

Symfony

Usages et principes

Maxime Puys

29 janvier 2024

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
- ▷ Organisation claire pour quelqu'un qui connaît 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 tâche 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

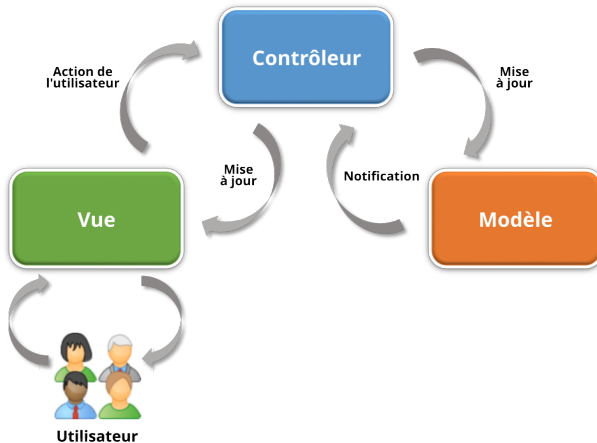


Figure – MVC - Wikipedia

Rappels ORM

□ Sans requête préparée :

```
1 $requete = "SELECT * FROM ma_table WHERE mon_champ = ma_valeur";  
2 $resultats = $connexion->query($requete);
```

□ Avec requête préparée :

```
1 $requete = "SELECT mon_champ FROM ma_table WHERE mon_champ = :ma_valeur";  
2 $resultats = $connexion->prepare($requete);  
3 $resultats->execute(array(":ma_valeur" => 5));
```

□ Avec un ORM :

```
1 $user = new User();  
2 $user->name = 'John';  
3 $user->password = 'Doe';  
4 $user->save();
```


Rappels ORM

Relational database (such as PostgreSQL or MySQL)

ID	FIRST_NAME	LAST_NAME	PHONE
1	John	Connor	+16105551234
2	Matt	Makai	+12025555689
3	Sarah	Smith	+19735554512
...	...	...	...

Python objects

```
class Person:  
    first_name = "John"  
    last_name = "Connor"  
    phone_number = "+16105551234"
```

```
class Person:  
    first_name = "Matt"  
    last_name = "Makai"  
    phone_number = "+12025555689"
```

```
class Person:  
    first_name = "Sarah"  
    last_name = "Smith"  
    phone_number = "+19735554512"
```

ORMs provide a bridge between
**relational database tables, relationships
and fields and Python objects**

Figure – ORM - Full Stack Python

Introduction à Symfony

Bible de Symfony

□ A lire et relire :

- ▷ <https://symfony.com/doc/current/index.html>
- ▷ <https://symfony.com/doc/6.2/the-fast-track/en/index.html>

Symfony vs. PHP seul

Front controller sans Symfony :

```
1 // index.php
2
3 // load and initialize any global libraries
4 require_once 'model.php';
5 require_once 'controllers.php';
6
7 // route the request internally (or use router package e.g. AltoRouter)
8 $uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
9 if ('/index.php' === $uri) {
10     list_action();
11 } elseif ('/index.php/show' === $uri && isset($_GET['id'])) {
12     show_action($_GET['id']);
13 } else {
14     header('HTTP/1.1 404 Not Found');
15     echo '<html><body><h1>Page Not Found</h1></body></html>';
16 }
```

Symfony vs. PHP seul

Modèle sans Symfony :

```
1 // model.php
2 function get_post_by_id($id)
3 {
4     $connection = open_database_connection();
5
6     $query = 'SELECT created_at, title, body FROM post WHERE id=:id';
7     $statement = $connection->prepare($query);
8     $statement->bindValue(':id', $id, PDO::PARAM_INT);
9     $statement->execute();
10
11     $row = $statement->fetch(PDO::FETCH_ASSOC);
12
13     close_database_connection($connection);
14
15     return $row;
16 }
```

Symfony vs. PHP seul

Controller sans Symfony :

```
1 // controllers.php
2 function list_action()
3 {
4     $posts = get_all_posts();
5     require 'templates/list.php';
6 }
7
8 function show_action($id)
9 {
10     $post = get_post_by_id($id);
11     require 'templates/show.php';
12 }
```

Symfony vs. PHP seul

Vue sans Symfony :

```
1 <!-- templates/show.php -->
2 <?php $title = $post['title'] ?>
3
4 <?php ob_start() ?>
5     <h1><?= $post['title'] ?></h1>
6
7     <div class="date"><?= $post['created_at'] ?></div>
8     <div class="body">
9         <?= $post['body'] ?>
10     </div>
11 <?php $content = ob_get_clean() ?>
12
13 <?php include 'layout.php' ?>
```

Symfony vs. PHP seul

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

```
1 # config/routes.yaml
2 blog_list:
3     path:      /blog
4     controller: App\Controller\BlogController::list
5
6 blog_show:
7     path:      /blog/show/{id}
8     controller: App\Controller\BlogController::show
```

```
1 // public/index.php
2 require_once __DIR__.'../../app/bootstrap.php';
3 require_once __DIR__.'../../src/Kernel.php';
4
5 use Symfony\Component\HttpFoundation\Request;
6
7 $kernel = new Kernel('prod', false);
8 $kernel->handle(Request::createFromGlobals())->send();
```


Symfony vs. PHP seul

Modèle avec Symfony (généré automatiquement) :

```
1 namespace App\Entity;
2 use App\Repository\BlogPostRepository;
3 use Doctrine\ORM\Mapping as ORM;
4 #[ORM\Entity(repositoryClass: BlogPostRepository::class)]
5 class BlogPost
6 {
7     #[ORM\Id]
8     #[ORM\GeneratedValue]
9     #[ORM\Column]
10    private ?int $id = null;
11
12    #[ORM\Column]
13    private ?\DateTimeImmutable $created_at = null;
14    #[ORM\Column(length: 255)]
15    private ?string $title = null;
16
17    #[ORM\Column(length: 255)]
18    private ?string $body = null;
19
20    // Getters and Setters
21 }
```

Symfony vs. PHP seul

Modèle avec Symfony (généré automatiquement) :

```

1 namespace App\Repository;
2 use App\Entity\BlogPost;
3 use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
4 use Doctrine\Persistence\ManagerRegistry;

5 /**
6  * @extends ServiceEntityRepository<BlogPost>
7  *
8  * @method BlogPost|null find($id, $lockMode = null, $lockVersion = null)
9  * @method BlogPost|null findOneBy(array $criteria, array $orderBy = null)
10 * @method BlogPost[]    findAll()
11 * @method BlogPost[]    findBy(array $criteria, array $orderBy = null, $limit = null)
12 */
13 class BlogPostRepository extends ServiceEntityRepository
14 {
15     public function __construct(ManagerRegistry $registry)
16     {
17         parent::__construct($registry, BlogPost::class);
18     }
19 }

```

Symfony vs. PHP seul

Controller avec Symfony (en partie généré automatiquement) :

```
1 // src/Controller/BlogController.php
2 namespace App\Controller;
3 use App\Entity\Post;
4 use Doctrine\Persistence\ManagerRegistry;
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6
6 class BlogController extends AbstractController {
7     public function list(ManagerRegistry $doctrine) {
8         $posts = $doctrine->getRepository(Post::class)->findAll();
9         return $this->render('blog/list.html.twig', ['posts' => $posts]);
0     }
1
1     public function show(ManagerRegistry $doctrine, $id) {
2         $post = $doctrine->getRepository(Post::class)->find($id);
3         if (!$post) {
4             throw $this->createNotFoundException();
5         }
6         return $this->render('blog/show.html.twig', ['post' => $post]);
7     }
8 }
```

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 }) } }">  
10             { { post.title } }  
11         </a>  
12     </li>  
13     {% endfor %}  
14 </ul>  
15 {% endblock %}
```

Gestion des Requêtes/Réponses avec PHP

```
1 $uri = $_SERVER['REQUEST_URI'];  
2 $foo = $_GET['foo'];  
  
3 header('Content-Type: text/html');  
4 echo 'The URI requested is: '.$uri;  
5 echo 'The value of the "foo" parameter is: '.$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

```
1 HTTP/1.1 200 OK  
2 Date: Sat, 03 Apr 2011 02:14:33 GMT  
3 Server: Apache/2.2.17 (Unix)  
4 Content-Type: text/html  
  
5 The URI requested is: /testing?foo=symfony  
6 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).

```
1 use Symfony\Component\HttpFoundation\Request;
2
3 $request = Request::createFromGlobals();
4
5 // Get the URI being requested (e.g., /about) minus any query parameters
6 $request->getPathInfo();
7
8 // Retrieve $_GET and $_POST variables respectively
9 $request->query->get('id');
10 $request->request->get('category', 'default category');
11
12 // Retrieve $_SERVER variables
13 $request->server->get('HTTP_HOST');
14
15 // Retrieve an instance of UploadedFile identified by "attachment"
16 $request->files->get('attachment');
```

Gestion des Requêtes/Réponses avec Symfony

```
1 // Retrieve a $_COOKIE value
2 $request->cookies->get('PHPSESSID');

3 // Retrieve an HTTP request header, with normalized, lowercase keys
4 $request->headers->get('host');
5 $request->headers->get('content-type');

6 $request->getMethod(); // e.g., GET, POST, PUT, DELETE, or HEAD
7 $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

```
1 use Symfony\Component\HttpFoundation\Response;
2
3 $response = new Response();
4
5 $response->setContent('<html><body><h1>Hello world!</h1></body></html>');
6 $response->setStatusCode(Response::HTTP_OK);
7
8 // Set a HTTP response header
9 $response->headers->set('Content-Type', 'text/html');
10
11 // Print the HTTP headers followed by the content
12 $response->send();
```


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.html`).

Bundles et Packages

Bundles et Packages

- `Composer` est *de facto* le gestionnaire 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 emballer le 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 :

```

1 "require": {
2     "vendor/package": "1.3.2", // exactly 1.3.2

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

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

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

11    // ^ | doesn't allow breaking changes (major version fixed)
12    "vendor/package": "^1.3.2", // >=1.3.2 <2.0.0
13    "vendor/package": "^0.3.2", // >=0.3.2 <0.4.0 // except if major is 0
14 }

```

Utilisation de Composer

- Ajout au fichier `composer.json` :

```
1 $ composer require monolog/monolog [--dev]
```

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

```
1 $ composer install
```

- Mettre à jour vers les dernières versions :

```
1 $ 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

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 pour lesquels on veut tester :

```
1 // tests/SpamCheckerTest.php
2 use PHPUnit\Framework\TestCase;
3 use App\Service\SpamChecker;
4
5 class SpamCheckerTest extends TestCase
6 {
7     public function testSpamScoreWithInvalidRequest(): void
8     {
9         $checker = new SpamChecker(...);
10        // ... (extrait de code)
11        $this->expectException(\RuntimeException::class);
12        // ... (extrait de code)
13        $this->assertTrue($checker->...);
14    }
15 }
```

Tests d'intégration au niveau des contrôleurs

- ❑ **En principe** : 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 ?

```
1 // tests/ControllerTest.php
2 use PHPUnit\Framework\TestCase;
3 use App\Controller\ConferenceController;
4
5 class ConferenceControllerTest extends TestCase
6 {
7     public function testController(): void
8     {
9         $controller = new ConferenceController(...);
10        // ... (extrait de code)
11        $controller->doit(???);
12    }
13 }
```

Tests d'intégration au niveau des contrôleurs

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

```
1 // tests/ControllerTest.php
2 use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
3 use App\Controller\ConferenceController;
4
5 class ConferenceControllerTest extends KernelTestCase
6 {
7     public function testController(): void
8     {
9         self::bootKernel();
10        $controller = static::$kernel->getContainer()->get(ConferenceController::class);
11        $incredibleParameter = static::$kernel->getContainer()->get(...);
12        // ... (extrait de code)
13        $controller->doit($incredibleParameter);
14    }
15 }
```

- Tests d'intégration compliqués en Symfony car on se retrouve à récupérer à la main les paramètres passés automatiquement 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.

```
1 // tests/Controller/ConferenceControllerTest.php
2 use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
3 use App\Controller\ConferenceController;

4 class ConferenceControllerTest extends WebTestCase
5 {
6     public function testIndex()
7     {
8         $client = static::createClient();
9         $client->request('GET', '/');
10        $this->assertResponseIsSuccessful();
11        $this->assertSelectorTextContains('h2', 'Give your feedback')
12    }
13 }
```


Soumettre un formulaire dans les tests

```
1 public function testCommentSubmission()
2 {
3     $client = static::createClient();
4     $client->request('GET', '/conference/amsterdam-2019');
5     $client->submitForm('Submit', [
6         'comment_form[author]' => 'Fabien',
7         'comment_form[text]' => 'Some feedback from an automated functional test',
8         'comment_form[email]' => 'me@automat.ed',
9         'comment_form[photo]' => dirname(__DIR__, 2).'/public/images/under-construction.jpg'
10    ]);
11     $this->assertResponseRedirects();
12     $client->followRedirect();
13     $this->assertSelectorExists('div:contains("There are 2 comments")');
14 }
```

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 étend WebTestCase :

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

3 class ConferenceControllerTest extends PantherTestCase
4 {
5     public function testIndex()
6     {
7         $symfonyClient = static::createClient(); // Le client de WebTestCase
8         $pantherClient = static::createPantherClient(); // Via Panther
9         // Et avec choix du navigateur!
0         $firefoxClient = static::createPantherClient(['browser' => static::FIREFOX]);
1     }
2 }
```

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.