**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".

1. Installation de PHPunit :

composer require phpunit/phpunit

Ajouter la version si vous voulez

composer require phpunit/phpunit 5.7

1. Pour vérifier que PHPUnit est bien installé :

* Sous windows :

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

* Sous linux/mac :

vendor/bin/phpunit --help

1. 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**

**namespace** AppBundle\Entity;

**class** Product

{

**const** *FOOD\_PRODUCT* = 'food';

**private** $name;

**private** $type;

**private** $price;

**public function** \_\_construct($name, $type, $price)

{

$this->name = $name;

$this->type = $type;

$this->price = $price;

}

**public function** computeTVA()

{

**if** ($this->price < 0) {

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

}

**if** (**self**::*FOOD\_PRODUCT* == $this->type) {

**return** $this->price \* 0.055;

}

**return** $this->price \* 0.196;

}

}

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**. Dans le code présenté ci-dessus, vous voyez deux return :  il va falloir deux tests pour faire en sorte que tous les cas soient couverts.

#### **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

La classe à créer est donc la suivante, dans le dossier tests/AppBundle/Entity :

#### **Créer les méthodes de test**

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

**<?php**

*/\*\**

*\* Created by PhpStorm.*

*\* User: hamdi*

*\* Date: 16/07/18*

*\* Time: 22:03*

*\*/*

**namespace** Tests\AppBundle\Entity;

**use** AppBundle\Entity\Product;

**use** PHPUnit\Framework\TestCase;

**class** ProductTest **extends** TestCase

{

**public function** testcomputeTVAFoodProduct()

{

$product = **new** Product('Un produit', Product::*FOOD\_PRODUCT*, 20);

$this->assertSame(1.1, $product->computeTVA());

}

**public function** testComputeTVAOtherProduct()

{

$product = **new** Product('Un autre produit', 'Un autre type de produit', 20);

$this->assertSame(3.92, $product->computeTVA());

}

// ici la methode doit assurer le throw d’une exception

**public function** testNegativePriceComputeTVA()

{

$product = **new** Product('Un produit', Product::*FOOD\_PRODUCT*, -20);

$this->expectException('LogicException');

$product->computeTVA();

}

}

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. Le résultat devrait être **1.1**(puisque 20\*0.055 = 1.1).

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

$ vendor/bin/phpunit

Il s'agit d'indiquer ce qui est attendu avant d'exécuter le code à tester : 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 :**

Avec les tests unitaires automatisés, il est important de tenter de couvrir l'ensemble du code (les chemins), mais aussi les cas limites liés à la logique métier de l'application. Il est aussi intéressant de s'assurer que son code fonctionne avec une suite de valeurs en entrée différentes sans pour autant avoir à créer une méthode de test différente.

Pour répondre à cette problématique, il existe les data provider ("fournisseur de données" en français).

Modifions la méthode de test testcomputeTVAFoodProduct pour faire en sorte d'exécuter cette méthode de tests avec un jeu de données particulier

***/\*\****

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

*\*/*

**public function** testcomputeTVAFoodProduct($price, $expectedTva)

{

$product = **new** Product('Un produit', Product::*FOOD\_PRODUCT*, $price);

$this->assertSame($expectedTva, $product->computeTVA());

}

**public function pricesForFoodProduct()**

{

**return** [

[0, 0.0],

[20, 1.1],

[100, 5.5]

];

}

Grâce à l'annotation @dataprovider, PHPUnit est en mesure de récupérer les données via la méthode indiquée dans l'annotation (pricesForFoodProduct). Cette dernière doit retourner un tableau de tableau(x) avec autant d'éléments que de paramètre(s) que l'on souhaite passer à la méthode de test qui recevra les données pour les exploiter.

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

vendor/bin/phpunit --filter=testComputeTVAOtherProduct