Symfony

flex.symfony.com :

Là où se situent les bundles de symfony

Liste des répertoires

Ouvrez donc le répertoire dans lequel vous avez extrait les fichiers. Vous pouvez voir qu'il n'y a pas beaucoup de fichiers ici, seulement des répertoires. En effet, tout est bien rangé dans chaque répertoire, il nous faut donc comprendre à quoi ils servent. En voici la liste :

- **bin**
- **config**
- **public**
- **src**
- **var**
- **vendor**

Le répertoire /bin

Ce répertoire contient tous les exécutable dont nous allons nous servir pendant le développement. Par exécutable, j'entends des commandes PHP.

Le répertoire /config

Ce répertoire contient toute la configuration de votre site. C'est ici que nous configurerons Symfony lui-même, mais aussi les plugins (ou bundles) que nous installerons par la suite. La configuration des différentes parties est éclatée dans différents répertoires et fichiers à l'intérieur de /config .

Le répertoire /public

Ce répertoire contient tous les fichiers destinés à vos visiteurs : images, fichiers CSS et JavaScript, etc. Il contient également le contrôleur frontal (index.php).
En fait, c'est le seul répertoire qui devrait être accessible à vos visiteurs, d'où son nom : « public ». Les autres répertoires ne sont pas censés être accessibles (ce sont vos fichiers de code source, ils vous regardent vous, pas vos visiteurs).

Le répertoire/src

Voici enfin le répertoire dans lequel on mettra le code source ! C'est ici que l'on passera le plus clair de notre temps. Vous pouvez voir que ce répertoire n'est pas vide : il contient en effet le noyau (kernel).

Le répertoire /template

Ce répertoire n'existe pas encore. Mais il contiendra tous les templates de notre application. Si vous connaissez déjà le modèle MVC, cela correspond aux vues. C'est principalement ici que sera le HTML que nous écrirons.

Le répertoire /var

Ce répertoire contient tout ce que Symfony va écrire durant son exécution : les logs, le cache, et d'autres fichiers nécessaires à son bon fonctionnement. Nous n'écrirons jamais dedans nous même.

Le répertoire/vendor

Ce répertoire contient toutes les bibliothèques externes à notre application. Dans ces bibliothèques externes, j'inclus Symfony ! Vous pouvez parcourir ce répertoire ; pour l'instant il n'y a quasiment que Symfony.

Le fichier .env :

C’est la où se situe APP_ENV = dev si c’est en développement sinon prof

Le contrôleur frontal

Le contrôleur frontal (front controller, en anglais) est le point d'entrée de votre application. C'est le fichier par lequel passent toutes vos pages. Vous devez surement connaître le principe d'index.php et des pseudo-frames (avec des URL du type index.php?page=blog) ; eh bien, cet index.php est un contrôleur frontal. Dans Symfony, le contrôleur frontal se situe dans le répertoire/public, il s'appelle également index.php .

Architecture MVC :

MVC signifie « Modèle / Vue / Contrôleur ». C'est un découpage très répandu pour développer les sites Internet, car il sépare les couches selon leur logique propre :

- Le Contrôleur (ou Controller) : son rôle est de générer la réponse à la requête HTTP demandée par notre visiteur. Il est la couche qui se charge d'analyser et de traiter la requête de l'utilisateur. Le contrôleur contient la logique de notre site Internet et va se contenter « d'utiliser » les autres composants : les modèles et les vues. Concrètement, un contrôleur va récupérer, par exemple, les informations sur l'utilisateur courant, vérifier qu'il a le droit de modifier tel article, récupérer cet article et demander la page du formulaire d'édition de l'article. C'est tout bête, avec quelques if(), on s'en sort très bien.

- Le Modèle (ou Model) : son rôle est de gérer vos données et votre contenu. Reprenons l'exemple de l'article. Lorsque je dis « le contrôleur récupère l'article », il va en fait faire appel au modèle Article et lui dire : « donne-moi l'article portant l'id 5 ». C'est le modèle qui sait comment récupérer cet article, généralement via une requête au serveur SQL, mais ce pourrait être depuis un fichier texte ou ce que vous voulez. Au final, il permet au contrôleur de manipuler les articles, mais sans savoir comment les articles sont stockés, gérés, etc. C'est une couche d'abstraction.

- La Vue (ou View) : son rôle est d'afficher les pages. Reprenons encore l'exemple de l'article. Ce n'est pas le contrôleur qui affiche le formulaire, il ne fait qu'appeler la bonne vue. Si nous avons une vueFormulaire, les balises HTML du formulaire d'édition de l'article y seront et au final le contrôleur ne fera qu'afficher cette vue sans savoir vraiment ce qu'il y a dedans. En pratique, c'est le designer d'un projet qui travaille sur les vues. Séparer vues et contrôleurs permet aux designers et développeurs PHP de travailler ensemble sans se marcher dessus.

Parcours d'une requête Symfony :

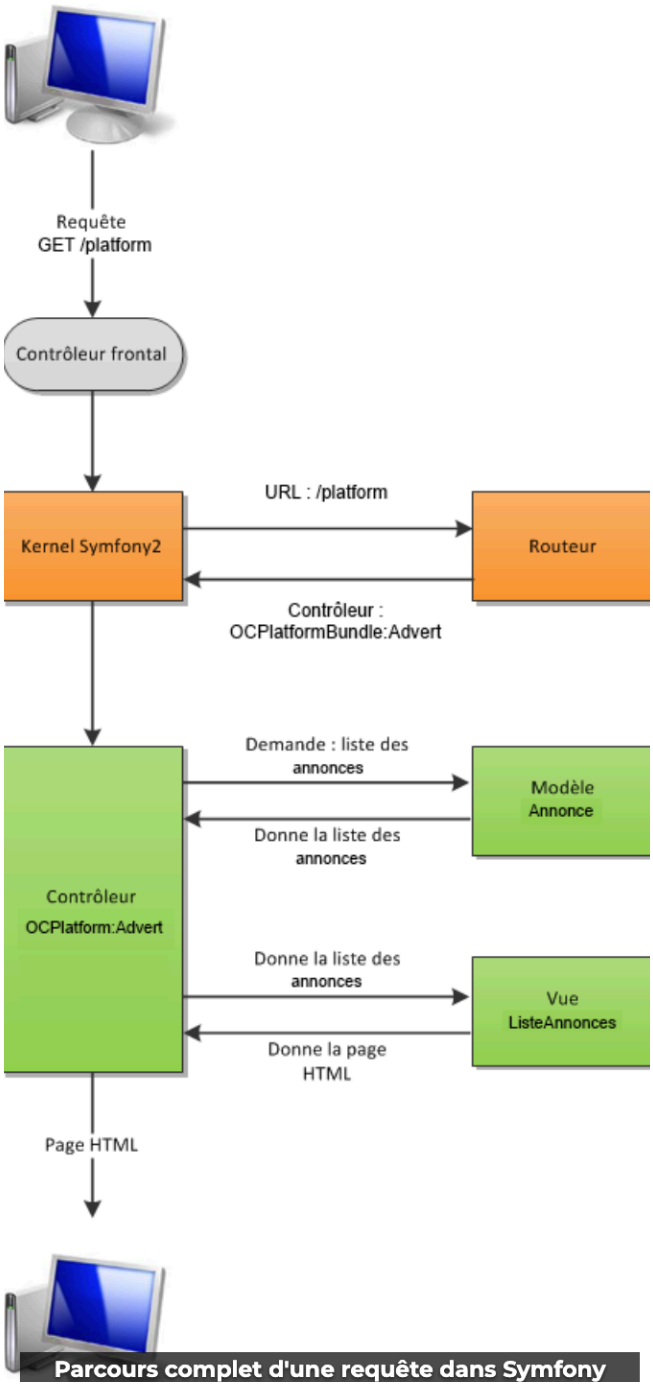
En le parcourant avec des mots, voici ce que cela donne :

1. Le visiteur demande la page **/platform ;**

2. Le contrôleur frontal reçoit la requête, charge le **Kernel** et la lui transmet ;

3. Le Kernel demande au Routeur quel contrôleur exécuter pour l'URL **/platform**. Ce Routeur est un composant Symfony qui fait la correspondance entre URL et contrôleurs. Le Routeur fait donc son travail, et dit au **Kernel** qu'il faut exécuter le **contrôleurOCPlatform:Advert ;**

4. Le **Kernel** exécute donc ce contrôleur. Le contrôleur demande au modèle **Annonce** la liste des annonces, puis la donne à la vue **ListeAnnonces** pour qu'elle construise la page HTML et la lui retourne. Une fois cela fini, le contrôleur envoie au visiteur la page HTML complète.



Parcours complet d'une requête dans Symfony

J'ai mis des couleurs pour distinguer les points où l'on intervient. En vert, les contrôleur, modèle et vue, c'est ce qu'on devra développer nous-mêmes. En orange, le Kernel et le Routeur, c'est ce qu'on devra configurer. On ne touchera pas au contrôleur frontal, en gris.

Doctrine gère l'accès aux données.
Twig gère l'affichage, le rendu.
Controller gère les traitements.

Créer un nouveau projet :

Composer create-project symfony/website-skeleton NomProjet 4.4.99
//Si on veut la version 4

C'est un modèle d'un projet de site web symfony
Ensuite un
git init

Installer une libraire :

Composer require [xxx]
Pour la suite on commence par installer la librairie **server —dev** (pour ne l'utiliser qu'en développement).

Lancer le serveur en local :

Composer require server —dev
On commence par installer la librairie **server —dev** (pour ne l'utiliser qu'en développement).
Et ensuite :
php bin/console server:run

_ _ _ _ _ _********** **********_ _ _ _ _ _ _ _ _-

Installer home brew :

Dans le fichier Users/Jéjé :
mkdir homebrew && curl -L https://github.com/Homebrew/brew/tarball/master | tar xz --strip 1 -C homebrew
(De la page https://docs.brew.sh/Installation)

- Ensuite installer Encore :
composer require symfony/webpack-encore-bundle

- Puis yarn :
Yarn install

- Install scss :
npm install node-sass@4.14.1

- Install Jquery + popper :
npm install --save jquery popper.js
npm add popper.js —also=dev

- Installer bootstrap :
Npm install bootstrap —also=dev

- Créer le fichier entreponts.json et lancer le serveur :
Npm Run build
Npm Run watch

- SELECT2 :
Après avoir fait tout le bordel ci dessus :
Npm add select2

- Dans le fichier app.js :
```
var $ = require('jquery')
require('select2')
$('select').select2();
```

- Dans le fichier webpack.config.js tout à la fin :
```
var config = Encore.getWebpackConfig();
  config.externals.jquery = "jQuery";
```

```
module.exports = config;
```

_ _ _ _ _ _**********. **********_ _ _ _ _ _ _ _ _

Installer WEBPACK ENCORE :

Pour la gestion du css (faut installer node.js pour ça) :
composer require symfony/webpack-encore-bundle
npm install @symfony/webpack-encore —save-dev (ou juste npm install)
Npm run dev-server

/*Créer un dossier css mettre app.css dedans idem pour js.
Modifier le chemin du css dans app.js et du js dans webpack.config.js
Pour l'utilisation de select2 entrer la commande npm i select2*/

Décommenter la ligne suivante dans webpack.config.js :

```
//Installer Sass auparavant
    // enables Sass/SCSS support
    .enableSassLoader()
```

Ensuite :

```
    .addEntry('app', './assets/js/app.js')
    /* app est le nom pour l'importer dans la base */
    .addStyleEntry('style', './assets/css/app.scss')
```

Importer les fichiers (par exemple CSS ou JS) dans le fichier app.scss :

```
@import "~bootstrap/scss/bootstrap";
```

Et pour finir :

npm run build //Permet d'actualiser tous les fichiers manuellement
npm run watch //Lance un serveur qui actualise automatiquement le css et tout ce qui change (JS comme CSS

Commandes encore :

"dev-server": "encore dev-server",
Cette ligne est présente dans package.json et signifie que la commande encore dev-server lance le serveur d'actualisation automatique.
"dev": "encore dev",
Permet de gérer les fichiers javascript en mode développement (ccd non m'initier et avec les ressources maps)
"watch": "encore dev --watch",
Idem que dev sauf que quand on modifie le fichier il le re-compile automatiquement.
"build": "encore production --progress"
Permet de générer des fichiers en production

Créer une entité représentant une table :

Php bin/console make:entity
Le nom de la table sera ensuite demandé. Exemple : Articles.

Voila, 2 dossiers sont créés :

- **src/Entity/Articles.php** qui représentera la table des articles
- **src/Repository/ArticlesRepository.php** nous permettra de faire des sélections sur les données de cette table.

Le terminal demande maintenant une nouvelle propriété, qui correspond dans la table de données à un champ, il faudra lui attribuer le type de données lui correspondant.
Une fois terminé il ne reste plus qu'à faire une migration pour envoyer les données sur la base.

Créer un controller complet relié à l'entity :

```
Php bin/console make:crud
```

Fabriquer un controller :

```
Php bin/console make:controller
```

Le terminal nous demande ensuite le nom pour la classe que nous allons créer.
Attention ! Il existe une convention de nommage qui est la première lettre en majuscule et le reste en came case (1 lettre d'un mot en majuscule), par exemple BlogController.
Ensuite Symfony créera un fichier **BlogController.php** dans le dossier **Controller** qui contiendra la classe. Un fichier sera créer aussi dans le dossier **Templates** un dossier **blog** avec dedans un fichier **index.html.twig**
Ci dessous la class dans le fichier php.

Indiquer la route et la méthode pour y accéder :

```
/**
 * @Route("/admin/property/{id}", name="admin.property.delete", methods="GET|POST|DELETE")
*/
    public function delete(Property $property, Request $request){
        $submittedToken = $request->request->get('token');
        if($this->isCsrfTokenValid('delete_property', $submittedToken)){
            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->remove($property);
            $entityManager->flush();
            return $this->redirectToRoute('admin.property.index');}}
```

Le commentaire dit au serveur, quand on vient sur la page /admin/property/{id} tu exécutes cette fonction.
Cette route n'est accessible que grâce aux méthodes GET, POST ou DELETE.
Pour faire une méthode accessible de façon personnalisée comme DELETE :

```
<form method="POST" action="{{ path('admin.property.delete', {id: property.id}) }} »
onsubmit="return confirm('Êtes-vous sûr ?')"
>
   <input type="hidden" name="_method" value=« DELETE">
   <input type="hidden" name="token" value="{{ csrf_token('delete_property') }}"/>
   <button class="btn btn-danger">Supprimer</button>
</form>
```

Afin de sécuriser ce formulaire et il est nécessaire de mettre un token. csrf('nomdubien' - 'sonID') et dans le controller vérifier si le token est bon :

```
 public function delete(Property $property, Request $request){
        $submittedToken = $request->request->get('token');
        if($this->isCsrfTokenValid('delete_property', $submittedToken)){}
```

Paramétrer la page d'accueil :

Dans le fichier qui vient d'être créer on rentre la fonction suivante :

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

Ensuite on créer le fichier **home.html.twig** dans le dossier **blog**.

PARAMCONVERTER :

PARAMCONVERTER convertit un paramètre en une entité.
LIRE LE COURS SUIVANT ET FAIRE UNE SYNTHÈSE :
https://zestedesavoir.com/tutoriels/620/developpez-votre-site-web-avec-le-framework-symfony2/397_astuces-et-points-particuliers/2008_utiliser-des-paramconverters-pour-convertir-les-parametres-de-requetes/#1-6426_theorie-pourquoi-un-paramconverter

Créer un lien :

Le **name** dans le code ci-dessus permet que l'on puisse aller sur ce lien avec le code suivant :

```
{{ path(''home'') }}
```

Défnir la route dans routes/annotations.yaml :

```
home: //<= ici le nom de la route (utiliser pour {{ path('home') }}
    path : /
    controller: App\Controller\HomeController::index
```

Utiliser un service :

```
Php bin/console //pour voir toutes les commandes possibles
Php bin/console debug:container //affiche les services actuels de l'application
```

Dans service.yaml :

```
App\Controller\HomeController:
        arguments :
            $twig = '@twig'
```

Ce qui veut dire charge @twig qui est une instance de Twig\Environment;

Php bin/console debug:autowiring Donne les listes des classes qui n'auront pas besoin d'être déclarées dans les services et qui seront automatiquement appelées. Exemple :

```
Use Twig\Environment;
public function __construct(Environment $twig){
        $this->twig = $twig;
    }
```

Insérer le titre des articles dans l'URL avec Slugify :

Chercher sur packagist Slugify (qui mène à ce lien)
https://github.com/cocur/slugify

Ou directement dans le terminal :

```
$ composer require cocur/slugify
```

Dans Property.php :

```
use Cocur\Slugify\Slugify;

public function getSlug(): string
    {
        return (new Slugify())->slugify($this->title);
    }
```

Dans Property.html.twig (ou autre) :

```
<a href="{{ path('property.show', {id: property.id, slug property.slug}) }}">{{ property.title }}</a>
```

Dans PropertyController.php

```
/**
     * @Route("/biens/{slug} - {id}", name="property.show")
     */
    public function show(): Response
    {
        return $this->render('property/show.html.twig', [
            'current_menu' => 'properties',
            'property' => $property
        ]);
    }
```

Obtenir toutes les configurations d'un composant :

Php bin/console config:dump-reference (Exemple :) security

Composant Security

Sécurité :

Config/packages/security.yaml

- **Firewalls** :

Permet de définir les composants qui vont permettre d'authentifier l'utilisateur.
C'est ici que l'on met ce qui est protégé et ce qui ne l'est pas, et comment c'est protégé (avec token ou non etc…)

- **Providers** :

Permet de dire comment fournir les données utilisateurs

- **Encoders:**

C'est ici que l'on détermine comment encoder/hasher les données sensibles.

User :

On commence par créer une entité User avec pour champ email, username, password

Php bin/console make:entity User

On lui implémente UserInterface :

```
use Symfony\Component\Security\Core\User\UserInterface;
```

implements UserInterface

Voici les méthodes à redéfinir :

- getRoles() :

Doit renvoyé un tableau d'une chaine de caractères qui détermine le rôle de l'utilisateur. Ex : return['ROLE_USER'];

- getPassword() //Déjà créé par nous
- getSalt()
- getUsername() //Déjà créé par nous
- eraseCredentials()

Ensuite on fait son formulaire (d'inscription)

```
Php bin/console make:form RegistrationType
```

Créer son controller de sécurité :

```
Php bin/console make:controller SecurityController
```

Y ajouter la fonction login() :

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

Dans le Config/packages/security.yaml écrire comment encoder :

```
security:
   encoders:
      App\Entity\User:
         algorithm: bcrypt
         cost: 12
```

Si le formulaire est valide et complété, il reste à encoder le mot de passe dans le controller :

```
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
(UserPasswordEncoderInterface $encoder,  ObjectManager $manager){
$hasch = $encoder->encodePassword($user, $user->getPassword());
$user->setPassword($hasch);
$manager->persist($user);
$manager->flush();
}
```

Rendre l'utilisateur unique (avec son mail) :

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

Faire un login :

Dans config/packages/security.yaml écrire où chercher les informations pour valider la connexion et sur quel champ baser le login.

```
providers:
        users_in_memory: { memory: null }//N'a pas été changé
        in_database:
            entity:
                class: App\Entity\User
                property: email
```
Puis au niveau du firewall :
```
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: in_database

            form_login:
                login_path: security_login
                check_path: security_login
```
Option : form_login //permet de préciser qu'on utilisera un formulaire login

Faire un formulaire de connexion sur la page login.html.twig :

```
<form action="{{ path('security_login') }}" method="POST">
        <input placeholder="Email" required type="email" name="_username"">
        <input placeholder="Mot de passe" required type="password" name=« _password">
        <button type="submit" class="btn btn-success">Connexion</button>
</form>
```

AuthentificationUtils :

Possède 2 méthodes :

- **getLastUsername** : Qui permet de récupérer le dernier nom tapé par l'utilisateur. (À insérer dans le value input de _username.
- **getLastAuthenticationError** : Permet de récupérer les erreurs.

SecurityController.php :

```
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
/**
    *
    * @Route("/login", name="login")
    */
   public function login(AuthenticationUtils $authenticationUtils){

       $error = $authenticationUtils->getLastAuthenticationError();
       $lastUsername = $authenticationUtils->getLastUsername();

       return $this->render('security/login.html.twig',[
           'lastUsername' => $lastUsername,
           'error' => $error
       ]);
   }
```

Une fois login.html.twig faire :
{{ error.messageKey | trans(error.messagedata, 'security') }}
Ce qui permet de traduire les messages d'erreurs

Créer un fonction vide logout() dans SecurityController :

```
/**
    * @Route("/deconnexion", name="security_logout")
    */
   public function logout(){}
```

Ajouter dans security.yaml à la suite de form_login :

```
logout:
      path: security_logout
      target: /
         // ou possibilité de faire comme ça aussi : target: recettes_index
```

Se déconnecter manuellement :

Une fois logger mettre **log:out@** devant l'adresse du site, exemple:
log:out@http://127.0.0.1:8001/admin/property

Créer un formulaire de commentaire :

Php bin/console make:form CommentType
Ensuite écrire Comment (nom de l'entité)

```
Public function show(Article $article, Request $request, EntityManagerInterface $manager){
        $comment = new Comment;
        $form = $this->createForm(CommentType::class, $comment);

        $form->handleRequest($request);
        if($form->isValid() && $form->isSubmitted()){
        $comment->setCreatedAt(new \DateTime())
                    ->setArticle($article);
        $manager->persist($comment);
        $manager->flush();
        redirectToRoute('blog_show', ['id' => $article->getId()])
}

Return $this->render('blog/show.html.twig', [
        'article' => $article,
        'commentForm'=>$form->createView()
])}
```

Voir tout ce qui concerne le composant :

Php bin/console config:dump-reference security

Plus d'infos dans les dossiers de Symfony :

Vendor/Symfony/Security/Firewall

Par exemple si l'on recherche les différents types de firewall on peut regarder directement dans le code afin de voir ses méthodes.

IMAGE

Installer le bundle :

https://github.com/dustin10/VichUploaderBundle/blob/master/docs/installation.md

```
composer require vich/uploader-bundle
```

Mettre le driver en ORM :

Dans config/packages/vich_uploader.yaml

```
vich_uploader:
    db_driver: orm
```

Faire un mapping et renommer les images par défaut :

Dans config/packages/vich_uploader.yaml

```
mappings:
        property_image:
            uri_prefix: /images/properties
            upload_destination: '%kernel.project_dir%/public/images/properties'
            namer : Vich\UploaderBundle\Naming\UniqidNamer
```

Le mapping s'appelle **property_image** et le dossier dans lequel on va envoyer l'image /images/properties (**uri_prefix**)
upload_destination: là où je souhaite envoyer les différents fichiers (téléchargé)
Pour renommer les nouvelles images avec **namer** : insérer le lien situé dans la doc de VichUploader à l'adresse :
https://github.com/dustin10/VichUploaderBundle/blob/master/docs/namers.md

Modifier l'entité concernée (par le téléchargement d'image) :

Dans le fichier nomEntité.php :
Déclaration des classes et définition de la classe comme étant Uploadable

```
use Symfony\Component\HttpFoundation\File\UploadedFile;
use Symfony\Component\HttpFoundation\File\File;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
/**
 * @ORM\Entity(repositoryClass=PropertyRepository::class)
 * @UniqueEntity("title")
 * @Vich\Uploadable()
 */
class Property{
// Déclarations des variables à ajouter + un nouveau champ dans l'entité :
/**
     * @var string|null
     * @ORM\Column(type="string", length=255)
    */
    private $fileName;
```

```
    /**
     * @var File|null
     * @Assert\Image(
     *  mimeTypes = « image/jpeg" //Permet de sélectionner le type de fichier demandé
     * )
     * @Vich\UploadableField(mapping="property_image", fileNameProperty="fileName")
    */
    private $imageFile;
    /**
     * @ORM\Column(type="datetime")
     */
    private $updated_at;
```

//LES GETTERS ET SETTERS :

```
 public function getFileName(){
        return $this->fileName;
    }
    public function setFileName($fileName){
        $this->fileName = $fileName;

        return $this;
    }

     public function getImageFile(){
        return $this->imageFile;
    }
    public function setImageFile($imageFile){
        $this->imageFile = $imageFile;
      if($this->imageFile instanceof UploadedFile){
            $this->updated_at = new \DateTime('now');
        }
        return $this;
    }
    public function getUpdatedAt(): ?\DateTimeInterface
    {
        return $this->updated_at;
    }

    public function setUpdatedAt(\DateTimeInterface $updated_at): self
    {
        $this->updated_at = $updated_at;

        return $this;
    }
```

Le updated_at permet de vérifier le dernier changement de photo et modifie en fonction de celui ci. Dans le setImageFile on vérifie si la photo à été changée, si oui, le updated_at est actualisé. Ensuite migrer vers la bdd les changements.

Modifier le formulaire de l'entité concernée :

Dans le fichier nomEntitéType.php :

```
use Symfony\Component\Form\Extension\Core\Type\FileType;
->add('imageFile', FileType::class, [
            'required' => false
        ])
```

Installer le bundle qui redimensionne les images automatiquement :

https://symfony.com/doc/current/bundles/LiipImagineBundle/installation.html#step-1-download-the-bundle

```
composer require liip/imagine-bundle
```

Modifier le fichier config/package/liip_imagine.yaml :

```
liip_imagine :
   filter_sets :
      cache : ~
      # the name of the "filter set"
      thumb :
         # adjust the image quality to 75%
         quality : 75
         # list of transformations to apply (the "filters")
         filters :
            # create a thumbnail: set size to 120x90 and use the "outbound" mode
            # to crop the image when the size ratio of the input differs
            thumbnail  : { size : [360, 230], mode : outbound }
```

Afficher l'image :

Dans le fichier nomEntité.html.twig

```
{% if property.fileName %}
        <img src="{{ vich_uploader_asset(property, 'imageFile') | imagine_filter('thumb') }}" alt="card-img-top" class="img-fluid">
    {% else %}
        <img src="{{ 'images/properties/empty.jpg' | imagine_filter('thumb') }}" alt="card-img-top" class="img-fluid">
    {% endif %}
```

Une méthode dans le bundle permet de ne pas à avoir à écrire le chemin des images en dur, inconvénient si on veut changer l'emplacement ensuite pour éviter ceci :
vich_uploader_asset() prend 2 paramètres, le **nom de l'entité** et **le nom de la propriété de notre entité.**
imagine_filter() est une méthode permettant le redimensionnement des images miniatures automatiquement, le nom du filtre à utiliser est **thumb.**

Faire vider le cache automatiquement :

Créer un fichier **ImageCacheSubscriber.php** dans le dossier src/**Listener** y écrire **:**

```
<?php
namespace App\Listener;
use App\Entity\Property;
use Doctrine\Common\EventSubscriber;
```

```
use Doctrine\ORM\Event\LifecycleEventArgs;
use Doctrine\ORM\Event\PreUpdateEventArgs;
use Liip\ImagineBundle\Imagine\Cache\CacheManager;
use Symfony\Component\HttpFoundation\File\UploadedFile;
use Vich\UploaderBundle\Templating\Helper\UploaderHelper;
class ImageCacheSubscriber implements EventSubscriber{
    /**
     *@var CacheManager;
     */
    private $cacheManager;
    /**
     *@var UploaderHelper;
     */
    private $uploaderHelper;
    public function __construct(CacheManager $cacheManager, UploaderHelper $uploaderHelper)
    {
        $this->cacheManager   = $cacheManager;
        $this->uploaderHelper = $uploaderHelper;
    }
    public function getSubscribedEvents(){
        return[
            'preRemove',
            'preUpdate'
        ];
    }
    public function preRemove(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();

        if(!$entity instanceof Property){
            return;
        }
        $this->cacheManager->remove($this->uploaderHelper->asset($entity, 'imageFile'));
    }
    public function preUpdate(PreUpdateEventArgs $args)
    {
        $entity = $args->getEntity();
        if(!$entity instanceof Property){
            return;
        }
        if($entity->getImageFile() instanceof UploadedFile){
            $this->cacheManager->remove($this->uploaderHelper->asset($entity, 'imageFile'));
        }
    }
}
```

Modifier le fichier config/packages/service.yaml :

Ajouter ceci à la fin pour permettre à ImageSubscriber d'être reconnu par symfony et accéder aux informations nécessaires :

```
App\Listener\ImageCacheSubscriber:
```

```
        tags:
            - { name: doctrine.event_subscriber }
```

Ne pas oublier de modifier le fichier .gitignore sur les images et media(cache) :

TWIG

Elements de syntaxe Twig :

Twig supporte nativement trois types de syntaxe :

- {{ ... }} permet l'affichage d'une expression ;
- {% ... %} exécute une action ;
- {# ... #} n'est jamais exécuté, c'est utilisé pour des commentaires.

Implémenter des variables à Twig à partir du controller :

Dans le controller, dans la classe correspondante on peut créer une fonction comme ceci :

```
class HelloController extends AbstractController{
    /**
     * Hello world, avec Twig cette fois :)
     *
     * @Route("/hello/{name}", name="hello")
     */
    public function hello($name)
    {
        $this->render('hello.html.twig', ['name' => $name]);
    }
}
```

Et le gabarit Twig correspondant :

```
{# /templates/hello.html.twig #}
Hello {{ name }}!
```

Par défaut, le framework Symfony ira chercher les gabarits dans son dossier **templates**.
La fonction **render** prend en paramètre le chemin vers le gabarit et un **tableau de paramètres**.
Les **paramètres** sont disponibles dans le gabarit.

Déclarer plusieurs variables dans Twig :

```
    public function home(){
        return $this->render('blog/home.html.twig', [
            'title' => 'Yo !',
            'age' => '26'
        ]);
    }
```

Opérateur Twig :

```
{% for key, element in elements %}
    {% if loop.index % 2 %}
        Element pair
    {% else %}
```

```
        Element impair
    {% endif %}
{% else %}
    Il n'y a aucun élément à afficher.
{% endfor %}
```

Autre opérateurs :

- **in**, **is** ;
- les opérateurs **mathématiques** (+, -, /, %, //, *, **, %) ;
- les opérateurs **de logique** (and, or, not...) ;
- les opérateurs de **comparaison** (==, !=, <, >, >=...) ;
- des opérateurs plus utilitaires (~, ?:...).

Tags :

Les tags sont des éléments de langage propres à Twig.
Voici la liste des tags les plus utilisés :

- **block** : définit un espace surchargeable.
- **do, if, else,for, (with)** : éléments de langage identiques à PHP.
- **import** : permet d'importer un fichier comprenant des macros.
- **set** : permet de définir une ou plusieurs variables.
- **spaceless** : supprime tous les espaces entre les tags HTML.
- **verbatim** : ne sera pas pris en compte par Twig.

Fonction Twig :

Une **fonction** peut changer la valeur d'une variable et peut avoir un ou plusieurs paramètres :

```
{% for i in range(1, 10) %}
    {{ i }}
{% endfor %}

{# Twig supporte les paramètres nommés #}
{{ renderWithOptions(foo = 'foo', bar = 'bar', baz = 'baz') }}
```

Parmi la liste des fonctions disponibles nativement, voici les plus utilisées :

- **constant** : appelle une constante PHP.
- **include** : retourne le rendu d'un fichier.
- **dump** : appelle la fonction dump (disponible uniquement dans Symfony).
- **min, max** : équivalents des fonctions PHP min() et max().

Pour plus de fonctions voir :

https://twig.symfony.com/doc/2.x/functions/index.html

Filtre Twig :

Là où une fonction peut changer la valeur d'une variable, un filtre change seulement son affichage. On utilise l'opérateur "pipe" | pour appliquer un filtre. Les filtres peuvent être chaînés.

```
{{ 'foo'|capitalize|reverse }} {# "Oof" #}
```

Par défaut, Twig protège et échappe toute valeur, donc s'il vous faut afficher un contenu HTML ou Javascript, utilisez le filtre **e** ou **raw** (aucune protection appliquée).

```
{{ '<h1>Foo</h1>' }} {# affichera "<h1>Foo</h1>" dans le navigateur #}
{{ '<h1>Bar</h1>'|raw }} {# affichera "Bar" tel qu'attendu #}
```

Parmi la liste des filtres disponibles nativement, voici les plus utilisés :

- **date** : permet de formater une date.
- **first** : affiche le premier élément.
- **last** : affiche le dernier élément.
- **length** : affiche la longueur d'un tableau, d'une chaîne de caractères.
- **number_format** : permet de formater un nombre.

Pour plus de filtres voir :
https://twig.symfony.com/doc/2.x/filters/index.html

Quelques exemples :

```
{% set tableau, article = [1,2,3], {
   'titre': 'Etendre Twig',
   'contenu': 'Il est possible ...'
   }
%}

{{ tableau.2 ~ ' ' ~ tableau|first  }} {# "2 1" #}}

{{ article.titre }} {# "Etendre Twig" #}
{{ article.contenu }} {# "Il est possible ..." #}
```

Tout d'abord, un exemple d'utilisation de ~ : il est utilisé pour la concaténation de chaînes de caractères.

Twig est capable, à l'aide de l'opérateur ., d'aller retrouver la clé d'un tableau, la propriété publique d'un objet ou même... l'accesseur correspondant ! Par exemple si l'objet article avait eu une fonction publique getContenu(), elle aurait été appelée.

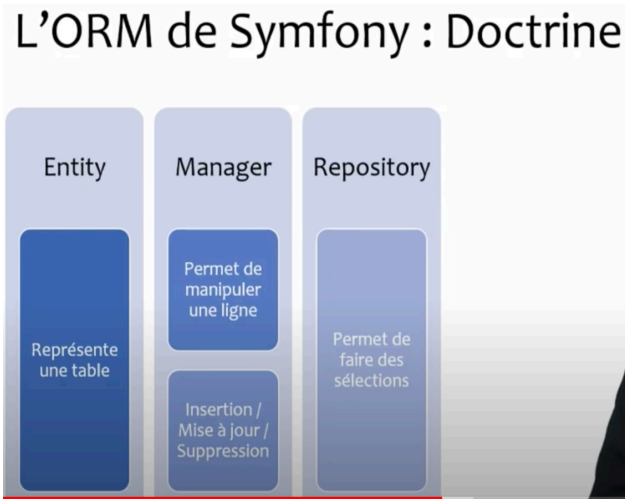
Créez vos propres fonctions et filtres

Pour ajouter vos propres filtres et fonctions, il faudra **créer une extension Twig**.
Une extension Twig est une classe qui permet de définir ses propres filtres et fonctions et qui implémente Twig_ExtensionInterface, mais on préférera étendre la classe abstraite AbstractExtension:

```
// src/Twig/AppExtension.php
namespace App\Twig;

use Some\LipsumGenerator;
use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;
use Twig\TwigFunction;

class AppExtension extends AbstractExtension
{
   public function getFilters()
   {
      return [
         new TwigFilter('price', [$this, 'filterPrice']),
      ];
   }

   public function getFunctions()
```

```
    {
        return [
            // appellera la fonction LipsumGenerator:generate()
            new TwigFunction('lipsum', [LipsumGenerator, 'generate']),
        ];
    }

    public function filterPrice($number, $decimals = 0)
    {
        $price = number_format($number, $decimals);
        $price = $price . '€';

        return $price;
    }
}
```

- Tout d'abord, dans une extension Twig, on peut définir des filtres et/où des fonctions.
- Ensuite, la déclaration se fait de la même façon : en premier argument, le nom du filtre (ou de la fonction) et en deuxième argument, l'appel à effectuer. Cet appel peut être une fonction de la classe que l'on référence avec $this, ou un appel à une autre classe.
- Grâce à l'autoconfiguration des services, Symfony va automatiquement reconnaître qu'il s'agit d'une extension Twig et rendre disponibles le filtre "price" et la fonction « lipsum".



Voici un exemple d'utilisation de ces extensions nouvellement créées :

```
{% set somePrice = 10.7567 %}
{{ somePrice|price(2) }} {# "10.75€" #}

{{ lipsum() }}
{#
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nam semper turpis et diam imperdiet, molestie scelerisque diam congue...
#}
```

En plus simple :
Dans property.php :

```
public function getFormattedPrice(): string
    {
        return number_format($this->price, '', ' ');
    }
```

Pour chaque fois que l'on utilise l'entité property dans twig on peut faire ceci :

```
<div class="text-primary"> {{ property.formattedPrice}} €</div>
```

Créez vos propres macros :

On définit une macro comme étant "une petite routine informatique pour automatiser une tâche répétitive ». Exemple :

```
<div class="alert alert-primary" role="alert">
        Message d'alerte principal
</div>
```

```
<div class="alert alert-success" role="alert">
        Message de succès
</div>
<div class="alert alert-danger" role="alert">
        Message d'erreur
</div>
<div class="alert alert-warning" role="alert">
        Message d'avertissement
</div>
<!-- ...  et bien d'autres —>
```

Pourrait être raccourci en :

```
{# /templates/macros/alertes.html.twig #}
{% macro alert(message, type) %}
<div class="alert alert-{{ type }}" role="alert">
  {{ message }}
</div>
{% endmacro%}
```

Et utiliser dans un template Twig :

```
{# /templates/exemple.html.twig #}
{% import 'alertes.html.twig' as utils %}

{{ utils.alert('Attention!', 'warning') }}
{{ utils.alert('Erreur fatale!', 'danger') }}
```

L'héritage de gabarit :

Par exemple, le template parent.html.twig :

```
{# /templates/parent.html.twig #}
<html>
   <head>
   <title>{% block titre %}Parent{% endblock%}</title>
   {% block stylesheets %}
      <link rel="stylesheet" href="styles.css" type="text/css" media="all" />
   {% endblock %}
   </head>
   <body>
      {% block contenuPrincipal %}
         Contenu du body
      {% endblock %}
      {% block javascripts %}{% endblock %}
   </body>
</html>
```

Et voici la page "Product" qui utilise "Parent" :

```
{# /templates/produit.html.twig #}
{% extends 'parent.html.twig' %}

{% block titre %}Page Produit{% endblock%}
```

```
{% block stylesheets %}
   {{ parent() }}
   <link rel="stylesheet" href="produit.css" type="text/css" media="all" />
{% endblock%}

{% block contenuPrincipal %}
   <p>Ceci est la page Produit!</p>
{% endblock %}

{% block javascripts %}
<script src="build/produit.js" />
{% endblock %}
```

Rendra le fichier HTML suivant à l'utilisateur :

```
<html>
   <head>
      <title>Page Produit</title>
      <link rel="stylesheet" href="styles.css" type="text/css" media="all" />
      <link rel="stylesheet" href="produit.css" type="text/css" media="all" />
   </head>
   <body>
      <p>Ceci est la page Produit!</p>
      <script src="build/produit.js" />
   </body>
</html>
```

La fonction parent() permet de récupérer le contenu du bloc dans le gabarit parent.
Pour résumer ce que nous avons fait ici :

- Nous avons rendu le titre de la page HTML configurable: par défaut, sa valeur est "Parent" si le bloc "titre" n'est pas surchargé ;
- Nous avons rendu le corps de la page configurable, et par défaut, il est vide ;
- Nous avons rendu l'ajout de fichier CSS et Javascript configurable : dans la page Produit, nous avons souhaité conserver l'appel au fichier CSS "styles.css" à l'aide de la fonction parent().

DOCTRINE

Lors d'un problème doctrine :

https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/index.html

Créer une base de données :

Afin de créer un base de données il faut d'abord se connecter à celle-ci. Rends toi sur le fichier .env et modifie les lignes suivante pour t'y connecter.

```
# IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
###< doctrine/doctrine-bundle ###
```

Ensuite insérer cette ligne de commande dans le terminal :

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

Doctrine :

Entity, Manager et Repository sont des classes qui nous permettent de n'avoir à écrire que très rarement du code SQL, elles le feront pour nous.

L'**Unit of Work** est lui utilisé pour gérer l'état des différents objets hydratés par l'**Entity Manager**. La synchronisation en base ne s'effectue que quand on exécute la méthode "**flush**" et est effectuée sous forme d'une transaction qui est annulée en cas d'échec.
L'Entity Manager fait donc le lien entre les "**Entités**", qui sont de simples objets PHP, et la base de données :

- À l'aide de la fonction **find**, il retrouve et hydrate un objet à partir d'informations retrouvées en base ;
- À l'aide de la fonction **persist**, il ajoute l'objet manipulé dans l'Unit of Work ;
- À l'aide de la fonction **flush**, tous les objets "marqués" pour être ajoutés, mis à jour et supprimés conduiront à l'exécution d'une transaction avec la base de données.

Pour marquer un objet pour la suppression, il faut utiliser la fonction **remove**.

Entity Manager :

Dès lors que les objets sont mappés et reconnus par l'Entity Manager, nous pouvons les manipuler. Pour mapper des entités à des tables, Doctrine est capable de comprendre les formats suivants :

- Des annotations dans les commentaires (dits "DocBlocks") des objets PHP ;
- Des fichiers XML ;

- Des fichiers PHP de configuration.

Le format recommandé par Doctrine est l'annotation.

Entity Doctrine :

Exemple d'une entité simple :

```
namespace AppBundle\Entity;
use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Entity()
* @ORM\Table(name="blog_article")
* */
class Article{
    /**
    * @ORM\Id()
    * @ORM\GeneratedValue(strategy="AUTO")
    * @ORM\Column(type="integer")
    */
    public $id;

    /**
    * @ORM\Column(type="string")
    */
    public $title;

    /**
    * @ORM\Column(type="text")
    */
    public $content;

    /**
    * @ORM\Column(type="datetime", name="date")
    */
    public $date;
}
```

L'annotation "Column"

L'annotation @ORM\Column permet de mapper une propriété PHP à une colonne de la base de données.

Par défaut, le nom de la colonne de la base de données sera le nom de la propriété PHP, mais il est possible de le surcharger. Si une propriété n'est pas marquée avec l'annotation, elle sera complètement ignorée.

Cette annotation a de nombreux attributs, tous documentés. Voici les plus utiles :

- **unique** qui définit que la valeur doit être unique pour la colonne donnée ;
- **nullable** qui autorise/interdit la valeur nulle ;
- **length**, pour les chaînes de caractères, définit la longueur.

Pour en savoir plus sur toutes les autres annotations possible :

https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/annotations-reference.html#annotations-reference

Notion de propriétaire et d'inverse

La notion de propriétaire et d'inverse est abstraite mais importante à comprendre. Dans une relation entre deux entités, il y a toujours une entité dite propriétaire, et une dite inverse. Pour comprendre cette notion, il faut revenir à la vieille époque, lorsque l'on faisait nos bases de données à la main. L'entité propriétaire est celle qui contient la référence à l'autre entité. Attention, cette notion — à avoir en tête lors de la création des entités — n'est pas liée à votre logique métier, elle est purement technique.
Prenons un exemple simple, toujours les commentaires d'un article de blog. Vous disposez de la table **commentaire** et de la table **article**. Pour créer une relation entre ces deux tables, vous allez mettre naturellement une colonne **article_id** dans la table commentaire. La table **commentaire** est donc propriétaire de la relation, car c'est elle qui contient la colonne de liaison **article_id**. Assez simple au final !
N'allez pas me créer une colonne **article_id** dans la table des commentaires ! C'est une image, de ce que vous faisiez avant. Aujourd'hui on va laisser Doctrine gérer tout ça, et on ne va jamais mettre la main dans PhpMyAdmin. Rappelez-vous : on pense objet dorénavant, et pas base de données.

Notion d'unidirectionnalité et de bidirectionnalité

Cette notion est également simple à comprendre : une relation peut être à sens unique ou à double sens. On ne va traiter dans ce chapitre que les relations à sens unique, dites unidirectionnelles. Cela signifie que vous pourrez faire <?php **$entiteProprietaire->getEntiteInverse()** (dans notre exemple <?php **$commentaire->getArticle())**, mais vous ne pourrez pas faire <?php **$entiteInverse->getEntiteProprietaire()** (pour nous, <?php **$article->getCommentaires()**). Attention, cela ne nous empêchera pas de récupérer les commentaires d'un article, on utilisera juste une autre méthode, via l'EntityRepository.
Cette limitation nous permet de simplifier la façon de définir les relations. Pour bien travailler avec, il suffit juste de se rappeler qu'on ne peut pas faire **$entiteInverse->getEntiteProprietaire()**.
Pour des cas spécifiques, ou des préférences dans votre code, cette limitation peut être contournée en utilisant les relations à double sens, dites bidirectionnelles

Les relations/associations :

Après avoir créé l'entité, le nom de celle-ci sera demandé et ensuite le nom et le nom d'un champ. A ce moment on écrit relation, on choisit la table à relier puis on choisit parmi ça :

ManyToOne :

Plusieurs entités liées à 1 entité (liée à une OneTo-Many) .

OneToMany :

1 entité liée à plusieurs.

ManyToMany :

Plusieurs entités liées à plusieurs.
La recette est inversedBy et le hashtag mappedBy.

OneToOne :

1 entité est liée à 1 entité .

Les relations de type 1-1 :

```
class Command
{
    /**
     * Il y a un seul panier possible par commande
     * @ORM\OneToOne(targetEntity="App\Entity\Cart")
     * @ORM\JoinColumn(name="cart_id", referencedColumnName="id")
     */
    private $cart;

    // ...
}

/**
 * Objet représentant un panier d'achats.
 * @Entity()
 */
class Cart
{
    // ...
}
```

Ici, nous avons utilisé deux annotations de Doctrine, **OneToOne** et **JoinColumn**:

-**OneToOne** permet de définir la relation entre les deux entités en permettant de définir une entité liée à l'aide du paramètre targetEntity ;

- **JoinColumn** est une annotation facultative : elle permet de définir la clé étrangère qui fait office de référence dans la table.

Pour une relation bidirectionnelle :
https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/association-mapping.html#one-to-one-bidirectional

Les relations de type 1-n :

```
/**
 * Objet qui définit un client
 */
class Customer
{
    /**
     * Un client a potentiellement plusieurs adresses
     * @ORM\OneToMany(targetEntity="App\Entity\Address", mappedBy="customer")
     */
    private $addresses;
```

```
    // ...

    public function __construct() {
        $this->addresses = new ArrayCollection();
    }
}

/**
 * Objet qui définit une adresse
 */
class Address
{
    // ...

    /**
     * Les adresses sont liées à un client
     * @ORM\ManyToOne(targetEntity="App\Entity\Customer", inversedBy="adresses")
     * @ORM\JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;
}
```

- L'entité **propriétaire** doit définir l'attribut "mappedBy" : il correspond à la **propriété de l'objet** "possédé" qui fait le lien entre les deux entités.
- L'entité **possédée** doit définir l'attribut "inversedBy" : il correspond à la **propriété de l'objet** "propriétaire" qui fait le lien entre les deux entités.

Soit :

- L'annotation **ManyToOne** a un attribut **inversedBy** ;
- L'annotation **OneToMany** a un attribut **mappedBy**.

<u>Les relations de type n-n :</u>

```
/**
 * Objet qui définit un produit
 */
class Product
{
    /**
     * Un produit peut être mis dans plusieurs paniers
     * @ORM\ManyToMany(targetEntity="App\Entity\Cart", inversedBy="products")
     * @JoinTable(name="products_carts")
     */
    private $carts;

    // ...

    public function __construct() {
        $this->carts = new ArrayCollection();
    }
}

/**
```

```
 * Objet qui définit un panier
 */
class Cart
{
    // ...

    /**
     * Les produits sont liés à un panier
     * @ORM\ManyToMany(targetEntity="App\Entity\Products", mappedBy="carts")
     */
    private $products;

    // ...

    public function __construct() {
        $this->products = new ArrayCollection();
    }
}
```

Comme pour les relations de type **1-n** : quelle entité, quel objet est "**propriétaire**" de cette relation ? Ici, c'est le panier qui possède des produits, nous utilisons donc "**mappedBy**" pour l'entité **Cart**.

Select2 jquery :

https://cdnjs.com/libraries/select2

Puis la libraire jQuery à mettre avant :

https://code.jquery.com/jquery/

Rendre une relation obligatoire :

Par défaut, une relation est facultative, c'est-à-dire que vous pouvez avoir un Article qui n'a pas d'Image liée. C'est le comportement que nous voulons pour l'exemple : on se donne le droit d'ajouter un article sans forcément trouver une image d'illustration. Si vous souhaitez forcer la relation, il faut ajouter l'annotation JoinColumn et définir son option nullable à false, comme ceci :

```
/**
 * @ORM\OneToOne(targetEntity="Sdz\BlogBundle\Entity\Image")
 * @ORM\JoinColumn(nullable=false)
 */
private $image;
```

Les migrations dans Symfony :

Php bin/console make:migration

Permet de créer une migration SQL à partir des entités présentes

Ecrire ensuite mettre à jour la base de données.

Pour voir l'état de la migration de la précédente :

Php bin/console doctrine:migrations:status

Pour revenir à une migration précédente faire :

Php bin/console make:migrate timestampDeLaVersionVoulue

Mettre à jour la base de données avec les migrations :

Php bin/console doctrine:migrations:migrate

Il y aura un travail de différence entre les classes de l'application et les tables de la base de données. Tout ce qui ne sera pas dans les fichiers Entity de Symfony sera ajouté (ou supprimé) de la base de données concernée.
Cette migration (comme toutes les autres) sera versionnée afin que n'importe qui possède le projet puisse utiliser toutes ces migrations pour à partir de rien, avoir la table comme celle modifiée dans le projet.
Ceci lancera les scripts de migrations afin de mettre à jour la base de données.

Intégrer les changements à la base de données sans migration :

Php bin/console doctrine:schema:update —force

Revenir à une migration précédente :

Php bin/console doctrine:migrations:status

Cette ligne permet de savoir sous quelle migration on est, ainsi que les précédentes.

Php bin/console doctrine:migrations:migrate 'DoctrineMigrations\Version20180605025653'

Et celle-ci fera revenir la base de données à la migration voulue.

Les fixtures :

Les fixtures permettent de créer des jeux de fausses données, utilisable à souhait et ré-utilisable par les autres.
Il faut tout d'abord l'installer (il n'est pas installé par défaut)

Composer require orm-fixtures —dev

Ensuite on peut créer une fixture :

Php bin/console make:fixtures

Le terminal demande le nom de la fixtures, exemple : ArticlesFixtures
Et un fichier à été créé dans src/DataFixtures/ArticlesFixtures.php avec dedans la classe suivante :
Namespace App/Entity est visible dans le haut du fichier Articles.php

```
        | |
        V
Use App\Entity\Articles   // <= Ne pas oublier !
public function load(ObjectManager $manager){
      for($i=0; $i < 10; $i++){
         $article = new Articles;
         $article->setTitle("Titre de l'article $i")
               ->setContent("contenu de l'article n°$i")
               ->setImage("http://placehold.it/350x150")
               ->setCreatedAt(new \Datetime());

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

$manager->persist($article) permet de mettre en attente la fixture qui sera validé par $manager->flush();

Charger les fixtures :

Php bin/console Doctrine:fixtures:load
Charge toutes nos fixtures dans la base.

Charger les fixtures sans effacer les données de la base :

Php bin/console Doctrine:fixtures:load —append

Installer faker pour les fausses données :

composer require fzaninotto/faker
Installation du bundle
Voici le code pour créer les fausses données automatiquement :

```
use App\Entity\Articles;
use Faker;

class ArticlesFixtures extends Fixture{
   public function load(ObjectManager $manager){
      $faker = Faker\Factory::create('fr_FR');
      for($i=0; $i < 10; $i++){
         $article = new Articles;
         $article->setTitle($faker->Company)
               ->setContent($faker->text)
               ->setImage("http://placehold.it/350x150")
               ->setCreatedAt($faker->Datetime())
         $manager->persist($article);
      }
      $manager->flush();}}
```

Ensuite on charge le tout :
php bin/console doctrine:fixtures:load
Voir https://github.com/fzaninotto/Faker#fakerprovideren_uscompany
Pour plus de types de données.

Le repository :

Les repositories Doctrine ORM ont accès à deux objets principalement :
- L'EntityManager.
- Un QueryBuilder (un constructeur de requêtes) pour vous aider à faire des recherches plus fines dans la collection d'entités disponibles.

Chaque entité pour laquelle nous n'avons pas déclaré de repository particulier dispose d'un repository par défaut accessible dans tous les contrôleurs de nos applications.

Pour chercher tout ce que l'on souhaite à un moment donné on fait cette ligne de code :
$repo = $this->getDoctrine()->getRepository(Articles::class);
Sans oublier de déclarer où on peut trouver la class (ici **use App/Entity/Articles;**)

Quelques recherches avec repository :
$article = $repo->**find**(12); //Va chercher l'article 12 dans la table Articles
$article = $repo->**findOneByTitle**('Le titre à trouver')//Trouvera un article qui a ce titre
$article = $repository->**findOneBy**(['title' => 'Amazing article']);// Recherche d'un seul article par son titre

```
$article = $repository->findOneBy([
    'title' => 'Amazing Article',
    'price' => 'Zozor',
]); // Ou par titre et nom d'auteur
$articles = $repo->findByTitle('Le titre à trouver')//Trouvera tous  les articles ayant ce titre
$articles = $repository->findBy(
    ['author' => 'Zozor'],
    ['title' => 'ASC'], // le deuxième paramètre permet de définir l'ordre
    10, // le troisième la limite
    2 // et à partir duquel on récupère ("OFFSET" au sens MySQL)
);// Recherche de tous les articles en fonction de multiples conditions
$articles = $repo->findAll();//Cherche tous les articles
```

Quant il y a plein de repository à faire :

```
private $repository;
    public function __construct(PropertyRepository $repository){
        $this->repository = $repository;
    }
    /**
     * @Route("/biens", name="property.index")
     */
    public function index(): Response
    {
        $property = $this->repository->findOneBy(['title' => 'Amazing Article');
        return $this->render('property/property.html.twig', [
            'controller_name' => 'PropertyController',
            'current_menu' => 'properties']);}
```

Créer sa recherche repository personnalisée :

Pour cela il faut aller dans le fichier repository afin de créer la fonction permettant de faire une recherche SQL spécifique :

```
public function findLastArticles()
    {
        return $this->findBy([], ['publicationDate' => 'DESC']);
    }
```

Pour des recherches en SQL pure :

https://symfony.com/doc/current/doctrine.html#querying-for-objects-the-repository

Exemple :

Dans le repository de l'entité :

```
public function findAllVisible()
    {
        return $this->createQueryBuilder('p')
            ->Where('p.sold = false')
            ->getQuery()
            ->getResult();
    }
```

Dans le controller dans un fonction :

```
use Doctrine\ORM\EntityManagerInterface;
private $em;
   public function __construct(PropertyRepository $repository, EntityManagerInterface $em){
      $this->repository = $repository;
      $this->em = $em;
   }
$property = $this->repository->findAllVisible();
$this->em->flush($property);
```

OU

```
public function index(): Response
   {
      $property = $this->repository->findAllVisible();

      $entityManager = $this->getDoctrine()->getManager();
      entityManager->flush($property);

      return $this->render('property/property.html.twig', [
         'controller_name' => 'PropertyController',
         'current_menu' => 'properties'
      ]);
   }
```

Créer sa recherche SQL :

```
Public function findAllVisible(){
   Return $this->createQueryBuilder(‘p’)
                        //Créer un objet qui conçoit une requête avec pour référence ‘p’
                  ->andWhere(‘p.exempleField = :val’)
                        //On peut créer des conditions
                  ->setParameter()
                        //Des paramètres
                  ->orderBy(‘p.id', ‘ASC’)
                        //Un classement
                  ->setMaxResults(10)
                        //Une limite de résulat
                  ->getQuery()
                        //On récupère la requête
                  ->getResult();
                        //Et enfin on récupère les résultats
}
```

Dans le fichier Repository concerné on peut créer une nouvelle fonction qui lancera la requête qu’on lui aura construite.

Ici on souhaite voir tous les appartements ayant pour sold => false car ce seront tous ceux qui n’ont pas encore été vendu/loué :

```
Public function findAllVisible(){
   Return $this->createQueryBuilder(‘p’)
                  ->andwhere(‘p.sold = false’)
                  ->getQuery()
```

```
                ->getResult() ;
}
```

Ensuite dans le controller on peut écrire ceci :

```
Public function __construct(PropertyRepository $repository){
        $this->repository = $repository;
}
$property = $this->repository->findAllVisible();
```

Transmettre les informations à Twig :

Dans controller, ne pas oublier d'appeler les classes avec use :

```
Use App\Entity\Articles;
public function index(){
      $repo = $this->getDoctrine()->getRepository(Articles::class);
      $articles = $repo->findAll();

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

Les données recherchées au dessus sont transmises ici pour pour y accéder au code en dessus qui est la page index.html.twig

```
{% for article in tb_articles %}
   <article class="container my-4 p-4 mx-auto border">
      <h2>{{ article.title }}</h2>
      <div class=« metadata">
            Ecrit le {{ article.createdAt | date('d/m/Y') }} à {{ article.createdAt | date('H:i') }}
      </div>
      <img src="{{ article.image }}">
      <div class="content">
         <p>{{ article.content }}</p>
      </div>
      <a href="{{ path("blog_show", {'id': article.id}) }}" class="btn btn-primary">Lire la suite</a>
   </article>
{% endfor %}
```

Le chemin path() est renseigné comme ceci :

```
/**
    * @Route("/blog/{id}", name="blog_show")
   */
   public function show($id){
      $repo = $this->getDoctrine()->getRepository(Articles::class);
      $article = $repo->find($id);

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

Ceci est le concept de la route paramétrée.

L'injection de dépendances :

```
use App\Repository\ArticlesRepository;

public function index(ArticlesRepository $repo)
    {
        $articles = $repo->findAll();

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

Simplifie le code fait auparavant, cela s'appelle la création d'une dépendance.

```
/**
     * @Route("/blog/{id}", name="blog_show")
    */
public function show(Articles $articles){

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

Symfony comprendra seul quel article prendre par rapport à son id (qui est demandé plus haut {id}

Rediriger la page sur une autre :

Exemple d'un bout de code à la fin d'un controller :

```
return $this->redirectToRoute('blog_show', ['id' => $article->getId()]);
```

En premier le nom de la route et en 2ème (facultatif) les paramètres.

Pagination :

https://github.com/KnpLabs/KnpPaginatorBundle

```
composer require knplabs/knp-paginator-bundle
```

Créer un fichier dans src/packages/knp_paginator.yaml et insérer :

```
knp_paginator:
    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 parameter 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
```

Dans le repository renvoyer non plus le résultat mais la requête :

```
Use Doctrine\ORM\Query;
public function findAllVisible(): Query
  {
    return $this->findVisibleQuery()
            ->getQuery();
  }
```

Afin d'avoir la pagination dans Twig :

```
<div class="navigation">
        {{ knp_pagination_render(properties) }}
    </div>
```

Dans knp_paginator.yaml, changer le template :

```
template:
    pagination: '@KnpPaginator/Pagination/twitter_bootstrap_v4_pagination.html.twig'
```

Changer next/suivant de la pagination :
Créer un fichier KnpPaginatorBundle.fr.yaml, puis faire la traduction des mots concernés :

```
label_Next: Suivant
label_Previous: Précédent
```

Les différents écouteurs doctrine :

https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/events.html

preRemove - The preRemove event occurs for a given entity before the respective EntityManager remove operation for that entity is executed. It is not called for a DQL DELETE statement.

postRemove - The postRemove event occurs for an entity after the entity has been deleted. It will be invoked after the database delete operations. It is not called for a DQL DELETE statement.

prePersist - The prePersist event occurs for a given entity before the respective EntityManager persist operation for that entity is executed. It should be noted that this event is only triggered on initial persist of an entity (i.e. it does not trigger on future updates).

postPersist - The postPersist event occurs for an entity after the entity has been made persistent. It will be invoked after the database insert operations. Generated primary key values are available in the postPersist event.

preUpdate - The preUpdate event occurs before the database update operations to entity data. It is not called for a DQL UPDATE statement nor when the computed changeset is empty.

postUpdate - The postUpdate event occurs after the database update operations to entity data. It is not called for a DQL UPDATE statement.

postLoad - The postLoad event occurs for an entity after the entity has been loaded into the current EntityManager from the database or after the refresh operation has been applied to it.

loadClassMetadata - The loadClassMetadata event occurs after the mapping metadata for a class has been loaded from a mapping source (annotations/xml/yaml). This event is not a lifecycle callback.

onClassMetadataNotFound - Loading class metadata for a particular requested class name failed. Manipulating the given event args instance allows providing fallback metadata even when no actual metadata exists or could be found. This event is not a lifecycle callback.

preFlush - The preFlush event occurs at the very beginning of a flush operation.

onFlush - The onFlush event occurs after the change-sets of all managed entities are computed. This event is not a lifecycle callback.

postFlush - The postFlush event occurs at the end of a flush operation. This event is not a lifecycle callback.

onClear - The onClear event occurs when the EntityManager#clear() operation is invoked, after all references to entities have been removed from the unit of work. This event is not a lifecycle callback.

Vider le cache pour actualiser :

Php bin/console cache:clear

LES FORMULAIRES

Créer un formulaire :

Php bin/console make:form

Demande l'entité concernée. Ce sera Articles pour l'exemple

Créer un fichier qui doit finir par Type (convention de nommage de formulaire) et à l'intérieur on pourra modifier le fichier quand on le souhaite.

2 fonctions sont à l'intérieur du fichier Type :

buildForm qui sert à construire le formulaire accessible à partir du fichier controller avec **createForm(PropertyType::class, $property)**

$property est une entité. Exemple :

```
/**
     * @Route("/admin/edit/{id}", name="admin.property.edit")
 */
  public function edit(Property $property){
     $form = $this->createForm(PropertyType::class, $property);
     return $this->render('admin/property/edit.html.twig', [
        'property' => $properties,
        'form' => $form->createView()
     ]);}
```

configureOptions qui sert à configurer les options.

Afficher un message après la validation d'un formulaire :

Fichier controller :

```
$this->addFlash('success', 'Bien supprimé avec succès');
```

Fichier Twig :

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

createFormBuilder :

```
$form = $this->createFormBuilder($article)
                ->add('title')
                ->add('content')
                ->add('image')
                ->getForm():
```

Chaque **->add** représente un champ ajouté. Le **getForm** finalise le formulaire et donne dans la variable **$form** l'objet souhaité. Et nous donnera un objet **$form** complexe mais simple à utiliser.

Pour les ManyToMany :

```
Use Symfony\Bridge\Doctrine\Form\Type\EntityType;
->add('hashtags', EntityType::class, [
        'class' => Option::class, //Ceci est la classe qui doit être ciblée
        'choix_label' => 'name' //Pour la sélection quel champ utiliser (ici name)
        'multiple'=> true //On peut sélectionner plusieurs valeurs
])
```

Voici ce qui doit être mis dans le controller :

```
use App\Form\ArticleType;
use Symfony\Component\HttpFoundation\Request;
public function create(Request $request){

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


    return $this->render('blog/create.html.twig', [
        'formArticle' => $form->createView()
    ]);
  }
```

Et dans TWIG :

```
{{ form_start(formArticle) }}
  {{ form_widget(formArticle) }}
{{ form_end(formArticle) }}
```

Traduire les labels :

Méthode à utiliser pour renommer les labels à notre sauce, plus élégante et potentiellement utile.
Dans le fichier Type :

```
public function configureOptions(OptionsResolver $resolver){
        $resolver->setDefaults([
            'data_class' => Property::class,
            'translation_domain' => 'forms'
        ]);}
```

Ensuite créer un fichier forms.fr.yaml dans le dossier translation.
Aller dans Config/Packages/translation.yaml puis :

```
framework:
    default_locale: fr
    translator:
        default_path: '%kernel.project_dir%/translations'
        fallbacks:
            - fr
```

Et maintenant dans le fichier forms.fr.yaml Exemple :

```
City : Ville
Address : Adresse
Postal_code : Code postal
Title : Titre
```

Mettre l'input souhaité via le controller :

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
->add('content', TextType::class)
```

Il faut bien déclarer la classe grâce à use et ensuite écrire le type dans l'input souhaité.
Plus d'info à l'adresse suivante :
https://symfony.com/doc/current/forms.html#form-types

Insérer un attribut dans l'input via le controller :

```
->add('content', TextType::class, [
                    'attr' => [
                        'placeholder' => "Contenu de l'article",
                        'class'       => "form-control"
                    ]
                ])
```

Afficher le formulaire dans TWIG :

```
 {{ form_start(formArticle) }}
        <div class="form-group">
            <label for="">Titre de l'article</label>
            {{ form_widget(formArticle.title) }}
        </div>
    {{ form_end(formArticle) }}
```

Cette façon de faire permet de créer le formulaire comme on le souhaite avec des balises html et le relier au controller grâce au **form_widget**.

Valider le formulaire :

Twig :

```
{# templates/default/new.html.twig #}
```

```
        {{ form_start(form) }}
            {{ form_row(form.title) }}
            {{ form_row(form.content) }}
            {{ form_row(form.author) }}

            <button type="submit" class="btn btn-primary">Créer</button>
        {{ form_end(form) }}
```

Controller :

```
// src/Controller/FormController.php

/**
 * @Route("/form/new")
 */
public function new(Request $request)
{
    $article = new Article();
    $article->setTitle('Hello World');
    $article->setContent('Un très court article.');
    $article->setAuthor('Zozor');

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

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        dump($article);
    }

    return $this->render('default/new.html.twig', array(
        'form' => $form->createView(),
    ));
```

Tout d'abord, nous mettons à jour le formulaire à l'aide des informations reçues de l'utilisateur : c'est le rôle de la fonction **handleRequest**().
Ensuite, nous vérifions que le formulaire a bien été soumis (**$form->isSubmitted**()) et qu'il est valide (**$form->isValid**()).
Enfin, nous pouvons manipuler notre article mis à jour avec les données du formulaire. En utilisant le WebProfiler, on accède aux données provenant du formulaire.

Dès lors que des règles de validation ont été définies, les formulaires sont capables d'utiliser le validateur automatiquement ! Dans notre cas, nous n'avions pas encore défini de règles de validation, alors notre formulaire est forcément valide

Le validator :

Le composant Validator de Symfony permet de créer un validateur :

```
use Symfony\Component\Validator\Validation;
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;
use App\Entity\Article;

$article = new Article();
```

```
$validator = Validation::createValidator();
$violations = $validator->validate($article, [
    new Length(['min' => 10]),
    new NotBlank(),
]);

if (0 !== count($violations)) {
    // Affiche les erreurs
    foreach ($violations as $violation) {
        echo $violation->getMessage().'<br>';
    }
}
```

Tout d'abord, la classe Validation du composant permet de créer une instance du validateur. Ce validateur a une méthode **validate** qui prend 2 arguments :

- La valeur à valider : ce peut être une chaîne de caractères, un tableau ou un objet ;
- Une collection de contraintes de validation.

Le résultat de cette fonction est une liste de violations qui permettent d'accéder aux messages d'erreurs correspondant aux contraintes qui n'ont pas été validées. Si la valeur est valide, alors la liste de violations est vide.

Lorsque nous soumettons notre formulaire et que nous appelons la fonction **isValid** du formulaire Symfony, les contraintes de validation déclarées dans la classe **Article** sont appliquées sur l'objet **$article** qui a été retrouvé à partir des données de la requête utilisateur.

Les contraintes de validation

Il existe de nombreuses contraintes de validation et il est possible d'en créer de nouvelles assez facilement. Voici la liste exhaustive des contraintes de validation supportées par Symfony :

Basique	Texte	Comparaison	Date
NotBlank	Email	EqualTo	Date
Blank	Length	NotEqualTo	DateTime
NotNull	Url	IdenticalTo	Time
IsNull	Regex	NotIdenticalTo	
IsTrue	Ip	LessThan	
IsFalse	Uuid	LessThanOrEqual	
Type		GreaterThan	
		GreaterThanOrEqual	
		Range	

Collection	Fichier	Nombres / Finances	Autres
Choice	File	Bic	Callback
Collection	Image	CardScheme	Expression
Count		Currency	All
UniqueEntity		Luhn	UserPassword
Language		Iban	Valid
Locale		Isbn	
Country		Issn	

Voir plus de contraintes :
https://symfony.com/doc/current/reference/constraints.html

Créer ses propres contraintes :
https://symfony.com/doc/current/validation/custom_constraint.html

Créer des règles à valider :

```
// src/Entity/Article.php
namespace App\Entity;
use Symfony\Component\Validator\Constraints as Assert;

class Article
{
    /**
     * @Assert\Length(
     *      min = 10,
     *      max = 70,
     *      minMessage = "Ce titre est trop court",
     *      maxMessage = "Ce titre est trop long"
     * )
     */
    private $title;

    /**
     * @Assert\Regex(« /^[0-9]{5}$/« )//Pour code postal

     */
    private $content;

    /**
     * @Assert\NotBlank(message = "Un auteur doit être associé à l'article")
     */
    private $author;

    /* ... */
}
```

Crée des objets/articles/entrées uniques :

Avant la création de l'entité :

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity(repositoryClass=PropertyRepository::class)
 * @UniqueEntity("title")
 */
class Property{}
```

Afficher ses messages d'erreurs :

Si on veut afficher les messages d'erreurs personnalisés au dessus il faut ajouter un attribut 'novalidate' au formulaire:

```
{{ form_start(form, {'attr': {'novalidate': 'novalidate'}}) }}
        {{ form_row(form.title) }}
        {{ form_row(form.content) }}
        {{ form_row(form.author) }}

        <button type="submit" class="btn btn-primary">Créer</button>
    {{ form_end(form) }}
```

Donner un rendu bootstrap au formulaire :

```
# config/packages/twig.yaml
twig:
   form_themes: ['bootstrap_4_layout.html.twig'] //Celle là (prise à l'adresse https://symfony.com/doc/current/form/bootstrap4.html)
```

Dans le chemin ci dessus il faut ajouter cette ligne, ce qui donne :

```
twig:
   default_path: '%kernel.project_dir%/templates'
   debug: '%kernel.debug%'
   strict_variables: '%kernel.debug%'
   exception_controller: null
   form_themes: ['bootstrap_4_layout.html.twig']
```

Maintenant pour l'utiliser dans TWIG :
En premier on dit d'où sort le thème :

```
{% form_theme formArticle ''''''bootstrap_4_layout.html.twig'''''' %}
//REFAIRE LES APOSTROPHES
```

Et ensuite on met simplement le reste à la suite :

```
 {{ form_start(formArticle) }}
    {{ form_widget(formArticle) }}
 {{ form_end(formArticle) }}
```

Les fonctions de Twig dédiées au composant Form :

- **form_start** rend la balise <form> : on peut définir l'action, l'attribut method et ajouter différents attributs à la balise ;
- **form_row** rend un champ de formulaire complet, label et messages d'erreurs compris ;
- **form_label** rend le label ;
- **form_widget** rend l'input ;
- **form_errors** rend le bloc d'erreurs ;
- **form_rest** rend tous les champs non encore rendus ;
- **csrf_token** rend un input hidden avec une clé contre les attaques de type CSRF ;
- **form_end** rend la balise fermante </form> .

Customisation d'un champ spécifique dans un gabarit :

```
{% form_theme productForm _self %}
```

```
{% block _product_price_widget %}
   <div class="price_widget">
      {{ block('form_widget_money') }}
      <i>Des frais de transaction seront ajoutés.</i>
   </div>
{% endblock %}

{{ form_widget(productForm.price) }}
```

Pour définir l'exact nom du fragment, il faut prendre le nom du formulaire (par exemple, pour la classe "ProductType", c'est "product", qui peut être redéfini grâce à la méthode getName() du type de formulaire) suivi du nom du champ, ici le champ « price".
On peut aussi le retrouver à partir de l'attribut "name" du champ HTML généré, qui vaudra product[price], ici.

Modifier la présentation d'un Row :

```
<div class="d-flex justify-content-center align-items-end mb-2">
        <div class="mr-2">
           {% block form_row %}
              {% set row_class = attr['form-group mb-0'] | default('') %}
              <div class="{{ row_class }}">
                 {{ form_label(form.nom) }}
                 {{ form_widget(form.nom) }}
                 {{ form_errors(form.nom) }}
              </div>
           {% endblock form_row %}
        </div>
     </div>
```

Rendu avec CSS :

```
{{ form_error(formArticle.title) }}
```

Permet d'avoir accès aux erreurs avec le css et d'en faire ce que l'on veut. Voir par soit même comment faire.

Gérer une collection de formulaire :

Manipuler la requête HTTP :

```
Use Symfony\Component\HttpFoundation\Request;
Request $request
```

C'est la classe qui permet d'analyser/manipuler la requête HTTP.

Traiter le formulaire :

```
use Symfony\Component\HttpFoundation\Request;
 $form->handleRequest($request);
      if($form->isSubmitted() && $form->isValid()){
         $article->SetCreatedAt( new \Datetime);

         $manager->persist($article);
         $manager->flush();
```

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

Regarder tout ce qui concerne le HTTP :

Use Symfony\Component\HttpFoundation\Request;
dd($request);
On peut regarder ensuite dans la vue le request:ParameterBag qui contient un objet contenant les données passées par POST
Pour le GET il faut regarder dans query::ParameterBag.

Accéder aux données transmises :

Use Symfony\Component\HttpFoundation\Request;
Request $request
Pour accéder aux données transmises on déclarer d'abord la classe on déclare la variable $request, on peut ensuite se servir de ses fonctions qui sont les suivantes :
$request->

- **request**: équivalent de $_POST;
- **query**: équivalent de $_GET ($request->query->get('name'));
- **cookies**: équivalent de $_COOKIE;
- **attributes**: pas d'équivalent - utilisé pour stocker les autres données
- **files**: équivalent de $_FILES;
- **server**: équivalent de $_SERVER;
- **headers**: Le plus souvent équivalent au sous-ensemble de $_SERVER ($request->headers->get('User-Agent')).

Je mets ça de côté :
Each property is a Symfony\Component\HttpFoundation\ParameterBag instance (or a sub-class of), which is a data holder class:

- **request**: Symfony\Component\HttpFoundation\ParameterBag or Symfony\Component\HttpFoundation\InputBag if the data is coming from $_POST parameters;
- **query**: Symfony\Component\HttpFoundation\InputBag;
- **cookies**: Symfony\Component\HttpFoundation\InputBag;
- **attributes**: Symfony\Component\HttpFoundation\ParameterBag;
- **files**: Symfony\Component\HttpFoundation\FileBag;
- **server**: Symfony\Component\HttpFoundation\ServerBag;
- **headers**: Symfony\Component\HttpFoundation\HeaderBag.

Récupérer les données du ParameterBag :

Toutes les instances Symfony\Component\HttpFoundation\ParameterBag on leurs méthodes pour récupérer et modifier les données :

- **all() :** Retourne les paramètres.
- **keys()** : Retourne les clés des paramètres.
- **replace()** : Remplace les paramètres actuels par des nouveaux.
- **add()** : Ajoute des paramètres.
- **get()** : Retourne un paramètre grâce au nom.
- **set()** : Définit un paramètre par son nom.
- **has()** : Retourne true si le paramètres existe.
- **remove()** : Supprime un paramètre.

- **count()** : Compte le nombre de paramètres présent dans la requête.

Trier les données du ParameterBag :

Les instances de Symfony\Component\HttpFoundation\ParameterBag ont aussi des méthodes permettant de filtrer les valeurs des input :

- **getAlpha()** : Retourne la valeur du paramètre par ordre alphabétique.
- **getAlnum()** : Retourne la valeur du paramètre par ordre alphabétique ET numérique.
- **getBoolean()** : Retourne la valeur du paramètre converti en booléen.
- **getDigits()** : Renvoie les chiffres de la valeur du paramètre.
- **getInt()** : Retourne la valeur du paramètre converti en nombre entier.
- **filter()** : Filtre le paramètres en utilisant la fonction PHP **filter_var**.

Gérer les getters :

Tous les getters prennent jusqu'à deux arguments : le premier est le nom du paramètre et le second est la valeur par défaut à retourner si le paramètre n'existe pas

```
// the query string is '?foo=bar'
$request->query->get('foo');
// returns 'bar'
$request->query->get('bar');
// returns null
$request->query->get('bar', 'baz');
// returns 'baz'
```

Lorsque PHP importe la requête, il traite les paramètres de la requête comme foo[bar]=baz d'une manière spéciale car il crée un array. Ainsi on peut obtenir le paramètre foo et retourner le array avec la clé bar et sa valeur;

```
// the query string is '?foo[bar]=baz'
$request->query->get('foo');
// returns ['bar' => 'baz']
$request->query->get('foo[bar]');
// returns null
$request->query->get('foo')['bar'];
// returns 'baz'
```

Identifier une requête :

```
// for a request to http://example.com/blog/index.php/post/hello-world
// the path info is "/post/hello-world"
$request->getPathInfo();
```

Simuler une requête :

```
$request = Request::create(
    '/hello-world',
    'GET',
    ['name' => 'Fabien']
);
```

On peut passer outre le PHP global avec :

```
 $request->overrideGlobals();
```

Rendu final du controller :

```
/**
     * @Route("/blog/new", name="blog_create")
     * @Route("/blog/{id}/edit", name="blog_edit")
    */
    public function form(Articles $article = null, Request $request, EntityManagerInterface $manager){
        if(!$article){
            $article = new Articles;
        }

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

        $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid()){
            $article->SetCreatedAt( new \Datetime);

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

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

        return $this->render('blog/create.html.twig', [
            'formArticle' => $form->createView()
        ]);
    }
```

Les validations pour le formulaire:

Dans l'entité concernée (ici Articles) :

```
Use Symfony\Component\Validator\Constraints as Assert;
 /**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(min=10, max=255, minMessage="Le titre est trop court ! Rallonge le."
)
     */
    private $title;
```

La validation se fait côté serveur (php) ET côté client (html).

Pour plus d'infos :
https://symfony.com/doc/current/validation.html

Les collections :

```
use Doctrine\Common\Collections\ArrayCollection;
$collection = new ArrayCollection([1, 2, 3]);
$filteredCollection = $collection->filter(function($count) {
    return $count > 1;
}); // [2, 3]
```

Permet d'avoir un tableau PHP avec des méthodes un peu comme le javascript, simplifie son utilisation et code.
Pour plus d'infos :
https://www.doctrine-project.org/projects/doctrine-collections/en/latest/index.html#selectable-methods

Créer un formulaire de recherche :

- Créer une entité (non relié à la base de données)
- Créer le formulaire utiliser \App\Entity\NomEntité pour le relier
- Insérer le formulaire dans un controller avec la vue dans le template (logique)
- Pour ne pas faire apparaitre de truc inutile dans la barre URL faire cette fonction dans EntitéType :

```
public function getBloxPrefixe(){
    return "";
}
```

-

CONTACT

Créer une entité contact :

Pas nécessairement connecté à la base de données avec dedans prénom, nom, tel, email, message et property(ou GuyDemarle::class) si l'on souhaite se renseigner sur un bien/moule

Insérer dans le controller concerné (du bien) le formulaire :

```
$contact = new Contact;
    $contact->setProperty($property);
    $form = $this->createForm(ContactType::class, $contact);

    $form->handleRequest($request);

    if($form->isValid() && $form->isSbmitted()){
        $this->addFlash('success','Votre email a bien été envoyé.');

        return $this->redirectToRoute('property.show', [
            'id' => $property->getId(),
            'slug' => $property->getSlug()
        ]);
    }
```

Créer un message de succès avec addFlash :

Dans le controller :

```
$this->addFlash('success', 'Le message à afficher');
```

Dans twig :

```
{% for message in app.flashs('success') %}
```

```
        <div class="alert alert-success »>{{ message }}</div>
{% endfor %}
```
Ou l'équivalent dans une div choisie.
{{ app.session.flashbag.peek('success')|lenght > 0 ? '' : « mb-5 »}}.

Créer un dossier /src/Notification :

Puis un fichier ContactNotification :
Créer une fonction public notify()

Installer mailDev (simule les mails) :
`npm i maildev`

Accéder aux super variables de PHP avec Request :

```
use Symfony\Component\HttpFoundation\Request;
// Récupération des valeurs accessibles dans les super variables
$request = Request::createFromGlobals();
// Récupérer l'url
$request->getPathInfo();
//Récupérer des attributs en GET ou POST
$request->query->get('name');
$request->request->get('name', 'nom par défaut');

$request->getMethod();    // e.g. GET, POST, PUT, DELETE ou HEAD
```

La classe **Request** permet de centraliser l'accès à toutes les super variables de PHP en une seule classe utilitaire.

```
public function show(Recettes $recette, HashtagRepository $hashtagRepo): Response
    {
        $hashtag = $hashtagRepo->findAll();

        $entityManager = $this->getDoctrine()->getManager();
        foreach($hashtag as $hash){
            $recette->addHashtag($hash);
            $entityManager->persist($recette);
        }

        $entityManager->flush();
        dd($recette);

        //dd($recette);
        //$recettesRepository->find();
```

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