Symfony 5 – TP1 : Introduction

I. Installation et configuration de la structure Symfony 5

- Pour créer votre nouvelle application Symfony, assurez-vous d'abord que vous utilisez ***PHP 7.1*** ou s upérieur (***http://www.wampserver.com/***) et que ***Composer*** est ***installé*** (***https://getcomposer.org/download/)***.

```
$ php -v
$ composer -V
```

- Créez votre nouveau projet en exécutant :

```
> composer create-project symfony/website-skeleton my-project
```

➔Cela va créer un nouveau répertoire ***my-project***, y télécharger des dépendances et même générer les répertoires et fichiers de base dont vous aurez besoin pour commencer. En d'autres termes, votre nouvelle application est prête !

II. Lancer votre application Symfony

- Pour le développement, il est pratique d'utiliser le ***serveur Web PHP de Symfony*** .

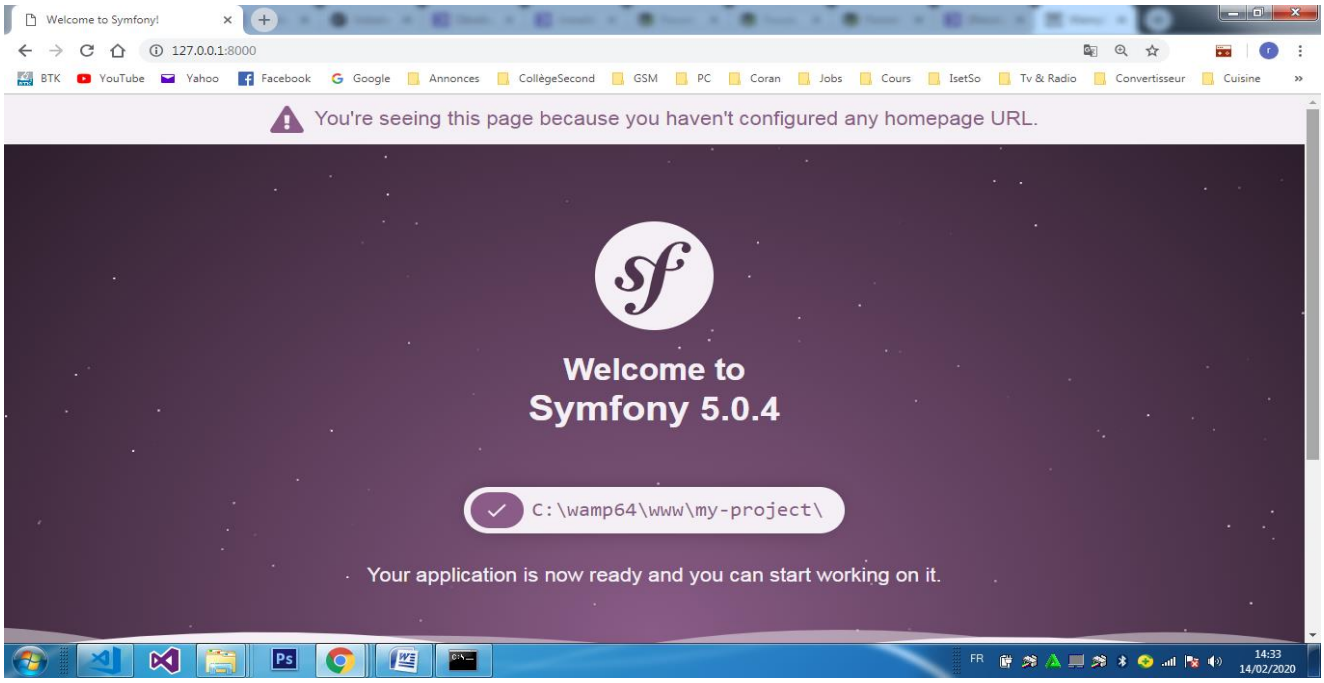
- Accédez à votre nouveau projet et démarrez le serveur :

```
>cd  my-project
>composer require profiler --dev
>php –S 127.0.0.1:8000 –t public ou bien php -S localhost:8000 -t public
```

- Ouvrez votre navigateur et accédez à ***http://localhost:8000/.*** Si tout fonctionne, vous verrez une page d'accueil.

- Une fois que vous avez fini de travailler, arrêtez le serveur en appuyant sur ***Ctrl+C*** à partir de votre terminal.

❖ Ouvrir l'url suivante : http://localhost:8000/



III. Créez votre première page dans Symfony 5

La création d'une nouvelle page, qu'il s'agisse d'une page HTML ou d'un nœud final JSON, est un processus en deux étapes :

1. ***Créer un itinéraire*** *:* un itinéraire est l'URL (par exemple ***/about***) de votre page et pointe vers un contrôleur ;
2. ***Créer un contrôleur*** *:* Un contrôleur est la fonction PHP que vous écrivez qui construit la page.

1. Créer une page : Route et contrôleur

Supposons que vous souhaitiez créer une page - ***/lucky/number*** - qui génère un nombre chanceux (bien aléatoire) et l'imprime.

❖ Créez une "classe de contrôleur" et une méthode "contrôleur" (éditer le fichier ***/src/Controller/LuckyController.php***) à l'intérieur de celle-ci :

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

use Symfony\Component\HttpFoundation\Response;

class LuckyController
{
Publicfunction number()
{
$number=random_int(0,100);

Return new Response(
'<html><body>Lucky number: '.$number.'</body></html>'
);
}
}
?>
```

- Associer cette fonction de contrôleur à une URL publique (par exemple, /lucky/number) afin que la méthode number() soit exécutée lorsqu'un utilisateur y accède. Cette association est définie en créant une **route** dans le fichier ***config/routes.yaml*** :

```yaml
# config/routes.yaml

# the "app_lucky_number" route name is not important yet
app_lucky_number:
   path : /lucky/number
  controller: App\Controller\LuckyController::number
```

C'est tout ! Si vous utilisez le serveur Web Symfony, essayez-le en allant à :

http: // localhost: 8000/lucky/number

Si un numéro porte-bonheur vous est imprimé à nouveau, félicitations ! Mais avant de jouer à la loterie, vérifiez comment cela fonctionne.

Rappelez-vous les deux étapes pour créer une page ?

- ***Créer une route*** **:** Dans ***config/routes.yaml***, la route définit l'URL de votre site.
  - ➔ page (path) et quel controller à appeler.
- ***Créer un contrôleur*** **:** il s'agit d'une fonction dans laquelle *vous créez* la page et, au final, vous retournez un objet Response.

2. Routes d'annotation

❖ Au lieu de définir votre itinéraire dans YAML, Symfony vous permet également d'utiliser ***des itinéraires d'annotation***. Pour ce faire, installez le package d'annotations :

```
> composer require annotations
```

❖ Ajouter votre itinéraire directement *au - dessus* du contrôleur :

```
// src/Controller/LuckyController.php

// ...
+ use Symfony\Component\Routing\Annotation\Route;

class LuckyController
{
+    /**
+     * @Route("/lucky/number")
+     */
    public function number()
    {
        // this looks exactly the same
}
}
```

➔La page - ***http://localhost:8000/lucky/number*** fonctionnera exactement comme avant ! Les annotations sont la méthode recommandée pour configurer les itinéraires.

3. Rendre un modèle

❖ Si vous retournez du code HTML à partir de votre contrôleur, vous souhaiterez probablement générer un modèle. Symfony est fourni avec ***Twig*** : un langage de gabarit simple, puissant et plutôt amusant.

❖ Assurez-vous que LuckyController étend la classe de base ***AbstractController*** Symfony:

```
// src/Controller/LuckyController.php
// ...
+ use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
- class LuckyController
+ class LuckyController extends AbstractController
{
    // ...
}
```

Maintenant, utilisez la fonction pratique ***render()*** pour rendre un modèle. Passez-lui une variable number afin que vous puissiez l'utiliser dans ***Twig***:

```
// src/Controller/LuckyController.php
// ...
class LuckyController extends AbstractController
{
/**
    * @Route("/lucky/number")
    */
public function number()
{
$number=random_int(0,100);
return $this->render('lucky/number.html.twig',['number'=>$number,]);
}
}
```

- Les fichiers de modèles résident dans le répertoire ***templates/***, créé pour vous automatiquement lors de l'installation de Twig.

- Créez un nouveau répertoire ***templates/lucky*** avec un nouveau fichier ***number.html.twig*** dans :

```
{# templates/lucky/number.html.twig #}
<h1>Your lucky number is {{number}}</h1>
```

- La syntaxe ***{{ number }}*** est utilisée pour *imprimer les* variables dans Twig. Actualisez votre navigateur pour obtenir votre *nouveau* numéro porte-bonheur !

http: // localhost: 8000 / lucky / number

❖ Ajouter ce code juste ***avant la dernière*** accolade fermante :

```
/**
  * @Route("/hello/", name="hello")
  */
public function hello()
{
        return $this->render('lucky/hello.html.twig');
}
```

Voici le contenu du fichier hello.html.twig :

```
<h1>Hello world</h1>
```

➔ Ouvrir l'url suivante : ***http://localhost:8000/hello***

➔ On devrait avoir le texte « Hello Word » comme suit :

POUR RÉSUMER :

- La méthode (fonction) hello () est appelée «**action»** Elle se trouve dans une classe appelée **Controller**
- Le Controller se trouve dans ***/src/Controller/***
- Une action est appelée depuis le navigateur via sa **route** (chemin, path, URL) Une route possède un nom (unique)
- Une action répond au navigateur en envoyant le contenu d'un **template** Twig
- Un fichier **Twig** se trouve dans ***/templates/*** (sur la racine ou sous-dossier)

  ATTENTION : Symfony est sensible à la casse (nom des fichiers et des dossiers)

4. Modifications

But : afficher une page qui contient « Salut, toi ! » ou « Bonsoir » ou « Bonjour Batman » (si l'utilisateur a précisé son nom *Batman* dans l'URL)

a) L'URL « /salut/ » affichera « Salut, toi !»

b) L'URL « /bonsoir/ » affichera « Bonsoir !»

c) L'URL « /bonjour/Superman» affichera « Bonjour Superman !»

Les URL données sont des url relatives par rapport à l'url de base de votre application !

a) La route/salut/

Ajouter une nouvelle action (à compléter) :

```
/**
 * @Route("...", name="salut")
 */
public function salut()
{
        return $this->render('lucky/salut.html.twig');
}
```

Créer ***/templates*/lucky/salut.html.twig**avec le code suivant :<h1>Salut, toi ! </h1>

b) La route/bonsoir/

Ajouter une nouvelle action ainsi que son template, bien tester.

c) La route/bonjour/X

Ajouter la nouvelle action suivante :

```
/**
 * @Route("/bonjour/{nom}", name="bonjour")
 */
public functionbonjour($nom)
{
      return $this->render('lucky/bonjour.html.twig', ['le_nom' => ...]);
}
```

Créer ensuite ***/templates*/lucky/bonjour.html.twig**avec le code suivant :

```
<h1>Bonjour {{le_nom}}</h1>
```

Sauvegarder le tout et ouvrir l'url suivante :

http://127.0.0.1:8000/bonjour/Batman Voici le résultat :

- Remplacer « Batman » par votre propre nom et prénom (utiliser les accentués, espaces ...)

Ajouter ceci dans le ***templatebonjour.html.twig***

```
<h1>Nombre de caractères : {{le_nom | length}}</h1>
```

Tester **puis** ajouter ceci :

```
<h1>En MAJUSCULE : Bonjour {{le_nom | upper}}</h1>
```

Tester puis ajouter ceci (à compléter) :

```
<h1>En minuscule : Bonjour {{le_nom | ...}}</h1>
```

Tester puis ajoute ceci :

```
<h1>Bonjour {{le_nom | title}}</h1>
```

Ce qui est après le « **|** » (***pipe***) s'appelle «***filtre twig***», on peut les appliquer en série (comme dans Unix)

5. Mise en forme des vues

Ajouter ceci au début de toutes les vues qu'on vient de créer :

```
<h1 style="text-align: center">Symfo Intro </h1>
<hr/>
```

On voudrait changer de titre ⇒ C'est fatiguant de refaire la manip, pour cela on va créer un fichier de base qui contient tout ce qui est en commun avec les différents template, ensuite les différents template vont hériter de ce fichier.

Créer le fichier ***/templates/base.html.twig***:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>{% block title %}Page Title{% endblock %}</title>
</head>
<body>
<h1 style="text-align: center">Symfo Intro </h1>
{% block body %}{% endblock %}
<hr/>©
2018
</body>
</html>
```

Voici le nouveau code de ***hello.html.twig*** :

```
{% extends 'base.html.twig' %}
{% block body %}
<h1>Hello world</h1>
{% endblock %}
```

Ouvrir l'url ***http://127.0.0.1:8000/hello/***:

IMPORTANT : on remarque qu'il y a une nouvelle barre en bas de l'écran, elle s'appelle ***WDT (Web Development Tool)***, elle ne s'affichait pas avant car il n'y avait pas de balise body !

Appliquer le même traitement aux autres Twig. Montrer le résultat à votre enseignant.

Exercice : Faire en sorte que chaque page possède une balise TITLE différente.

6. Appliquer Bootstrap au layout

Remplacer le code de ***base.html.twig*** par ceci :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>{% block title %}Page Title{% endblock %}</title>
<link href=" https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="jumbotron text-center">
<h1>Symfo Intro</h1>
</div>
{% block body %}{% endblock %}
<hr/>
<p class="text-center">(c) 2018</p>
</body>
</html>
```

⇒ On comprend mieux l'utilité de la base et de l'***héritage*** Twig entre templates.

7. Les liens entre pages

Créer une **nouvelle** route « /home/ » qui ouvre le template suivant (à créer)

```
{% extends 'base.html.twig' %}
{% block body %}
<h1>Home Page</h1>
Loremipsum dolor sit ametconsecteturadipisicingelit. Undeincidunt minima aspernatursuscipitperspiciatisharumveritatisdeleniti, alias rem a eveniet ad laboriosametnobispossimussintaperiamnulla animi?
{% endblock %}
```

Remplacer le code du Jumbotronde la**base**par ceci:

```
<div class="jumbotrontext-center">

<h1>Symfo Intro</h1>

<a href="{{ path('home') }}">Home</a> |

<a href="{{ path('salut') }}">Lien vers Salut</a>

<a href="">Lien vers Bonsoir</a>
</div>
```

Tester la nouvelle route, bien comprendre la fonction path () et compléter le dernier lien.

8. La gestion des formulaires

Symfony intègre un composant ***Form*** qui vous aide à gérer les formulaires.

❖ Installation

Exécutez cette commande pour installer la fonctionnalité de formulaire avant de l'utiliser :

```
composer require symfony/form
```

❖ Créer un formulaire simple

- Construisez une application simple de liste de tâches qui doit afficher des "tâches".

  ➔ ***Créer un formulaire.***

- Concentrez-vous d'abord sur la ***classe de Task générique*** qui représente et stocke les données pour une tâche unique :

```php
// src/Entity/Task.php
namespace App\Entity;
class Task
{
protected $task;
protected $dueDate;

public function getTask()
{
return $this->task;
}
public function setTask($task)
{
$this->task=$task;
}
public function getDueDate()
{
return $this->dueDate;
}
public function setDueDate(\DateTime$dueDate=null)
{
$this->dueDate=$dueDate;
}
}
```

C'est un objet PHP normal qui résout directement un problème au sein de *votre* application (par exemple, le besoin de représenter une tâche dans votre application).

❖ Construire le formulaire

- Dans Symfony, cela se fait en construisant un objet de formulaire, puis en le rendant dans un modèle. Pour l'instant, tout cela peut être fait depuis un contrôleur :

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

use App\Entity\Task;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Routing\Annotation\Route;

class TaskController extends AbstractController
{
/**
    * @Route("/form/", name="form")
   */
public function new(Request$request)
{
// creates a task and gives it some dummy data for this example
$task=new Task();
$task->setTask('Write a blog post');
$task->setDueDate(new\DateTime('tomorrow'));

$form=$this->createFormBuilder($task)
->add('task',TextType::class)
->add('dueDate',DateType::class)
->add('save',SubmitType::class,['label'=>'Create Task'])
->getForm();

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

- Cet exemple vous montre comment créer votre formulaire directement dans le contrôleur.
- Dans cet exemple,

✓ vous avez ajouté deux champs à votre formulaire, task et dueDate , correspondant aux propriétés task et dueDate de la ***classe de Task*** .

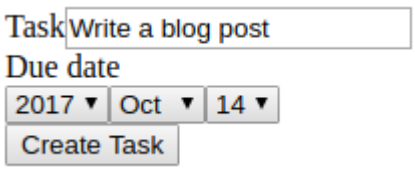
✓ Vous avez également affecté à chacun un "type" (par exemple, TextType et DateType), représenté par son nom de classe complet.

✓ Enfin, vous avez ajouté ***un bouton d'envoi*** avec une étiquette personnalisée pour l'envoi du formulaire au serveur.

❖ Rendre le formulaire

Maintenant que le formulaire a été créé, l'étape suivante consiste à le rendre. Ceci est fait en passant un objet "view" de formulaire spécial à votre modèle (notez le ***$form->createView()*** dans le contrôleur ci-dessus) et en utilisant un ensemble de fonctions d'aide de formulaire :

```
1  {# templates/task/new.html.twig #}
2  {{form(form)}}
```



- Avant de poursuivre, notez que le champ de saisie de la task rendue a la valeur de la propriété de ***$task objet $task*** (c.-à-d. "Rédiger un article de blog").

➔Il s'agit du premier travail d'un formulaire : extraire des données d'un objet et les traduire dans un format qui convient au rendu dans un formulaire HTML.

- Le système de formulaire est suffisamment intelligent pour accéder à la valeur de la propriété de la task protégée via les ***getTask()*** et ***setTask()*** de la ***classe Task***. Sauf si une propriété est publique, elle *doit* avoir une méthode "getter" et "setter" afin que le composant Form puisse obtenir et placer des données sur la propriété.

❖ Traitement des soumissions de formulaire

- Par défaut, le formulaire soumettra une demande POST au même contrôleur que celui-ci.
- Dans ce cas, le second travail d'un formulaire consiste à convertir les données soumises par l'utilisateur en propriétés d'un objet.

  ➔ Les données soumises par l'utilisateur doivent être écrites dans l'objet ***Form***.
- Ajoutez les fonctionnalités suivantes à votre contrôleur :

```
// ...
use Symfony\Component\HttpFoundation\Request;

public function new(Request$request)
{
// just setup a fresh $task object (remove the dummy data)
$task=newTask();

$form=$this->createFormBuilder($task)
->add('task',TextType::class)
->add('dueDate',DateType::class)
->add('save',SubmitType::class,['label'=>'Create Task'])
->getForm();

$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()){
// $form->getData() holds the submitted values
// but, the original `$task` variable has also been updated
$task=$form->getData();

// ... perform some action, such as saving the task to the database
// for example, if Task is a Doctrine entity, save it!
// $entityManager = $this->getDoctrine()->getManager();
// $entityManager->persist($task);
// $entityManager->flush();

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

return $this->render('task/new.html.twig',['form'=>$form->createView(),
]);
}
/**
   * @Route("/success", name="task_success")
   */
public  function successAction(){
return $this->render('task/success.html.twig');
}
```

```
42
```

Ce contrôleur suit un modèle commun de traitement des formulaires et comporte trois chemins possibles :

1. Lors du chargement initial de la page dans un navigateur, le formulaire est créé et rendu.

➔ ***handleRequest()*** reconnaît que le formulaire n'a pas été soumis et ne fait rien.

➔ ***isSubmitted()*** renvoie false si le formulaire n'a pas été soumis.

2. ***handleRequest()*** reconnaît cette information et écrit immédiatement les données soumises dans les propriétés task et dueDate de l'objet ***$task*** .

➔ Ensuite, cet objet est validé. S'il n'est pas valide (la validation est traitée dans la section suivante), ***isValid()*** renvoie false et le formulaire est à nouveau rendu, mais avec des erreurs de validation.

3. Lorsque l'utilisateur soumet le formulaire avec des données valides, les données soumises sont à nouveau écrites dans le formulaire, mais cette fois, ***isValid()*** renvoie true .

➔ Vous avez maintenant la possibilité d'exécuter certaines actions en utilisant l'objet ***$task*** (par exemple, en le conservant dans la base de données) avant de rediriger l'utilisateur vers une autre page (par exemple, une page "merci" ou "succès").

```
1
2   {# templates/task/success.html.twig#}
3   <h1>Success</h1>
4
```

❖ Validation du formulaire

- Avant d'utiliser la validation, ajoutez-la dans votre application :

```
composer requiresymfony/validator
```

➔ La validation est effectuée en ajoutant ***un ensemble de règles*** (appelées contraintes) à une classe.

➔ Pour voir cela en action, ajoutez des contraintes de validation afin que le champ de task ne puisse pas être vide et que le champ dueDate ne puisse pas être vide et doit être un objet DateTime valide.

- Annotations

```php
<?php
// src/Entity/Task.php
namespace App\Entity ;

use Symfony\Component\Validator\Constraints as Assert ;

class Task
 {
/**
    * @Assert\NotBlank
    */
public $task ;

/**
    * @Assert\NotBlank
    * @Assert\Type("\DateTime")
    */
protected  $dueDate ;

public function getTask ()
   {
return $this -> task ;
   }

public function setTask ( $task )
   {
$this -> task = $task ;
   }

public  function getDueDate ()
   {
return $this ->dueDate ;
   }

public  function setDueDate ( \DateTime$dueDate = null )
{
$this ->dueDate = $dueDate ;
   }
 }

?>
```

```twig
{# templates/task/new.html.twig #}
{{form_start(form,{'attr':{'novalidate':'novalidate'}} ) }}
{{form_widget(form)}}
{{form_end(form)}}
```