Framework Symfony

Installation du framework Symfony

Symfony est un framework d'application moderne qui permet d'écrire des applications web en php, tout en ayant une expérience de développement moderne et agréable.

Installation

Ajouter PHP au PATH

Copier le chemin de votre exécutable PHP : `bin\php\php-7.2.19-Win32-VC15-x64` à partir du répertoire d'installation de Laragon. Dans la recherche Windows, rechercher "*variables*" et ouvrir la première options. Cliquez sur "*Variables d'environnement systèmes*" et ajoutez le chemin de PHp à la variable PATH pour l'utilisateur et le système.


Capture-d’écran-2020-10-01-231653.png

Installer Composer & Symfony CLI

Composer est un gestionnaire de package pour PHP, (à la manière de maven / nugget / pip ...). Pour l'installer créer un dossier "*Composer*" dans votre dossier d'outils (le même où il y a le répertoire d'installation de Laragon, par exemple). Dans ce dossier, ouvrez une invite de commande :

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
'756890a4488ce9024fc62c56153228907f1545c228516cbf63f885e036d37e9a59d27d63f46af1d4d07ee0f76181c
7d3') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-
setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
echo @php "%~dp0composer.phar" %*>composer.bat
```

Téléchargez Symfony CLI via ce lien. Installez le.

Créer un projet Symfony sur PHP Storm

Créer un nouveau projet Composer :

- Choisir "*composer.phar*
- Renseigner le chemin de votre "*composer.phar* qui se trouve dans le dossier où vous avez installé Composer
- Renseigner votre interpréteur PHP qui est normalement déjà configuré donc vous n'avez qu'à le choisir dans le menu
- Dans "*Package*" choisissez `symfony/website-skeleton`

Capture-d’écran-2020-10-01-233202.png

Enfin attendez que le projet se génère.

Run le projet

Ouvrez le Terminal de PHP Storm et tappez la commande :

```
symfony server:start
```

Rendez vous ensuite à l'adresse http://127.0.0.1:8000 Si une page "welcome to symfony" s'affiche, tout fonctionne correctement, vous êtes prêt à coder !

Premiers pas avec symfony

Symfony est un framework MVC et qui utilise le paradigme orienté objet de PHP. Il permet de développer des applications complexes mais maintenables.

Controllers & Endpoints

Pour répondre à des requêtes HTTP, votre application a besoin de *Controllers* les controllers sont des classes qui contienent les méthodes endpoints. Une méthode endpoint est une méthode qui va gérer une requête sur une route particulière et avec une méthode HTTP particulière.

Premier Controller

Pour commencer, créez dans le dossier `src/Controller` de votre projet une nouvelle classe PHP :

2020-10-02-13_19_18-tuto-symfony-–-bas.png

Ensuite, pour configurer la route exécutez la commande suivante afin d'installer un package symfony :

```
composer require annotations
```

Vous pouvez ensuite écrire une méthode enpoint dans votre classe :

```
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class HelloWorldController
{
   /**
    * @Route("/helloworld")
    */
    public function HelloWorld(){
        return new Response("Hello World from Symfony !");
    }
}
```

L'annotation *@Route* est dans un commentaire mais elle est bien comprise par le framework, elle permet de lier votre méthode endpoint à la route `/helloworld`. Pour essayer tout ça, exécutez :

```
symfony server:start
```

puis rendez vous à : http://127.0.0.1:8000/helloworld

Les Templates

Pour être MVC, il faut dissocier la vue du Controller. Pour cela, nous allons utiliser des Templates afin de décrire les vues.

Dans un premier temps, exécutez la commande suivante pour installer le module de Symfony pour les templates :

```
composer require twig
```

Puis créez dans le dossier `src/templates` un nouveau dossier `hello` et dans ce dossier un fichier `hello.html.twig` avec le contenu suivant :

```
{# templates/lucky/number.html.twig #}
```

Ensuite, changez le code de votre Controller comme ceci, afin de rendre le template :

```
namespace App\Controller;



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

class HelloWorldController extends AbstractController
{
    /**
    * @Route("/helloworld")
    */
    public function HelloWorld(){

        $name = "Shepard";
        return $this->render("hello/hello.html.twig", [
            "name" => $name
```

```
        ]);
    }
}
```

Notre Controller étends désormais la classe `AbstractController` afin de pouvoir accèder aux méthodes de rendu de template. On utilise donc la méthode `render` avec en paramètre le nom de notre template ainsi que des données à afficher via un tableau associatif. Enfin, ajouter cette ligne à votre template pour afin d'afficher les données en question :

```
<h1>Hello {{ name }} !</h1>
```

Lancez ensuite votre application et allez sur l'URL de l'endpoint pour constater le rendu de votre template !

A propos des Routes

Nommage

Afin de faciliter le debuggage et la génération d'URLs, il est recommander de nommer ses URLs avec l'attribut `name` :

```
    /**
    * @Route("/helloworld", name="hello")
    */
    public function HelloWorld(){ ... }
```

Méthode HTTP

En définissant la Route d'une méthode endpoint, vous pouvez filtrer par méthode HTTP de la façon suivante :

```
    /**
    * @Route("/helloworld",name="hello" methods={"GET"})
    */
    public function HelloWorld(){ ... }
```

Cela est très utile car cela aide à séparer automatique la logique de présentation et la logique de modification d'un formulaire par exemple.

Paramètres de Route

Il est possible de passer des paramètres dans une route. Attention à ne pas confondre avec les paramètre d'URL qui se trouve après un `?` et sont séparés par des `&`. Un paramètre de route est

une partie dynamique de l'URL, que le client utilise pour passer un paramètre au serveur. Souvent, il s'agit du nom ou identifiant de la ressource concernée par la requête. Par exemple dans une application gèrant un blog, la route /posts/6 concernerait le poste de blog avec l'identifiant 6.

Pour déclarer un paramètre de route, procédez ainsi :

```
 /**
     * @Route("/blog/{id}", name="blog_show", methods = {"GET"})
     */
    public function show(int $id) { ... }
```

Les Controller Symfony

Symfony repose sur le modèle MVC et utilise donc des Controllers. Nous en avons vu les bases dans la partie précédente, mais nous allons revenir dessus dans cette partie pour présenter certaines fonctionnalités.

Génération d'URLs et Redirections

Dans la partie précédente, nous avons nommé nos routes. C'est ici que cela va devenir utile. En effet on peut utiliser le nom d'une route pour générer une URL de cette route, en remplissant ses paramètres.
Reprennons cet exemple de route paramètrée :

```
 /**
     * @Route("/blog/{id}", name="blog_show", methods = {"GET"})
     */
    public function show(int $id) { ... }
```

On peut donc générer une URL qui correspond à cette route comme ceci :

```
$url = $this->generateUrl('blog_show', ['id' => 10]);
```

On peut également faire des redirections vers des routes :

```
 return $this->redirectToRoute('blog_show', ['id' => 10]);
```

Ou vers des URLs :

```
  return $this->redirect('http://symfony.com/doc');
```

Session

On peut également, via un Controller Symfony, accèder à la session et y lire/écrire des données. Pour cela, il suffit de prendre en paramètre de la méthode endpoint un paramètre de type `SessionInterface` qui sera fourni par le framework.

```
 /**
     * @Route("/auth/login", name="login", methods = {"POST"})
```

```
     */
    public function login(SessionInterface $session) { ... }
```

Il s'agit d'un tableau associatif avec le lequel vous pouvez interagir via les méthodes `set` et `get` :

```
 /**
     * @Route("/auth/login", name="login", methods = {"POST"})
     */
    public function login(SessionInterface $session) {
      $session->set('key', 'value');
      $value = $session->get('key');
    }
```

L'objet Request

L'objet Request qui peut être fourni en paramètre d'une méthode endpoint par le framework, permet d'accèder aux données de la requête :

```
public function myEndpoint(Request $request){ ... }
```

Il permet de récupérer des paramètres du Body de la requête (équivalent $_POST) :

```
$request->request->get('myParam');
```

Ou des paramètres GET (équivalent $_GET) :

```
$request->query->get('myUrlParam');
```

Vous pouvez aussi récupérer un fichier dans le Body de la requête (équivalent $_FILES) :

```
 $request->files->get('myFile');
```

Retourner un fichier

Au lieu de retourner un Template rendu, vous pouvez aussi retourner un fichier qui sera téléchargé au client :

```
return $this->file('/chemin/du/fichier.pdf');
```

Les Templates Twig

Twig est un moteur de template HTML qui permet d'écrire des templates de pages HTML avec une syntaxe plus légère et plus lisible que du PHP Vanilla.

Les trois syntaxes principales de twig sont :

- `{{ ... }}` pour afficher une variable passée au template ou évaluer une expression
- `{% ... %}` pour afficher du contenu en suivant une logique comme des conditions ou des boucles
- `{# ... #}` pour afficher des commentaires qui ne seront pas rendus dans le HTML

Afficher des variables

Pour afficher une variable, utilisez la syntaxe `{{ ... }}` :

```
<p> Hellow {{ name }}<p/>
```

Si votre variable est une objet ou un tableau associatif, vous pouvez utiliser l'opérateur `.` pour accèder au champs d'un objet (publics ou exposé par getter / setter) ou à une valeur d'un tableau associatif :

```
<p> Hellow {{ user.name }}<p/>
```

Twig vous protège des attaques type faille XSS car il s'occupe d'échapper les caractères HTML.

Conditions

Pour afficher des conditions, on peut faire un simple if :

```
{% if user.isLoggedIn %}
     Hello {{ user.name }}!
{% else %}
     You are not logged in
{% endif %}
```

Boucles

On peut afficher le contenu de colletions en utilisant des boucles :

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

Réutiliser un Template

On peut importer un Template dans un autre pour factoriser du contenu (et éventuellement lui passer des paramètres) :

```
{{ include('block/menu.html.twig', {user: user}) }}
```

Héritage et Layout

Twig permet à un Template d'hériter d'un autres. Cela est très utile notamment quand on a un squelette de page à répéter sur beaucoup d'autres avec un contenu différent. Le Template parent va donc définir des `block` nommés qui pouront être remplis par les templates enfant.

Par exemple avec le template parents suivant qui correspond à un squelette générique de page HTML :

```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
            <title>{% block title %}{% endblock %} - Website</title>
        {% endblock %}
    </head>
    <body>
        <div id="content">{% block content %}{% endblock %}</div>
        <div id="footer">
            {% block footer %}
                Legal notices : ...
            {% endblock %}
        </div>
    </body>
</html>
```

Un exemple d'enfant :

```
{% extends "base.html" %}

{% block title %}Home{% endblock %}
{% block head %}
    <meta name="description" content="My website ...">
{% endblock %}
{% block content %}
    <h1>Home</h1>
    <p class="important">
        Welcome to my homepage.
    </p>
{% endblock %}
```

Il est également possible pour un block dans un enfant d'appeller le contenu du parent pour l'ajouter à son propre contenu au lieu de le remplacer :

```
 {{ parent() }}
```

Doctrine ORM

Doctrine est un framework de Mapping Relationnel-Objet (ORM en anglais). Une tel framework permet de traduire des données représentée dans un modèle orientée objet (utilisé dans les langages de proframmation OO) vers un modèle relationnel (utilisée des les systèmes de gestion de bases de données relationnels). Les ORMs permettent de gagner beaucoup de temps dans une application qui manipule des données, en déchargeant le programmeur d'une grande quantité de code très redondant.

Installation de Doctrine ORM

Installez le package Doctrine via Composer :

```
composer require symfony/orm-pack
composer require --dev symfony/maker-bundle
```

Ensuite, pour configurer les accès à la base de donnée, ouvrez le fichier `.env` à la racine de votre projet et cherchez le paramètre `DATABASE_URL`, et assignez lui la valeur suivante :

```
mysql://db_user:db_password@localhost:3306/db_name?serverVersion=5.7
```

en remplaçant `db_user`, `db_password`, et `db_name` par les valeurs correspondant à votre base.

Créer une Entity

Une Entity est une classe PHP dont les données seront persistées dans dans la base de donnée par Doctrine. Pour en créer, il existe un utilitaire en ligne de commande. Pour y faire appel, utilisez :

```
php bin/console make:entity
Class name of the entity to create or update:
> Product
New property name (press <return> to stop adding fields):
> name


Field type (enter ? to see all types) [string]:
> string
```

```
Field length [255]:
> 255


Can this field be null in the database (nullable) (yes/no) [no]:
> no


New property name (press <return> to stop adding fields):
> price


Field type (enter ? to see all types) [string]:
> integer


Can this field be null in the database (nullable) (yes/no) [no]:
> no


New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

L'outil en ligne de commande va ensuite vous demander les champs de votre entité un par un, ainsi que les informations à propos de vos champs (type, taille, etc ...).

Quand vous aurez fini, l'outil va vous créer une classe sous `src/Entity`:

```
namespace App\Entity;
use App\Repository\ProductRepository;
use Doctrine\ORM\Mapping as ORM;


/**
```

- ```
  @ORM\Entity(repositoryClass=ProductRepository::class)
  /
  ```
  ```

```
class Product
{
/*

 ◦ @ORM\Id()

 ◦ @ORM\GeneratedValue()

 ◦ @ORM\Column(type="integer")
   */
   private $id;


/**

 ◦ @ORM\Column(type="string", length=255)
   */
   private $name;


/**

 ◦ @ORM\Column(type="integer")
   */
   private $price;


public function getId(): ?int
{
return $this->id;
}

// ... getter and setter methods
}
```

Vous pouvez constater que les infos que vous avez données à l'outil ont été converties en annotations, qui vous ensuite permettre à l'ORM dans sérialiser la classe dans une base de donnée relationnelle.

Migrations

Pour pouvoir utiliser l'ORM, il faut d'abord synchroniser le schéma de votre base de donnée avec vos Entity PHP. Pour cela il faut créer puis exécuter une migration.

Créer une migration :

```
php bin/console make:migration
```

Exécuter une migration :

```
php bin/console doctrine:migrations:migrate
```

Au moment de la migration, Symfony vas exécuter le SQL nécessaire pour mettre à jours son schéma de façon à gérer les Entities.

Persister des données

Pour faire appelle à la couche de persistence dans le controller, il faut faire appel à l'Entity Manager de doctrine. Vous pouvez t accèder dans un controlleur comme ceci :

On peut ensuite créer un objet Entity :

```
$product = new Product();
$product->setName('Vis en acier inoxydable');
$product->setPrice(1999);
$product->setDescription('6mm de diamètre');
```

On peut ensuite persister l'Entity :

```
$entityManager->persist($product);
```

Et enfin commiter la transaction :

```
$entityManager->flush();
```

Requêter des données

Pour récupérer des données via Doctrine, cela se passe avec le Repository de l'Entity, que vous obtenez ainsi :

```
$repository = $this->getDoctrine()->getRepository(Product::class);
```

Vous pouvez ensuite récupérer des Entities par Id :

```
$product = $repository->find($id);
```

Vous pouvez ensuite filtrer sur un ou plusieurs propriétés avec la méthode `findOneBy()` et en lui passant un tableau clé-valeur avec en clé le nom du champs à filtrer et en valeur la valeur sur laquelle filtrer. Exemple :

```
$product = $repository->findOneBy([
    'name' => 'Keyboard',
    'price' => 1999,
]);
```

Mettre à jours des données

Pour mettre à jours une Entity, il suffit de la récupérer avec le Repository, de la modifier puis de commiter les changements avec `$entityManager->flush()` :

```
$entityManager = $this->getDoctrine()->getManager();
$product = $entityManager->getRepository(Product::class)->find($id);
$product->setName('Nouveau nom');
$entityManager->flush();
```