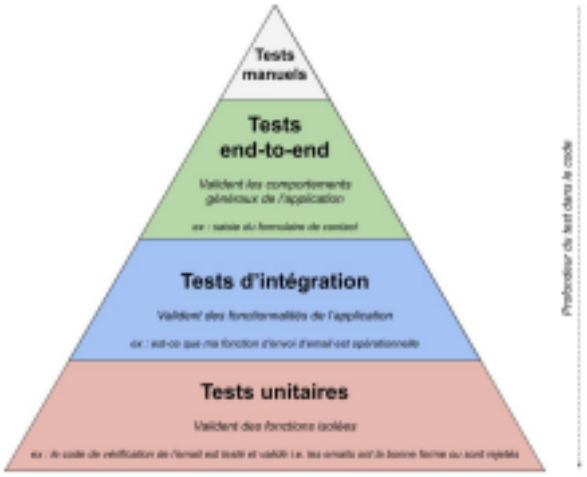
TP Symfony - Tests unitaires et d'intégration

Le développement de logiciels nécessite des mécanismes de vérification pour s'assurer que le produit fonctionne comme prévu. On dénombre 4 types de tests qu'on peut présenter sous forme d'une pyramide.



Comme le montre le tableau comparatif ci-après, Les tests end-to-end et manuels sont relativement complexes à mettre en œuvre car ils nécessitent de la main d'œuvre (ex: équipe de test produit, analyste Quality Insurance, etc) ou des outils avancés pour simuler des interactions humaines. Les tests unitaires et les tests fonctionnels, ou d'intégration, ont, quant à eux, la particularité d'être facilement automatisables. Nous allons donc nous focaliser sur ces deux cas dans ce TP.

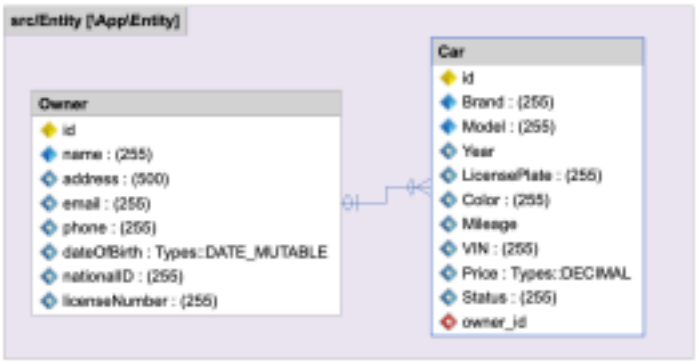
Critère	Tests unitaires	Tests d'intégration	Tests end-to-end	Test manuels
Portée	Petite (fonction / méthode)	Large (système / application entière)	Spécifique à une application ou un module	Spécifique à une tâche ou un scénario
Objectif	Vérifier le bon fonctionnement du code	Vérifier la conformité aux exigences	Vérifier l'intégration et la performance d'une application	Vérifier le comportement et l'interface utilisateur
Fréquence d'exécution	Très fréquente	Selon les besoins	À des étapes clés du développement	Avant les livraisons ou les mises en production
Automatisation	Généralement automatisé	Peut être automatisé	Souvent automatisé	Généralement manuel
Exécutant	Développeur	Développeur / Testeur / Analyste QA	Testeur / Analyste QA	Testeur / Utilisateur

Lien avec le référentiel

Bloc de compétences n°2 Option « Solutions logicielles et applications métiers » Conception et développement d'applications	
Compétences	**Savoirs associés**
Concevoir et développer une solution applicative ● Exploiter les ressources du cadre applicatif (framework). ● Réaliser les tests nécessaires à la validation ou à la mise en production d'éléments adaptés ou développés. ● Exploiter les fonctionnalités d'un environnement de développement et de tests.	**Savoirs technologiques** ● Architectures applicatives : concepts de base et typologies. ● Programmation au sein d'un cadre applicatif (framework) : structure, outil d'aide au développement et de gestion des dépendances, techniques d'injection des dépendances. ● Techniques et outils de tests et d'intégration de composants logiciels.

Prérequis

Pour ce TP, nous devons disposer d'un jeu de données exploitable. Nous utiliserons la projet **Garage** du TP précédent. Les entités avaient été alimentées par nos soins via des données CSV pour créer une API REST.



1 - Tests unitaires

1.1 Une application Web Symfony est normalement préinstallée avec les tests unitaires opérationnels. Pour le vérifier, lancez la commande suivante.

```
bin/phpunit
```

Que se passe-t-il?
Quand nous lançons la commande, le message suivant est affiché :

```
PHPUnit 9.6.13 by Sebastian Bergmann and contributors.
No tests executed!
```

1.2 Faire une recherche sur **PHPUnit** et présenter en quelques lignes sont concept.
PHPUnit est un framework open source de tests unitaires dédié au langage de programmation PHP. Il permet l'implémentation des tests de régression en vérifiant que les exécutions correspondent aux assertions prédéfinies.

1.3 Dans les tests unitaires, le terme "**assertion**" revient régulièrement. Faire une recherche et expliquer ce que ce terme signifie.
Une assertion est une expression qui doit être évaluée comme vraie. Si cette évaluation échoue elle peut mettre fin à l'exécution du programme, ou bien lancer une exception.

1.4 Lancer ensuite la commande suivante

```
symfony console make:test
```

Que nous propose-t-elle? Classer les dans les 4 types de test existant sous forme d'un tableau.

`TestCase` : tests PHPUnit de base	**test unitaire**
`KernelTestCase` : tests de base qui ont accès aux services Symfony	**test d'intégration**

`WebTestCase` : pour exécuter des scénarios de type navigateur, mais qui n'exécutent pas de code JavaScript	**test d'intégration**
`ApiTestCase` : pour exécuter des scénarios orientés API	**test unitaire**
`PantherTestCase` : pour exécuter des scénarios e2e (end to end), en utilisant un vrai navigateur ou un client HTTP et un vrai serveur web	**test end to end**

Nous pouvons voir que le test manuel est manquant dans le tableau.

1.5 Choisir **TestCase** et valider puis nommer notre classe **CarTest**. Quel fichier est créé dans notre projet?
Création du fichier « **`tests/CarTest.php`** »

1.6 Relancer la commande suivante :

```
bin/phpunit
```

Que se passe-t-il?
La commande exécute les tests unitaires contenus dans le répertoire « **`tests`** » et nous informe du nombre de tests, du nombre de tests affirmés (assertion) et du nombre de tests échoués.

1.7 Dans la méthode ***`testSomething()`***, de la classe **`CarTest`**, modifier l'appel à la méthode ***`assertTrue()`*** afin que son paramètre d'entrée soit à **`false`**. Relancer la commande des tests unitaires. Que se passe-t-il maintenant?

> **Tips and tricks:** dans une classe de test unitaire, toutes les méthodes commençant par "test" sont exécutées comme autant de tests indépendants.

Désormais, le test unitaire nous informe que 1 test à échoué.

1.8 Remplacer maintenant la méthode de test par défaut par la méthode suivante

src/tests/CarTest.php

```php
    public function testBrand(): void
    {
        $myCar = New Car();
        $myCar->setBrand("Toyota");

        if( $myCar->getBrand() === "Toyota")
        {
            $this->assertTrue(true);
        }
        else
        {
            $this->assertTrue(false);
        }
    }
```

Penser à rajouter la ligne :

```php
use App\Entity\Car;
```

Relancer la commande des tests. Que se passe-t-il maintenant? Expliquer pourquoi et documenter le code en conséquence.

Dans le test, nous avons créé dans la fonction « `testBrand()` » une nouvelle instance de l'objet class « **`Car`** » avec la variable « `$myCar` », dans laquelle nous définissons le champ ***brand*** sous « `Toyota` ». Nous vérifions que le champ ***brand*** de la variable « `$myCar` » soit « `Toyota` » pour renvoyer **true**, sinon nous renvoyons **false**.

1.9 Ajouter la méthode de test suivante

src/tests/CarTest.php

```php
    public function testMileage(): void
    {
        $myCar = New Car();
        $myCar->setMileage(-10000);

        if( $myCar->getMileage() === -10000)
        {
            $this->assertTrue(true);
        }
        else
        {
            $this->assertTrue(false);
        }
    }
```

Relancer la commande des tests. Que se passe-t-il maintenant? Est-ce fonctionnellement normal?

La commande nous informe que deux tests ont été effectués. Oui, c'est fonctionnellement normal.

1.10 Modifier le code de la classe Car pour borner le kilométrage sur la plage [0,∞[. Modifier également le test unitaire pour qu'il soit cohérent avec cette règle: effectuer un test avec kilométrage négatif, un test avec kilométrage à zéro et un test avec kilométrage positif.

Car.php

```php
public function setMileage(?int $mileage): static
{
    if($mileage < 0)
    {
        $mileage = null;
    }

    $this->mileage = $mileage;

    return $this;
}
```

Le test avec kilométrage ***négatif*** renvoie un **échec**, tandis que les tests avec kilométrage ***à zéro*** et kilométrage ***positif*** renvoient une **assertion**.

1.11 Notre classe de test hérite (extends) de la classe **TestCase** qui, elle-même, hérite de la classe mère **Assert**. Explorer la classe **Assert** et lister sous forme d'un tableau les méthodes d'assertion qu'elle propose avec la documentation associée (ce sont toutes les méthodes commençant par assert...).

Tips and tricks: Les assertions sont fondamentales pour les tests unitaires. Elles définissent ce qui est considéré comme un comportement correct ou incorrect pour l'unité de code testée. Sans assertions, il serait difficile de déterminer si un test a réussi ou échoué.

2- Tests fonctionnels ou d'intégration

Un **test d'intégration** teste une plus grande partie de votre application par rapport à un test unitaire (par exemple, une combinaison de services). Les tests d'intégration peuvent avoir à utiliser le noyau (kernel) de Symfony pour récupérer des services dans le conteneur d'injection de dépendances. Ils sont donc un peu plus complexes à configurer.

2.1 Comme pour les tests unitaires, lancer la commande suivante

```
bin/console make:test
```

- Test type would you like: **KernelTestCase**
- Class name for your test: **GarageTest**

Relancer la commande des tests :

```
bin/phpunit
```

Que se passe-t-il? Pouvons-nous supprimer ces warnings?
La commande nous indique qu'il y à des avis de dépréciation (warning). Oui, nous pouvons supprimer ces warnings.

2.2 Recopier maintenant dans le fichier "**.env.test**" la ligne de configuration de notre base de données (clé DATABASE_URL) et **modifier le nom de la base en data-test.db**

.env.test

```
# define your env variables for the test env here
KERNEL_CLASS='App\Kernel'
APP_SECRET='$ecretf0rt3st'
SYMFONY_DEPRECATIONS_HELPER=999999
PANTHER_APP_ENV=panther
PANTHER_ERROR_SCREENSHOT_DIR=./var/error-screenshots
DATABASE_URL="sqlite:///%kernel.project_dir%/var/data-test.db"
```

Ensuite lancer les commandes suivantes :

```
bin/console doctrine:database:create --env=test
bin/console doctrine:migrations:migrate -n --env=test
bin/console doctrine:fixtures:load --env=test
```

Analyser et expliquer ce que nous venons de faire.
Nous avons créé une nouvelle base de données sous le nom « **data-test.db** » avec la commande « **bin/console doctrine:database:create --env=test** ».
Son chemin absolu est :
/home/[dossier(s) intermédiaire(s)]/garage/var/data-test.db

La commande « **bin/console doctrine:migrations:migrate -n --env=test** » permet la migration de la nouvelle base de données « **data-test.db** ».

Pour finir, la commande « **bin/console doctrine:fixtures:load --env=test** » va libérer de l'espace dans la base de données et supprimer les données obsolètes qui ne sont plus utiles.

```
Careful, database "main" will be purged. Do you want to continue?
(yes/no) [no]:
> yes

> purging database
> loading App\DataFixtures\AppFixtures
> loading App\DataFixtures\GarageFixture
```

2.3 Nous souhaitons vérifier le nombre d'éléments dans la base de test. Replacer la méthode par défaut par la suivante:

src/tests/GarageTest.php

```php
public function testCars(): void
{
    self::bootKernel();
    $container = self::$kernel->getContainer();

    $carRepository =
    $container->get('doctrine')->getRepository(Car::class);
    $count = $carRepository->count([]);

    $this->assertEquals(19, $count);
}
```

Note: attention à l'inclusion de la classe Car.

```php
use App\Entity\Car;
```

Lancer la commande des tests puis analyser et expliquer pourquoi le test est valide.

Le test vérifie que nous ayons exactement le même nombre d'éléments que la valeur contenu dans la méthode « **assertEquals** », à savoir 19. Il est valide étant donné que nous avons 19 éléments (voitures) dans la table « **Car** ».

2.4 Ajouter la méthode de test suivante

```
public function testRedCars(): void
{
    self::bootKernel();
    $container = self::$kernel->getContainer();

    $carRepository =
    $container->get('doctrine')->getRepository(Car::class);
    $count = $carRepository->count(['Color' => 'red']);

    echo "-------\n";
    echo $count."\n";

    $this->assertEquals(3, $count);
}
```

Lancer la commande des tests puis analyser et expliquer pourquoi le test est ici invalide.
Le test est invalide, étant donné que la commande « **`$count = $carRepository->count(['Color' => 'red']);`** » est incorrect.

2.5 Apporter les corrections nécessaires pour que le test précédent soit validé.
Pour que le test soit valide, il faut écrire le nom du champ "**`Color`**" avec une minuscule et le nom de la donnée recherchée "**`red`**" avec une majuscule, étant donné que le programme est **key sensitive**.

2.6 En quoi les tests fonctionnels sont-ils plus avancés que les tests unitaires?
Là où les tests unitaires vérifient le bon fonctionnement des requêtes, les tests fonctionnels servent à savoir si les résultats fournis par les requêtes correspondent aux résultats attendus.