eBook Gratuit

APPRENEZ phpunit

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#phpunit

Table des matières

À propos

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

Chapitre 1: Démarrer avec phpunit

Remarques

Cette section fournit une vue d'ensemble de ce qu'est phpunit et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tout sujet important dans phpunit, et établir un lien avec les sujets connexes. La documentation de phpunit étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Version	Fin du support	Pris en charge dans ces versions de PHP	Date de sortie
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 et PHP 5.6.	2015-08-07

Examples

Créer le premier test PHPUnit pour notre classe

Imaginons que nous ayons une classe `Math.php` avec une logique de calcul du fiobanacci et des nombres factoriels. Quelque chose comme ça:

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

Le test le plus simple

Nous voulons tester la logique des méthodes `fibonacci` et `factorial` . Créons le fichier `MathTest.php` dans le même répertoire avec `Math.php` . Dans notre code, nous pouvons utiliser **différentes assertions** . Le code le plus simple sera quelque chose comme ça (nous utilisons uniquement `assertEquals` et `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));
    }
}
```

Nous pouvons exécuter ce test depuis la console avec la commande `phpunit MathTest` et la sortie sera:

```
    PHPUnit 5.3.2 by Sebastian Bergmann and contributors.

...                                                                 3 / 3 (100%)

Time: 88 ms, Memory: 10.50Mb

OK (3 tests, 3 assertions)
```

Utilisation de dataProviders

Une méthode de test peut accepter des arguments arbitraires. Ces arguments doivent être fournis par une méthode de **fournisseur de données** . La méthode du fournisseur de données à utiliser est spécifiée à l'aide de l'annotation `@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),
        );
    }
}
```

Nous pouvons exécuter ce test depuis la console avec la commande `phpunit MathTest` et la sortie sera:

```
    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 des exceptions

Nous pouvons tester si une exception est levée par le code sous test en utilisant la méthode `expectException()` . Dans cet exemple, nous avons également ajouté un test échoué pour afficher la sortie de la console pour les tests ayant échoué.

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

Nous pouvons exécuter ce test depuis la console avec la commande `phpunit MathTest` et la sortie sera:

```
        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 et TearDown

Aussi `PHPUnit` prend en charge le partage du code d'installation. Avant qu'une méthode de test ne soit exécutée, une méthode de modèle appelée setUp () est appelée. `setUp()` est l'endroit où vous créez les objets contre lesquels vous allez tester. Une fois la méthode de test terminée, qu'elle ait réussi ou échoué, une autre méthode de modèle appelée `tearDown()` est appelée. `tearDown()` est l'endroit où vous nettoyez les objets contre lesquels vous avez testé.

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

Plus d'informations

Il y a beaucoup plus d'opportunités de `PHPUnit` que vous pouvez utiliser dans votre test. Pour plus d'infos voir en **manuel**

Exemple de test minimal

Étant donné une classe PHP simple:

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

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

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

Vous pouvez écrire un test PHPUnit pour tester le comportement de la classe testée en appelant les méthodes publiques et vérifier si elles fonctionnent comme prévu:

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

Pièces importantes:

1. La classe de test doit dériver de PHPUnit_Framework_TestCase.
2. Chaque nom de fonction de test doit commencer par le préfixe "test"
3. Utilisez `$this->assert...` pour vérifier les valeurs attendues.

Cours de moqueur

La pratique consistant à remplacer un objet par un double de test qui vérifie les attentes, par exemple en affirmant qu'une méthode a été appelée, est appelée moqueur.

Supposons que nous avons SomeService à tester.

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

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

Et nous voulons tester si `methodToTest` appelle vraiment la méthode `save` du dépôt. Mais nous ne voulons pas réellement instancier un référentiel (ou peut-être que `Repository` est juste une interface).

Dans ce cas, nous pouvons simuler un `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();
    }
}
```

Exemple de PHPUNIT avec APItest utilisant Stub And Mock

Classe pour laquelle vous allez créer un test élémentaire. `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);
}
```

}

Maintenant, nous ne voulions aucune interaction externe de notre code de test, nous devons donc créer un objet simulé pour la fonction callAPI car cette fonction appelle en réalité une URL externe 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'));
}
```

}

Voici le code de la classe curl (juste un exemple) de la `class Curl{ function callAPI($url, $method){`

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

}

Lire Démarrer avec phpunit en ligne: https://riptutorial.com/fr/phpunit/topic/3268/demarrer-avec-phpunit

Chapitre 2: Assertions

Examples

Assert un objet est une instance d'une classe

PHPUnit fournit la fonction suivante pour déterminer si un objet est une instance d'une classe:

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

Le premier paramètre `$expected` est le nom d'une classe (chaîne). Le second paramètre `$actual` est l'objet à tester. `$message` est une chaîne optionnelle que vous pouvez fournir en cas d'échec.

Commençons par une classe Foo simple:

```
class Foo {

}
```

Quelque part ailleurs dans un espace de noms `Crazy` , il y a un cours de `Bar` .

```
namespace Crazy

class Bar {

}
```

Maintenant, écrivons un test élémentaire qui vérifierait un objet pour ces 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");

    }
```

```
}
```

Notez que PHPUnit devient grincheux si vous envoyez un nom de classe qui n'existe pas.

Affirmer qu'une exception est déclenchée

PHPUnit fournit les fonctions suivantes pour surveiller les exceptions levées, qui ont été publiées avec la version 5.2.0:

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

Ceux-ci sont utilisés pour surveiller une exception à émettre et inspecter les propriétés de cette exception.

Commençons par une fonction mathématique qui divise (pour simplifier). Il déclenchera une exception si le dénominateur est zéro.

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

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

}
```

Maintenant, pour le code de test.

```
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 fonction `test_divide()` commence par affirmer que la fonction a correctement divisé 4 par 2 et a répondu 2. Cette assertion passera.

Ensuite, les attentes pour l'exception à venir sont définies. Notez qu'ils sont placés avant le code qui lancera l'exception. Les quatre assertions sont affichées à des fins de démonstration, mais cela n'est normalement pas nécessaire.

Le `divide(4,0)` lancera alors l'exception attendue et toute la fonction expect * passera.

Mais notez que le code `$this->assertSame(0,1)` ne sera pas exécuté, le fait que ce soit un échec importe peu, car il ne fonctionnera pas. L'exception de division par zéro provoque la sortie de la méthode de test. Cela peut être source de confusion lors du débogage.

Affirmer la valeur d'une propriété publique, protégée et privée

PHPUnit a deux assertions pour vérifier les valeurs des propriétés de la classe:

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

Ces méthodes vérifient la valeur d'une propriété d'objet indépendamment de la visibilité.

Commençons par une classe à tester. C'est une classe simplifiée qui possède trois propriétés, chacune avec une visibilité différente:

```
class Color {

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

}
```

Maintenant, pour tester la valeur de chaque propriété:

```
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);
    }

}
```

Comme vous pouvez le constater, l'assertion fonctionne pour n'importe quelle visibilité, facilitant ainsi l'échantillonnage de méthodes protégées et privées.

En outre, il y a assertAttributeEquals, assertAttributeContains, assertAttributeContainsOnly, assertAttributeEmpty ... etc, correspondant à la plupart des assertions impliquant la comparaison.

Lire Assertions en ligne: https://riptutorial.com/fr/phpunit/topic/6525/assertions

Chapitre 3: Premiers pas avec PHPUnit

Examples

Installation sous Linux ou MacOSX

Installation globale à l'aide de l'archive 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
```

Installation globale à l'aide de 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
```

Installation locale avec 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
```

Lire Premiers pas avec PHPUnit en ligne: https://riptutorial.com/fr/phpunit/topic/3982/premiers-pas-avec-phpunit

Chapitre 4: Test Doubles (Mocks et Stubs)

Examples

Simple se moquer

introduction

Le manuel PHPUnit décrit les moqueries comme telles:

> La pratique consistant à remplacer un objet par un double de test qui vérifie les attentes, par exemple en affirmant qu'une méthode a été appelée, est appelée moqueur.

Ainsi, au lieu de supprimer le code, un observateur est créé qui non seulement remplace le code qui doit être réduit au silence, mais observe qu'une activité spécifique se serait produite dans le monde réel.

Installer

Commençons par une classe de journalisation simple qui, pour plus de clarté, affiche simplement le texte envoyé dans le paramètre (normalement, cela ferait quelque chose qui peut être problématique pour les tests unitaires, tels que la mise à jour d'une base de données):

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

Maintenant, créons une classe Application. Il accepte un objet Logger en tant que paramètre de la méthode **run** , qui à son tour appelle la méthode de **journalisation** de Logger pour capturer le démarrage de l'application.

```
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 le code suivant a été exécuté comme écrit:

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

Ensuite, le texte *"Application a démarré"* serait affiché selon la méthode de **journalisation à l'** intérieur de l'enregistreur.

Test unitaire avec moquage

Le test d'unité de classe d'application n'a pas besoin de vérifier ce qui se passe dans la méthode de **journal** Logger, il suffit de vérifier qu'il a été appelé.

Dans le test PHPUnit, un observateur est créé pour remplacer la classe Logger. Cet observateur est configuré pour garantir que la méthode de **journalisation** est appelée une seule fois, avec la valeur du paramètre *"Application a démarré"* .

Ensuite, l'observateur est envoyé dans la méthode run, qui vérifie que la méthode de journalisation a été appelée une seule fois et que le test élémentaire réussit, mais aucun texte n'a été affiché.

```
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

Il existe parfois des sections de code difficiles à tester, telles que l'accès à une base de données ou l'interaction avec l'utilisateur. Vous pouvez extraire ces sections de code, ce qui permet de tester le reste du code.

Commençons par une classe qui invite l'utilisateur. Pour plus de simplicité, il ne dispose que de deux méthodes, une qui invite l'utilisateur (qui serait utilisé par toutes les autres méthodes) et celle que nous allons tester, qui invite et filtre uniquement les réponses oui et non. Veuillez noter que ce code est trop simpliste à des fins de démonstration.

```
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);
    }

}
```

Pour tester `getYesNoAnswer` , le `readUserInput` doit être supprimé pour imiter les réponses d'un utilisateur.

```
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 première ligne de code crée le stub et utilise `getMockBuilder` au lieu de `createMock` . `createMock` est un raccourci pour appeler `getMockBuilder` avec les valeurs par défaut. L'une de ces valeurs par défaut consiste à supprimer toutes les méthodes. Pour cet exemple, nous voulons tester `getYesNoAnswer` afin qu'il ne puisse pas être supprimé. `getMockBuilder` appelle `setMethods` pour demander que seul `readUserInput` soit supprimé.

La deuxième ligne de code crée le comportement de stubbing. Il `readUserInput` méthode `readUserInput` et configure deux valeurs de retour lors des appels suivants, "yes" suivi de "junk".

Les troisième et quatrième lignes du test de code `getYesNoAnswer` . La première fois, le faux utilisateur répond par "oui" et le code testé renvoie correctement "oui", car il s'agit d'une sélection valide. La deuxième fois, le faux utilisateur répond par "indésirable" et le code testé renvoie correctement null.

Lire Test Doubles (Mocks et Stubs) en ligne: https://riptutorial.com/fr/phpunit/topic/4979/test-doubles--mocks-et-stubs-

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec phpunit	alexander.polomodov, arushi, Community, eskwayrd, Gerard Roche, Joachim Schirrmacher, Katie, Martin GOYOT, Xavi Montero, Ziumin
2	Assertions	Katie
3	Premiers pas avec PHPUnit	alexander.polomodov, Joachim Schirrmacher, Katie, Martin GOYOT, Ziumin
4	Test Doubles (Mocks et Stubs)	Joachim Schirrmacher, Katie, olvlvl, Xavi Montero