



Symfony

MOUSSA CAMARA
MSA.CAMARA@GMAIL.COM

SYMFONY

- L'objectif de cette formation est de découvrir le framework symfony avec tous ces avantages et inconvénients.
- C'est quoi un framework déjà ?

FRAMEWORK

- Un Framework est une boîte à outils pour un développeur web. Frame signifie cadre et work se traduit par travail. Un Framework contient des composants autonomes qui permettent de faciliter le développement d'un site web ou d'une application. Ces composants résolvent des problèmes souvent rencontrés par les développeurs (CRUD, arborescence, normes, sécurités, etc.). Ils permettent donc de gagner du temps lors du développement du site.

QUEL INTÉRÊT À UTILISER UN FRAMEWORK POUR UN PROJET WEB ?

L'intérêt à utiliser un Framework lors du développement de votre projet web se situe à plusieurs niveaux :

- Rapidité : une base de travail existe déjà, donc le développeur web n'a pas besoin de partir de zéro pour créer votre site web.
- Flexibilité : vous pouvez choisir d'utiliser ou non certains composants du Framework pour améliorer le référencement naturel de votre site.
- Architecture : en utilisant un bon Framework, vous avez du code propre et fonctionnel qui ne ralentit pas le fonctionnement du site.
- Productivité : que ce soit un développement en solo ou en équipe, un Framework est un outil puissant puisque tout est parfaitement organisé.
- Communauté : vous bénéficiez de l'appui de toute une communauté en ligne (support et forum) qui vous aidera à corriger les bugs ou résoudre des problèmes de programmation.

QUELQUES FRAMEWORKS



Symfony



IL ETAIT UNE FOIS ... SYMFONY

- **Symfony est un puissant Framework** qui permet de réaliser des sites complexes rapidement, mais de façon structurée et avec un code clair et maintenable. En un mot : le paradis du développeur !
- Développé par [SensioLabs](#), la première de symfony commença en 2005
- Cette première version était surtout une collection de librairie PHP et quelques méthodes, une architecture assez simple et pas suffisamment normée. Manque de conventions.

IL ETAIT UNE FOIS ... SYMFONY 2

- Arrivé de la version 2 de Symfony ce juillet 2011
- Avec Symfony 2 plusieurs innovations sont arrivées
 - L'injection de dépendances
 - Tout est un Bundle dans Symfony
 - Bar de débug
 - Générateur interactif
 - Gestion de la sécurité
 - Les Normes du Web (Dont les spécifications HTTP, PHPUnit, PSR...)
- La dernière LTS (Long Term Support) de la version 2 est la 2.8 qui sera maintenu jusqu'en Nov. 2018
- N'hésitez pas à aller lire [l'annonce de la sortie](#) ou Fabien Potencier détaille les innovations de Symfony 2

IL ETAIT UNE FOIS ... SYMFONY 3

- Arrivée de la version 3 de Symfony en novembre 2015
- Contrairement à la version 2, Symfony 3 n'apporte pas de grosse évolution.
D'ailleurs Fabien Potencier a écrit dans [l'annonce de la sortie \(de la beta\)](#) :
 - « Considérez la version 3 comme une version 2.8 sans les couches dépréciés »
- Il existe une passerelle de migration assez simple (dépendant de la qualité du code) d'une version 2.X vers la 3.X
 - Un guide a été fait par l'équipe de Symfony et est accessible [ici](#).

IL ETAIT UNE FOIS ... SYMFONY 4

- Arrivée de la version 4 de Symfony en novembre 2017
- « Symfony 4.0 = Symfony 3.0 + toutes les fonctionnalités présentes dans la version 3.x — fonctionnalités obsolètes + une *nouvelle façon de développer des applications* »
- Plus besoin d'installer un outil tiers pour installer et démarrer un projet Symfony. Flex et composer viendront à votre rescousse! Avec la commande ci-dessous, nous créons une nouvelle application Symfony 4.

IL ETAIT UNE FOIS ... SYMFONY 5

- Le 21 novembre 2019, marque la **sortie de Symfony 5**.
- La mise à jour contient toutes les fonctionnalités prévues dans Symfony 4.4, et des fonctionnalités expérimentales.
- La série d'articles sur Symfony 4 sera mise à jour progressivement avec des précisions sur les différences pour les utilisateurs de Symfony 5.

IL EST UNE FOIS ... SYMFONY 6

- Réputé pour son architecture modulaire, sa flexibilité et sa large gamme de composants réutilisables, le framework Symfony a dévoilé fin mai sa dernière version, Symfony 6.3. Depuis sa dernière mise à jour majeure en novembre 2021, Symfony ne cesse de trouver des axes d'amélioration.
- Enfin, Symfony 6.0 permet une intégration renforcée des fournisseurs de services de notification. Beaucoup de bridges ont été ajoutés au composant Notifier (pour l'envoi de SMS, de messages sur des chats, etc.).

QUELQUES NOUVEAUTÉS

- Prise en charge de Bootstrap5 et Tailwind pour le rendu des formulaires. Vous pouvez désormais les utiliser en incluant le thème globalement dans la configuration twig ou localement dans le modèle de page.
- Nouvelles fonctionnalités dans le component String : trimPrefix et trimSuffix.
- Nouvelle classe Path appartenant au composant Filesystem, que vous pouvez utiliser pour normaliser l'accès aux répertoires et aux fichiers.
- Vous pouvez maintenant ajouter des paramètres à l'objet Translatable.
- Amélioration des bundle Doctrine orm qui fournissent une passe de compilation pour enregistrer les mapping pour les classes de modèles.
- Gestionnaires de messagerie configurables par le biais d'attributs, sans avoir à les ajouter à la configuration.
- Ajout des types PHP dans toutes les propriétés, this, arguments et valeurs du return des private, protected et public function chaque fois que possible
- Représentation des réponses des services HTTP au format json grâce à l'use de la class JsonResponse définie à true.
- Consommation par lot des messages du messager via l'interface BatchHandlerInterface.

LES NORMES PSR- PHP STANDARD REQUIREMENTS

- Les standards sont faits par un groupe qui discute des projets PHP et standardise le développement.
- Aujourd'hui les normes existantes sont 0, 1, 2, 3, 4, 6 et 7
- Les PSR-0 et PSR-4 concernent l'autoload.
- A noter que PSR-0 est déprécié en faveur de PSR-4 mais le PSR-0 continue à fonctionner
- Symfony utilise Composer pour l'autoload, ce qui fait que les 2 normes PSR-0 et 4 restent valable car Composer gère très bien les 2.
- Le site officiel: <http://www.php-fig.org/>

LES NAMESPACES

- Le namespace représente **le répertoire dans lequel se trouve le fichier**. C'est une sorte de chemin virtuel donné à php pour faciliter l'écriture lors l'importation de fichier traditionnellement accompli par la fonction "include()", qui sera remplacé par la fonction "use".

LES NAMESPACES

- Les espaces de nom ont été ajouté pour résoudre un problème du langage.
- Le problème:
 - Nous ne pouvions avant les espaces de nom avoir des constantes ou classe portant le même nom dans notre application. Sinon PHP nous lancera une exception.
- La solution:
 - Utiliser les espaces de nom résout ces problèmes.
- Si vous ne connaissez pas les namespace, je vous encourage à lire [ce tuto](#) qui traite du sujet

COMPOSER

- Composer est un Gestionnaire de dépendance.
- Il gère les dépendances qu'on lui demande de gérer via un fichier `composer.json`
- Composer n'est pas un gestionnaire de paquets comme Yum ou Apt. Il ne gère les dépendances que localement et non globalement comme le ferait un gestionnaire de paquets
- En plus de gérer les dépendances, il s'occupe aussi de faire l'autoload pour nous 😊
- Pour avoir un système d'autoload gratuitement, il nous suffira de faire un import du fichier autoload de composer qu'il place dans le dossier `vendor/autoload.php`
- N'hésitez pas à aller voir le [site officiel](#)

LES MÉTHODES HTTP

- Ce qui nous intéresse dans les méthodes HTTP
- Méthodes:
 - GET
 - POST
 - PUT/PATCH/EDIT
 - DELETE
- A noter qu'il existe d'autres méthodes du protocole qui sont aujourd'hui encore assez peu utilisé. (OPTIONS, CONNECT, TRACE, PATCH, HEAD)

API REST

- Le REST signifie “Representational State Transfer” est un style d’architecture qui repose sur le protocole HTTP créé en 2000 par Roy Fielding. [Source de données fiable](#)
- Le REST est bien une architecture et non une technologie. Il impose les contraintes suivantes:
 - la séparation entre le client et le serveur
 - Est sans état
 - Gestion du cache
 - Une interface uniforme
 - Un système hiérarchisé par couche
 - Code-on-demand (facultatif)
- A noter qu'il existe une autre architecture appelé [SOAP](#) de moins en moins utilisé mais qui reste tout de même une référence.

REST & HTTP

- Au final en combinant le REST et le protocole HTTP, nous allons avoir des URLs qui ressembleront à ça:
- `http://mondomaine.com/ressource/identifiant/actions`
- En utilisant les différentes méthodes du protocole HTTP notre serveur sera capable de traiter les demandes du client.
- Exemple:
 - `http://webforce.com/formateur/moussa/edit`
 - => [GET] => affiche le formulaire pour modifier les informations de la ressource identifiée
 - => [PUT] => modifie la ressource identifiée

TWIG

- Twig est un moteur de template en PHP fait par Sensiolabs
- Il facilite le développement front end et est plus simple à apprendre.
- L'un des objectifs de Sensiolabs était de faire un moteur de template avec une syntaxe qu'on retrouve souvent dans d'autres langages de sorte que les développeurs front-end puisse développer sans avoir à connaître le PHP
- Il existe d'autres moteurs de template comme Blade, Mustache ou encore Smarty
- A noter:
 - Blade est le moteur de template de Laravel
 - Mustache est un moteur de template logic-less et portable (utilisable dans presque tous les langages)
 - Smarty, un moteur de template très répandu et réputé plus performant que Twig (dans un comparatif fait en 2011... donc à revoir car depuis Twig a beaucoup évolué)
- Doc et site officiel : <http://twig.sensiolabs.org/>

SYMFONY – INSTALLATION SYSTÈME

- Pour votre environnement de développement et production vous devez:
 - avoir au moins PHP 8
 - Pour votre environnement de développement vous devez avoir:
 - Composer installé localement ou globalement
 - Git (Recommandé)
- Son installation a un peu évolué par rapport aux versions antérieures

<https://symfony.com/doc/current/setup.html>

SYMFONY – CONFIGURATION

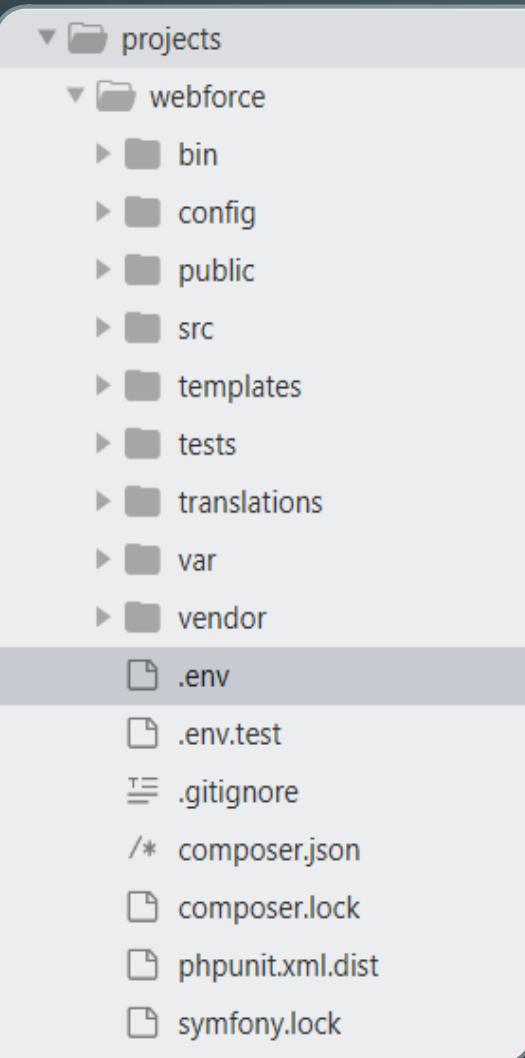
- Une fois votre projet installé et que vous avez vérifier que votre environnement est bien configuré, vous devez maintenant configurer votre nouvelle application.
- Pour ce faire vous allez modifier le fichier .env dans lequel vous allez indiquer quelques informations dont l'application aura besoin par la suite
- **IMPORTANT:**
 - VOUS NE DEVEZ PAS VERSIONNER LE FICHIER .env

INSTALLATION

- Lancer la commande et commençons notre prière:

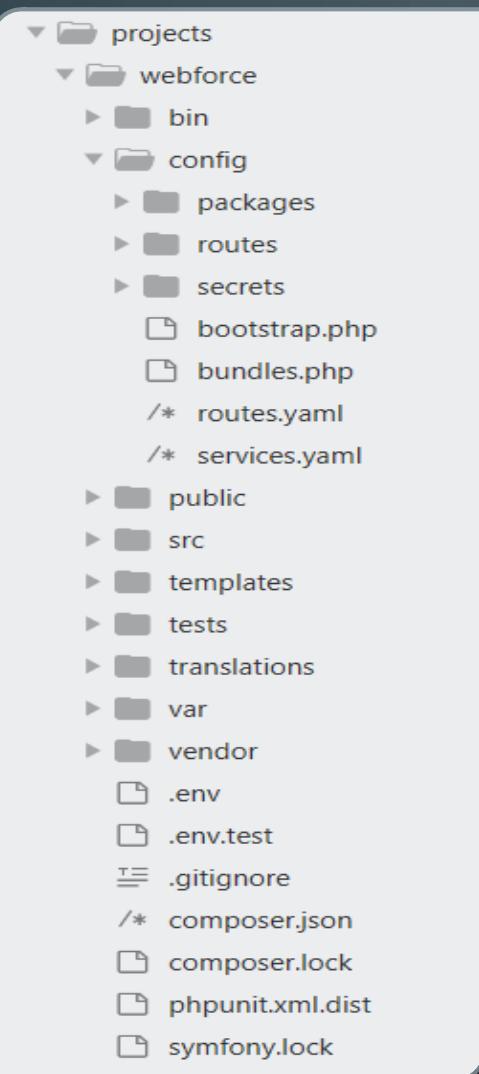
- `symfony new my_project_directory --version="6.3.*" --webapp`
- ou
 - `composer create-project symfony/skeleton:"6.3.*" my_project_directory`
 - `cd my_project_directory`
 - `composer require webapp`

SYMFONY - ARCHITECTURE



- Le dossier bin/ contient les exécutables
- Le dossier config/ contient les fichiers de config
- Le dossier public contient le fichier dans lequel vous devez ajouter les Bundles que vous créer ou importer afin que Symfony puisse les trouver quand il en a besoin
- Le dossier src/ contiens le code que vous allez développer. C'est dans ce dossier que vous passerez 95% de votre temps
- Le dossier Templates lui contient des layouts avec une haute priorité d'exécution. (On verra ce que ça veut dire)
- Le dossier vendor/ qui contient toutes librairies ou bundle téléchargé via composer dont Symfony qui est un bundle
- Le dossier var/ lui contient les logs, caches et sessions

SYMFONY - ARCHITECTURE



- Le fichier `routing.yml` est le fichier dans lesquels vous définissez des routes pour l'environnement de travail.
- Le fichier `services.yml` est le fichier dans lequel vous configurer vos services.

SYMFONY – BASE DE DONNÉE

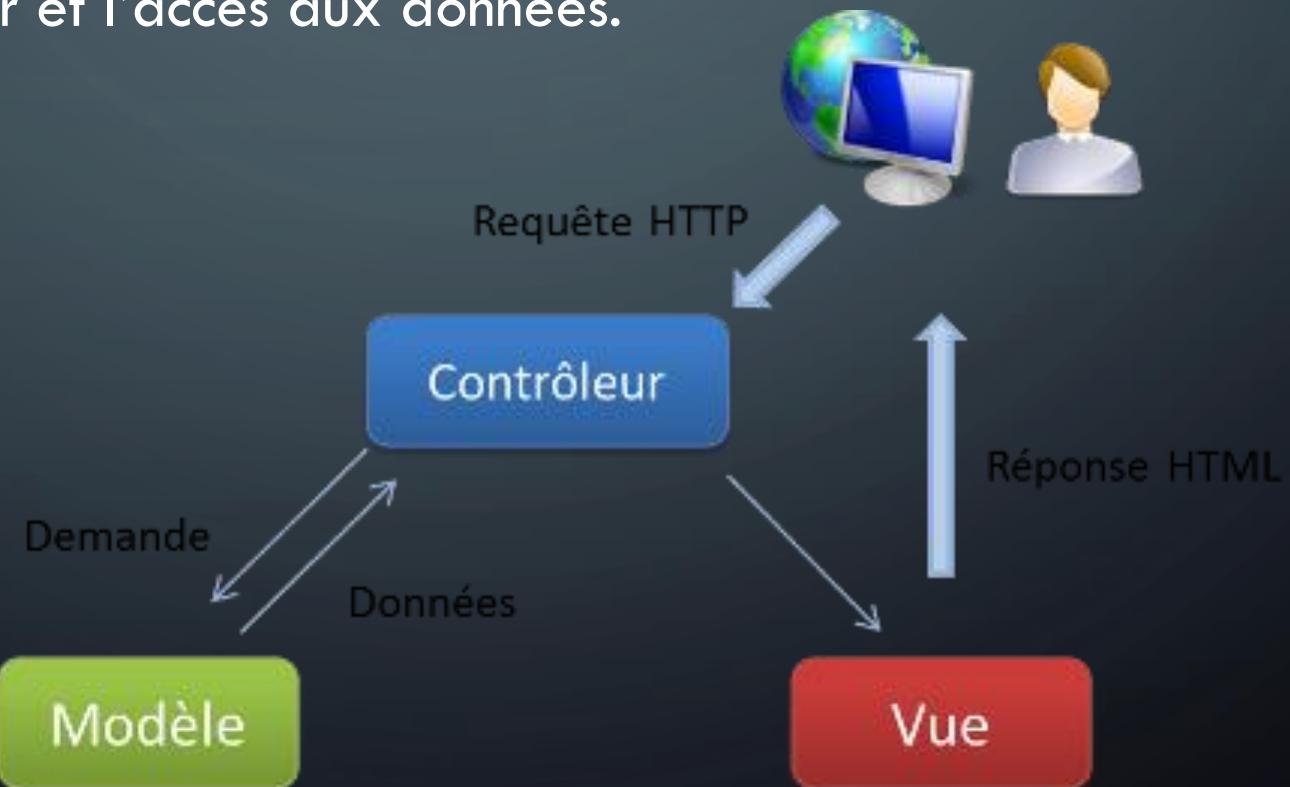
- Créons notre base de données
 - Définir les accès et le nom de votre base de données dans le fichier .env
 - `php bin/console doctrine:database:create` ou
 - `php bin/console d:d:c`

MVC

- Symfony s'appuie sur l'architecture MVC (Modèle Vue Contrôleur)
- Avant de nous lancer dans un développement, clarifions ce motif de conception (pattern design) dit Modèle-Vue-Contrôleur (MVC). Il est maintenant très répandu et accepté sans contestation comme un de ceux menant, notamment pour la réalisation de sites Web dynamiques, à une organisation satisfaisant le but recherché d'une organisation rigoureuse et logique du code.
- Programmer en utilisant MVC sépare votre application en 3 couches principales

MVC -

- MVC est un patron de conception (*design pattern* en anglais) très répandu pour réaliser des sites web. Ce patron de conception est une solution éprouvée et reconnue permettant de séparer l'affichage des informations, les actions de l'utilisateur et l'accès aux données.



MVC - CONTROLLER

- La couche Controller gère les requêtes des utilisateurs. Elle est responsable de retourner une réponse avec l'aide mutuelle des couches Model et Vue.
- Les Controllers peuvent être imaginés comme des managers qui ont pour mission que toutes les ressources souhaitées pour accomplir une tâche soient déléguées aux travailleurs corrects. Il attend des requêtes des clients, vérifie leur validité selon l'authentification et les règles d'autorisation, délèguent les données récupérées et traitées par le Model, et sélectionne les types de présentation correctes que le client accepte, pour finalement déléguer le processus d'affichage à la couche Vue.
- C'est aussi l'intermédiaire principal entre la vue et le modèle. Par exemple, la vue soumet un formulaire au contrôleur, qui gère sa validation via du code métier, et demande au modèle de faire des modifications dans la base de données.

MVC – MODELE /ENTITE

- La couche Model représente la partie de l'application qui exécute la logique métier. Cela signifie qu'elle est responsable de récupérer les données, de les convertir selon des concepts chargés de sens pour votre application, tels que le traitement, la validation, l'association et beaucoup d'autres tâches concernant la manipulation des données.
- A première vue, l'objet Model peut être vu comme la première couche d'interaction avec n'importe quelle base de données que vous pourriez utiliser pour votre application. Mais plus globalement, il fait partie des concepts majeurs autour desquels vous allez exécuter votre application.

MVC - VUE

- La Vue retourne une présentation des données venant du model. Etant séparée par les Objets Model, elle est responsable de l'utilisation des informations dont elle dispose pour produire une interface de présentation de votre application.
- Par exemple, de la même manière que la couche Model retourne un ensemble de données, la Vue utilise ces données pour fournir une page HTML les contenant. Ou un résultat XML formaté pour que d'autres l'utilisent.
- La couche Vue n'est pas seulement limitée au HTML ou à la représentation en texte de données. Elle peut aussi être utilisée pour offrir une grande variété de formats en fonction de vos besoins, comme les vidéos, la musique, les documents et tout autre format auquel vous pouvez penser.

EXEMPLE

- On a défini ce que sont les trois composantes du pattern, mais ça reste une explication théorique. Pour mieux comprendre ce fonctionnement, nous allons voir un cas d'utilisation concret.
- Prenons l'exemple de application web d'une banque.
 1. On aurait dans les models les modèles de données des comptes, clients, transactions, etc. Ainsi que tout ce qui permet de les transformer (changer l'adresse d'un client, ajouter ou débiter une somme sur son compte, etc.).
 2. Les vues seraient les interfaces auxquelles accéderaient les utilisateurs : page du compte, historique des transactions, formulaire de virement, etc.
 3. Et les contrôleurs, eux, seraient les responsables de toute la logique : vérification que le format de l'adresse est correct pour le changement, vérification du provisionnement d'un compte pour le virement, etc.
- Voici un exemple d'utilisation : l'utilisateur se connecte via une vue, le contrôleur vérifie que ses identifiants sont corrects, il crée le modèle de l'utilisateur et renvoie une autre vue avec les données remplies.

SYMFONY – LES ENTITY (MODELS)

- Avant d'aller voir ce que c'est exactement Symfony, l'architecture de fichier etc... faisons une petite application rapidement juste histoire d'avoir un peu de vocabulaire pour la suite.
- Vous allez juste exécuter les commandes que je vous indiquerai.

SYMFONY - FRIEND ENTITY

- Fields
 - Name
 - Email

SYMFONY - CREATION ENTITY

- `php bin/console make:entity`

Class name of the entity to create or update (e.g. FierceElephant):

> `Friend`

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

> `name`

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

>

Field length [255]:

>

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

>

updated: src/Entity/Friend.php

SYMFONY - UPDATE DATABASE

- `php bin/console doctrine:schema:update --dump-sql`
 - `php bin/console d:s:u --dump-sql`
- `php bin/console doctrine:schema:update --force`

SYMFONY - CREATION CRUD

- `php bin/console make:crud` ou `php bin/console m:cr`
 - The class name of the entity to create CRUD (e.g. VictoriousKangaroo):
 - > **Friend**

SYMFONY - FRIEND CRUD

- created: src/Controller/FriendController.php
created: src/Form/FriendType.php
created: templates/friend/_delete_form.html.twig
created: templates/friend/_form.html.twig
created: templates/friend/edit.html.twig
created: templates/friend/index.html.twig
created: templates/friend/new.html.twig
created: templates/friend/show.html.twig
- Accessible path **/friend**

SYMFONY - FRIEND ENTITY (ADD TEL)

- Supposons qu'on souhaite ajouter un numéro de téléphone à notre ami
- // Entity/Friend.php
- #[ORM\Column(length: 15)]
- private ?string \$phone = null;

SYMFONY - FRIEND ENTITY (ADD TEL)

- Terminal

- `php bin/console doctrine:schema:update --dump-sql`
- `php bin/console doctrine:schema:update --force`

```
vagrant@ubuntu-xenial:/var/www/projects/webforce$ php bin/console doc
                                         Application Migrations

WARNING! You are about to execute a database migration that could res
Migrating up to 20191203154944 from 0

++ migrating 20191203154944

    -> ALTER TABLE friend ADD tel VARCHAR(255) NOT NULL

++ migrated (took 987.7ms, used 16M memory)

-----
++ finished in 1077.7ms
++ used 16M memory
++ 1 migrations executed
++ 1 sql queries
```

SYMFONY - FRIEND ENTITY (ADD TEL)

```
//Form/FriendType.php
```

- `->add('phone', Form\TextType::class, [`
- `'label' => "Entrez votre tél",`
- `'required' => true,`
- `])`

SYMFONY - FRIEND ENTITY (ADD TEL)

```
// .../index.html.twig  
  
<th>Tel</th>  
<td>{{ entity.phone }}</td>
```

```
// .../show.html.twig  
<tr>  
    <th>Tel</th>  
    <td>{{ entity.phone }}</td>  
</tr>
```

QUELQUES COMMANDES PRATIQUES

\$ php bin/console cache:clear: Pour vider les caches

\$ php bin/console debug:router: Pour lister toutes les routes de l'application

\$ php bin/console doctrine:database:create: Création de la Base de donnée

\$ php bin/console doctrine:schema:update --dump-sql: Afficher la requete avant de l'exécuter

\$ php bin/console doctrine:schema:update –force: Exécute la requete

\$ php bin/console make:entity: Création de mon model de donnée

\$ php bin/console make:crud: Create Read Update Delete

REVIEW

- Revoyons ce qu'on a fait:
 - On a généré l'architecture MVC avec Symfony.
 - Ensuite on a généré une entité (un model) avec Symfony.
 - Généré le CRUD
 - Modifié l'entité
- Avec ces quelques lignes de commandes nous avons la une application fonctionnelle. Il nous suffit juste de faire un peu de personnalisation et d'envoyer en prod. ☺

Lets build a blog...



LETS HAVE SOME FUN...



SYMFONY - PREMIERE APPLICATION

On va maintenant créer une première application “Hello word”

Pour cela, on va créer un contrôleur avec une route et sa vue

SYMFONY - PREMIERE APPLICATION

Php bin/console make:controller

```
• class HomeController extends AbstractController
• {
•     #[Route('/', name: 'home')]
•     public function index(CategoryRepository
• $categoryRepo): Response
•     {
•         return $this-
• >render('home/index.html.twig');
•     }
• }
```

SYMFONY - CONTROLLEUR

- Toute action doit obligatoirement retourner une Réponse.
- `$this->render` est une méthode de l'objet Response qui permet de retourner un template
- Ici nous retournons dans template le fichier `index.html.twig` dans le dossier `home`

SYMFONY - TWIG

- Les templates vont nous permettre de séparer le code PHP du code HTML/XML/Text, etc.
- le moteur de templates Twig offre son pseudo-langage à lui. Ce n'est pas du PHP, mais c'est plus adapté

Twig:
{{ mavar }}

PHP:
<?php echo \$mavar; ?>

SYMFONY - TWIG

- Avec twig nous allons pouvoir par exemple définir des variables, afficher des valeurs, boucler sur les tableaux...



TWIG – DÉFINIR UNE VARIABLE

- Comment définir une variable dans notre template ?

```
{% set myVariable = "Contenu d'une variable définie en twig" %}  
{{ myVariable }}
```

TWIG – BOUCLE

```
{% for i in range(0, 10, step=2) %}  
    {{ loop.index }} => {{ i }}<br>  
{% endfor %}
```

TWIG – BOUCLE SUR UN TABLEAU

```
{% set myArray = ["Moussa", "Florence" , "Arsène", "Lupin"] %}
```

```
{% for nom in myArray %}  
    {{ loop.index }} => {{ nom }}<br>  
{% endfor %}
```

TWIG – FILTER | UPPER (EN MAJUSCULE)

```
{% set myArray = ["Moussa", "Florence", "arsène", "lupin"] %}  
{% for nom in myArray %}  
    {{ loop.index }} => <b>nom upper</b>: {{ nom|upper }}  
{% endfor %}
```

TWIG – FILTER | LENGTH

```
{% set myArray = ["alfred", '<i>hitchcock</i>', 'arsène', 'lupin'] %}

{% for nom in myArray %}

    {{ loop.index }} =>

        <b>nom upper             </b>: {{ nom|upper }} ||
        <b>nom raw               </b>: {{ nom|raw }} ||
        <b>nom raw title         </b>: {{ nom|raw|title }} ||
        <b>nom striptags         </b>: {{ nom|striptags }} ||
        <b>nom striptags length</b>: {{ nom|striptags|length }} ||
        <b>nom length             </b>: {{ nom|length }} ||

        <br><br>

    {% endfor %}
```

TWIG – FILTER | DATES

```
 {{ "now" | date(null, "Europe/Paris") }}
```

```
<br>
```

```
<br>
```

```
 {{ "now" | date('D d/m/y H:i:s', "Europe/Paris") }}
```

TWIG – FILTERS

<https://twig.symfony.com/doc/3.x/>



TWIG – INCLUDES

En plus des méthodes et filtres, twig nous permet d'aller plus loin comme inclure des templates

Pour ce faire nous utiliseront une méthode ‘include’

Créons d'abord un template à inclure `fileToInclude.html.twig` dans notre répertoire templates et rajoutons un peu de contenu dans ce fichier

Rajoutons du contenu « lorem » dans ce fichier

TWIG – INCLUDES

Dans notre fichier `index.html.twig` rajoutons la ligne suivante:

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

Et maintenant en rafraîchissant notre navigateur nous avons bien le contenu du fichier `fileToInclude` qui s'affiche

Notez que lorsque vous faites un `include` vous devez donner le chemin complet vers le fichier à inclure comme ici on indique à Symfony que nous voulons inclure un fichier `fileToInclude.html.twig` qui se trouve dans le dossier `templates`

TWIG – EXTENDS && BLOCKS

- Dans notre `hello/index.html.twig` on va
 - entourer tout le contenu de `block body`. Ne pas oublier le `endblock` à la fin de la page

```
{% block body %}  
Tout notre contenu ici  
{% endblock %}
```
 - On a toujours au tout début de la page la ligne suivante:
 - `{% extends 'base.html.twig' %}`

TWIG – EXTENDS && BLOCKS

- Il arrive souvent qu'on ait besoin de surcharger une partie d'une vue, par exemple vous avez un contenu qui change dépendant du template utilisé.
- Twig nous permet de surcharger le contenu d'un parent ainsi.
- Rajoutons dans notre layout.html.twig le code suivant:

```
{% block header %}  
    <h1>Header</h1>  
    <hr>  
{% endblock %}
```

```
{% block navigation %}
```

- En rafraîchissant notre navigateur, nous pouvons voir afficher le titre 'Header'

TWIG – EXTENDS && BLOCKS

- Nous pouvons surcharger ce contenu depuis notre page `hello.html.twig` en créant un `block` avec le même nom que le `block` que nous voulons surcharger

```
{% block header %}
```

```
<h3>Extends</h3>
```

```
This part has been overriden
```

```
<hr>
```

```
{% endblock %}
```

LES ASSETS

- Pour charger nos css et images qui sont dans le dossier public
- Symfony nous propose une solution => **asset**
- La méthode asset nous retourne le dossier public. Donc si je fais
 - `{{ asset('images') }}` => ça retourne le dossier ./public/images

SYMFONY - BLOG



CAHIER DES CHARGES

- Nous allons créer notre blog d'articles.
 - Nous allons avoir besoin d'un système de gestion des utilisateurs.
 - Les utilisateurs non identifiés pourront lire les articles (Post)
 - Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
 - Les utilisateurs identifiés pourront commenter des articles (Post)
-
- Pendant la phase de développement, nous aurons la possibilité de charger du contenu pré-défini à chaque fois qu'on aura besoin étant donné qu'on videra très souvent notre base de données

CAHIER DES CHARGES

Nous allons commencer par la gestion des utilisateurs avec un menu de navigation

- Si l'utilisateur est connecté, afficher « *Bonjour Moussa || Se déconnecter* »
- Sinon afficher les boutons: *Se connecter || S'inscrire*



CAHIER DES CHARGES

- Commençons par la partie inscription car il faut bien un compte avant de pouvoir se connecter.



CAHIER DES CHARGES

- Gestion des utilisateurs:
 - Inscription – Cr茅ation du formulaire de register
 - Make:registration-form
- NB:
 - Nous allons utiliser la commande make:user de symfony qui g茅e automatiquement l'email, le role et le mot de passe.
- On peut ajouter d'autres entit茅s toujours make:entity User
 - Civility
 - Firstname
 - Lastname
 - Adresse
 - CP
 - State

CONNECTION

- Gestion des utilisateurs:
 - connection –
 - make:auth pour mettre à jour la configuration de sécurité, générer un template pour la connexion et créer une classe d'authentification (authenticator)

<https://symfony.com/doc/current/the-fast-track/fr/15-security.html#configurer-le-système-d-authentification>

BLOG – WHAT NEXT?

- Nous avons une structure pour commencer notre application.
 - Commençons par les articles (Posts)
-
- A votre avis par quoi devons-nous commencer ?

CAHIER DES CHARGES

- Nous allons créer notre blog d'articles.
- ~~Nous allons avoir besoin d'un système de gestion des utilisateurs.~~
- Les utilisateurs non identifiés pourront lire les articles (Post)
- Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
- Les utilisateurs identifiés pourront commenter des articles (Post)
- Pendant la phase de développement, nous aurons la possibilité de charger du contenu pré-défini à chaque fois qu'on aura besoin étant donné qu'on videra très souvent notre base de données

BLOG – WHAT NEXT?

- Nous allons créer des posts (Article).
- Effectivement on ne peut pas commencer par les lire car on n'a pas de post à lire.

WHAT DO WE NEED ?

Dans notre blog, on doit pouvoir:

- Poster un article
- Lister les articles
- Voir un article
- Effacer un article

En résumé un CRUD pour les Articles

LES ARTICLES - POSTS



Chaque article est un 'post'



On va donc avoir besoin d'une entité Post dans notre blog qui lui gèrera les Posts



Chaque Post devra contenir:

- 1 Titre
- 1 résumé du contenu
- 1 contenu
- Date de création
- 1 auteur
- 1 image



Tout le CRUD pourra être géré à partir des routes préfixé de /post

TP – CREATION DE POST

- Ajoutons maintenant notre entité Article
 - php bin/console ...
 - Et on rajoute les champs correspondants
 - titre (string)
 - résumé (string)
 - contenu (text)
 - created_at (datetime)
 - auteur (text)
 - image (text)

RAPPEL

- Pour interagir avec la base de données depuis PHP, nous allons nous appuyer sur Doctrine, un ensemble de bibliothèques qui nous aide à gérer les bases de données : Doctrine DBAL (une couche d'abstraction de la base de données), Doctrine ORM (une librairie pour manipuler le contenu de notre base de données en utilisant des objets PHP), et Doctrine Migrations.
- **L'entité** est un fichier PHP contenant des propriétés, des getters et setters. Ce fichier est la structure de notre table SQL, c'est avec celui-ci que Doctrine va travailler.
- Elles sont au centre des modèles. C'est à travers elles que vous allez manipuler tous les éléments propres à votre application (utilisateurs, produits, articles, messages, etc.).
 - Si je dois créer un blog, je vais gérer mes articles via l'entité Article
 - Si je crée un site e-commerce, je vais gérer mes produits via l'entité Product
 - Si je crée un forum je vais gérer mes messages via l'entité Post

- MAKE: CRUD
- Le terme **CRUD** désigne l'ensemble des opérations que l'on va pouvoir effectuer sur une ressource :
 - Création (**Create**)
 - Récupération (**Read**)
 - Modification (**Update**)
 - Suppression (**Delete**)

LES ARTICLES - POSTS

- Php bin/console make:entity
- Comme nous avons un changement dans notre modèle de données (suite à l'ajout de notre entité Post), il nous faut mettre à jour notre base de données:

POST - CRUD

- Désormais de nouvelles routes sont disponible.
- En vous rendant sur la l'url **/post** vous trouverez la liste des posts ou articles

POST - CRUD

- Nous pouvons maintenant:
 - Lister les Posts
 - Voir un Post
 - Ajouter un Post
 - Modifier un Post
 - Supprimer un Post

POST -

- Nous avons maintenant notre page des articles. Par contre on a quelques petits réglages.
 - Mettre la date du jour
 - Faire un peu de style (Form)
- <https://symfony.com/doc/current/reference/forms/types.html>

- Pour mettre la date du jour, il nous faut le constructeur

```
public function __construct(){
    $this->createdAt = new \DateTimeImmutable();
}
```

CUSTOM FORM

- Pour customiser notre formulaire, on peut utiliser `form_row`
- Par défaut, nous avons `form_widget` qui prend l'ensemble du form (tous les champs)
- Avec `form_row`, on peut récupérer champs par champs

```
<div class="row">
    <div class="col-md-6">
        {{ form_label(form.resume, 'Saisir résumé', {'label_attr': {'class': 'form-label'}}) }}

        {{ form_row(form.resume, {'attr': {'class': 'form-control', 'placeholder': "Résumé"}) }}}
    </div>

    <div class="col-md-6">
        {{ form_row(form.contenu, {'attr': {'class': 'form-control', 'placeholder': "Contenu"}) }}}
    </div>
</div>
```

CUSTOM FORM - VALIDATION

- Nous pouvons aussi aller plus loin avec les validations des formulaires
 - [composer require symfony/validator](#)
- Une fois intégré nous pouvons ainsi utiliser les ASSERT

```
#[ORM\Column(length: 100)]
#[Assert\NotBlank(message:'Le titre ne doit pas être null')]
#[Assert\Length(
    min: 5,
    max: 10,
    minMessage: 'Your first name must be at least {{ limit }} characters long',
    maxMessage: 'Your first name cannot be longer than {{ limit }} characters',
)]
private ?string $titre = null;
```

```
->add('titre', TextType::class, [
    'label' => false,
    'required' => false,
])
```

<https://symfony.com/doc/current/reference/constraints.html>

CUSTOM FORM

- Pour avoir le style sur les messages d'erreurs, rajoutez dans config/packages/twig.yaml
 - `form_themes: ['bootstrap_5_layout.html.twig']`

Dans notre formulaire, on aura:

```
 {{ form_start(form, {'attr': {'novalidate': 'novalidate'}}) }}  
  
    <div class="row">  
        <div class="col">  
            {{ form_row(form.titre, {'attr': {'class': 'form-control'}}) }}  
        </div>  
    </div>
```

POST – IMAGE UPLOAD

- Jusqu'ici nous avons un champ de saisi (input) pour l'image.
- Nous allons permettre à l'utilisateur d'uploader une image et pour cela nous allons utiliser un bundle.
- On préfère utiliser un bundle comme [VichUploaderBundle](#)
- <https://github.com/dustin10/VichUploaderBundle/blob/master/docs/installation.md>

POST – IMAGE UPLOAD

- Configuration de notre bundle

```
# config/packages/vich_uploader.yaml
```

```
vich_uploader:  
    db_driver: orm  
  
    mappings:  
        article:  
            uri_prefix: /images/article  
            upload_destination: '%kernel.project_dir%/public/images/article'  
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
```

POST – IMAGE UPLOAD

- Pour commencer nous aurons besoin de UploadFile des du validateur de symfony dans notre entité. Commençons par les importer
 - use Symfony\Component\HttpFoundation\File\File;
 - use Vich\UploaderBundle\Mapping\Annotation as Vich;

POST – IMAGE UPLOAD

- Ensuite nous avons besoin de dire à doctrine que notre entité contient des annotations sinon il ne les interprétera pas.

```
#[ORM\Entity(repositoryClass:  
ArticleRepository::class)]  
#[Vich\Uploadable]  
class Article  
{
```

POST – IMAGE UPLOAD

- En lisant la doc, j'ai vu qu'il me faut 2 attributs qui serviront à stocker l'image.

```
#[ORM\Column(length: 255)]  
private ?string $image = null;  
  
#[Vich\UploadableField(mapping: 'article', fileNameProperty: 'image')]  
private ?File $imageFile = null;
```

```
public function getImage(): ?string
{
    return $this->image;
}

public function setImage(string $image): static
{
    $this->image = $image;

    return $this;
}

/**
 * @param File|\Symfony\Component\HttpFoundation\File\UploadedFile|null $imageFile
 */
public function setImageFile(?File $imageFile = null): void
{
    $this->imageFile = $imageFile;
}

public function getImageFile(): ?File
{
    return $this->imageFile;
}
```

POST – IMAGE UPLOAD

- On peut modifier le formulaire afin d'avoir le bouton upload d'image
 - On importe avant la classe vichImageType
 - `use Vich\UploaderBundle\Form\Type\VichImageType;`
 - Ensuite:
`->add('imageFile', VichImageType::class)`

POST – IMAGE UPLOAD

- A noter on a maintenant un nouveau dossier dans **public/images/articles** pour stocker nos images de posts
- Enfin, il faut mettre à jour la vue `index.html.twig`
 - `<td></td>`

POST – MESSAGE FLASH

- Commençons par notre contrôleur

```
if ($form->isSubmitted() && $form->isValid()) {  
    $entityManager->persist($article);  
    $entityManager->flush();  
  
    $this->addFlash('success', 'Votre article '.$article->getTitre().' a été  
bien ajouté');  
  
    return $this->redirectToRoute('app_article_index', [],  
Response::HTTP_SEE_OTHER);  
}
```

POST – MESSAGE FLASH

- Affichage du message dans base.html.twig

```
{% for message in app.flashes('success') %}  
    <div class="success alert alert-success">  
        {{ message }}  
    </div>  
{% endfor %}
```

BLOG – CAHIER DES CHARGES

- ~~Pour notre blog, nous allons avoir besoin d'un système de gestion des utilisateurs.~~
- ~~Les utilisateurs non identifiés pourront lire les articles (Post)~~
- Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
- Les utilisateurs identifiés pourront commenter des articles (Post)
- Pendant la phase de développement, nous aurons la possibilité de charger du contenu pré-défini à chaque fois qu'on aura besoin étant donné qu'on videra très souvent notre base de données

POST – USER

- Pour le moment, nous avons un champ de saisi (input) pour le nom de l'auteur dans lequel on peut mettre n'importe qui comme auteur de l'article. 😞
- Or dans notre cahier des charges, seuls les utilisateurs identifiés et autorisés devraient pouvoir poster des articles
- Commençons par l'authentification. On s'occupera de l'autorisation plus tard

POST – USER

- Il va falloir pouvoir dire à notre application que si:
 - Un utilisateur n'est pas connecté, il ne doit pas pouvoir accéder au formulaire de création d'un Post.
 - Un utilisateur est connecté, lorsqu'il crée un Post, le Post doit lui être associé
- Cela implique qu'un Post est forcément lié à un utilisateur
- Un utilisateur devrait pouvoir poster plusieurs Posts.

- Nous avons précédemment créé une entité isolée : elle n'a pas de lien avec d'autres entités de notre base de données. Mais en réalité, nous avons souvent besoin de mettre en place une structure plus complexe. On va alors créer des relations entre nos entités afin qu'elles puissent interagir entre elles.

DOCTRINE – ASSOCIATION MAPPING

- Doctrine nous propose un système de relation entre les entités.
- Comme nous pouvons avoir plusieurs types de relations entre nos entités, doctrine nous propose une solution pour chaque type de relation.
- <https://symfony.com/doc/current/doctrine/associations.html>

DOCTRINE – ASSOCIATION MAPPING

- Many-To-One
- One-To-One
- One-To-Many
- Many-To-Many

```
~/Sites/the_spacebar — php -v php ./bin/console server:run          ~/Sites/the_spacebar — php ./bin/console make:entity
ManyToOne   Each Comment relates to (has) one Article.  
Each Article can relate/has to (have) many Comment objects  
  
OneToMany   Each Comment relates can relate to (have) many Article objects.  
Each Article relates to (has) one Comment  
  
ManyToMany  Each Comment relates can relate to (have) many Article objects.  
Each Article can also relate to (have) many Comment objects  
  
OneToOne    Each Comment relates to (has) exactly one Article.  
Each Article also relates to (has) exactly one Comment.  
  
-----  
  
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:  
> ManyToOne  
  
Is the Comment.article property allowed to be null (nullable)? (yes/no) [yes]:  
> no  
  
Do you want to add a new property to Article so that you can access/update Comment object  
s from it - e.g. $article->getComments()? (yes/no) [yes]:  
>
```

DOCTRINE – ASSOCIATION MAPPING

- On distingue 4 grands types de relations possibles, variant selon le nombre de liens entre entités : on parle de multiplicité.
 - **One-To-One (1-1)** : deux entités A et B sont liées de manière unique.
 - **Many-To-One (n-1)** : plusieurs entités A peuvent être liées à une unique entité B.
 - **One-To-Many (1-n)** : une entité A peut être liée à plusieurs entités B.
 - **Many-To-Many (n-n)** : plusieurs entités A peuvent être liées à plusieurs entités B.
- Outre le nombre de liens existant entre 2 entités, une notion de direction de la relation entre en jeu.
- On distingue deux types de directions :
 - Unidirectionnelle
 - Bidirectionnelle

DOCTRINE – ASSOCIATION MAPPING

- **Unidirectionnelle:** Cela signifie que l'entité A a accès à une entité B et non l'inverse. A est l'entité propriétaire de la relation, c'est celle qui « possède » l'autre.
- **Bidirectionnelle:** Cela signifie qu'une entité A a accès à une entité B et inversement. Dans le cas d'une relation bidirectionnelle, on définit en plus de l'entité propriétaire une entité inverse, c'est celle qui « est possédée » par l'autre.
 - Dans le cas des relations 1-1 : l'entité correspondant à la table qui contient la clé étrangère est propriétaire.
 - Dans le cas des relations 1-n et n-1 : l'entité du côté n est toujours propriétaire et l'entité du côté 1 est toujours inverse.

DOCTRINE – ASSOCIATION MAPPING

- Dans notre cas, un utilisateur peut avoir plusieurs posts et un post ne peut avoir qu'un utilisateur.
 - C'est donc un Many-To-One dans notre entité Post.
-
- #[ORM\ManyToOne(inversedBy: 'articles')]
 - private ?User \$Auteur = null;

DOCTRINE – POSTS

- Maintenant on va créer un nouveau Post et on va aller voir ce qu'il y a dans notre base de données comment ça s'est rempli

Pour Rappel:
Un Post est forcément lié à un utilisateur

DOCTRINE – POSTS

- Dans l'énoncé, seuls les utilisateurs identifiés et autorisés peuvent poster des articles donc cela implique que nous devons être connecté pour accéder à cette page.
- Il nous faut donc une condition qui vérifie dans notre controller si l'utilisateur est connecté ou pas.
- Si l'utilisateur n'est pas connecté on doit rajouter un message pour lui informer qu'il doit être connecté.

DOCTRINE – POSTS

```
{% if app.user %}  
    <h1>Create new Article</h1>  
  
    {{ include('article/_form.html.twig') }}  
  
    <a href="{{ path('app_article_index') }}">back to list</a>  
{% endif %}
```

DOCTRINE – POSTS

- Et maintenant nous n'avons plus qu'à utiliser notre setter depuis notre contrôleur pour ajouter l'utilisateur courant à notre Post comme ceci:

```
if ($form->isSubmitted() && $form->isValid()) {  
    $em = $this->getDoctrine()->getManager();  
    $article->setAuteur($this->getUser());  
    $em->persist($post);  
    $em->flush();
```

- Pour notre blog, nous allons avoir besoin d'un système de gestion des utilisateurs.
- Les utilisateurs non identifiés pourront lire les articles (Post)
- Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
- Les utilisateurs identifiés pourront commenter des articles (Post)
 - Pendant la phase de développement, nous aurons la possibilité de charger du contenu pré-défini à chaque fois qu'on aura besoin étant donné qu'on videra très souvent notre base de données

TP – GESTION DES COMMENTAIRES

Etant satisfait de votre travail, le client souhaite maintenant aller loin en mettant en place un système de gestion de commentaire.

Dans la partie de détail(show) de chaque article, il veut :

- Afficher le listing de tous les commentaires des différents utilisateurs avec le nom, la date
- Afficher le formulaire permettant de saisir un commentaire si et seulement si l'utilisateur est connecté.

NB : Les commentaires sont liés à un article et l'utilisateur connecté. ***Le formulaire n'est pas accessible que pour les utilisateurs connectés tandis que la liste des commentaires est visible pour tout le monde.***

Les attributs du commentaire :

- Contenu
- Created_at
- User
- Article

Résultat attendu :

Posté par **Moussa Camara** le 02/07/2023
Symfony est trop cool !

Posté par **Cécile Dupont** le 04/07/2023
J'aime bien le sujet traité !

Posté par **Arsène Lupin** le 05/07/2021
On vous propose des formations intensives

TP – GESTION DES COMMENTAIRES

- Les etapes de création:
 - Make entity
 - Make crud
 - Intégrer la création du commentaire dans le show (ArticleController)
 - Include du formulaire commentaire dans le twig
 - Mettre à jour
 - la date
 - User
 - Article

TP – GESTION DES COMMENTAIRES

```
#Route('/{id}', name: 'app_article_show', methods: ['GET', 'POST'])
public function show(Article $article, Request $request, EntityManagerInterface $entityManager): Response
{
    $comment = new Comment();

    $form = $this->createForm(CommentType::class, $comment);
    $form->handleRequest($request);

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

        $comment->setArticle($article);

        $entityManager->persist($comment);
        $entityManager->flush();

        return $this->redirectToRoute('app_article_show', ['id' => $article->getId()])
    }

    return $this->render('article/show.html.twig', [
        'article' => $article,
        //formulaire comment
        'form' => $form->createView()
    ]);
}
```

MAPPING ENTITY

- L'EntityRepository propose par défaut quelques méthodes pour vous éviter de les écrire.
 - **find** prend un unique paramètre et recherche l'argument dans la clé primaire de l'entité.
 - **findBy** prend 4 paramètres (`$criteria`, `$orderBy`, `$limit`, `$offset`). Cette méthode retourne des résultats correspondant aux valeurs des clés demandées.
 - **findAll** est un alias de `findBy([])`. Il retourne par conséquent tous les résultats.
 - **findOneBy** fonctionne comme la méthode `findBy` mais retourne un unique résultat et non pas un tableau
- <https://symfony.com/doc/current/doctrine.html#fetching-objects-from-the-database>

TP – GESTION DES COMMENTAIRES

- Listing

- // FindBy avec entity post
 - \$comments = \$entityManager->getRepository(Comment::class)->findByArticle(\$article);

- Twig:

```
<h3>Comments listing</h3>

{% for comment in comments %}

    <hr>

        <i>Posted by: <b>{{ comment.auteur.email }}</b> on the <em>{{ comment.createdAt|date('d/m/y h:i') }}</em></i>
        à {{ comment.createdAt|date('h:i') }}</em></i>
        <p>{{ comment.comment }}</p>

    {% endfor %}

    <hr>
```

- Pour notre blog, nous allons avoir besoin d'un système de gestion des utilisateurs.
 - Les utilisateurs non identifiés pourront lire les articles (Post)
 - Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
 - Les utilisateurs identifiés pourront commenter des articles (Post)
-
- Pendant la phase de développement, nous aurons la possibilité de charger du contenu pré-défini à chaque fois qu'on aura besoin étant donné qu'on videra très souvent notre base de données

- En phase de développement on vide souvent notre base de données car elle est constamment en train de changer
- Certaines données de notre base est utilisé par notre application et à chaque fois qu'on la vide, on doit remettre ces données.
- Exemple, quand on teste une partie de notre application qui nécessite une authentification, à chaque fois que vide notre base de données nous devons remettre des utilisateurs afin qu'on puisse continuer notre développement.
- Doctrine nous propose pour cela un Bundle qui nous permet de remplir des fichiers qui seront exécuter pour mettre des données en base.

- Fixtures (jeu de données) est un ensemble de données qui permet d'avoir un environnement de développement proche d'un environnement de production avec des fausses données.
- Pour utiliser les fixtures, il faut l'installer dans notre projet Symfony via composer :
 - composer require --dev orm-fixtures

<https://symfony.com/bundles/DoctrineFixturesBundle/current/index.html>

- Enregistrement dans le fichier bundles.php
 - `Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle::class => ['dev' => true, 'test' => true],`
- Une fois notre bundle installé, nous pouvons l'utiliser directement.
- Nous allons en faire une première utilisation, nous allons charger des utilisateurs dans notre base de données.

- Les fixtures se situent dans le dossier src, le dossier s'appelle *DataFixtures*.
- Il y a 2 conventions à respecter:
 1. Il faut que le nom fichier se termine par fixtures
 2. Il faut que le fichier se trouve dans le dossier DataFixtures de votre bundle
- Pour utiliser les fixtures il nous faut étendre de la classe AppFixtures et on aura besoin de l'Object Manager
 - use Doctrine\Bundle\FixturesBundle\Fixture;
 - use Doctrine\Common\Persistence\ObjectManager;
- Nous devons écrire nos fixtures dans une méthode appelé load()

- Une fois notre objet user fait il nous faut exécuter la méthode persist() puis la méthode flush()
 - Persist:
 - Dis à doctrine de gérer cet objet.
 - A l'exécution de cette méthode aucune requête n'est faite.
 - Flush
 - Met à jour la base de données en exécutant une requête

Pour tester

php bin/console doctrine:fixtures:load

EXAMPLE

```
class AppFixtures extends Fixture
{
    private $encoder;

    public function __construct(UserPasswordHasherInterface $encoder)
    {
        $this->encoder = $encoder;
    }
}
```

```
public function load(ObjectManager $manager): void
{
    $user = new User();

    $user->setEmail('admin@test.fr');
    $user->setRoles(['ROLE_ADMIN']);
    $user->setNom('Camara');
    $user->setPrenom('Moussa');

    $password = $this->encoder->hashPassword($user, 'pass_1234');

    $user->setTel('0612156352');
    $user->setPassword($password);

    $manager->persist($user);

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

AUTRE EXEMPLE

```
public function load(ObjectManager $manager): void
{
    for ($i=0; $i < 5 ; $i++) {
        $user = new User();
        $user->setEmail("macin$i@test.fr");
        $user->setRoles(['ROLE_USER']);
        $user->setNom("Bidule$i");
        $user->setPrenom("Machin$i");
        $user->setTel("0611563$i");
        $password = $this->encoder->hashPassword($user, "pass_134");
        $user->setPassword($password);
        $manager->persist($user);
    }
    $manager->flush();
}
```

DATAFIXURES

- Deux problèmes à résoudre:
 1. Comment fait-on si on a besoin d'utiliser une donnée créée avec des fixtures dans une autre fixture ?
 - Exemple: Nous avons des users et des posts. Dans le fixture de post, nous avons besoin de spécifié le user. Il nous faut donc pouvoir faire référence à un user.
 2. Comment dit-on à doctrine que nous voulons charger une fixture avant une autre
 - Exemple: Toujours avec nos users et nos posts. Nous devons charger nos users avant de pouvoir charger nos posts car nos posts ont besoin des users.

DATAFIXURES

- Doctrine nous propose des solutions à ces problèmes



DATAFIXTURES

1. Référencer une donnée:

- Nous pouvons référencer une donnée en faisant
 - `public const USER_REFERENCE = 'user-user';`
 - `$this->addReference(self::USER_REFERENCE, $user);`
- Quand nous aurons besoin de cette donnée dans une autre fixture, nous aurons juste à faire
 - `$one->setAuteur($this->getReference(UserFixtures::USER_REFERENCE));`

```
public function getDependencies()
{
    return array(
        UserFixtures::class,
    );
}
```

2. Ordre de chargement

- Pour pouvoir indiquer à doctrine qu'il doit charger nos fixtures dans un ordre précis, on aura besoin de (ne pas oublier le use)
 - `Doctrine\Common\DataFixtures\OrderedFixtureInterface;`

Il nous faut ensuite l'implémenter dans la déclaration de notre classe qui ressemble désormais à ça:

- `class ArticleFixtures extends Fixture implements DependentFixtureInterface`

DATAFIXURES

- Maintenant nous avons fini avec notre fixture il ne nous reste plus qu'a dire à doctrine de nous le charger en base de données.
- Une ligne de commande nous suffit pour ça:
 - `php bin/console doctrine:fixtures:load`
- A noter que doctrine va purger notre base de données avant de charger les fixtures
- Nos users sont maintenant charger dans notre base de données.

FAKER : LE GÉNÉRATEUR DE FIXTURES

- Lorsque l'on développe un site avec Symfony, il est souvent pratique d'utiliser les fixtures pour remplir sa BDD de valeurs bidons. Le problème c'est qu'il faut écrire chacune de ces fausses données à la mano, une par une. C'est là qu'utiliser le petit framework Faker peut nous être utile, en nous permettant de créer beaucoup de fausses données lisibles par un humain.
- **Faker** est un bundle php créé par l'excellent François Zaninotto permettant de générer facilement tout type de données. Il est utilisable comme une bibliothèque indépendante mais nous allons voir ici comment l'intégrer dans un projet Symfony3 pour remplir la BDD de son site.

FAKER : LE GÉNÉRATEUR DE FIXTURES

- Installation
- *composer require fzaninotto/faker*

La doc:

<https://github.com/fzaninotto/Faker>

FAKER : LE GÉNÉRATEUR DE FIXTURES

```
public function load(ObjectManager $manager): void
{
    // initialisation de l'objet Faker
    $faker = Faker\Factory::create('fr_FR');
    for($i =0; $i < 10; $i++){
        $user = new User();
        $user->setNom($faker->firstName);
        $user->setPrenom($faker->lastName);
        $user->setEmail($faker->email);
        $user->setTel($faker->phoneNumber);
        $password = $this->encoder->hashPassword($user, "pass_134");
        $user->setPassword($password);

        $manager->persist($user);
    }

    $manager->flush();

    $this->addReference(self::USER_REFERENCE, $user);
}
```

LIVRAISON

- Après livraison, le client nous a fait quelques remarques sur la sécurité de notre application.
- Il a remarqué que tous les utilisateurs peuvent modifier ou supprimer des commentaires alors que ces actions ne sont destinées que pour l'administrateur
- Corrigeons vite ...

POST – SÉCURISER LA GESTION DES POSTS

- On avait dit que sur Symfony l'authentification et l'autorisation étaient 2 choses bien distinct et complètement différent.
- **Autorisation**
 - L'autorisation est fait par l'access control.
 - Il vérifie si un utilisateur à le droit d'accéder à une URL.
 - Les droits avec symfony sont gérés avec un système de ROLE par hiérarchie
 - Dans votre security.yml vous avez une option dans laquelle vous définissez les roles avec leur hierarchie.
 - Dans la section access control, vous allez définir les URLs pour lesquelles vous devez avoir des ROLES (droits) spécifique pour y accéder

POST – SÉCURISER LA GESTION DES POSTS

- Modifions la ou il faut la hierarchie des roles:

Security.yaml

role_hierarchy:

 role_hierarchy:

 ROLE_SUPER_ADMIN: [ROLE_SUPER_ADMIN]

 ROLE_ADMIN: [ROLE_ADMIN]

 ROLE_USER: [ROLE_USER]

 ROLE_COMMENTATEUR: [ROLE_COMMENTATEUR]

 ROLE_CONTRIBUTOR: [ROLE_COMMENTATEUR, ROLE_CONTRIBUTOR]

- Sécurisons les URLs d'accès:

access_control:

- { path: ^/admin, roles: ROLE_ADMIN }
- { path: ^/comment/new, role: ROLE_USER }
- { path: ^/comment/[0-9]+/edit, role: ROLE_ADMIN }
- { path: ^/comment/[0-9]+/delete, role: ROLE_ADMIN }

https://symfony.com/doc/current/security/access_control.html



- Maintenant seuls les utilisateurs avec le role admin peuvent éditer et supprimer les posts.
- Quand un autre user essaie d'accéder à ces pages, il aura un access denied

POST – ACCESS DENIED

- D'ailleurs on en profite pour styliser notre la page d'erreur.
- On va créer une nouvelle page d'erreur

- Php bin/console make:controller accesDenied
- Dans security.yaml: `access_denied_url: /acces/denied`
- Reste maintenant à customiser notre template

LES VOTERS

- Une fois que nous avons terminé avec la partie admin (Sécurité), passons maintenant à la partie de l'auteur de l'article.
- On a dit que seul, l'auteur peut modifier ou supprimer son article.
- Pour cela nous allons utiliser Voters:

LES VOTERS

- Un Voter est un service qui sera appelé par la couche de sécurité de Symfony au moment où l'on vérifiera des droits d'accès.
- <https://symfony.com/doc/current/security/voters.html>

LES VOTERS

- Pour le mettre en place en ligne de commande:

- `Php bin/console make:voter`
- Ou
- `Php bin/console m:vo`

```
$ php bin/console m:vo

The name of the security voter class (e.g. BlogPostVoter):
> PostVoter

created: src/Security/Voter/PostVoter.php

Success!

Next: Open your voter and add your logic.
Find the documentation at https://symfony.com/doc/current/security/voters.html
```

LES VOTERS

```
class ArticleVoter extends Voter
{
    public const EDIT = 'POST_EDIT';
    public const DELETE = 'POST_DELETE';

    protected function supports(string $attribute, mixed $subject): bool
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, [self::EDIT, self::VIEW])
            && $subject instanceof \App\Entity\Article;
    }
}
```

```
protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
{
    $user = $token->getUser();
    // if the user is anonymous, do not grant access
    if (!$user instanceof UserInterface) {
        return false;
    }

$article = $subject;

    // ... (check conditions and return true to grant permission) ...
    switch ($attribute) {
        case self::EDIT:
            // logic to determine if the user can EDIT
            // return true or false
            return $article->getAuteur() == $user;

            break;
    }
    return false;
}
```

DANS LA VUE - TWIG

```
<a href="{{ path('app_article_show', {'id': article.id}) }}>show</a>
{% if is_granted('POST_EDIT', article) %}
    <a href="{{ path('app_article_edit', {'id': article.id}) }}>edit</a>
{% endif %}
```

LES VOTERS

- Maintenant le button ne sera visible que pour les utilisateurs qui ont le post mais il peut avoir des petits malins qui sans être l'auteur essaie de le modifier en écrivant directement l'URL
- Sauf que symfony gère parfaitement cette partie de sécuriter

```
#[Route('/{id}/edit', name: 'app_article_edit', methods: ['GET', 'POST'])]
public function edit(Request $request, Article $article,
EntityManagerInterface $entityManager): Response
{
    // check for "edit" access: calls all voters
    $this->denyAccessUnlessGranted('POST_EDIT',$article);
```

LES VOTERS

- Nous venons d'autoriser que l'auteur de l'article à modifier ou supprimer son article même l'admin ne peut rien faire.
- Symfony pense vraiment à tout ;) Dans notre Voter.php on peut ainsi donner tous les droits à l'admin (En même temps c'est le Big Boss)

<https://symfony.com/doc/current/security/voters.html#checking-for-roles-inside-a-voter>

LES VOTERS

```
private $security;  
public function  
__construct(Security $security)  
{  
    $this->security = $security;  
}
```

```
protected function voteOnAttribute(string  
$attribute, mixed $subject, TokenInterface $token):  
bool  
{  
    // ROLE_SUPER_ADMIN can do anything! The  
power!  
    if($this->security->isGranted('ROLE_ADMIN')) {  
        return true;  
    }  
}
```

LES ERREURS

- Maintenant qu'on a une application fonctionnelle, le métier souhaite avoir de beau message d'erreur, éviter le rouge trop agressif
- Pour voir les pages d'erreur en production
 - `# config/routes/framework.yaml`
 - `_errors:`
 - `resource: '@FrameworkBundle/Resources/config/routing/errors.xml'`
 - `prefix: /_error`
- `http://localhost/index.php/_error/{statusCode}`
 - Exemple: `http://localhost:8000/_error/404`

LES ERREURS

- 401 : utilisateur non authentifié ;
- 403 : accès refusé ;
- 404 : page non trouvée ;
- 500 et 503 : erreur serveur ;
- 504 : le serveur n'a pas répondu. Le temps d'attente pour accéder à la passerelle est expiré.

LES ERREURS

Symfony nous permet ainsi de bien nos messages d'erreur

https://symfony.com/doc/current/controller/error_pages.html

LES ERREURS

- On va surcharger les templates d'erreurs qui sont dans vendor
 - On va créer notre dossier `templates/bundles/TwigBundle/Exception/`
 - On peut maintenant créer nos fichiers d'erreurs selon le statut
 - `templates/bundles/TwigBundle/Exception/error404.html.twig`
 - `templates/bundles/TwigBundle/Exception/error403.html.twig`
 - `templates/bundles/TwigBundle/Exception/error.html.twig # All other HTML errors (including 500)`

NB: n'oubliez pas de modifier le `.env` → `APP_ENV=prod`

SWIFTMAILER

- On va maintenant mettre en place une gestion d'envoie de mail
- On va installer et configurer le bundle

`composer require symfony/swiftmailer-bundle`

<https://symfony.com/doc/current/email.html>

SWITFMAILER

- On peut maintenant tester l'envoie de mail apres l'ajout du post

```
/* SWITFMAILER - gmail*/
$message = (new \Swift_Message('Hello Email'))
    ->setFrom('testmouski@gmail.com')
    ->setTo('testmouski@gmail.com')
    ->setBody('You should see me from the profiler!')

;

$mailer->send($message);
/* FIN Swiftmailer */
```

NE PAS OUBLIER : \Swift_Mailer \$mailer

TP

- Après une nouvelle livraison, le client nous informe qu'il souhaite être notifié par mail après chaque nouvelle inscription. Aussi faire une page de bienvenue au nouveau membre.

SWITFMAILER

- 1- Crédation du Template de confirmation pour l'utilisateur
- 2- Crédation du Template de confirmation pour l'admin
- 3- Configurer l'envoie du mail dans registration.php



PROBLÈME BLOQUANT – REPOSITORY

- Le métier vient de nous informer que nous avons un gros problème de performance sur notre application.
 - Lorsqu'on affiche le listing des posts (`blog_post_homepage`), nous récupérons trop de post.
- Après discussion on a convenu du suivant avec eux:
 - On n'affiche pas l'id et le contenu
 - On affiche uniquement les 3 derniers posts

LES REPOSITORY – POUR FAIRE DU SQL

- Dans postRepository.php

```
public function getLastInserted($entity, $amount)
{
    return $this->getEntityManager()
        ->createQuery(
            "SELECT e FROM $entity e ORDER BY e.id DESC"
        )
        ->setMaxResults($amount)
        ->getResult();
}
```

DANS POSTCONTROLLER

```
#[Route('/', name: 'app_article_index', methods: ['GET'])]

public function index(ArticleRepository $articleRepository): Response
{
    return $this->render('article/index.html.twig', [
        'articles' => $articleRepository->getLastInserted('App:Article', 3)
    ]);
}
```

- Nous pouvons ainsi mettre une pagination de la liste pour une meilleure ergonomie.
- *composer require knplabs/knp-paginator-bundle*

```
# config/packages/paginator.yaml
knpPaginator:
    page_range: 5 # number of links showed in the pagination menu (e.g: you have 10 pages, a page_range
of 3, on the 5th page you'll see links to page 4, 5, 6)
    default_options:
        page_name: page # page query parameter name
        sort_field_name: sort # sort field query parameter name
        sort_direction_name: direction # sort direction query parameter name
        distinct: true # ensure distinct results, useful when ORM queries are using
GROUP BY statements
        filter_field_name: filterField # filter field query parameter name
        filter_value_name: filterValue # filter value query paameter name
    template:
        pagination: '@KnpPaginator/Pagination/sliding.html.twig' # sliding pagination controls
template
        sortable: '@KnpPaginator/Pagination/sortable_link.html.twig' # sort link template
        filtration: '@KnpPaginator/Pagination/filtration.html.twig' # filters template
```

REPOSITORY

- Nous allons faire une requête sql qui va afficher le résultat par ordre décroissant dans repository.

```
public function filter()
{
    return
    $this
        ->createQueryBuilder('a')
        ->orderBy('a.id', 'DESC')
    ;
}
```

CONTROLLER

```
use Knp\Component\Pager\PaginatorInterface;

#[Route('/', name: 'app_article_index', methods: ['GET'])]
public function index(ArticleRepository $articleRepository,
Request $request, PaginatorInterface $paginator): Response
{
    $articles = $paginator->paginate(
        $articleRepository->filter(),
        $request->query->getInt('page', 1),
        5
    );
    return $this->render('article/index.html.twig', [
        'articles' => $articles,
    ]);
}
```

TWIG

- Dans `index.html.twig` en dehors de la boucle.
- `<div class="navigation">`
- `{{ knp_pagination_render(articles) }}`
- `</div>`

CUSTOM KNP

- On peut aussi changer le style de la pagination. Pour cela, on s'inspire de Knplabs depuis le vendor. Une fois le template choisi on peut surcharger en créant un dossier au niveau des templates (copie – coller) et changer le fichier de conf
- Par exemple avec bootstrap
- Templates/paginator/paginator.html.twig

```
{% extends '@KnpPaginator/Pagination/twitter_bootstrap_v4_pagination.html.twig' %}
```

```
#Confif/package/paginator.yaml

template:
    pagination: 'paginator/paginator.html.twig'
    sortable: '@KnpPaginator/Pagination/sortable_link.html.twig' # sort link template
    filtration: '@KnpPaginator/Pagination/filtration.html.twig' # filters template
```

CUSTOM KNP

- Autre version: on peut changer directement au niveau de config/packages/paginator

```
pagination: '@KnpPaginator/Pagination/bootstrap_v5_pagination.html.twig'  
sortable: '@KnpPaginator/Pagination/sortable_link.html.twig' # sort link  
template  
filtration: '@KnpPaginator/Pagination/filtration.html.twig' # filters template
```

EASY ADMIN

- EasyAdmin est un bundle permettant de mettre en place un back-office d'administration
- EasyAdmin permet réaliser des opérations **SCRUD** (Search / Create / Read / Update / Delete) facilement sur des entités Doctrine (ORM)
- Disons que c'est la partie de CMS de symfony 😊

EASY ADMIN - INSTALLATION

- composer require easycorp/easyadmin-bundle
- Une fois installation terminée, on a
- Dans bundles.php: `EasyCorp\Bundle\EasyAdminBundle\EasyAdminBundle::class => ['all' => true],`

<https://symfony.com/doc/current/bundles/EasyAdminBundle/crud.html>

EASYADMIN

- Configurons notre dashboard
 - Changer le controller
 - Décommenter l'option 3
 - `return $this->render('admin/dashboard.html.twig');`
 - Créer le template
 - `Admin/dashboard.html.twig`

EASYADMIN

- Suivons toujours la doc en executant la commande `make:admin:dashboard`
- On doit aussi generer le crud de nos différentes entités
 - `Php bin/console make:admin:crud`

<https://symfony.com/bundles/EasyAdminBundle/current/fields.html#field-types>

EASYADMIN - USER

- Si nous gardons la manière classique, nous remarquerons que le mot de passe est en clair 😞

```
public function configureFields(string $pageName): iterable
{
    return [
        TextField::new('prenom'),
        TextField::new('nom'),
        TextField::new('tel'),
        ArrayField::new('roles'),
        TextField::new('email'),
        TextField::new('password')
            ->setLabel("New Password")
            ->setFormType(PasswordType::class),
    ];
}
```

EASYADMIN - USER

- Nous allons le corriger très rapidement et pour cela nous allons utiliser `UserPasswordHasherInterface` comme dans la fixture de User puis surcharger la méthode `persistEntity` du parent pour mettre un mot de passe par défaut

```
private $encoder;

public function __construct(UserPasswordHasherInterface $encoder)
{
    $this->encoder = $encoder;
}
```

```
public function persistEntity(EntityManagerInterface $entityManager, $entityInstance): void
{
    if (!$entityInstance instanceof User) return;
    $password = $this->encoder->hashPassword(new User, "pass_134");
    $entityInstance->setPassword($password);

    parent::persistEntity($entityManager, $entityInstance);
}
```

EASYADMIN - ARTICLE

```
public const BASE_PATH = 'images/article';
    public const UPLOAD_DIR = 'public/images/article';
```

```
public function configureFields(string $pageName): iterable
{
    return [
        TextField::new('titre'),
        TextField::new('resume'),
        TextEditorField::new('contenu'),
        AssociationField::new('Auteur'),
        DateTimeField::new('createdAt')->setFormat('yyyy.MMMM.dd G hh:mm aaa'),
        ImageField::new('image')
            ->setBasePath(self::BASE_PATH)
            ->setUploadDir(self::UPLOAD_DIR)
    ];
}
```

EASYADMIN - EXEMPLE COMMENTAIRE

```
public function configureFields(string $pageName): iterable
{
    return [
        TextField::new('comment'),
        AssociationField::new('auteur'),
        AssociationField::new('article'),
    ];
}

public function persistEntity(EntityManagerInterface $entityManager, $entityInstance):
void
{
    if (!$entityInstance instanceof Comment) return;
    $entityInstance->setCreatedAt(new \DateTimeImmutable());

    parent::persistEntity($entityManager, $entityInstance);
}
```

FIN

- Notre application est maintenant faite tel que demandé par le métier.



QUESTIONS ?



RÉPONSE

- Il n'y a pas de secret
 - Documentaire

Coder et coder

```
<?php  
  
namespace AppBundle\Controller;  
  
use AppBundle\Entity\Author;  
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Request;  
  
class HomeController extends Controller  
{  
    /**  
     * @Route("/", name="homepage")  
     */  
    public function homeAction()  
    {  
        $welcome = "Symfony";  
  
        $author = new Author();  
        $author->setName("Fabien");  
  
        return $this->render('home/homepage.html.twig', array(  
            'base_dir' => realpath($this->container->getParameter('kernel.root_dir') . '/../'),  
            'welcome' => $welcome, 'author' => $author  
        ));  
    }  
}
```



SINON ...

