



# Cours de Symfony

May the force be with you !

# Sommaire

- Définition de Symfony
  - La structure du projet
  - Les Contrôleurs
  - Les Vues
  - Les modèles
  - Les Formulaires
  - Les Services
  - L'authentification
- 
- Travaux Pratique
- 
- Bilan

# Définition de Symfony

Symfony est un framework de développement web qui permet aux développeurs de construire des applications web plus rapidement en fournissant un ensemble de composants réutilisables et une structure de travail commune.

Il est développé par la société française "Sensiolabs", la première version de ce Framework est sortie en 2005. Ce framework repose sur le design pattern MVC.

## Framework PHP

Un framework PHP définit une structure générale pour la création d'applications web et offre un contrôle sur les différents composants et flux de travail de l'application.

## Librairie PHP

C'est une collection de fonctionnalités ou de classes qui peuvent être utilisées pour étendre les fonctionnalités de l'application.

## Le modèle MVC

- Modèle: Il représente les modèles de données et est en lien avec les ORM qui gère les relations avec la BDD.
- Vue: C'est la partie visible de l'application pour les utilisateurs, comme les pages HTML, le CSS et les scripts JavaScript.
- Contrôleur: Il gère les interactions entre le modèle et la vue. C'est dans le contrôleur que tous les traitements ont lieu.

# Pourquoi utiliser Symfony ?

Symfony fournit un cadre complet qui inclut des fonctionnalités telles que la gestion des routes, la gestion des bases de données, la sécurité, la gestion des formulaires, etc. En utilisant Symfony, les développeurs peuvent se concentrer sur la logique métier de leur application plutôt que sur les tâches répétitives de développement.



## Gain de temps

- Composants réutilisables
- Structure définie

## Sécurité

Séparation des Responsabilités  
Point d'accès centralisé



## Documentation

- Communauté active
- Documentation complète

## Testing

Inclut des outils de test intégrés pour assurer la qualité du code et faciliter le développement d'applications fiables



Il existe d'autres avantages à utiliser un Framework comme Symfony, comme la Flexibilité, la Performance, ou la Maintenabilité.

# Environnement de Travail

Prérequis :

- PHP >= 8.2
- Composer 2
- Wamp / Lamp / Xamp
- **Symfony CLI**

**Symfony CLI** : c'est un outil en ligne de commande pour les développeurs qui utilisent le framework Symfony. Il permet d'automatiser certaines tâches courantes telles que la création de nouveaux projets, la génération de code, la gestion des dépendances, etc.

Pour installer Symfony CLI (Pour Command-Line Interface) :

## sous Windows :

Pour installer Symfony CLI sous Windows, il vous faudra au préalable installer scoop, un gestionnaire de paquets pour Windows.

Puis lancer la commande suivante :

```
scoop install symfony-cli
```

## sous Linux // Mac :

via wget:

```
wget https://get.symfony.com/cli/installer -O - | bash
```

via curl:

```
curl -sS https://get.symfony.com/cli/installer | bash
```

via Homebrew (Homebrew requis):

```
brew install symfony-cli/tap/symfony-cli
```

# Environnement de Travail

Si "Symfony CLI" est correctement installé, vous pouvez lancer la commande:

```
symfony help
```

Cette commande vous affichera votre version Symfony et la liste des commandes disponibles.

---

Il est nécessaire de les vérifier avant de démarrer un nouveau projet ou de mettre à jour un projet existant.

Afin de valider que tous les prérequis sont installés et configurés correctement, vous pouvez lancer la commande suivante:

```
symfony check:requirements
```

```
[OK]  
Your system is ready to run Symfony projects
```

# Environnement de Travail

## Création d'un projet symfony :

Grâce à Symfony CLI, nous sommes désormais capables de créer un projet grâce à la commande suivante.

```
symfony new `Nom_du_projet`
```

Il est possible de personnaliser l'installation d'un projet avec les options suivantes:

```
--version="X.X.XX"
```

Permet de spécifier la version

```
--webapp
```

Créer un projet complet avec toutes les fonctionnalités nécessaires pour développer une application web

```
--no-git
```

Cela indique à Symfony CLI de ne pas initialiser un dépôt Git pour le nouveau projet.

```
--demo
```

Permet de créer un projet de démonstration avec des exemples de code.

Afin de connaître toutes options d'une commande, vous pouvez faire :

```
symfony help `command`
```

Exemple :

```
symfony help new
```

# Environnement de Travail

Pour lancer votre serveur pour le projet Symfony, vous devez faire la commande suivante:

```
symfony serve
```

Vous pouvez aussi utiliser l'option `-d`, pour lancer votre serveur pour le lancer en fond.

Il est désormais possible d'accéder à votre projet en vous à l'adresse suivante: 127.0.0.1:8000

Si le port 8000 n'est pas disponible, votre serveur prendra directement le suivant.

Pour couper votre serveur, il faudra faire :

```
symfony server:stop
```





# La structure de Symfony



bin

Si vous avez réalisé une installation avec l'option `--webapp`, vous trouverez les deux fichiers suivant dans ce dossier:



console



phpunit

Le fichier `console` est un script de ligne de commande (CLI) utilisé pour exécuter les tâches du projet Symfony. Vous pouvez accéder à la liste des commandes disponible grâce à:

`php bin/console`

ou

`symfony`

Le fichier `phpunit` est un script de ligne de commande (CLI) qui permet de lancer les tests unitaires d'une application Symfony en utilisant le framework de tests unitaires PHPUnit.

Vous pouvez exécuter vos test à l'aide de la commande suivante: `php bin/phpunit`

# La structure de Symfony

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock



config

Il contient les fichiers de configuration de votre application au format **Yaml\***.

Les fichiers de configuration sont généralement découpés en plusieurs fichiers distincts pour une gestion plus claire et plus facile des différentes configurations du projet.

Il est également possible de définir des configurations spécifiques à des environnements tels que le développement, la production, etc... avec la syntaxe suivante:

```
when@dev  
when@prod  
when@test
```

Cela permet de séparer les configurations en fonction de l'environnement, facilitant ainsi la gestion et le déploiement de l'application.

**Yaml** (Yet Another Markup Language): C'est une alternative populaire aux formats de configuration plus complexes tels que XML et JSON.

Dans YAML, les données sont représentées sous forme de clés et de valeurs, séparées par des deux-points et organisées en blocs indentés.

# La structure de Symfony

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock

## migrations

Ce dossier contient les scripts de migration de base de données. Les migrations de base de données sont utilisées pour gérer les modifications apportées à la structure de la base de données d'une application au fil du temps.

Pour rappel, il est interdit de modifier directement la BDD. Il faut toujours passer par des migrations.

Une migration décrit les modifications à apporter à la base de données et peut être exécutée pour mettre à jour la base de données.

Les migrations peuvent également être annulées pour restaurer la base de données à un état antérieur.

Créer une migration : `symfony console make:migration`

Checker le status de nos migrations : `symfony console doctrine:migrations:status`

Lancer une migrations : `symfony console doctrine:migrations:migrate`

# La structure de Symfony



A vertical list of files and folders in a Symfony project. Folders are represented by a folder icon and files by a document icon. The items are: bin, config, migrations, public, src, templates, tests, translations, var, vendor, .env, composer.json, and composer.lock.

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock

## public

Il contient les fichiers qui sont accessibles directement depuis le navigateur web. C'est le point d'entrée pour les requêtes HTTP et le contenu qui est servi aux utilisateurs.

On y retrouve le fichier `index.php` qui a pour but de créer une nouvelle instance de **Kernel**.

Le Kernel est le cœur d'une application Symfony. Il s'agit d'une classe qui définit le fonctionnement général de l'application et qui est utilisée pour traiter les requêtes HTTP.

Le Kernel est responsable de :

- Charger les différents fichiers configurations de l'application.
- Enregistrer les services nécessaires pour le fonctionnement de l'application.
- Enregistrer les routes de l'application et les lier à des contrôleurs.
- Gérer les erreurs et les exceptions pendant l'exécution de l'application.

# La structure de Symfony



C'est dans ce dossier que 80% de votre code va se trouver car on y retrouve :

- Vos Modèles dans le dossier `Entity`.
- Vos Contrôleurs dans le dossier du même nom.

On va également trouver un dossier Repository qui va contenir l'ensemble des requêtes vers la BDD que l'on pourra appeler depuis un contrôleur.

C'est aussi ici que se trouve la classe Kernel dont nous avons parlé précédemment.



Il contient toutes les Vues de notre application, au format twig.

C'est donc ici que la structure des pages HTML de votre application va se trouver.

Le format twig permet d'y ajouter des variables récupérer de vos contrôleurs et des instructions conditionnelles.

# La structure de Symfony

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock


## translations

Il sert à stocker les fichiers de traduction pour les différentes langues prises en charge par l'application.

L'utilisation de fichiers de traduction permet une meilleure flexibilité dans la prise en charge des différentes langues et une maintenance plus facile des traductions.


## var

On y trouve les dossiers suivants:

 **cache** Symfony va venir stocker certaines informations ici pour charger plus rapidement.



```
symfony console cache:clear
```

 **log** Permet de récupérer l'historique des logs de l'application, on peut également les récupérer dans la console avec :

```
symfony server:log
```

# La structure de Symfony



Il contient toutes les dépendances externes utilisées par l'application, telles que les librairies PHP. Il est généré lors de l'installation de ces dépendances en utilisant un outil de gestion de paquets, comme Composer.

Il est interdit de modifier directement les fichiers du dossier `'vendor'`, car ces modifications seront perdues lors de la prochaine mise à jour des paquets.

# La structure de Symfony

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock



Il sert à stocker toutes les informations qui vont varier en fonction de l'environnement, comme les informations nécessaires pour se connecter à la base de données:

```
DATABASE_URL="mysql://root:root@127.0.0.1:3306/app?serverVersion=5.7.24 &charset=utf8"
```

Étant donné que ce fichier peut contenir des données sensibles, il est recommandé de créer une copie de ce fichier qui s'appellera `.env.local`.

Votre application utilisera par défaut les variables d'environnement présentes dans `.env.local`. Il ne faudra pas oublier ce dernier car c'est maintenant lui qui contient des données sensibles, mais vous pourrez oublier le fichier `.env` en y indiquant toutes les variables nécessaires pour faire fonctionner votre application avec des valeurs par défaut.

```
DATABASE_URL="TYPE://USERNAME:PASSWORD@URL:PORT/app?serverVersion=X.X.XX &charset=utf8"
```

C'est dans ce fichier que vous allez définir si vous êtes en `prod` ou en `dev`:

```
APP_ENV=dev
```



# La structure de Symfony

- bin
- config
- migrations
- public
- src
- templates
- tests
- translations
- var
- vendor
- .env
- composer.json
- composer.lock

 composer.json

Il regroupe toutes les dépendances de votre projet. Il indique les paquets PHP que votre projet a besoin pour fonctionner et les versions de ces paquets que vous souhaitez utiliser.

 composer.lock

Il contient les informations exactes sur les versions des paquets installés. Il est utilisé pour garantir que toutes les installations sont reproductibles et que les dépendances sont installées de manière cohérente.

Il est généré automatiquement par Composer lors de l'installation ou de la mise à jour de packages.

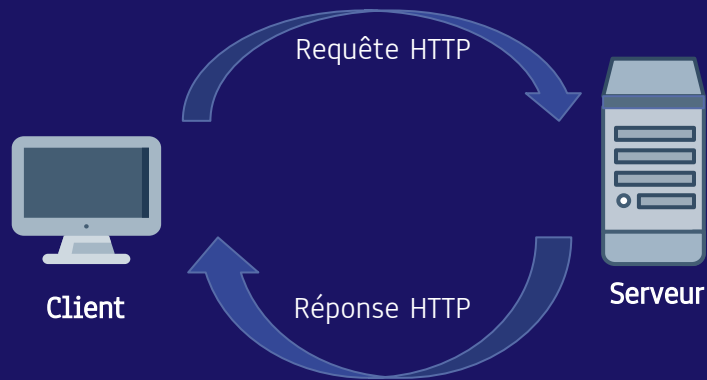
# Les contrôleurs

## Rappel: Les requêtes HTTP

Une requête HTTP (HyperText Transfer Protocol) est un message envoyé à un serveur Web pour demander une action ou une information. Elles peuvent être utilisées pour afficher des pages Web, soumettre des formulaires en ligne, envoyer des données à une API, etc.

Une requête HTTP se compose généralement de plusieurs parties, telles que :

- Le verbe HTTP (GET, POST, PUT, DELETE, etc.), qui définit l'action demandée
- L'URL cible, qui spécifie la ressource demandée
- Les en-têtes HTTP, qui fournissent des informations supplémentaires sur la requête, telles que le type de contenu accepté ou la préférence de langue
- Le corps de la requête, qui peut contenir des données supplémentaires envoyées avec la requête, telles que les données d'un formulaire Web.



# Les contrôleurs

## Créer un Contrôleur

```
symfony console make:controller
```

Vous allez devoir saisir le nom de votre contrôleur. La bonne pratique veut que nom de votre contrôleur sort formaté de la manière suivante:

```
ExempleController
```

Cette commande va vous créer un contrôleur avec une méthode 'index' et un fichier twig dans vos template, qui sera directement lié à votre contrôleur.

Au dessus de votre méthode index, vous allez trouver une annotation, c'est la route à utiliser pour accéder à cette méthode

ExempleController.php

```
class ExempleController extends AbstractController
{
    #[Route('/exemple', name: 'app_exemple')]
    public function index(): Response
    {
        return $this->render( view: 'exemple/index.html.twig', [
            'controller_name' => 'ExempleController',
        ]);
    }
}
```

<https://localhost:8000/exemple>

**Hello ExempleController! **

This friendly message is coming from:

- Your controller at [src/Controller/ExempleController.php](#)
- Your template at [templates/exemple/index.html.twig](#)

# Les contrôleurs

## Gérer les routes

Le système de route dans Symfony contrôle la correspondance entre les URL de votre application et les actions de vos contrôleurs. Chaque route est associée à une URL spécifique et à une méthode dans un contrôleur.

Il est possible de récupérer les éléments passer en paramètre de l'URL dans le contrôleur.

Exemple :

```
#[Route('/exemple/{id}', name: 'show_exemple')]
public function showExemple($id): Response
{
    return new Response( content: 'Exemple : '.$id);
}
```

Cette commande permet de lister toutes les routes de votre application :

```
symfony console debug:router
```

Il est conseillé d'utiliser les attributs `requirements` & `methods` d'affiner et sécurisé vos routes:

```
#[Route('/exemple/{id}', name: 'show_exemple',
    requirements: ['id'=>'\d+'],
    methods: ['GET']
)]
public function showExemple($id): Response
{
    return new Response( content: 'Exemple : '.$id);
}
```

# Les vues

## Introduction


Comme nous l'avons vu précédemment, nous affichons la réponse envoyée par le contrôleur.

Nous allons donc faire en sorte que notre méthode retourne une Vue et pour ce faire nous allons utiliser la méthode `render()` du `AbstractController`. Nous allons pouvoir y ajouter également des paramètres de la manière suivante :

```
return $this->render( view: 'exemple/index.html.twig', [
    'controller_name' => 'ExempleController',
]);
```

## Les variables

Pour accéder à nos variables dans un fichier twig, il faudra utiliser des doubles accolades.

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

# Les vues

## L'Instruction (if, else)

Voici la syntaxe à utiliser pour réaliser une condition *`if else`* :

```
{% if characters[length > 0 %}  
  <p>Voici une liste de personnages:</p>  
  <ul>  
    <li>{{ characters[0] }}</li>  
    <li>{{ characters[1] }}</li>  
    <li>{{ characters[2] }}</li>  
  </ul>  
  
{% else %}  
  <p>Aucun personnage disponibles</p>  
{%endif %}
```



Attention les opérateurs tels que *`&&`* ou *`||`* ne fonctionnent pas pour ces conditions, il faudra utiliser *`and`* ou *`or`*.

Il est également possible de faire un *`elseif`* avec la même syntaxe.

[Lien vers la doc](#)



# Les vues

## L'Instruction for

Voici la syntaxe à utiliser pour réaliser une condition `for` :

[Lien vers la doc](#)



```
{% if characters|length > 0 %}
  <p>Voici une liste de personnages:</p>
  <ul>
    {% for index, character in characters %}
      <li>{{ index }} : {{ character }} </li>
    {% endfor %}
  </ul>
{% else %}
  <p>Aucun personnage disponibles</p>
{% endif %}
```

A l'intérieur d'une boucle `for`, vous pouvez accéder à une variable `loop`.

loop.index  
loop.index0  
loop.first  
loop.last  
loop.length

L'index de l'itération à partir de 1  
L'index de l'itération à partir de 0  
Retourne `true` si c'est le premier  
Retourne `true` si c'est le dernier  
Le nombre total d'itération

## L'Instruction set

```
{% set name = 'Ellen Ripley' %}
<h1>Hello {{ name }}!</h1>
```

[Lien vers la doc](#)



# Les Vues

## Utiliser des vues

Si vous souhaitez utiliser ou afficher un élément sur différentes pages de votre application, il va falloir utiliser la balise `include`.

```
{% include 'template_à_inclure.html.twig' %}
```

Cela permet d'inclure un template dans un autre template. Elle permet de séparer le code HTML en plusieurs morceaux, ce qui peut être très utile pour organiser et maintenir plus facilement le code, cela permet de ne pas dupliquer le code.



# Les Vues

## Les filtres et les fonctions

[Lien vers la doc](#)



**Les filtres:** Ce sont des transformations que vous pouvez ajouter à une variable dans le modèle pour changer sa valeur. Par exemple, le filtre "length" peut être utilisé pour récupérer la taille d'un tableau, ou encore le filtre "date" va nous permettre de modifier l'affichage d'un objet Datetime.

```
{{ tableau|length }} // Affiche le nombre d'élément
```

```
{{ date("d/m/Y" )}} // Affiche aux format Jour/Mois/Année
```

**Les fonctions :** Elles sont plus complexes et permettent de créer et manipuler de nouvelles variables. L'une des plus utiles est la fonction "path()" qui permet de générer les URL.

```
{{ path("nom_de_route", {"param":valeur}) }}
```

[Lien vers la doc](#)



**Créer filtres et fonctions:** Il est possible de créer de nouveaux filtres et fonctions grâce à la commande suivantes:

```
symfony console make:twig-extension
```

# Les Vues

## Ajouter du CSS et du JS

Vous allez vous rendre dans le point d'entrée de votre application (/public) et créer un dossier "css" et un dossier "js".

Dans le dossier css, créer un fichier "style.css".

Dans le dossier js, créer un fichier "app.js".

Nous allons utiliser le paquet "symfony/asset" afin d'accéder à ces fichiers. Ce paquet est déjà installé si vous avez installé votre projet avec l'option "--webapp", sinon pour l'installer, il vous faudra passer par composer.

```
composer require symfony/asset
```

Grâce à ce paquet nous allons accéder à nos fichiers, comme ci-dessous:

```
<link href="{{ asset('css/style.css') }}" rel="stylesheet">
```

```
<script type="text/javascript" src="{{ asset('js/app.js') }}"></script>
```



Il n'est pas recommandé d'importer ces fichiers dans les blocs : car il seront écrasés si cet bloc est appelé sur une autre page, à moins que vous n'utilisiez la fonction "parent()" sur chacune d'entre elles.

```
{% block stylesheets %}{% endblock %}  
{% block javascripts %}{% endblock %}
```

Je vous conseille d'installer une librairie comme bootstrap afin de gérer le style de votre application.

[Lien vers la doc](#)



# Les Entités et les ORM

**Définition:** Les entités sont des objets qui représentent des données stockées dans une base de données. Les ORM (Object-Relational Mapping) sont des outils qui permettent de faire le lien entre les objets en PHP et les tables dans une base de données relationnelle.

## Création de la base de données

Afin d'être en mesure de créer une base de données depuis Symfony, il faut que votre "MySQL" soit lancé et que les informations relatives à votre base de données soit renseignés dans le fichier .env.local dans la variables DATABASE\_URL comme ci dessous:

```
DATABASE_URL="mysql://root:root@127.0.0.1:3306/app?serverVersion=5.7.24 & charset=utf8"
```

Cette variable sera utilisé par le fichier de configuration "doctrine.yaml"

```
symfony console doctrine:database:create
```

[Lien vers la doc](#)



# Les Entités et les ORM

## Création d'une entité

Afin de créer une entité, vous pouvez lancer la commande suivante:  
Cela va également vous créer un Repository dédié à cette entité.

```
symfony console make:entity
```

Une fois le nom de votre entité saisie, vous allez devoir spécifier toutes les propriétés de votre classe avec à chaque fois:

- Le nom
- Le type et sa longueur
- "Nullable" ou non



Inutile de spécifier l'id

Cela va également vous créer automatiquement vos accesseurs et mutateurs et vos ORM.

# Les Entités et les ORM

## Création d'une migration de mon entité

Afin de créer une migration pour une entité récemment créée, vous pouvez lancer la commande suivante:

```
symfony console make:migration
```

Votre fichier de migrations va comporter deux fonctions:

- La fonction 'up()' qui appliquera les modifications de la migration à la base de données.
- La fonction 'down()' supprimera les modifications de la migration à la base de données.

Pour appliquer les modifications à la base de données, il vous faudra lancer la commande:

```
symfony console doctrine:migrations:migrate
```

Vous pouvez également consulter la liste de vos migrations à l'aide de la commande:

```
symfony console doctrine:migrations:list
```

# Les Entités et les ORM

## Enregistrement d'une entité

L'enregistrement d'une entité a en général lieu dans un contrôleur. Il faut importer l'entité dans le contrôleur.

Ensuite nous allons créer un objet basé sur la classe de l'entité et utiliser les mutateurs précédemment créés pour donner des valeurs aux propriétés de notre objet :

```
#[Route('/character/add', name: 'add_exemple')]
public function addCharacter(EntityManagerInterface $manager): Response
{
    $character = new Character();
    $character->setAge( age: 2);
    $character->setName( name: 'Jones');
    $character->setFirstname( firstname: 'Johnsy');

    $manager->persist($character);
    $manager->flush();

    dump($character);
}
```

Afin de pouvoir enregistrer notre objet, nous allons avoir besoin d'un "EntityManager" avec les méthodes "persist" et "flush".

**"Persist"** est utilisé pour enregistrer les données en mémoire. Cette méthode indique au gestionnaire d'entités (EntityManager) de Symfony de tenir compte d'une nouvelle entité ou de toutes les modifications apportées à une entité existante.

**"Flush"**, quant à lui, exécute réellement la requête SQL correspondante pour enregistrer les données en base de données.

# Les Entités et les ORM

## Générer des données avec les DataFixtures

Ils permettent de définir des jeux d'essais pour tester notre application, ce qui peut être plus rapide et plus pratique que de saisir manuellement des données dans une interface utilisateur.

Une fois en place, elles peuvent être facilement rechargées pour tester différents scénarios de données.

Pour cela nous allons avoir besoin d'installer le paquet "orm-fixtures" :

```
composer require orm-fixtures -dev
```

Une fois le paquet installé, nous créons notre fichiers de fixtures :

```
symfony console make:fixtures
```

C'est dans ce fichier que nous allons générer nos données de test. Pour lancer les Fixtures, vous pouvez utiliser:

```
symfony console doctrine:fixtures:load
```

```
public function load(ObjectManager $manager): void
{
    $faker = \Faker\Factory::create('fr_FR');

    for($i = 0; $i < 10; $i++){
        $character = new Character();
        $character->setAge($faker->numberBetween('int1: 0', 'int2: 100'));
        $character->setName($faker->lastName());
        $character->setFirstname($faker->firstName());

        $manager->persist($character);
        $manager->flush();
    }
}
```

[Lien de Faker](#)



# Les Entités et les ORM

## Créer des requêtes avec les repository

Les repositories sont des classes spéciales dans Symfony qui sont créées en même temps que vos entités et qui sont utilisées pour effectuer des requêtes sur la base de données. Elles permettent de déléguer une grande partie de la logique de requêtage à une classe dédiée, ce qui peut aider à organiser et structurer votre code de manière plus claire et maintenable. Les repositories sont souvent utilisées pour les entités, qui représentent des tables dans la base de données.

Par défaut, il comprennent déjà une liste de fonctions récurrentes chez les développeurs comme "find()" ou "findAll()".

### Exemple 1:

```
#[Route('/exemple', name: 'app_exemple')]
public function index(CharacterRepository $characterRepository): Response
{
    $characters = $characterRepository->findAll();

    return $this->render( view: 'exemple/index.html.twig', [
        'controller_name' => 'ExempleController',
        'characters' => $characters
    ]);
}
```

### Exemple 2:

```
#[Route('/exemple', name: 'app_exemple')]
public function index(CharacterRepository $characterRepository): Response
{
    $characters = $characterRepository->findBy(
        [], ['age' => 'ASC']
    );

    return $this->render( view: 'exemple/index.html.twig', [
        'controller_name' => 'ExempleController',
        'characters' => $characters
    ]);
}
```



# Les Entités et les ORM

## Créer vos propres requêtes

C'est dans le repository que vous allez mettre toutes vos requêtes les plus récurrentes. Voici un exemple d'utilisation:

```
public function findAllSenior(int $age = 60)
{
    $query = $this->createQueryBuilder( alias: 'character');
    $query->select( select: 'character');
    $query->where( predicates: 'character.age > :age');
    $query->setParameter( key: 'age', $age);

    return $query->getQuery()->getResult();
}
```

[Lien vers la doc](#)



# Les Entités et les ORM

## ParamConverter

Dans certains cas, il est possible de récupérer plus rapidement vos objets en utilisant le ParamConverter.

```
#[Route('/exemple/{id}', name: 'show_exemple',
    requirements: ['id'=>'\d+'],
    methods: ['GET'])]
public function showExemple(Character $character): Response
{
    return $this->render('view: exemple/show.html.twig', [
        'character' => $character
    ]);
}
```

Ici on récupère notre objet en utilisant un ParamConverter.

[Lien vers la doc](#)



# Les Entités et les ORM

## Créer des relations (ManyToOne...)

Il existe plusieurs types de relations que vous pouvez définir, tels que :

- **OneToOne**: relation entre deux entités où une entité a une seule relation avec une autre entité
- **OneToMany**: relation entre deux entités où une entité à plusieurs relations avec une autre entité
- **ManyToOne**: relation entre deux entités où plusieurs entités ont une seule relation avec une autre entité
- **ManyToMany**: relation entre deux entités où plusieurs entités ont plusieurs relations avec une autre entité

Afin de définir les relations entre vos entités, vous pouvez passer par commande:

```
symfony console make:entity
```

Ensuite vous allez devoir ajouter le champ qui sera votre *"foreign key"* et lui donner en type la relation désirée.  
En fonction de la relation

# Les Formulaires

## Créer un formulaire

Un formulaire dans Symfony est un objet qui permet de générer et de valider automatiquement un formulaire HTML. Il est utilisé pour recueillir des données auprès de l'utilisateur, généralement pour soumettre des informations à une base de données. Symfony fournit un certain nombre de composants pour construire et gérer des formulaires, notamment **FormBuilder**.

```
no usages
#[Route('/character/add', name: 'add_exemple')]
public function addCharacter(Request $request, EntityManagerInterface $manager): Response
{
    $form = $this->createFormBuilder()
        ->add('name', type: TextType::class, ['label' => 'Nom de famille'])
        ->add('firstname', type: TextType::class, ['label' => 'Prénom'])
        ->add('age', type: IntegerType::class, ['label' => 'Âge'])
        ->getForm();

    return $this->renderForm(views: 'exemple/add.html.twig', [
        'form' => $form
    ]);
}
```

Ainsi vous allez pouvoir accéder à votre formulaire dans la vue avec la syntaxe suivante:

```
{{ form(form) }}
```

[Lien vers la doc](#)



# Les Formulaires

## Modifier votre formulaire

Vous allez pouvoir modifier votre formulaire pour qu'il corresponde mieux à vos attentes dans la vue:

*Exemple:*

```
{{ form_start(form) }}

    {{ form_label(form.name, null, {"label_attr": {'class': 'form-label'}}) }}
    {{ form_widget(form.name, {"attr": {'class': 'form-control'}}) }}

    {{ form_label(form.firstname, null, {"label_attr": {'class': 'form-label'}}) }}
    {{ form_widget(form.firstname, {"attr": {'class': 'form-control'}}) }}

    {{ form_label(form.age, null, {"label_attr": {'class': 'form-label'}}) }}
    {{ form_widget(form.age, {"attr": {'class': 'form-control'}}) }}

    {{ form_rest(form) }}
    <div class="text-center mt-2">
        <button type="submit" class="btn btn-primary">Sauvegarder</button>
    </div>

{{ form_end(form) }}
```

# Les Formulaires

```
no usages
#[Route('/character/add', name: 'add_exemple')]
public function addCharacter(Request $request, EntityManagerInterface $manager): Response
{
    $form = $this->createFormBuilder()
        ->add( child: 'name', type: TextType::class, ['label' => 'Nom de famille'])
        ->add( child: 'firstname', type: TextType::class, ['label' => 'Prénom'])
        ->add( child: 'age', type: IntegerType::class, ['label' => 'Âge'])
        ->getForm();

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()){
        $character = new Character();

        $character->setName($form->get('name')->getData());
        $character->setFirstname($form->get('firstname')->getData());
        $character->setAge($form->get('age')->getData());

        $manager->persist($character);
        $manager->flush();

        return $this->redirectToRoute( route: 'app_exemple');
    }

    return $this->renderForm( view: 'exemple/add.html.twig', [
        'form' => $form
    ]);
}
```

## Enregistrer votre formulaire

Afin d'enregistrer votre formulaire, vous allez devoir utiliser récupérer la requête de votre "form" avec Request.

N'oubliez pas de l'importer :

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

Vous pouvez effectuer plusieurs actions sur le formulaire comme vérifier que le formulaire est valide avant d'enregistrer son contenu dans une entité.

N'oubliez pas de faire une redirection à la fin de l'enregistrement.

# Les Formulaires

## Rattacher un formulaire à une entité

Avec Symfony, il est possible de créer des formulaire en passant par les entités, pour faire cela, il faut faire:

```
symfony console make:form
```

La convention pour le nom d'un formulaire est "EntityType"

Exemple :

```
no usages
#[Route('/character/add', name: 'add_exemple')]
public function addCharacter(Request $request, EntityManagerInterface $manager): Response
{
    $character = new Character();
    $form = $this->createForm( type: CharacterType::class, $character);

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()){

        $manager->persist($character);
        $manager->flush();

        return $this->redirectToRoute( route: 'app_exemple');
    }

    return $this->renderForm( view: 'exemple/add.html.twig', [
        'form' => $form
    ]);
}
```

```
no usages
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add( child: 'name')
        ->add( child: 'firstname')
        ->add( child: 'age')
    ;
}
```

# Les Formulaires

## Formulaire d'édition

Avec Symfony, il est possible d'attribuer plusieurs routes à une seule méthode du contrôleur, voici un exemple qui permet de gérer l'édition et l'ajout avec une seule méthode:

```
#[Route('/character/add', name: 'add_exemple')]
#[Route('/character/{id}/update', name: 'update_exemple')]
public function addCharacter(
    Character $character = null,
    Request $request,
    EntityManagerInterface $manager,
    InsultDetector $insultDetector,
    FlashBagInterface $flashBag
): Response
{
    if($character == null){
        $character = new Character();
    }

    $form = $this->createForm( type: CharacterType::class, $character);

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()){

        if ($insultDetector->checkForInsults($character->getName().' '.$character->getFirstname())) {
            $flashBag->add( type: 'danger', message: "Le nom ne doit pas contenir d'insulte");
        } else {
            if($character->getId() == null){
                $manager->persist($character);
            }

            $manager->flush();
            return $this->redirectToRoute( route: 'app_exemple');
        }
    }

    return $this->renderForm( view: 'exemple/add.html.twig', [
        'form' => $form
    ]);
}
```



# Les “Asserts”

**Définition:** Les Asserts sont des vérifications effectuées pour s'assurer que certaines conditions sont remplies dans le code. En Symfony, elles sont utilisées pour vérifier les entrées de données et garantir que les valeurs attendues sont présentes et correctes. Les assertions peuvent être utilisées pour vérifier la validité des champs de formulaire, des variables, des objets, etc. Si l'assert échoue, une exception est levée et un message d'erreur est affiché.

Nous allons définir nos asserts dans nos entités, pour ce faire nous devons les importer:

```
use Symfony\Component\Validator\Constraints as Assert;
```

Voici un exemple sur le champ 'name':

```
#[ORM\Column(length: 255)]
#[Assert\Length(
    min: 5,
    max: 50,
    minMessage: 'Le champ nom ne doit pas être inférieur à {{ limit }} caractères',
    maxMessage: 'Le champ nom ne doit pas dépassé {{ limit }} caractères',
)]
private ?string $name = null;
```

Il est possible de récupérer les erreurs dans le formulaire grâce à “form\_errors”:

```
{{ form_label(form.name, null, {'label_attr': {'class': 'form-label'}}) }}
{{ form_widget(form.name, {'attr': {'class': 'form-control'}}) }}
<div class="invalid-feedback">
    {{ form_errors(form.name) }}
</div>
```

# Les Services

**Définition:** Les services dans Symfony sont des objets dédiés à une tâche spécifique. Ils sont généralement enregistrés dans le conteneur de services de Symfony et peuvent être injectés dans d'autres objets de l'application, tels que les contrôleurs ou les modèles, pour être utilisés. Ce sont des classes d'outils que l'on peut facilement réutiliser.

Nous avons également la possibilité de créer nos propres services. Il faut les faire dans le dossier "Service" dans "src".

```
namespace App\Service;

class InsultDetector
{
    /**
     * @usage
     */
    private array $insults = ['idiot', 'stupide', 'dégueulasse', 'inutile'];

    /**
     * @usage
     */
    public function checkForInsults(string $message): bool
    {
        foreach ($this->insults as $insult) {
            if (strpos($message, $insult) !== false) {
                return true;
            }
        }

        return false;
    }
}
```

```
$form->handleRequest($request);
if($form->isSubmitted() && $form->isValid()){

    if ($insultDetector->checkForInsults($character->getName().' '.$character->getFirstname())) {
        $flashBag->add( type: 'danger', message: "Le nom ne doit pas contenir d'insulte");
    } else {
        $manager->persist($character);
        $manager->flush();
        return $this->redirectToRoute( route: 'app_exemple');
    }
}
```

Vous pouvez voir l'ensemble de vos services avec la commande:

```
symfony console debug:autowiring
```

# L'authentification

La première chose à faire si l'on veut gérer de l'authentification, c'est créer une table user.

```
symfony console make:user
```

N'oubliez pas de réaliser vos migrations.

Nous allons créer notre premier utilisateur, pour cela nous allons utiliser un DataFixtures.

Pour Hasher le mot de passe, nous allons utiliser :

```
symfony console security:hash-password
```

Maintenant que nous avons un utilisateur, nous allons commencer l'authentification.

```
symfony console make:auth
```

partir d'ici il faudra seulement définir la page vers laquelle vous serez redirigé après vous être authentifié.

Dans le fichier "security.yaml", vous allez pouvoir définir vos routes accessibles par role.

```
access_control:
  - { path: ^/admin, roles: ROLE_ADMIN }
  # - { path: ^/profile, roles: ROLE_USER }
role_hierarchy:
  ROLE_ADMIN: ROLE_USER
```



```
{% if is_granted('ROLE_ADMIN') %}
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
```

# Travaux pratiques

## Consigne

Le but de ce TP est de créer une application web avec le framework Symfony. L'application sera un gestionnaire de projets avec des fonctionnalités CRUD (Create, Read, Update, Delete) sur l'entité Project. De plus, l'application devra inclure un système d'authentification avec un rôle "ADMIN" & un rôle "USER".

Chaque projet doit avoir une catégorie parmi les suivantes :

- Projet personnel
- E-Commerce
- Application mobile
- Veille technique

Chaque projet doit avoir une ou plusieurs technos

- PHP
- JS
- Python
- Etc...

L'application devra avoir un back office avec EasyAdmin, pour gérer les entités (CRUD) de votre projet.

Un projet doit contenir les informations suivantes :

- Un nom de projet
- Une miniature pour illustrer le projet
- Une description détaillée
- Un lien (nullable)
- Une date de début
- Une date de fin (nullable)
- Une date de création.
- Une date de modification.

---

Pour le CSS de l'application, vous pouvez prendre un template bootstrap ou tailwind.

Un administrateur pourra accéder au back-office EasyAdmin.

# Fin

Merci de m'avoir écouté =)