# Symfony : Comprendre le Framework !

### Tout ce qui sera écrit en orange seront des lignes de commande à faire dans le terminal (Windows)

### Quelques commandes pour se déplacer dans le terminal de Windows :

* **cd ou chdir ⬄ Affiche le nom du répertoire courant ou change le répertoire courant**
* **cls ⬄ Efface le contenu de la fenêtre d’invite de commande.**
* **copy ⬄ Copie un fichier d’un emplacement à un autre.**
* **del ⬄ Supprime un ou plusieurs fichiers**
* **dir ⬄Affiche la liste des fichiers et sous-dossiers contenus dans le répertoire.**
* **exit ⬄ Quitte le programme cmd.exe ou le script batch courant.**
* **md ou mkdir ⬄ Crée un répertoire ou un sous-répertoire.**
* **move ⬄ Déplace un fichier d’un dossier à un autre.**
* **rd ou rmdir ⬄ Supprime un répertoire.**
* **ren ou rename ⬄ Change le nom d’un fichier ou d’un ensemble de fichiers.**
* **php -v** ⬄ Permet de voir la version PHP installer sur notre machine
* **composer -V** ⬄ Pour vérifier la version de composer et donc si composer fonctionne bien , qu’il existe et qu’il est bien installer sur la machine.

Une fois que les 2 points précédent soit ok on peut donc commencer à découvrir Symfony.

# Créer une application Symfony avec composer.

Se rendre dans un dossier où l’ont veut créer le projet, on est pas obligé d’aller dans le dossier www de wamp car on ne va pas travailler avec apache pour faire tourner Symfony mais avec un serveur interne.

* **composer create-project symfony/website-skeleton demo**

Create-project ⬄ Créer une structure de dossiers/fichiers à partir d’un modèle.

Symfony/website-skeletin ⬄Modèle d’un projet de site web symfony.

Demo ⬄ Le nom de mon nouveau projet.

Je dois ensuite me rendre (avec le terminal) dans le dossier où j’ai créé mon application.

* **cd demo**

L’ouvrir avec mon éditeur de code (Visuel Studio code)

* **code .**

Une fois ouvert, je ne vais plus utiliser mon invite de commande externe, mais je peux maintenant commencer à utiliser directement au sein même de mon Visuel Studio Code pour taper toute sortes de commande.

Pourquoi toute sortes de commande ? Car Symfony c’est beaucoup de fichiers et beaucoup de codes. On peut tout faire à la main mais les développeurs de Symfony nous ont fournis la CLI ⬄ Command light interface. C’est un petit assistant qu’on peut appeler par des lignes de commandes et qui va nous aider à gérer notre application Symfony.

# Installer un serveur personnaliser pour lancer mon application Symfony

* **composer require server --dev ⬄** Que je ne vais utiliser que lorsque je vais développer, le jour où je vais passer mon site en prod, tu n’auras pas besoin d’installer cette librairie là.

Composer require [xxx]⬄ Permet d’installer une librairie et de l’utiliser dans le projet !

A la fin de l’installation, on va pouvoir lancer notre application dans notre navigateur.

* **php bin/console server:run**

Ça va lancer notre serveur sur localhost :8000

Notre application ne tourne donc pas sur le port 80, le port 80 c’est pour mon apache/wamp si je veux accéder à d’autres projets qui tourne sur wamp par exemple ou à phpmyadmin.

**ATTENTION Dépendamment de la version de Symfony utilisé, la commande précédente affichera une erreur mais on peut toujours lancer le serveur avec la commande suivante :**

**Depuis la version 4.4 de Symfony il faut Aller sur** [**https://symfony.com/download**](https://symfony.com/download) **et télécharger et installer le logiciel Symfony (Bouton Download). Re démarrer le pc, se rendre dans le terminal à l’endroit où se trouve le projet et on peut maintenant lancer le serveur avec :**

* **symfony server:start**

## Commençons par inspecter notre dossier.

On va s’intéressé particulièrement au dossier **src** qui va contenir l’entièreté des **codes** **php** de notre application ainsi qu’au dossier **template** qui lui va nous permettre de **gérer** nos **fichiers d’affichage**/rendu avec **twig**.

***Ouvrir un 2eme onglet de terminal dans Visuel Studio Code pour ce qui va suivre !***

Pour créer un controller qui va se retrouver au sein du dossier controller.

* **Php bin/console make :controller**

**make:controller** ⬄ **Permet de fabriquer un controller simplement !**

Choisir un nom pour la classe controller : 1ere lettre en Maj et le reste en camelCase

* **BlogController**

**Un gros bouton vert avec le mot « Success ! » apparait.**

Ma ligne de commande à créer dans le dossier des controllers un fichier qui s’appelle **BlogController.php** et qui **contient** une **classe** et dans notre dossier templates un dossier qui s’appelle **blog** avec déjà un premier fichier qui s’appelle **index.html.twig** pour qu’on puisse tout de suite commencer à travailler.

# Les Annotations

Inspections ce nouveau fichier BlogController.php

N’importe quel commentaire qui commence par un **@** est une **annotation.**

Ici, **cette annotation s’appelle route**, on explique à Symfony que **quand un navigateur appellera mon site.com/blog , voici la fonction que tu dois appeler. La fonction index,** peut importe ce qu’elle fait.

Ici, notre **but dans cette fonction est de traiter la demande et de renvoyer une réponse. Ici, je renvoie le fait d’afficher un fichier html qui s’appel index.html.twig et qui se trouve dans le dossier blog.** On remarque que on a pas préciser que ce dossier se trouve dans le dossier template car Symfony sait que tout ses templates sont dans ce dossier-là. Donc on ne met que blog/index.html et on comprend que c’est bien **ce fichier twig qui nous permet de rendre un affichage.**

## Première page avec Symfony !

Se rendre sur : <http://localhost:8000/blog>

Je vois que Symfony m’a créé une nouvelle page, elle me dit que si je veux **modifier les traitement** pour cette page ça se trouve **dans BlogController.php**, si je veux **modifier l’affichage** pour cette page ça se trouve **dans blog.index.html.twig**: test : Changer le mot Hello en salut !

## Création de l’ensemble des routes qui serviront à ce blog.

Se rendre dans le BlogController.php et créer la fonction public home. Car je veux que ce soit ça qui soit appeler quand je vais sur mon-site.com/**rien** Donc, ça sera la page d’accueil du site.

# Langage Twig

## Simplicité

-Facilite l’écriture des affichages

-Apporte beaucoup de fonctionnalités (que le templating php avait du mal à nous apporter).

## Absence de PHP

-Permet d’abstraire les affichages de balises PHP

-Plus simple pour un intégrateur

Twig existe en dehors de Symfony !

{# Interpolation : {{ var }} <=> Affiche le contenu d'une variable ! #}

{# Commandes : {% cmd %} <=> Permet d'appeler une commande / structure de contrôle ! / Les commandes c'est tout les traitements possible. #}

# Mise en place : Ont créé des pages !

## Boostswatch.com ⬄ Des thèmes bootstrap très sympas !

Choisir un thème, par exemple Flatly. Aller sur Flatly et cliquer sur l’onglet Flatly en haut et choisir bootstrap.ss

Dans Twig si on regarde notre dossier templates on a effectivement le dossier blog avec les vues qui concerne le blog mais on a aussi un fichier qui s’appelle base.html.twig qui constitue un template de base. Si je veux que quelque chose se retrouve dupliquer sur toute mes pages, j’ai intérêt à les mettre dans ce template de base puis à chaque fois que je vais créer une autre page plus tard, je dirais qu’elle s’inscrit dans ce template. On remarque que dans ce fichier base on a plusieurs blocks :

block title

block stylesheets

block body

block javascript

Ces blocks sont des emplacements que toutes les autres pages pourront personnaliser. En fonction des pages que je vais créer, je vais demander à chacune d’entre elles de se placer par exemple au sein du block body et il faut imaginer par exemple que chaque page va mettre du contenu dans le block body ou dans le block title, etc. Donc chaque page va pouvoir personnaliser ces différents blocks mais tout ce qui se trouve dans ce template de base se trouvera dupliquer sur toute mes pages.

Donc ce que je vais faire, juste au-dessus du block stylesheets, me créer une balise link et mettre dans le href le lien vers mon css. Désormais, toute mes pages bénéficient de ce lien vers ce css.

Ce que je vais faire aussi, c’est prendre la navbar qui m’intéresse le plus.

Sur chaque composant, il y a 2 petites flèches (<>) sur lesquelles on peut cliquer pour voir son code source. Je copie la navbar que je veux et je vais la coller juste en dessous de mon block body et je supprime le formulaire car il ne me sert à rien pour le moment. Je renomme mon lien <a> et je garde que deux <li>, j’enlève la classe active du 1er <li> et je le fait aller vers Articles et le 2eme sera par exemple créer un article. Je peux aussi modifier les chemins des 2 <li> /blog et blog/new mais on ne les créera pas pour le moment. Pareil pour ma page d’accueil car je sais qu’elle est dans /

Ensuite, je veux que tout mon contenu soit dans une div bootstrap qui aura comme class container  et donc je veux que mon block body soit à l’intérieur d’un container comme ça je sais que chaque page qui enrichira ce blog, le contenu sera contenu.

    <div class="container">

        {% block body %}{% endblock %}

    </div>

### Comment faire en sorte qu’une page s’inscrive dans ce template la ?

Par exemple dans ma page [home.html.twig](http://home.html.twig) je peux utiliser la commande

{% extends 'base.html.twig' %} {# TWIG : EXTENDS <=> Permet de placer un gabarit au sein d'un template général // J'hérite du template-thème de base. #}

Attention, je ne peux pas l’utiliser tel quel. Si j’hérite du template base il faut absolument que mon contenu soit placer dans un certain block. Ici j’ai envie de modifier ce qui se trouve dans le block body donc je vais mettre tout mon code à l’intérieur et je n’oubli pas de mettre la commande {% endblock %} à la fin.

{% block body %} {# TWIG : Surcharge de blocks <=> On projète du contenu dans les différents blocks du template #}

# L’ORM Doctrine : Symfony et les bases de données.

Symfony utilise un ORM ⬄ Object Relationnal Mapping ⬄ C’est une brique logicielle qui fait le lien entre une application que ce soit en php/java/n’importe quelle autre langage et une BDD. Le but c’est qu’on gère au sein de notre application par des classes notamment et des objets nos données et que ce que l’on fait dans notre application se reflète automatique dans la BDD grâce à l’ORM.

Le but c’est que nous on ai presque jamais à toucher à nos BDD on écrira presque jamais de SQL, on utilisera des simples objets, des classes dans notre application et l’ORM se chargera de faire en sorte que les manipulations qu’on fait avec nos objets dans notre application se reflète dans la BDD.

**L’ORM** de **Symfony** s’appelle **Doctrine,** mais doctrine n’est pas du tout lié à Symfony, on peut donc l’utiliser dans d’autres projet php.

Dans Symfony grâce à Doctrine, on peut gérer nos tables, nos lignes de nos tables, les ajouter / les supprimer / les mettre à jour / faire des sélections / etc.

## Les outils qui vont nous servir.

**Entity** ⬄ Dans notre application Symfony, on va créer des classes qu’on appelle des entités et qui représente des tables.

**Manager** ⬄ Sert à manipuler des lignes / Insertion, mise à jour, suppression.

**Repository** ⬄ Permet de faire des sélections de données

# Les migrations dans Symfony

La philosophie de Symfony c’est de privilégié les fichiers car c’est ce qui sera partagé entre les différents développeurs. Quand on va faire un dépôt GIT et que nos collègues vont télécharger ce dépôt. Ils vont télécharger des fichiers pas une BDD. Et donc, la BDD doit venir des fichiers. Il faut que nos fichiers exprimes à quoi ressemble notre BDD.

Une migration de Symfony c’est un script qui nous dit je veux faire passer la BDD d’un état A à un état B.

Imaginons un fichier de Migration au #1 qui créé 2 tables

**Migration #1**

**-Je créé 2 tables**

Je peux avoir un autre fichier de migration (Bien sur ces fichiers ont un ordre pour qu’ils puissent être exécuté dans un certain ordre.

**Migration #2**

**-Je modifie les champs d’une table**

**-J’en créé une autre**

**-J’en supprime une autre**

**Migration #3**

**-Je supprime un champ d’une table**

**-J’ajoute une relation entre deux tables**

Tout ça est consigné dans des fichiers, donc quand je passerait mes fichiers à mes collègues il suffira qu’il fasse tourner ces script de migration en 1 sec pour qu’il se retrouve avec la même BDD que moi.

## Les Fixtures

C’est un script qui va créer des faux jeux de fausses données au sein de votre BDD. C’est encore une fois un fichier qui est exécutable à souhait, donc on peut faire n’importe quoi dans notre BDD. On sait que si on ré exécute notre fixture on remettra des fausses données dans notre BDD. Et surtout, c’est réutilisable par les autres.

**Si je développe un projet Symfony et que un collègue me rejoint et télécharge tout mes fichiers il n’a pas besoin d’avoir à faire du SQL. Il va pouvoir créer sa BDD via un script, il va pouvoir créer les tables dans les même état que les miennes via un script et il va pouvoir créer un jeu de fausses données grâce à un script. En gros, en 3 lignes de commandes, il sera opérationnel, il aura une BDD en place au même niveau que moi avec les mêmes tables et surtout, il a même des fausses données dedans pour déjà commencer à travailler et a voir son site internet évoluer.**

# Création d’une Base de données.

Pour créer une BDD, il faut se rendre dans le fichier .env du projet Symfony

Tout en bas de ce fichier se trouve une variable qui se nomme DATABASE\_URL qui est la pour expliquer à Symfony où se trouve ma BDD. Il faut donc remplacer ces valeurs.

DATABASE\_URL=mysql://db\_user:db\_password@127.0.0.1:3306/db\_name?serverVersion=5.7

Devient pour ma part

DATABASE\_URL=mysql://root:@127.0.0.1:3306/blog

Je peux maintenant demander à Doctrine via différente ligne de commande d’interagir avec MySql pour mettre en place ce dont j’ai besoin.

**Pour créer cette BDD qui n’existe pas encore chez moi**.

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

Doctrine me dit : J’ai bien créé une BDD qui s’appelle blog avec ta connexion par défaut.

Si je vais maintenant dans phpMyAdmin je vois qu’effectivement la BDD blog à bien été créé.

Mais elle n’a pas encore de table.

On va créer une table qui s’appellera article et qui contiendra l’ensemble de mes articles pour mon blog.

Rappel : Dans Symfony avec Doctrine on a des classes PHP qui représente des tables.

**Pour créer une de ces classes on va utiliser la commande.**

**php bin/console make:entity** ⬄ **Permet de créer une classe Entité qui représente une table**

**La ligne de commande me propose de donner le nom de la classe que je veux (Article).**

**Je remarque que ma ligne de cmd m’a créé 2 nouveaux fichiers :**

created: src/Entity/Article.php ⬄ **Représentera la table des articles**

created: src/Repository/ArticleRepository.php ⬄ **Qui me permettra de faire des sélections sur les données de cette table.**

**Maintenant grâce à la ligne de cmd je vais pouvoir créer les différentes propriétés, les différents champs de ma table. Ici, on parle des propriétés d’une classe mais quand Doctrine voit une classe avec des propriétés, ce qu’il voit enfaite, ce qu’il peut faire, c’est le transformer en une table avec des champs.**

**-Donc la, je sais que chaque article doit avoir une** **propriété** **=> title**

**Et là, il me demande de saisir le type de données pour cette propriété ( Ce sont les types doctrine et non pas les types SQL [On ne va pas écrire varchar255 par exemple]).**

**Si on veut voir tout les types qui existes, on peut taper ? et l’invite de commande va dérouler tout les différents type qui existe.**

**Le type string est le type proposé par défaut donc on peut appuyer sur entrer sans rien avoir noté.**

**La longueur 255 est proposé par défaut, ça me va aussi. Donc entrer.**

**Il propose si ce champ pour être nul ou pas. Non je ne veux aucun article sans titre donc je met no**

**Il me propose maintenant d’entrée un autre champ.**

**Content – text – no**

**Image – string – 255 – no**

**Bien penser à utiliser le CamelCase car on crée les propriétés d’une classe et la convention ici est le CamelCase. Attention, il faut savoir que dans la BDD quand doctrine va regarder cette entité et créer la table qui correspond dans la BDD, le champ CreatedAt ne sera pas en CamelCase, il utilisera la convention nommage de MySQL : created\_at**

**createdAt – datetime – no**

**Quand j’ai terminé, je peux taper « ENTRER » et la il me dit SUCCESS ! On t’a créé une entité et un repository.** Donc on va pouvoir créer une table article et on va pouvoir gérer cette table grâce au **Manager** qui va nous permettre de manipuler les données de la table et le **Repository** qui va nous permettre de faire des sélections sur ces données de la table.

Si je vais dans le **dossier Entity** de mon projet, je retrouve bien une classe particulière qui a des annotations au-dessus qui explique à doctrine : J’ai une classe Article qui correspond d’ailleurs pour les sélections au repository qui s’appelle Article et dedans j’ai des champs, des propriétés qui ont été classé en privé. Donc j’ai un Id qui sera une clé primaire avec une valeur qui sera auto généré/ qui aura un auto incrémente de type integer. J’ai un champ title qui sera un tring, donc un varchar de longueur 255, j’ai un champ content qui sera du text, etc, etc… Comme ces données ont été mise en privé, évidemment, la commande m’a généré automatique des guetter et setters. Donc tout ce qu’il me faut pour pouvoir travailler avec cette classe.

**Les tables n’existent toujours pas dans ma BDD phpMyAdmin**

**Je dois donc créer cette fameuse migration**

**Cette migration va permettre d’analyser mon code, donc Doctrine va regarder mes entité, elle va voir tout ce qui devrait exister dans ma base si on se base dans ces identité, elle va regarder d’un autre coté ma BDD. Elle va faire la différence entre les deux et ça va lui permettre de dire, attends, ta BDD il lui manque tel champ, tel table, tel ceci tel cela. Donc créons un script SQL pour amener ces tables-là.**

**php bin/console make:migration ⬄ Permet de créer une migration SQL à partir des entités présentes.**

**Et la il y a ce travail de différence qui va se faire entre les classes qui existent dans mon application et les tables qui existent dans ma BDD. Si Doctrine découvre qu’il y a des différences, mes fichiers ont la priorité, c’est mes fichiers qui ont la vérité et donc si dans mes fichiers il y a des choses qu’il n’y a pas dans la BDD ou si dans mes fichiers il n’y a pas les choses qu’il y a dans la BDD, Doctrine va créer ici une migration qui va mettre à jour la BDD pour qu’elle reflète complètement mes fichiers de mes classes.**

**Alors là, quand j’ai fait make:migration, le travail de différenciation a été fait et Doctrine à vu que j’avais une entité Article dans mon application mais que je n’avait pas de table article dans ma BDD. Donc il m’a créé une migration dans le dossier Migrations qui est versionnée, donc qui à un ordre. Celle-ci sera la version 1, la première et on voit qu’il y a une fonction up qui va créer une table. D’ailleurs il y a du script SQL ici qui va aller créer la table qui correspond à mon entité Article.**

**Si jamais demain je fais bouger mon entité Article en rajoutant/supprimant un champ ou que je refait une migration, l’analyse de Doctrine va montrer que il va falloir aussi faire un petit script SQL qui pourra les modifier ou supprimer un champ dans la table. Et donc tout ça, ça va se faire au fur et à mesure que je développe mon application, je vais avoir de plus en plus de fichiers de migration. Ce qui est cool, c’est que quelqu’un qui arrive en cours de projet, qui aura télécharger les fichiers, il n’aura plus qu’à lancer les migrations, lancer le script de migration pour passer d’une BDD complètement vide à une BDD avec toute mes tables et tout mes champs à la bonne version.**

**Maintenant que j’ai créé ma migration je vais la lancer.**

**php bin/console doctrine:migrations:migrate ⬄ Lance les scripts de migrations à fin de mettre à jour la base.**

WARNING! You are about to execute a database migration that could result in schema changes and data loss. Are you sure you wish to continue? (y/n)

**Y**

La migration est bien exécutée. Vérifier dans phpMyAdmin en actualisant la page.

# Créer une fixture.

**Installer le composant de création de fixture dans Symfony qui n’est pas livré par défaut dans le website/skeleton**

**composer require orm-fixtures –dev**

**php bin/console make:fixtures**

**Il me propose par défaut AppFixtures comme nom de fixture, moi je vais faire ArticleFixtures**

**C’est la fixture qui va me permettre de créer des faux articles.**

**SUCCESS !**

Il m’a créé dans mon dossier **DataFixtures** un fichier qui s’appelle **ArticleFixtures**

Examinons ce fichier

C’est un petit fichier avec une petite classe avec simplement une **fonction load** qui va recevoir le fameux **Manager** qui nous permet d’insérer mettre à jour et supprimer des lignes dans notre BDD.

Je supprime les 2 lignes de commentaire et je crée une boucle pour créer 10 articles

**Après avoir fini la création de ma fonction load()**

**php bin/console doctrine:fixtures:load ⬄** Charge toutes nos fixtures dans la base !

**Il me prévient que si je veux lancer ma fixture ça va supprimer toutes les données actuellement dans ma BDD pour mettre en place mes fausses données. Donc je dis y. Et je vais purger ma BDD et rajouter mes fausses données.**

**On peut maintenant constater dans phpMyAdmin la création de mes 10 articles.**

# Utiliser Doctrine : Accéder aux données dans Symfony.

On se retrouve dans le controller pour aller exploiter les données de la BDD qui existe désormais belle et bien.

Donc pour ça, si j’ai besoin de faire des **sélections** j’ai besoin d’un **Repository**, si je veux faire des **manipulations** j’ai besoin d’un **Manager**.

Pour le **Manager** on a vu dans la **Fixture** comment on pouvait **créer** un **article**, **modifier** ses **propriétés** et le **faire** **persister**. C’est exactement la même chose si on veut le faire mettre à jour. Donc on choppe un article existant, on modifie ses propriétés et on le fait persister. Ou le supprimer, c’est exactement les mêmes principes.

On va donc maintenant utiliser le **Repository.**

Je veux que dans ma page **/blog** je puisse **récupérer** la **liste** de mes **articles** et les **donner** à **twig** pour qu’il **les** **affiche** correctement.

Pour faire ça, j’ai besoin d’un **Repository,** le **Repository** **des** **articles** qui a été créer lorsqu’on a créé l’entité Article. (**Dossier : Repository** => **Fichier : ArticleRepository.php**)

## Avoir accès à ce repository au sein de mon controller pour sélectionner les données dans ma table.

**Rappel : Le repository ⬄ Il permet de sélectionner des données dans une table !**

Dans le **fichier BlogController.php** du **dossier Controller**

**Je crée une variable $repo dans ma function index() et je lui dit que je veux discuter avec Doctrine et je veux que tu me donne un Repository, celui qui gère l’entité Article.**

**Attention, si j’utilise la classe Article, je vais devoir expliquer à PHP où elle se trouve en utilisant le use et je sais qu’elle est dans App\Entity\Article ;**

**Et la dans $repo j’ai un repository qui sert à récupérer les articles.**

**Je peux dire :**

$article = $repo->find(12);

// à ce moment là il irait me trouver l’article n°12 et il me le filerait dans Article.

**Mais je peux aussi dire :**

$article = $repo->findOneByTitle("Titre de l'article");

**ASTUCE : LIRE LA DOC ⬄ On y trouve toutes les fonctions des repository !**

// Ça irait chercher un article dont le titre serait celui-ci ("Titre de l'article");

Le repository sait que dans ma table il y a un title car ce repository travail avec la classe Article et donc il sait que les articles de ma BDD ont un champ title donc je peux utiliser ça.

$articles = $repo->findByTitle("Titre de l'article");

// Et donc je vais trouver tout les articles qui ont ce titre.

$articles = $repo->findAll(); // Pour trouver tout mes articles

Et bien sûr, il y a énormément d’autres fonctions.

**Maintenant, je vais passer à twig l’ensemble de tous mes articles.**

**Donc je vais créer dans twig une variable qui s’appelle articles et qui contiendra le contenu de ma variable $articles.**

**Comment exploiter ça au niveau de mon twig ? Je me rend dans mon dossier des templates à la page index.html.twig et au lieu d’avoir des faux articles ici, je vais supprimer 2 de mes faux articles, je n’en garde qu’un seul sauf que cet article je vais le placer dans une boucle for de twig.**

{% for article in articles %}

{% endfor %}

**Ici, la variable article n’existe pas, elle se crée au moment du for et la variable articles (au pluriel) elle existe car je l’ai passé à twig dans le BlogController**

Je possède maintenant 10 articles dans l’url <http://127.0.0.1:8000/blog> mais pour l’instant les données ne changent pas.

Donc maintenant que à l’intérieur de cette boucle for, la variable article (au singulier) éxiste donc au lieu de dire titre de l’article en dur, je peux demander à twig de faire une interpolation avec les doubles accolades et d’afficher les contenus de la variable article.title

**Attention on utilise pas $article->title ou quoi que ce soit. Pour accéder au titre de l’article, la syntaxe twig c’est article.title**

**Dans notre classe Article.php du dossier Entity, le title est en privé alors comment il peut faire .title ? C’est parce que twig ce qu’il va faire ici, c’est qu’il va essayer d’accéder à article.title et il va voir que c’est privé et il ne va pas y arriver donc il va essayer un autre truc, il va essayer article.getTitle et à ce moment la il va voir que oui ça existe, donc je te le prend. Il va essayer comme ça plusieurs choses et à chaque fois que quelque chose rate, il en essaye une autre et au bou d’un certain nombre d’essai, getTitle / asTitle / isTitle / etc, s’il n’arrive à rien il fait une erreur. Mais ici, getTitle va fonctionner donc je n’ai pas besoin d’écrire ici article.getTitle() je peux utiliser simplement .title et twig sera le faire.**

**Je fais pareil pour le texte / pour l’image / et pour la date d’ajout.**

**Pour la date, c’est un peu particulier. Je vais vouloir interposer le {{ article.createdAt }} mais ça va causer une erreur car l’interpolation de twig me permet d’afficher le contenu d’une variable mais se limite aux données primitive. Les chaines de caractères, les booléens et les nombres. Hors ici, createdAt est un objet complexe, un datetime qui en plus ne possède pas de méthode tout string donc c’est impossible pour twig d’afficher ça. À ce moment-là, on peut utiliser la notion de filtre.**

**FILTRES TWIG ⬄ Formatter une variable avant de l’afficher.**

**Filtres me permet en plaçant ce symbole | de dire, tu vas afficher cette donnée mais en la formattant d’une certaine façon. Donc twig à un certain nombre de filtre (dispo dans la doc [twig filters]). Il a notamment un filtre qui va permettre de transformer une date complexe, un objet datetime en une chaine de caractère, de formater une date.**

**Ce filtre s’appelle : date et il prend en paramètre le format de date que l’on veut. Et là, twig va réellement afficher de création effectivement mais en ayant passé au préalable dans le filtre de date.**

 {{ article.createdAt | date('d/m/Y')}}

**Et je vais faire pareil pour l’heure.**

{{ article.createdAt | date('H:i')}}

**Et pour le moment on laisse la catégorie en dur car on en a pas encore.**

## ROUTES PARAMETREES

**Alors, maintenant j’aimerais que lorsque je clique sur un article ou que j’écris dans l’url blog/id qu’il m’affiche l’article qui correspond.**

**On va donc intervenir désormais sur la fonction show de notre controller et plutôt que de lui dire qu’elle recevra toujours 12, j’aimerais lui dire qu’elle recevra quelque chose de variable.**

**ROUTES PARAMETREES ⬄ Intégrer des paramètres variables dans une route.**

@Route("/blog/12", name="blog\_show")

**Devient**

@Route("/blog/{id}", name="blog\_show")

**Et là, Symfony comprend que dans une route qui est blog/quelque chose, ce quelque chose, c’est l’identifiant.**

**Pour récupérer cet identifiant, Symfony va le passer à ma fonction show()**

    public function show($id)

**Il me reste à créer un repository en disant**

// Je veux parler avec Doctrine, je veux avoir un repository, le quel me demande doctrine donc je lui dit Article::class

**Et là, j’ai mon repo, ce que je peux faire c’est dire :**

**J’ai besoin ici d’un article qui sera :**

$repo->find($id);

// Trouve moi l’article qui à l’identifiant qu’on m’a envoyé dans l’adresse en haut.

**Je dois maintenant passer un tableau à twig avec les variables que je veux qu’il utilise, notamment ici la seule variable qui m’intéresse c’est de dire à twig : tu vas devoir utiliser un article dans ton template et il contiendra les données de mon article.**

**Je peux donc maintenant aller dans ma vue qui s’appelle show.html.twig et je sais que maintenant dans tout ce template j’ai accès à une variable qui s’appelle article. Je peux donc dynamiser mon article.**

**J’ai maintenant accès à chaque page article mais j’ai maintenant une erreur quand je veux voir la page avec la liste des articles.**

**On a un problème au moment où twig essaye de me donner l’adresse de chaque article pour aller lire la suite, pourquoi ? L’erreur me dit ici, tu essaye de rendre la route ici qui s’appelle blog\_show mais il y a un problème car la route blog\_show ici (BlogController.php)**

@Route("/blog/{id}", name="blog\_show")

**C’est une route qui doit avoir un paramètre qui s’appelle identifiant, hors moi dans ma liste quand je dis ici (index.html.twig) : affiche moi la route**

<a href="{{ path('blog\_show') }}"

**On voit bien ici que je lui dit juste blog\_show et je ne lui précise pas l’identifiant, je ne précise pas le paramètre. Et ça, c’est un problème.**

**FONCTION TWIG :PATH ⬄ Permet de préciser une route et ses paramètres.**

**La fonction path de twig demande 2 paramètres, le nom d’une route et éventuellement si cette route est paramétrée, quelle sont les paramètres à intégrer dans cette route.**

**On lui passe donc un 2eme paramètre qui est un objet javascript, c’est comme du JSON et la on va lui dire : dans le paramètre id, tu dois mettre article.id**

<a href="{{ path('blog\_show', {'id': article.id}) }}"

## Bonus de style ! Introduction à l’injection de dépendances dans Symfony.

Dans Symfony, on a ce qu’on appelle un service container. En gros, tout ce qui est contenu dans Symfony est gérer par Symfony. On remarque ici **(BlogController.php)** La classe Controller que j’ai créé, mon BlogController je ne l’ai jamais moi-même instancier pour appeler les fonctions. C’est Symfony lui-même qui s’en charge. Ce qui veut bien dire que Symfony à un grand rôle à jouer dans notre application, il instancie des classes et il appelle ces fonctions.

Alors Symfony, dans ce service container, il a la possibilité de nous donner ce dont on a besoin quand on lui dit.

Par exemple : cette fonction index() (Rappel : son but c’est de nous afficher la liste des articles) et pour fonctionner elle a besoin d’un repository, le repository des articles qui va aller chercher la liste des articles. Quand une fonction à besoin de quelque chose pour fonctionner, on appelle ça une dépendance. Ma fonction dépend du fait d’avoir un repository des articles pour aller chercher la liste des articles.

**NOTION DE DEPENDANCE** **⬄** **Quand un classe/fonction a besoin de quelque chose**

Donc, si j’ai une dépendance, je peux demander à Symfony de me la fournir plutôt que d’aller la fabriquer moi-même.

En effet, cette fonction index() ce n’est jamais moi qui vais l’appeler, c’est Symfony qui va construire le controller quand il va voir la route /blog il va construire le controller et il va lui-même appeler la fonction index()

Symfony est capable d’examiner cette fonction pour voir, est-ce que moi Symfony, j’ai besoin d’envoyer à cette fonction quelque chose dont la fonction à besoin pour marcher.

Moi ici, j’ai besoin d’une instance d’une classe qui s’appelle le repository des articles. (Rappel : cette classe se trouve dans le dossier Repository et elle s’appelle : ArticleRepository et elle fait partie du namespace App\Entity\Article)

Donc ici je peux exprimer à Symfony pour que cette fonction marche bien, j’ai besoin d’un $repo mais il faut surtout car à ce stade-là, Symfony me dit bon d’accord mais qu’est-ce que tu veux que je te fasse dans le repo ? Il faut surtout que je donne un indice à Symfony, j’ai besoin d’un repo qui enfaite va contenir une instance de ma classe ArticleRepository.

public function index(ArticleRepository $repo)

Si je dis ici que j’ai besoin d’une classe ArticleRepository, il ne pas oublier d’ajouter le :

use App\Repository\ArticleRepository;

Donc la PHP saura de quoi je parle.

**PHP Type Hinting ! ⬄** **On donne des indices sur le type de paramètre attendu !**

Et là, Symfony quand il va vouloir appeler la fonction index() il va savoir que cette fonction à besoin d’une instance de la classe ArticleRepository qui s’appellera $repo

Du coup, je n’ai plus du tout besoin de cette ligne :

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

Puisque dans cette ligne, je demandais à Doctrine de me donner un repository mais je n’en ai plus besoin car Symfony, grâce au système d’injection de dépendance qui existe dans d’autres Framework aussi évidemment, comme dans Angular par exemple, a compris que quand il appellerait index, il passerait à cette fonction un Repository dans la variable $repo

Si je retourne dans ma liste d’article, je vois que tout fonctionne correctement, rien ne s’est cassé car j’ai simplement demandé à Symfony de m’injecter une dépendance le repository des articles car j’en ai besoin dans ma fonction.

Je peux faire exactement la même chose dans ma fonction show()

Je peux me passer de cette ligne-là

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

Tout simplement en demandant la dépendance, je vais demander à Symfony de me passer un repo qui serait un ArticleRepository et là ça marcherait puisqu’ici

$article = $repo->find($id);

J’utilise $repo pour trouver l’article qui à cette identifiant-là.

Je peux carrément me passer de cette ligne qui crée l’article

$article = $repo->find($id);

Et tout simplement demander à Symfony de me passer ici une variable article qui sera de type $article

public function show (Article $article)

**Comment il peut savoir quel est l’article que tu veux ? C’est grâce au :**

**ParamConverter ⬄ Converti un param de la requête en objet !**

**C’est un petit brique logiciel qui est dans Symfony. Que fait-il ? Il voit ici qu’on a une route qui va contenir un identifiant**

 @Route("/blog/{id}", name="blog\_show")

**Il voit par ailleurs que ma fonction à besoin d’un article donc il se dit ok je vais aller chercher l’article qui a cet identifiant et je vais te le filer.**

**Et la si on actualise la page /blog/1**

**Tout fonctionne correctement car Symfony comprend que ma fonction show() a besoin de lui passer un article. Au début il ne sait pas quel article passer mais en même temps il se rend compte que dans la route on lui parle d’un identifiant donc il se dit ok je vais aller te chercher l’article qui a cet identifiant-là. Et ça pourrait marcher avec d’autres champs que l’identifiant, ça peut être par le titre ou par autre chose.**

**Et donc grâce à l’injection de dépendance on se retrouve avec des fonctions beaucoup plus courtes, beaucoup plus propre et surtout on ne s’embête pas essayer d’instancier des classes nous-même, Symfony sait le faire mieux que nous, il a son service container qui sait en gros comment fonctionne notre application et quand on lui demande de nous passer des choses, il nous les passe gentiment. Grâce à Symfony, on va beaucoup plus vite et c’est aussi beaucoup plus clair.**

# Symfony : Forms ! Créer des formulaires dans Symfony.

## Créer une page : Première étape vers la gloire !

**(Controller/BlogController.php)**

**3 Piliers pour une page : ⬄ Une fonction / une route / une réponse (Affichage / Redirection).**

    /\*\*

     \* @Route("/blog/new", name="blog\_create")

     \*/

    public function create ()

    {

        return $this->render('blog/create.html.twig');

    }

**Attention : Confusion de routes ⬄ Quand 2 routes ont un pattern trop similaire**

**Quand je vais appeler /blog/new , Symfony va croire que je vais appeler un nouvel identifiant. Donc je peux soit préciser que l’identifiant ça doit être un nombre soit on change l’ordre des routes en faisant remonter cette nouvelle fonction au-dessus de la fonction qui reçoit l’identifiant. Du coup la fonction show apparaitra que si on a /blog/n’importe\_quoi\_ici\_sauf\_new.**

**(templates/blog)**

**Création du fichier create.html.twig**

**TWIG : Héritage … ⬄ Vous pouvez hériter d’un template plus large et le spécialiser**

{# Ce template doit faire parti de mon template de base donc il doit hériter de base.html.twig / Il hérite donc de la nav des styles css etc #}

{# Ce template étend le block body #}

Dynamisé la navigation qui amène vers cette page.

<a class="nav-link" href="/blog/new">Créer un article</a>

Devient

 <a class="nav-link" href="{{ path('blog\_create') }}">Créer un article</a>

**TWIG : Fonction path() ⬄ Elle permet de retrouver l’URL en fonction du nom d’une Route.**

**En profiter pour dynamisé le lien vers les Articles et la page home.**

## Création du FORM ! Approche Symfony !

***Rappel :***

**Injection de dépendances ⬄ Symfony peut nous fournir les éléments dont on a besoin !**

**HttpFoundation\Request ⬄ C’est la classe qui permet d’analyser / manipuler la requpete http**

* **Use Symfony\Compenent\HttpFoundation\Request ;**

**Doctrine : l’ObjectManager** ⬄ **Il permet de gérer une ligne d’une table (insert / update / delete)**

* **Use Doctrine\Common\Persistence\ObjectManager;**

**redirectToRoute (fonction)** ⬄ **Permet de créer une redirection vers une autre route !**

**(Controller/BlogController.php – function create)**

**Création de la fonction create.**

// Ce que je peux faire désormais c'est faire afficher ce formulaire à twig / dans create.html.twig

**(templates/blog/create.html.twig)**

**TWIG : FONCTION form()** ⬄ **Permet d’afficher un formulaire Symfony | Je peux passer un formulaire dans cette fonction.**

[**http://127.0.0.1:8000/blog/new**](http://127.0.0.1:8000/blog/new)

**Le formulaire s’affiche.**

// Symfony est capable, comme il est liée à une entité (plus haut, je lui ai dit que le formulaire est lié à l'entité $article). Il est capable d'aller voir l'entité article et quand il voit que ce champ s'appelle content, il va voir dans mon entité article ce qu'est content et il se rend compte que content, c'est du gros texte alors il en déduit qu'il va nous afficher du textArea

**Il faut savoir qu’on est pas obligé de faire confiance à Symfony pour l’affichage, on peut très bien utiliser nos propre volonté. Si je veux par exemple que le content ne soit pas une text area mais que ce soit un input normal. On peut utiliser un 2eme paramètre qui serait TextType ::class**

 ->add('content', TextType::class)

// Ne pas oublier le use pour le TextType pour expliquer à PHP d'où vient le textType

use Symfony\Component\Form\Extension\Core\Type\TextType;

**Pourquoi TextType ? Car Symfony nous fournis tout un ensemble de type de champs possible, on a le : TextType/DateType/SubmitType/TextAreaType/etc etc** [**https://symfony.com/doc/current/forms.html**](https://symfony.com/doc/current/forms.html)

// Dans la plupart des cas, on fait confiance à Symfony mais on peut toujours si on le souhaite, le configurer à notre guise.

**On peut encore donner un dernier paramètre à cette fonction add pour encore plus configurer notre champ. Et ce dernier paramètre représente les options de notre champ.**

**Pour title : je veux un tableau d’option. Je peux lui donner différentes options qui ont trait à un champ, mais je ne parle pas d’option html, si je veux lui donner des options html, je dois créer ici une clé attr. Parmis les options, il y a les options des attributs et donc les attributs html il y en a plusieurs donc j’ai peut-être envie de donner une classe css / un identifiant / placeholder / etc etc.**

**Moi par exemple ici je veux donner un placeholder.**

**Donc on voit que j’ai un tableau d’option pour mon champ title et parmis les options possible il y a une option qui s’appel attr qui est elle-même un tableau d’attribut html que j’ai envie de passer.**

{{ form\_start(formArticle)}} {# form\_start => c'est une fonction de twig qui va nous afficher la balise ouvrante du formulaire avec la bonne méthode, la bonne action et certaines données essentiel au fonctionnement du formulaire. #}

    <div class="form-group">

        <label for="">Titre</label>

        {{ form\_widget(formArticle.title) }} {# form\_widget => J'aimerais que twig m'affiche un champ et je peux utilier comme ça formArticle.title et twig comprend qu'on veut afficher le title #}

    </div>

    {{ form\_end(formArticle)}}  {# form\_end => J'arrête le formulaire #} {# Il va essayer d'afficher les 3 champs, à partir du form\_end si on a pas afficher nos champs comme on le souhaite, lui il va dire : tout les champs qui n'ont pas été affiché, je vais les afficher. #}

**Du coup ce que ça me permet de faire, c’est de dire je veux un autre form-group pour Contenu et pour Image.**

**Et la on a un affichage qui a changé avec des form-group, à la bootstrap entre guillemet.**

**(Controller/BlogController.php)**

**Ce qu’il nous reste à faire c’est de préciser dans les attributs de chacun de nos champs une classe qui serait form-control.**

                    'placeholder' => "Titre de l'article",

                    'class' => 'form-control'

**Et la si je rafraichi, j’ai un formulaire bootstrap !**

**Maintenant, on peut rendre ça plus fluide.**

**TWIG : Les templates ⬄ Créer des templates de forms pour décider leur affichage.**

**Par exemple Bootstrap qui est beaucoup utilisé. Les dev de Symfony et de sa communauté ont créé ce qu’on appel des moteurs de rendu twig. On a donc la possibilité en une seul ligne de faire en sorte que le formulaire bootstrap fonctionne.**

**(templates/blog/create.html.twig)**

**On a pas besoin de faire tout ça :**

    <div class="form-group">

        <label for="">Titre</label>

        {{ form\_widget(formArticle.title) }} {# form\_widget => J'aimerais que twig m'affiche un champ et je peux utilier comme ça formArticle.title et twig comprend qu'on veut afficher le title #}

    </div>

    <div class="form-group">

        <label for="">Contenu</label>

        {{ form\_widget(formArticle.content) }}

    </div>

    <div class="form-group">

        <label for="">Image</label>

        {{ form\_widget(formArticle.image) }}

    </div>

**Devient :**

{{ form\_widget(formArticle)}}

**Il s’affiche donc à la bootstrap et du coup sans que l’on ai besoin de placer aucune classe form-control**

**(Controller/BlogController)**

                'attr' =>

                [

                    'placeholder' => "Image de l'article",

                    'class' => 'form-control'

                ]

**Re Devient**

                'attr' =>

                [

                    'placeholder' => "Image de l'article"

                ]

**Comment faire pour aller très vite, quand on fait du bootstrap on veut prototyper vite on a pas le temps de passer à faire des dizaines de ligne. On veut ne pas avoir à préciser quoi que ce soit et faire en sorte que le formulaire soit beau.**

**Il y a une extension twig qui s’appelle un thème bootstrap pour les formulaires.**

**Ces thèmes de formulaire on peut les créer nous-même, on peut donc controller les rendus de formulaire. On pourrait par exemple, créer un thème Bulma**

**Pour l’installer :** [**https://symfony.com/doc/current/form/bootstrap4.html**](https://symfony.com/doc/current/form/bootstrap4.html)

**Il faut aller dans la configuration de twig pour expliquer que il existe un thème que je veux utiliser pour les formulaires.**

**Sur le site je copie :**

**twig:**

form\_themes: ['bootstrap\_4\_layout.html.twig']

**Et je la colle dans (config/packages/twig.yaml)**

**Attention à ne pas copier twig, on est déjà dans twig**

**Je te précise que tu peux avoir certains thème de formulaire. Pour l’instant c’est un tableau qui ne contient qu’un seul thème qui est : bootstrap\_4\_layout.html.twig**

**On ne sait pas où c’est ni d’où ça vient mais c’est comme ça qu’on fait !**

**Ensuite, (templates/blog/create.html.twig)**

**Là où je vais utiliser mon formulaire, on a une ligne à rajouter qui est de préciser à twig qu’on va utiliser un thème de formulaire pour le formulaire formArticle**

{% form\_theme formArticle 'bootstrap\_4\_layout.html.twig' %}

{# On veut utiliser un thème de formulaire pour le formulaire formArticle et ce thème c'est bootstrap\_4\_layout.html.twig #}

**Je rafraichis ma page et je retrouve mon formulaire au couleur de bootstrap, beau, propre, nickel.**

**On a même plus besoin du**

{{ form\_start(formArticle)}}

**Ni du**

{{ form\_end(formArticle)}}

**Et**

{{ form\_widget(formArticle)}}

**Devient**

{{ form(formArticle)}}

**Ça donne toujours la même chose !**

**Mais on préférera toujours garder le form\_start et le form\_end au as ou on aurait envie de mettre des choses entre le début du formulaire, l’affichage du formulaire et la fin du formulaire.**

## Bouton pour soumettre.

**(Controller/BlogController)**

**En dernier champ on rajoute un bouton qu’on va appeler save et qui se ra de type Submit et on lui passe en tableau d’option label => enregistrer. Ne pas oublier d’aller rajouter le use pour le SubmitType.**

            ->add('save', SubmitType::class,

            [

                'label' => 'Enregistrer'

            ])

use Symfony\Component\Form\Extension\Core\Type\SubmitType;

**Cependant, on ne fait pas souvent ça car notre formulaire peut servir tantôt à ajouter un article tantôt à modifier un article voir autre chose. Du coup on ne veut pas que notre bouton soit totalement bloquer sur « Enregistrer » / avec le label enregistrer par exemple.**

**Ce que j’aimerais faire c’est aller dans mon formulaire et créer moi-même mon bouton.**

**(templates/blog/create.html.twig)**

<button type="submit" class="btn btn-success">Ajouter l'article</button>

**Je n’oublie pas de supprimer mon précédent bouton !**

## Séparation du code : Garder de bonnes pratiques.

**(Controller/BlogController)**

**Je vais faire confiance à Symfony et je supprime toutes les options particulière que j’ai moi-même apporter. Donc je le laisse gérer les types de champs et je ne mets aucune options dessus.**

**Mais du coup, je perd mes placeholders dans mes champs.**

**Ce qu’on peut faire c’est un compromis ! Je garde mon formulaire simple comme ça mais je peux m’en occuper dans :**

**(templates/blog/create.html.twig)**

**En disant champ par champ ce que je veux.**

**La fonction : Form\_row() ⬄ Nous permet d’afficher tout ce qu’il faut pour un champ. C’est-à-dire le label, le champ et les erreurs potentiels qu’on verra plus tard.**

**Quand on utilise form\_row on peut non seulement passer le champ du titre que l’on veut mais on peut surtout commencer à donner des autres options. Le 2eme paramètre de form\_row c’est un tableau d’options. Il ne faut pas oublier qu’on est en twig ici, donc un tableau ne s’écrit pas avec des crochets [] mais avec des accolades {}, c’est du JSON enfaite qu’on écrit ici. Et donc on va dire par exemple pour le title qu’on veut une option (par ailleurs, c’est le même tableau d’options qu’on avait vu quand on créait les champs). Je veux donc une option qui s’appelle attr qui parle des attributs et dans cette option c’est un autre tableau dans lequel je vais dire le placeholder c’est titre de l’article.**

**TWIG : Tableaux associatifs ⬄ On les écrit comme des objets en JS.**

{{ form\_widget(formArticle)}}

**Devient :**

{{ form\_row(formArticle.title, {'attr': {'placeholder': "Titre de l'article"}} ) }}

**Je fais pareil pour le content et pour l’image.**

**J’ai donc bien mon formulaire avec les placeholders mais qui sont cette rangé au niveau de mon code.**

## Traiter le form ! Rien à faire…

**Les infos qui sont envoyé par le formulaire, elles me sont par la request, elle se trouve dans la request.**

**(Controller/BlogController)**

$form->handleRequest($request);

// => Formulaire, essaye d'analyser la requête http que je te passe ici en paramètre.

// La 1ere chose que je veux faire, c'est afficher ce formulaire. Donc je veux passer ce formulaire à twig. Je vais donc lui passer une variable qui soit relativement facile à afficher. | Il va analyser, il va voir si ça été soumis ou pas et si ça été soumis, tout les champs qu'on attendait à trouver s'y trouve, est-ce que tout va bien?

**Donc on va pouvoir commencer à vraiment pouvoir analyser cette requête. Laus ce b’est oas biys qui allons le faire, c’est le formulaire qui va le faire pour nous. Que fait-il ?**

**Il cherche dans la requête si il y a un title, un content, une image. Et si il trouve un title il va le bind avec le title de l’article, pareil pour content et image.**

**Tester :**

dump($article);

**Remplir les champs et remarquer que le dump nous rend bien ce qu’on a rempli.**

**Maintenant on veut enregistrer notre article.**

 if($form->isSubmitted() && $form->isValid())

// C'est une méthode de la classe form qui me permet de savoir si on est en train d'arriver sur la page et rien n'a été soumis on affiche juste le formulaire ou est-ce qu'on est à la 2eme page, quand la personne à rempli le formulaire et cliquer sur enregistrer. | Et surtout, est-ce que le form est valide ? Car des fois on peut soumettre dans un champ email un numéro de téléphone, et ce ne sera donc pas valide.

**On verra d’ailleurs qu’on peut faire de la gestion d’erreurs au niveau de l’entité.**

**Il me reste à ajouter à l’article sa date de création et je demande au manager d’envoyer la requête après avoir faire persister l’article.**

**ATTENTION : J’ai changé**

Use Doctrine\Persistence\ObjectManager;

**Et**

public function create(Request $request, objectManager $manager)

**Par**

use Doctrine\ORM\EntityManagerInterface;

public function create(Request $request, EntityManagerInterface $manager)

**Car ça ne fonctionnait pas contrairement au tuto de Lior.**

## New + Update

**Si l’article n’est pas vide ! Par exemple je vais donner un titre à mon article et je vais aussi lui donner un contenu.**

$article->setTitle("Titre d'exemple")

                ->setContent("Le contenu de l'article");

**Si j’actualise, mon formulaire est déjà prérempli avec les données de l’article.**

## MISE à JOUR ! Un formulaire > 2 actions

**Mon but est d’utiliser la même fonction pour ajouter un nouvel article qui est vide ou modifier un article qui éxiste.**

public function create

**Devient :**

public function form

**Car c’est une fonction qui va montrer un formulaire car elle peut à la fois nous faire créer mais aussi mettre à jour. Et je lui donne une seconde route.**

    /\*\*

     \* @Route("/blog/new", name="blog\_create")

     \* @Route("/blog/{id}/edit, name="blog\_edit")

     \*/

**Cette fonction pourra donc être appeler par 2 adresses différente : soit /blog/new soit blog/{id}/edit**

**Si je recois un identifiant dans la route c’est que je peux le recevoir en paramètre de ma fonction, mais on se rappel du paramconverter, on peut demander à symfony de carrement nous passer un Article qu’on va appeler $article**

**Le paramconverter ⬄ Convertit un paramètre en une entité**

public function form(Article $article,Request $request, EntityManagerInterface $manager)

**Je vais maintenant commenter cette ligne :**

$article = new Article();

**Mais en faisant ça, si je vais sur** [**http://127.0.0.1:8000/blog/new**](http://127.0.0.1:8000/blog/new) **Je vais avoir une erreur car en paramètre de la fonction form j’ai demandé à Symfony de me passer l’article, l’article qui correspondrait éventuellement à cette route :**

@Route("/blog/{id}/

**Mais la je ne suis pas sur cette route, je n’ai pas d’identifiant donc je ne peux pas aller chercher un article. Je suis sur la route new donc Symfony ne sait pas me donner un article, c’est impossible pour lui. Donc ce que je peux faire c’est dire que parfois cet article la sera null.**

**Et la si j’actualise je n’aurais plus de soucis.**

**Sur new l’article est null et sur /11/edit il est pas null il contient bien les données qu’il faut.**

Article $article = null

**Donc ça c’était juste pour éviter une erreur.**

**Maintenant je veux dire que si je n’ai pas d’article donc article = null je veux que ce soit un véritable article, donc une instance de la classe Article. Donc si je n’ai pas d’article je veux créer un nouvel article.**

if(!$article)

        {

            $article = new Article();

        }

**Si je vais sur la route new j’ai la possibilité d’écrire un nouvel article.**

**Ce que j’ai fait la du coup marche aussi pour la modification et pour la création à une chose près, c’est que j’ai pas envie de créer à chaque fois la date de création.**

**Je veux dire que si dans mon article il y a un identifiant (donc qu’il existe déjà) je n’ai surtout pas envie de mettre en place la nouvelle date de création. Donc je vais dire que si l’article n’a pas d’identifiant alors la je veux mettre une date de création. Et sinon, je ne touche pas à la date de création. On est juste en train de le modifier.**

**Et la je peux aller tester.**

**Il me reste un problème, c’est que j’ai le même bouton que je sois dans la création d’un article ou de modification d’un article. Je veux donc changer le texte de ce bouton.**

**Je vais donc passer une variable à mon twig :**

**Je vais l’appeler editMode et je veux un booléen true (si l’article existe déjà) ou false (si l’article n’existe pas) en voyant son identifiant. S’il est différent de null ça sera true et on sera en editMode s’il est égal à null l’editMode sera à fals.**

'editMode' => $article->getId() !== null

**(templates/blog/create.html.twig)**

**Dans mon twig j’ai une variable qui s’appelle editMode et qui me permet de savoir si on est en mode d’édition ou pas. Et donc dans mon bouton submit, je peux faire varier mon texte en fonction de cela. (Et pas que le texte d’ailleurs).**

    <button type="submit" class="btn btn-success">

        {% if editMode %}

            Enregistrer

        {% else %}

        Ajouter l'article

    </button>

**Blog/new j’ai bien un bouton : Ajouter l’article et sur blog/12/edit j’ai bien un bouton : Enregistrer**

**Faire pareil pour les <h1> en haut de page…**

**En très peu de code, j’ai le formulaire de création et le formulaire d’édition d’un seul coup.**

**Ça peut être encore plus simple ! Grâce à la console : la CLI**

**Créons le formulaire de manière automatique.**

**php bin/console make:form ⬄ Crée rapidement un formulaire dans son propre fichier !**

**Il demande le nom du formulaire qu’on veut faire. La convention c’est que le nom du formulaire finisse par Type ArticleType**

**Est-ce que ce formulaire ce base sur une entité ? Donc oui il se base sur : Article Il pourrait ne pas se baser sur une entité, pas de problème ! En lui fournissant l’entité sur la quel je me base, ben la je dit ce formulaire sert à manipuler des articles et la on va voir qu’il m’a créer un formulaire**

**Success !**

**On peut retrouver ce formulaire dans (Form/ArticleType.php)**

**Et la c’est une classe qui va sensiblement la même chose que ce que je viens de faire dans mon controller. Elle va créer ici un formulaire. On remarque que les champs sont mis automatiquement. Ce que je peux faire à la limite c’est dire que je ne veux pas la date de création en le supprimant mais le reste je le veux.**

**Ce qu’il me reste à faire dans mon Controller, plutôt que de créer moi-même le formulaire en ajoutant moi-même les champs, je peux tout simplement le supprimer/ou le commenter. Et ajouter la méthode :**

$form = $this->createForm(ArticleType::class, $article);

// Ne pas oublier le use App\Form\ArticleType;

**Ce qui est encore mieux avec cette méthode la c’est que si j’ai besoin de créer un formulaire pour gérer les articles dans plusieurs fonctions de mon controller, dans plusieurs endroit de mon site. Je ne vais pas avoir envie de dupliquer du code car c’est le mal !**

**Duplications = problèmes ! ⬄ Temps de maintenance / erreurs / évolutivité**

**Donc ce qu’on fait ici, on a créer un formulaire qui est dans un fichier qui s’appelle ArticleType et maintenant, partout dans mes controller, dés que j’ai besoin de ce formulaire je l’appelle tout simplement, j’appelle ArticleType.**

## La validation ! Sur nos entités.

**Je veux gérer un minimum de validation sur le remplissage d’un article lors de sa création !**

[**https://symfony.com/doc/current/validation.html**](https://symfony.com/doc/current/validation.html)

**Un peu comme Ruby, Symfony à choisis de faire de la validation directement sur le Model. C’est-à-dire ici, sur l’entité des articles. La validation monte du model vers le formulaire et c’est grâce au model que le formulaire va savoir faire des validations particulière.**

**Ce qu’il faut faire maintenant c’est travailler sur notre entité Article (Entity/Article.php)**

**Il va falloir ajouter un namespace bien particulier qui sert à toute les validations. Ce namespace on peut le trouver sur la même pas de la documentation.**

use Symfony\Component\Validator\Constraints as Assert;

**C’est un namespace qui nous amène vers une classe qui s’appelle constraints mais on le renommé en Assert car Assert est un lagage de test unitaire et c’est donc plus pratique pour s’y retrouver. Tout comme Mapping qu’on a appelé ORM, et du coup on a des annotations qui s’appelle @ORM au lieu de s’appeler Mapping.**

**Les contraintes Symfony ⬄ Elles permettent de soumettre des données à des contraintes.**

**Ce que je veux faire, c’est utiliser les différentes fonctions de tests et de validations qui éxiste. Ces fonctions, je peux les retrouver en bas de la même doc, on peut y voir un certain nombre de contrainte qui existe et qui sont utilisable, il y en a vraiment pas mal. On peut faire beaucoup de chose, même évidemment créer nos propres contraintes sans aucun soucis avec des expressions régulière avec tout ce dont on a envie de faire.**

**Moi, je veux utiliser par exemple, les contraintes Length/ la contrainte de longueur.**

**Dans la doc, je vois que je peux l’utiliser simplement en plaçant par-dessus la propriété qui m’interesse @Assert\Length et c’est une fonction qui va prendre différents paramètre.**

**Les paramètres sont aussi dans la doc, juste au dessus de l’exemple et ils sont : le min / le max / un charset / un msg d’erreur si on est pas dans le minimum / etc etc**

**On va donc le faire :**

    /\*\*

     \* @ORM\Column(type="string", length=255)

     \* @Assert\Length(min=10, max=255)

     \*/

    private $title;

**Essayons de créer un nouvel article avec moins de 10 caractères comme titre et on remarque une erreur : Veuillez modifier la valeur pour correspondre au format demander.**

**Validation de formulaires ⬄ La validation se fait des deux côtés : HTML et PHP**

**Ce qui veut dire que même si je supprime la validation côté HTML dans l’inspecteur, la validation du formulaire me sera quand même refusé car il y a aussi une validation php qui me dit ici que la valeur n’est pas bonne.**

**Faisons pareil pour le contenu : je veux un minimum de 10 caractère mais pas de maximum**

    /\*\*

     \* @ORM\Column(type="text")

     \* @Assert\Length(min=10, max=255)

     \*/

    private $content;

**Pour l’image, c’est un peu particulier. Je ne veux pas utiliser du Length avec du min et du max. Je veux dire : Assure toi que l’image est bien une URL. C’est une contrainte d’URL.**

**@Assert\Url() ⬄ On s’assure que la données ait bien le format URL**

    /\*\*

     \* @ORM\Column(type="string", length=255)

     \* @Assert\Url()

     \*/

    private $image;

**Ce qui est propre c’est que la validation est faite directement au sein de l’entité et non pas au sein du formulaire, ce qui fait que si demain je veux faire un nouveau formulaire mais sur la même entité, sur les articles mais peut être qu’il sera présenté différemment et qui servira à autre chose. Ce n’est pas grave car les validations sont centralisé ici donc je n’ai absolument rien à faire et tout ça fonctionne uniquement parce que on utilise le formulaire de Symfony qui est ultra performent, qui peut aussi gérer les erreurs et ça fonctionne aussi car dans notre twig, quand on utilise la fonction twig qui est form\_row() on gère à la fois le label mais on gère aussi l’input mais on gère aussi les potentiels erreurs qu’on peut avoir la dessus.**

**On peut biensur modifier les messages d’erreurs du formulaire**

    /\*\*

     \* @ORM\Column(type="text")

     \* @Assert\Length(min=10, max=255, minMessage="Votre titre est bien trop court l'ami !")

     \*/

    private $content;

**C’est le template bootstrap de twig qui a tout gérer, le rouge, le tag error etc mais si on veut utiliser notre propre css et notre propre présentation il n’y a pas de problème pour le faire notamment parce que twig nous propose une fonction qui s’appelle :**

**(templates/blog/create.html.twig)**

{{ form\_error(formArticle.title)}}

{# La il va nous donner une liste d'erreur, donc on peut boucler dedans on peut les présenter avec des div avec les classes qu'on veut, c'est à nous de nous débrouiller la dessus. #}

# Les relations entre entités doctrine.

**Je vais créer une nouvelle table qui va s’appeler category qui seront des catégories d’articles. Mais je ne veux pas que cette table flotte indépendamment de la table des articles, je veux que les deux soit lié l’une à l’autre. Je vais donc créer une nouvelle entité.**

# Les entités : On rajoute des tables : (Category & Comment)

## Category :

**Pour créer une entité en Symfony on appel notre ligne de commande.**

**php bin/console make:entity ⬄ Permet de créer une entité (table) facilement.**

**Je vais appeler cette entité : Category**

**Et la il nous dit qu’il a créé :**

**created: src/Entity/Category.php**

**created: src/Repository/CategoryRepository.php**

**Ce que je veux comme champ dans ma table Category : Pas besoin de préciser l’id, ça se fait automatiquement.**

**title - string - 255 – No**

**Je veux un 2eme champ !**

**description – text – yes**

**Je peux via Doctrine maintenant lui demander une relation.**

**On me demande si je veux rajouter une propriété :**

**articles - ?**

**Je vais dire que articles c’est un champ que je viens de créer qui sera de type relation**

**A quelle classe cette entité devrait être relié ? Article**

**Il me demande maintenant quel type de relation que je veux :**

**ManyToOne – OneToMany – ManyToMany – OneToOne**

**La il me dit, on va aussi rajouter une propriété category dans la classe Article pour représenter dans chacun des articles la catégorie. Mais si tu veux tu peux lui donner un autre nom. category**

**Maintenant il me dit : est-ce que tu pense qu’on a le droit de faire en sorte que la catégorie d’un article peut être null (est-ce qu’il peut exister des articles qui n’ont pas de catégorie ?) ça va donc dépendre de notre cas. Ici on va mettre : no**

**Est-ce que lorsqu’un article perd sa catégorie je veux le supprimer ? no**

**La il me dit, ok j’ai tout mis à jour ! Je dis donc que je veux arrêter ici ! Success !**

**Examinons les fichier (Entity/Article.php & Category.php)**

    /\*\*

     \* @ORM\ManyToOne(targetEntity=Category::class, inversedBy="articles")

     \* @ORM\JoinColumn(nullable=false)

     \*/

    private $category;

    /\*\*

     \* @ORM\OneToMany(targetEntity=Article::class, mappedBy="category")

     \*/

    private $articles;

    public function \_\_construct()   // On voit que dés qu'il construit une catégorie il dit que : la propriété articles c'est un type : ArrayCollection => C'est une surcouche des tableau classic en PHP qui leur rajoute pleins de méthode ultra sympas à utiliser, on se croirait presque en js. C’est un tableau ++

    {

        $this->articles = new ArrayCollection();

    }

**Classe ArrayCollection ⬄ Un wrapper pour les tableaux (uber sympas !)**

public function getArticles(): Collection

**Va nous permettre de récupérer une collection des articles.**

public function addArticle(Article $article): self

**Nous permettra de rajouter un article au sein de la catégorie.**

public function removeArticle(Article $article): self

**Nous permettra de supprimer un article au sein de cette catégorie.**

**Mais je n’ai toujours pas de table catégorie dans ma BDD car je n’ai pas encore dit de faire la migration !**

**php bin/console make:migration ⬄ Mise à jour de la base de données en fonction des fichiers.**

**Doctrine va vérifier toute nos entités et il va donc pouvoir dé sceller ce qu’il manque dans notre BDD par rapport au entité qu’il y a dans nos fichiers. C’est les fichiers qui font fois, la BDD elle ne fait que suivre.**

**(Migrations/Version20200601154458.php)**

**Attention !**

$this->addSql('ALTER TABLE article ADD category\_id INT NOT NULL');

**On a ici une requête qui va rajouter un champ category\_id dans la table des articles mais elle va dire que ce champ ne peut pas être null. Sauf que si je vais voir mon phpMyAdmin, j’ai déjà des articles et ces articles quand on va leur rajouter le champ category\_id ça sera égale à null donc tout va exploser. Donc ce qu’on va faire, surtout qu’on s’en fou de ces données là. On va donc tout supprimer. Et la c’est bon, ma migration va bien se passer.**

**Mais comment on peut faire si on rajoute une fonctionnalité alors que le site est en prod ? La meilleurs des solutions serait de permettre lorsqu’on crée notre entité que notre category\_id puisse être null à fin que cette migration ne pose aucun soucis à aucun moment.**

**Ma migration est donc prête !**

**Je vais maintenant intégrer cette migration à ma BDD.**

**php bin/console doctrine:migration:migrate ⬄ Permet de lancer des migrations qui sont en attente.**

**Attention ça va modifier ta BDD, est-tu sure ? y**

**J’actualise phpMyAdmin et je retrouve bien une table category et dans ma table des articles je retrouve un champ category\_id qui permettra de faire le lien entre les articles et les catégories.**

## Comment :

**J’aimerais maintenant créer des commentaires :**

**php bin/console make:entity**

**Comment – author – string - 255 – no – content – text – no – createdAt -datetime – no**

**Ces commentaires, comment ils s’articule avec le reste de ma BDD, avec le reste de mes données, de mes entités ? Il faut qu’on crée ici une nouvelle relation avec l’entité article.**

**Donc je vais dire que j’ai une propriété qui s’appel article de type relation avec la classe Article et avec le type de relation : ManyToOne semble correspondre.**

**Est-ce que la propriété article dans les commentaires peuvent être nul / si un commentaire peut être lié à aucun article ? no Est-ce que tu veux qu’on rajoute une propriété dans la classe Article qui fasse référence à nos commentaires ? Comme ça quand on se balade sur un article on peut accéder facilement au commentaire de cet article. yes Comment veux-tu appeler cette propriété au sein de ta classe Article ? Il me propose comments au pluriel par défaut. Je suis d’accord : comments**

**Est-ce que tu veux supprimer automatiquement des commentaires qui serait orphelins et la pour le coup je vais mettre : yes**

**Et la il m’a crée mon entité commentaire et j’arrête mon script ici. Success !**

**Allons examiner à quoi ressemble cette fameuse entité commentaire.**

**(Entity/Comment.php)**

**Je vois ma relation ici :**

    /\*\*

     \* @ORM\ManyToOne(targetEntity=Article::class, inversedBy="comments")

     \* @ORM\JoinColumn(nullable=false)

     \*/

    private $article;

**(Entity/Article.php)**

**Non seulement il avait modifier tout à l’heure pour les categories mais maintenant il a re modifier pour les commentaires.**

    /\*\*

     \* @ORM\OneToMany(targetEntity=Comment::class, mappedBy="article", orphanRemoval=true)

     \*/

    private $comments;

**Et il a fait pareil que dans les catégories, il a dit que les commentaires c’est un ArrayCollection, donc un tableau ++**

    public function \_\_construct()

    {

        $this->comments = new ArrayCollection();

    }

**Et il nous a donner la possibilité avec des méthodes qu’il a écrit lui-même automatiquement de chopper les commentaires :**

public function getComments(): Collection

**D’ajouter des commentaires :**

public function addComment(Comment $comment): self

**De supprimer un commentaire sur cet article :**

public function removeComment(Comment $comment): self

**Donc : on a créé l’entité, on a créé le repository mais on a toujours pas la table comment. Donc on retourne sur la console :**

**php bin/console make:migration**

**Il nous crée donc un fichier de migration (Migrations/Version20200601163216.php)**

**qui nous dit : je vais envoyer une requête de création de table pour mes commentaires**

$this->addSql('CREATE TABLE comment

**et surtout, je vais aller modifier la table commentaire une fois qu’elle sera créé pour lui rajouter une contrainte etc qui fait le lien avec la table des articles.**

$this->addSql('ALTER TABLE comment ADD CONSTRAINT

**Maintenant que j’ai fini, je peux faire un :**

**php bin/console doctrine:migration:migrate**

**y**

**Si j’actualise phpMyAdmin, j’ai bien une nouvelle table qui s’appelle : comment et qui concernera un article et on voit qu’il à un lien avec article\_id ce qui fait un lien avec les articles.**

**Qu’est-ce qu’il me manque ? Il me manque des données !!**

# Fixtures et faker ! Créer des jeux de fausses données.

**Les scripts fixtures ⬄ Permettent de remplir les tables avec de fausses données.**

**On va faire un peut plus compliqué que le précédent fichier de fixture : (DataFixtures/ArticleFixtures.php)**

**Mon but ici, ça va être de créer 3 catégories. Pour chaque catégorie, de créer peut être 5 articles et pour chaque article de créer peut être 3 ou 4 commentaires.**

**On va parler de la librairie Faker.**

**Librairie faker (Merveille) ⬄ Permet de créer des données « fake » aléatoires !**

**Faker n’est pas vraiment une technologie. Faker c’est une idée. Ça existe à peu près tout les langage, c’est une librairie qui existe en PHP / JS / Ruby / il y en a dans tout les langages. Il va nous permettre de créer des fausses données un peu plus convainquent que ce qu’on à fait précédemment.**

**Mon but maintenant c’est d’avoir des phrases, des Lorem Lupsum, si je veux avoir des prénoms / des noms de familles / des pays / des villes / etc etc. Je vais donc utiliser la librairie Faker !**

**https://github.com/fzaninotto/Faker**

## Installation

**composer require fzaninotto/faker je peux même rajouter --dev pour lui dire que ça ne servira que pendant que je développe.**

**Une fois installer, je peux utiliser la librairie faker au sein de mon projet.**

**Pour l’utiliser :**

**Il faut créer une variable qui sera une instance de la classe faker en appelant dans l’espace de nom Faker la classe Factory qui a une méthode statique qui s’appelle create et qui va donc nous filer une instance de la classe Faker. Je peux même lui passer une local pour avoir des fausses données mais en français. Des noms des prénoms des villes etc.**

$faker = Faker\Factory::create('FR\_fr');

**Ce que je peux faire avec Faker, si je regarde la doc :**

    echo $faker->name;

        // 'Lucy Cechtelar';

    echo $faker->address;

        // "426 Jordy Lodge

        // Cartwrightshire, SC 88120-6700"

    echo $faker->text;

        // Dolores sit sint laboriosam dolorem culpa et autem. Beatae nam sunt fugit

        // et sit et mollitia sed.

        // Fuga deserunt tempora facere magni omnis. Omnis quia temporibus laudantium

        // sit minima sint.

**Maintenant que j’ai mon faker, j’ai envie de dire que je veux créer 3 catégories fakées. Je vais donc faire une boucle.**

for($i = 1; $i <= 3; $i++)

**Après avoir fini de coder**

public function load(ObjectManager $manager)

**Doctrine:fixtures:load ⬄ Charge les fixtures dans la base de données !**

**php bin/console doctrine:fixtures:load**

**yes**

**Je peux maintenant aller vérifier dans mon phpMyAdmin et je possède bien mes 3 tables. Et pareil dans :** [**http://127.0.0.1:8000/blog**](http://127.0.0.1:8000/blog) **J’ai bien tout mes faux articles !**

# Afficher avec twig : On affiche les entités liées entre elles !

**(templates/blog/show.html.twig)**

**Je veux maintenant dynamiser mes catégories. Je veux que « Politique » soit variable, qu’elle soit le nom de la catégorie en question.**

**Pour faire ça, je vais demander à twig d’interpoler dans l’article, va chercher la catégorie et dans la catégorie va chercher le title.**

<div class="metadate">Ecrit le {{ article.createdAt | date('d/m/Y') }} à {{ article.createdAt | date('H:i') }} dans la catégorie {{ article.category.title }}</div>

**On va créer maintenant en dessous de notre article une petite section qui aura l’identifiant commentaires.**

<section id="commentaires">

    {% for comment in article.comments %} {# Pour chaque commentaire dans les commentaires de l'article #}

        <div class ="comment">

            <div class="row">

                <div class="col-3">

                    {{ comment.author }} (<small>{{comment.createdAt | date('d/m/Y à H:i')}}</small>)

                </div>

            </div class="col">

            {{comment.content | raw}}

        </div>

    {% endfor %}

</section>

# Formulaires ? Manipuler les relations dans les formulaires !

**Il nous reste à modifier le formulaire de création d’un article car pour l’instant dans ce formulaire on ne tient pas compte qu’il faut absolument qu’un article ai une catégorie et c’est d’ailleurs pareil pour quand je veux éditer un article.**

**(Form/ArticleType.php)**

**Je vais rajouter un nouveau champ :**

->add('category')

**Mais si je fais ça comme ça, je vais faire planter mon formulaire car Symfony n’arrive pas à deviner quel type de champ il doit sortir pour category puisque category est une relation. Ce n’est donc pas aussi simple que le title, le content et l’image. Ici, il faut préciser qu’est-ce que ce champ.**

[**https://symfony.com/doc/current/forms.html**](https://symfony.com/doc/current/forms.html)

**https://symfony.com/doc/current/reference/forms/types/entity.html**

**EntityType ⬄ Me permet de créer une select list ou une radio bouton etc qui me permet de présenter des objets qui sont dans la BDD. Donc moi ce que je veux faire c’est utiliser ce champ là.**

**Il me dit que si je veux l’utiliser je dis que le champ que je veux rajouter est de type EntityType ::class et je donne certaines options.**

->add('category', EntityType::class, [

**Et je ne dois pas oublier aussi de rajouter d’où vient Entity Type.**

use Symfony\Bridge\Doctrine\Form\Type\EntityType;

**J’actualise mon localhost** [**http://127.0.0.1:8000/blog/4/edit**](http://127.0.0.1:8000/blog/4/edit) **et je ne vois toujours pas ma catégorie qui s’affiche mais ça, ça vient de mon template.**

**Je vais donc dans (templates/blog/create.html.twig)**

**Et je n’oublie pas d’ajouter en dessous du title, une nouvelle ligne qui cette fois-ci ne s’intéressa à la catégorie et pour cette ligne je n’ai pas besoin du tableau d’option que j’avais placé sur les autres puisque je n’ai pas de placeholder.**

{{ form\_row(formArticle.category) }}

**Et si j’actualise maintenant, j’ai bien ma category qui apparait avec une liste déroulante avec les catégories possible.**

**Je peux modifier sa catégorie et ça fonctionne correctement.**

# Le composant Security de Symfony

* **Firewalls ⬄** 
  + - **Quelles parties de l’application on protège ?**
    - **Comment on protège ?**
* **Providers ⬄** 
  + - **Où sont les données des utilisateurs (annuaire LDAP, base de données, fichiers …) ?**
    - **Comment reconnaitre les utilisateurs ?**
* **Encoders ⬄**
  + - **Comment créer des hash, quel algorithme ?**
    - **Possibilités d’encodeurs différents en fonction des entités.**

**(config/packages/security.yaml)**

**Comment faire en sorte de pouvoir identifier les gens ? Il faut créer une entité Utilisateurs qui va stocker tout les utilisateurs.**

# L’entité User : On stocke nos users dans la BDD.

**php bin/console make:entity User**

**email – string – 255 - no**

**username – string – 255 – no**

**password – string – 255 – no**

**php bin/console make:migration**

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

**y**

# Inscription : Créons le formulaire d’inscription

**php bin/console make:form RegistrationType**

**Est-ce que ce formulaire va nous permettre de remplir une certaine entité dans notre application ? User**

**(Form/RegistrationType.php)**

**On va rajouter à la main :**

->add('confirm\_password')

**Attention : Si on rajoute un champ confirm\_password, la façon dont fonctionne nos formulaire Symfony, il faut aussi que ça existe dans l’entité des utilisateurs.**

**(Entity/User.php)**

**Rajouter un champ :**

public $confirm\_password;

**Et il ne faut surtout pas lui mettre de propriété ORM pour lui expliquer qu’il n’a rien à voir avec la BDD. C’est un champ de notre entité mais qui n’existe pas au sein de la BDD.**

\* @ORM\Column(type="x", length=x)

**On verra plus tard pour faire en sorte que lorsqu’une personne s’inscrit, que le mdp soit le même qui est dans confirm\_password.**

**Créer une action dans un controller pour afficher le formulaire.**

**php bin/console make:controller**

**On va y mettre dedans le formulaire d’inscription mais aussi le formulaire de connexion.**

**SecurityController**

**(Controller/SecurityController.php)**

**Supprimer la route index et créer une public function registration()**

**Création de l’affichage du formulaire. Créer : (templates/security/registration.html.twig)**

**Pour ne pas afficher le MDP lors de la saisie du formulaire. Aller dans : (Form/RegistrationType.php)**

->add('password', PasswordType::class)

**Extension VSCode : PHP Namespace Resolver ⬄ Extention VSCode qui permet de retrouver les namespaces.**

**RACCOURCIS / CTRL + ALT + i sur le nom pour afficher les namespaces disponible.**

**Je veux pouvoir cliquer sur Inscription maintenant.**

**(Controller/securityController.php)**

public function registration(Request $request, ObjectManager $manager)

// La fonctionne registration à besoin de :

// La requête HTTP => pour pouvoir l'analyser, pour que notre formulaire en tire les informations qui sont ressorti.

// L'objectManager de doctrine => qui va permettre d'enregistrer l'utilisateur en BDD.

**Attention : ObjectManager Deviens EntityManagerInterface**

use Doctrine\ORM\EntityManagerInterface; // A la place de ObjectManager car ne fonctionnait pas.

**Créer un utilisateur et vérifier son existence dans la BDD.**

**Je veux maintenant rajouter des validations pour être sur que le mdp est le même que la confirmation du mdp.**