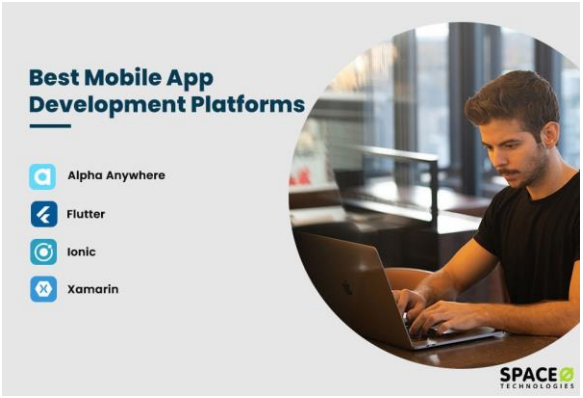
TP1 Web 2.0 : Symfony 6
FIA2-GL

Installation et Programmation contrôleur Web

sofiene.benahmed@issatso.rnu.tn

Table des matières



Best Mobile App Development Platforms (**Last Updated:** February 2, 2022)

Introduction

Cette séance permet de familiariser les étudiants avec l'environnement de programmation PHP et de mise au point basé sur Symfony, pour l'écriture de programmes interactifs en PHP lancés en ligne de commande.

Prérequis

On considère que vous avez acquis les notions de base du langage PHP.

Étape 1 : Prise en main de l'environnement d'exécution PHP en ligne de commande pour Symfony

Cette étape consiste à prendre en main l'environnement d'exécution de PHP en ligne de commande, ainsi que certains outils de mise au point pour le framework Symfony

Étape 1-a : vérification de l'environnement de mise au point PHP

1. Installer composer à travers ce lien : *https://getcomposer.org/download/*
2. Lancez un terminal
3. Vérifiez que la commande composer est présente sur notre installation de PHP, et que vous disposez d'une version de Composer supérieure ou égale à 2 :
4. composer -V

   par exemple :

   ```
   Composer version 2.1.6 2021-08-19 17:11:08
   ```

5. Vérifiez que l'interpréteur PHP en ligne de commande est présent, et que la version de PHP installée est supérieure ou égale à 7.4 :
6. php -v

   par exemple :

```
PHP 8.1.1 (cli) (built: Dec 15 2021 10:36:13) (NTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.1.1, Copyright (c) Zend Technologies
```

Étape 1-b : découverte de Composer

L'objectif de cette séquence est d'examiner les fonctionnalités de type ligne de commande fournies par la console Symfony, pour les développeurs

Définition :
Composer permet d'installer des centaines de packages librement disponibles. On les trouve sur Packagist . Il permet de gérer les dépendances d'un projet et également de créer le squellette d'une application Symfony et également d'installer des *recettes flex* (recipes) comme *composer*

require mailer pour installer *SwiftMailer* ou *composer require api* pour installer APIPlatform avec toutes ses dépendances.

Vous allez utiliser la commande create-project de composer, qui va créer le répertoire de développement d'une application Symfony et y télécharger le code source d'un squelette d'application.

1. Avec l'invite de commandes créer un nouveau dossier et se déplacer dedans.
2. Taper *composer create-project symfony/skeleton demo*
3. Taper *cd demo*
4. Lancer Visual studio Code en tapant *code .* (le point est exprès)
   Voici la signification des arguments :
   create-project commande create-project de composer (cf. documentation composer)
   symfony/skeleton référence du squelette d'application standard d'application Web
   demo répertoire de développement Symfony à créer.

   Ignorez les éventuels messages affichés à la fin de l'installation (« What's next? » , « Database Configuration » et « How to test? ») qui ne sont pas pertinents dans notre contexte, pour l'instant.

   Notations utile :

   bin/console : script exécutable PHP présent dans le sous-répertoire bin/, utilisé en ligne de commande pour le développement Symfony;

   src/ sources du squelette d'application Symfony. Il contient le code PHP de base de l'application sur laquelle vous travaillerez par la suite;

   vendor/ contient le code de toutes les bibliothèques PHP du framework Symfony qui seront utilisées. Elles ont été téléchargées par composer.

Étape 1-c : ligne de commande Symfony

L'objectif de cette séquence est d'examiner les fonctionnalités de type ligne de commande fournies par la console Symfony, pour les développeurs

La commande bin/console fournie propose d'accéder en ligne de commande à différentes fonctionnalités du framwork Symfony, notamment des utilitaires intéressants pour les phases de développement et de tests.

Essayez les commandes suivantes dans un terminal, depuis l'intérieur du répertoire du projet Symfony :

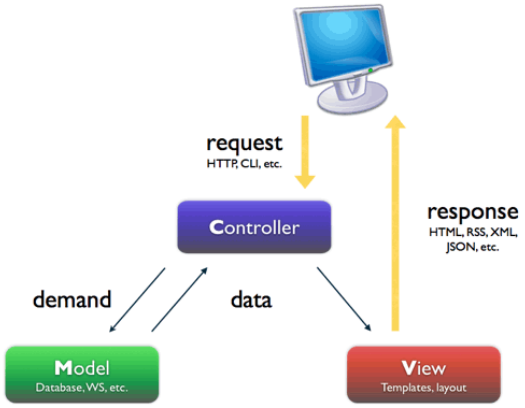
1. Vérifiez la version de Symfony utilisée dans le projet
2. php bin/console -V
3. Affichez la liste des sous-commandes disponibles (avec la sous-commande list, ou directement sans argument passé à bin/console) :
4. bin/console [list]

Lancer notre application web à travers Symfony CLI en tapant *symfony server:start*

L'architecture MVC

Pour le développement Web, les solutions les plus populaires pour organiser notre code de nos jours est la mise en place d'une **architecture MVC**. En résumé, l'architecture MVC définit un cadre d'organisation de notre code en accord avec sa nature. Ce modèle permet une séparation de notre code en **trois couches** :

- La couche **Modèle** contenant le traitement logique de vos données (les accès à la base de données se trouvent dans cette couche).
- La **Vue** est la couche où interagit l'utilisateur (un moteur de template fait parti de cette couche). Dans symfony, la couche vue est principalement faite de Templates PHP. Ces fichiers sont stockés dans les différents dossiers templates/ .
- Le **Contrôleur** est un morceau de code qui appelle le modèle pour obtenir certaines données qu'il passe à la Vue pour le rendu au client.



Étape 2 : Création d'un contrôleur

on doit installer tout d'abord le maker *symfony composer req maker –dev*

1) *php bin/console make:controller*
2) Nommer le contrôleur *BlogController*
3) On obtient les fichiers : *BlogController.php* et *template/blog/index.html.twig*
4) Examiner les 2 fichiers.
5) Dans le fichier *index.html.twig* changer le mot hello par salut et recharger la page *https://127.0.0.1:8000/blog*.
6) Dans la classe *BlogController* du fichier *BlogController.php* ajouter la méthode

```
#[Route('/', name: 'home')]
    public function home()
    {
        return $this->render('blog/home.html.twig');
    }
```

Et pour que ça fonctionne il faut créer un nouveau fichier dans le répertoire *blog*. Voici son code pour le moment : *<h1>Soyez le Bien Venue</h1> (p>lorem40)*2* (ajouter l'extension Emmet à Visual Studio code). Dans le navigateur chercher *https://127.0.0.1:8000/*

Le langage Twig

Définition

Twig est un moteur de templates pour le langage de programmation PHP, utilisé par défaut par le framework Symfony. Il a été inspiré par Jinja, moteur de template Python.

- Twig permet de séparer la présentation des données du traitement : Twig permet de définir en dehors de la page web des filtres que l'on pourra appliquer à la donnée.
- Twig permet la personnalisation de page web : Un block menu, un block recherche, un block contenu. . . le tout défini dans un template lui-même héritable.
- Twig permet de rendre les pages web plus lisibles, plus claires. Twig par son langage est moins invasif que PHP et se substitue à celui-ci.
- Twig est rapide.
- Twig apporte de nouvelles fonctionnalités.
- Twig est facile à apprendre.

Exercice

1) Modifier le fichier `blog/home.html.twig` à fin qu'il affiche les entiers entre 1 et 10
2) Modifier le fichier `blog/home.html.twig` à fin qu'il affiche les entiers entre 1 et x (x est un paramètre de la méthode `render`.

Étape 3 : Construction de Vue : le fichier templates/base.html.twig

Il est nécessaire d'installer twig en tapant `symfony composer req twig`
Observer le fichier base.html.twig :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

Ce fichier joue le rôle de référence de mise en page pour tout les interfaces. Il décrit tout ce qu'on doit mettre dans le head et le body de la page. Pour ce faire bootswatch nous plusieurs templates à exploiter. Visitez le site https://bootswatch.com et observer les thèmes offerts par bootsrap. Choisir le thème Flatly puis afficher dans le navigateur le fichier `flatly/bootstrap.min.css`. Il suffit maintenant d'insérer le lien vers ce thème.

```
    <!DOCTYPE html>
<html>
    <head>
```

```
    <meta charset="UTF-8">
    <title>{% block title %}Welcome!{% endblock %}</title>
<link rel="stylesheet" href="https://bootswatch.com/5/flatly/bootstrap.min.css">
    {% block stylesheets %}{% endblock %}
</head>
```

Puis choisir le code source d'un Navbar de votre choix et l'insérer juste après le <body> dans le fichier.

```
<body>
#insérer ici le code associé votre Navbar puis éliminer les composants qui ne sont pas nécessaire à ce moment
    {% block body %}{% endblock %}
    {% block javascripts %}{% endblock %}
</body>
```

En fin pour que les pages s'inscrivent au nouveau thème il suffit d'ajouter à l'entête de fichier *index.html.twig* par exemple :

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

Et indiquer le contenu de body :

```
{% extends 'base.html.twig' %}
{% block body %}
#inserer ici votre contenu exemple :
(article>h2{Titre de l'Article }+div.metadata{Edité le
06/02/2022}+div.content>img+(p>lorem15)*2+a.btn.btn-primary{Lire la suite})*3
{% endblock %}
```

Pour accéder au détail de chaque Article on ajoute dans le fichier *BlogController* la méthode suivante :

```
 #[Route('/blog/12', name: 'blog_show')]

    public function show()
    {
       return $this->render('blog/show.html.twig');
    }
```

Dans le fichier index.htm.twig ajouter (Pour chaque article)

```
<a href="{{ path('blog_show) }}" class="btn btn-primary">Lire la suite</a>
```

Dans le dossier blog créer un nouveau fichier show.html.twig et écrire dedans le code suivant :

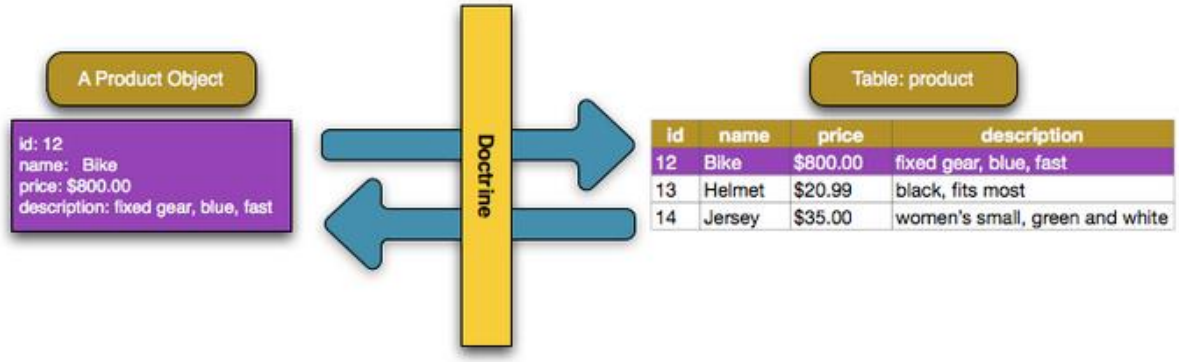
```
{% extends 'base.html.twig'  %}

{% block body %}

<article>
<article>
        <h2>Titre de l'Article </h2>
        <div class="metadata">Edité le 06/02/2022</div>
        <div class="content">
            <img src="http://placehold.it/100" alt="">
            <p>Lorem ipsum dolor sit, amet consectetur adipisicing elit. Rem
debitis maiores facilis exercitationem quas explicabo.</p>
```

```
            <p>Sequi molestiae natus voluptates blanditiis dolores nihil earum
amet, vel dignissimos voluptatem, beatae animi quos.</p>
        </div>
    </article>
</article>
{% endblock %}
```

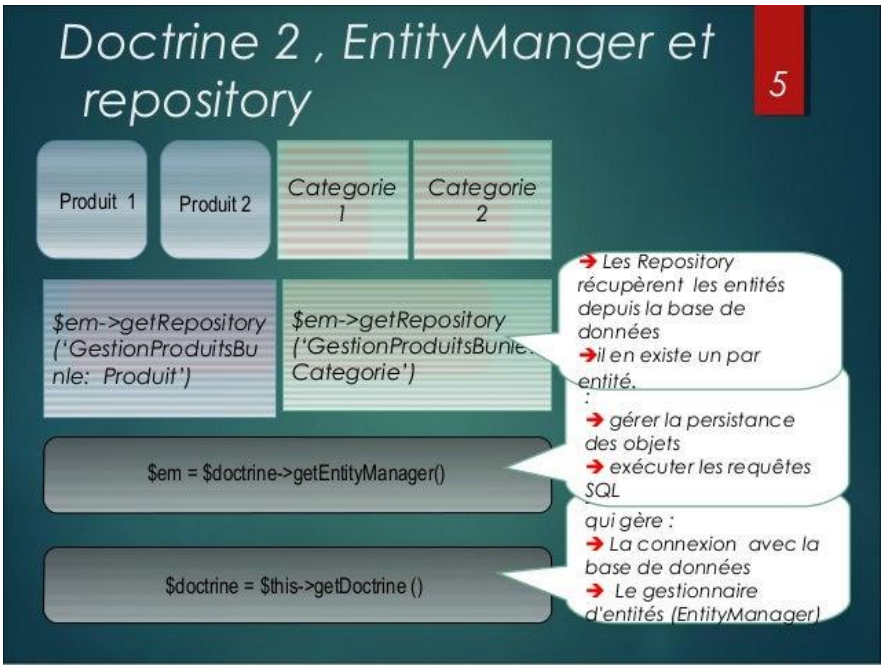
Etape 4 : L'ORM (Object-Relatinal Mapping) de Symfony Doctrine

Définition Les ORM sont des frameworks qui permettent de créer une correspondance entre un modèle objet et un modèle relationnel de base de données. Un ORM fournit généralement les fonctionnalités suivantes :

- Génération à la volée des requêtes SQL les plus simples (CRUD)
- Prise en charge des dépendances entre objets pour la mise en jour en cascade de la base de données
- Support pour la construction de requêtes complexes par programmation



Doctrine de Symfony



Différentes fonctionnalités offertes par Doctrine

Doctrine-migrations

Une *migration* est une classe qui décrit les changements nécessaires pour mettre à jour un schéma de base de données, de son état actuel vers le nouveau, en fonction des attributs de l'entité.

Les fixtures

Les fixtures vont permettre d'insérer de fausses données en base de données pour nous aider à développer plus facilement. Elles sont très pratiques, car elles nous évitent de devoir systématiquement remplir des formulaires ou insérer à la main des données pour vérifier le fonctionnement de certaines parties du code.

Installation :

1) composer require symfony/orm-pack
   composer require --dev symfony/maker-bundle
   composer req orm-fixtures
2) Charger le fichier php.ini et enlever le commentaire de ligne correspondante à votre serveur de BDD.
   Exemple :

```
extension=pdo_pgsql
;extension=pdo_sqlite
extension=pgsql
```

3) Charger le fichier .env puis modifier la ligne correspondante à votre serveur de BDD. Exemple :

```
DATABASE_URL="postgresql://postgres:root@127.0.0.1:5432/blog1?server
Version=13&charset=utf8"
```

4) Création d'une base des données.

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

- Donner le nom Article
- Donner les diffèrent attribues de cette classe (attention aux types doctrine)
  - Titre de type string
  - Contenu de type text
  - Pour ajouter autre champs après validation on écrit php bin/console make:entity Article.
  - Ajouter les deux champs :

5) Création de migration : php bin/console make:migration
6) Pour mettre à jour notre BDD on appelle : php bin/console doctrine:migrations:migrate
7) Pour installer fixture : symfony composer req orm-fixtures
8) Création de fixture : php bin/console make:fixtures nommé ArticleFixtures
9) Dans le fichier ArticleFixtures ajouter le code suivant dans la fonction load()

```
for($i=1;$i<=10;$i++)
        {
           $article=new Article();
            $article->setTitle("Titre de l'article n° $i")
            ->setContent("<p>Le contenu de l'article n° $i</p>")
-
>setImage("http://assets.stickpng.com/thumbs/584df6256a5ae41a83dde
e0f.png")
            ->setCreatedAt(new \datetime_immutable());
            $manager->persist($article);
}
        $manager->flush();
```

Ajouter la dépendance :

```
use app\Entity\Article;
```

10) Exécuter la méthode load en tapant : php bin/console doctrine:fixtures:load
11) Vérifier les données dans votre BDD.

- **Etape 5 : Exploiter aux données dans symfony**

Modifier la fonction index de la classe BlogController :

```
use App\Entity\Article;
use App\Repository\ArticleRepository;
use Doctrine\Persistence\ManagerRegistry;
```

```
…
public function index(ManagerRegistry $doctrine, ArticleRepository
$repo): Response
    {
        $articles=$repo->findAll('Titre de l\'article');
Return  $this->render('blog/index.html.twig',  ['controller_name'  =>
'BlogController','articles'=>$articles,]);
    }
```

Et le fichier index.html.twig :

```
{% block body %}
<section class='articles'>
{% for article in articles %}

    <article>
        <h2>{{article.title}} </h2>
        <div class="metadata">Edité le {{article.createdAt |
date("m/d/Y")}} à {{article.createdAt | date("h/i")}}</div>
        <div class="content">
            <img src="{{article.image}}" alt="">
            {{article.content | raw }}
            <a href="{{ path('blog_show') }}" class="btn btn-
primary">Lire la suite</a>
        </div>
    </article>
{% endfor %}
</section>
```

```
{% endblock %}
```

Pour afficher maintenant le détail de chaque article on va modifier la fonction show comme suit :

```
#[Route('/blog/{id}', name: 'blog_show')]


    public function show($id,ManagerRegistry $doctrine,
ArticleRepository $repo)
    {
        $article = $repo->find($id);
       return $this-
>render('blog/show.html.twig',['article'=>$article]);


    }
```

Et dans le fichier index.html.twig :

```
<a href="{{ path('blog_show' , {id: article.id}) }}" class="btn btn-
primary">Lire la suite</a>
```

En fin on modifie le fichier show.html.twig comme suit :

```
{% block body %}
<article>
    <h2>{{article.title}} </h2>
    <div class="metadata">Edité le {{article.createdAt |
date("m/d/Y")}} à {{article.createdAt | date("h/i")}}</div>
    <div class="content">
        <img src="{{article.image}}" alt="">
        {{article.content | raw }}
    </div>
</article>
```

```
{% endblock %}
```

Etape 6. Les formulaires

1. Méthode native :

   On veut créer à présent un un formulaire pour créer un nouveau article.

Les étapes à suivre sont :

- Dans la classe BlogController ajouter la méthode suivante :

```
#[Route('/blog/new', name: 'blog_create')]
public function create(Request $request, EntityManagerInterface
$manager)
        {

         if($request->request->count()>0)
           {
            $article = new Article();
            $article->setTitle($request->request->get('title'))
                ->setContent($request->request->get('content'))
                ->setImage($request->request->get('image'))
                ->setCreatedAt(new \DateTimeImmutable());
                $manager->persist($article);
                $manager->flush();
return $this->redirectToRoute('blog_show',['id'=>$article-
>getId()]);
            }
            return $this->render('blog/create.html.twig');
        }
```

N'oublier pas d'ajouter la dépendance pour gérer les injections:

```
use Doctrine\ORM\EntityManagerInterface;
```

Sous templates/blog créer le fichier **`create.html.twig`**

```
{% extends 'base.html.twig' %}
{% block body %}
<h1> Création d'un nouveau article!</h1>
<form action "" method="post">
    <input type="text" name="title" placeholder="Titre de
l'article">
    <textarea name="content" placeholder="Contenu de
l'article"></textarea>
```

```
    <input type="text" name="image" placeholder="image de
l'article">
    <button type="submit">Save</button>
</form>
{% endblock %}
```

Dans le fichier base.html.twig renommer le bouton feature par exemple en New et changer son href en :

```
href="{{ path('blog_create') }}
```

Vérifier le résultat

2. Méthode de Symfony :

Tout d'abord il faut installer symfony composer symfony/form

Modifier la fonction create comme suit :

```
#[Route('/blog/new', name: 'blog_create')]
  public function create(Request $request)
  {

          $article = new Article();
          $form =$this->createFormBuilder($article)
          ->add('title')
          ->add('content')
          ->add('image')
          ->getForm();
Return
$this>render('blog/create.html.twig',['formArticle'=>$form-
>createView()]);
  }
```

Explication en classe.

- **Configuration de formulaire**

```
$form =$this->createFormBuilder($article)
          ->add('title')…
```

Sachant que l'objet article est une instance de la classe Article Symfony fait la correspondance avec les champ de la classe et sait bien que le paramètre 'title' correspond effectivement à l'attribut title de la classe Article. Mais on peut configurer la méthode add, visiter : https://symfony.com/doc/current/reference/forms/types.html

La fonction devient :

```
#[Route('/blog/new', name: 'blog_create')]
  public function create(Request $request)
  {
```

```
          $article = new Article();
          $form =$this->createFormBuilder($article)
          ->add('title',TextType::class,[
              'attr'=>['placeholder'=>'Titre de l\'article']
          ])
          ->add('content',TextareaType::class,[
            'attr'=>['placeholder'=>'Contenu de l\'article']
        ])
          ->add('image',TextType::class,[
            'attr'=>['placeholder'=>'Image de l\'article']
        ])
          ->getForm();


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

Affichage de formulaire :

- A l'aide de twig

Le fichier create.html.twig :

Méthode 1 : {{ form(paramètre)}}

```
{% extends 'base.html.twig' %}
{% block body %}
<h1> Création d'un nouveau article!</h1>
{{form(formArticle)}}

{% endblock %}
```

Méthode 2 : {{ formwedget(paramètre)}}

```
{% extends 'base.html.twig' %}
{% block body %}
<h1> Création d'un nouveau article!</h1>
{{ form_start(formArticle)}}
<div class="form-group">
  <label for="">Titre de l'article</label>
  {{form_widget(formArticle.title)}}
  </div>
```

```
  <div class="form-group">
    <label for="">Contenu de l'article</label>
    {{form_widget(formArticle.content)}}
    </div>
    <div class="form-group">
        <label for="">Image de l'article</label>
        {{form_widget(formArticle.image)}}
        </div>
{{ form_end(formArticle)}}
{% endblock %}
```

Ajouter l'option 'class-=> 'form-control' dans attr du controlleur
Méthode 3 : bootstrap
Il suffit décrire :

```
{% extends 'base.html.twig' %}
{% block body %}
<h1> Création d'un nouveau article!</h1>
{{ form_start(formArticle)}}
     {{ form_widget(formArticle) }}

{{ form_end(formArticle)}}

{% endblock %}
```

Question Ajouter un bouton pour enregistrer de deux façons dans le contrôleur puis dans twig !
On peut alléger la fonction create en supprimant les option attr et les mettre dans le fichier twig :
Exemple :

```
{{ form_row(formArticle.title,{'attr':{'placeholder':'Titre de
l\'article'}}) }}
```

En fin pour manipuler les donner on va demander à symfony d'analyser la requête en tapant la fonction `handleRequest` si tout est est bon (c.à.d que la requête est validé) alors on ajoute la date de création avec la methode getCreatedAt. La fonction create devient alors :

```
#[Route('/blog/new', name: 'blog_create')]
    public function create(Request $request,EntityManagerInterface
$manager)
    {

            $article = new Article();
            $form =$this->createFormBuilder($article)
            ->add('title')
```

```
            ->add('content')
            ->add('image')
            ->getForm();
        $form->handleRequest($request);
        if($form->isSubmitted()&&$form->isValid())
        {
            $article->setCreatedAt(new \DateTimeImmutable());
            $manager->persist($article);
        $manager->flush();
    return $this->redirectToRoute('blog_show',['id'=>$article-
>getId()]);
        }

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

Expliquer l'instruction suivante :

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

Exercice transformer le formulaire à fin d'avoir deux fonctionnalité enregistrer un nouveau article et modifier article.

Indication il faut penser à ajouter une deuxième route.

La partie suivante on va voir comment créer un formulaire à l'aide de commandes.

Référence :

3. https://symfony.com
4. https://openclassrooms.com/fr/courses/5489656-construisez-un-site-web-a-l-aide-du-framework-symfony-5
5. Livre : Développer avec Symfony2 Clément Camin (Eyrolles)