# **Unit Testing : concepts fondamentaux et guide rapide**

**Crédits**: Ce tutoriel est une adaptation personnelle de :

https://jtreminio.com/2013/03/unit-testing-tutorial-introduction-to-phpunit/

# Unit Tests ou Tests Unitaires

Les **tests** **unitaires** sont une partie fondamentale du développement d’une application. Ils doivent être considérés pendant toutes les étapes du développement :

1. **Avant:** Installer et configurer les outils de test (dans notre cas, **XDebug** et **PHPUnit**)
2. **Pendant :** Créer de tests pour **chaque partie de l’application qui a été développée**, ça nous aidera à être plus confiants dans la stabilité de notre code. Ça nous permettra en plus de détecter **les possibles erreurs générées quand on rajoute du nouveau code** (le code existant devrait continuer à fonctionner ! ;))
3. **Après :** Chaque fois qu’**on met à jour l’application on peut rajouter de nouveaux tests**. Si on trouve un bug on peut aussi le lancer.

Les tests sont groupés en **test** **cases** qui se trouvent dans une **test suite**. PHPUnit est un outil qui nous sert à créer et lancer de tests.

Note : Il existe un autre type de tests (functional tests) qui vérifient le fonctionnement correct de l’ensemble de l’application, mais ici on traite les tests unitaires

# Quoi tester

En théorie, on doit **tester le bon fonctionnement de chaque classe. On doit tester chaque méthode** pour nous assurer qu’elle fonctionne correctement (de là le nom « **unit** test »). On doit **écrire une méthode « test »** pour chaque **comportement** de chacun de ces éléments.

Une **méthode** **d’une classe** peut avoir plein de comportements différents (ex. : une fonction agit d’une façon différente selon les paramètres d'entrée, et peut renvoyer une valeur ou une autre).

**Exemple :** nous avons une classe ClientManager contenant une méthode getClientById(int) qui renvoie un objet client. Nous voulons écrire de tests pour cette méthode.

Cette fonction a un comportement diffèrent selon l’élément recherché se trouve dans la BD et ou pas. Si l’élément existe, la méthode renverra un objet Client. Autrement elle renverra null.

**Un test, dans notre contexte, sera juste une méthode créée par nous-mêmes qui testera le bon fonctionnement d’une certaine méthode déjà existante.**

**Pour chaque comportement on devra écrire une fonction de test**. Dans notre cas on crée deux méthodes dans la classe test qui appellent la méthode getClientById. Dans la première (testObtenirClientExistant) on fournit à la méthode de la classe (getClientById) un id qui existe dans la BD et on vérifie qu’elle nous renvoie un resultSet ; dans la deuxième (testObtenirClientEchec) on l’envoie un id qui n’existe pas et on vérifie qu’on reçoit null.

Le but de ces deux méthodes est de nous assurer qu’on obtient la réponse appropriée dans chaque cas. On aura créé alors deux méthodes pour tester une méthode existante. Chaque méthode créée teste alors un comportement.

**La question qui nous vient à l’esprit est : pourquoi on n’utilise pas des instructions conditionnelles (if) pour tester tout ça ?** On discutera en cours sur ce sujet.

# Structure d’une classe de test

Pour créer de tests on doit créer **une ou plusieurs classes** **contenant les méthodes de test.** Il s’agit d’une classe normale **qui hérite de PHPUnit\_Framework\_Testcase**. Cette classe contient plusieurs méthodes dont le nom commence par le mot **test**.

Chaque méthode teste un comportement **grâce à une suite d’actions** (appels aux méthodes de la classe à tester) **et d’assertions**. Une assertion est une expression qui doit être évalue à **vrai**. **Si toutes les assertions sont évaluées à vrai, le test est accompli avec succès.**

**Une assertion est une affirmation d’un fait**. Il y a plein de types, comme par exemple :

* Affirmer qu’une variable contient une certaine valeur
* Affirmer qu’une variable est un objet d’une certaine classe
* Affirmer qu’un array contient une certaine valeur
* Etc…

On pourrait réaliser ces affirmations en utilisant des ifs

Ex :

**if ($prixFinal > 0) { return true; }** **// la méthode fonctionne proprement**

Mais nous allons disposer d’autres outils beaucoup plus clairs et puissants.

Pour mieux comprendre… passez à la pratique dans les sections suivantes !

# Installation de PHPUnit et XDebug

Nous allons utiliser deux outils pour réaliser les tests : **PHPUnit** lance les tests, **Xdebug** fournit des informations importantes sur les tests.

1. **Installez composer**
2. **Allez dans le dossier contenant le projet à tester**
3. **Installez phpunit**

composer require --dev phpuni t/phpunit

Cette commande installera la dernière version de phpunit. Si vous avez besoin d’une version spécifique vous pouvez l’indiquer en utilisant un fichier composer.json ou dans les paramètres de la commande **require**

1. **Installez XDebug**

Voir fichier guide **Installation de XDebug**

# Procédure générale

Voici la procédure générale pour réaliser un Unit Test. Lisez-la et appliquez-la dans la pratique dans la section suivante :

1. **Créez un dossier « src » contenant le code source de votre projet**
2. **Créez un dossier « tests » qui va contenir toutes les classes comprenant les tests** (PHPUnit lancera tous les fichiers qui se trouvent dans ce dossier et dont le nom finit par **Test.php).**

**Le nom de chaque classe qui réalise de tests finit par le mot Test** et le nom du fichier doit finir par le suffixe **Test.php**.

Le nom de chaque méthode doit commencer par « test »

1. **Créez les classes « test » contenant les méthodes de test**
2. **Créez un fichier de configuration phpunit.xml** (phpunit.xml dans la racine du projet d’exemple)

Ce fichier indique à PhpUnit la localisation des tests qu’il doit lancer, ainsi que d’autres options concernant le testing

1. Lancez **phpunit. L’executable se trouve dans <dossierProjet>\vendor\bin**

Ouvrez une console, allez dans le dossier du projet et tapez :

**.\vendor\bin\phpunit**

**Attention** : ne tapez pas uniquement **phpunit** car vous lancerez une ancienne version existante dans le path (dans xampp\php).

# Exemple pratique

**Projet : UnitTestingExemple**

Considérons une classe URL (URL.php) **contenant une méthode preparerURL qui reçoit une chaine de caractères et renvoie une URL valide**. Nous allons créer de tests unitaires pour cette méthode. On suit la procédure déjà mentionnée:

1. **Créez un dossier « src » contenant le code source de votre projet** (fichier URL.php, explications dans le code)
2. **Créez un dossier « tests » qui contiendra toutes les classes qui contiennent les tests**
3. **Créez ces classes « test » contenant au moins une méthode de test** (fichier **URLTest.php**, **suivez les explications dans le code**). Importez le fichier de la classe à tester (src/URL.php) Créez les **assertions** nécessaires.
4. **Créez un fichier de configuration phpunit.xml** (fichier phpunit.xml). Dans ce fichier vous indiquez :

* Un nom pour l'ensemble de tests
* Le dossier contenant les classes qui contiennent les tests
* La liste de dossiers accessibles par PhpUnit

1. Lancez **phpunit** depuis le dossier du projet

## Liste d’assertions de PHPUnit

<https://phpunit.de/manual/current/en/appendixes.assertions.html>

# Eviter la duplication du code (DRY)

Observez que dans la classe URLtest on duplique le code pour chaque test. Pour résoudre ce problème on va utiliser un **dataProvider** ou **fournisseur de données.** Un fournisseur de données peut être utilisé pour créer de multiples ensembles d'informations à transmettre en un seul test, supprimant la nécessité de créer des tests en double.

Au lieu de créer des méthodes d'essai multiples, vous créez **simplement une méthode unique qui accepte les paramètres correspondant aux données qui sont variables pour chaque test**, et vous créez aussi une méthode de fournisseur de données pour **fournir ces données**.

Vous avez un exemple complet et commenté dans UnitTestingExample2.