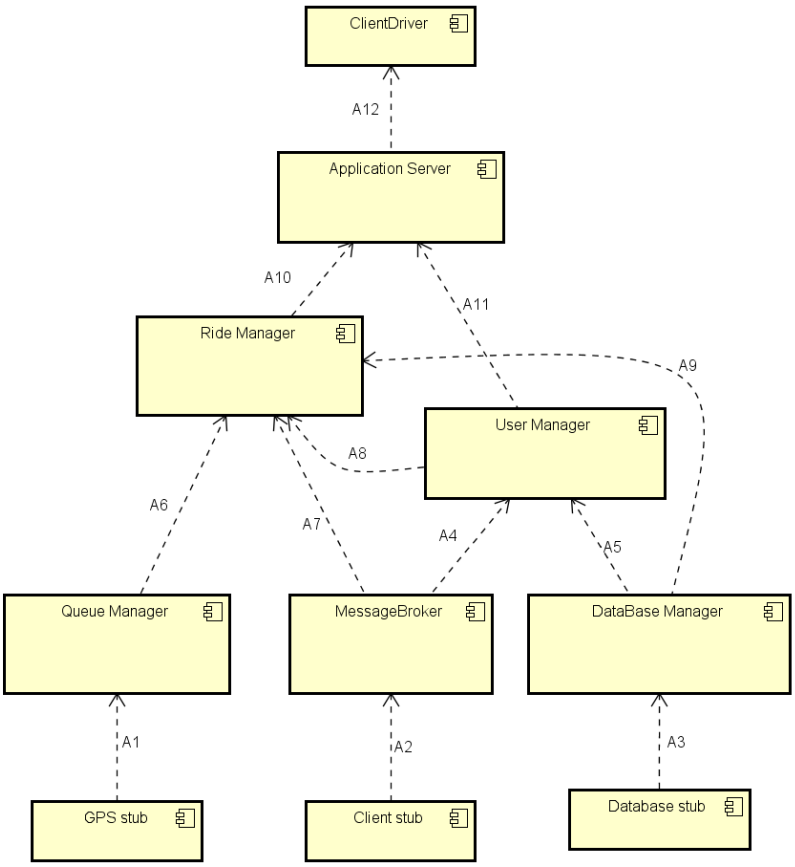


3. Individual Steps and Test Description

How to read the tables headers:

- **Test case identifier** – Identifies the integration test case in the image.
- **Tested items** – Identifies the component to be tested in the following format: *Component1 -> Component2*, where *Component1* uses *Component2* ("->" symbolize the *dependency* relationship)
- **Input Specification** – Input or context that is required to reconstruct in order to perform the integration testing. On a practical point of view, may consist in one or more method calls performed by the first component towards the second component. More inputs (i.e. different method calls that must be tested between the two components) are represented with a numbered list.
- **Output specification** – Output, or final context, that the integration must produce. More outputs are represented with a numbered list and are related to the correspondent inputs.
- **Environmental needs** – Preconditions needed before proceeding with the integration testing of these components.

3.1 Application Server



Test case identifier	A1
Tested items	QueueManager -> GPS stub
Input Specification	Request the GPS position of a taxi driver
Output specification	The coordinates are correctly received, and belong to the correct taxi
Environmental needs	None

Test case identifier	A2
Tested items	MessageBroker -> ClientStub
Input Specification	1. Send an email to a specific user 2. Send a notification, update or other type of messages to the mobile client of a specific user 3. Send a notification, update or other type of messages to the web client interface of a specific user
Output specification	1. The email have been prepared and addressed correctly to the appropriate account 2. The message have been correctly formulated and sent 3. The message have been correctly formulated and sent
Environmental needs	None

Test case identifier	A3
Tested items	DatabaseManager -> DatabaseStub
Input Specification	1. Create, modify and delete data on the database (some relevant data: users account, rides, taxis) 2. Formulate a query and retrieve data from the database.
Output specification	1. The data have been correctly created, modified and deleted. 2. The data have been correctly retrieved from the database
Environmental needs	None

Test case identifier	A4
Tested items	UsersManager -> MessageBroker
Input Specification	Create and send a message or notification for one or more users
Output specification	The message or notification is correctly received, together with any required additional information (e.g. name of the user that must receive the message).
Environmental needs	The MessageBroker must have been tested.

Test case identifier	A5
Tested items	UsersManager -> DatabaseManager
Input Specification	1. Require the creation, modification or elimination of an user's account. 2. Retrieve all the information about a certain user.
Output specification	1. The creation, modification or elimination request have been correctly received and executed. 2. The requested information are retrieved and returned.
Environmental needs	The DatabaseManager must have been tested.

Test case identifier	A6
Tested items	RideManager -> QueueManager
Input Specification	1. Add or remove a taxi from a queue. 2. Require to update the position of a taxi. 3. Move a taxi to the bottom of a queue.
Output specification	1. The taxi have been added or removed correctly. 2. The position of the taxi have been correctly updated. 3. The taxi have been moved to the bottom of the queue.
Environmental needs	The QueueManager must have been tested.

Test case identifier	A7
Tested items	RideManager -> MessageBroker
Input Specification	Create and send a message or notification for one or more users related to a ride or taxi.
Output specification	The message or notification is correctly received, together with any required additional information.
Environmental needs	The MessageBroker must have been tested.

Test case identifier	A8
Tested items	RideManager -> UsersManager
Input Specification	Retrieve the information about a certain user.
Output specification	The information is correctly retrieved and returned.
Environmental needs	The UsersManager must have been tested (as well as all the components needed to test the UserManager itself).

Test case identifier	A9
Tested items	RideManager -> DatabaseManager
Input Specification	1. Create, modify or delete a ride. 2. Retrieve the information about a specific ride.
Output specification	1. The ride has been correctly created, modified or deleted. 2. The information related to the ride is correctly returned.

Environmental needs	The DatabaseManager must have been tested.

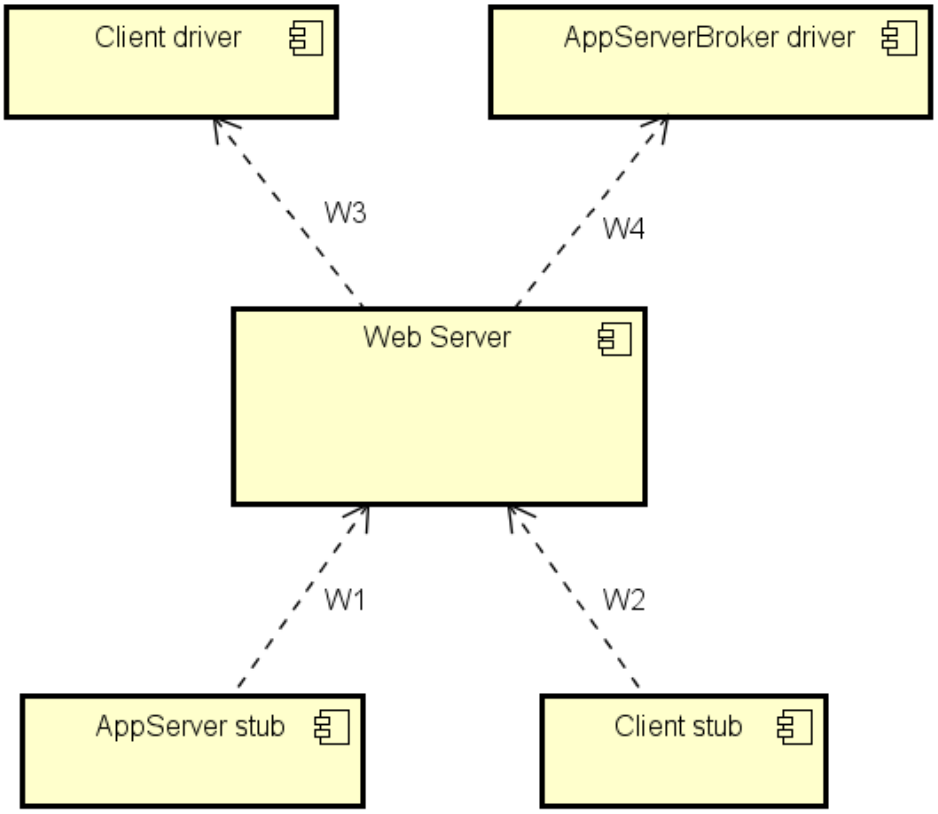
Test case identifier	A10
Tested items	ApplicationServer -> RideManager
Input Specification	1. Create or delete a ride. 2. Retrieve the information about a specific ride. 3. Associate a user to a ride. 4. Retrieve the rides associated to a user.
Output specification	1. The ride has been correctly created or deleted. 2. The information has been correctly returned. 3. The user have been correctly associated to the ride (as a customer or driver). 4. The rides have been correctly returned.
Environmental needs	The RideManager must have been tested (as well as all the components needed to test the RideManager itself).

Test case identifier	A11
Tested items	ApplicationServer -> UsersManager
Input Specification	1. Create, modify or delete a user. 2. Check if some information (i.e. the couple email and password) are valid and belongs to a user.
Output specification	1. The user have been correctly create, modified or deleted. 2. The validation is executed correctly.
Environmental needs	The UsersManager must have been tested (as well as all the components needed to test the UserManager itself).

Test case identifier	A12
Tested items	ClientDriver -> ApplicationServer
Input Specification	1. Register a new user. 2. Login a registered user. 3. Create, update or delete taxi rides. 4. Require the rides related to a user. 5. Require specific information about a user or a ride. 6. Communicate the availability, the acceptance and refusal of rides (by taxi drivers). 7. Perform functions reserved to administrators: create and delete driver's accounts, modify rides' status and drivers' status.
Output specification	1. The user is correctly registered. 2. The user has correctly performed the log-in. 3. The taxi ride has been correctly created, updated or deleted. 4. The rides have been correctly returned.

	5. The ride's information have been correctly returned. 6. The availability, acceptance and refusal have been correctly communicated. 7. All the functions reserved to the administrators have been correctly performed.
Environmental needs	The ApplicationServer must have been tested (as well as all the components needed to test the ApplicationServer itself).

3.2 Web Server



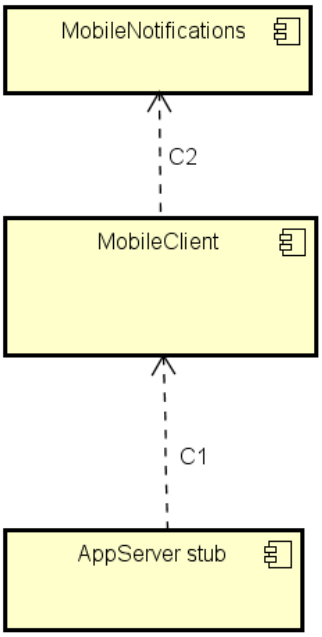
Test case identifier	W1
Tested items	WebServer -> AppServerStub
Input Specification	Require the execution of a service requested by the client.
Output specification	The service requested by the client have been correctly mapped to the correspondent ApplicationServer's service, which have been invoked.
Environmental needs	None

Test case identifier	W2
Tested items	WebServer –> ClientStub
Input Specification	Create and send a web page to the client.
Output specification	The web page is correctly received.
Environmental needs	None

Test case identifier	W3
Tested items	Client Driver -> WebServer
Input Specification	Require the execution of a service.
Output specification	The service has been correctly identified and executed.
Environmental needs	None

Test case identifier	W4
Tested items	AppServerBroker Driver -> WebServer
Input Specification	Send notification or other messages.
Output specification	The message or notification have been received and handled accordingly.
Environmental needs	None

3.3 Client



Test case identifier	C1
Tested items	MobileClient -> AppServer Stub
Input Specification	Require the execution of a service.
Output specification	The service request is sent and identified correctly.
Environmental needs	None

Test case identifier	C2
Tested items	MobileNotifications -> MobileClient
Input Specification	Send a notification or other message to the mobile client.
Output specification	The notification or message is correctly received and handled.
Environmental needs	None

4. Tools and test equipment required

In this section we recommend some tools that might be used during the integration test phase. These advices are not mandatory since the RASD document and the Design Document to which this document refers to do not enter in technical details and do not point towards the choice of a particular language/platform, leaving it to the implementation phase.

However, we think that it is very likely that the final choice will fall on the Java EE platform. For this reason, we suggest some possibly useful tools to be used based on it.

For what concerns unit testing, for example unit testing necessary to fulfil the entry criteria of chapter 2.1, **Mockito** could be an interesting choice together with **Junit**.

Instead, for what concerns the most relevant part of this document, which is integration test, **Arquillian** might be an easy choice, always together with JUnit.

In addition to these "core" tools, we find that it might be useful to look for specific frameworks concerning the handling of messages, since the remote communication among the subsystems identified in the previous chapters is not trivial at all, and it's a central part of the whole system.

For example, **Citrus** is a framework that provides this kind of service.

Since we decided to include an email service in MyTaxiService application, it could be useful to have a tool specifically devoted to the testing of this part. Such a tool might be, for example, **GreenMail**.

As a final suggestion, we recognize that for some parts of the software there might be the need to apply manual testing, for example during the testing of web client / web pages.

As we said at the beginning, almost all these suggestions relies on the choice of Java EE as platform for the implementation. If that won't be the case, a good strategy would be to look for similar tools that accomplish as much as possible the same purposes of the ones described above, recurring to manual testing when it is not possible.

5. Program Stubs and Test Data Required

5.1 Program stubs

As previously stated many times, all 3 subsystems need quite complex stubs that simulate the behavior of remote components to which they ask services. There is no need of other stubs during integration phases due to the fact that the approach chosen is mostly bottom-up.

Stubs for ApplicationServer components:

- Email server stub (MessageBroker)
- Database stub (DatabaseManager)
- GPS stub (QueueManager)
- Client stub for notifications (MessageBroker)
- WebServer stub for web notifications (MessageBroker)

Stubs for WebServer component:

- AppServer stub
- WebNotifications stub (basically a web client stub)

Stubs for MobileClient component:

- AppServer stub

We do not mention the drivers here because they are much simpler to program, since they only need to call procedures of the components.
In addition, as previously stated in chapter 2, we do not include stubs for the web client component because it is only a logical component already implemented by any kind of browser.

5.2 Test data

In order to perform meaningful tests, some “fake” data sets are required:

- Test set of users’ accounts, both drivers and customers.
- Test set of rides, both requests and reservations.
- Test sets of taxi and GPS coordinates.
- Test sets of zones and related queues.

6. Appendix

6.1 Hours of work

Alessandro Pozzi ~ 9 hours

Marco Romani ~ 9 hours

6.2 Software and tools used

- Microsoft Word (https://products.office.com/it-it/word) to redact and to format this document.
- Astah Professional (http://astah.net/) to create the Component Diagram and the other integration test plan images.
- GitHub (https://github.com) to share the working material of this project.