PHPUnit Lab Exercises

PHPUnit Lab

Part 1

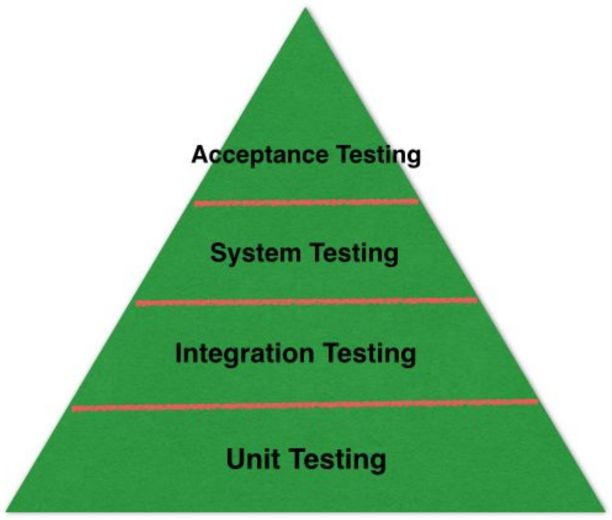
1. Types of tests

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. :

?

```
<?php

class Calculator

{



    public function add($a, $b)

    {

        return $a + $b;

    }



}
```

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          $this->assertEquals(3, $result);
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
```

Part 2

1. When to use data provider

When we write a function, we want to make sure it passes a series of edge cases. The same applies to tests. Which means we will need to write multiple tests to test the same function using different set of data. For instance, if we want to test our Calculator class using different data. Without data provider, we would have multiple tests as below:

?

```
<?php
require 'Calculator.php';

class CalculatorTests extends PHPUnit_Framework_TestCase
{
        private $calculator;

        protected function setUp()
        {
            $this->calculator = new Calculator();
        }

        protected function tearDown()
        {
            $this->calculator = NULL;
        }

        public function testAdd()
```

```
            {
                $result = $this->calculator->add(1, 2);
                $this->assertEquals(3, $result);
            }

            public function testAddWithZero()
            {
                $result = $this->calculator->add(0, 0);
                $this->assertEquals(0, $result);
            }

            public function testAddWithNegative()
            {
                $result = $this->calculator->add(-1, -1);
                $this->assertEquals(-2, $result);
            }

        }
```

In this case, we can use data provider function in PHPUnit to avoid duplication in our tests.

2. How to use data provider

A data provider method return an array of arrays or an object that implements the Iterator interface. The test method will be called with the contents of the array as its arguments.

Some key points as blow when using data provider:

- Data provider method must be public.
- Data provider return an array of a collection data.
- Test method use annotation(@dataProvider) to declare its data provider method.

Once we know the key points. It is actually quite straightforward to use data provider. First we create a new public method, which returns an array of a collection data as arguments of the test method.Then we add annotation to the test method to tell PHPUnit which method will provide arguments.

3. Add data provider to our first unit test

Let's modify our tests above using data provider.

?

```
<?php

require 'Calculator.php';



class CalculatorTests extends PHPUnit_Framework_TestCase

{

    private $calculator;



    protected function setUp()

    {

        $this->calculator = new Calculator();

    }



    protected function tearDown()

    {

        $this->calculator = NULL;

    }



    public function addDataProvider() {

        return array(
            array(1,2,3),

            array(0,0,0),

            array(-1,-1,-2),

        );

    }


    /**

     * @dataProvider addDataProvider

     */

    public function testAdd($a, $b, $expected)

    {

        $result = $this->calculator->add($a, $b);

        $this->assertEquals($expected, $result);

    }


}
```

- line 18: add a data provider method. Take note that a data provider method must be declared as public.
- line 27: use annotation to declare test method's data provider method.

Now run our test again. It should pass. As you can see, we have utilized data provider to avoid code duplication. Instead of writing three test methods for essentially the same method. We now have only one test method.

Part 3

1. When to use test double

As mentioned in the first part of this series. One of PHPUnit's powerful features is test double. It is very common in our code, a function of one class is calling another class's function. In this case, we have a dependency in these two classes. In particular, the caller class has a dependency on calling class. But as we already know in part 1, unit test should test the smallest unit of functionality, in this case, it should test only the caller function. To solve this problem, We can use test double to replace the calling class. Since a test double can be configured to return predefined results, we can focus on testing the caller function.

2. Types of test doubles

Test double is a generic term for objects we use, to replace real production ready objects. In our opinion, it is very useful to categorize test doubles by their purpose. It does not only make it easy for us to understand the test case, but also make our code friendly to other parties.

Accordingly to Martin Fowler's post, There are five types of test double:

1. **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
2. **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production.
3. **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
4. **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
5. **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

3. How to create test double

PHPUnit's method **getMockBuilder** can be used to create any similar user defined objects. Combining with its configurable interface. We can use it to create basically all five types of test doubles.

4. Add test double to our first unit test

It is meaningless to use test double to our calculator test case, since currently the Calculator class has no dependency on other classes. However to demonstrate how to use test double in PHPUnit, we will create a stub Calculator class and test it.

Let's add a test case called **testWithStub** to our existing class:

?

```
    public function testWithStub()
    {
        // Create a stub for the Calculator class.
        $calculator = $this->getMockBuilder('Calculator')
                           ->getMock();

        // Configure the stub.
        $calculator->expects($this->any())
                   ->method('add')
                   ->will($this->returnValue(6));

        $this->assertEquals(6, $calculator->add(100,100));
    }
```

1. **getMockBuilder()** method creates a stub similar to our Calculator object.
2. **getMock()** method returns the object.
3. **expects()** method tells the stub to be called any number of times.
4. **method()** method specifies which method it will be called.
5. **will()** method configures the return value of the stub.