*Testez et suivez l’état de votre application avec PHP*

**Mise en place d’un outil pour implémenter ses tests unitaires**

* Un test se décompose en trois actions :

1. Instanciation d’une classe
2. Appel d’une méthode avec passage ou non de paramètres
3. Vérification de la sortie

Pour lancer les tests on utilise dans la console la commande suivante **: « bin/phpunit »**

* Le code coverage est un rapport indiquant si toutes les parties de notre code sont couverts par les tests ou non.

Pour récupérer le code coverage : **« «bin/phpunit –coverage-html web/test-coverage »**

**/!\ Attention :** Il faut que xdebug soit installé pour que le code coverage soit généré.

Il n’est pas forcement pertinent de tester la totalité du code, car cela peut s’avérer très chronophage ! Il faut en premier lieu se concentrer sur la logique centrale et/ou sensible de l’application.

**Premiers pars avec PHPUnit et les tests unitaires**

* Un test unitaire consiste à exécuter du code en provenance de l’application et à vérifier si tout s’est bien passé (résultat de sortie cohérent et/ou fonctions sensées être appelées correctement appelées). Les tests unitaires sont dits « tests boite blanche » parce qu’il faut en amont connaitre le code pour pouvoir le tester.
* Le nombre de tests à écrire pour tester une fonction donnée équivaut au nombre de « return » possibles de la dite fonction, additionnés à son nombre d’exceptions possibles. De la sorte on est sûr que tous les cas sont couverts.
* Un test doit être écrit dans une classe portant le même nom que la classe cible suffixé par « test » et contenue dans le dossier « tests » du projet tout en respectant l’arborescence de la classe cible : **« /tests/Bundle/Entity/CibleTest.php »**
* Les méthodes de test, elles, peuvent prendre n’importe quel nom mais sont obligatoirement préfixé par « test ».
* Il est possible, grâce aux data providers de passer un jeu de paramètres à nos tests. De la sorte PHPUnit va exécuter plusieurs fois le même test mais avec différentes valeurs en entrée, passé par le data provider.
* Si on ne veut pas exécuter tous les tests mais juste une seule fonction on peut utiliser l’option **« --filter=testFunction »**

**Les doublures (mocks)**

Le principe des doublures, ou mocks, est de ne pas dépendre d’autre système pour son test. Par exemple, si l’authentification dépend de Facebook, l’intérêt de créer un mock est de simuler le bon comportement de Facebook pour ne pas en dépendre lors de nos tests.

Dummy : Objet qui remplit un contrat

Stub : Dummy auquel est rajouté un comportement

Mock : Stub qui a des attentes (expectations). On précise le comportement attendu lors de l’appel d’une fonction.

Les doublures peuvent notamment s’utiliser lorsqu’une méthode de classe à tester requiert des dépendances, particulièrement si celles si sont difficiles à instancier. Au lieu d’instancier les dépendances, on crée des doublures pour chacune d’entre elles.

**/!\ Attention :** Plus le code contient de dépendances et plus il sera difficile à tester.

**TP1 : Tester une classe contenant de nombreuses dépendances**

* Dans un premier temps il faut instancier la classe à tester, autrement dit on s’arrête d’abord sur son constructeur et on instancie (avec le constructeur originel ou avec une doublure) un objet pour chaque dépendance dont il a besoin.
* Ensuite il ne nous reste plus qu’à fournir, pour chaque test, le résultat attendu en sortie avant de lancer le test.

**Que tester et quand le tester ?**

* Il est vain de vouloir atteindre 100% de couverture dans nos tests de code. En effet il est important de tester ce qui est important pour le buisines de notre application ! Autrement dit ce qui est réellement important et donc critique pour le bon fonctionnement de notre application.
* On peut écrire des tests à n’importe quel moment de vie de notre projet :
  + Avant l’écriture du code fonctionnel
  + Après l’écriture du code fonctionnel
  + A la rencontre d’un bug
* **/!\ Attention** à ne pas se dire par excès de confiance que le temps passé à l’écriture de tests est gaspillé

**En quoi consiste l’intégration continue ?**

Le but des tests et de valider le fonctionnement de notre code, et de surcroit celui de notre application en général. Les tests peuvent être lancés à la main mais il existe des solutions permettant de les rendre automatiques : c’est l’intégration continue.

Pour ce faire c’est Github qui est utilisé. En effet lorsque des modifications de code sont envoyées sur Github ce dernier va notifier notre serveur d’intégration continue pour qu’il ordonne le lancement des différents tests auprès des outils concernés (PHPUnit par exemple). Enfin le serveur d’intégration continue envoi une notification à Github avec le résultat. Le tout forme ce qu’on appelle un build.

**Define « build » :** *Le build correspond au fait de lancer l’ensemble de logiciel qui permettent de valider des changements dans l’application.*

**TP2 : Mettre en place un outil d’intégration continue : Travis**

* Travis dispose d’une intégration facilité à Github, d’où sa popularité.
* Par default un build et lancé à chaque pull / push

**Conclusion**

* Les tests doivent être indépendants les uns des autres.
* Les tests doivent être répétables à l’infini et toujours renvoyé un résultat « okay ».
* Les tests doivent avoir un caractère anticipatif en couvrant le plus de cas possible.