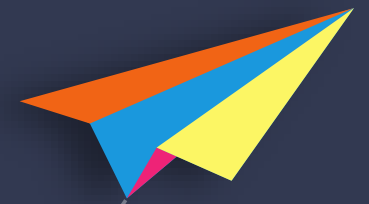
Symfony 4

Pr. Bouchra Honnit



28/04/2020

Programme

1	**Introduction**	C’est quoi un framework Symfony , architecture de symfony.
2	**Installation**	Préparation d’environnement de développement, créer votre premier projet.
3	**Architecture**	Modèle MVC, Créer une première page web
4	**Routage**	Créer, configurer les routes
5	**Contrôleurs**	Créer, configurer une classe et une fonction contrôleur
6	**Template**	Manipulation des templates
7	**formulaire**	Manipulation des formulaires

INTRODUCTION

Chapitre 1

Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique

- Un Framework est un ensemble d’outil applicatif structurant qui répond aux problèmes rencontrés le plus souvent par les développeurs.
- Il possède généralement des fonctionnalités prêt à l’emploi qui permet de structurer les projets informatiques et de réduire le temps de son développements.
- Avantages :
  - Meilleure organisation du code
  - Découpage logique du code source
  - Factorisation de composants communs, réutilisabilité du code
  - Meilleure maintenabilité et évolutivité

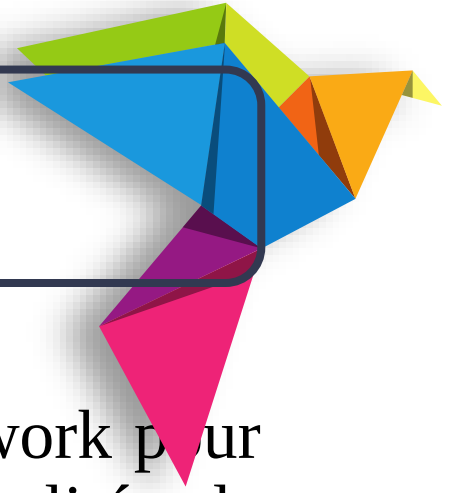
Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique

- Framework php coté serveur, qui permet:
  - structuration du code (MVC)
  - simplification du développement
  - nombreux modules existants (bibliothèque ou bundles)
- Parmi ses avantages:
  - routage facile et url propres (via annotations)
  - contrôleurs : PHP et objet
  - manipulation des bases de données : Doctrine
  - langage de templates : Twig
  - gestion de formulaires facilitée

Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique



- En 2005, la startup SensioLabs a voulu de développer un Framework pour ses propres besoin pour éviter de récréer les mêmes fonctionnalités de gestion des utilisateurs, de base des données, etc plusieurs fois. ➔ D’où l’introduction de la première version de symfony
- Les problématiques auxquels répond le Framework sont les mêmes rencontrés par les autres développeurs, C’est pour cela le code a été partagé par la suite avec la communauté des développeurs de PHP.
- Le projet prend le nom de Symfony suite à la volonté de Fabien Potencier (le créateur du framework) de conserver les initiales S et F de Sensio Framework.

Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique

Les fonctionnalités de la **première version** :

- Une séparation en trois couches selon le modèle MVC ➔meilleure maintenabilité et évolutivité.
- Des performances optimisées et un système de cache pour assurer un temps de réponse optimal.
- Une gestion des URL parlante, permettant à une page d'avoir une URL distincte de sa position dans l'arborescence.
- Un système de configuration en cascade utilisant le langage YAML.
- Couche de mapping ORM, Ajax,
- …

Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique

- La **2eme version** à été publié le 28 juillet 2011.
- Cette version n’est pas compatible avec la première.
- Les fonctionnalités de la **deuxième version** :
  - support pour des librairies telles qu'Assetic, Twig, Imagine et Monolog ;
  - des injections de dépendance et introduction de la notion des bundle;
  - Les ESI (Edge Side Include), le cache HTTP ou encore le support d'un reverse proxy comme Varnish;
  - une barre de débug plus complète que la précédente.

Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique

- La **3eme version** à été publié le 30 novembre 2015.
- Il a été adapté pour fonctionner avec une version minimal de php 5.5.9.
- Compatibilité entre la 2 et la 3 version

Introduction

- C’est quoi un Framework Php ?
- C’est quoi un Framework Symfony ?
- Historique

- La **4eme version** à été publié le 30 novembre 2015.
- N’est pas compatible avec la version 3;
- Il existe une méthode à suivre pour convertir un projet symfony de la version 3 à la version 4;
- représente une refonte complète des versions antérieures
- Modifications:
  - Changement de l’architecture;
  - Élimination de la notion des bundles.

Chapitre 2

Symfony 4

INSTALLATION

Installation

- Exigences Techniques & plateforme
- Création d'un premier projet en Symfony
- Configuration d'une Base de donnée

Exigences technique:

- PHP 7.2.5 ou plus : installer wampserser ou easy php ou tout autre plateforme de développement web
- Composer pour l'installation des packages PHP

  (Lien : https://getcomposer.org/download/)

- **Optionnel** *(recommander)* : installer le Symfony Cli, pour activer la commande symfony. Il offre une commande qui vérifie si votre ordinateur répond à toutes les exigences :

  symfony check:requirements

  Lien: https://symfony.com/download

Installation

- Exigences Techniques & plateforme
- Création d’un premier projet en Symfony
- Configuration d’une Base de donnée

Environnement de développement :

- Invite de commande
- Un éditeur de texte : codeStudio, notepade, ou autre

Installation

- Exigences Techniques & plateforme
- Création d'un premier projet en Symfony
- Configuration d'une Base de donnée

Création:

- Dans l'invite de commande :

1- accéder au répertoire web www (wamp, easyPhp) ou htdocs : utiliser la commande cd

2- Créer un projet nommé : PrjtSymfony ou un nom de votre choix

symfony new PrjtSymfony –full

Ou

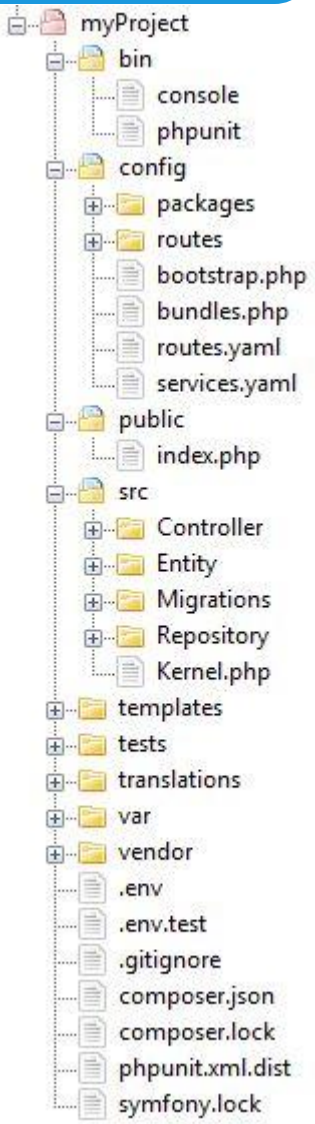
composer create-project symfony/website-skeleton PrjtSymfony

➔Voilà on a crée un nouveau projet symfony dans le répèrtoire

Installation

- Exigences Techniques & plateforme
- Création d'un premier projet en Symfony
- Configuration d'une Base de donnée

- **bin :** contient le fichier relatif à la console et aux tests unitaires
- **config:** contient les fichiers de configurations des routes et des librairies
- **public :** contient les assets (image, css, …) ainsi que le .htaccess
- **src :** contient les entités (classes), les contrôleurs, les formulaires…
- **templates :** contient les templates de l'application (page web)
- **translations :** contient les fichiers de traduction
- **var :** contient les différents fichiers de caches et de logs
- **vendor :** contient les différentes librairies installées / utilisées
- **.env :** Fichier contenant la configuration générale de l'application (exemple : données d'accès au serveur mail, à la base des données, etc)



Création de base des données:

Tout au long de ce cours, on va travailler avec la base de données, représentée par le schème relationnel suivante:

Client(idCli, nom, prenom, type)
Commande(idCom, dateCom, idCli➔Client)
Produit(idPdt, libele, prix, qteStock)
ComPdt(idCom➔Commande, idPrdt➔produit, qteCom)

➔Vous pouvez utiliser le fichier myprojectDB.sql pour créer la base des données.

Installation

Configurer l'accès à la base de données :

1. Accéder au fichier : .env dans la racine de votre projet
2. Dans ce fichier, vous allez trouver la ligne suivante

DATABASE_URL=mysql://db_user:db_password/db_name?serverVersion=5.7@127.0.0.1:3306

- **mysql:** ➔mysql:3306 (Db provider(postgreSql, oracle) + le port d'accès à la base de données
- **Db_user**➔root (le nom d'utilisateur qui peut accèder à la BD)
- **db_password**➔mot de passe d'accès q'il y a un, sinon laisser le vide
- **db_name:** myprojectdb (nom de la base des données).

Installation

- Exigences Techniques & plateforme
- Création d'un premier projet en Symfony
- Configuration d'une Base de donnée

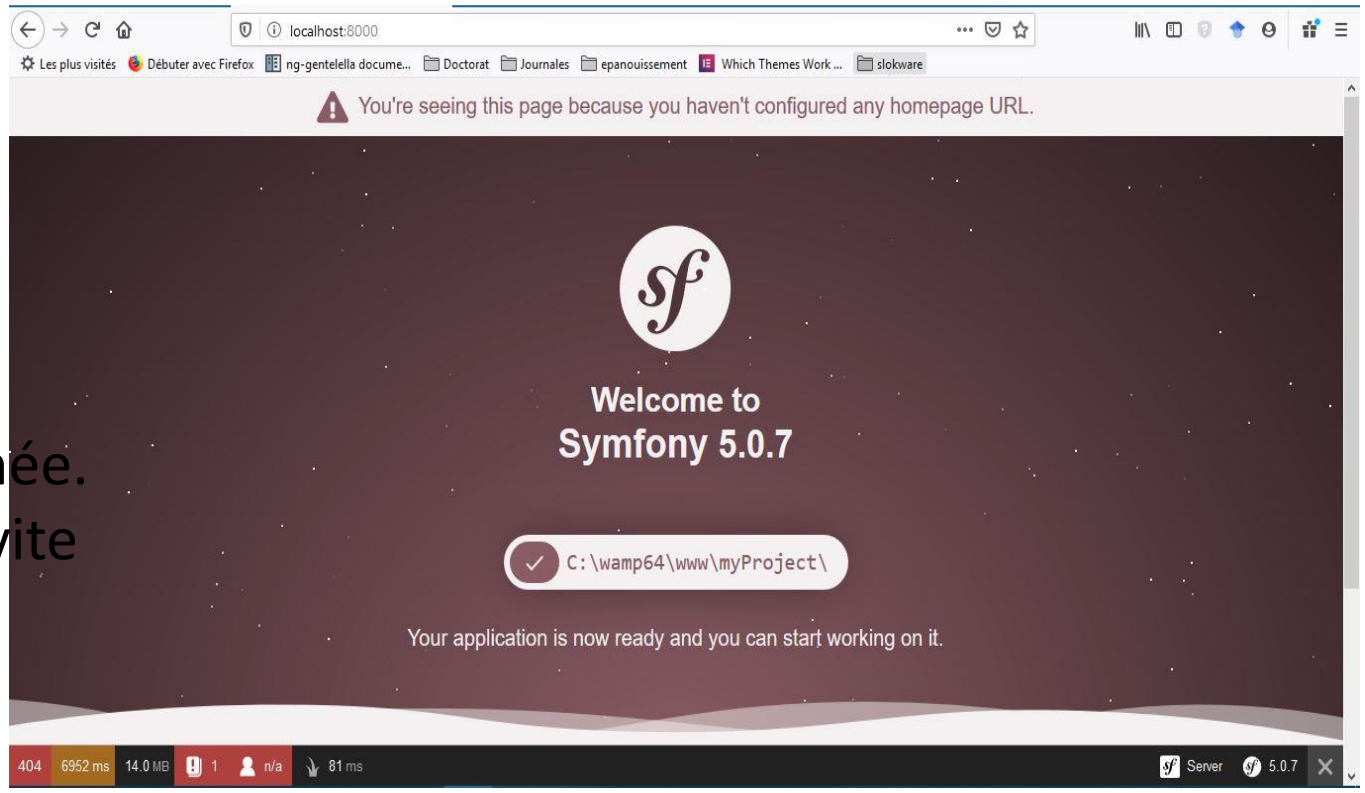
Exécuter votre projet:

- Méthode 1 : dans l'invite de commande

1. Accéder à votre projet en utilisant la commande cd. Exemple si je suis dans la racine C:\ : cd wamp64/www/myproject
2. Démarrer le serveur par la commande : symfony server:start
3. Dans votre navigateur accéder à : http://localhost:8000/

Une page d'accueil symfony sera affichée.

NB. : pour arrêter le serveur, dans l'invite de commande vous cliquez ctrl+c



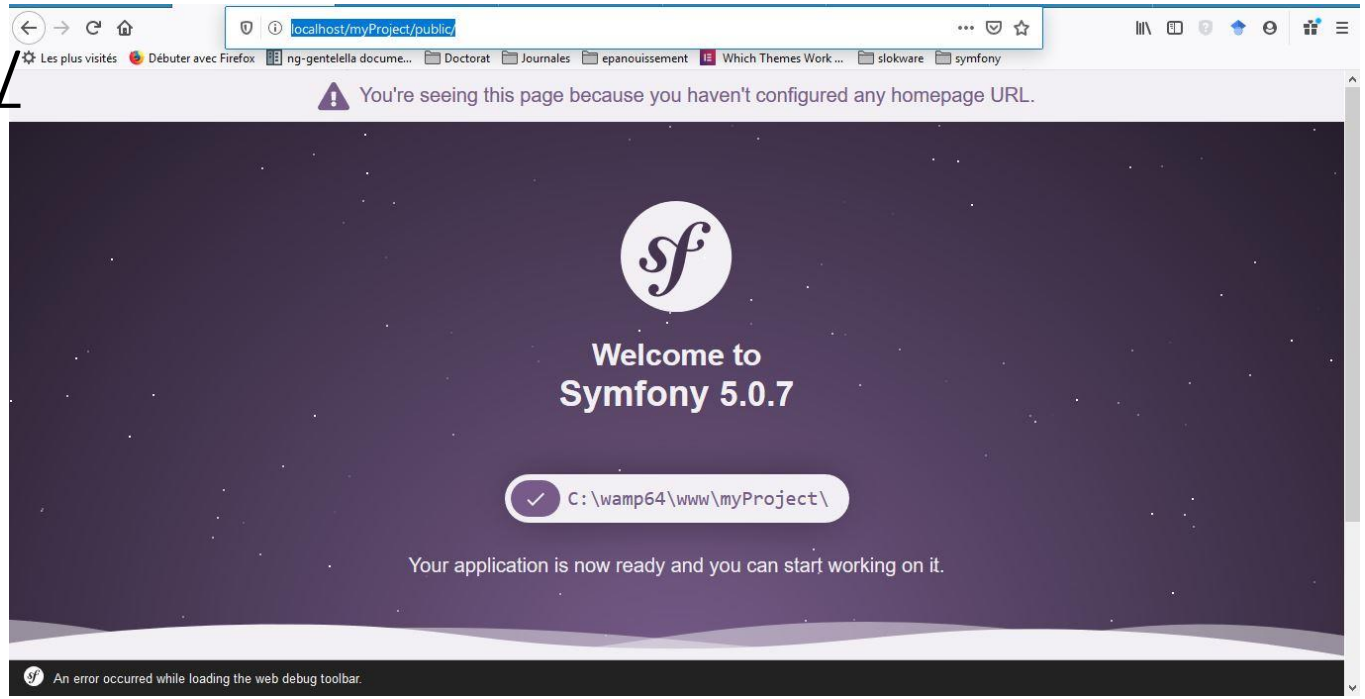
Installation

- Exigences Techniques & plateforme
- Création d'un premier projet en Symfony
- Configuration d'une Base de donnée

Exécuter votre projet:

- Méthode 2 :

1. Démarrer le wamp (ou la plateforme que vous utilisez easyPhp, ou Xamp)
2. Dans votre navigateur accéder à : http://localhost/myProject/public/
3. Une page d'accueil symfony sera affichée.



ARCHITECTURE

Chapitre #3

Architecture

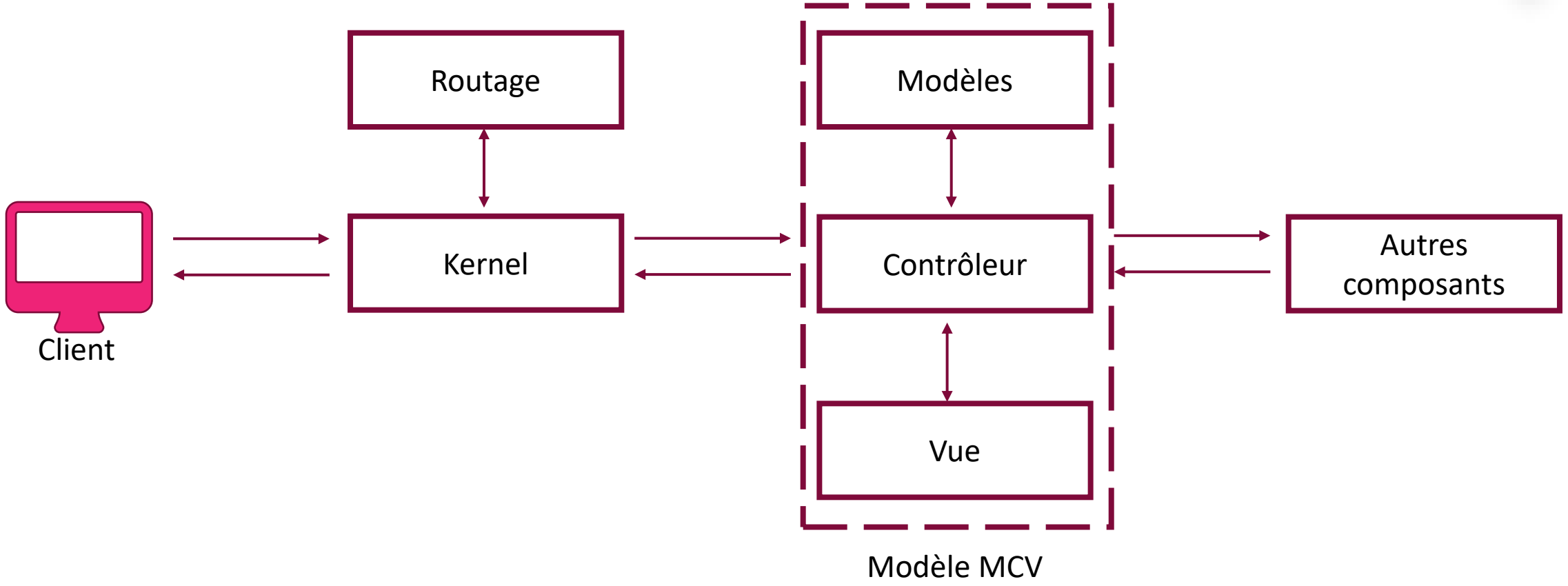
- Modèle MVC
- Créer une première page web

- Un projet symfony repose sur le pattern MVC (Model View Controller ou Model Vue Contrôleur en français) ➔ organisation du code en trois couches:
  - **La couche Modèle :** contenant le traitement logique de vos données (on y retrouve les traitements métier, les accès à la base de données, ...).
  - **La couche Vue :** représente la modélisation de l'IHM (Interface Homme Machine) : page web, données JSON, XML, etc.
  - **La couche Contrôleur :** va exécuter les traitements liés à notre application et transmettre les résultats à la vue pour que ces derniers puissent être affichés à l'utilisateur. c'est un ensemble de code qui utilise les modèles pour traiter la demande de client et donner le résultat à la vue. Dans un projet Web, cela va se matérialiser par des classes qui contiennent des fonctions qui seront appelées en fonction d'une URL. Ces dernières renverront un objet de type Symfony\Component\HttpFoundation\Respons.

Architecture

- Modèle MVC
- Créer une première page web

Schéma de traitement d'une requête sous Symfony



Architecture

- Modèle MVC
- Créer une première page web

Schéma de traitement d'une requête sous Symfony:

Exemple : Supposant que le client (utilisateur) souhaite afficher la liste des produits. Cette demande sera traité comme suit:

1. Le client clique le lien | bouton « Afficher »
2. Le kernel reçoit la demande http (Get ou Post), il cherche l'action à exécuter dans le fichier routage, ensuite il lance l'exécution de l'action enregistrer dans le contrôleur
3. Le contrôleur utilise le modèle (classe, entité) Produit pour récupérer la liste des produits depuis la base des données. Dès la réception des données, l'action va retourné la vue à afficher
4. Le kernel reçoit la réponse du contrôleur et l'affiche à l'écran du client.

Architecture

- Modèle MVC
- Créer une première page web

La création d'une page passe par deux étapes :

1. **Création de route** : le lien (URL) a votre page et qui pointe vers l'action (fonction) dans le contrôleur
2. **Créer un contrôleur :** la fonction php qui traite la requête passée comme paramètre de la fonction, pour créer une réponse de type Response qui peut contenir une page web, ou fichier binaire comme une image, pdf, word, etc.

Architecture

- Modèle MVC
- Créer une première page web

Exemple1 : on va créer notre page d'accueil

1. Dans le dossier src/Controller créer le fichier DefaultController.php

```
<?php
// src/Controller/DefaultController.php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
class DefaultController
{
    public function accueil()
    {
                $d=date('d/m/y');
                return new Response(
            '<html><body>Accueil: '.$d.'</body></html>'        );
    }
}
```

Architecture

- Modèle MVC
- Créer une première page web

Exemple 1 :

. On va lier notre fonction à un url (route):

Méthode 1 : Ajouter les lignes suivantes dans le fichier config/route.yml:

```
index:
    path: /
    controller: App\Controller\DefaultController::accueil
```

1er ligne « index » : un nom de votre choix qui représente le nom de la route, il peut être utiliser dans l'attribut src de la balise <a>, ou dans l'action du formulaire.

2eme ligne « path » : l'url qui sera affiché dans la barre de la navigation. Si l'utilisateur tape de lien dans le navigateur cette route sera exécuté.

3eme ligne « controller » : spécifie le contrôleur et la fonction à exécuter dans ce contrôleur.

Architecture

- Modèle MVC
- Créer une première page web

Exemple 1 :

2. **On va lier notre fonction à un url ou action:**

Méthode 2 : au lieu d'insérer les routes dans le fichier route.yml , il est possible d'utiliser des annotations directement dans le contrôleur :

- Modifier le contrôleur DefaultController comme suit :

Architecture

- Modèle MVC
- Créer une première page web

```
<?php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
class DefaultController
{
/**
     * @Route("/")
     */
   public function accueil()
   {
                              $d=date('d/m/y');
      return new Response(
         '<html><body>Accueil: '.$d.'</body></html>'
      );
   }
}
```

Architecture

- Modèle MVC
- Créer une première page web

Exemple 2 : au lieu de retourner un texte html , la fonction accueil va retourner cette fois une page web (Template twig).

1- Le contrôleur doit hériter la classe AbstractController, et la fonction render sera utilisée➔modifier le contrôleur DefaultController comme suit :

Architecture

- Modèle MVC
- Créer une première page web

```
<?php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
class DefaultController extends AbstractController
{
   /**
     * @Route("/")
     */
   public function accueil()
   {
            $d=date('d/m/y');
      return $this->render('default/accueil.html.twig', [
         'dateTeste' => $d,
      ]);    }
}
```

Architecture

- Modèle MVC
- Créer une première page web

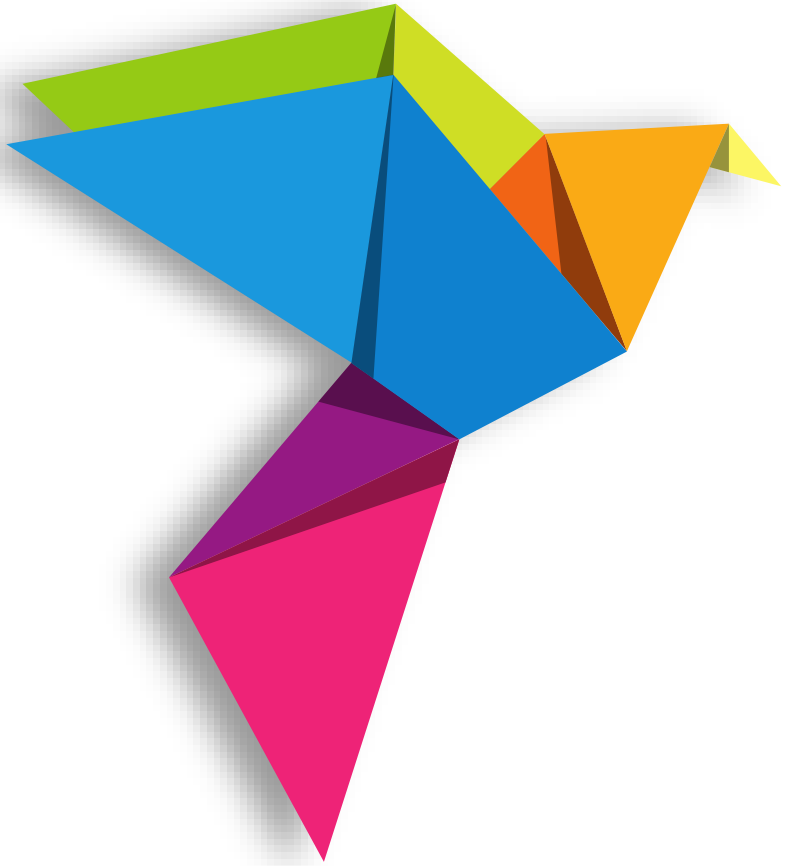
Exemple 2 :

2- Dans le dossier templates créer un autre dossier nommé default et dedans vous créer un fichier accueil.html.twig contenant le code suivant :

```
<h1>Je suis Votre page d'accueil</h1>
<p>La date d'aujourd'hui est  {{ dateTeste }}</p>
```

➔La terme {{dateTeste}} va afficher la valeur du variable dateTeste qui a été envoyé par la fonction render.

➔Actualisez la page dans votre navigateur pour visualier les modifications



Routage

Chapitre 4

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

- Les routes peuvent être configurer dans un format YAML, XML, PHP, ou des annotations. Toutes les méthodes offrent les mêmes caractéristiques. Cependant, le format annotation est le plus recommandé puisque il permet de regrouper les routes et les contrôleurs dans le même emplacement.

- Le nom des routes doivent être unique.

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

- **Format annotation :** il s'agit du format montré dans le diapositive 27
- La configuration existante dans le fichier config/routes/annotations.yaml informe symfony qu'il doit chercher les routes dans les contrôleurs . Le code de ce fichier est comme suit:

controllers:

```
controllers:
        resource: ../../src/Controller/
        type: annotation
kernel:
        resource: ../../src/Kernel.php
        type: annotation
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

- **Format Yaml, XML, PHP:** au lieu de déclarer les routes dans le contrôleur, il est possible de les mettre dans des fichiers séparés.
- Avantages : n'exige pas l'installation des nouvelles packages
- Inconvénient : il faut travailler avec plusieurs fichiers
- Exemple : ci après un exemple de la route index utilisé avant, dans les trois formats :

Format Yaml, XML, PHP:

YAML :

```
# config/routes.yaml
index:
    path: /
    # La valeur de controller a le format suivant 'controller_class::method_name'
    controller: App\Controller\DefaultController::accueil
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

Format Yaml, XML, PHP:

XML :

```
<!-- config/routes.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing
        https://symfony.com/schema/routing/routing-1.0.xsd">

<route id="index" path="/"
          controller="App\Controller\DefaultController::accueil"/>
</routes>
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

Format Yaml, XML, PHP:

PHP :

```
// config/routes.php
use App\Controller\BlogController;
use Symfony\Component\Routing\Loader\Configurator\RoutingConfigurator;

return function (RoutingConfigurator $routes) {
    $routes->add('index', '/')
        // la valeur de controller a le format [controller_class, method_name]
        ->controller([DefaultController::class, 'accueil']);
};
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

- Par défaut: une route sera exécuté quand il y a un appel normal ou un appel HTTP avec les méthodes(GET, POST, …)
- Pour exiger quand est ce la route sera exécuté ➔il faut spécifier la valeur de l'attribut « method ».

Annotations:

```
/** *
@Route("/produit/{id}", methods={"GET"})
*/
public function show(int $id)
   {     …    }
```

YAML

```
api_produit_show:
   path:       /produit/{id}
   controller: App\Controller\ProduitController::show
   methods:    GET
```

XML

```
<route id="api_produit_show" path="/produit/{id}"
controller="App\Controller\ProduitController::show"
methods="GET"/>
```

PHP

```
$routes->add('api_produitt_show', '/produit/{id}') -
>controller([ProduitController::class, 'show']) -
>methods(['GET', 'HEAD']) ;
```

Routage

- Les paramètres de la routes sont mentionnées dans **{...}**.
- Une route peut accepter plusieurs paramètres mais ils doivent avoir des noms différents.
- **Exemple**

Annotations:

```
/** *
@Route("/produit/{id}", methods={"GET"})
*/
public function show(int $id)
   {     …    }
```

YAML

```
api_produit_show:
   path:       /produit/{id}
   controller: App\Controller\ProduitController::show
   methods:    GET
```

XML

```
<route id="api_produit_show" path="/produit/{id}"
controller="App\Controller\ProduitController::show"
methods=“GET"/>
```

PHP

```
$routes->add('api_produitt_show', '/produit/{id}') -
>controller([ProduitController::class, 'show']) -
>methods(['GET', 'HEAD']) ;
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

- Exemple: l'url suivant permet d'afficher le produit ''tomate'': /produit/{tomate}; la route qui correspond cette url est montré dans le tableau ci-après. Dans la fonction show, la variable label va prendre toujours la partie dynamique du lien. Dans notre exemple : label=''tomate''
- Exemple de route avec plusieurs paramètres : /produit/{param1}/{param2}

Annotations: */** * @Route("/produit/{label}") */* public function show(String $label) { … }	**YAML** api_produit_show: path: /produit/{label} controller: App\Controller\ProduitController::show
XML <route id="api_produit_show" path="/produit/{label}" controller="App\Controller\ProduitController::show"/>	**PHP** $routes->add('api_produitt_show', '/produit/{label}') - >controller([ProduitController::class, 'show'])

Routage

- Les paramètres peuvent être des objets.
- Exemple:

```
/** * @Route("/produit/{pdt}") */
public function show(Produit pdt)
{    …    }
```

Quand le contrôleur donne une indication sur le type de paramètre, dans cet exemple c'est Produit, l'outils « param converter » (convertisseur de type des paramètres) vérifie l'existence d'une entité ou classe Produit, s'il n'existe pas symfony va retourner le message d'erreur 404.

Ajouter des critères sur les paramètres

- Supposant que notre application possède deux routes : /produit/{page} et /produit/{label}; Symfony sera incapable de différencier les deux quand il recevra cet url /produit/5; et généralement, la première trouvé sera exécuter ➔ un problème ➔ Solution : définition des exigences et critères au paramètres
- L'attribut « **requirements** » d'une route permet de spécifier les critères que les paramètres doivent vérifier pour que toute la route soit valider.
- L'option « requirements » prend comme valeur une expression régulière en php.
- Il est possible d'insérer les critères à coté des paramètre au lieu de les mettre dans l'attribut « requirements »

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

Exemple : les deux routes /produit/{page} et /produit/{label} seront définies comme suit

Annotations:

```
/** * @Route("/produit/{page}“,
name=“produit_list”, requirements={“page”=“\d+”}
) */
public function list(int $page)
   {     …    }

/** * @Route("/produit/{label}“, name=“produit_show”)
*/
public function show(String $label)
   {     …    }
```

YAML

```
api_produit_list:
   path:       /produit/{page}
   controller: App\Controller\ProduitController::list
   requirements:
      page: '\d+'

api_produit_show:
   path:       /produit/{label}
   controller: App\Controller\ProduitController::show
```

XML

```
<route id="api_produit_list" path="/produit/{page}"
controller="App\Controller\ProduitController::list">
<requirement key="page">\d+</requirement>
</route>
```

PHP

```
$routes->add('api_produitt_list', '/produit/{page}’)
->controller([ProduitController::class, ‘list’])
->requirements(['page' => '\d+’])
;
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

Exemple : (méthode2) les deux routes /produit/{page} et /produit/{label} seront définies comme suit

Annotations:	**YAML**
/** * @Route("/produit/{page<\d+>}“, name=“produit_list”) */ public function list(int $page) { … }	api_produit_list: path: /produit/{page<\d+>} controller: App\Controller\ProduitController::list
XML	**PHP**
<route id="api_produit_list" path="/produit/{page<\d+>}" controller="App\Controller\ProduitController::list“/>	$routes->add('api_produitt_list', '/produit/{page<\d+>}’) ->controller([ProduitController::class, ‘list’]);

Paramètre optionnel & valeur par défaut

- Les paramètres utilisés au niveau des routes sont par défaut obligatoire ➔ s'il n'a pas de valeur passé dans l'url, la route ne sera pas exécuté ➔ aucune correspondance entre la route et l'url ➔ Solution: l'ajout d'une valeur par défaut
- Format annotation : la valeur par défaut est définie dans la fonction du contrôleur
- Format Yaml, XML, PHP : l'utilisation de l'option « default »
- La valeur par défaut peut être aussi insérer aligné avec les paramètres.

Paramètre optionnel & valeur par défaut

Exemple: méthode 1

Annotations:

```
/** * @Route("/produit/{page<\d+>}“,
name=“produit_list”) */
public function list(int $page = 1)
   {     …    }
```

YAML

```
api_produit_list:
   path:       /produit/{page<\d+>}
   controller: App\Controller\ProduitController::list
   defaults:
      page: 1
```

XML

```
<route id="api_produit_list"
path="/produit/{page<\d+>}"
controller="App\Controller\ProduitController::list“>
<default key="page">1</default>
</route>
```

PHP

```
$routes->add('api_produitt_list', '/produit/{page<\d+>}’)
->controller([ProduitController::class, ‘list’])
->defaults(['page' => 1]);
```

Paramètre optionnel & valeur par défaut

- Exemple: méthode 2

Annotations:

```
/** * @Route("/produit/{page<\d+>?1}“,
name=“produit_list”) */
public function list(int $page)
   {     …    }
```

YAML

```
api_produit_list:
   path:      /produit/{page<\d+>?1}
   controller: App\Controller\ProduitController::list
```

XML

```
<route id="api_produit_list"
path="/produit/{page<\d+>?1}"
controller="App\Controller\ProduitController::list“/>
```

PHP

```
$routes->add('api_produitt_list', '/produit/{page<\d+>?1}’)
->controller([ProduitController::class, ‘list’]);
```

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

- Il est courant qu'un groupe de routes partage certaines options (par exemple, toutes les routes liées au produit commencent par /produit) C'est pourquoi Symfony inclut une fonctionnalité pour partager la configuration de la route.

- Une route principale est déclaré en haut du contrôleur, ensuite avant chaque fonction on définie sa propre route.

- La route principale peut aussi utiliser l'option requirements.

Routage

- Création des routes
- Correspondance des routes
- Passage des paramètres
- Groupe de route
- Génération des routes
- Vérification des routes

Exemple : (en utilisant les annotations)

- La route index & show sera nommé automatiquement : produit_index & produit_show
- L'url de la route index & show sera respectivement : /produit/ & /produit/produits/{pdt}

```
/**
 * @Route("/produit", name="produit_")
 */
class ProduitController
{
    /**
     * @Route("/", name="index")
     */
    public function index()
    {
        // ...
    }
```

```
    /**
     * @Route("/produits/{pdt}", name="show")
     */
    public function show(Produit pdt)
    {
        // ...
    }
}
```

Routage

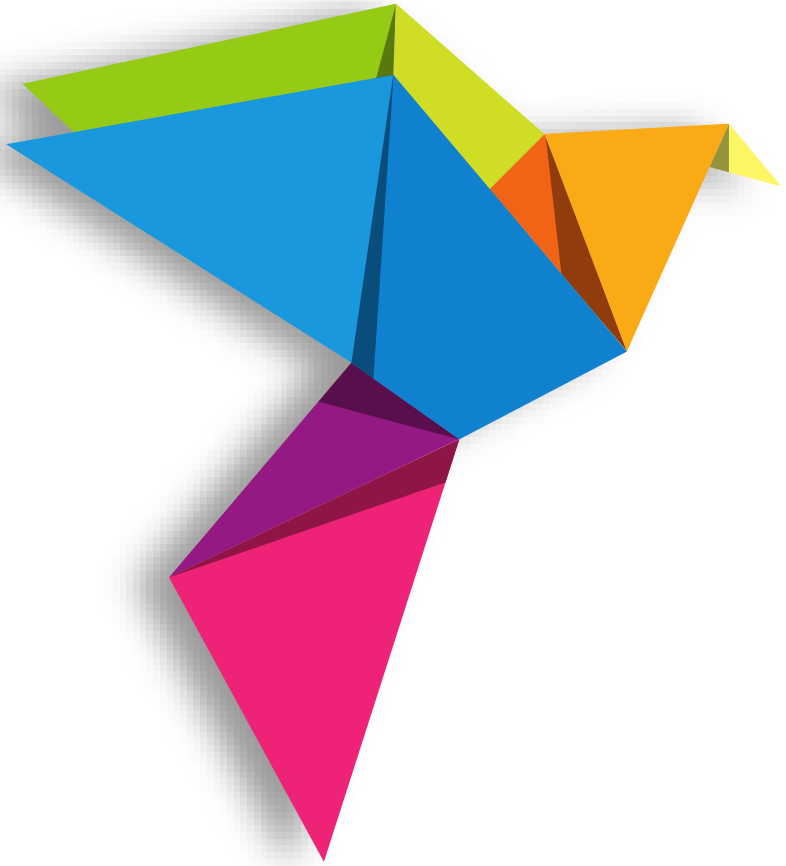
- Il est possible de générer l'url des routes, il faut savoir seulement le nom de la route et ses paramètres.
- **Exemple** de génération de l'url d'une route dans le code javascript : dans le bloc script de votre page web vous pouvez faire :

```
<script>
   const route = "{{ path('produit_show', {label: 'tomate'})|escape('js') }}";
</script>
```

NB. : si par exemple label doit prendre la valeur d'un champ text dans votre page web, ce code ne va pas fonctionner, il faut passer par un le bundle : FOSJsRoutingBundle

Routage

- Utiliser la commande suivante dans le console pour vérifier s'il y a des erreurs dans les routes définies : php bin/console debug:router
- La commande va lister toutes les routes selon l'ordre d'évaluation utilisé par symfony.
- Pour consulter le détail d'une route spécifique passer son nom dans la commande : php bin/console debug:router nomd_de_route
- Pour savoir le nom de route qui correspond à un URL spécifique utiliser cette commande : php bin/console match:router /produit/45
  - S'il existe une route qui correspond à cet URL, elle sera affiché précéder par OK, sinon, un message d'erreur sera affiché.



Contrôleur - Controller

Chapitre 5

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

- Le contrôleur est une fonction php qui récupère les paramètres de la route (peut être de type Request) pour créer et retourner une réponse de type Response.
- L'objet Response peut être de type HTML, JSON, Pdf, image, une redirection vers une autre route, une erreur 404, etc.
- La classe de base du contrôleur est : AbstractController. Il est préférable que la classe contrôleur hérite de cette classe pour bénéficier de plusieurs fonctionnalités prédéfinies.
- La liaison entre un URL et une fonction contrôleur se passe via la définition des routes (comme on a fait dans la partie routage).

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

Génération automatique des classes contrôleurs :

- La commande à utiliser : php bin/console make:Controller Commande**Controller**
- Le nom de la classe contrôleur doit se termine obligatoirement par le mot : Controller
- Génération de toute les opérations possible pour un objet ou une entité :

php bin/console make:crud Produit

- **Cette commande va générer automatiquement :**
  - ProduitController dans le dossier src/Controller/
  - ProduitType dans le dossier src/Form/ (formulaire)
  - _delete_form.html.twig, _form.html.twig, Edit.html.twig, index.html.twig, new.html.twig, show.html.twig dans le dossier templates/produit/ :

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

- La classe de base du contrôleur est : AbstractController. Il est préférable que la classe contrôleur hérite de cette classe pour bénéficier de plusieurs fonctionnalités prédéfinies.
- Pour utiliser la classe de base du contrôleur, il faut importer : Symfony\Bundle\FrameworkBundle\Controller\AbstractController
- La liaison entre un URL et une fonction contrôleur se passe via la définition des routes (comme on a fait dans la partie routage).

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

Dans une fonction contrôleur il est possible de :

- **Gérer des variables sessions :**
  - Enregistrer une variable dans la session : $session->set(‘nom’, ‘ahmed’)
  - Récupérer une variable dans la session : $variable=$session->get(‘nom’)
- **Générer un url** : $url = $this->generateUrl('app_produit_show', [‘label' => ‘tomate’]);
- **Retourner une template (page web) et passage des pramètres :**

return $this->generateUrl(‘/produits/afficheProduit.html.twig', [‘produit' => pdt]);

➔les pages web doivent être enregistrer dans le dossier templates. Dans l’exemples la page afficheProduit.

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

- **Retourner une réponse Json :** return $this->json([‘libele’=>’tomate’])
- **Retourner un fichier :** return $this->file(‘/path/produits/fichier.pdf’)
- **Rediriger vers un autre url ou route** :
  1. Redirection vers une route: ex. route index : return $this->redirectToRoute(‘index’);
  2. Redirection vers une route avec paramètre:

     return $this->redirectToRoute(‘produit_show’, [‘label’=>’tomate’]);
  3. Redirection vers une route en gardant la même variable passés à la fonction contrôleur dans la variable request de type Request :

     return $this->redirectToRoute(‘produit_show’, $request->query->all());
  4. Redirection vers un lien externe : return $this->redirect(‘www.site.com’);

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

Gérer des messages flush:

- Il s'agit des messages exceptionnels qui sont stockés au niveau de la session et supprimer une fois afficher dans une page web.
- Ils sont utiles pour l'affichage des notifications.
- Pour ajouter un message flush dans la session on tulise l'instruction suivante:

$this->addFlash(type, 'message à afficher'), le type précise la mise en forme (style css) du type. Il peut être:

- 'notice' ou 'info': pour notification ou information
- 'success' : afficher quand une opération est effectué avec succès. Ex. l'ajout d'un produit
- 'warning' : afficher un message d'alerte : par exemple une erreur dans le formulaire

Contrôleur

- Génération des contrôleurs
- Contrôleur de base

Gérer des messages flush:

- Utiliser le code suivant pour afficher les messages flush dans vos pages web :

```
{# lire et afficher un seul type#}
{% for message in app.flashes('notice') %}
    <div class="flash-notice">
        {{ message }}
    </div>
{% endfor %}
```

```
{# lire et afficher plusieurs types#}
{% for label, messages in app.flashes(['success', 'warning']) %}
{% for message in messages %}
        <div class="flash-{{ label }}">
            {{ message }}
        </div>
    {% endfor %}
{% endfor %}
```

```
{# lire et afficher tous les messages #}
{% for label, messages in app.flashes %}
    {% for message in messages %}
        <div class="flash-{{ label }}"> {{ message }} </div>
    {% endfor %}
{% endfor %}
```

Templates - Twig

Chapitre 6

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- Le mot Template désigne les vues ou encore pages web.
- Ils sont utilisés dans les contrôleurs pour générer | retourner du code HTML ou du contenu des emails dans.
- Dans Symfony, les Template sont gérés par le plugin TWIG.
- Dans Symfony, une page web peut avoir du code HTML, Javascript, et code Twig.
- Par défaut les Templates sont enregistrées dans le dossier templates sous la racine de votre projet.

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- Twig se base sur ces trois concepts :
  - **{{…}} :** pour afficher la valeur des paramètres, des variables, ou le résultat d'une expression
  - **{% … %} :** pour faire des boucles ou conditions
  - **{# … #} :** pour ajouter des commentaires, contrairement à l'HTML, ces commentaires ne seront pas visualiser dans la page affiché au client
- Une page web basée sur Twig n'accepte pas du code PHP.
- Twig propose plusieurs fonctionnalités qui peuvent remplacer le code PHP. Par exemple : {{label **| upper**}} va convertir le contenu de label en majuscule avant de l'afficher au client.

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- **Exemple 1 : <h1>{{user.name}}</h1>** dans cette exemple Twig va essayer de trouver la valeur du variable dans l'ordre suivant :
- $user[name] : teste si user est un tableau ou liste
- $user->name : teste si name est un attribut public d'une classe user
- $user->name() : teste si name est une fonction public déclarée dans la classe du variable user
- $user->getName() : si name est un attribut private ou protected dans la classe du variable user
- $user->isserName() : si name est un objet ou une méthode isser
- $user->hasName() : si name est un objet ou une méthode hasser
- Si aucune de ces valeurs existe, Twig va utiliser la valeur null.

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

Varibale globale : app

- Il s'agit d'une variable globale offerte par Twig et gérée par Twig. Cette variable possède des informations sur l'application.
- app.user : l'utilisateur connecté à l'application ou null s'il n'est pas authentifié
- app.request : l'objet request de type Request qui possède les données de la requête courante
- app.session : représente la session de l'utilisateur courant ou null, s'il n'y a aucun utilisateur
- app.flashes : les messages flashes enregistrées ou insérer par la fonction contrôleurs
- app.environment : le nom de l'environnement courant : dev ou prod.
- app.debug : true ou flase, spécifie si le mode debug est ectivé
- app.token : represente l'objet token de sécurité

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- Pour préciser un lien hypertext ou une action d'une bouton, il est possible d'utiliser le nom des routes au lieu des urls, avec la fonction Twig : **path()**
- **Exemple 1 :** Accueil
- **Exemple 2 :**

```
{% for produit in produits %}
   <h1>
      <a href="{{ path('produit_show', {libele:produit.libele}) }}">{{ produit.libele }}</a>
   </h1>
   <p>{{ produit.prix }}</p>

{% endfor %}
```

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- Le mot assets désignes les fichiers CSS, javascript, images, etc
- Les assets sont généralement enregistrés dans le dossier public qui se trouve dans la racine du projet Symfony. Ce dossier peut contenir trois sous dossiers appelé : images (contient toutes les images de votre projet); css (tous les fichiers css); js (tous les fichiers js)
- **Image:**
- **css:** <link href="{{ asset('css/site.css') }}" rel="stylesheet"/>
- **Javascript :** <script src="{{ asset('js/loader.js') }}"></script>

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- Il est possible d'inclure une ou plusieurs template dans une seule en utilisant la fonction **include**
- **Etapes :**
  - Créer le sous template et nommé la : _soustemplate.html.twig. Le ' _ ' n'est pas obligatoire mais c'est une convention pour distinguer les templates normales des sous templates.
  - Dans le template principale et à l'endroit où vous voulez insérer le contenu de sous template vous ajoutez l'instruction suivante : **{{ include('produits/_soustemplate.html.twig') }}**
  - Si le sous template a besoin des paramètres vous pouvez utiliser l'instruction suivant :

{{ include('blog/_soustemplate.html.twig', {paramName: paramValue}) }}

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

- Dans une application, il existe souvent des sections statique : header, footer, menu, etc; pour éviter la répétition de ces parties dans chaque page web, il est possible de créer une template de base qui définie la structure générale de vos pages. Ensuite chaque nouvel page va hériter seulement de la page principale.

- **Symfony propose l'utilisation de trois niveau d'héritage montrés ci-après :**

1. Création de template de base : base.html.twig; qui définie le contenu des section : <head>, <header> et <footer>. Chaque section sera inèrer dans un block twig avec la balise {% block nomBlock %} … {% endblock %}
2. Création du template layout.html.twig qui hérite de base.html.twig. Cette template va définir le contenu de chaque section
3. Créer vos pages et mentionner qu'ils hérite du template layout.

Un exemple de code est montré ci-après.

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

{# templates/base.html.twig #}

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Mon Application{% endblock %} </title>
        {% block stylesheets %}
            <link rel="stylesheet" type="text/css" href="/css/base.css"/>
        {% endblock %}
    </head>
    <body>
        {% block body %}
            <div id="sidebar">
                {% block sidebar %}
                    <ul>
                        <li><a href="{{ path('index') }}">Accueil</a></li>
                        <li><a href="{{ path('produit_list') }}">Produit</a></li>
                    </ul>
                {% endblock %}
            </div>
            <div id="content">
                {% block content %} ……..{% endblock %}
            </div>
        {% endblock %}
    </body>
</html>
```

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

{# templates/blog/layout.html.twig #}

```
{% extends 'base.html.twig' %}
{% block content %}
    <h1>Produit</h1>
    {% block page_contents %}  {% endblock %}
{% endblock %}
```

{# templates/blog/index.html.twig #}

```
{% extends 'blog/layout.html.twig' %}
{% block title %} Index{% endblock %}
{% block page_contents %}
    {% for article in produit %}
        <h2>{{ article.libele }}</h2>
        <p>{{ article.prix }}</p>
    {% endfor %}
{% endblock %}
```

Template

Conditions :

```
{% if online == false %}
        <p>Our website is in maintenance mode. Please, come back later.</p>
{% endif %}
```

Ou :

```
{% if not online %}
        <p>Our website is in maintenance mode. Please, come back later.</p>
{% endif %}
```

Pour tester plusieurs conditions dans le même block if, il est possible d'utiliser : **and** et **or**

Template

Condition multiples :

```
{% if produit.qteStock > 10 %}
        Available
{% elseif produit.qteStock> 0 %}
        Only {{ produit.qteStock }} left!
{% else %}
        Sold-out!
{% endif %}
```

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

Boucle for: parcours des éléments d'une séquence

```
<h1>Members</h1>
<ul>
{% for user in users %}
        <li>{{ user.username|e }}</li>
{% endfor %}
</ul>
```

Boucle for: parcours des éléments dans un intervalle

```
{% for i in 0..10 %}
        * {{ i }}
{% endfor %}
```

Template

- Variables
- Lien & URL
- Assets
- Include & Héritage
- Instruction
- Vérification

Boucle for: afficher un message si la séquence ne contient aucun élément

```
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% else %} <li><em>aucun élèment</em></li>
    {% endfor %} </ul>
```

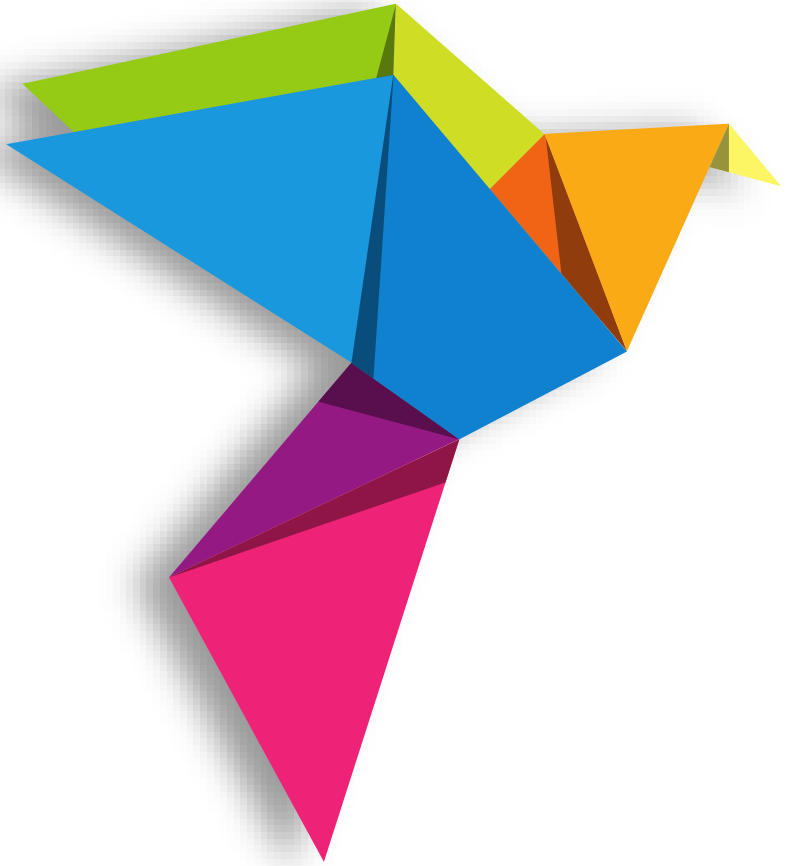
Boucle for: parcours de quelques éléments de la liste (ex: les 10 premiers éléments)

```
<h1>Top Ten Members</h1>
<ul>
{% for user in users|slice(0, 10) %}
    <li>{{ user.username|e }}</li>
{% endfor %}
</ul>
```

Template

- Vérification des erreurs de syntaxe :

- Vérification de toutes les templates : php bin\console lint:twig
- Vérifier un seul répertoire : php bin\console lint:twig templates/produits
- Vérifier une seule template : php bin\console lint:twig template/default/index.html.twig



Formulaire

Chapitre 7

Formulaire

La création des formulaires en Symfony passe par trois étapes :

1. Création de formulaire : dans une fonction contrôleur ou via une classe spécifique Form
2. Afficher le formulaire dans une template
3. Manipulation de formulaire : validation, soumission, enregistrement dans la base de données

Ci-après on considère qu'on va créer un formulaire pour l'ajout d'un produit.

Un produit possède les informations suivantes : idPdt, libele, prix, et qteStock

Formulaire

La création des formulaires en Symfony passe par trois étapes :

1. Création de formulaire : dans une fonction contrôleur ou via une classe spécifique Form
2. Afficher le formulaire dans une template
3. Manipulation de formulaire : validation, soumission, enregistrement dans la base de données

Ci-après on considère qu'on va créer un formulaire pour l'ajout d'un produit.

Un produit possède les informations suivantes : idPdt, libele, prix, et qteStock

Formulaire

La classe Produit (entity)

```
<?php
namespace App\Entity;
class Produit
{
   private $idpdt;
   private $libele;
   private $prix;
    private $qtestock;
}
```

Formulaire

Création de formulaire :

Il est possible de générer le formulaire par la commande :

```
php bin/console make:Form;
```

- Vous aurez un message pour saisir le nom du formulaire (dans ce cas Produit), ensuite le nom de la classe qui sera lié au formulaire (Produit).
- Le formulaire sera généré par le nom, ProduitType ➔ c'est une convention d'ajouter le mot Type aux noms des classes formulaires.
- Les formulaires sont enregistrés dans le dossier src/Form

Formulaire

La classe formulaire : ProduitType

```
class ProduitType extends AbstractType{
public function buildForm(FormBuilderInterface $builder, array $options)    {
$builder->add('idpdt',TextType::class )
            ->add('libele', )
            ->add('prix', IntegerType::class, array('required' => true,'label'=>'Prix'))
            ->add('qtestock');
}
public function configureOptions(OptionsResolver $resolver){
            $resolver->setDefaults([ 'data_class' => Produit::class, ]);
}}
```

Formulaire

La fonction contrôleur qui retourne le formulaire :

```
public function ajouterProduit(Request $request) {
        $produit = new Produit();
        // ...
        $form = $this->createForm(ProduitType::class, $produit);
        return $this->render('produit/new.html.twig', [ 'form' => $form->createView(), ]);
}
```

Formulaire

Le template new.html.twig :

```
{{ form_start(form) }}
        {{ form_errors(form.idpdt) }}
        <label class="control-label col-md-3 col-sm-3 col-xs-12" for="first-name">
                Réfèrence <span class="required">*</span>
        </label>
        {{ form_widget(form.idpdt,{'attr': {'required':'required'}})}}
        <label class="control-label col-md-3 col-sm-3 col-xs-12" for="first-name">
                Libelé <span class="required">*</span>
        </label>
        {{ form_widget(form.libele,{'attr': {'required':'required'}})}}
        ……
        <button type="submit" id="soumettre" class="btn btn-success">Valider</button>
{{ form_end(form) }}
```

Formulaire

La manipulation de formulaire : soumission

- Il s'agit de transférer les données saisies par l'utilisateur et les stockes dans les attributs de l'objet.
- Pour la maintenabilité du code il est préférable de fusionner le code qui retourne le formulaire avec celui qui la traite.
- La fonction contrôleur ajouterProduit sera modifiée pour permettre le transfère des données saisie à la base de données.

Formulaire

- **La fonction contrôleur qui retourne le formulaire :**

```
public function ajouterProduit(Request $request)
{

   $produit = new Produit();

   $form = $this->createForm(ProduitType::class, $produit,
    'action' => $this->generateUrl(‘produit_new'), 'method' => ‘POST');

   $form->handleRequest($request);
   if ($form->isSubmitted() && $form->isValid()) {
      // $form->getData() possède les données saisies
      // aussi la variable `$produit` a été mise à jour
      $produit = $form->getData();

      // ... manipulation des données: stockage dans le base de données
      $entityManager = $this->getDoctrine()->getManager();
      $entityManager->persist($produit);
      $entityManager->flush();

      return $this->redirectToRoute('produit_success');
   }

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

Formulaire

D’autre type des champs :

ChoiceType Exemple : ->add('sexe',ChoiceType::class, array('choices' => array('Femme' => 'Femme', 'Homme' => 'Homme'), 'expanded' => true))	**Entités ou classes:** ->add(‘clients', EntityType::class, array('required' => false, 'class' => Client::class, 'placeholder' => ‘-- Client --', 'choice_value' => ‘idclt', 'multiple'=>false))
EmailType : ->add('email',EmailType::class)	**Texte area** : ->add('detail', TextareaType::class)
Checkbox ->add('agreeTerms',CheckboxType::class)	**Date :** ->add(‘date', DateType::class)

" Ce que nous devons apprendre à faire, nous l'apprenons en le faisant. "

Aristote