FREE eBook

LEARNING phpunit

#phpunit

Table of Contents

About

Chapter 1: Getting started with phpunit

Remarks

This section provides an overview of what phpunit is, and why a developer might want to use it.

It should also mention any large subjects within phpunit, and link out to the related topics. Since the Documentation for phpunit is new, you may need to create initial versions of those related topics.

Versions

Version	Support ends	Supported in those PHP versions	Release date
5.4	2016-08-05	PHP 5.6, PHP 7.	2016-06-03
4.8	2017-02-03	PHP 5.3, PHP 5.4, PHP 5.5, and PHP 5.6.	2015-08-07

Examples

Create first PHPUnit test for our class

Imagine that we have a class `Math.php` with logic of calculating of fiobanacci and factorial numbers. Something like this:

```
<?php
class Math {
    public function fibonacci($n) {
        if (is_int($n) && $n > 0) {
            $elements = array();
            $elements[1] = 1;
            $elements[2] = 1;
            for ($i = 3; $i <= $n; $i++) {
                $elements[$i] = bcadd($elements[$i-1], $elements[$i-2]);
            }
            return $elements[$n];
        } else {
            throw new
                InvalidArgumentException('You should pass integer greater than 0');
        }
    }

    public function factorial($n) {
        if (is_int($n) && $n >= 0) {
            $factorial = 1;
            for ($i = 2; $i <= $n; $i++) {
                $factorial *= $i;
            }
            return $factorial;
        } else {
            throw new
                InvalidArgumentException('You should pass non-negative integer');
        }
    }
}
```

The simplest test

We want to test logic of methods `fibonacci` and `factorial`. Let's create file `MathTest.php` into the same directory with `Math.php`. In our code we can use **different assertions**. The simplest code will be something like this (we use only `assertEquals` and `assertTrue`):

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

use PHPUNIT_Framework_TestCase as TestCase;
// sometimes it can be
// use PHPUnit\Framework\TestCase as TestCase;

class MathTest extends TestCase{
    public function testFibonacci() {
        $math = new Math();
        $this->assertEquals(34, $math->fibonacci(9));
    }

    public function testFactorial() {
        $math = new Math();
        $this->assertEquals(120, $math->factorial(5));
    }

    public function testFactorialGreaterThanFibonacci() {
        $math = new Math();
        $this->assertTrue($math->factorial(6) > $math->fibonacci(6));
    }
}
```

We can run this test from console with command `phpunit MathTest` and output will be:

```
    PHPUnit 5.3.2 by Sebastian Bergmann and contributors.

...                                                                  3 / 3 (100%)

Time: 88 ms, Memory: 10.50Mb

OK (3 tests, 3 assertions)
```

Using dataProviders

A test method can accept arbitrary arguments. These arguments are to be provided by a **data**

provider method. The data provider method to be used is specified using the `@dataProvider` annotation. :

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

use PHPUNIT_Framework_TestCase as TestCase;
// sometimes it can be
// use PHPUnit\Framework\TestCase as TestCase;

class MathTest extends TestCase {
    /**
     * test with data from dataProvider
     * @dataProvider providerFibonacci
     */
    public function testFibonacciWithDataProvider($n, $result) {
        $math = new Math();
        $this->assertEquals($result, $math->fibonacci($n));
    }

    public function providerFibonacci() {
        return array(
            array(1, 1),
            array(2, 1),
            array(3, 2),
            array(4, 3),
            array(5, 5),
            array(6, 8),
        );
    }
}
```

We can run this test from console with command `phpunit MathTest` and output will be:

```
    PHPUnit 5.3.2 by Sebastian Bergmann and contributors.

......                                                              6 / 6 (100%)

Time: 97 ms, Memory: 10.50Mb

OK (6 tests, 6 assertions)


<?php
require 'Math.php';
use PHPUNIT_Framework_TestCase as TestCase;
// sometimes it can be
// use PHPUnit\Framework\TestCase as TestCase;
```

Test exceptions

We can test whether an exception is thrown by the code under test using method `expectException()`. Also in this example we add one failed test to show console output for failed tests.

```
<?php
require 'Math.php';
use PHPUNIT_Framework_TestCase as TestCase;
// sometimes it can be
// use PHPUnit\Framework\TestCase as TestCase;

class MathTest extends TestCase {
    public function testExceptionsForNegativeNumbers() {
        $this->expectException(InvalidArgumentException::class);
        $math = new Math();
            $math->fibonacci(-1);
    }

    public function testFailedForZero() {
        $this->expectException(InvalidArgumentException::class);
        $math = new Math();
        $math->factorial(0);
    }
}
```

We can run this test from console with command `phpunit MathTest` and output will be:

```
        PHPUnit 5.3.2 by Sebastian Bergmann and contributors.

.F                                                                  2 / 2 (100%)

Time: 114 ms, Memory: 10.50Mb

There was 1 failure:

1) MathTest::testFailedForZero
Failed asserting that exception of type "InvalidArgumentException" is thrown.

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

SetUp and TearDown

Also `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.

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

use PHPUNIT_Framework_TestCase as TestCase;
// sometimes it can be
// use PHPUnit\Framework\TestCase as TestCase;

class MathTest extends TestCase {
    public $fixtures;
    protected function setUp() {
        $this->fixtures = [];
    }

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

    public function testEmpty() {
        $this->assertTrue($this->fixtures == []);
    }
}
```

More info

There are much more great opportunities of `PHPUnit` which you can use in your test. For more info see in official manual

Minimal example test

Given a simple PHP class:

```
class Car
{
    private $speed = 0;

    public getSpeed() {
        return $this->speed;
    }

    public function accelerate($howMuch) {
        $this->speed += $howMuch;
    }
}
```

You can write a PHPUnit test which tests the behavior of the class under test by calling the public methods and check whether they function as expected:

```
class CarTest extends PHPUnit_Framework_TestCase
{
    public function testThatInitalSpeedIsZero() {
        $car = new Car();
        $this->assertSame(0, $car->getSpeed());
    }

    public function testThatItAccelerates() {
        $car = new Car();
        $car->accelerate(20);
        $this->assertSame(20, $car->getSpeed());
    }

    public function testThatSpeedSumsUp() {
        $car = new Car();
        $car->accelerate(30);
        $car->accelerate(50);
        $this->assertSame(80, $car->getSpeed());
    }
}
```

Important parts:

1. Test class needs to derive from PHPUnit_Framework_TestCase.
2. Each test function name should start with the prefix 'test'
3. Use `$this->assert...` functions to check expected values.

Mocking classes

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.

Lets assume we have SomeService to test.

```
class SomeService
{
    private $repository;
    public function __construct(Repository $repository)
    {
        $this->repository = $repository;
    }

    public function methodToTest()
    {
        $this->repository->save('somedata');
    }
}
```

And we want to test if `methodToTest` really calls `save` method of repository. But we don't want to actually instantiate repository (or maybe `Repository` is just an interface).

In this case we can mock `Repository`.

```
use PHPUnit\Framework\TestCase as TestCase;

class SomeServiceTest extends TestCase
{
    /**
     * @test
     */
    public function testItShouldCallRepositorySavemethod()
    {
        // create an actual mock
        $repositoryMock = $this->createMock(Repository::class);

        $repositoryMock->expects($this->once()) // test if method is called only once
             ->method('save')                   // and method name is 'save'
             ->with('somedata');                // and it is called with 'somedata' as a
parameter

        $service = new SomeService($repositoryMock);
        $service->someMethod();
    }
}
```

Example of PHPUNIT with APItest using Stub And Mock

Class for which you will create unit test case. `class Authorization {`

```
/* Observer so that mock object can work. */
 public function attach(Curl $observer)
{
    $this->observers = $observer;
}

/* Method for which we will create test */
public  function postAuthorization($url, $method) {

    return $this->observers->callAPI($url, $method);
}
```

}

Now we did not want any external interaction of our test code thus we need to create a mock object for callAPI function as this function is actually calling external url via curl. `class AuthorizationTest extends PHPUnit_Framework_TestCase {`

```
protected $Authorization;

/**
 * This method call every time before any method call.
 */
protected function setUp() {
    $this->Authorization = new Authorization();
}

/**
 * Test Login with invalid user credential
 */
function testFailedLogin() {

    /*creating mock object of Curl class which is having callAPI function*/
    $observer = $this->getMockBuilder('Curl')
                     ->setMethods(array('callAPI'))
                     ->getMock();

    /* setting the result to call API. Thus by default whenver call api is called via this
function it will return invalid user message*/
    $observer->method('callAPI')
            ->will($this->returnValue('"Invalid user credentials"'));

    /* attach the observer/mock object so that our return value is used */
    $this->Authorization->attach($observer);

    /* finally making an assertion*/
    $this->assertEquals('"Invalid user credentials"',           $this->Authorization-
>postAuthorization('/authorizations', 'POST'));
}
```

```
}
```

Below is the code for curl class(just a sample) `class Curl{ function callAPI($url, $method){`

```
    //sending curl req
}
```

```
}
```

Read Getting started with phpunit online: https://riptutorial.com/phpunit/topic/3268/getting-started-with-phpunit

Chapter 2: Assertions

Examples

Assert an Object is an Instance of a Class

PHPUnit provides the following function to assert whether an object is an instance of a class:

```
assertInstanceOf($expected, $actual[, $message = ''])
```

The first parameter `$expected` is the name of a class (string). The second parameter `$actual` is the object to be tested. `$message` is an optional string you can provide in case it fails.

Let's start with a simple Foo class:

```
class Foo {

}
```

Somewhere else in a namespace `Crazy`, there is a `Bar` class.

```
namespace Crazy

class Bar {

}
```

Now, let's write a basic test case that would check an object for these classes

```
use Crazy\Bar;

class sampleTestClass extends PHPUnit_Framework_TestCase
{

    public function test_instanceOf() {

        $foo = new Foo();
        $bar = new Bar();

        // this would pass
        $this->assertInstanceOf("Foo",$foo);

        // this would pass
        $this->assertInstanceOf("\\Crazy\\Bar",$bar);

        // you can also use the ::class static function that returns the class name
        $this->assertInstanceOf(Bar::class, $bar);

        // this would fail
        $this->assertInstanceOf("Foo", $bar, "Bar is not a Foo");

    }
}
```

Note that PHPUnit gets grumpy if you send in a classname that doesn't exist.

Assert an Exception is Thrown

PHPUnit provides the following functions to watch for thrown exceptions, which were released with 5.2.0:

- `expectException($exception)`
- `expectExceptionMessage($message)`
- `expectExceptionCode($code)`
- `expectExceptionMessageRegExp($messageRegExp)`

These are used to watch for an exception to be thrown and inspect the properties of that exception.

Let's start with a math function that divides (just for simplicity). It will raise an exception if the denominator is zero.

```
function divide($numerator, $denominator) {

    if ($denominator !== 0) {
        return $numerator/$denominator;
    } else {
        throw new \Exception("Cannot divide by zero", 100);
    }

}
```

Now for the test code.

```
class DivideTest extends PHPUnit_Framework_TestCase
{

    public function test_divide() {

        $this->assertSame(2,divide(4,2));

        $this->expectException("Exception");
        $this->expectExceptionCode(100);
        $this->expectExceptionMessage("Cannot divide by zero");
        $this->expectExceptionMessageRegExp('/divide by zero$/');

        // the expectations have been set up, now run the code
        // that should throw the exception
        divide(4,0);

        // The following code will not execute, the method has exited
        $this->assertSame(0,1);

    }

}
```

The `test_divide()` function starts by asserting that the function correctly divided 4 by 2 and answered 2. This assertion will pass.

Next, the expectations for the upcoming exception are set. Notice, they are set before the code that will throw the exception. All four assertions are shown for demonstration purposes, but this is normally not necessary.

The `divide(4,0)` will then throw the expected exception and all the expect* function will pass.

But note that the code `$this->assertSame(0,1)` will not be executed, the fact that it is a failure doesn't matter, because it will not run. The divide by zero exception causes the test method to exit. This can be a source of confusion while debugging.

Assert the Value of a Public, Protected and Private property

PHPUnit has two assertions to check values of class properties:

```
assertAttributeSame($expected, $actualAttributeName, $actualClassOrObject, $message = '')
assertAttributeNotSame($expected, $actualAttributeName, $actualClassOrObject, $message = '')
```

These methods will check the value of a object property regardless of the visibility.

Let's start with a class to be tested. It is a simplified class that has three properties, each with a different visibility:

```
class Color {

    public $publicColor       = "red";
    protected $protectedColor = "green";
    private $privateColor     = "blue";

}
```

Now, to test the value of each property:

```
class ColorTest extends PHPUnit_Framework_TestCase
{
    public function test_assertAttributeSame() {

        $hasColor = new Color();

        $this->assertAttributeSame("red","publicColor",$hasColor);
        $this->assertAttributeSame("green","protectedColor",$hasColor);
        $this->assertAttributeSame("blue","privateColor",$hasColor);

        $this->assertAttributeNotSame("wrong","privateColor",$hasColor);
    }

}
```

As you can see, the assertion works for any visibility, making it easy to peer into protected and private methods.

In addition, there is assertAttributeEquals, assertAttributeContains, assertAttributeContainsOnly, assertAttributeEmpty...etc, matching most assertions involving comparison.

Read Assertions online: https://riptutorial.com/phpunit/topic/6525/assertions

Chapter 3: Getting Started with PHPUnit

Examples

Installation on Linux or MacOSX

Global installation using the PHP Archive

```
wget https://phar.phpunit.de/phpunit.phar         # download the archive file
chmod +x phpunit.phar                             # make it executable
sudo mv phpunit.phar /usr/local/bin/phpunit       # move it to /usr/local/bin
phpunit --version                                 # show installed version number
```

Global installation using Composer

```
# If you have composer installed system wide
composer global require phpunit/phpunit  # set PHPUnit as a global dependency
phpunit --version                        # show installed version number

# If you have the .phar file of composer
php composer.phar global require phpunit/phpunit  # set PHPUnit as a global dependency
phpunit --version                                 # show installed version number
```

Local installation using Composer

```
# If you have composer installed system wide
composer require phpunit/phpunit  # set PHPUnit as a local dependency
./vendor/bin/phpunit --version    # show installed version number

# If you have the .phar file of composer
php composer.phar require phpunit/phpunit  # set PHPUnit as a local dependency
./vendor/bin/phpunit --version             # show installed version number
```

Read Getting Started with PHPUnit online: https://riptutorial.com/phpunit/topic/3982/getting-started-with-phpunit

Chapter 4: Test Doubles (Mocks and Stubs)

Examples

Simple mocking

Introduction

The PHPUnit Manual describes mocking as such:

> 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.

So instead of stubbing out code, an observer is created that not only replaces the code that needs to be silenced, but observes that a specific activity would have happened in the real world.

Setup

Let's start with a simple logger class that for the sake of clarity, simply displays the text sent into the parameter (normally it would do something that can be problematic to unit testing, such as updating a database):

```
class Logger {
    public function log($text) {
        echo $text;
    }
}
```

Now, let's create an Application class. It accepts a Logger object as a parameter to the **run** method, which in turn invokes the Logger's **log** method to capture that the application has started.

```
class Application {
  public function run(Logger $logger) {
    // some code that starts up the application

    // send out a log that the application has started
    $logger->log('Application has started');
  }
}
```

If the following code was executed as written:

```
$logger = new Logger();
$app = new Application();
$app->run($logger);
```

Then the text *"Application has started"* would be displayed as per the **log** method inside of the Logger.

Unit Testing with Mocking

The Application class unit testing does not need to verify what happens within the Logger **log** method, it only needs to verify that it was called.

In the PHPUnit test, an observer is created to replace the Logger class. That observer is set up to ensure that the **log** method is invoked only once, with the parameter value *"Application has started"*.

Then, the observer is sent into the run method, which verifies that in fact the log method was called just once and the test case passes, but no text was displayed.

```
class ApplicationTest extends \PHPUnit_Framework_TestCase {

  public function testThatRunLogsApplicationStart() {

    // create the observer
    $mock = $this->createMock(Logger::class);
    $mock->expects($this->once())
        ->method('log')
        ->with('Application has started');

    // run the application with the observer which ensures the log method was called
    $app = new Application();
    $app->run($mock);

  }
}
```

Simple stubbing

Sometimes there are sections of code that are difficult to test, such as accessing a database, or interacting with the user. You can stub out those sections of code, allowing the rest of the code to be tested.

Let's start with a class that prompts the user. For simplicity, it has only two methods, one that actually prompts the user (which would be used by all the other methods) and the one we are going to test, which prompts and filters out only yes and no answers. Please note this code is overly simplistic for demonstration purposes.

```
class Answer
{
    // prompt the user and check if the answer is yes or no, anything else, return null
    public function getYesNoAnswer($prompt) {

        $answer = $this->readUserInput($prompt);

        $answer = strtolower($answer);
        if (($answer === "yes") || ($answer === "no")) {
            return $answer;
        } else {
            return null;
        }

    }

    // Simply prompt the user and return the answer
    public function readUserInput($prompt) {
        return readline($prompt);
    }

}
```

To test `getYesNoAnswer`, the `readUserInput` needs to be stubbed out to mimic answers from a user.

```
class AnswerTest extends PHPUnit_Framework_TestCase
{

    public function test_yes_no_answer() {

        $stub = $this->getMockBuilder(Answer::class)
                ->setMethods(["readUserInput"])
                ->getMock();

        $stub->method('readUserInput')
            ->will($this->onConsecutiveCalls("yes","junk"));

        // stub will return "yes"
        $answer = $stub->getYesNoAnswer("Student? (yes/no)");
        $this->assertSame("yes",$answer);

        // stub will return "junk"
        $answer = $stub->getYesNoAnswer("Student? (yes/no)");
        $this->assertNull($answer);


    }

}
```

The first line of code creates the stub and it uses `getMockBuilder` instead of `createMock`. `createMock` is a shortcut for calling `getMockBuilder` with defaults. One of these defaults is to stub out all the methods. For this example, we want to test `getYesNoAnswer`, so it can't be stubbed out. The `getMockBuilder` invokes `setMethods` to request that only `readUserInput` be stubbed out.

The second line of code creates the stubbing behavior. It stubs out `readUserInput` method and sets up two return values upon subsequent calls, "yes" followed by "junk".

The third and fourth lines of code test `getYesNoAnswer`. The first time, the fake person responds with "yes" and the tested code correctly returns "yes", since it is a valid selection. The second time, the fake person responds with "junk" and the tested code correctly returns null.

Read Test Doubles (Mocks and Stubs) online: https://riptutorial.com/phpunit/topic/4979/test-doubles--mocks-and-stubs-

Credits

S. No	Chapters	Contributors
1	Getting started with phpunit	alexander.polomodov, arushi, Community, eskwayrd, Gerard Roche, Joachim Schirrmacher, Katie, Martin GOYOT, Xavi Montero, Ziumin
2	Assertions	Katie
3	Getting Started with PHPUnit	alexander.polomodov, Joachim Schirrmacher, Katie, Martin GOYOT, Ziumin
4	Test Doubles (Mocks and Stubs)	Joachim Schirrmacher, Katie, olvlvl, Xavi Montero