Les tests Unitaires avec PHPUnit et Symfony

Introduction

Les tests unitaires sont importants lorsque l'on code. Ils permettent de tester tout ou un fragment de votre code très rapidement, sans passez par l'interface graphique de votre site / application.

Pour les mettre en place sur Symfony, il nous faudra installer PHPUnit, et écrire un petit morceau de code.

installer PHPUnit

Pour installer PHPUnit sur votre projet, ouvrez une console (git bash par exemple) à la racine de votre projet et tapez la commande suivante :

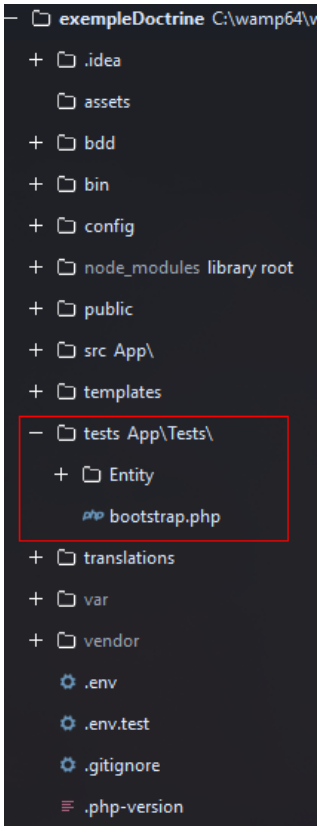
- `php bin/phpunit`

Cette commande sert de base au lancement de PHPUnit, mais s'il n'est pas encore installé sur votre projet, elle lancera automatiquement l'installation.

Si Vous relancez cette commande, un message apparaitra signalant qu'aucun test n'a été effectué. C'est normal, nous devons maintenant coder ces tests !

Pour l'exemple, nous testerons le système de validation de notre entité `Products`. Afin de rendre l'exemple opérationnel et pour bien comprendre le système des tests untaires, veillez à bien supprimer toutes formes de contraintes de validations sur cette entité (dans l'entité elle-même, ou bien dans `ProductsType.php`).

Pour l'écriture de ces tests, nous nous placerons dans le dossier `test`, à la racine de notre projet :



Une bonne pratique de mise en places des tests unitaires et de garder la même structure que celle de notre projet (dans le dossier `src`).

Le chemin de notre entité `Products` est `src\Entity\Products.php`, le chemin vers son test unitaire sera donc `test\Entity\ProductsTest.php`.

Créons donc maintenant notre classe `ProductsTest.php` comme ceci (attention aux imports, bien entendu) :

```php
<?php

namespace App\Tests\Entity;

use App\Entity\Products;
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class ProductsTest extends KernelTestCase
{

}
```

En ce qui concerne les classes de test, vous pouvez les étendre à plusieurs autres classes :

- TestCase : pour effectuer des tests de base.
- KernelTestCase : permet d'écrire des tests dans le context du kernel, utilisé pour des tests fonctionnel, context application, etc.
- WebTestCase : permet de tester des controllers, appli dans leur globalité

lorsque les tests sont effectués, il y a un changement d'environnement. C'est à dire : par défaut, lorsque PHPUnit sera lancé, Symfony va désactiver certaines fonctionnalités (désactivation mail, etc...). Il est possible de définir des réglages pour cet environnement de test, notamment d'indiquer une base de donnée de test. Pour cela il suffit de se rendre dans le fichier `.env.test` et de définir la base de données de test de la même manière que l'on a définit la base de données de notre projet. Au final, les même réglages que le fichier `.env` peuvent y être définis.

Test de l'entité `Products`

Ecrivons notre premier test permettant de tester les contraintes de validations de l'entité `Products`.

```php
<?php

namespace App\Tests\Entity;

use App\Entity\Products;
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class ProductsTest extends KernelTestCase
{
    public function testValid()
    {
        // définissions du produit à insérer en bdd
        $product = (new Products())->setProductName('Produit')
            ->setCategoryId(1)
            ->setUnitPrice(183)
            ->setQuantityPerUnit(8)
            ->setUnitsInStock(56)
            ->setUnitsOnOrder(32)
```

```
            ->setRedorderLevel(5)
            ->setDiscontinued(0)
            ->setDescription('test php unit');
        // validation
        self::bootKernel();
        // récupération depuis le container, le validator
        $error = self::$container->get('validator')->validate($product);
        // on s'attend à ne pas avoir d'erreur
        $this->assertCount(0, $error);
    }
}
```

Nous commençons par créer une fonction que l'on appellera `testValid()`. A l'intérieur, nous allons d'abord définir les propriétés de l'entité que nous voulons tester. Puis nous lançons la validation (`self::bootKernel();`), et nous récupérons le validator depuis le container (`$error = self::$container->get('validator')->validate($product);`). Enfin nous indiquons que nous attendons aucune erreur (l'insertion doit se faire sans erreur, vu que nous testons des données cohérentes avec notre base de données) et nous récupérons les erreurs.

Maintenant que notre première fonction de test est faite, nous pouvons nous rendre dans la console et lancer PHPUnit avec : `php bin/phpunit` > On peut filtrer les tests sur plusieurs classes de test existantes. Plutôt que de lancer tous les tests, on peut lancer une seule série de test de la manière suivante : `php bin/phpunit --filter <nom de la class de test>`

Si le test est valide, vous devriez avoir un résultat similaire à l'image ci-dessous :

ne tenez pas compte du message de dépréciation

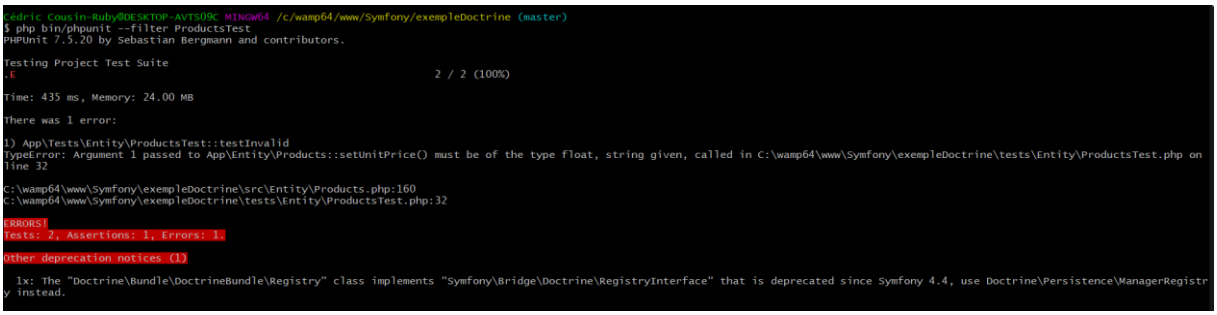
De la même manière, nous allons créer une méthode qui teste les contraintes de validations lorsque qu'une donnée n'est pas attendue en base de données :

```
 public function testInvalid() {
        $product = (new Products())->setProductName('Pro213duit')
            ->setCategoryId(1)
            ->setUnitPrice(183)
            ->setQuantityPerUnit(8)
            ->setUnitsInStock(56)
            ->setUnitsOnOrder(32)
            ->setRedorderLevel(5)
            ->setDiscontinued(0)
            ->setDescription('test php unit');
        // validation
        self::bootKernel();
        // récupération depuis le container, le validator
        $error = self::$container->get('validator')->validate($product);
// on s'attend à avoir 1 erreur
```

```
        $this->assertCount(1, $error);
    }
```

Seul changement ici, on indique que l'on s'attend à avoir une erreur (erreur de validation) : `$this->assertCount(1, $error);`.

S'il n'y a pas de contrainte de validation sur votre entité, vous devriez obtenir le résultat suivant :

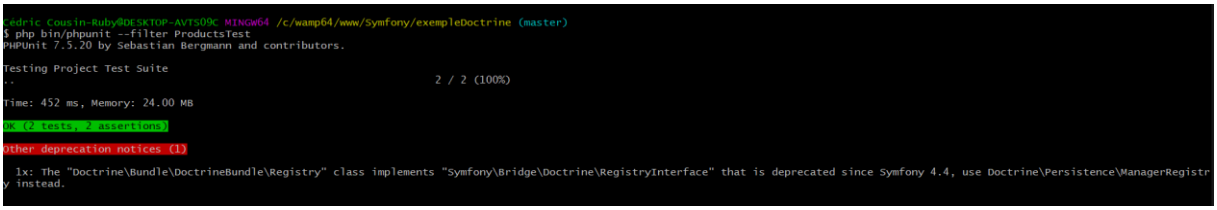


Une erreur est signalée, ainsi que la méthode sur laquelle elle a été faite. >ATTENTION : ne pas confondre l'erreur que l'on attend (ici la contrainte de validation doit retourner une erreur), et l'erreur indiquée par PHPUnit. Cette dernière indique justement que le test n'est pas valide.

Le test a échoué car il n'y a pas encore de contrainte de validation définie sur la propriété `$ProductsName` de notre entité. Ajoutons-y donc une contrainte de validations. On acceptera uniquement les lettres majuscule et minuscule, ainsi que les espaces :

```
 /**
     * @ORM\Column(type="float")
     * @Assert\Regex("/^[a-zA-Z\s]+$/")
     */
    private $ProductName;
```

Renouvelons le test :



Le test est validé. La contrainte de validation est prise en compte et ne laisse pas passer les chiffres. Il y a bien eu une erreur d'insertion.

Factorisation du code

Même au sein des tests unitaires, il important de garder une bonne structure sur le code, et éviter les répétitions. Dans notre cas, nous avons 2 méthodes qui ont presque le même code, à quelques détails prêts. Nous pouvons factoriser les choses de la manière suivante:

- Création d'une méthode pour récupérer l'entité

```
public function getEntity(): Products {
    return (new Products())->setProductName('test Produit')
      ->setCategoryId(1)
      ->setUnitPrice(183)
```

```
      ->setQuantityPerUnit(8)
      ->setUnitsInStock(56)
      ->setUnitsOnOrder(32)
      ->setRedorderLevel(5)
      ->setDiscontinued(0)
      ->setDescription('test php unit');
}
```

- Création d'une méthode pour la validation et récupération des erreurs :

```
public function assertHasError(Products $products, $number = 0)     {
      // validation         self::bootKernel();
      // récupération depuis le container, le validator
      $error = self::$container->get('validator')->validate($products);
      // on s'attend à ne pas avoir d'erreur
      $this->assertCount($number, $error);
}
```

- Appelle des méthodes dans les fonctions `testValid` et `testInvalid` :

```
public function testValid()
{
    $this->assertHasError($this->getEntity(), 0);
}

public function testInvalid()
{
    $this->assertHasError($this->getEntity()->setProductName('t23est
Produit'), 1);

}
```

Pour continuer notre exemple, faisons une méthode permettant le test sur une chaine de caractère vide :

```
  public function testInvalid()
    {
        // test pour chaine de caractères erroné
        $this->assertHasError($this->getEntity()->setProductName('t23est
Produit'), 1);
        // test pour une chaine de caractères vide
        $this->assertHasError($this->getEntity()->setProductName(''), 1);

    }
```

Nous avons une erreur qui s'affiche car il n'y pas encore de contrainte de validation indiquant que les chaines de caractères vides ne sont pas acceptées.

Ajouter donc une nouvelle contrainte de validation et relancez le test.

Mettre tous les tests dans une seule et même méthode n'est pas forcément pratique pour cibler un problème lorsque l'on en rencontre un. Il est préférable de faire une méthode pour chaque test :

```
public function testValid()
{
    $this->assertHasError($this->getEntity(), 0);
}

public function testDigitInvalid()
{
    $this->assertHasError($this->getEntity()->setProductName('t23est
Produit'), 1);
}
```

```php
public function testBlankInvalid()
{
    $this->assertHasError($this->getEntity()->setProductName(''), 1);
}
```

En enlevant la contrainte pour chaine de caractère vide :

```
There was 1 failure:

1) App\Tests\Entity\ProductsTest::testBlankInvalid
Failed asserting that actual size 0 matches expected size 1.

C:\wamp64\www\Symfony\exempleDoctrine\tests\Entity\ProductsTest.php:30
C:\wamp64\www\Symfony\exempleDoctrine\tests\Entity\ProductsTest.php:44

FAILURES!
Tests: 3, Assertions: 3, Failures: 1.

Remaining indirect deprecation notices (1)
```

On observe que PHPUnit nous signale une erreur : on attendait une erreur, hors aucune n'a été trouvé, et il nous indique sur quelle méthode se trouvent le problème

Récupération des erreurs

Il pourrait être intéressant de récupérer les erreurs de PHPUnit pour avoir un peu plus d'informations en cas de problème.

Vous pouvez procéder de la manière suivante :

- dans la méthode `assertHasError()` :

```php
public function assertHasError(Products $products, $number = 0)
{
    // validation
    self::bootKernel();
    // récupération depuis le container, le validator
    $errors = self::$container->get('validator')->validate($products);
    $messages = [];
    /**
     * @var ConstraintViolation $error
     */
    foreach ($errors as $error) {
        $messages[] = $error->getPropertyPath() . ' => ' . $error-
>getMessage();
    }
    // on s'attend à ne pas avoir d'erreur
    $this->assertCount($number, $errors, implode(',', $messages));
}
```

Conclusion

Lest tests unitaires ne concernent pas que la vérification des contraintes de validation, ils permettent aussi de tester des contrôleurs, une logique, un morceau de code, etc.