# 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 :**

* **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 articles 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.**