

**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 2](#_Toc440384598)

[1.1 Revision History 2](#_Toc440384599)

[1.2 Purpose and scope 2](#_Toc440384600)

[1.3 List of Definitions and Abbreviations 2](#_Toc440384601)

[1.4 List of Reference Documents 2](#_Toc440384602)

[2. Integration Strategy 3](#_Toc440384603)

[2.1 Entry Criteria 3](#_Toc440384604)

[2.2 Elements to be integrated 4](#_Toc440384605)

[2.3 Integration test strategy 6](#_Toc440384606)

[2.4 Sequence of Component/Function integration 7](#_Toc440384607)

[2.4.1 Application Server 7](#_Toc440384608)

[2.4.2 Web Server 12](#_Toc440384609)

[2.4.3 Client 12](#_Toc440384610)

[3. Individual Steps and Test Description 13](#_Toc440384611)

[4. Tools and test equipment required 14](#_Toc440384612)

[5. Program Stubs and Test Data Required 15](#_Toc440384613)

[6. Appendix 16](#_Toc440384614)

[6.1 Hours of work 16](#_Toc440384615)

[6.2 Software and tools used 16](#_Toc440384616)

# Introduction

## Revision History

January 12, 2016 – First Version (1.0) of this document.

## 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 behaviour of these external and network components in order to test correctly all the set of functionalities of the other components.

## List of Definitions and Abbreviations

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

## List of Reference Documents

* MyTaxiService’s RASD (Alessandro Pozzi, Marco Romani)
* MyTaxiService’s DD (Alessandro Pozzi, Marco Romani)

# Integration Strategy

## 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.

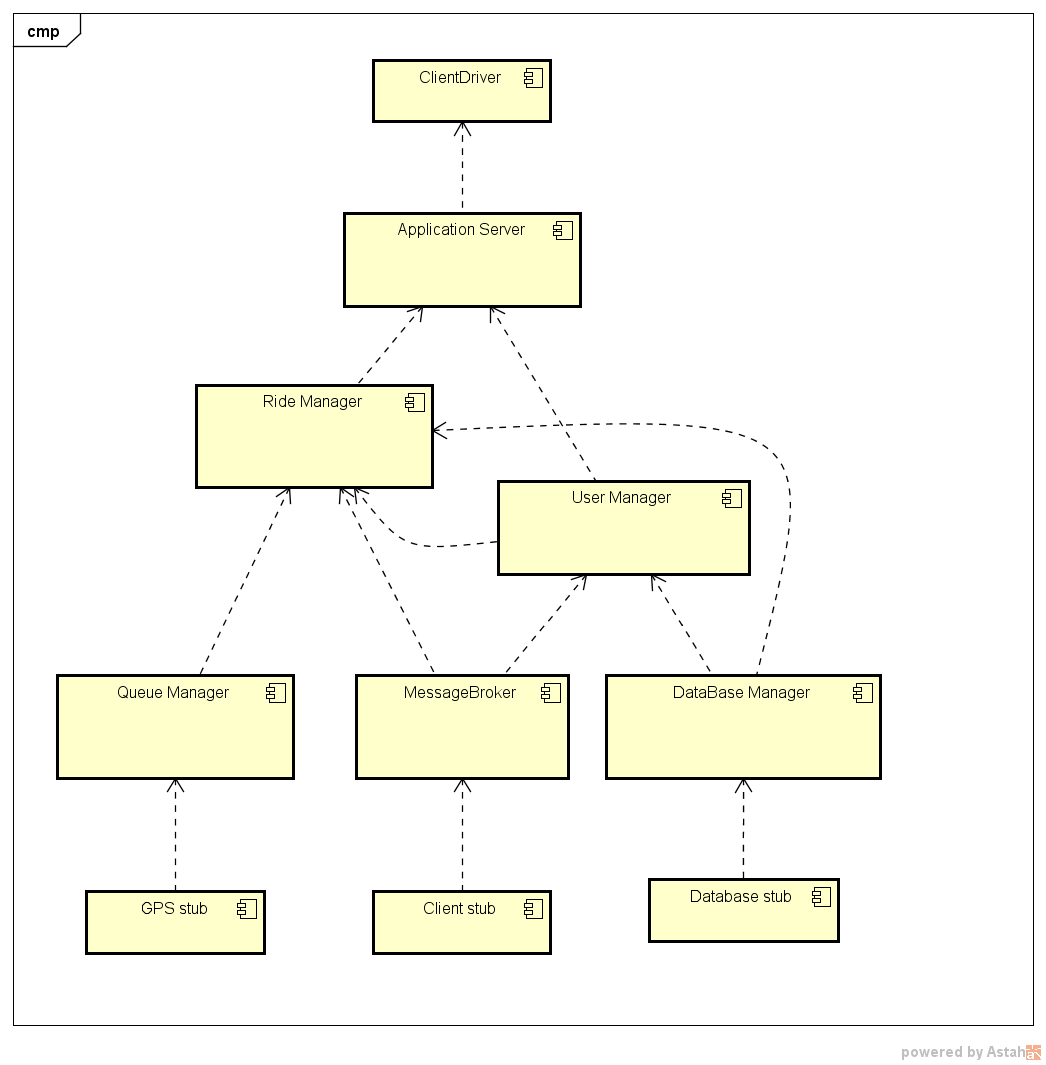
In this way, during integration test phase, testers and developers will focus only on issues related to components’ interaction.

## Elements to be integrated

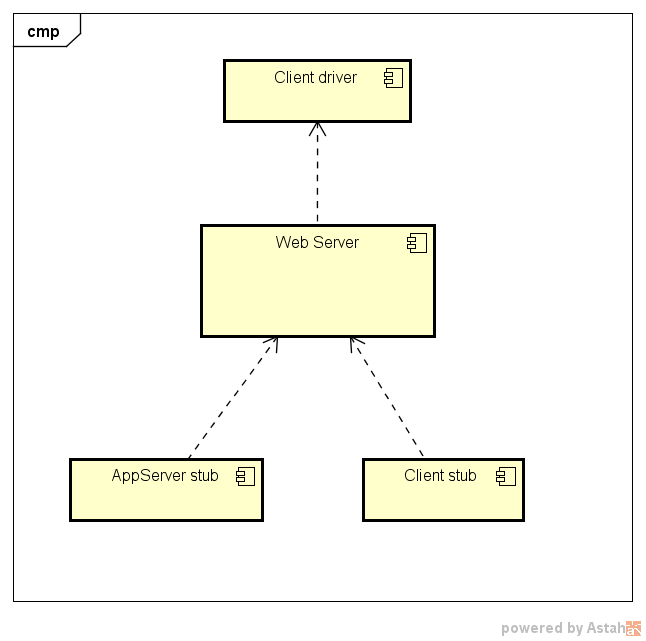
The elements to be integrated during the integration test are the components defined in the Design Document, plus some stubs that act as placeholders for remote components.

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.

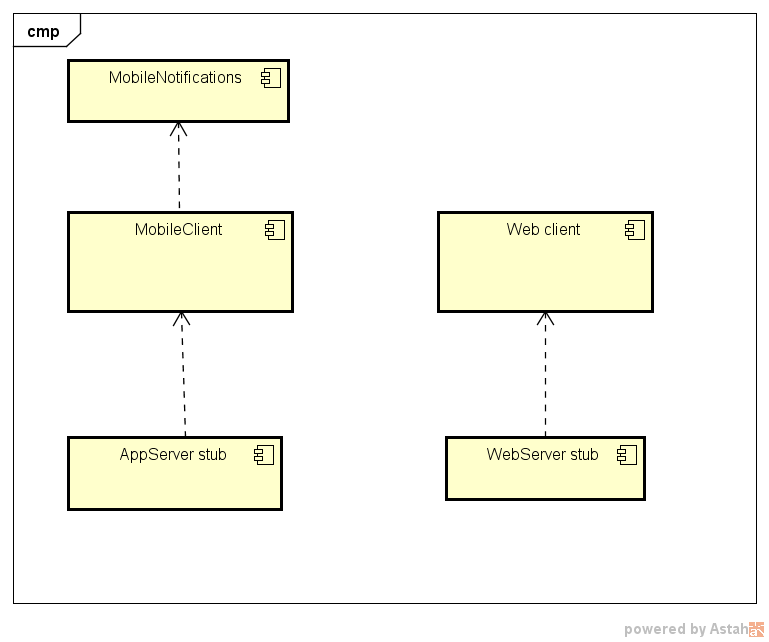


The web server subsystem is very simple and, at this level of abstraction, contains only one component which needs to be tested using a couple of drivers and stubs simulating inputs/outputs from/to the client and the application server.



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.



## 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.

## 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*.

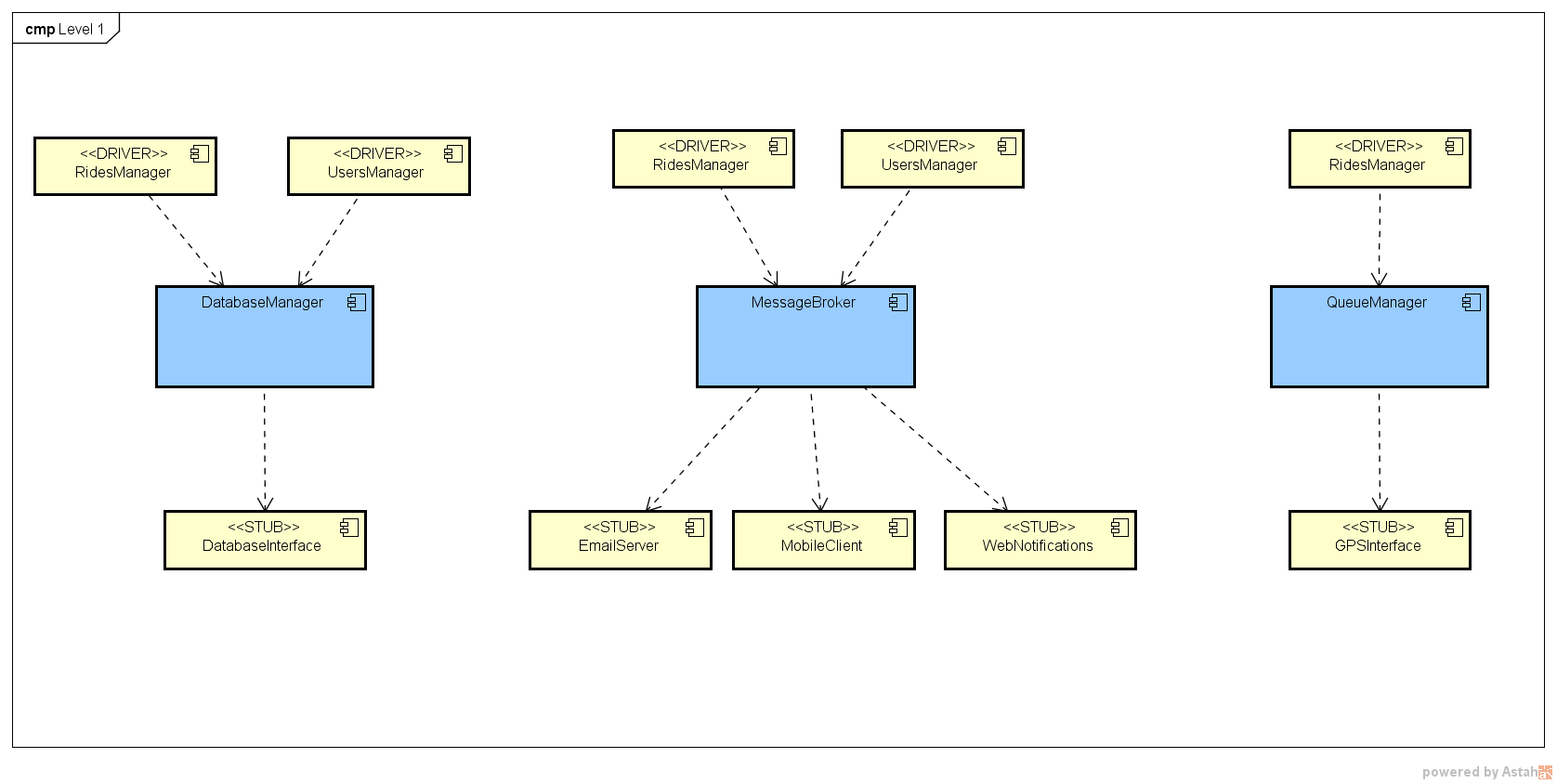
The blue components in the images represent the components that are actually tested at that level.

### Application Server

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

#### Level 1 (DatabaseManager, MessageBroker, QueueManager)

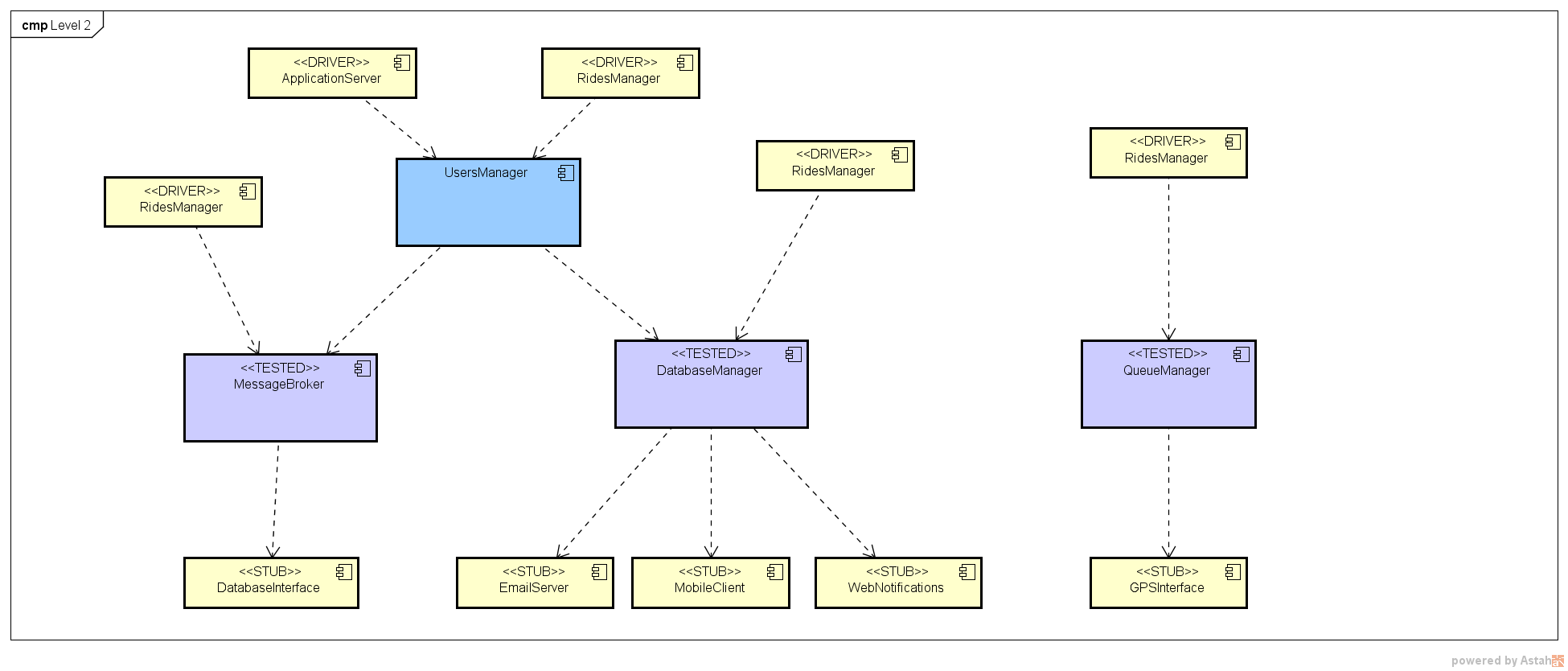
*DatabaseManager*, *QueueManager* and *MessageBroker* are components that can be tested independently in any order. The image shows the required interfaces’ stubs and the components’ drivers.



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

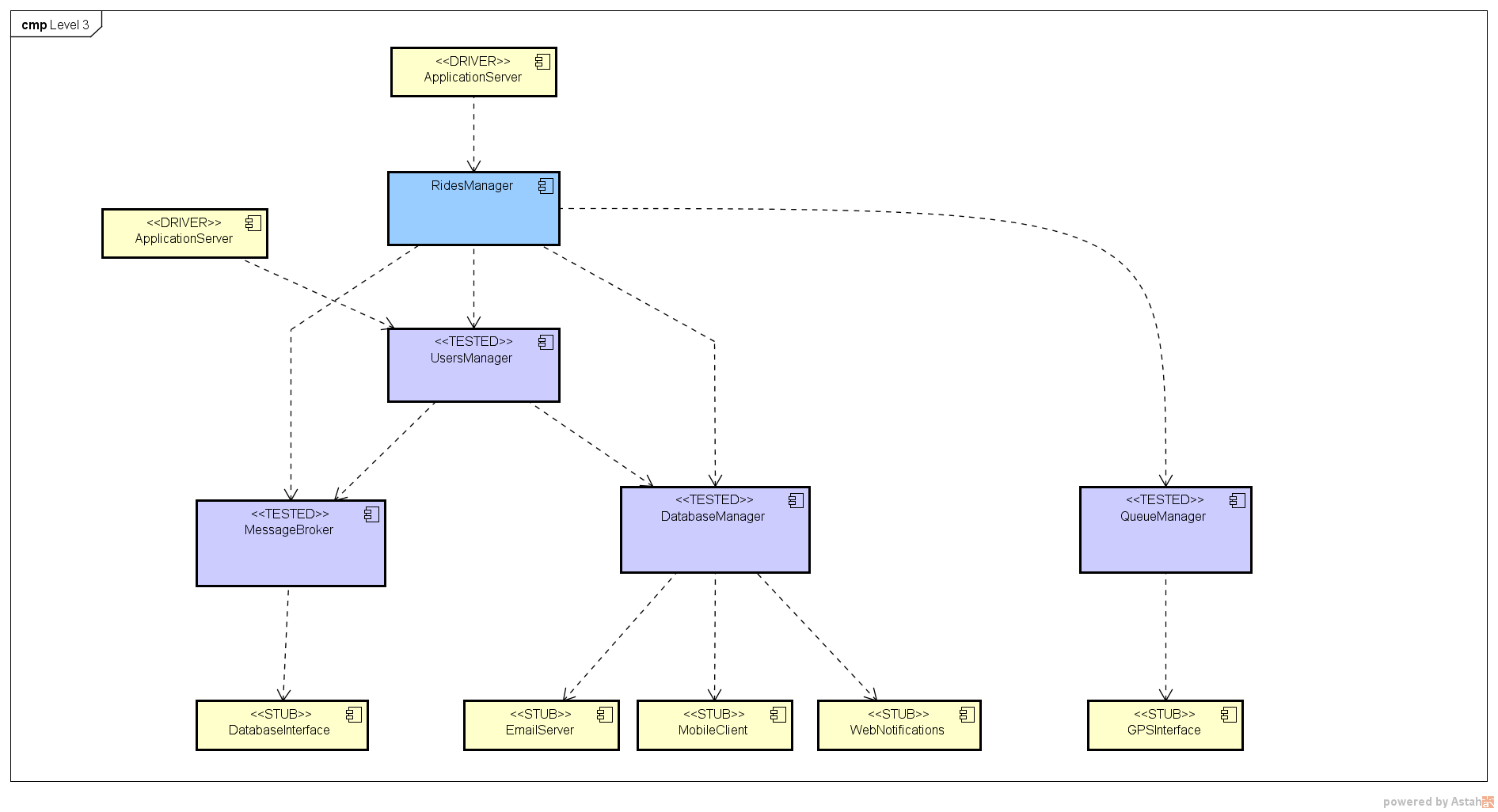
#### Level 2 (UsersManager)

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



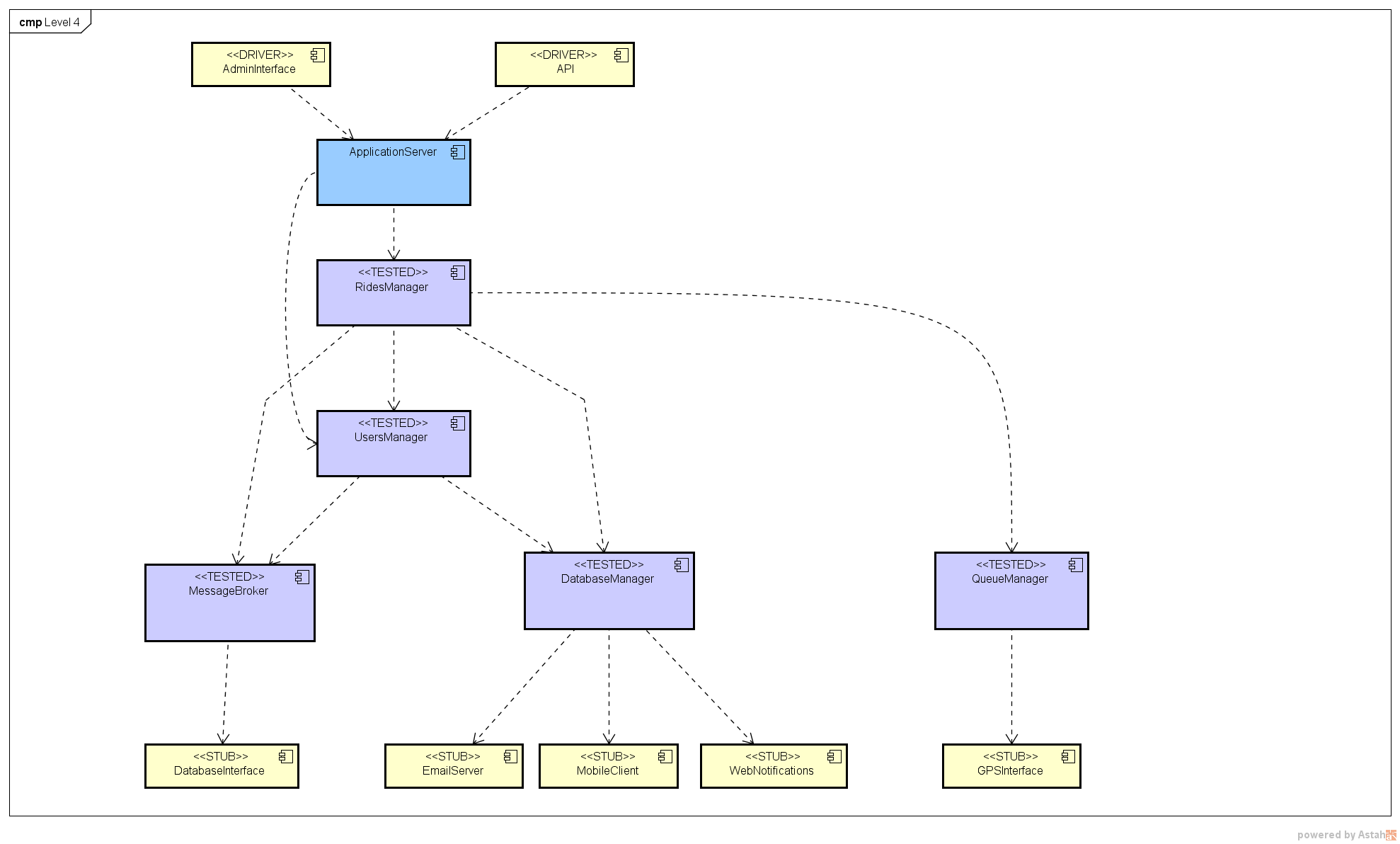
#### Level 3 (RideManager)

Now the RidesManager is tested.

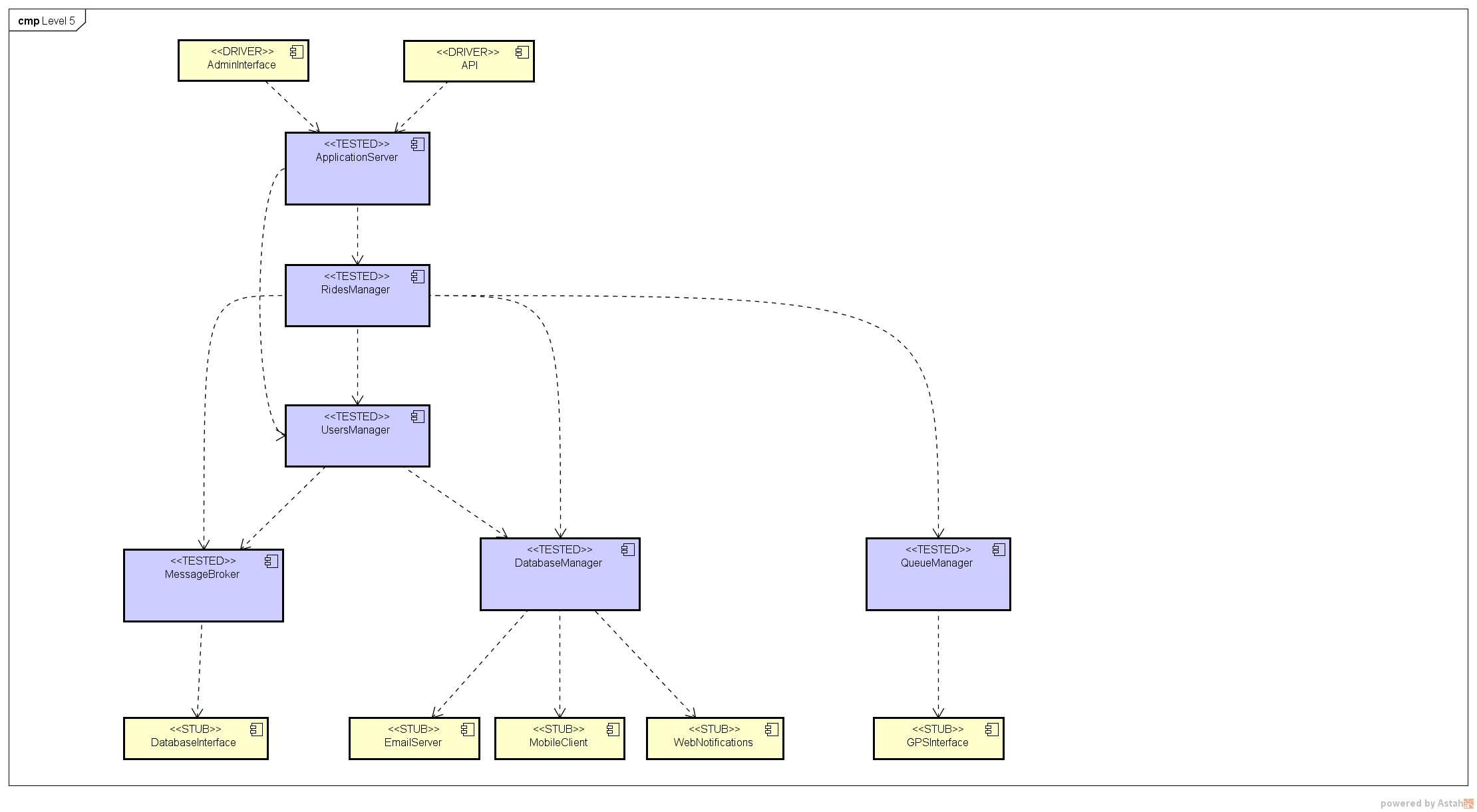


#### Level 4 (ApplicationServer)

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

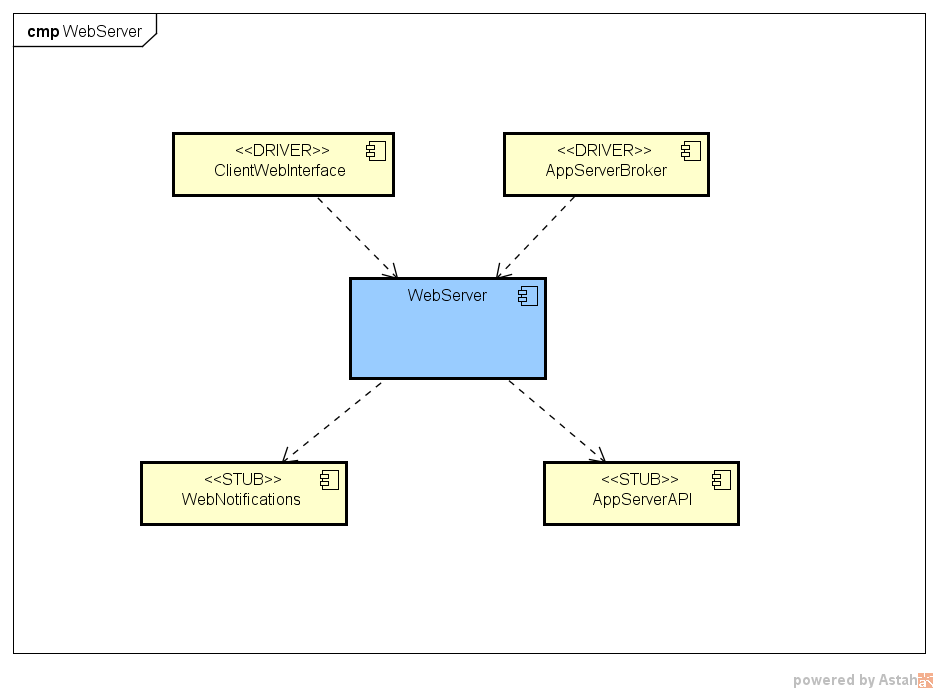


#### Final result

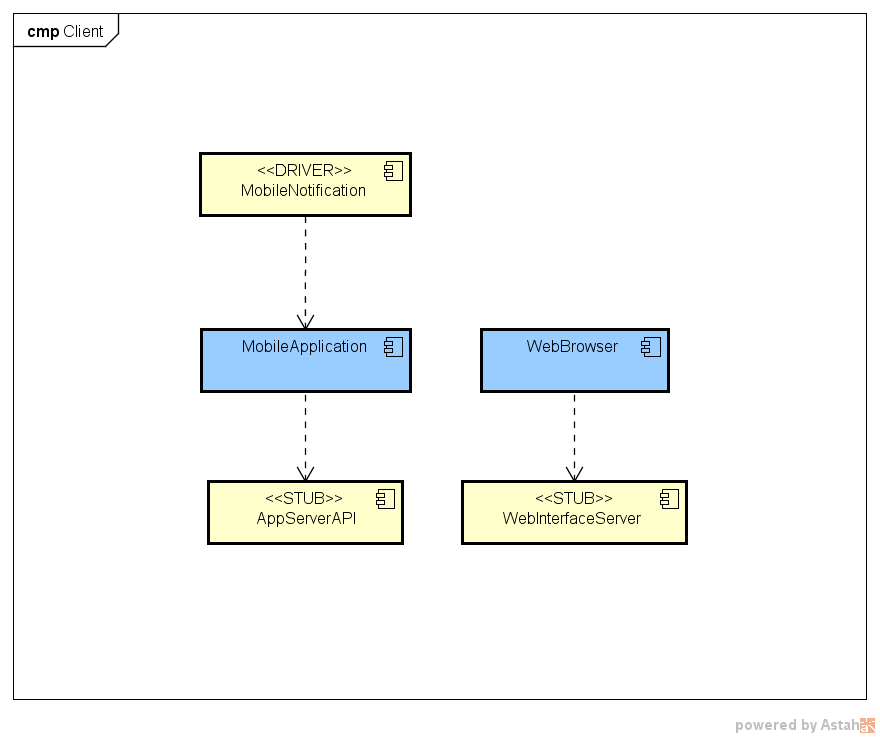


### Web Server

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



### Client



# Individual Steps and Test Description

# Tools and test equipment required

# Program Stubs and Test Data Required

# Appendix

## Hours of work

Alessandro Pozzi ~25 hours

Marco Romani ~25 hours

## 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.