**Test Unitaire**

**1.Qu'est-ce qu'un test unitaire ?**

Un test unitaire est un procédé permettant de s'assurer du bon fonctionnement d'une unité de programme.Un test unitaire est dit "test boîte blanche".

**2.Installation de PHPunit :**

**composer require phpunit/phpunit**

Ajouter la version si vous voulez

**composer require phpunit/phpunit 5.7**

**3.Pour vérifier que PHPUnit est bien installé :**

Executer la commande :

* Sous windows :

**C:/wamp64/www/nomprojet/vendor/bin/phpunit.bat --help**

* Sous linux/mac :

**vendor/bin/phpunit --help**

**4.Pour lancer les tests unitaires :**

**vendor/bin/phpunit**

**5.Générer le rapport de couverture de code :**

pour utiliser le code-coverage on doit installer xdebug :

## **Install and Configure xDebug on Ubuntu for PhpStorm**

* Assuming that you have already installed php and apache
* **Install**[**xDebug**](https://xdebug.org/docs/install)**php extension**

# Ubuntu 16.04,18.04 php 7.x

sudo apt-get install php-xdebug

# Ubuntu 14.04, php 5.6

sudo apt-get install php5-xdebug

* **Edit your xdebug.ini**
* Your xdebug.ini file path should look like this
  + /etc/php/7.1/mods-available/xdebug.ini
* Add these lines without modifying exiting

xdebug.remote\_enable = 1

xdebug.remote\_port = 9000

xdebug.idekey = PHPSTORM

xdebug.show\_error\_trace = 1

xdebug.remote\_autostart = 0

xdebug.file\_link\_format = phpstorm://open?%f:%l

pour generer le coverge on html :c’est pour assurer la pourcentage de couverture de test executé la commande suivante :

vendor/bin/phpunit --coverage-html web/code-coverage.

web/code-coverage : c’est le nom de dossier pour contenir le code coverge en html

**6.Creation du test unitaire :**

Exemple : on va créer une classe pour tester :

**<?php**

*/\*\**

*\* Created by PhpStorm.*

*\* User: hamdi*

*\* Date: 04/10/18*

*\* Time: 15:03*

*\*/*

**namespace** AppBundle\ClasseDeTest;

**class** Exemple

{

*/\*\**

*\* retourner la somme de deux nombres*

*\** ***@param*** *$a*

*\** ***@param*** *$b*

*\** ***@return*** *float|int*

*\*/*

**public function** sum($a,$b)

{

**return** $a+$b;

}

*/\*\**

*\** ***@param*** *$a*

*\** ***@param*** *$b*

*\** ***@return*** *float|int*

*\*/*

**public function** product($a,$b)

{

**return** $a\*$b;

}

*/\*\**

*\** ***@param*** *$a*

*\** ***@param*** *$b*

*\** ***@return*** *float|int*

*\*/*

**public function** divide($a,$b)

{

**if** ($b === 0) {

**throw new** \LogicException('The b cannot be negative.');

}

**else**{

**return** $a/$b;

}

}

}

**7.Combien de test(s) va-t-il falloir implémenter ?**

Une indication pour connaître le nombre de tests qu'il va falloir écrire concerne **les sorties possibles de la méthode(le nombre de return) et le nombre d’exceptions**.

* + - 1. **8.Créer la classe de test**

Tous les tests doivent être écrits dans des classes. Vous allez donc créer votre première classe de tests. Comme vous écrivez ces tests dans un projet Symfony, il y a quelques règles à respecter :

* Cette classe doit être contenue dans le dossier Tests du projet.
* Il faut reproduire l'arborescence de la classe que vous souhaitez tester.
* La classe de test doit avoir le même nom que la classe à tester, suffixée par Test

**<?php**

*/\*\**

*\* Created by PhpStorm.*

*\* User: hamdi*

*\* Date: 04/10/18*

*\* Time: 15:23*

*\*/*

**namespace** Tests\AppBundle\ClasseDeTest;

**use** AppBundle\ClasseDeTest\Exemple;

**use** PHPUnit\Framework\TestCase;

**class** ExempleTest **extends** TestCase

{

*/\*\**

*\** ***@dataProvider*** *sumProvider*

*\*/*

**public function** testSum($a,$b,$expectedSum)

{

$exemple = **new** Exemple();

$this->assertSame($expectedSum,$exemple->sum($a,$b));

}

*/\*\**

*\** ***@dataProvider*** *productProvider*

*\*/*

**public function** testProduct($a,$b,$expectedProduct){

$exemple = **new** Exemple();

$this->assertSame($expectedProduct,$exemple->product($a,$b));

}

*/\*\**

*\** ***@dataProvider*** *divideProvider*

*\*/*

**public function** testDivide($a,$b,$expectedDivition)

{

$exemple = **new** Exemple();

$this->assertSame($expectedDivition,$exemple->divide($a,$b));

}

**public function** testDivideZero()

{

$exemple = **new** Exemple();

$this->expectException('LogicException');

$exemple->divide(10,0);

}

**public function** sumProvider()

{

**return** [

[10,5,15],

[1,5,6],

[0,5,5],

];

}

**public function** productProvider()

{

**return** [

[1,0,0],

[1,9,9]

];

}

**public function** divideProvider()

{

**return** [

[3,3,1],

[10,2,5]

];

}

}

#### **Créer les méthodes de test**

Une règle à respecter : le nom de **toutes** vos méthodes de test doit être préfixé par test

**Qu'est-ce qu'une assertion ?**

Concrètement, du fait que nous étendons la classe PHPUnit\Framework\TestCase, nous avons la possibilité d'appeler la méthode assertSame qui prend en paramètre deux valeurs : la valeur attendue et le résultat du code exécuté.

Il ne vous reste plus qu'à lancer la suite de tests avec la commande suivante :

$ vendor/bin/phpunit

la méthode expectException prend en paramètre une chaîne de caractères correspondant au FQCN (Full Qualified Class Name, c'est-à-dire le namespace complet de la classe) de l'exception qui devrait être levée au moment de l'exécution du code. Cette méthode nous permet d'ajouter une nouvelle assertion à notre suite de tests !

**7.Créer une suite de tests avec une suite de valeurs définies : les data providers :**

***/\*\****

*\** ***@dataProvider*** *pricesForFoodProduct*

*\*/*

Grâce à l'annotation @dataprovider, PHPUnit est en mesure de récupérer les données via la méthode indiquée dans l'annotation (productProvider).

L'option --filter permet de ne lancer qu'une méthode de test.

**vendor/bin/phpunit –filter=testComputeTVAOtherProduct**

**8.Qu'est-ce qu'une doublure ?**

Une doublure est un élément que nous aurons créé de toutes pièces pour maîtriser une dépendance externe.

**Un dummy** est un objet un peu particulier qui remplit un contrat.

<?php

namespace Tests\AppBundle;

use PHPUnit\Framework\TestCase;

class ClassTest extends TestCase

{

public function testExemple()

{

$client = $this→getMock('GuzzleHttp\Client'); // ici un dummy

// …

}

}

**Un stub** est un dummy auquel on ajoute un comportement. Cela signifie concrètement que vous indiquez ce que la méthode d'un objet doit toujours retourner lorsqu'elle est appelée.

Exemple :

<?php

namespace Tests\AppBundle;

use PHPUnit\Framework\TestCase;

use Symfony\Component\HttpFoundation\Request;

class ClassTest extends TestCase

{

public function testExemple()

{

$request = new Request();

$client = $this->getMock('GuzzleHttp\Client');

$client->method('get')->willReturn($request); //stub

// …

}

}

**Un mock** est un stub qui a des attentes ("expectations" en anglais). Avec un mock, quand vous allez appeler une méthode, vous pouvez préciser le comportement attendu lors de cet appel (par exemple, vous allez appeler la méthode une seule fois).

<?php

namespace Tests\AppBundle;

use PHPUnit\Framework\TestCase;

use Symfony\Component\HttpFoundation\Request;

class ClassTest extends TestCase

{

public function testExemple()

{

$request = new Request();

$client = $this->getMock('GuzzleHttp\Client');

$client

→expects($this->once()) //mock

->method('get')

->willReturn($request);

// …

}

}

**9.Quelques cas requérant la création d'une doublure**

Un objet est difficile à instancier

**<?php**

**namespace** JMS\Serialiser;

// …

**class** Serializer

{

//…

**public function** \_\_construct(

MetadataFactoryInterface $factory,

HandlerRegistryInterface $handlerRegistry,

ObjectConstructorInterface $objectConstructor,

MapInterface $serializationVisitors,

MapInterface $deserializationVisitors,

EventDispatcherInterface $dispatcher = **null**,

TypeParser $typeParser = **null**,

ExpressionEvaluatorInterface $expressionEvaluator = **null**

)

{

//…

}

C'est pourquoi, si vous avez besoin d'une instance du Serializer dans un test, ce serait plus facile de créer un dummy en demandant à PHPUnit de ne pas faire appel au constructeur original :

**<?php**

**namespace** Tests\AppBundle\Folder;

**use** PHPUnit\Framework\TestCase;

**class** ExempleClassTest **extends** TestCase

{

**public function** testExemple()

{

$serializer = $this

->getMockBuilder('JMS\Serializer\Serializer')

->disableOriginalConstructor()

->getMock();

$classToTest = **new** ExempleClass($serializer);

// …

}

}

**SetUp**

La méthode setUp est une méthode provenant de la classe PHPUnit\_Framework\_TestCase, que l'on peut surcharger pour exécuter des instructions avant chaque test de la classe.

**public function** setUp()

{

$this->client = **static**::*createClient*();

}

**tearDown**

Il est également possible d'intervenir à chaque fin de test (après chaque méthode de test de la classe) grâce à la méthode tearDown.

**public function** tearDown()

{

$this->client = **null**;

}

}