Formation Symfony 4

Symfony

Les Pré-Requis

- Pratique de PHP et connaissances en HTML/CSS
- Bases de la POO en PHP
- Avoir un IDE (IntelliJ, PhpStorm, Sublim Text, Atome, etc.)
- Savoir utiliser un serveur local type WAMP (Windows) / XAMP (multi-plateforme) / LAMP (Linux)

  => Apache / PHP 7.1 / MySQL

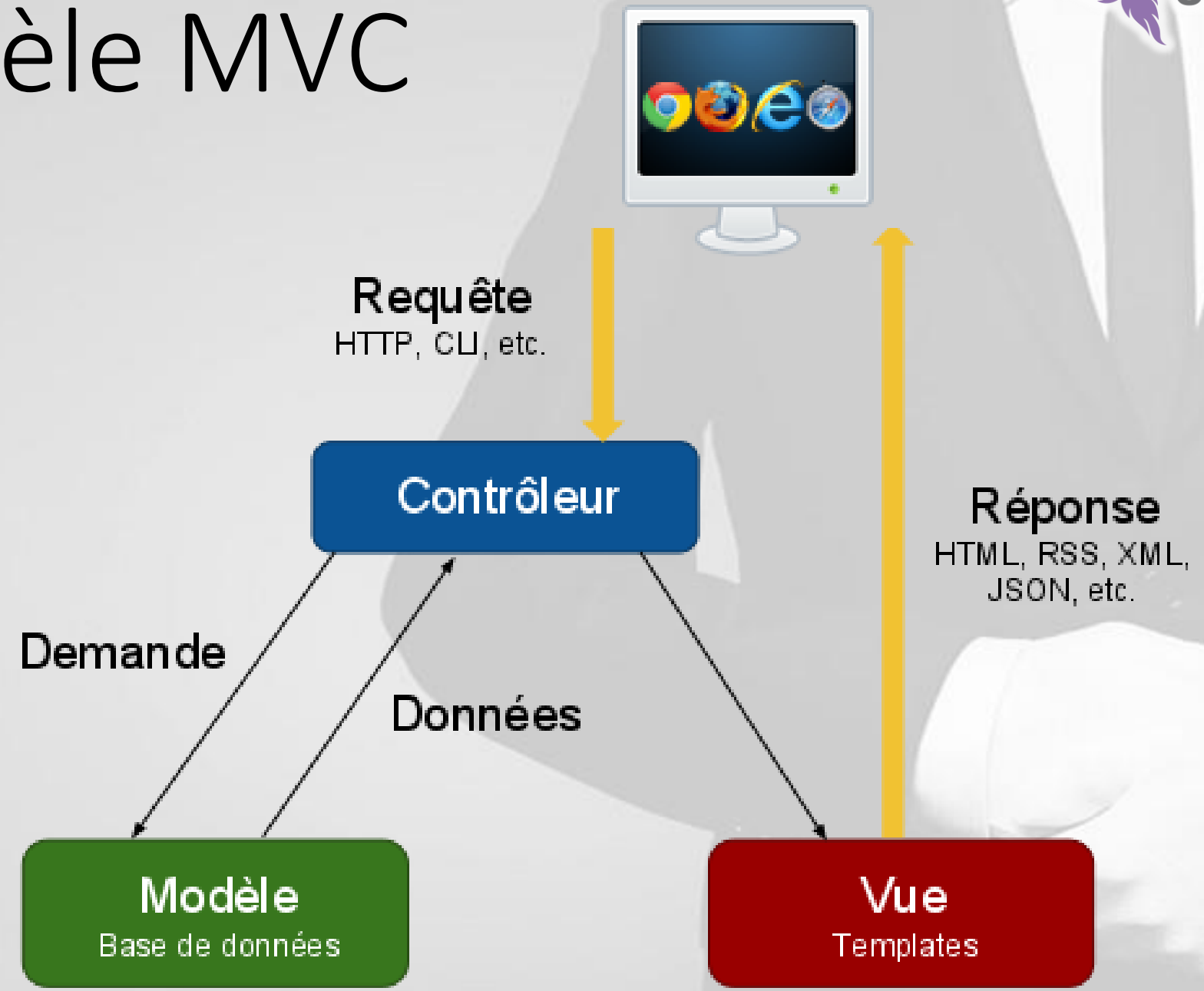
Pourquoi Symfony ?

- Framework MVC PHP Open-Source (gratuit et libre)
- Développer des sites web de qualité professionnelle
- Grande communauté de développeurs et beaucoup de documentation
- Système de bundles indépendants et flexibles
- Philosophie : « Ne pas réinventer la roue »

Mais qu’est ce qu’un Framework ?

- Définition : C’est un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application. C'est le framework qui établit les fondations d’une application ou son squelette applicatif.
- Cela permet notamment de :
  - ✓ Améliorer la productivité des développeurs
  - ✓ Permettre le travail en équipe grâce à une structure bien organisée
  - ✓ Garantir la maintenance, la sécurité et l’évolutivité

Modèle MVC


Requête
HTTP, CLI, etc.
Contrôleur
Réponse
HTML, RSS, XML,
JSON, etc.
Demande
Données
Modèle
Base de données
Vue
Templates

Composer

JUNIORISEP

- Téléchargement : https://getcomposer.org/download/

(faire attention à bien choisir la dernière version de PHP)

- Gestionnaire de dépendance PHP (bibliothèques, bundles, etc.)
- Utilisé en ligne de commande (CLI)
- Inclue le plugin Symfony Flex (pour les versions >= 3.3) qui rajoute une flexibilité sur la gestion des bundles

Utiliser composer (avec Symfony Flex)

- Mise à jour de toutes les dépendances :

$ composer update

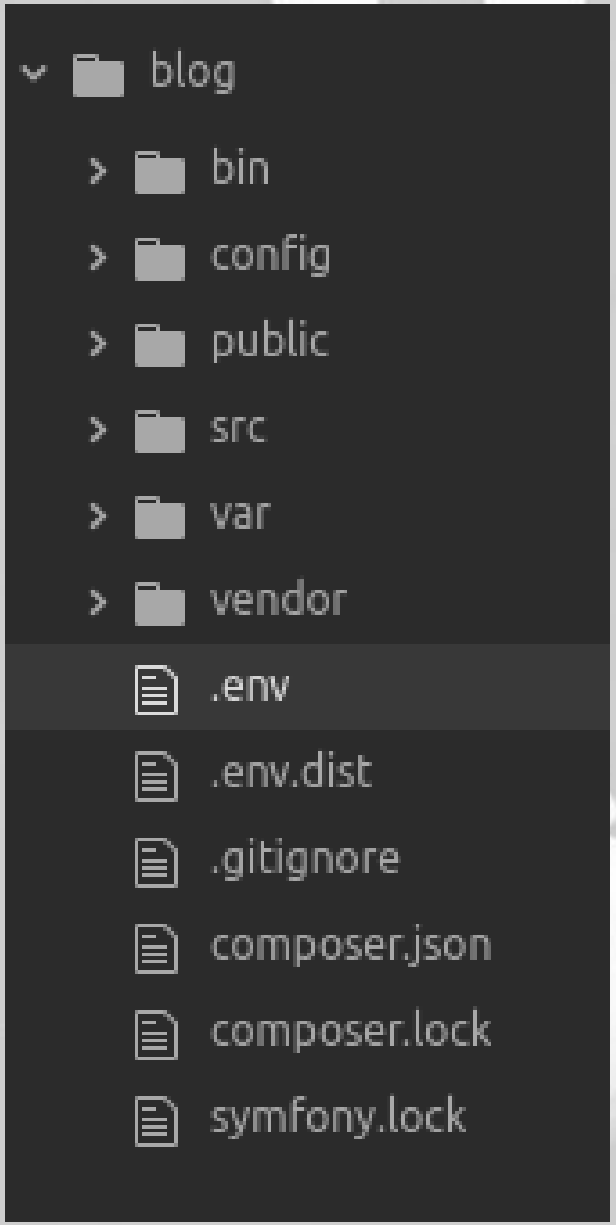
- Ajouter un bundle : $ composer require nom_bundle
- Supprimer bundle : $ composer remove nom_bundle

- Installer les dépendances : $ composer install

=> Lit le fichier composer.json, résout les dépendances et les installe dans le dossier vendor (pour importer un projet par exemple, très utile car il suffit juste d'avoir le composer.json)

Structure d’un projet

```
$ composer create-project symfony/skeleton blog
```



Démarrer le serveur de développement de Symfony

Dans le répertoire de votre projet :

$ composer require server --dev

Puis lancer le serveur avec : $ php bin/console server:run

Si tout va bien, rendez-vous dans votre navigateur et tapez :

localhost:8000/

Installer les principaux bundles

- maker-bundle => créer des Controller, Entity, Form, etc. facilement
- annotations => permet de mettre les routes directement dans le Controller
- profiler => outil pour le débuggage
- twig => moteur de template
- orm => base de données
- form => formulaires
- …

Notre premier « Hello world » !

- Créer un Controller (classe PHP)

$ php bin/console make:controller ControllerName

- Créer une méthode index() dans ce Controller qui retourne un objet Response avec en paramètre « Hello World »

```php
/**
 * @Route("/hello")
 */
public function index()
{
    return new Response( content: "Hello World");
}
```

Le Routeur

- Le rôle du routeur est de déterminer quel route utiliser pour la requête courante.
- Le rôle d'une route est d'associer une URL à une action du contrôleur

⇒ Dans notre exemple :

URL : /hello -> action controller : index()

- On peut faire passer des paramètres dans une route
- Pour voir toutes les routes : $ php bin/console debug:router

Le moteur de template TWIG

- Permet de séparer le code PHP (au niveau du Controller) du code HTML (au niveau des vues)
- Simplifier l'affichage des données et le rendre plus lisible
- Système d'héritage très utile pour l'organisation des vues

Syntaxe TWIG usuelle

- Afficher une variable : {{ maVariable }}
- Afficher l'attribut d'un objet : {{ user.email }}
- Déclarer un bloc : {% block block_name %} … {% endblock %}
- Conditions : {% if … %} …. {% elseif %} …. {% else %} …. {% endif %}
- Boucle : {% for user in listusers %}
  {{ user.info }}
  {% endfor %}

Le Système d'héritage

JUNIORISEP

base.html.twig

```twig
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Symfony{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>

    <body style="text-align: center">
        <h1>Mon premier site en Symfony !</h1>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

layout.html.twig

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

{% block title %}{{ parent() }}{% endblock %}

{% block body %}
    <div>
        <p>Mon site est géniale !</p>
        <p>Vous trouverez pas mieux ailleurs</p>
    </div>
{% endblock %}
```

Les services

- Un service est une classe PHP global qui remplit une fonction bien spécifique comme par exemple envoyer des mails, effectuer une tâche récurrente ou tout simplement instancier une autre classe
- Permet de séparer la logique métier dans l'application (on décharge du controller ce qu'il n'est pas censé effectuer, il fera seulement appel aux services)
- Permet une réutilisabilité partout dans le code
- Tout est service ! (Twig, templating, kernel, profiler, session, …)
- Pour voir les services par défaut : $ php bin/console debug:container
- Pour voir tous les services : $ php bin/console debug:autowiring

Comment créer et utiliser un service ?

- Configurer le fichier services.yaml
- Créer un répertoire Services dans /src (tous nos services se trouveront là)
- Créer votre classe PHP dans Services avec toute sa logique
- Injecter directement les dépendances dans le Controller (dans les paramètres des méthodes de votre choix)

Un exemple peut être?

JUNIORISEP

Greet.php

```php
<?php
/** Created by PhpStorm. ...*/

namespace App\Services;

class Greet
{
    public function greeting(){
        $greeting = ['Hey', 'Hello', 'Hola'];
        $greet = $greeting[array_rand($greeting)];
        return $greet;
    }
}
```

Dans le Controller

```php
//exemple d'un service basique
/**
 * @Route("/greet")
 */
public function greet(Greet $greet)
{
    return new Response($greet->greeting());
}
```

Un exemple plus compliqué

SendEmail.php

```php
<?php
/** Created by PhpStorm. ...*/

namespace App\Services;


class SendEmail
{
    protected $mailer;

    //Notre service necessite un autre service pour fonctionner (ici Swiftmailer)
    public function __construct(\Swift_Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function sendMessage($subject, $text, $from, $to){
        $mail = $this->mailer;
        $message = (new \Swift_Message($subject))
            ->setFrom($from)
            ->setTo($to)
            ->addPart($text);
        $mail->send($message);
    }
}
```

Dans le Controller

```php
//exemple de service d'envoi de mail
/**
 * @Route("/mail")
 */
public function mail(SendEmail $sendEmail)
{
    $sendEmail->sendMessage( subject: 'Salutation',
                             text: "<h1>Bonjour l'ami ! </h1>",
                             from: 'charles.boissy@isep.fr',
                             to: 'paul.duval@isep.fr'
                           );
    return $this->redirectToRoute( route: 'home');
}
```


A vous de jouer !

TP 1

- On souhaite que l'application se souvienne de nous. A chaque fois que la méthode hello du controller est appelé avec un paramètre (un nom), on veut que ce nom soit stocké dans une session (en remplaçant éventuellement celui qui existait déjà). A chaque fois que la méthode hello est appelé sans paramètre, on veut retrouver le nom précédemment enregistré en session. Finalement, si la méthode hello est appelé pour la première fois sans paramètre, on génère un nom aléatoire qu'on stocke en session
- Exemple : Si j'appelle une fois http://localhost:8000/hello/charles cela va m'afficher « Hello charles ! ». Si ensuite j'appelle http://localhost:8000/hello cela va toujours m'afficher « Hello charles ! ». En revanche, si on appelle pour la première fois http://localhost:8000/hello (sans paramètre) alors on va obtenir un nom aléatoire qui va être stocké en session pour les appels futurs.

TP 2

- On souhaite que l’url /color affiche le mot de la même couleur dynamiquement sur le navigateur. Ex : http://localhost:8000/color/bleu affiche « bleu » en bleu.
- Utiliser l'héritage twig pour afficher la vue.
- Pour la couleur rouge, afficher en plus le message : "Attention risque de virus" en rouge.

A la prochaine séance

- Nous verrons :
  - Les Entités et les Repository et de manière générale comment se servir d'une base de données (avec Doctrine)
  - Les formulaires
  - Comment créer un CRUD (Create Read Update Delete)