**DOCUMETATION ON PHPUnit**

Table of Contents

[**1.Overview**3](bookmark://_Toc43166710)

[**2.What is Unit Tests**3](bookmark://_Toc43166711)

[**3.What is PHPUnit**3](bookmark://_Toc43166712)

[**4.Implementation**4](bookmark://_Toc43166713)

[**a.Global Installation**4](bookmark://_Toc43166714)

[**b.Composer**4](bookmark://_Toc43166715)

[**c.Webserver**5](bookmark://_Toc43166716)

[**d.Writing Tests for PHPUnit**6](bookmark://_Toc43166717)

[**e.Composing a Test Suite Using XML Configuration**7](bookmark://_Toc43166718)

1. **Overview**

This document is the brief introduction and implementation of PHPUnit and its implementation in Kommed Project.

1. **What is Unit Tests**

         Unit Tests are primarily written as a good practice to help developers identify and fix bugs, to refactor code and to serve as documentation for a unit of software under test. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, unit tests find problems early in the development cycle.

1. **What is PHPUnit**

PHPUnit is a [unit testing](https://en.wikipedia.org/wiki/Unit_testing) [framework](https://en.wikipedia.org/wiki/Software_framework) for the [PHP programming language](https://en.wikipedia.org/wiki/PHP). PHPUnit is based on the idea that developers should be able to find mistakes in their newly committed code quickly and assert that no [code regression](https://en.wikipedia.org/wiki/Regression_testing) has occurred in other parts of the code base. Much like other [unit testing](https://en.wikipedia.org/wiki/Unit_testing) frameworks, PHPUnit uses [assertions](https://en.wikipedia.org/wiki/XUnit#Assertions) to verify that the behaviour of the specific component or unit being tested behaves as expected.

1. **Implementation**
2. **Global Installation**

Please note that it is not recommended to install PHPUnit globally, as /usr/bin/phpunit or /usr/local/bin/phpunit, for instance.

Instead, PHPUnit should be managed as a project-local dependency.

1. **Composer**

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you. It manages them on a per-project basis, installing them in a directory (e.g. vendor) inside your project.

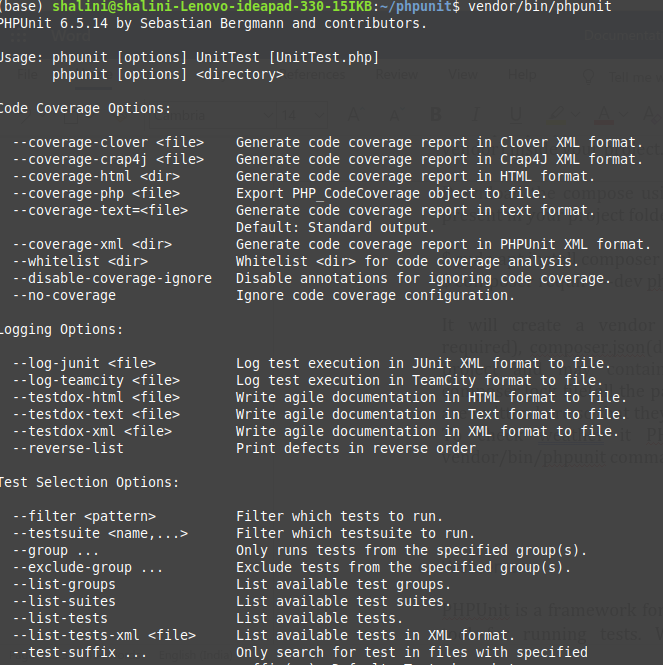
Download the compose using following command (should be present in your project folder):-

$ sudo apt install composer

$ composer require --dev phpunit/phpunit ^9.2

It will create a vendor folder(contains all the libraries required), composer.json(describes the dependencies of your project and may contain other metadata as well) and composer.lock file(all the packages and exact versions of them are written here so that they can’t be updated automatically ).

To check weather it PHPUnit is working or not run vendor/bin/phpunit command.



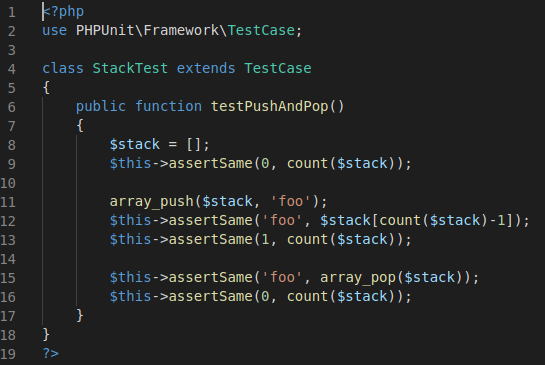
1. **Webserver**

PHPUnit is a framework for writing as well as a command line tool for running tests. Writing and running tests is a development-time activity. There is no reason why PHPUnit should be installed on a webserver.

If you upload PHPUnit to a webserver then your deployment process is broken. On a more general note, if your vendor directory is publicly accessible on your webserver then your deployment process is also broken.

1. **Writing Tests for PHPUnit**

Create a folder inside your project folder and within that folder write all your test files. In Kommed Project unitTests is the name of the folder where all the test files are written.



Above program is just a basic program to check basic array operation

Line 2 shows the library for the framework.

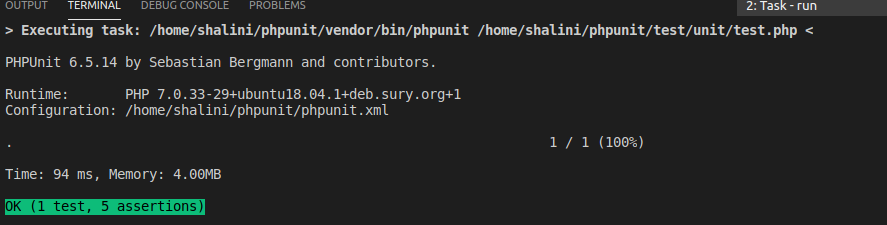
Line 4 create the class which inherit TestCase class from the above library.

Line 6 create function and its name start with test. Whenever the test function is created it always start with test.

Line 9 we use inbuilt function assertSame which checks the 2 values are same or not. There are many such functions and are called assertions.

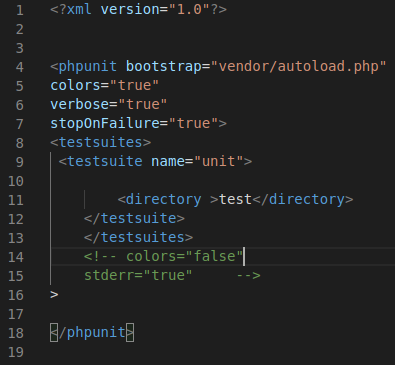
Output:-

For running the above test file open the project in the VS code. Install the extension Better PHPUnit . Open your file which you want to run and press shift+ctrl+p and type run then your file will run



1. **Composing a Test Suite Using XML Configuration**

This file is create within the project folder. In kommed project it is inside the unitTests folder (configuration.xml)



Line 4 here it starts with phpunit tag. Bootstrap here is the starting point that is the autoloader file.

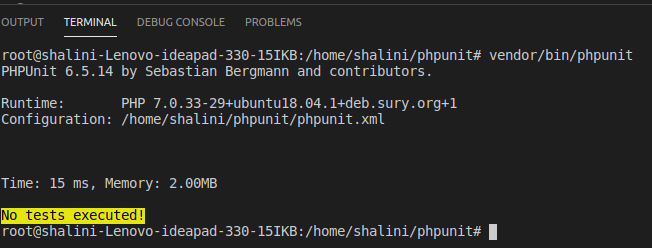
Line 5 gives colour to the execution line in the output (yellow colour in the output)

Line 6 it adds the information to the output (Runtime and configuration)

Line 8 is for running test suites.

Output:-

For running the above file open the terminal. you should be in your project directory. Then run the command vendor/bin/phpunit.



1. **Fixtures**

One of the most time-consuming parts of writing tests is writing the code to set the world up in a known state and then return it to its original state when the test is complete. This known state is called the fixture of the test.

PHPUnit supports sharing the setup code. Before a test method is run, a template method called setUp() is invoked. setUp() is where you create the objects against which you will test. Once the test method has finished running, whether it succeeded or failed, another template method called tearDown() is invoked. tearDown() is where you clean up the objects against which you tested.

1. **Risky Tests**

* **Useless Tests**

PHPUnit is by default strict about tests that do not test anything. This check can be disabled by using the ‘*--dont-report-useless-tests'* option on the [command line](https://phpunit.readthedocs.io/en/9.2/textui.html#textui-clioptions) or by setting ‘*beStrictAboutTestsThatDoNotTestAnything="false"* ‘ in PHPUnit’s [configuration file](https://phpunit.readthedocs.io/en/9.2/configuration.html#appendixes-configuration).

* **Unintentionally Covered Code**

PHPUnit can be strict about unintentionally covered code. This check can be enabled by using the ‘*--strict-coverage*' option on the [command line](https://phpunit.readthedocs.io/en/9.2/textui.html#textui-clioptions) or by setting ‘*beStrictAboutCoversAnnotation="true"’*in PHPUnit’s [configuration file](https://phpunit.readthedocs.io/en/9.2/configuration.html#appendixes-configuration).

* **Output During Test Execution**

PHPUnit can be strict about output during tests. This check can be enabled by using the ‘*--disallow-test-output'* option on the [command line](https://phpunit.readthedocs.io/en/9.2/textui.html#textui-clioptions) or by setting *‘beStrictAboutOutputDuringTests="true"’*in PHPUnit’s [configuration file](https://phpunit.readthedocs.io/en/9.2/configuration.html#appendixes-configuration).

* **Global State Manipulation**

PHPUnit can be strict about tests that manipulate global state. This check can be enabled by using the --strict-global-state option on the [command line](https://phpunit.readthedocs.io/en/9.2/textui.html#textui-clioptions) or by setting beStrictAboutChangesToGlobalState="true" in PHPUnit’s [configuration file](https://phpunit.readthedocs.io/en/9.2/configuration.html#appendixes-configuration).

1. **Assertions**

An assertion is a [predicate](https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)) connected to a point in the program, that always should evaluate to true at that point in code execution.

Some Assertion of PHPUnit are defined below: -

* **assertSame()**

assertSame(mixed $expected, mixed $actual[, string $message = ''])

Reports an error identified by $message if the two variables $expected and $actual do not have the same type and value.

* **assertCount()**

assertCount($expectedCount, $haystack[, string $message = ''])

Reports an error identified by $message if the number of elements in $haystack is not $expectedCount.

assertNotCount() is the inverse of this assertion and takes the same arguments.

* **assertContainsOnly()**

assertContainsOnly(string $type, iterable $haystack[, boolean $isNativeType = null, string $message = ''])

Reports an error identified by $message if $haystack does not contain only variables of type $type.

$isNativeType is a flag used to indicate whether $type is a native PHP type or not.

assertNotContainsOnly() is the inverse of this assertion and takes the same arguments.

* **assertEqualsIgnoringCase()**

assertEqualsIgnoringCase(mixed $expected, mixed $actual[, string $message = ''])

Reports an error identified by $message if the two variables $expected and $actual are not equal.

Differences in casing are ignored for the comparison of $expected and $actual.

assertNotEqualsIgnoringCase() is the inverse of this assertion and takes the same arguments.

* **assertLessThan()**

assertLessThan(mixed $expected, mixed $actual[, string $message = ''])

Reports an error identified by $message if the value of $actual is not less than the value of $expected.

1. **Test Doubles**

Sometimes it is just plain hard to test the system under test (SUT) because it depends on other components that cannot be used in the test environment. This could be because they aren’t available, they will not return the results needed for the test or because executing them would have undesirable side effects. In other cases, our test strategy requires us to have more control or visibility of the internal behavior of the SUT.

When we are writing a test in which we cannot (or chose not to) use a real depended-on component (DOC), we can replace it with a Test Double. The Test Double doesn’t have to behave exactly like the real DOC; it merely must provide the same API as the real one so that the SUT thinks it is the real one!

The createStub($type), createMock($type), and getMockBuilder($type) methods provided by PHPUnit can be used in a test to automatically generate an object that can act as a test double for the specified original type (interface or class name). This test double object can be used in every context where an object of the original type is expected or required.

* Stubs

The practice of replacing an object with a test double that (optionally) returns configured return values is referred to as *stubbing*. You can use a *stub* to “replace a real component on which the SUT depends so that the test has a control point for the indirect inputs of the SUT. This allows the test to force the SUT down paths it might not otherwise execute”.

* Mock Objects

The practice of replacing an object with a test double that verifies expectations, for instance asserting that a method has been called, is referred to as *mocking*.