Integration test document - v1.0

Gianpaolo Branca Luca Butera Andrea Cini

POLITECNICO
MILANO 1863

Contents

1 Introduction

1.1 Purpose

The aim to this document is to structure a document an integration testing strategy for the system we are going to develop. This activity is crucial for the success of the project due to its highly distributed and partitioned nature. Our intention is to define a clear plan of the procedure to follow referring to well-known practice for testing, keeping in mind the very structure of the system as presented in the Design Document.

1.2 Scope

The system, as designed in the DD, will consist in different components, deployed in different machines and part of the logic will be highly distributed over the single cars meaning that integration of components has to be tested in advance to prevent wrong behavior and bugs hard to fin once the system has been deployed. The main subsystems that we can identify in our system are: the **CarSystem**, the **PWEService**, the **MonitoringWebApp** and the **MobileApp**; in the next sections we are going to identify the steps needed for the final integration testing of these components as well as the tools and the techniques we are going to exploit.

1.3 Definitions

- **Integration test**: the phase in software testing in the single components are combined together and tested in as a whole.
- **Unit test**: Unit testing is a testing activity concerning only one element of the system (a component in our case), while integration testing is about the system as a whole.
- **Bottom-up**: a strategy that starts testing from the lower level to the higher level components, this usually leads to the need of having a lot of drivers, but only a few stubs.
- **Stubs and drivers**: elements that simulate the behavior of components of the system not integrated yet. In particular stubs simulate functionalities and drivers simulate requests.
- **Arquillian**: a tool for the integration testing of JEE applications.
- **JUnit**: a framework for running unit tests of Java applications.
- **Mockito**: a mocking tool for Java tests.
- **Ripple**: a tool for testing PhoneGap powered applications. More information *here*.

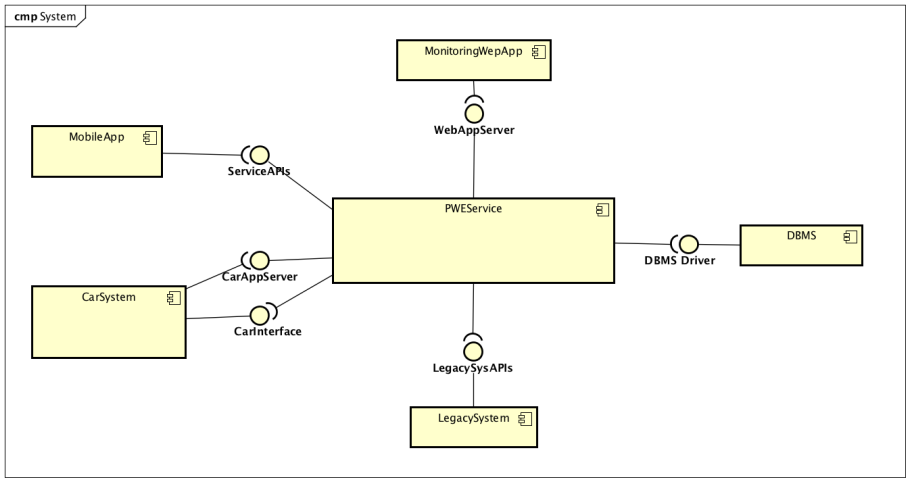
1.4 Abbreviations

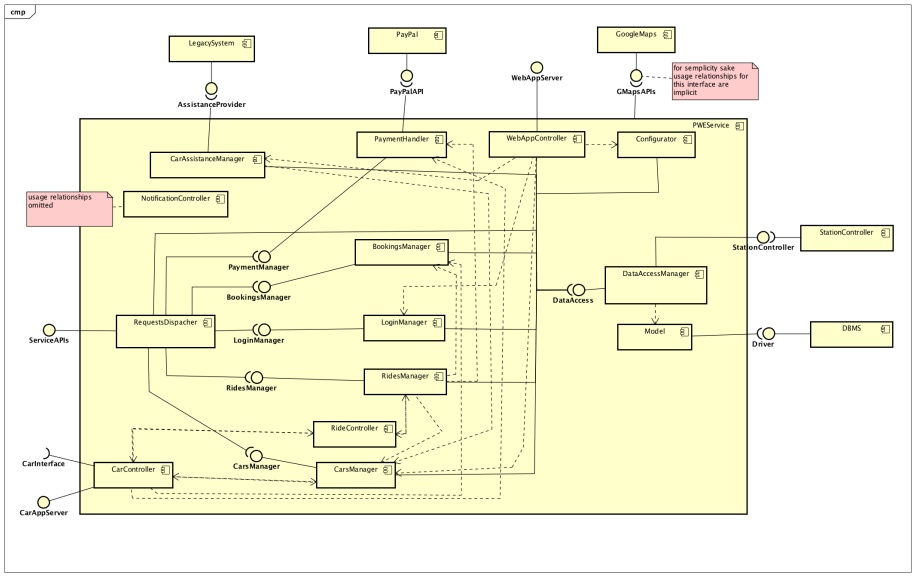
- **GUI:** Graphical unit interfaces
- **GPS:** Global position system
- **N/A:** Not applicable

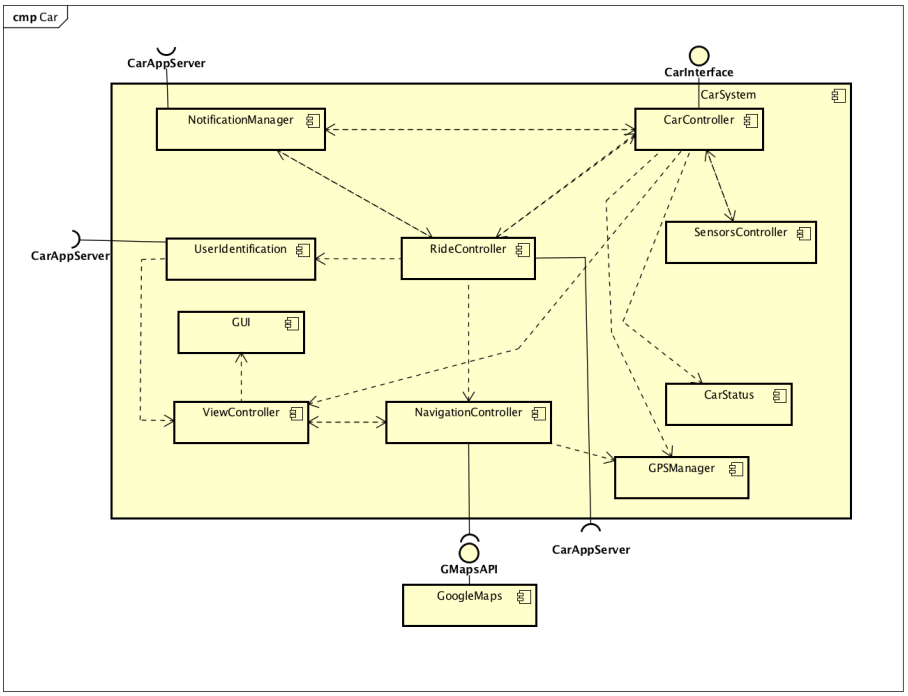
1.5 Reference documents

- The specification document.
- The RASD.
- The DD.
- Integration test document samples on the Beep platform.

2 Integration strategy




cmp
LegacySystem
PayPal
GoogleMaps
AssistanceProvider
PayPalAPI
WebAppServer
GMapsAPIs
for semplicity sake usage relationships for this interface are implicit
PWEService
PaymentHandler
WebAppController
Configurator
CarAssistanceManager
NotificationController
usage relationships omitted
StationController
StationController
PaymentManager
BookingsManager
BookingsManager
DataAccess
DataAccessManager
ServiceAPIs
RequestsDispacher
LoginManager
LoginManager
Model
Driver
DBMS
RidesManager
RidesManager
RideController
CarInterface
CarController
CarsManager
CarsManager
CarAppServer


cmp Car
CarAppServer
CarInterface
CarSystem
NotificationManager
CarController
CarAppServer
UserIdentification
RideController
SensorsController
GUI
CarStatus
ViewController
NavigationController
GPSManager
GMapsAPI
CarAppServer
GoogleMaps

2.1 Entry criteria

Before starting the integration test activity a solid version of both the RASD and the DD must have been provided so that the interactions between the components are reasonably clear and well defined. In our belief, for the integration tests to be effective and meaningful, the followings points have to be reached:

- The data access layer must have been fully developed (**DataAccessManager**, **Model**, **StationController**) and the DBMS integrated.
- At least 70% of the **PWEService** and **CarSystem** components functionalities must be implemented.
- The **MobileApp** and the **MonitoringWebApp** components development is not critical for integration purpose (they offer presentation functionalities), but it must have reached a point in which they provide a way to call every service provided by the central node through its interfaces.
- Agreements with the external services providers must have been reached and the services must be available.
- A reliable stub for the LegacySystem must have been produced.
- **StationController**

We do do expect the integration of PayPal and GoogleMaps to be problematic(the reliability of the services offered by PayPal and Google is not in doubt). We think that it will be more effective to test the proper usage of these API during the unit tests of the components that will use them directly, so we will not consider the integration of these external services in the scope of this document. Every single component must be unit tested with at least an 80% branch coverage and its main functionalities fully developed before going into the integration test phase.

2.2 Elements to be integrated

Starting from the High level component view of our system, the subsystems to be considered for integration testing are integrated are:

- The **PWEService** the **CarSystem** are the most critical components to be tested.
- The **MonitoringWebApp** and the **MobileApp**, as already mentioned, are not essential for the integration test progression as they can be easily substituted by simple drivers until the system integration test.
- Assuming that the **LegacySystem** works correctly a stub seems rather good for our purposes.
- The integration of the **DBMS** should be straightforward with the usage of the JPA framework.

2.3 Integration test strategy

For our integration test we will use a mixed approach because, while going always bottom-up or always top-down will give us the benefit of a simpler integration plan, a more dynamic approach based on the specific group of components under consideration will allow us to test in more effective and meaningful way. At the start we will focus on the integration of the subcomponents of the two main subsystem that we have identified that will be performed in parallel:

- **CarSystem**: At first the tests will be carried out on a virtual machine to simulate the car environment with a stub and a driver for the **SensorsController**, later on, after the unit test of the sensor controller in a real vehicle, the whole Car Application will be deployed on a car and the integration with the **SensorsController** properly tested.
- **PWEService**: To test the components of the central node we will use a strategy based on the bottom-up approach, starting from the back-end going towards the front-end:
  - Step 1: we will test the integration of the component of the data access layer.
  - Step 2: we will test the integration of the internal components(the ones which do not realize any interface accessible from external components) with the data access components.
  - Step 3: we will test the integration of the internal components within each other.
  - Step 4: we will test the integration of the most external components with the rest of the system.

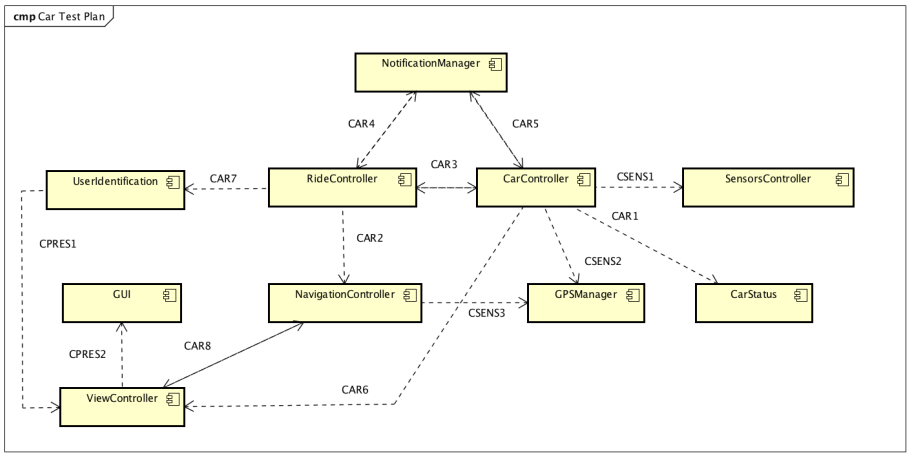
2.4 Sequence of component integration

The following test are meant to be in context of a procedure that replacing on stub/driver after another with the real components will lead to test of the integration of the whole system. However, in some cases, the single tests can be useful at the level of unit-testing and in this case most of them are easy to reproduce using stubs and drivers for the part of the components (these kind of testes will not substitute the execution of the test case during the integration test process).

2.4.1 Software integration sequence

In this section we are going to analyze the tests that are needed for the subcomponents of each subsystem. To describe the sequence in which the components will be integrated, we will use diagrams of components linked with labeled and oriented arrows. The label will identify the flow of tests and the order within the flow.

Car System



The **CarSystem** sub-components integration will happen in two steps and for both of the two steps a stub and a driver for the **PWEService** will be needed:

- Step 1, performed in a virtual machine environment: We will follow two parallel flows of test:
  - CAR testes: using stubs for the **SensorsController** and **GPSManager** we will test the critical components of the subsystem in the order highlighted in the diagram(arrows with CAR labels). These components are the most domain specific ones and their integration has to be considered with particular care.
  - CPRES testes: tests for the integration on the components of the presentation layer of the car application, they can be carried out in parallel from the CAR testes.
- Step 2 :
  - After the tests of the step 1 has been performed the application will be deployed in a real vehicles and the integration of the **SensorsController** and **GPSManager** will be tested with the possibility to manipulate the car to simulate mechanical problems(CSENS testes).

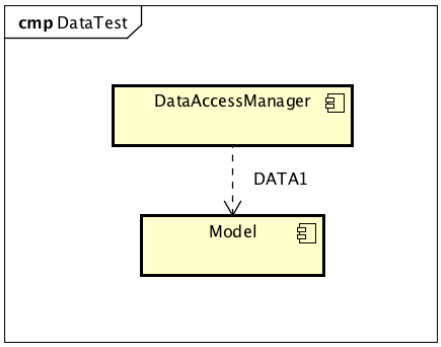
PWEService

In the following section we are going to analyze the necessary steps that we going to perform to test, to do that we will use a diagram for each step. It's important to notice that the diagrams are complementary and that each step incrementally concurs to the integration testing of the entire subsystem. For simplicity sake in each step diagram the connections with the components tested in previous steps are omitted in the diagrams, but obviously we will not use stubs or drivers in the place of components which integration has already been tested (e.g. the in

the second step we won't show the component **Model** even if it is involved in the testing activity). The correct integration of the **NotificationController** will be tested mainly during the integration of the subsystems because at this level it would not be so meaningful.

As already mentioned in the previous sections we will start the integration of the subcomponents of the central node starting from the data access ones.

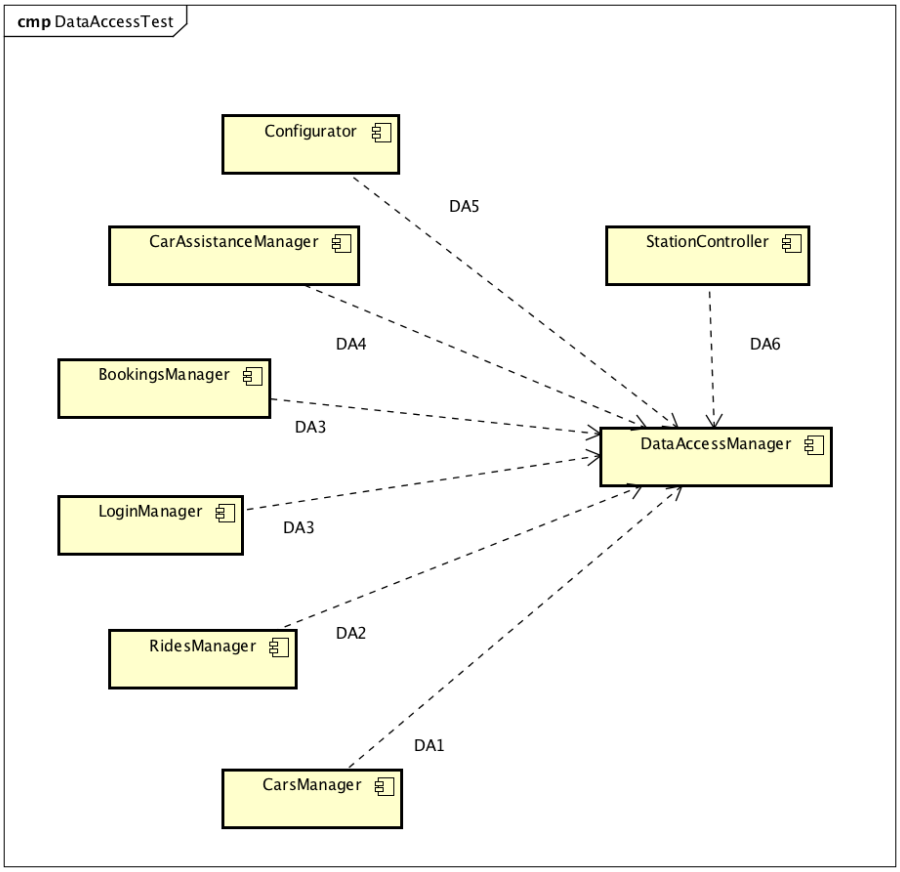
Step 1:



The integration of the **DataAccessManager**(a component its job is to make queries and manipulate the data using the services of the JPA framework) with the **Model** EntityManagers has to be be performed with care to ensure that the set of data that are retrieved and the ones that are generated are correct and consistent.

After the first step has been performed we will move on to the next set of tests that will test if the core components of the system can access the data correctly through the interface provided by the **DataAccessManager** component.

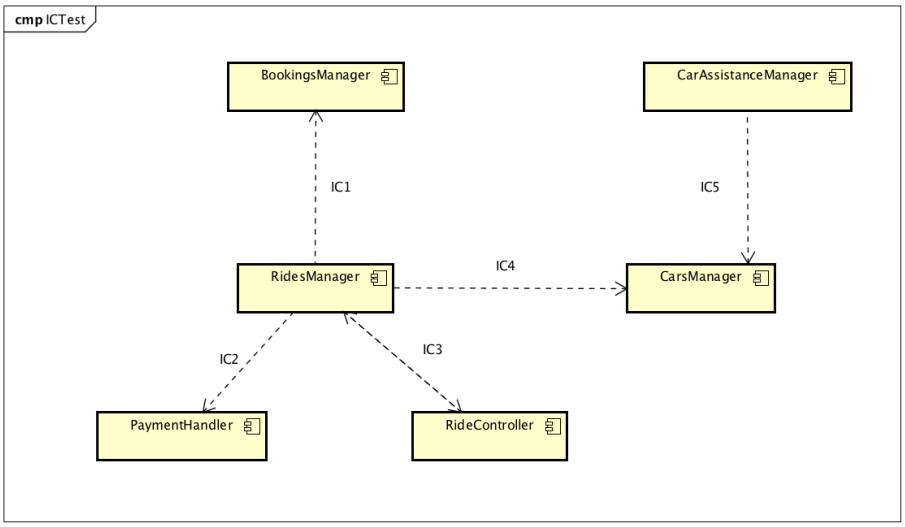
Step 2:



If the integration test of the **DataAccessManager** with the EntityManagers as been performed correctly this part should be quite straightforward and should not be so many problems occurring. The order in which the tests will be performed is not essential and they can be done in parallel, the labels names suggest a possible order.

Step 3:

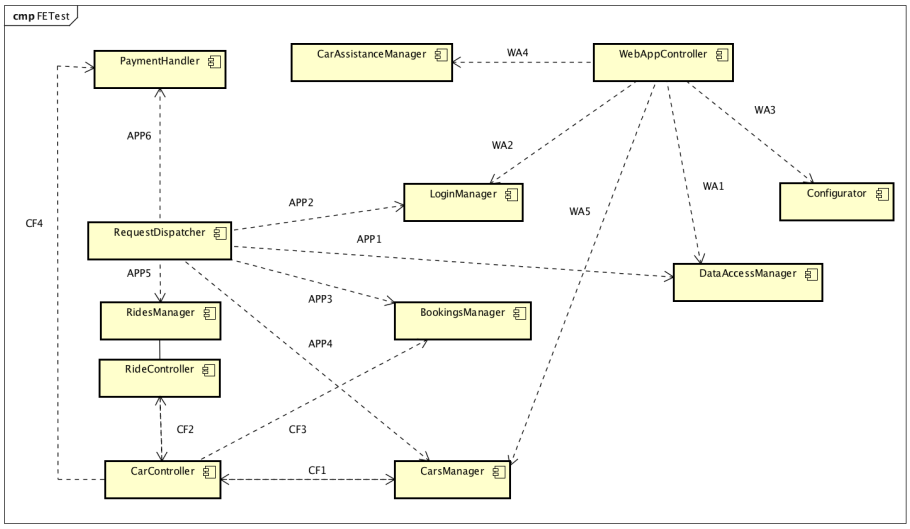
After the operation concerning the data access has been tested the next step will be testing the integration of the internal components.



In this step we are going to test the expected behavior of the core business logic functionalities our system against the reality, a large collection of test cases is needed to correctly verify the robustness of the system up to this point.

Step 4

Finally we are going to test the integration of the front-end components with the rest of the system starting from the integration of the **CarController** component, then the **WebAppController** component and last the component providing the API for the mobile app, the **RequestDispatcher**.



The tests regarding the integration of the **CarController**(CF labels) will be the first to be performed because it is the more independent from the other 2 front-ends, than the **WebAppController**(WA labels) who needs modules the **CarController** to be integrated first to be more meaningful, and last the **RequestDispatcher** which with its integration will be the final stress-test for the cohesion of al the other modules.

2.4.2 Subsystems integration sequence

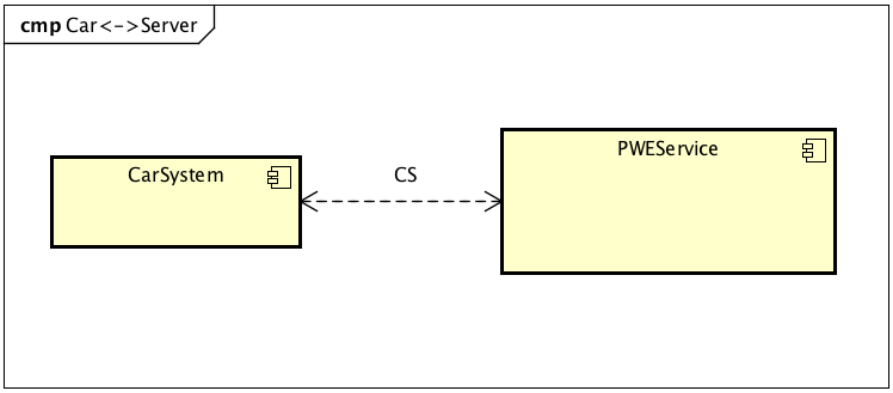
After the car and the central subsystems have been tested properly at the level of their sub-components we will proceed with the integration between the subsystems themselves. Recapping, the 4 subsystems to be integrated are:

- The **PWEService**
- The **CarSystem**
- The **MobileApp**
- The **MonitoringWebApp**

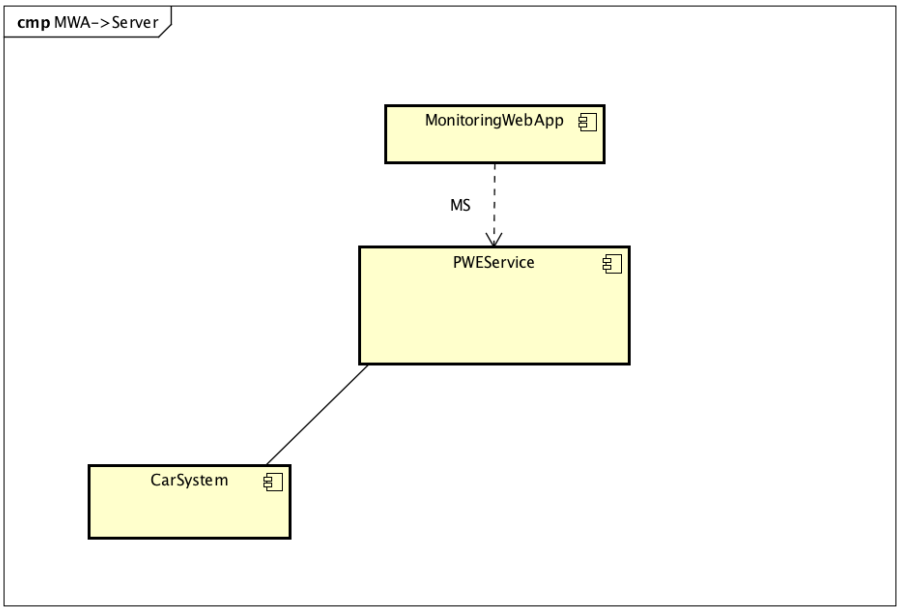
We will proceed with the integration as it fallows(bottom-up):

- **Starting point (Step 0)**: We will use the **PWEService** as the starting element of the integration and will go on substituting one stub/driver of the other components at a time with the real component.
- **Step 1**: The first component we are going to integrate is the **CarSystem**. This integration has to be performed first so that the next integration tests (that otherwise would have needed a stub and a driver for the **CarSystem**) will be more meaningful. This step is probably the most critical (functionally and performance wise) over the whole integration testing activity as the interaction between these two components are the most

complex ones. After the testes have been carried out in a virtual machine to emulate the car environment a test on a real vehicle will be needed.

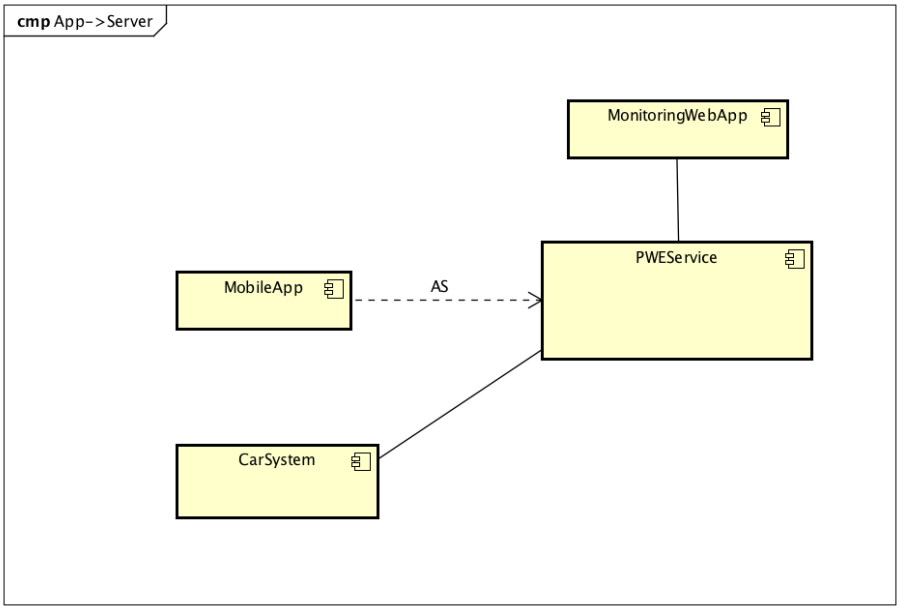


- **Step 2**: This integration tests the correct interaction between the **MonitoringWebApp** and the **PWEService** through the provided interface. We do not expect particular difficulties in this test phase. This test could be iterated multiple times as the refinement of the web application pages goes on.



- **Step 3**: The final step will be the integration of the **MobileApp**, as for the web app, this test should be straightforward (if the other test as been carried out with care). Due to the flexibility of Cordova part on the test

can be performed on a browser(using the Chrome extension Ripple, for example), then on simulators (iOS, android) and finally on physical devices. This test will prove the robustness of the RESTful api provided by the server.



3 Individual steps and test description

4 Tools and Test equipment

4.1 Tools

We will use reliable and well-known tools to make our testing activity as effective as possible. We will use **Arquillian** will be our "best friend" for testing the proper behavior of our system and its proper integration with Glassfish. Arquillian will also allow as to verify that the right components are being injected and that the interactions with the DBMS are correct. Other than Arquillian, we will use **JUnit**(on which Arquillian relies) for basic testing functionalities and **Mockito** to mock the components before their integration with the rest of the system(more details in the stubs section of the document). **Ripple**, together with the tools provided in the iOS and android SDKs, will be an useful tools to test our PhoneGap application (note that once the executables has been produced by the PhoneGap engine they can be executed in standard simulation/emulation environments).

4.2 Test equipment

The final tests have to be performed on specific platform in specific platform.

The **CarSystem** after the initial tested ha to be tested on at least one car from all the different models the company wants to use. We will have to make sure that every car in which our application will be deployed uses a compatible OBD protocol.

For the **MobileApplication**, after a first phase in which the tools mentioned above will be used to emulate the execution environment, a series of tests on real devices will be needed. The version of the OS and the display diagonal are not fundamental for the integration test(this aspect concerns more the unit-testing activity), but at least a test for each of the supported OS versions will be has to be performed.

The **MonitoringWebApp** will be properly tested, even during the integration phase, on all the most used browsers(Chrome, Firefox, Safari, Edge and InternetExplorer).

The central node has to tested on the **GlassFish** application server with **Apache Web Server** as load balancer on **Ubuntu Server** with **MySQL** as DBMS, as stated in the DD.

5 Program stubs/drivers and test data required

5.1 Stubs

Station Controller

Usages:
Description: this stub is used to test the informations retrieved by the Recharging areas, because using a real net of recharging station would be very expensive and would slow down the test.

Car Service

Usages:
Description: this stub is used to test the server independently from the the cars, so that the testing for the systems can proceed in parallel.

Car App server

Usages:
Description: this stub is used to test the car system independently from the the server, so that the testing for the car can proceed in parallel.

Legacy System

Usages:
Description: this stub is used for simulate the forwarding of a request to the legacy server, because sending many fake requests can interfere with the other tasks of the company.

PayPal

Usages:
Description: this stub simulate the interaction with PayPal API, avoiding the usage of real money transfers for testing.

Sensor Controller

Usages:
Description: this stub simulate the interaction with the car sensors, avoiding to break cars each time we want to test if an assistance request is properly sent.

GPS Manager

Usages:
Description: this stub simulate the interaction with the GPS antenna and return a fake positions, simulating rides. Obviously driving a real car for integration testing is very expensive in terms of money and time.

5.2 Test Data

We will populate the data base with fake users, cars, and safe areas. they will be generated in an automatic way with Arquillian

6 Effort spent