Cours : Symfony 4.2

0. À propos du cours

> **Durée du cours :** 4 à 5 jours

> **Requirements :**
>
> - PHP > 7.1
> - Composer
> - Git

> **Recommandations :**
>
> - Un IDE (Visual Studio Code, PHPStorm...)
> - Des modules d'autocomplétion pour :
> -   - PHP (VSCode : PHP Intelephense)
> -   - Twig (VSCode : TWIG Pack)
> -   - Yaml (VSCode: YAML)
> -   - .env (VSCode : DOTENV)
> - Configurez les "tabulations" de votre IDE en mode `Spaces: 4` (4 caractères "espace" plutôt qu'un caractère "tab")
> - **Consultez la doc fournie à chaque chapitre !**

1. Installation

> Documentation : Installing & Setting up the Symfony Framework

1.1. Créer un nouveau projet

Créez un nouveau projet Symfony avec la commande suivante :

```
composer create-project symfony/website-skeleton nom-du-project
cd nom-du-project
```

Nous pouvons également utiliser `composer create-project symfony/skeleton` qui contient les éléments minimaux d'une application web (microservices, APIs...) et nous laisse le choix d'installer les outils dont nous aurions besoin, néanmoins `symfony/website-skeleton` contient tous les outils nécessaires pour bien commencer.

1.2. Lancer le serveur intégré

L'installation nous a donné tout une boîte à outils en CLI : `php bin/console`. Pour lancer le serveur depuis le dossier de l'app : `php bin/console server:run`

1.3. Configurer l'application

Vous pouvez lancer `php bin/console about` pour consulter la configuration actuelle de l'application.

Pour la modifier, modifiez le fichier `.env`.

2. Routes et controllers, introduction

> Documentation Routing

2.1. Créer une première route

Il existe plusieurs façons de déclarer des routes dans Symfony :

routes.yaml

```
# config/routes.yaml

about:
    path: /a-propos
    controller: App\Controller\PagesController::about
```

Dans ce cas, nous nommons (c'est simplement un nom interne à l'application) notre route `about`, et nous indiquons à Symfony de se diriger vers le contrôleur `PagesController` et la méthode `about` lorsque l'utilisateur va sur l'URI `/a-propos` (donc l'URL `http://127.0.0.1:8000/a-propos` par exemple).

2.2 Créer le controller

Il faut donc créer un `PagesController` : d'après le fichier `composer.json`, dans la key `psr-4`, on sait que le namespace `App/` pointe vers le dossier `src/`.

Nous allons donc créer un contrôleur dans `src/Controller` :

```
// src/Controller/PagesController.php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;

class PagesController {

    public function about() {
        return new Response('Hello world!');
    }
}
```

> **Important :** Notez bien l'usage de l'alias de `Symfony\Component\HttpFoundation\Response`! Utilisez bien l'autocomplétion de votre IDE afin de bien importer les alias nécessaires (tip : commencez à taper le nom d'une classe et choisissez avec les flèches du clavier la classe souhaitée, aidez vous du namespace pour savoir quelle est la bonne classe à inclure !).

Et voilà, nous avons fait notre premier Hello world.

2.3. Annotations

Une autre manière de créer des routes dans Symfony sont les annotations. Toujours dans `PagesController.php` :

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

    /**
    * @Route("/home", name="home")
    */
    public function home() {
        return new Response ('Bienvenue sur la page d\'accueil !');
    }
```

Notez bien l'utilisation de `Symfony\Component\Routing\Annotation\Route` !

Les annotations permettent de déclarer les routes juste au dessus de la méthode qui prendra en charge l'URI. C'est donc plus pratique car tout est au même endroit, mais plus dispersé que d'avoir toutes les routes dans un fichier *.yaml.

2.4. Routes et paramètres

Nous pouvons écouter des paramètres dans les routes en ajoutant des `{variables}` dans l'URL :

```
/**
* @Route("/users/{userId}/books/{bookId}", name="user_book")
*/
public function users(int $userId, int $bookId) {
    return new Response ('Vous consultez le livre #' . $bookId . ' de
l\'utilisateur numéro '. $userId);
}
```

2.5. Paramètres : wildcards

Nous pouvons utiliser des wildcards dans les routes, c'est à dire une chaîne de caractères quelconque que l'on peut valider par des expressions régulières (regex) :

```
/**
 * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
 */
```

Dans ce cas là, la route n'acceptera que les cas où l'argument `{page}` correspond à la regex `\d+` (= valeurs numériques uniquement).

2.6. Paramètres: valeur par défaut

Si jamais je souhaite pouvoir accéder à l'URI `/blog/` malgré tout, avec une valeur par défaut (par exemple je veux que par défaut, `page = 1`), je peux le passer en paramètre de l'action :

```
/**
 * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
 */
public function list($page = 1)
{
    // ...
}
```

2.7. Request et Response

Nous utilisions les Request et Response du package HttpFoundation pour gérer les requêtes et réponses HTTP. Grâce à l'autowiring (autochargement des classes), nous pouvons directement appeler la requête dans les arguments de la méthode :

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

// ...

    /**
```

```
     * @Route("/post-user", name="create_user", methods={"POST"})
     */

    public function create(Request $request) : Response
    {
        dump($request);
    }
```

Plusieurs nouveaux concepts ici :

- Nous avons importé les classes `Request` et `Response`
- Grâce à l'autowiring, nous pouvons appeler l'object `Request $request` dans notre action (la méthode `create()`)
- Nous avons précisé les méthodes autorisées pour cette route avec `methods={"POST"}`
- Nous avons indiqué le type de retour de la fonction (`: Response`)

Et voilà, l'object `Request $request` issu de l'envoi d'un formulaire par exemple disponible à l'utilisation ! Nous pouvons accéder aux valeurs POST par exemple avec `$request->get('name');`.

2.8. Routing avancé : localisation des URI

```
/**
 * @Route({ "fr": "/a-propos", "en": "/about-us"}, name="about")
 */
public function about()
{
    // ...
}
```

2.9. Lister les routes

Un outil de la console nous permet de lister toutes les routes déclarées (pratique notamment lorsque l'on utilise les annotations !) : `php bin/console debug:router`

3. Vues et Twig

> Documentation : Creating and Using Templates

3.1. Afficher une vue

Maintenant que nous avons vu le routeur et le controller, nous allons voir comment retourner une vue depuis un controller.

Symfony utilise Twig comme moteur de template : grâce à au container d'injection de dépendances, il peut être disponible directement auprès du contrôleur :

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
// ...
class PagesController extends AbstractController {

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

En héritant de `AbstractController`, nous pouvons dorénavant utiliser la méthode `render()` qui prend en premier paramètre le fichier Twig à utiliser.

3.2. Créer une vue

Les vues se trouvent dans le dossier `/templates` (défini par dans `twig.default_path` le fichier de configuration `twig.yaml`).

Nous allons donc créer le fichier `templates/home.html.twig` :

```
{# templates/home.html.twig #}

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

{% block title %}Page Home !{% endblock %}

{% block body %}
<div class="example-wrapper">
    <h1>Hello Page Home !</h1>
</div>
{% endblock %}
```

Détaillons ce code :

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

Cette ligne indique à Twig d'utiliser le fichier de template `base.html.twig`, qui se trouve aussi dans `/templates`.

Si on regarde le fichier `base.html.twig`, on voit qu'il s'agit d'un fichier HTML classique avec des éléments `{%block ... %}{% endblock %}`.

Les blocs du template de "extends", `base.html.twig`, sont les éléments "extensibles" : ils peuvent contenir une valeur par défaut, comme `{% block title %}Welcome!{% endblock %}` ou rien du tout.

En fait, nous allons remplir leur contenu par les fichiers de vues comme `home.html.twig`, comme avec ce morceau de code :

```
{% block body %}
<div class="example-wrapper">
    <h1>Hello Page Home !</h1>
</div>
{% endblock %}
```

Le code HTML généré pour le client sera donc `base.html.twig` avec ce code ci-dessus dans son bloc `body` !

3.3. Passer des variables à la vue

Nous pouvons évidemment passer des variables à la vue depuis le contrôleur :

```
// PagesController.php
// ...

public function home() {

        $pageTitle = "Mon super site";

        $movies = [
            [
                "title" => "Inception",
                "length" => 135,
            ],
            [
                "title" => "Rocky",
                "length" => 126,
            ]
        ];

        return $this->render('home.html.twig', [
            'pageTitle' => $pageTitle,
            'movies' => $movies
        ]);
    }
```

Le deuxième argument de `render()` prend un tableau : la `key` est le nom de la variable passé à Twig, la `value` est le contenu de la variable.

Si, comme pour l'exemple ci-dessus, le nom des variables pour Twig et pour PHP ont le même nom, on peut rendre le code plus conscis avec `compact()` (doc PHP) :

```
return $this->render('home.html.twig', compact('title', 'movies'));
```

Nous pouvons maintenant utiliser les variables dans Twig :

```
{% block title %}Page : {{ pageTitle }} {% endblock %}

{% block body %}
<div class="example-wrapper">
    <h1>Hello, vous êtes sur la page {{ pageTitle}} !</h1>
</div>
{% endblock %}
```

3.4 Loops

Pour afficher les données d'un array ou d'une collection d'objects, nous pouvons utiliser la boucle `for` de Twig :

```
{% block body %}
    <h1>Films</h1>
    <ul>
        {% for movie in movies %}
            <li>{{ movie.title }} (Durée : {{ movie.length }} min)</li>
        {% endfor %}
    </ul>
{% endblock %}
```

3.5 If/else

Nous pouvons faire un affichage conditionnel en Twig :

```
{% if not user.subscribed %}
    <p>Vous n'êtes pas encore inscrit à la mailing list.</p>
{% endif %}
```

```
{% if temperature > 18 and temperature < 27 %}
    <p>It's a nice day for a walk in the park.</p>
{% endif %}
```

3.5 Filters

Nous pouvons modifier la donnée à la volée grâce aux filters (pipes) :

```
{{ 'bienvenue'|upper }} {# retourne : 'BIENVENUE' #}
```

3.6. Documentation

La documentation complète de Twig est disponible ici : documentation Twig.

4. Doctrine, Entities et Repositories

> Documentation : Databases and the Doctrine ORM OpenClassrooms : Gérez vos données avec Doctrine ORM`

4.1. Création de la base de données

Doctrine est un ORM (Object-relationnal Mapping), qui implémente le pattern Data Mapper. Concrètement, le Data Mapper synchronise un object dans le PHP avec la base de données, ce qui nous donne une couche Model performante dans notre MVC.

Pour commencer, vous devrez configurer votre base de données dans le fichier `.env` qui se trouve à la racine du projet :

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
```

Remplacez les valeurs de `db_user` et `db_password` par les valeurs qui correspondent à votre configuration.

> Attention : si votre mot de passe est vide, laissez bien les deux-points avant le @, exemple : `DATABASE_URL=mysql://root:@127.0.0.1:3306/db_name`

Pour le champ `db_name`, vous pouvez en créer un nouveau : nous allons pouvoir le créer depuis l'outil `console` de Symfony !

Une fois configuré, ouvrez une console **dans le dossier du projet** et saisissez :

```
php bin/console doctrine:database:create
```

Et voilà, la base de données a été créée !

4.2. Création des entités

Nous allons créer des Entity : ce sont l'équivalent des Model du MVC, il s'agit de la classe qui mappera la table correspondante en base de données.

Pour cela, ouvrez une console et saisissez :

```
php bin/console make:entity Article
```

> **Attention :** les entités ont la première lettre en majuscules et sont au singulier.

Le CLI vous guidera pour créer les champs un par un : créez par exemple les champs suivants :

```
title (string, NOT NULL)
description (text, NOT NULL)
created_at (datetime, NOT NULL)
```

Une fois l'entité créée, nous pouvons aller la voir dans `src/Entity/Article.php`.

4.3. Migrations

L'entité est un mapping de notre base de données : c'est à dire que le fichier Entity correspond, grâce aux annotations `@ORM` notamment, à ce à quoi ressemble notre table en base de données.

Si nous souhaitons faire une modification dans les tables, nous avons deux manières de faire :

- Modifier le fichier `Entity/Article.php`
- Ajouter un champ grâce à `php bin/console make:entity Article` : le fait de reprendre le nom `Article` ici va éditer l'entity existante `Article` !

> **Attention :** Nous n'avons donc plus besoin de modifier la base de données directement dans PHPMyAdmin !

IMPORTANT: Maintenant que vous avez créé et éventuellement modifié votre fichier Entity, vous créerez un fichier *migration* :

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

Consultez le fichier créé qui se trouve dans `src/Migrations` : une migration est en fait une instruction de DB qui nous indique quoi faire par rapport à l'état de nos fichiers Entity : par exemple là, vous verrez dans la méthode `up()` un `CREATE TABLE Article ...`.

Ce qui se passe en fait : Doctrine, l'ORM de Symfony, va comparer l'état de la base de données actuellement et à quoi ressemblent les Entity ! Là en effet, on n'a pas de table `Article` en base de données mais on a une Entity `Article`... La migration nous propose donc un `CREATE TABLE`.

Il existe aussi une migration `down` (la méthode `down()` du fichier migration) : il s'agit de l'opposé de la migration `up` : en effet, si vous voulez revenir à l'état précédent de la base de données, plutôt que de vous souvenir de vos modifications, il sera possible de faire une migration dite `down` pour l'annuler !

4.4. Migrer les fichiers migrations

Maintenant que les fichiers migrations sont fait, c'est à dire les instructions à donner à la base de données, nous allons migrer ces fichiers afin que la base de données lance ces commandes SQL :

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

> Les deux commandes ci-dessus sont équivalentes, la seconde, plus courte, est simplement un alias.

Comme vous l'avez remarqué lors de la création de la base de données (4.1. Création de la base de données), une table `migrations` a été créée : elle va en simplement enregistrer la liste des fichiers de migration qui ont été exécutés afin de garder une trace de ce qu'il reste à faire !

En exécutant cette commande, les fichiers migrations restant à migrer vont donc être exécutés.

Il est très important de se souvenir du fonctionnement des migrations, rappel :

1. Modification du fichier `Entity` (avec `php bin/console make:entity Article` ou en modifiant le fichier à la main)
2. L'entité est modifiée, il faut persister ces changements en base de données : `php bin/console make:migration`
3. `make:migration` va simplement créer un fichier migration en comparant à quoi ressemble la base de données et à quoi ressemble le fichier Entity
4. Immédiatement, je peux persister les changements `php bin/console migrate`.

TRÈS IMPORTANT : Après une modification de l'entity, et surtout après un `make:migration`, exécutez systématiquement un `migrate` : en effet, cela vous évite de refaire un second `make:migration` qui n'aurait pas été migré et donc générer des erreurs, exemple **à ne pas faire** :

1. J'ajoute une nouvelle `Entity`, `User` par exemple
2. Je fais un `make:migration`
3. Le fichier migration créé ressemble à : `CREATE TABLE User...`
4. Je refais un `make:migration` au lieu d'un `migrate`: un autre fichier de migration se créée, et refait un `CREATE TABLE User...` (en effet, il n'y a toujours rien dans ma DB, Doctrine pense devoir refaire un `CREATE TABLE User` !)
5. Je migre un peu trop tard: `migrate` et... j'ai une erreur (en effet, j'aurai deux `CREATE TABLE User` au lieu d'un seul, MySQL lèvera une erreur)

> **Résumé:** Pour éviter ce problème facilement, il suffit juste de faire un `migrate` avant chaque `make:migration` afin d'exécuter les migrations précédentes s'il en restait à faire :

```
// Créer une migration :
php bin/console migrate # On migre les précédentes migrations éventuelles
php bin/console make:migration # On créée la nouvelle migration
php bin/console migrate # On migre la nouvelle migration
```

4.5. Constructeur et createdAt

Comme notre entité représente notre table en base de données, nous pouvons gérer les données comme tel : pour donner une valeur par défaut au champ `created_at`, nous pouvons créer un constructeur dans `Article.php` : les getters et setters sont déjà générés !

```
// Article.php
// ...

public function __construct() {
```

```
        $this->setCreatedAt(new \DateTime());
    }
```

4.6. Enregistrer une donnée : service Doctrine et EntityManager (CREATE)

> Un service est une classe qui remplit une fonction bien précise, accessible partout dans notre code grâce au container de services.

Dans une méthode d'un contrôleur, nous allons créer un nouvel objet `Article` et lui donner quelques données grâce aux setters.

```
$article = new Article();
$article->setTitle('Nouveau titre !');
$article->setContent('Lorem ipsum....');
```

Doctrine est le service qu va nous permettre de gérer la base de données et de persister les données en base de données, c'est à dire d'enregistrer l'objet créé en une ligne de la base de données. Il est accessible depuis le contrôleur comme n'importe quel autre service :

```
$doctrine = $this->getDoctrine();
```

Doctrine s'occupe de plusieurs choses : d'une part la connexion à la base de données (`$doctrine->getConnection($name))` récupère une connection à une base de données par exemple), et d'autre part de la partie `EntityManager`, c'est la partie ORM, qui va persister les données :

```
$entityManager = $doctrine->getManager();
```

Nous pouvons avoir plusieurs EntityManager: un par connexion à une base de données par exemple (dans le cas où vous gérez plusieurs BDD pour votre projet).

Nous allons donc persister les données (enregistrer l'object en tant que ligne de DB) grâce à l'EntityManager :

```
$entityManager->persist($article); // On prépare l'article à être
enregistré en BDD
$entityManager->flush(); // On execute effectivement la requête !
```

En résumé :

```


// On créée un nouvel object Article
$article = new Article();
$article->setTitle('Nouveau titre !');
```

```
$article->setContent('Lorem ipsum....');

// On récupère l'EntityManager du service Doctrine :
// Notez que le code est plus court que dans l'expliation ci-dessus !
$em = $this->getDoctrine()->getManager();

// On donne l'object en gestion à Doctrine pour qu'il "persiste" l'object,
c'est à dire qu'il prépare la requête
$em->persist($article);

// On execute effectivement la requête :
$em->flush();
```

Et voilà ! L'article est enregistré en base de données. On peut dorénavant **(sur le même object que ci-dessus !)**, faire un `$article->getId()` pour récupérer l'ID de l'objet nouvellement enregistré.

4.7. Lire des données (READ)

Lors de la création de notre entité, un fichier `Repository\ArticleRepository.php` a été créé : le repository est le fichier qui s'occupe de récupérer les données de la base de données.

Voici comment il s'utilise :

```
// On importe le repository de l'entity Article
$articleRepository = $this->getDoctrine()->getRepository(Article::class);

// Tous les articles
$articles = $articleRepository->findAll();

// Un article (par son ID)
$article = $articleRepository->find(43);

// Une collection d'articles (search par un champ)
$articles = $articleRepository->findBy(['title' => 'Hello title!']);
```

4.8. Lire des données: requêtes complexes avec le Repository et le QueryBuilder de Doctrine

> Documentation: Doctrine - Working with Query Builder

On peut bien sûr exécuter des requêtes plus complexes avec le repository, éditions par exemple le fichier `src/Repository/ArticleRepository.php`.

Le fichier contient deux exemples commentés, décommentons le premier exemple :

```
/**
 * @return Article[] Returns an array of Article objects
 */

public function findByExampleField($value)
```

```
{
    return $this->createQueryBuilder('a')
        ->andWhere('a.exampleField = :val')
        ->setParameter('val', $value)
        ->orderBy('a.id', 'ASC')
        ->setMaxResults(10)
        ->getQuery()
        ->getResult()
    ;
}
```

On voit comment est composée une requête avec le QueryBuilder, avec par exemple :

- l'ajout de paramètres : on a `$value` en paramètres de la méthode. On prépare la requête avec une clé `:val`dans le `andWhere()`, et on va ajouter le paramètre à la requête avec `setParameter(key, $var)`.
- `setMaxResults(10)` : permet de limiter les résultats... à 10 !

Pour utiliser cette requête, on peut l'appeler dans le contrôleur. Disons que nous l'avons renommée `findByName($name)` au lieu de `findByExampleField($value)`:

```
$articles = $articleRepository->findByName('sciences');
```

4.8. Mettre à jour (UPDATE)

Maintenant que nous savons lire une donnée et écrire une donnée, nous allons mixer les deux et faire une méthode d'update.

C'est aussi l'occasion de voir des notions nouvelles :

- Nous passons en argument à la méthode la requête qui vient du client, `Request $request`, afin de récupérer les données issues d'un formulaire
- Nous passons un paramètre à la route, `id`, un nom interne à l'application `articles_edit` et une liste de méthodes HTTP autorisées sur cette route `POST` (ce qui veut dire qu'aller sur `/articles/{id}/edit` depuis un navigateur en GET ne marchera pas !). Pour prendre en compte l'id, on doit le passer en argument à la méthode : on peut aussi forcer le type ! `Article $article`. Grâce à cela, Symfony s'occupera pour nous de récupérer l'article dont l'id est égal à `{id}`.

```
// @Route("/articles/{id}/edit", name="articles_edit", methods={"POST"})
public function update(Request $request, Article $article) {

}
```

Sans appeler l'article `$article` en paramètres avec `{id}` nous aurions aussi pu faire :

```
// @Route("/articles/{id}/edit", name="articles_edit", methods={"POST"})
public function update(Request $request, int $id) {

    $articleRepository = $this->getDoctrine()-
>getRepository(Article::class);
    $article = $articleRepository->find($id);

}
```

Maintenant que nous avons notre Entity `$article`, nous allons l'éditer et la flusher comme pour un insert :

```
// @Route("/articles/{id}/edit", name="articles_edit", methods={"POST"})
public function update(Request $request, Article $article) {

    // On met à jour l'article
    $article->setTitle('Nouveau titre mis à jour');
    // On récupère l'EntityManager et on met à jour (sans persister, juste
flush)
    $entityManager = $this->getDoctrine()->getManager();
    $entityManager->flush();
}
```

4.9. Supprimer un object (DELETE)

La suppression est très facile en utilisant tout ce que nous venons de voir :

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

5. Commandes make

5.1. make:controller

Vous pouvez créer un nouveau contrôleur avec la commande `make:controller PagesController`. Ce contrôleur contiendra une première page `index()` par défaut avec un template dans `templates/pages/index.html.twig` !

5.3. Automagique : make:crud !

On peut créer automatiquement un CRUD pour une entité (qui doit exister avant de faire la commande) : `make:crud Article`.

La commande va créer un `controller`, un ficher `Type` (le formulaire généré) et des vues dans `/template`.

> **Attention:** Il est très important de bien comprendre les fonctionnements que nous voyons de voir jusqu'à présent ! Bien que la commande `make:crud` fait "tout ça d'un coup", c'est important de comprendre

> tout ce que nous avons vu plutôt que d'utiliser des générateurs afin de savoir comment les débugger !

6. Forms

6.1. make:form

> Ce formulaire est aussi celui qui se trouve généré avec `make:crud` !

Vous pouvez créer un formulaire auto-généré (`Type`) pour une entité : Symfony lira l'Entity et créera le formulaire correspondant : `make:form Article`. Cela créera un fichier dans `src/Form/ArticleType.php`.

Pour intégrer le formulaire, il suffira ensuite de l'appeler dans le contrôleur de cette façon :

```
// /new est accessible en 2 méthodes:
// GET : pour AFFICHER le formulaire
// POST : pour TRAITER le formulaire

/**
* @Route("/new", name="product_new", methods={"GET","POST"})
*/
public function new(Request $request): Response
    {

    // CAS GET (affichage) :
        // On prépare l'article à créer avec le formulaire
        $article = new Article();

        // On prépare le formulaire : on utilise le service createForm
avec en arguments: le formulaire généré (ArticleType) et l'objet traité
par le formulaire ($article)
        $form = $this->createForm(ArticleType::class, $article);

    // CAS POST (traitement) :
        // On indique au formulaire de traiter la requête
        $form->handleRequest($request);

        // Si le formulaire a été envoyé et est valide, on le traite
        if ($form->isSubmitted() && $form->isValid()) {

            // On enregistre la donnée
            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($product);
            $entityManager->flush();

            // On redirige vers la page article_index
            return $this->redirectToRoute('article_index');
        }

    // CAS GET ou CAS POST SI FORMULAIRE INVALIDE (if ci-dessus) :
    // On affiche le formulaire
        return $this->render('product/new.html.twig', [
```

```
            'product' => $product,
            'form' => $form->createView(),
        ]);
    }
```

6.2. Styliser nos formulaires générés

Les formulaires autogénérés peuvent prendre le style Boostrap en modifiant `config/packages/twig.yaml` et en ajoutant l'attribut suivant :

```
twig:
    form_themes: ['bootstrap_4_layout.html.twig']
```

> **Attention**, ce sont bien 4 espaces et non pas une tabulation !

6.3. FormTypes

> Documentation Forms Documentation Form Types Reference

Les formulaires peuvent donc être créés de 3 façons :

- Par la commande `make:crud` qui génère entre autres le formulaire généré pour une Entity
- Par la commande `make:form` qui ne génère que le formulaire généré pour une Entity
- Directement à la main dans un fichier Type ou dans le controller

Voyons comment sont composés les formulaires générés dans Symfony, prenons par exemple un `LocationType` (formulaire d'ajout d'adresses) :

```
public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', TextType::class)
            ->add('street_number', IntegerType::class)
            ->add('street_name', TextType::class)
            ->add('zip', IntegerType::class)
            ->add('city', TextType::class)
            ->add('country', CountryType::class)
            ->add('longitude')
            ->add('latitude')
        ;
    }
```

On utilise une instance de `FormBuilderInterface` pour générer les formulaires.

Chaque champ est ajouté avec `add()` qui prend 3 arguments :

- le nom du champ
- la classe `Type` correspondante, qui va gérer le formulaire selon le type (DateTimeType, EmailType...)

6.4. Relations 1-N et N-N et formulaires

On peut ajouter une relation dans un formulaire, de sorte à ce que, par exemple, avec `Article 1-N Category`, nous ayons la liste des catégories dans un select !

```
//...
->add('category', EntityType::class, [
    'class' => Category::class, // Quelle classe est reliée au champ
category
    'choice_label' => 'name', // Quel champ de Category afficher dans le
select
])
//...
```

Dans le cas d'une relation N-N (`Tag N-N Article`), on aurait plutôt un select multiple :

```
//...
->add('tags', EntityType::class, [
    'class' => Tag::class,
    'choice_label' => 'name',
    'multiple' => true
])
//...
```

> **Attention :** Choisissez le bon cas d'usage selon votre relation (mettre un selct multiple ou non), sinon vous aurez un bug !

6.4. Validations

> Documentation : Validation Documentation : Constraints

Les formulaires peuvent être validés de plusieurs façons :

6.4.1. Validations par les annotations

Ces validations se font au niveau de l'entité, par exemple on rend ici unique le titre avec `UniqueEntity` et on limite la taille du titre à entre 2 et 50 caractères.

```
// ...
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @UniqueEntity("title")
 */
class Article {
```

```
     /**
     * @Assert\Length(
     *      min = 2,
     *      max = 50,
     *      minMessage = "Your first name must be at least {{ limit }}
characters long",
     *      maxMessage = "Your first name cannot be longer than {{ limit
}} characters"
     * )
     * @ORM\Column(type="string")
     */
    private $title;
// ...
```

6.4.2. Validation par les FormType

Cette fois, dans le fichier `src/Form/UserType` :

```
->add('nickname', TextType::class,[
    'constraints' => [
        new Length([
            'min' => 1,
            'minMessage' => 'Your nickname should be at least {{ limit }}
characters',
            'max' => 20,
            'maxMessage' => 'Your nickname should be maximum {{ limit }}
characters',
        ]),
    ],
])
```

7. Notions diverses

> Maintenant que nous avons vu comment créer un CRUD en Symfony, gérer les routes, le MVC..., il s'agit surtout d'apprendre des pratiques et techniques au cas par cas.

7.1. Rediriger vers une autre route

```


// Pour rediriger vers /articles/{id} (name="articles_show")
return $this->redirectToRoute('articles_show', [
        'id' => $article->getId()
    ]);
```

7.2. Entities : Relations

On peut ajouter une relation entre deux entités de la façon suivante :

```
php bin/console make:entity Article # On édite l'entité Article

# ATTENTION: Au pluriel ou au singulier en fonction de la relation !!!
# ATTENTION: On ne met pas l'id mais le nom de la relation !!!
New property name (press <return> to stop adding fields):
> category


# Vous pouvez taper directement le type de relation ou taper "relation"
pour avoir la liste des relations disponibles
Field type (enter ? to see all types) [string]:
> relation

# On parle bien de l'entité (singulier, première lettre majuscule)
What class should this entity be related to?:
> Category

What type of relationship is this? # ManyToOne, OneToMany, OneToOne,
ManyToMany

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne

# Accéder aux articles depuis une catégorie ?
Do you want to add a new property to Category so that you can
access/update Article objects from it - e.g. $category->getArticles()?
(yes/no) [yes]:
> yes

A new property will also be added to the Category class so that you can
access the related Article objects from it.

New field name inside Category [articles]:
> articles
```

Pensez à migrer :

```
migrate
make:migration
migrate
```

Dorénavant nous aurons accès depuis une entity à une autre. Par exemple, la catégorie depuis l'article :

```


// On prend le repository de Article
$articleRepository = $this->getDoctrine()->getRepository(Article::class);
```

```
// On récupère le premier article
$article = $articleRepository->find(1);

// On a accès à sa catégorie
$category = $article->getCategory(); // object Category
```

Les articles depuis la catégorie :

```
// On prend le repository de Category
$categoryRepository = $this->getDoctrine()-
>getRepository(Category::class);

// On récupère la catégorie Sciences
$article = $categoryRepository->findBy(['name' => 'Sciences']);

// On a accès à ses articles
$category = $article->getArticles(); // object Collection<Article>
```

7.3. Affichage des éléments d'une collection

On peut, dans la page d'une catégorie par exemple, afficher tous les éléments :

```
{% for article in category.articles %}

    <li>
        <a href="{{ path('article_show', { id: article.id }) }}">
            {{ article.title }}
        </a>
    </li>

{% endfor %}
```

> **Note :** Voyez comme nous avons passé un argument à la route `article_show`! En effet la route est quelque chose comme `/article/{id}` et c'est ici la manière de passer l'argument `{id}` avec `path()` dans Twig.

8. Security et Auth

> Documentation : Security

L'authentification peut être générée par Symfony en suivant une petite recette :

- On créée la classe User via le générateur
- On créée l'authenticateur
- On créée le formulaire d'enregistrement

8.1. User

Dans la console : `php bin/console make:user`

La console vous demandera quelques informations à propos de votre classe User (le nom, la clé unique...). Il faut noter qu'elle implémentera `UserInterface` de sorte à pouvoir fonctionner avec l'authentification de Symfony.

8.2. Authenticateur

Dans la console : `php bin/console make:auth` Pour les questions du CLI :

- Style of authentication : `Login Form Authenticator`
- Classname : `FormAuthenticator`
- Controller class: `SecurityController`

Et voilà, la route `/login` a été créée ainsi que le système d'authentification !

Vous devez modifier le fichier `src\Security\LoginAuthenticator` dans la méthode `onAuthenticationSuccess()` (vers la ligne 89) de la façon suivante :

```
// Supprimer la ligne suivante :
throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);

// Ajouter la ligne suivante :
return new RedirectResponse($this->urlGenerator->generate('some_route'));
```

> **Attention:** Assurez vous de mettre une route existante à la place de `some_route` !!! Il s'agit du nom de la route vers laquel on est redirigé après s'être loggué. L'espace membres ou l'accueil par exemple !

8.3. Route Logout

Pour ajouter la route Logout, nous devons :

8.3.1. Modifier security.yaml

Ajoutez la partie `firewalls/main/logout` de la façon suivante dans `/src/config/security.yaml` (attention, ce n'est que le bloc `logout` qu'il faut rajouter, le reste existe déjà !) :

```
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            guard:
                authenticators:
                    - App\Security\FormAuthenticator
            logout:
```

```
                path: /logout
                target: /
```

8.3.2. Ajouter une route dans routes.yaml

Ajoutez la route suivante dans `src/config/routes.yaml` :

```
logout:
    path: /logout
```

Et voilà, la route `/logout` sera accessible pour déconnecter l'utilisateur.

8.4. Registration Form

Dans la console : `php bin/console make:registration-form`

Nous allons générer le formulaire et le contrôleur de création de compte. Répondez les réponses par défaut au CLI.

Voilà, vous avez un formulaire généré dans la route `/register` !

8.5. Utiliser l'authentification dans le controller

Par défaut, nos utilisateurs ont tous un rôle `ROLE_USER` (défini dans `User::getRoles()`). Nous pouvons utiliser l'annotation `@IsGranted` pour bloquer l'accès à une route :

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
// ...
    /**
     * @IsGranted("ROLE_ADMIN")
     * @Route("/", name="location_index", methods={"GET"})
     */
    public function index(LocationRepository $locationRepository):
Response
    {
        return $this->render('location/index.html.twig', [
            'locations' => $locationRepository->findAll(),
        ]);
    }
```

8.5.1. À toutes les routes d'un contrôleur :

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
// ...
/**
 * @IsGranted("ROLE_USER")
```

```
 * @Route("/location")
 */
class LocationController extends AbstractController
{
    /**
     * @Route("/", name="location_index", methods={"GET"})
     */
    public function index(LocationRepository $locationRepository):
Response
    { /* ... */ }

    /**
     * @Route("/new", name="location_new", methods={"GET","POST"})
     */
    public function new(Request $request): Response
    { /* ... */ }
```

> Toutes les routes `/location` ne sont accessibles qu'aux utilisateurs logués (`ROLE_USER`).

8.5.2. Ou un mix des deux !

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
// ...
/**
 * @IsGranted("ROLE_USER")
 * @Route("/location")
 */
class LocationController extends AbstractController
{
    /**
     * @Route("/", name="location_index", methods={"GET"})
     */
    public function index(LocationRepository $locationRepository):
Response
    { /* ... */ }

    /**
     * @IsGranted("ROLE_ADMIN")
     * @Route("/new", name="location_new", methods={"GET","POST"})
     */
    public function new(Request $request): Response
    { /* ... */ }
```

> Toutes les routes `/location` ne sont accessibles qu'aux utilisateurs logués (`ROLE_USER`), de plus, `/location/new` n'est accessible qu'aux administrateurs (`ROLE_ADMIN`).

> **Attention:** N'oubliez pas le `use` pour pouvoir utiliser l'annotation !

8.6. Utiliser l'authentication dans Twig

On peut bien sûr vérifier l'authentification d'un utilisateur dans twig, par exemple :

```
{% if app.user %}
    <a href="{{ path('user_home') }}">Accédez à votre espace membre</a>
{% endif %}

{% if is_granted('ROLE_ADMIN') %}
    <a href="{{ path('admin_dashboard') }}">Accédez à votre espace
administrateur sécurisé !</a>
{% endif %}

{% if not is_granted('ROLE_AUTEUR') %}
    <p>Désolé, cet espace n'est accessible qu'aux auteurs !</p>
{% endif %}
```

9. Injection de services

Il existe 3 façons dans une classe d'injecter un service. Voyons par exemple comment injecter un Repository (on peut bien sûr en injecter plusieurs de la même manière s'il y a besoin de plus de dépendances !).

9.1. Injection par le constructeur

```
class ArticleController extends AbstractController {

    private $articleRepository;

    public function __construct(ArticleRepository $articleRepository) {
        $this->articleRepository = $articleRepository;
    }

    public function index() {

        $articles = $this->articleRepository->findAll();

        return $this->render('articles/index.html.twig, [
            'articles' => $articles
        ]);
    }
}
```

9.2. Injection par la méthode (autowiring)

> C'est ce que l'on fait quand on appelle `Request $request` dans une méthode !

```
class ArticleController extends AbstractController {
```

```
    public function index(ArticleRepository $articleRepository) {

        $articles = $articleRepository->findAll();

        return $this->render('articles/index.html.twig, [
            'articles' => $articles
        ]);
    }
}
```

9.3. Appel par le containeur de services

```
class ArticleController extends AbstractController {

    public function index() {

        $articleRepository = $this->getDoctrine()-
>getRepository(Article::class);

        $articles = $articleRepository->findAll();

        return $this->render('articles/index.html.twig, [
            'articles' => $articles
        ]);
    }
}
```