Symfony

Usages et principes

Maxime Puys

22 mars 2025

Plan

1 Introduction Générale
2 Introduction à Symfony
3 Bundles et Packages
4 Jeux de Test

Introduction Générale

Utilité des Frameworks PHP

- **Développement plus rapide** :
  - Simplicité pour créer des applications web
  - Utilisation de librairies existantes (routage, ORM, etc)
- **Meilleur organisation** :
  - Découpage forcé par le framework
  - Plus petits fichiers
  - Oragnisation claire pour quelqu'un qui connait le framework
- **Sécurité** :
  - Framework gère SA propre sécurité
  - Framework offre des fonctions de sécurité
  - Attention à mettre à jour le framework et les packages / !\

Principes de développement

- **DRY** : “Don’t repeat yourself”
  - Code qui se répète (y compris dans les tests)
  - Données dupliquées en BDD vs. dans un fichier de conf vs. en dur, etc
- **KISS** : “Keep it simple, stupid”
  - Chaque composant a une seule tache simple
  - Si trop complexe, on le découpe

Principes de développement

- **SOLID** :
  - **Single-responsibility** : Une classe doit avoir une seule tâche.
  - **Open–closed** : Les objets ou entités devraient être ouverts à l'extension mais fermés à la modification.
  - **Liskov substitution** : Chaque sous-classe doit être substituable au niveau de leur classe parent.
  - **Interface segregation** : Une interface trop générale oblige à implémenter des méthodes inutiles pour certaines classes.
  - **Dependency inversion** : Une classe doit dépendre de son abstraction, pas de son implémentation. Autrement dit, on évite de passer des objets en paramètre lorsqu'une interface est disponible.

Rappels MVC

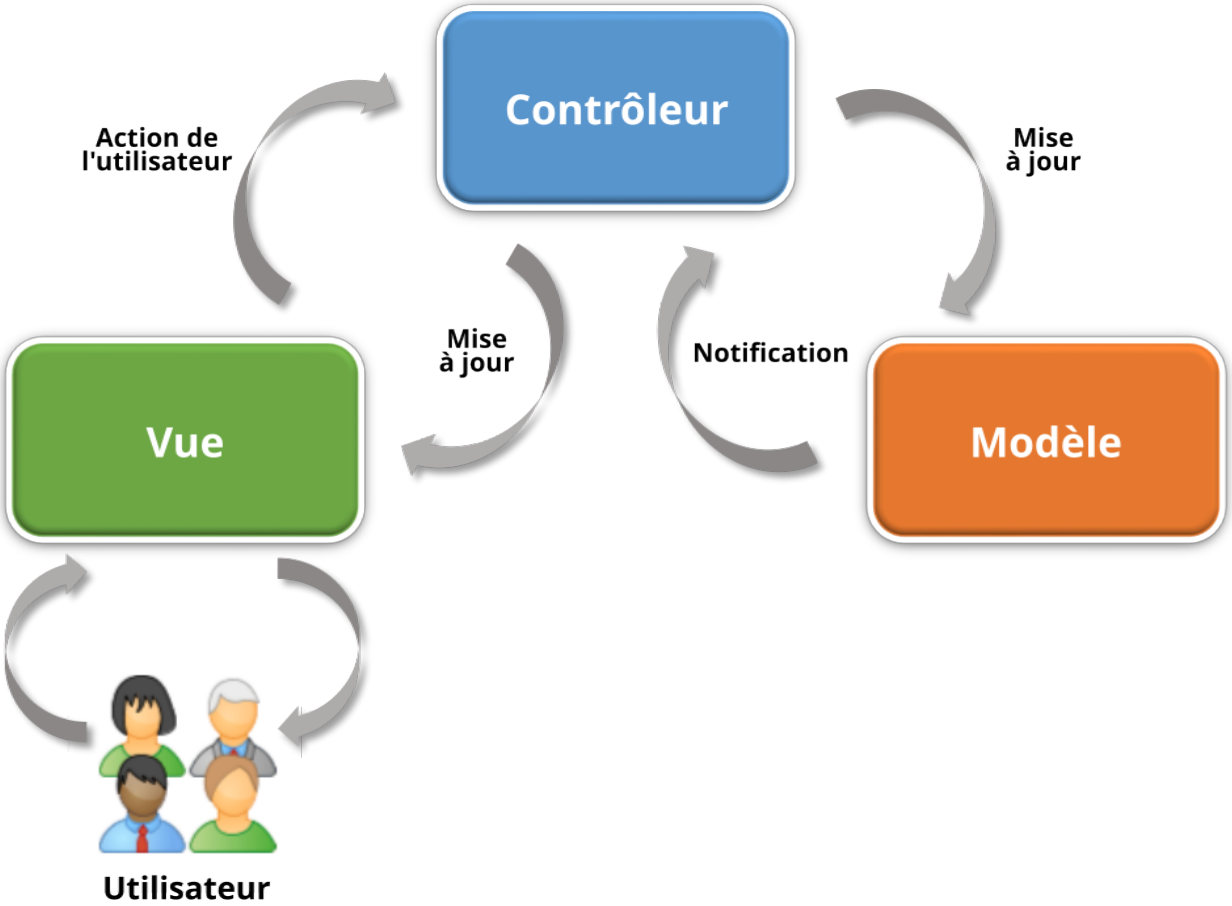

Figure – MVC - Wikipedia

Rappels ORM

- Sans requête préparée :

```php
$requete = "SELECT * FROM ma_table WHERE mon_champ = ma_valeur";
$resultats = $connexion->query($requete);
```

- Avec requête préparée :

```php
$requete = "SELECT mon_champ FROM ma_table WHERE mon_champ = :ma_valeur";
$resultats = $connexion->prepare($requete);
$resultats->execute([":ma_valeur" => 5]);
```

- Avec un ORM :

```php
$user = new User();
$user->setName('John');
$user->setPassword('Doe');
$user->save();
```

Rappels ORM

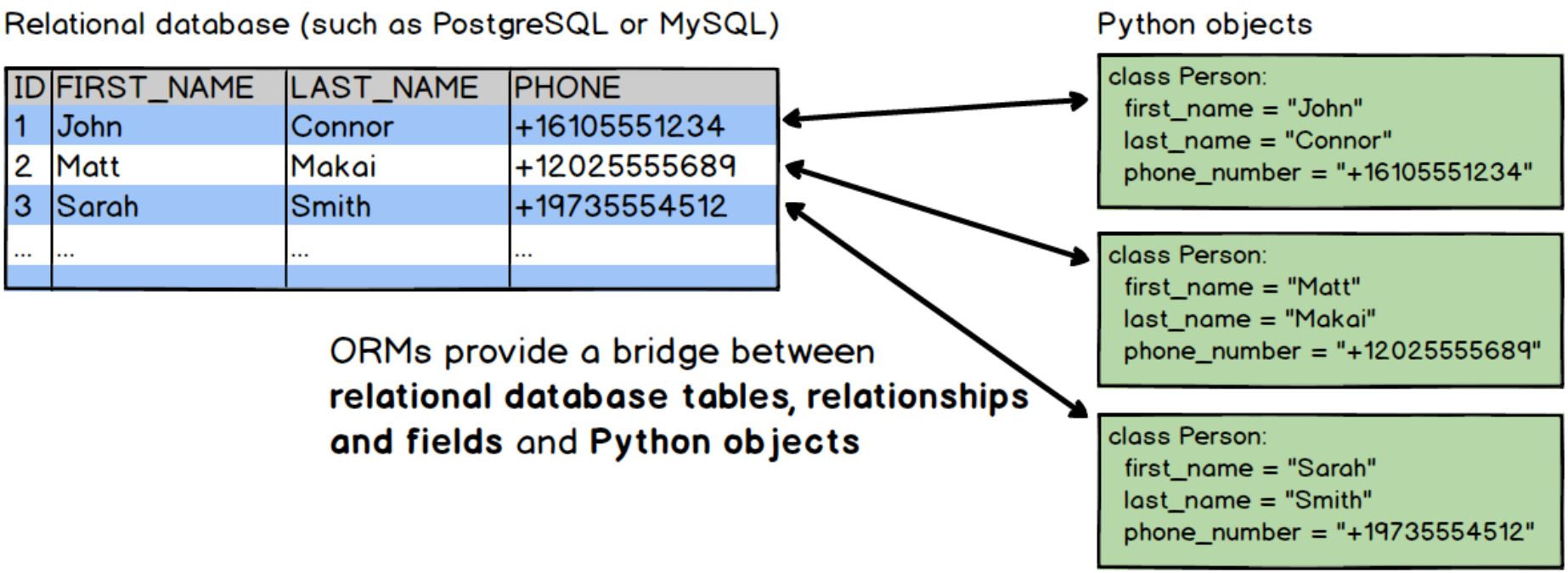

Figure – ORM - Full Stack Python

Introduction à Symfony

Bible de Symfony

- A lire et relire :
  - `https://symfony.com/doc/current/index.html`

Symfony vs. PHP seul

Front controller sans Symfony :

```php
// index.php

// load and initialize any global libraries
require_once 'model.php';
require_once 'controllers.php';

// route the request internally (or use router package e.g. AltoRouter)
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ('/index.php' === $uri) {
    list_action();
} elseif ('/index.php/show' === $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('HTTP/1.1 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

Symfony vs. PHP seul

Modèle sans Symfony :

```php
// model.php
function get_post_by_id($id)
{
    $connection = new PDO('mysql:host=localhost;dbname=test', $user, $pass);
    //connection established

    $query = 'SELECT created_at, title, body FROM post WHERE id=:id';
    $statement = $connection->prepare($query);
    $statement->bindValue(':id', $id, PDO::PARAM_INT);
    $statement->execute();

    $row = $statement->fetch(PDO::FETCH_ASSOC);

    $connection = null;
    //connection closed

    return $row;
}
```

Symfony vs. PHP seul

Controller sans Symfony :

```php
// controllers.php
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

Symfony vs. PHP seul

Vue sans Symfony :

```
1  <!-- templates/show.php -->
2  <?php $title = $post['title'] ?>
3  <?php ob_start() ?>
4      <h1><?= $post['title'] ?></h1>
5      <div class="date"><?= $post['created_at'] ?></div>
6      <div class="body">
7          <?= $post['body'] ?>
8      </div>
9  <?php $content = ob_get_clean() ?>
0  <?php include 'layout.php' ?>
```

Symfony vs. PHP seul

Front controller avec Symfony (index.php généré automatiquement) :

```yaml
# config/routes.yaml
blog_list:
    path:     /blog
    controller: App\Controller\BlogController::list

blog_show:
    path:     /blog/show/{id}
    controller: App\Controller\BlogController::show
```

```php
// public/index.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../src/Kernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new Kernel($_SERVER['APP_ENV'], false);
$kernel->handle(Request::createFromGlobals())->send();
```

Symfony vs. PHP seul

Modèle avec Symfony (généré automatiquement) :

```php
namespace App\Entity;
use App\Repository\BlogPostRepository;
use Doctrine\ORM\Mapping as ORM;
#[ORM\Entity(repositoryClass: BlogPostRepository::class)]
class BlogPost
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column]
    private ?\DateTimeImmutable $created_at = null;
    #[ORM\Column(length: 255)]
    private ?string $title = null;

    #[ORM\Column(length: 255)]
    private ?string $body = null;

    // Getters and Setters
}
```

Symfony vs. PHP seul

Modèle avec Symfony (généré automatiquement) :

```php
namespace App\Repository;
use App\Entity\BlogPost;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

/**
 * @extends ServiceEntityRepository<BlogPost>
 *
 * @method BlogPost|null find($id, $lockMode = null, $lockVersion = null)
 * @method BlogPost|null findOneBy(array $criteria, array $orderBy = null)
 * @method BlogPost[]    findAll()
 * @method BlogPost[]    findBy(array $criteria, array $orderBy = null, $limit = nu
 */
class BlogPostRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, BlogPost::class);
    }
}
```

Symfony vs. PHP seul

Controller avec Symfony (en partie généré automatiquement) :

```php
// src/Controller/BlogController.php
namespace App\Controller;
use App\Entity\Post;
use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class BlogController extends AbstractController {
    public function list(EntityManagerInterface $entityManager): Response
    {
        $posts = $entityManager->getRepository(Post::class)->findAll();
        return $this->render('blog/list.html.twig', ['posts' => $posts]);
    }

    public function show(EntityManagerInterface $entityManager, $id): Response
    {
        $post = $entityManager->getRepository(Post::class)->find($id);
        if (!$post) {
            throw $this->createNotFoundException();
        }
        return $this->render('blog/show.html.twig', ['post' => $post]);
    }
```

Symfony vs. PHP seul

Vue avec Symfony :

```
1  {# templates/blog/list.html.twig #}
2  {% extends 'base.html.twig' %}
3  {% block title %}List of Posts{% endblock %}
4  {% block body %}
5  <h1>List of Posts</h1>
6  <ul>
7      {% for post in posts %}
8      <li>
9          <a href="{{ path('blog_show', { id: post.id }) }}">
0              {{ post.title }}
1          </a>
2      </li>
3      {% endfor %}
4  </ul>
5  {% endblock %}
```

Gestion des Requêtes/Réponses avec PHP

```php
$uri = $_SERVER['REQUEST_URI'];
$foo = $_GET['foo'];

header('Content-Type: text/html');
echo 'The URI requested is: '.$uri;
echo 'The value of the "foo" parameter is: '.$foo;
```

- PHP abstrait les requêtes et réponses avec des variables superglobales
- La fonction `header` spécifie les headers HTTP de la réponse
- PHP génère la réponse avec les headers et le contenu affiché

```
HTTP/1.1 200 OK
Date: Sat, 03 Apr 2011 02:14:33 GMT
Server: Apache/2.2.17 (Unix)
Content-Type: text/html

The URI requested is: /testing?foo=symfony
The value of the "foo" parameter is: symfony
```

Gestion des Requêtes/Réponses avec Symfony

- La classe `Request` facilite l'accès aux informations (URI, paramètres, headers, cookies, etc).

```php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// Get the URI being requested (e.g., /about) minus any query parameters
$request->getPathInfo();

// Retrieve $_GET and $_POST variables respectively
$request->query->get('id');
//Second parameter is default value if variable not exists
$request->request->get('category', 'default category');

// Retrieve $_SERVER variables
$request->server->get('HTTP_HOST'); //given by .env file for example

// Retrieve an instance of UploadedFile identified by "attachment"
$request->files->get('attachment');
```

Gestion des Requêtes/Réponses avec Symfony

```php
// Retrieve a $_COOKIE value
$request->cookies->get('PHPSESSID');

// Retrieve an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content-type');

$request->getMethod();    // e.g., GET, POST, PUT, DELETE, or HEAD
$request->getLanguages(); // an array of languages the client accepts
```

Gestion des Requêtes/Réponses avec Symfony

- La classe `Response` spécifie le contenu, les codes HTTP, les headers, etc
- Le module `HttpFoundation` permet aussi de gérer les sessions, cookies, etc

```php
use Symfony\Component\HttpFoundation\Response;

$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(Response::HTTP_OK);

// Set a HTTP response header
$response->headers->set('Content-Type', 'text/html');

// Print the HTTP headers followed by the content
$response->send();
```

Les commandes Symfony

Symfony propose également un composant appelé **Console**. Ce composant permet de créer des scripts PHP bénéficiant de toute la puissance de Symfony qui pourront être lancés en ligne de commande.
Les commandes Symfony deviennent le deuxième point d'entrée d'une application Symfony.

- Une application Symfony réagit soit :
  - par un appel HTTP (donc un Controller derrière)
  - soit une ligne de commande (tâches automatisées, maintenance manuelle)

```
1  $ composer require symfony/console
```

Ces commandes permettent beaucoup d'applications :

- Gestion de tâches automatisées (par exemple avec le crontab)
- Exécution de commandes personnalisées (arguments, options..)
- Gestion de migrations (Doctrine migrations)

Première commande Symfony !

```php
use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;

#[AsCommand(
    name: 'app:create-user',
    description: 'Creates a new user.'
)]
class CreateUserCommand extends Command
{
    protected function configure(): void {
    $this->addArgument('username', InputArgument::REQUIRED, 'The username');
}
    protected function execute(InputInterface $input, OutputInterface $output)
    {
    $output->writeln('Start user creation');
    $output->writeln($this->createUser($input->getArgument('username')));
    $output->write('End user creation');

    return Command::SUCCESS;
    }
}
```

Structure d'un Projet Symfony

- **config/** : Contient les fichiers de configuration pour les routes, les services et les packages.
- **src/** : L'emplacement de tout votre code PHP.
- **templates/** : Les modèles Twig de votre projet.

La plupart du temps, vous travaillerez dans src/, templates/ ou config/.

- **bin/** : Contient le fichier bin/console (et d'autres fichiers exécutables moins importants).
- **var/** : Stocke les fichiers créés automatiquement (cache, log, etc).
- **vendor/** : Stocke les packages téléchargées ici via Composer.
- **public/** : La racine du site-web, où vous placez les fichiers accessibles publiquement (notamment le front controller `index.php`).

Bundles et Packages

Bundles et Packages

- `Composer` est *de facto* le gestionaire de packages de PHP depuis 2011
- Le repository `packagist.org` contient plus de 385.000 packages

Packages Composer

- **Objectif** : Bibliothèques ou outils PHP généraux gérés par Composer.
- **Fonctionnalités** : Pas nécessairement spécifiques à Symfony ; peuvent inclure diverses fonctionnalités liées à PHP.
- **Intégration** : Composer gère les dépendances ; les packages peuvent ou non s'intégrer aux fonctionnalités de Symfony.
- **Origine** : Typiquement obtenus depuis Packagist ou d'autres dépôts Composer.
- **Enregistrement** : Chargés automatiquement par Composer ; aucun enregistrement explicite dans `AppKernel.php` n'est requis.

Bundles Symfony

- **Objectif** : Spécifiques à Symfony, utilisés pour organiser et empaqueter du code spécifique à Symfony.
- **Fonctionnalités** : Étendent la fonctionnalité de l'application Symfony, encapsulant des fonctionnalités pour la modularisation.
- **Intégration** : Étroitement intégrés à l'écosystème Symfony, utilisant des fonctionnalités telles que le routage et l'injection de dépendances.
- **Origine** : Peuvent être créés sur mesure, officiels de Symfony ou obtenus auprès de sources tierces.
- **Enregistrement** : Doivent être enregistrés dans `AppKernel.php` pour être utilisés dans une application Symfony.

Configuration avec composer.json

- `composer` maintient un fichier `composer.json` avec la liste des packages installés avec leur version :

```
"require": {
    "vendor/package": "1.3.2", // exactly 1.3.2

    // >, <, >=, <= | specify upper / lower bounds
    "vendor/package": ">=1.3.2", // anything above or equal to 1.3.2
    "vendor/package": "<1.3.2", // anything below 1.3.2

    // * | wildcard
    "vendor/package": "1.3.*", // >=1.3.0 <1.4.0

    // ~ | allows last digit specified to go up
    "vendor/package": "~1.3.2", // >=1.3.2 <1.4.0
    "vendor/package": "~1.3", // >=1.3.0 <2.0.0

    // ^ | doesn't allow breaking changes (major version fixed)
    "vendor/package": "^1.3.2", // >=1.3.2 <2.0.0
    "vendor/package": "^0.3.2", // >=0.3.2 <0.4.0 // except if major is 0
}
```

Utilisation de Composer

- Ajout au fichier `composer.json` :

```
$ composer require monolog/monolog [--dev]
```

- Installer les packages du fichier `composer.json` :

```
$ composer install
```

- Mettre à jour vers les dernières versions :

```
$ composer update
```

- Fichier `composer.json` traduit en `composer.lock` :
  - `composer.json` = Contraintes sur les versions
  - `composer.lock` = Versions installées
  - A committer pour assurer la reproductibilité.

Appels dans le cas de Symfony

- Symfony gère les binaires externes :

```
1 $ composer require monolog/monolog
2 # VS
3 $ symfony composer require monolog/monolog
```

- Ne pointe pas forcément sur le même binaire de `composer`
- Utiliser la version de symfony pour reproductibilité
- Idem pour `symfony php`, etc

Démo de création d'un projet

Démo !

- On crée un project vide et on le rempli un peu avant de lancer le serveur.

Jeux de Test

Types de tests

- **Tests unitaires** : test de la classe, méthodes, etc
- **Tests d'intégration** : vérifient que les différents composants utilisés par votre application fonctionnent bien ensemble
- **Tests fonctionnels** : validation du comportement global vis à vis du cahier des charges (user stories, etc)
- **Autres tests** :
  - Tests de robustesse
  - Tests de performance
  - Tests d'ergonomie
  - Tests de sécurité/pénétration
  - Etc

Introduction aux tests dans Symfony

- Symfony s'appuie sur PHPUnit pour les tests unitaires.
- Commençons par installer PHPUnit :

```
1 $ symfony composer require phpunit --dev
```

- Lancer les tests :

```
1 $ symfony php bin/phpunit
```

Configuration de l'environnement de test

- Par défaut, les tests PHPUnit sont exécutés dans l'environnement Symfony test tel qu'il est défini dans le fichier de configuration de `phpunit.xml.dist` :

```
1  <phpunit>
2    <php>
3      <ini name="error_reporting" value="-1" />
4      <server name="APP_ENV" value="test" force="true" />
5      <server name="SHELL_VERBOSITY" value="-1" />
6      <server name="SYMFONY_PHPUNIT_REMOVE" value="" />
7      <server name="SYMFONY_PHPUNIT_VERSION" value="8.5" />
8      ...
```

Écriture de tests unitaires

- Utiles pour tester les services, les entités, les bundles, les modèles, etc
- Inclusion de la classe à tester avec `use` et appels directs aux méthodes avec les paramètres pourlesquels on veut tester :

```php
// tests/SpamCheckerTest.php
use PHPUnit\Framework\TestCase;
use App\Service\SpamChecker;

class SpamCheckerTest extends TestCase
{
  public function testSpamScoreWithInvalidRequest(): void
  {
    $checker = new SpamChecker(...);
    // ... (extrait de code)
    $this->expectException(\RuntimeException::class);
    // ... (extrait de code)
    $this->assertTrue($checker->...);
  }
}
```

Tests d'intégration au niveau des contrôleurs

- **En prinipe** : Même idée que les tests unitaires mais au niveau des contrôleurs
  - Ici on vient tester des fonction un peu plus grosses
- **Problème** : Comment gérer le routage et les paramètres passés automatiquement ?

```php
// tests/ControllerTest.php
use PHPUnit\Framework\TestCase;
use App\Controller\ConferenceController;

class ConferenceControllerTest extends TestCase
{
  public function testController(): void
  {
    $controller = new ConferenceController(...);
    // ... (extrait de code)
    $controller->doit(???);
  }
}
```

Tests d'intégration au niveau des contrôleurs

- Pour gérer les paramètres passés automatiquement, on utilise la classe `KernelTestCase`

```php
// tests/ControllerTest.php
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
use App\Controller\ConferenceController;

class ConferenceControllerTest extends KernelTestCase
{
  public function testController(): void
  {
    self::bootKernel();
    $controller = static::$kernel->getContainer()->get(ConferenceController::class)
    $increadibleParameter = static::$kernel->getContainer()->get(...);
    // ... (extrait de code)
    $controller->doit($increadibleParameter);
  }
}
```

- Tests d'intégration compliqués en Symfony car on se retrouver à récupérer à la main les paramères passés automatiquent hors des tests.

Tests fonctionnels des contrôleurs

- **Morale de l'histoire** : Tester les contrôleurs est un peu différent de tester une classe PHP "ordinaire" car nous voulons les exécuter dans le contexte d'une requête HTTP.
- On utilise la classe `WebTestCase` pour les scénarios sans exécution de JavaScript.

```php
// tests/Controller/ConferenceControllerTest.php
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
use App\Controller\ConferenceController;

class ConferenceControllerTest extends WebTestCase
{
  public function testIndex()
  {
    $client = static::createClient();
    $client->request('GET', '/');
    $this->assertResponseIsSuccessful();
    $this->assertSelectorTextContains('h2', 'Give your feedback')
  }
}
```

Tester la méthode d'un service

- Pour tester la logique d'un service, appelons directement le Manager pour tester sa méthode

```
1  class CommentTest extends KernelTestCase {
2  public function setUp(): void
3  {
4    //Thankfully to KernelTestCase we can getContainer and get Services
5    //If there's some services in CommentManager constructor
6    //gonna be handled by Dependency Injection
7    $this->commentManager = $this->getContainer()->get(CommentManager::class);
8    parent::setUp();
9  }
10 public function testCommentSubmission()
11 {
12   $isValid = $this->commentManager
13                   ->checkCommentIntegrity('Je suis un commentaire intègre');
14   //Assert that first value (expected value)
15   //is equal to computed value by service
16   $this-assertEquals(true, $isValid);
17 }
18 }
```

Soumettre un formulaire dans les tests

```php
public function testCommentSubmission()
{
  $client = static::createClient();
  $client->request('GET', '/conference/amsterdam-2019');
  $client->submitForm('Submit', [
      'comment_form[author]' => 'Fabien',
      'comment_form[text]' => 'Some feedback from an automated functional test',
      'comment_form[email]' => 'me@automat.ed',
      'comment_form[photo]' => dirname(__DIR__, 2).'/public/images/under-constructi
  ]);
  $this->assertResponseRedirects();
  $client->followRedirect();
  $this->assertSelectorExists('div:contains("There are 2 comments")');
}
```

Tests fonctionnels avec Panther

- Pour les tests de bout en bout (e2e) avec un vrai navigateur
  `PantherTestCase`
- Pas présent pour le moment sur les machines de l'IUT donc pas de JS dans les tests !
- `PantherTestCase` étand `WebTestCase` :

```php
// tests/Controller/ConferenceControllerTest.php
use Symfony\Component\Panther\PantherTestCase;

class ConferenceControllerTest extends PantherTestCase
{
  public function testIndex()
  {
    $symfonyClient = static::createClient(); // Le client de WebTestCase
    $pantherClient = static::createPantherClient(); // Via Panther
    // Et avec choix du navigateur!
    $firefoxClient = static::createPantherClient(['browser' => static::FIREFOX]);
  }
}
```

En résumé

- `TestCase` : Tests PHPUnit basiques ;
- `KernelTestCase` : Tests basiques ayant accès aux services Symfony ;
- `WebTestCase` : Pour exécuter des scénarios à la manière d'un navigateur, mais sans exécution du code JavaScript ;
- `ApiTestCase` : Pour jouer des scénarios orientés API ;
- `PantherTestCase` : Pour jouer des scénarios e2e, en utilisant un vrai navigateur ou client HTTP et un vrai serveur web.