Projet Symfony

Travaux Pratiques

Timothée Robert

Table of Contents

Partie 1 : Installation et Configuration

Installation des outils

Exercice 0 : Installation de Scoop (Prérequis Windows)

Scoop est un gestionnaire de paquets en ligne de commande pour Windows, très pratique pour installer des outils de développement.

Ouvrez un terminal **PowerShell** et exécutez la commande suivante pour autoriser l'exécution de scripts :

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Ensuite, lancez l'installation de Scoop avec cette commande :

```
irm get.scoop.sh | iex
```

Pour vérifier que l'installation a réussi, fermez et rouvrez votre terminal, puis tapez `scoop help`.

Exercice 1 : Installation de la CLI Symfony

Installez la CLI Symfony en utilisant Scoop.

```
scoop install symfony-cli
```

Une fois l'installation terminée, vérifiez son succès avec la commande suivante :

```
symfony --help
```

Enfin, lancez le vérificateur de configuration de Symfony pour identifier les optimisations possibles sur votre environnement PHP (OPcache, limites mémoire, etc.).

```
symfony check
```

Création et lancement du projet

Exercice 2 : Création d'un nouveau projet

Créez un nouveau projet Symfony nommé `vente-materiel`. Nous utilisons ici la convention `kebab-case` (minuscules séparées par des tirets), qui est la meilleure pratique pour les noms de projets.

```
symfony new vente-materiel
```

Le tableau ci-dessous résume les conventions de nommage :

Convention	Exemple	Usage typique	Recommandé pour un projet ?
`kebab-case`	`vente-materiel`	Noms de projet/dépôt, URL	**Oui, le meilleur choix**
`snake_case`	`vente_materiel`	Noms de bases de données/tables	**Oui, très bonne alternative**
`camelCase`	`venteMateriel`	Noms de variables/fonctions	**Non, à éviter**
`PascalCase`	`VenteMateriel`	Noms de Classes en PHP	**Non, à réserver au code**

Exercice 3 : Lancement du serveur local

Placez-vous dans le répertoire du projet et lancez le serveur web local.

```
cd vente-materiel
```

Pour une navigation sécurisée en HTTPS, installez le certificat local de Symfony :

```
symfony server:ca:install
```

Démarrez ensuite le serveur :

```
symfony serve
```

Enfin, vérifiez que le serveur fonctionne en ouvrant votre navigateur à l'adresse https://127.0.0.1:8000.

Partie 2 : Templating avec Twig

Installation et utilisation de Twig

Exercice 4 : Installation et création d'un template

Installez le moteur de template Twig via Composer.

```
composer require twig
```

Ensuite, créez un répertoire `templates/main/` et à l'intérieur, un fichier `accueil.html.twig`.

Modifiez votre `MainController` pour qu'il utilise la méthode `$this->render()` afin d'afficher ce nouveau template.

```
<?php
// src/Controller/MainController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        return $this->render('main/accueil.html.twig');
    }
}
```

Exercice 5 : Passage de données au template

Modifiez le contrôleur pour passer un tableau de données au template Twig.

```
// ...
public function homepage(): Response
{
    $nombreProduits = 42; // Exemple de données à passer
    return $this->render('main/accueil.html.twig', [
        'nombreProduits' => $nombreProduits
    ]);
}
// ...
```

Utilisez l'héritage de template en ajoutant `{% extends 'base.html.twig' %}` au début de votre fichier `accueil.html.twig`. Utilisez les blocs `{% block title %}` et `{% block body %}` pour surcharger le contenu de la page parente. C'est le même principe que la surcharge de méthodes en POO.

Partie 3 : Débogage

Outils de débogage

Exercice 6 : Installation et utilisation de la Debug Toolbar

Installez le pack de débogage de Symfony.

```
composer require debug --dev
```

Cet outil ajoute une barre de débogage très complète en bas de page dans votre navigateur, permettant d'inspecter les requêtes, les routes, les performances, etc.

Pour déboguer en ligne de commande, vous pouvez lister toutes les routes ou les templates de votre application.

```
# Lister les routes
php bin/console debug:router

# Lister les templates Twig
php bin/console debug:twig
```

Partie 4 : Création d’une API REST

Mise en place du contrôleur d'API

Exercice 7 : Création du contrôleur et de la première route

Créez un nouveau contrôleur pour votre API.

```
symfony console make:controller ProduitApiController
```

Dans ce nouveau contrôleur, ajoutez une méthode `getListe()` qui retournera une liste de produits au format JSON. Définissez sa route avec l'attribut `#[Route]`.

```
<?php
// src/Controller/ProduitApiController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ProduitApiController extends AbstractController
{
    #[Route('/api/produits')]
    public function getListe(): Response
    {
        $produits = [
            ['id' => 1, 'nom' => 'Stylo feutre noir', 'prix' => 2.00],
            ['id' => 2, 'nom' => 'Trousse SIO', 'prix' => 4.00],
            ['id' => 3, 'nom' => 'Cahier SLAM', 'prix' => 2.00],
        ];

        return $this->json($produits);
    }
}
```

Testez votre API en accédant à l'URL https://127.0.0.1:8000/api/produits dans votre navigateur.

Exercice 8 : Refactoring avec des objets

Créez un répertoire `src/Model` et à l'intérieur, une classe `Produit.php` avec un constructeur et des getters.

```
<?php
// src/Model/Produit.php
namespace App\Model;

class Produit
{
    public function __construct(
        private int $id,
```

```
        private string $nom,
        private float $prix
    ) {}

    public function getId(): int { return $this->id; }
    public function getNom(): string { return $this->nom; }
    public function getPrix(): float { return $this->prix; }
}
```

Modifiez ensuite votre contrôleur pour utiliser cette nouvelle classe.

```
// ... dans ProduitApiController.php
use App\Model\Produit;
// ...
public function getListe(): Response
{
    $produits = [
        new Produit(1, 'Stylo feutre noir', 2.00),
        new Produit(2, 'Trousse SIO', 4.00),
        new Produit(3, 'Cahier SLAM', 2.00)
    ];
    return $this->json($produits);
}
// ...
```

En rafraîchissant la page, vous constaterez que la réponse est vide ! Pour corriger cela, installez le composant Serializer.

```
composer require serializer
```

Rafraîchissez à nouveau la page. Le flux JSON devrait maintenant être correctement formaté.

NOTE

Exercice 9 : Analyse du composant Serializer

Pourquoi le flux JSON est-il maintenant bien formaté, alors qu'il était vide auparavant ? Comment le composant Serializer de Symfony a-t-il permis de résoudre ce problème ?

Partie 5 : Les Services et l’Injection de Dépendances

Utilisation des services existants

Exercice 10 : Utilisation du service de logging

Pour voir la liste de tous les services disponibles dans l'application, utilisez la commande suivante :

```
php bin/console debug:container
```

Nous allons maintenant utiliser le service de logging. Pour ce faire, injectez `LoggerInterface` en tant que paramètre de votre méthode `getListe()`.

```
// ...
use Psr\Log\LoggerInterface;

public function getListe(LoggerInterface $monLog): Response
{
    // ...
}
```

Pour inspecter l'objet injecté, utilisez la fonction `dd()` (dump and die) de Symfony.

```
public function getListe(LoggerInterface $monLog): Response
{
    dd($monLog);
    // ...
}
```

NOTE

Exercice 11 : Analyse de la fonction dd()

Expliquez ce que fait la fonction `dd()` dans le contexte de Symfony et comment elle aide les développeurs à déboguer leur code.

Exercice 12 : Enregistrement d'un log

Remplacez `dd($monLog);` par un appel à la méthode `info()` pour enregistrer un message.

```
public function getListe(LoggerInterface $monLog): Response
{
    $monLog->info('Liste des produits demandée');
    // ...
}
```

Vérifiez que le message apparaît bien dans le fichier `var/log/dev.log` ou dans l'onglet "Logs" du Profiler Symfony.

Création d'un service personnalisé : le Repository

Exercice 13 : Refactoring avec un Repository

Créez un répertoire `src/Repository` et à l'intérieur une classe `ProduitRepository.php`. Cette classe aura pour rôle de centraliser l'accès aux données des produits.

Ajoutez une méthode `findAll()` dans ce nouveau Repository, qui retournera le tableau de produits que nous avions dans le contrôleur.

```
<?php
// src/Repository/ProduitRepository.php
namespace App\Repository;

use App\Model\Produit;
use Psr\Log\LoggerInterface;

class ProduitRepository
{
    public function __construct(private LoggerInterface $logger) {}

    public function findAll(): array
    {
        $this->logger->info('Récupération de la liste de tous les produits.');
        return [
            new Produit(1, 'Stylo feutre noir', 2.00),
            new Produit(2, 'Trousse SIO', 4.00),
            new Produit(3, 'Cahier SLAM', 2.00)
        ];
    }
}
```

Modifiez ensuite le contrôleur `ProduitApiController` pour qu'il utilise ce Repository via l'injection de dépendances.

```
// ... dans ProduitApiController.php
use App\Repository\ProduitRepository;

#[Route('/api/produits')]
public function getListe(ProduitRepository $repository): Response
{
    $produits = $repository->findAll();
    return $this->json($produits);
}
```

NOTE *Exercice 14 : Analyse de l'approche Repository*

Effectuez le refactoring. Quel est l'intérêt de cette approche ? Pourquoi est-il préférable d'utiliser un repository pour gérer les données plutôt que de les manipuler directement dans le contrôleur ?

Exercice 15 : Application sur la page d'accueil

Modifiez le `MainController` pour utiliser le `ProduitRepository`, récupérer la liste des produits et passer le **nombre** de produits au template.

Ensuite, modifiez le template `accueil.html.twig` pour afficher un produit choisi au hasard dans la liste.

Comment peut-on afficher un produit au hasard en utilisant Twig ? Essayez de le faire sans regarder les indications.

NOTE

Astuce 1 : Passez le tableau complet des produits au template, en plus du nombre. **Astuce 2 :** Utilisez la fonction `random()` de Twig pour sélectionner un produit aléatoire dans le tableau. **Astuce 3 :** Affichez son nom avec `{{ produitAleatoire.nom }}`.

Ajout d’une route pour un produit spécifique

Exercice 16 : Création de la méthode find() dans le Repository

Ajoutez une nouvelle route dans `ProduitApiController` pour afficher un seul produit via son ID.

```
// ... dans ProduitApiController.php

#[Route('/api/produits/{id<\d+>}', methods: ['GET'])]
public function get(int $id, ProduitRepository $repository): Response
{
    $produit = $repository->find($id);

    if (!$produit) {
        throw $this->createNotFoundException('Produit non trouvé');
    }

    return $this->json($produit);
}
```

Implémentez maintenant la méthode `find(int $id)` dans votre `ProduitRepository`. Elle doit parcourir la liste des produits et retourner le produit correspondant à l’ID, ou `null` s’il n’est pas trouvé.

NOTE	La signature de la méthode doit être `public function find(int $id): ?Produit`. À quoi sert la méthode `createNotFoundException` dans ce contexte ? Quel est son rôle dans la gestion des erreurs ?

Partie 6 : Contrôleur et Template HTML

Affichage d'une page produit

Exercice 17 : Création d'un contrôleur et d'une route HTML

Un même Repository peut servir à la fois une API et des pages HTML. Créez un nouveau `ProduitController.php` (sans le "Api") qui étendra `AbstractController`.

Ajoutez une méthode `show(int $id)` avec la route `/produits/{id}`.

```
<?php
// src/Controller/ProduitController.php
namespace App\Controller;

use App\Repository\ProduitRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ProduitController extends AbstractController
{
    #[Route('/produits/{id<\d+>}', name: 'app_produit_show')]
    public function show(int $id, ProduitRepository $repository): Response
    {
        $produit = $repository->find($id);

        if (!$produit) {
            throw $this->createNotFoundException('Produit non trouvé');
        }

        return $this->render('produit/show.html.twig', [
            'produit' => $produit
        ]);
    }
}
```

Créez le template `templates/produit/show.html.twig` associé. PhpStorm peut vous aider à le créer directement (Alt+Entrée sur le nom du template).

```
{% extends 'base.html.twig' %}

{% block title %}{{ produit.nom }}{% endblock %}

{% block body %}
    <h1>{{ produit.nom }}</h1>
    <p>Prix : {{ produit.prix }} €</p>
{% endblock %}
```

Vérifiez que l'accès à la page fonctionne pour différents produits (ex: `/produits/1`).

Exercice 18 : Création d'un lien vers la page produit

Maintenant que notre route `app_produit_show` est nommée, nous pouvons l'utiliser pour créer des liens.

Dans votre template d'accueil (`accueil.html.twig`), modifiez le code qui affiche un produit au hasard pour qu'il affiche un lien cliquable vers la page de détail de ce produit.

Le code à utiliser est le suivant :

```
<a href="{{ path('app_produit_show', {id: produitDuJour.id}) }}">
    {{ produitDuJour.nom }}
</a>
```