

# PHPUnit Documents

## Table Of Content

1. [Types of tests](#)
2. [What is PHPUnit](#)
3. [Your first unit test](#)
4. [The end](#)

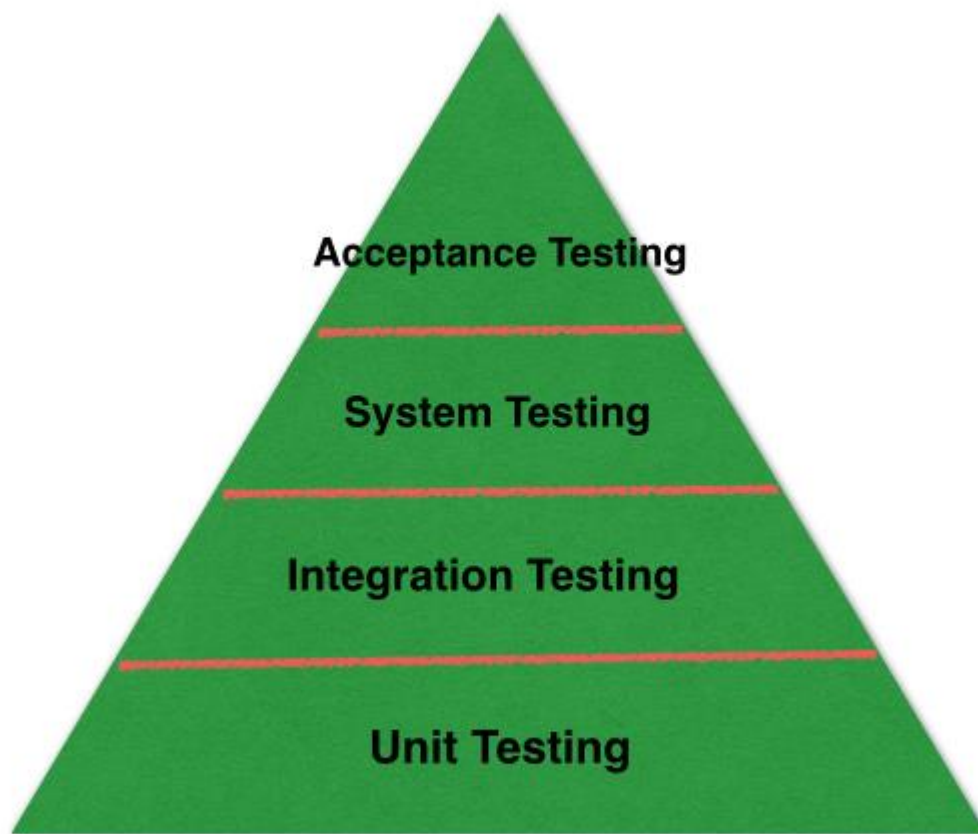
### 1. Types of tests

Before we dive into PHPUnit, let's understand different types of tests. Depends on how you want to categorize it, there are all kinds of tests in software development.

We would like to categorize tests based on the level of specificity of it. According to [Wikipedia](#). There are generally four recognized levels of tests:

- Unit testing: It tests the smallest unit of functionality. From a developer's point of view, its focus is to make sure one function does what it is supposed to do. Thus it should have minimum or no dependence on other function/class. It should be done in memory, which means it should not connect to database, access the network or use the file system and so on. Unit testing should be as simple as possible.
- Integration testing: It combines units of codes and tests the combination works correctly. It is built on top of unit testing and it is able to catch bugs we could not spot by unit testing. Because integration testing checks if Class A works with Class B.
- System testing: It is created to simulate the real time scenarios in a simulated real life environment. It is built on top of integration testing. While integration testing ensure different parts of system work together. System testing ensures the whole system works as user expected before sending it to acceptance testing.
- Acceptance testing: When tests above are for developers at development stage. Acceptance tests are actually done by the users of the software. Users do not care about the internal details of the software. They only care how the software works.

If we put types of testing in a pyramid. It looks like as below:



## 2. What is PHPUnit

From the pyramid above, we can tell that, Unit testing is the building bricks of all other testing. When we build a strong base, we are able to build a solid application. However writing tests manually and running through them each time you make a change is tedious. What if there is a tool to automate the process for you, it will become absolutely more enjoyable to write tests.

This is where PHPUnit comes in. PHPUnit is currently the most popular PHP unit testing framework. Besides providing features such as mocking objects, it provides code coverage analysis, logging and tons of other powerful features.

Let's get it installed in our systems.

1. Download: PHPUnit is distributed in a PHAR(PHP Archive) file, download it [here](#).
2. Add it to system \$PATH: after downloading the PHAR file, make sure it is executable and add it to your system \$PATH. So that you can run it anywhere.

Assuming you are on OSX machine. The process can be done in your terminal via commands below:

```
wget https://phar.phpunit.de/phpunit.phar
chmod +x phpunit.phar
sudo mv phpunit.phar /usr/local/bin/phpunit
```

If you have done everything right, you should be able to see PHPUnit version by typing command below in your terminal.

```
phpunit --version
```

### 3. Your first unit test

Time to create your first unit test! Before that, we need a class to test. Let's create a very simple class called **Calculator**. And write a test for it.

Create a file with the name of "Calculator.php" and copy the code below to the file. This Calculator class only has a **Add** function. :

```
?
1
2     <?php
3     class Calculator
4     {
5         public function add($a, $b)
6         {
7             return $a + $b;
8         }
9     }
10
```

Create the test file "CalculatorTest.php", and copy the code below to the file. We will explain each function in details.

```
?
1     <?php
2     require 'Calculator.php';
3
4     class CalculatorTests extends PHPUnit_Framework_TestCase
5     {
6         private $calculator;
7
8         protected function setUp()
9         {
10             $this->calculator = new Calculator();
11         }
12
13         protected function tearDown()
14         {
15             $this->calculator = NULL;
16         }
17
18         public function testAdd()
19         {
20             $result = $this->calculator->add(1, 2);
21         }
22     }
23
```

```

18         $this->assertEquals(3, $result);
19     }
20 }
21
22
23
24

```

- Line 2: include class file **Calculator.php**. This is the class that we are going to test against, so make sure you include it.
- Line 8: **setUp()** is called before each test runs. Keep in mind, it runs before each test, which means, if you have another test function. It will run **setUp()** before it too.
- Line 13: similar to **setUp()**, **tearDown()** is called after each test finishes.
- Line 18: **testAdd()** is the test function for **add** function. PHPUnit will recognize all functions prefixed with **test** as a test function and run it automatically. This function is actually very straightforward, we first call **Calculator.add** function to calculate the value of 1 plus 2. Then we check if it returns correct value by using PHPUnit function **assertEquals**.

Last part of job is to run PHPUnit and make sure it passes all tests. Navigate to the folder where you have created the test file and run commands below from your terminal:

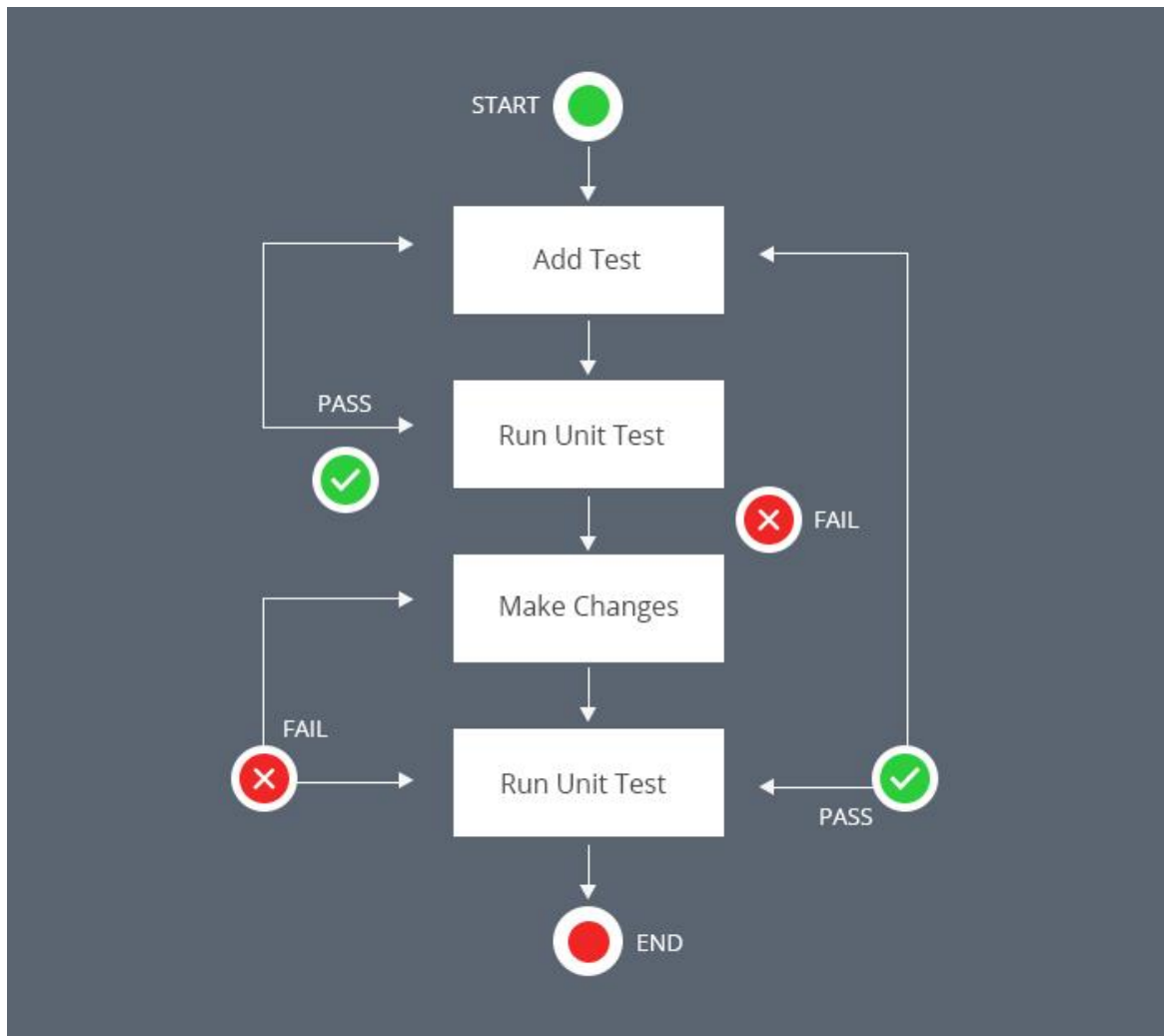
```
phpunit CalculatorTest.php
```

# PHP Unit Testing Using PHPUnit Framework

## What is Unit Testing?

The word `Unit` refers to a block of code, method or an individual or independent class. Unit testing is a software testing process in which code blocks are checked to see whether the produced result matches the expectations. The units are tested by writing a unique test case.

The unit test is generally automatic but could be implemented manually. I have designed a short algorithm to define it visually how unit test work.



## Why do Unit Testing?

Test Driven Development is a practice in which you write a test first and code later to pass the test. This approach makes sure that the number of bugs must be solved while development. The developer tightly writes the test cases first and then the code, as this practice enables him to quickly find out the potential bugs.

- Testing can be automated for speed.
- Higher probability of bug free code
- Easy to understand inherited code
- Refactor code to the point of testability and understandability
- Figure out what went wrong quickly with test cases

- New features can be coded easily
- It also minimizes the cost of change in software.

## PHPUnit Supported Version

Right now, PHPUnit has multiple supported versions and got some massive updates due to the latest PHP versions released. So it is duly compatible with the current and upcoming versions of the PHP. The compatibility table is shown below.

Major Version	PHP Compatibility	Initial Release	Support
PHPUnit 8	PHP 7.2, PHP 7.3, PHP 7.4	February 1, 2019	Support ends on February 5, 2021
<a href="#">PHPUnit 7</a>	PHP 7.1, PHP 7.2, PHP 7.3	February 2, 2018	Support ends on February 7, 2020
<a href="#">PHPUnit 6</a>	PHP 7.0, PHP 7.1, PHP 7.2	February 3, 2017	Support ends on February 1, 2019
<a href="#">PHPUnit 5</a>	PHP 5.6, PHP 7.0, PHP 7.1	October 2, 2015	Support ended on February 2, 2018

## Introducing PHPUnit 7

PHPUnit 7 is released on 2nd Feb 2018 come with interesting features, changes and removes exciting features.

**Note: PHPUnit 7 requires PHP 7.1 or newer version**

## PHPUnit Solves Previous Compatibility Issues

The new version of the library solves the a number of compatibility issues faced in the previous versions of PHPUnit. For a detailed discussion on the topic, please check out the [official documentation](#). I also suggest looking into the [changelog](#) of the library.

## How to Install PHPUnit?

To explain PHP unit testing, I will use PHPUnit library, developed by `Sebastian Burgmen`.

Before using the library, you need to set up a proper environment. If you are doing unit testing in a local environment, you must install XAMPP, Composer, Enable Xdebug, and finally install PHPUnit.

However, thanks to Cloudways [PHP Hosting](#), you do not need to install anything except PHPUnit. Xdebug is easily enabled at a single click in the `**Advance`** tab in Server Management.

**Related:** [How To Host PHP On DigitalOcean](#)

Login to **SSH** terminal by using **Master Credentials** and go to the application folder by using the command

```
1. $ cd applications/qmsfumgabd/public_html/
```

To install PHPUnit from `Composer`, run the following command in SSH terminal.

```
1. public_html$ composer require --dev phpunit/phpunit ^7
```

For installing from **composer.json**, add the following lines of code.

```
1. {  
2.   "require-dev": {  
3.     "phpunit/phpunit": "^7"  
4.   },  
5.   "autoload": {  
6.     "psr-0": {
```

```
7.  "UnitTestFiles": ""
8.  }
9.  }
10. }
```

You can also install PHPUnit manually by downloading the latest version of PHPUnit from the [official Website](#).

## How to Check PHP Version?

If you need to check your PHP Version so type following command on SSH Terminal

```
1. public_html$ php -v
```

## Directory Structure

Before starting the test process using PHPUnit, a proper directory structure for the files to be tested is required. At present, the directory structure will look like this.

```
1. |--vendor
2. |--composer.json
3. |--composer.lock
4. |--index.php
```

I will create a new folder for the files to be tested. Create a folder ``UnitTestFiles``. In this folder, create a subfolder ``Test``. Create a new file ``phpunit.xml`` in this subfolder and add the following code to it.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <phpunit bootstrap="vendor/autoload.php" colors = "true" verbose="true" stopOnFailure="false">
3.
4. <testsuites>
5. <testsuite name="Application Test Suite">
6. <directory>./UnitTestFiles/Test/</directory>
7. </testsuite>
8. </testsuites>
9. </phpunit>
```

The ``colors="true"`` will show the results in highlighted colors and `<directory>./UnitTestFiles/Test/</directory>` will ask PHPUnit for the location of the files to be tested.



Now the directory structure will look like:

1. |--vendor
2. |--UnitTestFiles/Test
3. |--phpunit.xml
4. |--composer.json
5. |--composer.lock
6. |--index.php

All the project files will be located in the root, while the files to be tested will be located in the **Test** folder

You might also like: [Introduction To Laravel Dusk: Testing Todo App](#)

## Basic Conventions to Write Unit Test Case

Following are some basic conventions and steps for writing tests with PHPUnit:

1. Test File names should have a suffix **Test**. For example, if **First.php** needs to be tested, the name of the test file name will be **FirstTest.php**
2. Similarly, If the class name is ``MyFirstClass`` than the test class name will be ``MyFirstClassTest``.
3. Add **test** as the prefix for method names. For example, if the method name is ``getuser()``, then in test class, it will become ``testgetuser()``. You can also use `@test` annotation in document block to declare it as a testing method.
4. All testing methods are ``public``
5. **MyFirstClassTest** should be inherited from ``PHPUnit\Framework\TestCase``

These are the major ground rules for the PHP unit testing framework. The essential configurations and settings are all setup. It is now time to write the first test case.

## Writing The First Unit Test Case in PHP

I will now start with a very simple test case.

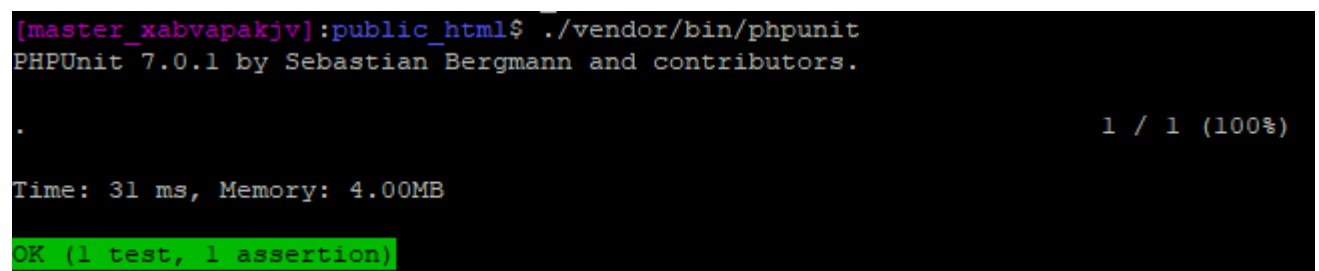
Create a file **FirstTest.php** in **UnitTestFiles/Test**. Add the following code to it.

```
1. ?php
2. namespace UnitTestFiles\Test;
3. use PHPUnit\Framework\TestCase;
4.
5.
6. class FirstTest extends TestCase
7. {
8.     public function testTrueAssetsToTrue()
9.     {
10.         $condition = true;
11.         $this->assertTrue($condition);
12.     }
13. }
14. ?>
```

This is very basic test. In the first line, I mentioned the namespace where the test files are located.

In the second line, I initialized the PHPUnit testing framework. **FirstTest** is the test class that extends Testcase. I have declared a method ``testTrueCheck()`` with prefix **test**.

In the method, I simply declares a variable ``$condition = true``; I will next validate (true or false) this variable with ``assertTrue``. Save the file and run it through the SSH terminal by using the following command



```
[master_xabvapakjv]:public_html$ ./vendor/bin/phpunit
PHPUnit 7.0.1 by Sebastian Bergmann and contributors.

.                                                                    1 / 1 (100%)

Time: 31 ms, Memory: 4.00MB

OK (1 test, 1 assertion)
```

Here is the result of our first unit test. To do another test, pass **false** to the variable ``$condition``. The present result ``$condition = true`` is changed to ``$condition = false``; Run the test command in SSH terminal.

```

Time: 26 ms, Memory: 4.00MB

There was 1 failure:

1) UnitTestFiles\Test\SampleTest::testTrueAssetsToTrue
Failed asserting that false is true.

/home/144478.cloudwaysapps.com/trxbnbphae/public_html/tests/unit/

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.

```

This red line indicates the failure of the test since `assertTrue` reports an error and identified \$condition as false.

Let's look at another example in which a developer has created a code to take input an email address and validate it for the possible use case may be regex or for other email validation rules.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. final class Email
6. {
7. {
8.
9. private $email;
10.
11. private function __construct(string $email)
12. {
13. {
14.
15. $this->ensureIsValidEmail($email);
16.
17. $this->email = $email;
18.
19. }
20.
21. public static function fromString(string $email): self
22. {
23. {
24. return new self($email);
25. }
26.
27. public function __toString(): string
28. {
29. {
30.
31. return $this->email;
32.

```

```

33. }
34.
35. private function ensureIsValidEmail(string $email): void
36.
37. {
38.
39. if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
40.
41. throw new InvalidArgumentException(
42.
43. sprintf(
44.
45. ""%s" is not a valid email address',
46.
47. $email
48.
49. )
50.
51. );
52.
53. }
54.
55. }
56.
57. }

```

I've created a class called **Email** which saves the email value in private variable and pass it to the constructor. The constructor has **ensureIsValidEmail()** method which further checks the email using `FILTER_VALIDATE_EMAIL` and throws an exception if not valid.

Now in the **UnitTestFiles/Test** folder, create a new file called **EmailTest.php** and add the following test code:

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace UnitTestFiles\Test;
6.
7. use PHPUnit\Framework\TestCase;
8.
9.
10.
11. final class EmailTest extends TestCase
12.
13. {
14.
15. public function testCanBeCreatedFromValidEmailAddress(): void
16.
17. {
18.

```

```

19. $this->assertInstanceOf(
20.
21. Email::class,
22.
23. Email::fromString('user@example.com')
24.
25. );
26.
27. }
28.
29. public function testCannotBeCreatedFromInvalidEmailAddress(): void
30.
31. {
32.
33. $this->expectException(InvalidArgumentException::class);
34.
35.
36.
37. Email::fromString('invalid');
38.
39. }
40.
41. public function testCanBeUsedAsString(): void
42.
43. {
44.
45. $this->assertEquals(
46.
47. 'user@example.com',
48.
49. Email::fromString('user@example.com')
50.
51. );
52.
53. }
54.
55. }

```

Here I've used `assertInstanceOf()` method which reports an error identified by `$message` if `$actual` is not an instance of `$expected`.

So I've passed two arguments, the actual email class and the expected email address. The second test will take the input value and throw an invalid exception if the email value is not validated. The third test takes input value and treats it as a string.

Let's run the test on composer by running the following command:

```

1. ./vendor/bin/phpunit --bootstrap vendor/autoload.php Tests/EmailTest

```

## Conclusion

This article explains a basic setup that help you in getting started with PHPUnit for PHP unit testing.