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 morceaux par morceaux de manière régulière afin de vérifier que les changement apporter au code ne détériore pas le reste de l'application. Ces tests sont automatisés et souvent associer à un processus de déploiement continu afin d'accélérer le développent d'une application pour la sortie de ses mise 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.

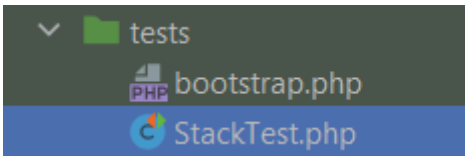
2 - Installation :

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

Dans notre projet GSB, les test sont réalisé 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 permette d'effectuer les tests sont appeler assertion.

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 disponible 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
<?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 utilisé[2] . En suite dans cette exemple on créer un tableau *$stack* qui est vide [3] et on test 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 précèdent ceci :

```php
    /**
     * @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* défini que le test suivant *testPush* sera exécuter uniquement si le test *testEmpty* est validé.

Les tests peuvent aussi être exécuter sur des entité comme sur cet exemple :

L'exemple précèdent ceci :

```
<?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);
    }
}
```

4 – Exécuter les test :

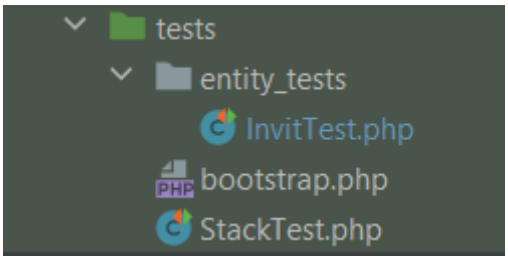
Une fois les tests écrit, il suffit de lancer phpunit a l'aide de cette commande :

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

Ceci va exécuter tous les test présent dans le dossier test.

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 dans le terminal ceci :

```
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 , il F est présent et lorsque la syntaxe d'un test est incorrecte un E apparait :

```
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.
```

5 – 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 créer le pipeline CI/CD qui consiste en un fichier qui est exécuter a chaque push de celui-ci 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*

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

En premier *security-chercker* est un bundle de Symfony qui vérifie si les dépendance que l'application utilise n'a pas de faille de sécurité sérieuse.

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

Enfin on lance *phpunit* pour exécuter les tests développer plus tôt.

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