EBook Gratis

APRENDIZAJE phpunit

Free unaffiliated eBook created from
Stack Overflow contributors.

#phpunit

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: phpunit

It is an unofficial and free phpunit ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official phpunit.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con phpunit

Observaciones

Esta sección proporciona una descripción general de qué es phpunit y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema importante dentro de phpunit y vincular a los temas relacionados. Dado que la Documentación para phpunit es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Finaliza el soporte	Soportado en esas versiones de PHP	Fecha de lanzamiento
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 y PHP 5.6.	2015-08-07

Examples

Crea la primera prueba PHPUnit para nuestra clase

Imagina que tenemos una clase `Math.php` con lógica de cálculo de fiobanacci y números factoriales. Algo como esto:

```
<?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');
        }
    }
}
```

La prueba mas sencilla

Queremos probar la lógica de los métodos `fibonacci` y `factorial` . Vamos a crear el archivo `MathTest.php` en el mismo directorio con `Math.php` . En nuestro código podemos utilizar **diferentes aserciones** . El código más simple será algo como esto (solo usamos `assertEquals` y `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));
    }
}
```

Podemos ejecutar esta prueba desde la consola con el comando `phpunit MathTest` y la salida será:

```
    PHPUnit 5.3.2 by Sebastian Bergmann and contributors.

...                                                                 3 / 3 (100%)

Time: 88 ms, Memory: 10.50Mb

OK (3 tests, 3 assertions)
```

Usando los proveedores de datos

Un método de prueba puede aceptar argumentos arbitrarios. Estos argumentos deben ser proporcionados por un método de **proveedor de datos** . El método del proveedor de datos que se utilizará se especifica mediante la anotación `@dataProvider` . :

```
<?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),
        );
    }
}
```

Podemos ejecutar esta prueba desde la consola con el comando `phpunit MathTest` y la salida será:

```
    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;
```

Excepciones de prueba

Podemos probar si el código bajo prueba lanza una excepción usando el método `expectException()` . También en este ejemplo, agregamos una prueba fallida para mostrar el resultado de la consola para las pruebas fallidas.

```
<?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);
    }
}
```

Podemos ejecutar esta prueba desde la consola con el comando `phpunit MathTest` y la salida será:

```
        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 y TearDown

También `PHPUnit` admite compartir el código de configuración. Antes de ejecutar un método de prueba, se invoca un método de plantilla denominado setUp (). `setUp()` es donde creas los objetos contra los cuales `setUp()` . Una vez que el método de prueba ha terminado de ejecutarse, ya sea que haya tenido éxito o haya fallado, se invoca otro método de plantilla llamado `tearDown()` . `tearDown()` es donde limpia los objetos contra los que ha probado.

```
<?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 == []);
    }
}
```

Más información

Hay muchas más grandes oportunidades de `PHPUnit` que puede utilizar en su prueba. Para más información ver en **manual oficial.**

Ejemplo de prueba mínima

Dada una simple clase de PHP:

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

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

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

Puede escribir una prueba PHPUnit que pruebe el comportamiento de la clase bajo prueba llamando a los métodos públicos y verifique si funcionan como se espera:

```
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());
    }
}
```

Partes importantes:

1. La clase de prueba debe derivarse de PHPUnit_Framework_TestCase.
2. Cada nombre de función de prueba debe comenzar con el prefijo 'prueba'
3. Use las funciones `$this->assert...` para verificar los valores esperados.

Clases de burla

La práctica de reemplazar un objeto con un doble de prueba que verifique las expectativas, por ejemplo, afirmar que se ha llamado a un método, se conoce como burlón.

Asumamos que tenemos SomeService para probar.

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

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

Y queremos probar si `methodToTest` realmente llama al método de `save` del repositorio. Pero no queremos en realidad crear una instancia del repositorio (o quizás `Repository` es solo una interfaz).

En este caso podemos simular `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();
    }
}
```

Ejemplo de PHPUNIT con APItest usando Stub And Mock

Clase para la que creará un caso de prueba unitaria. `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);
}
```

}

Ahora no queríamos ninguna interacción externa de nuestro código de prueba, por lo que necesitamos crear un objeto simulado para la función callAPI, ya que esta función en realidad está llamando a la URL externa a través de 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'));
}
```

}

A continuación se muestra el código para la clase de curl (solo una muestra) `class Curl{ function callAPI($url, $method){`

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

}

Lea Empezando con phpunit en línea: https://riptutorial.com/es/phpunit/topic/3268/empezando-con-phpunit

Capítulo 2: Afirmaciones

Examples

Afirmar un objeto es una instancia de una clase

PHPUnit proporciona la siguiente función para afirmar si un objeto es una instancia de una clase:

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

El primer parámetro `$expected` es el nombre de una clase (cadena). El segundo parámetro `$actual` es el objeto a probar. `$message` es una cadena opcional que puede proporcionar en caso de que falle.

Empecemos con una clase simple de Foo:

```
class Foo {

}
```

En otro lugar en un espacio de nombres `Crazy` , hay una clase de `Bar` .

```
namespace Crazy

class Bar {

}
```

Ahora, escribamos un caso de prueba básico que verifique un objeto para estas clases

```
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");
```

```
    }
}
```

Tenga en cuenta que PHPUnit se pone gruñón si envía un nombre de clase que no existe.

Afirmar que se lanza una excepción

PHPUnit proporciona las siguientes funciones para observar las excepciones lanzadas, que se lanzaron con 5.2.0:

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

Estos se utilizan para ver si se produce una excepción e inspeccionar las propiedades de esa excepción.

Comencemos con una función matemática que divide (solo por simplicidad). Se producirá una excepción si el denominador es cero.

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

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

}
```

Ahora para el código de prueba.

```
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);

    }
```

```
}
```

La función `test_divide()` comienza afirmando que la función dividió correctamente 4 por 2 y contestó 2. Esta aserción pasará.

A continuación, se establecen las expectativas para la próxima excepción. Observe que están establecidos antes del código que lanzará la excepción. Las cuatro afirmaciones se muestran con fines de demostración, pero esto normalmente no es necesario.

La `divide(4,0)` lanzará la excepción esperada y toda la función expect * pasará.

Pero tenga en cuenta que el código `$this->assertSame(0,1)` no se ejecutará, el hecho de que sea un error no importa, porque no se ejecutará. La excepción de división por cero hace que el método de prueba salga. Esto puede ser una fuente de confusión durante la depuración.

Afirmar el valor de una propiedad pública, protegida y privada

PHPUnit tiene dos aserciones para verificar los valores de las propiedades de la clase:

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

Estos métodos verifican el valor de una propiedad de objeto independientemente de la visibilidad.

Vamos a empezar con una clase para ser probado. Es una clase simplificada que tiene tres propiedades, cada una con una visibilidad diferente:

```
class Color {

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

}
```

Ahora, para probar el valor de cada propiedad:

```
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);
    }

}
```

Como puede ver, la afirmación funciona para cualquier visibilidad, haciendo que sea fácil mirar en métodos protegidos y privados.

Además, hay assertAttributeEquals, assertAttributeContains, assertAttributeContainsOnly, assertAttributeEmpty ... etc, que coinciden con la mayoría de las aserciones que implican comparación.

Lea Afirmaciones en línea: https://riptutorial.com/es/phpunit/topic/6525/afirmaciones

Capítulo 3: Dobles de prueba (Mocks y Stubs)

Examples

Burla simple

Introducción

El Manual de la Unidad de PHP describe la burla como tal:

> La práctica de reemplazar un objeto con un doble de prueba que verifique las expectativas, por ejemplo, afirmar que se ha llamado a un método, se conoce como burlón.

Entonces, en lugar de apagar el código, se crea un observador que no solo reemplaza el código que necesita ser silenciado, sino que observa que una actividad específica habría ocurrido en el mundo real.

Preparar

Comencemos con una clase de registrador simple que, en aras de la claridad, simplemente muestra el texto enviado al parámetro (normalmente haría algo que puede ser problemático para las pruebas unitarias, como actualizar una base de datos):

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

Ahora, vamos a crear una clase de aplicación. Acepta un objeto Logger como parámetro para el método de **ejecución** , que a su vez invoca el método de **registro** del Logger para capturar que la aplicación se ha iniciado.

```
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');
  }
}
```

Si el siguiente código fue ejecutado como está escrito:

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

Luego, el texto *"La aplicación ha comenzado"* se mostrará según el método de **registro** dentro del **registrador** .

Pruebas unitarias con burla

La prueba de unidad de clase de aplicación no necesita verificar lo que sucede dentro del método de **registro del registrador** , solo necesita verificar que se llamó.

En la prueba PHPUnit, se crea un observador para reemplazar la clase Logger. Ese observador está configurado para garantizar que el método de **registro** se invoque solo una vez, con el valor del parámetro *"La aplicación ha comenzado"* .

Luego, el observador se envía al método de ejecución, que verifica que, de hecho, el método de registro se llamó solo una vez y el caso de prueba pasa, pero no se mostró ningún texto.

```
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

A veces hay secciones de código que son difíciles de probar, como acceder a una base de datos o interactuar con el usuario. Puede borrar esas secciones de código, permitiendo que el resto del código sea probado.

Vamos a empezar con una clase que solicita al usuario. Para simplificar, solo tiene dos métodos, uno que realmente solicita al usuario (que sería usado por todos los otros métodos) y el que vamos a probar, que solicita y filtra solo las respuestas de sí y no. Tenga en cuenta que este código es demasiado simplista para fines de demostración.

```
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);
    }

}
```

Para probar `getYesNoAnswer` , el `readUserInput` debe ser apagado para imitar las respuestas de un usuario.

```
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);


    }

}
```

La primera línea de código crea el código auxiliar y usa `getMockBuilder` lugar de `createMock` . `createMock` es un acceso directo para llamar a `getMockBuilder` con valores predeterminados. Uno de estos valores predeterminados es eliminar todos los métodos. Para este ejemplo, queremos probar `getYesNoAnswer` , por lo que no se puede apagar. `getMockBuilder` invoca a `setMethods` para solicitar que solo se `readUserInput` .

La segunda línea de código crea el comportamiento de apéndice. `readUserInput` método `readUserInput` y establece dos valores de retorno en las llamadas subsiguientes, "sí" seguido de "basura".

La tercera y cuarta líneas de prueba de código `getYesNoAnswer` . La primera vez, la persona falsa responde con "sí" y el código probado devuelve correctamente "sí", ya que es una selección válida. La segunda vez, la persona falsa responde con "basura" y el código probado devuelve correctamente el valor nulo.

Lea Dobles de prueba (Mocks y Stubs) en línea:
https://riptutorial.com/es/phpunit/topic/4979/dobles-de-prueba--mocks-y-stubs-

Capítulo 4: Empezando con PHPUnit

Examples

Instalación en Linux o MacOSX

Instalación global utilizando el archivo PHP

```
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
```

Instalación global utilizando 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
```

Instalación local utilizando 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
```

Lea Empezando con PHPUnit en línea: https://riptutorial.com/es/phpunit/topic/3982/empezando-con-phpunit

Creditos

S. No	Capítulos	Contributors
1	Empezando con phpunit	alexander.polomodov, arushi, Community, eskwayrd, Gerard Roche, Joachim Schirrmacher, Katie, Martin GOYOT, Xavi Montero, Ziumin
2	Afirmaciones	Katie
3	Dobles de prueba (Mocks y Stubs)	Joachim Schirrmacher, Katie, olvlvl, Xavi Montero
4	Empezando con PHPUnit	alexander.polomodov, Joachim Schirrmacher, Katie, Martin GOYOT, Ziumin