Les tests Phpunit & Symfony

1) Les tests unitaires Phpunit :

1 - Introduction :

Les tests unitaires peuvent se résumer par cette citation :

A chaque fois que vous avez la tentation de saisir quelque chose dans une instruction « print » ou dans une expression de débogage, écrivez-le plutôt dans un test. Martin Fowler

Ils permettent de tester des fonctionnalités morceau par morceau de manière régulière afin de vérifier que les changements apportés au code ne détériorent pas le reste de l'application. Ces tests sont automatisés et souvent associés à un processus de déploiement continu afin d'accélérer le développent d'une application pour la sortie de ses mises à jour. Plus tôt un bug est trouvé , plus il sera facile et rapide de le corriger. C'est pour cela que les tests sont une étape fondamentale lors du lancement d'un projet.

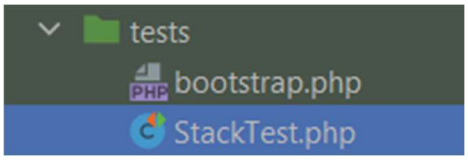
2 - Installation :

Documentation complète :
https://phpunit.readthedocs.io/en/stable/index.html

Dans notre projet GSB, les test sont réalisés grâce au package phpUnit installable via Composer :

```
symfony composer req phpunit –dev
```

Cela va créer s'il n'existe pas déjà , un dossier tests dans le projet :



Celui-ci répertorie tous les test de l'application

3 – Ecrire un test :

Les élément qui permettent d'effectuer les tests sont appelés assertions.

assertArrayHasKey()	assertFileExists()	assertJsonFileEqualsJsonFile()
assertClassHasAttribute()	assertFileIsReadable()	assertJsonStringEqualsJsonFile()
assertClassHasStaticAttribute()	assertFileIsWritable()	assertJsonStringEqualsJsonString()
assertContains()	assertGreaterThan()	assertLessThan()
assertStringContainsString()	assertGreaterThanOrEqual()	assertLessThanOrEqual()
assertStringContainsStringIgnoringCase()	assertInfinite()	assertNan()
assertContainsOnly()	assertInstanceOf()	assertNull()
assertContainsOnlyInstancesOf()	assertIsArray()	assertObjectHasAttribute()
assertCount()	assertIsBool()	assertMatchesRegularExpression()
assertDirectoryExists()	assertIsCallable()	assertStringMatchesFormat()
assertDirectoryIsReadable()	assertIsFloat()	assertStringMatchesFormatFile()
assertDirectoryIsWritable()	assertIsInt()	assertSame()
assertEmpty()	assertIsIterable()	assertSameSize()
assertEquals()	assertIsNumeric()	assertStringEndsWith()
assertEqualsCanonicalizing()	assertIsObject()	assertStringEqualsFile()
assertEqualsIgnoringCase()	assertIsResource()	assertStringStartsWith()
assertEqualsWithDelta()	assertIsScalar()	assertThat()
assertObjectEquals()	assertIsString()	assertTrue()
assertFalse()	assertIsReadable()	assertXmlFileEqualsXmlFile()
assertFileEquals()	assertIsWritable()	assertXmlStringEqualsXmlFile()
		assertXmlStringEqualsXmlString()

Liste non exhaustive des assertions disponibles dans phpunit.

La description complète de chaque assertion est disponible sur cette page : https://phpunit.readthedocs.io/en/stable/assertions.html

Voici un exemple de test simple :

```
<?php
namespace App\Tests;
use PHPUnit\Framework\TestCase;       //1

class StackTest extends TestCase      //2
{
    public function testEmpty()
    {
        $stack = [];                  //3
        $this->assertEmpty($stack);   //4

        return $stack;
    }
```

Pour commencer il faut utiliser la class *TestCase* [1] , puis créer une class qui hérite de cette class[2] . Ensuite, dans cette exemple, on crée un tableau *$stack* qui est vide [3] et on teste s'il est bien vide avec l'assertion *assertEmpty* [4] .

Une fonctionnalité intéressante est la dépendance des tests, si l'on ajoute à l'exemple précédent ceci :

```
    /**
     * @depends testEmpty
     */
    public function testPush(array $stack)
    {
        array_push($stack, 'foo');
        $this->assertSame('foo', $stack[count($stack)-1]);
        $this->assertNotEmpty($stack);
```

```
        return $stack; }
```

La balise *@depends* définit que le test suivant *testPush* sera exécuté uniquement si le test *testEmpty* est validé.

Les tests peuvent aussi être exécutés sur des entités comme sur cet exemple :

```
<?php
namespace App\Tests\entity_tests;


use PHPUnit\Framework\TestCase;
use App\Entity\Invitation;


class InvitTest extends TestCase
{
    public function testSetStatus()
    {
        $invit = new Invitation();
        $invit->setStatus("envoyer");
        $test= $invit->getStatus();


        $this->assertStringContainsString('envoyer',$test);
    }
}
```

On peut également utiliser une fonction séparée pour fournir des valeurs de test : on appelle cela un Data Provider. Voici un exemple :

```php
<?php
use PHPUnit\Framework\TestCase;
class DataTest extends TestCase{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected){
        $this->assertEquals($expected, $a + $b);}
    public function additionProvider()
    {
        return [
            [0, 0, 0],
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
//     return [
//           'adding zeros'  => [0, 0, 0],
//           'zero plus one' => [0, 1, 1],
//           'one plus zero' => [1, 0, 1],
//           'one plus one'  => [1, 1, 3]
//       ];
//     return new CsvFileIterator('data.csv');
    }
}
?>
```

4 – Exécuter les tests :

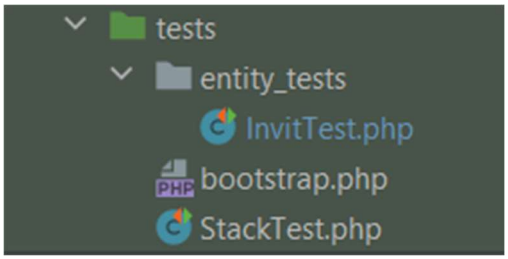
Une fois les tests écrits, il suffit de lancer phpunit a l'aide de cette commande :

```
./vendor/bin/phpunit
```

Ceci va exécuter tous les tests présents dans le dossier tests.

Il est aussi possible d'exécuter un fichier ou un sous-dossier précis :

```
./vendor/bin/phpunit tests/StackTest.php
```




```
./vendor/bin/phpunit tests/entity_tests
```

On obtient donc ceci dans le terminal de commande :

```
PHPUnit 9.5.10 by Sebastian Bergmann and contributors.

Testing
....                                                                4 / 4 (100%)

Time: 00:00.229, Memory: 10.00 MB

OK (4 tests, 6 assertions)
```

Chaque point est un test réussi. Lorsqu'un test échoue , un F est présent et lorsque la syntaxe d'un test est incorrecte un E apparaît :

```
Testing
...F                                                                4 / 4 (100%)

Time: 00:00.038, Memory: 10.00 MB

There was 1 failure:

1) App\Tests\entity_tests\InvitTest::testSetStatus
Failed asserting that 'envoyer' contains "autre".

C:\Users\mbkiw\PhpstormProjects\pharma-symfony-testing\tests\entity_tests\InvitTest.php:16

FAILURES!
Tests: 4, Assertions: 6, Failures: 1.
```

```
Testing
...E                                                                4 / 4 (100%)

Time: 00:00.040, Memory: 10.00 MB

There was 1 error:

1) App\Tests\entity_tests\InvitTest::testSetStatus
Error: Call to undefined method App\Tests\entity_tests\InvitTest::assertStringContainsStrin()

C:\Users\mbkiw\PhpstormProjects\pharma-symfony-testing\tests\entity_tests\InvitTest.php:16

ERRORS!
Tests: 4, Assertions: 5, Errors: 1.
```

2) Exécution des tests via le pipeline gitlab CI/CD :

La finalité des tests est qu'ils soient exécutés automatiquement à chaque déploiement d'une nouvelle version de l'application. C'est pour cela que gitlab a créé le pipeline CI/CD qui est un fichier exécuté à chaque push dans une branche.

Ce fichier se nomme *.gitlab-ci.yml :*

```yaml
image: jakzal/phpqa:php7.4
before_script:
  - composer install --no-scripts
cache:
  paths:
    - vendor/
stages:
  - SecurityChecker
  - CodingStandards
  - UnitTests
security-checker:
  stage: SecurityChecker
  script:
    - local-php-security-checker security:check composer.lock
  allow_failure: false
twig-lint:
  stage: CodingStandards
  script:
    - twig-lint lint ./templates
  allow_failure: false
phpunit:
  stage: UnitTests
  script:
    - ./vendor/bin/phpunit
  allow_failure: false
```

En premier lieu, l'application est hébergée sur un image Docker , ici *php7.4*

Docker est un programme de haut niveau qui permet d'empaqueter une application ainsi que ses dépendances dans un conteneur isolé et qui pourra ainsi être exécutée sur n'importe quel serveur (Windows, Mac, Linux). On peut piloter Docker facilement grâce à une panoplie de commandes.

Ensuite, nous installons *composer* et définissons les différentes étapes (*stage*) que la machine va exécuter.

La première étape *security-chercker* est un bundle de Symfony qui vérifie si les dépendances de l'application ne pas atteintes par des failles de sécurité sérieuses. Celui-ci va vérifier si de nouvelles mises à jour des packages sont disponibles sur le référentiel de Symfony et va en signaler la présence si c'est le cas.

Puis *twig-lint* va vérifier la syntaxe des vues encodées en twig.

Enfin on lance *phpunit* pour exécuter les tests développés en amont.

Pour finir nous pouvons observer les résultat sur la page CI/CD du dépôt gitlab :

Le pipeline est exécuté lorsqu'une branche a subi une modification, c'est-à-dire un push, un merge ou une modification dans gitlab même.