SYMFONY : 7-Comprendre les formulaires

Créer la fonction create() dans BlogController.php après la fonction show()

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

Si la fonction create() est déclarée après la fonction show() , il va y avoir une confusion de route. En effet la fonction show() a pour route « /blog/{id} » donc la fonction attend un ID comme paramètre. En déclarant la fonction create() après la fonction show(), Symfony va penser que « new » dans la route sera un ID.

Dans le fichier create.html.twig ,n'écrire seulement :

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

{% block body %}

<h1>Création d'un article</h1>

{% endblock %}
```

Taper dans l'Url : http://localhost:8000/blog/new

Se rendre ensuite dans le fichier base.html.twig et modifier les href dans le menu (ex : path('home'))

Symfony nous permet de générer des balises afin de créer un formulaire, nous allons toutefois créer un formulaire comme habituellement afin de bien comprendre l'utilité de symfony et de sa simplicité.

Rendons nous dans la page create.html.twig et créons un formulaire html dans le block 'body'

```
<form action="" method="post">
    <label>Title</label>
    <input type="text" name="title" placeholder="titre de l'article">

    <label>Contenu</label>
    <textarea name="content" placeholder="Contenu de l'article"></textarea>

    <label>Image</label>
    <input type="text" name="image" placeholder="titre de l'article">

    <input type="submit" value="Enregistrer">
</form>
```

Nous allons maintenant demander à symfony d'exécuter une requête d'insertion à la soumission du formulaire, pour cela nous allons évoquer une injection de dépendance. La fonction recevra en argument un objet de la classe 'Request'

```
public function create(Request $request)
```

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

Ne pas oublier d'appeler la classe 'Request'

```
use Symfony\Component\HttpFoundation\Request;
```

Dans la fonction faire :

```
dump($request);
```

La classe 'Request' contient toute les données véhiculées par les superglobales ($_POST, $_GET, $_FILES etc…)

Remplir le formulaire et observer dans le dump() → request (parameterBag : sac de paramètres)→ parameters (ARRAY) , on observe un tableau ARRAY contenant les données saisies dans le formulaire.

Ajouter le code ci-dessous à la fonction create()

```
public function create(Request $request, EntityManagerInterface $manager)
    {
        // echo '<pre>'; var_dump($request); echo '</pre>';
        dump($request);

        // Si les paramètres sont supérieurs à 0, donc si nous avons bien saisie des donné
es dans le formulaire (voir dump(request->parameters))
        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 \DateTime());

            $manager->persist($article); // on prépare l'insertion
            $manager->flush(); // on insère
        }

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

Attention, nous faisons appel au manager, il faut faire appel à la classe :

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

Tester maintenant une insertion via le formulaire HTML, nous observons que l'insertion s'execute !! Cliquez sur le lien « Articles » dans la navigation.

Ajouter cette ligne pour rediriger vers la page show.html.twig après insertion :

```
return $this->redirectToRoute('blog_show', ['id' => $article-
>getId()]); // on redirige vers la page 'show.html.twig' avec le bon ID une fois l'article
 crée
```

Nous nous rendons compte que c'est un travail long et récurant, si nous avions d'autre champs, cela ferait beaucoup plus de code à écrire, il manque tous les contrôles, nous allons voir maintenant une solution beaucoup plus simple et plus rapide pour l'insertion d'un formulaire

Ajouter le code suivant (annule et remplace le code précédent) :

```
/**
     * @Route("/blog/new", name="blog_create")
     */
    public function create(Request $request, EntityManagerInterface $manager)
    {
        $article = new Article(); // nous déclarons un article qui est vide mais pret à êt
re rempli

        // $form est un objet complexe, nous allons demander à symfony de nous stocker le
formulaire dans une variable simple à utilier.
      $form = $this-
>createFormBuilder($article) // cela va créer un objet qui est lié à notre article
                -
>add('title') // add() fonction permettant de créer des champs dans un formulaire
                    ->add('content')
                    ->add('image')
                    ->getForm(); // permet d'afficher le rendu final

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

Ajouter un 2ème paramètre à la fonction render() afin de récupérer l'affichage du formulaire.

```
return $this->render('blog/create.html.twig', [
            // createView() va retourner un petit objet qui représente l'affichage du form
ulaire, on le récupère sur la page create.html.twig
            'formArticle' => $form->createView()
        ]);
```

Supprimer ensuite le formulaire codé à la main et écrire la ligne d'interpolation suivante :

```
{{ form(formArticle) }}
```

Form() est une fonction TWIG qui permet d'afficher le rendu d'un formulaire.

Rendre ensuite sur la page create.html.twig, on observe bien la création du formulaire

On observe que le champ 'content' est bien une balise <textarea>, en fait symfony est capable de savoir que ce champs est lié à une entité dans la BDD, et donc un champ de type 'TEXT' donc du texte long.

Nous pouvons maintenant ajouter les différents type de champs, en effet, symfony nous propose tout une série de type de champs (voir doc). Ne pas oublier l'appel de la classe :

```
 ->add('content', textType::class)
```

```
use Symfony\component\Form\Extension\Core\Type\TextType;
```

Nous avons modifié le <textarea> en champ type « text »

```
->add('title', TextType::class, [
                        // On définit les attributs du champs 'title'
```

```
                        'attr' => [
                            'placeholder' => "Contenu de l'article"
                        ]
                    ]) // add() fonction permettant de créer des champs dans un formulaire
```

Nous pouvons aussi définir le type de champs (text, textarea etc…) mais aussi les attributs comme ci-dessus mais autant faire confiance à Symfony et le laissé affecter les bonnes balises.

Symfony va nous permettre de gérer à la ligne près notre formulaire Symfony.

Ajouter ensuite sur create.html.twig les lignes suivantes :

```
{{ form_start(formArticle) }}
```

Form_start() va nous permettre d'appeler la bonne balise avec les bons attributs (method, action etc…), des données essentielles pour le formulaire.

Ajouter le code ci-dessous, si nous appelons seulement le champ 'title' par exemple, Symfony appel quand même les 2 autres champs même si ils ne sont pas déclarés, en effet symfony a un registre contenant tous les champs déclarés comme entités. Form_widget() permet de faire appel à un champ.

```
{{ form_start(formArticle) }}

    {# form() : fonction TWIG #}
    {# {{ form(formArticle) }} #}

    <div class="form-group">
        <label for="titre">Titre</label>
        {{ form_widget(formArticle.title) }}
    </div>

    <div class="form-group">
        <label for="titre">Contenu</label>
        {{ form_widget(formArticle.content) }}
    </div>

    <div class="form-group">
        <label for="titre">Image</label>
        {{ form_widget(formArticle.image) }}
    </div>

    {{ form_end(formArticle) }}
```

Nous pourrions aussi ajouter une classe au différents champs déclarés afin d'avoir un rendu bootstrap :

```
->add('content', TextType::class, [
                        'attr' => [
                            'placeholder' => "Choix de l'image",
```

```
                            'class' => 'form-control'
                        ]
                    ])
```

Symfony a créé des moteurs de rendu TWIG, nous allons donc rendre plus fluide le code et surtout plus simple en y intégrant un package bootstrap

Revenir sur cette synthaxe dans create.html.twig :

```
{{ form_start(formArticle) }}

    {{ form_widget(formArticle) }}

    {{ form_end(formArticle) }}
```

Ajouter cette ligne de code, c'est un package bootstrap, dans le dossier config->package->twig.yaml

```
# config/packages/twig.yaml
twig:
    form_themes: ['bootstrap_4_layout.html.twig']
```

Ajouter ensuite cette ligne dans create.html.twig pour préciser que nous allons utiliser le package bootstrap :

```
{% form_theme formArticle 'bootstrap_4_layout.html.twig' %}
```

Actualise la page et vous avez un formulaire bootstrap !!

Pour faire les fainéants, nous pouvons retirer

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

Ne laisser que :

```
 {{ form(formArticle) }}
```

Nous préférerons quand même garder form_start() et form_end() afin de personnalisé le formulaire à souhait. Ou bien ajouter des choses entre le début et la fin du formulaire.

Nous pourrions ajouter un bouton dans le BloController.php grâce aux lignes suivantes :

```
use Symfony\component\Form\Extension\Core\Type\SubmitType;
->add('save', SubmitType::class, [
                        'label' => 'Enregistrer'
                    ])
```

Le problème est que le bouton pourrait servir à la fois à supprimer, à modifier ou à insérer et que dans ce cas ci-dessous, le bouton aurait uniquement pour rôle d'insérer.

Nous allons donc nous-mêmes créer notre bouton submit :

```
{{ form_start(formArticle) }}

    {{ form_widget(formArticle) }}
```

```
    <button type="submit" class="btn btn-primary">Enregistrer</button>

    {{ form_end(formArticle) }}
```

Nous allons maintenant ré alléger le formulaire et faire confiance à SYMFONY, pour cela dans la fonction create() du BlogController.php , nous allons supprimer les type de champs ainsi que les attributs :

```
$form = $this-
>createFormBuilder($article) // cela va créer un objet qui est lié à notre article
                    // add() fonction permettant de créer des champs dans un formulaire
                    ->add('title')
                    ->add('content')
                    ->add('image')
                    ->getForm(); // permet d'afficher le rendu final
```

En revanche nous n'avons plus les attributs 'placeholder' ou 'classe' etc… Nous allons utiliser form_row() qui va nous permette d'afficher tout ce qu'il faut pour un champ (label, le champ et les erreurs potentielles)

Modifier le code sur le fichier create.html.twig

```
{{ form_start(formArticle) }}

    {{ form_row(formArticle.title, { 'attr': { 'placeholder': "Titre de l'article" } }) }}
    {{ form_row(formArticle.content, { 'attr': { 'placeholder': "Contenu de l'article" } }
) }}
    {{ form_row(formArticle.image, { 'attr': { 'placeholder': "URL de l'image" } }) }}

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

    {{ form_end(formArticle) }}
```

TRAITEMENT DES DONNEES DU FORMULAIRE :

Ajouter cette ligne dans le BlogController.php , on demande à symfony de controller si nous avons soumis le formulaire, il va chercher dans la requete si il y a une image, un titre, du contenu.

Si il trouve un title, il va le bindé avec le titre de l'article, si il y a une image, il va le bindé avec l'image de l'article etc…

```
$form->handleRequest($request);
```

Nous allons faire un dump($article) et on s'aperçoit qu'il est bel et bien vide. Nous allons remplir le formulaire et le valider. Regarder à nouveau le dump() et l'article est rempli !! Nous avons un article complètement vide mais lié au formulaire, symfony est capable de dire, tient il y a un champ 'title' dans la requete, nous allons le donner au champ 'title' de l'article

Pour traiter les données du formulaire, vous devrez appeler la handleRequest() méthode:

Dans les coulisses, cela utilise un NativeRequestHandler objet pour lire les données des superglobaux PHP corrects (ie $_POSTou $_GET) en fonction de la méthode HTTP configurée sur le formulaire (POST est par défaut).

Pour insérer ajouter le code suivant :

```
// si le formulaire est bien soumit et valide
        if($form->isSubmitted() && $form->isValid())
        {
            $article->setCreatedAt(new \DateTime()); // on ajoute la date à l'insertion

            $manager->persist($article); // on prépare l'insertion
            $manager->flush(); // on insère

            return $this->redirectToRoute('blog_show', ['id' => $article-
>getId()]); // on redirige vers la page 'show.html.twig' avec le bon ID une fois l'article
 crée
        }
```

Si on ajoute les 2 setteurs, on aperçoit qu'au premier chargement de la page les champs sont pré-remplis

```
$article = new Article(); // nous déclarons un article qui est vide mais pret à être rempl
i

        $article->setTitle('Titre à la con')
                ->setContent('Contenu de l\'article');
```

Nous allons donc nous servir de cette fonction pour à la fois ajouter et modifier un article, nous allons donc modifier le nom de la fonction , de create() à form()

```
/**
     * @Route("/blog/new", name="blog_create")
     * @Route("/blog/{id}/edit", name="blog_edit")
     */
    public function form(Request $request, EntityManagerInterface $manager)
    {   // initialement function create()
```

Ajouter un argument à la fonction form() : Article $article

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

Et commenter la ligne suivante :

```
$article = new Article();
```

Saisir l'url suivante : http://localhost:8000/blog/12/edit, on voit bien que le formulaire est bien rempli par l'article 12

Attention, si nous nous rendons sur la page : http://localhost:8000/blog/new, nous constatons une erreur, il y a une erreur parce que symfony pense aller chercher un article (il prend le mot 'new' pour un ID), ajouter donc la valeur de 'null' à l'argument $article :

```
public function form(Article $article = null, Request $request, EntityManagerInterface $ma
nager)
```

Ajouter cette condition : s'il n'y a pas d'article, on en crée un nouveau, lorsqu'il y a blog/new dans l'url

```
// Si il n' y a pas d'articles, ajouter un nouvel article
        if(!$article)
        {
            $article = new Article(); // nous déclarons un article qui est vide mais pret à être rempli
        }
```

Ajouter cette condition pour générer la date en cas d'insertion :

```
// si l'article n'a pas d'identifiant, donc pour une insertion, on ajoute la date de creation
            if(!$article->getId())
            {
                $article->setCreatedAt(new \DateTime()); // on ajoute la date à l'insertion
            }
```

Créer et modifier un article pour tester.

Nous allons maintenant modifier la valeur du bouton en cas d'insertion et de modification

Ajouter l'indice editMode à la méthode render() pour savoir si un ID existe ou non

```
return $this->render('blog/create.html.twig', [
            // createView() va retourner un petit objet qui représente l'affichage du formulaire, on le récupère sur la page create.html.twig
            'formArticle' => $form->createView(),
            'editMode' => $article->getId() !== null
        ]);
```

Modifier ensuite le titre h1 et le text du bouton avec des conditions TWIG :

```
<h1 class="text-center mt-3">
        {% if editMode %}
            Modification de l'article
        {% else %}
            Ajouter un article
        {% endif %}
    </h1><hr>
```

```
<button type="submit" class="btn btn-primary">
        {% if editMode %}
            Enregistrer les modifications
        {% else %}
            Ajouter l'article
        {% endif %}
    </button>
```

Nous aurions pu faire encore plus simples pour créer le formulaire à l'aide de la console :

php bin/console make:form

The name of the form class (e.g. VictoriousChefType):

> ArticleType

The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):

> Article

Dans le dossier Form, un fichier ArticleType.php a été crée. Nous pouvons supprimer le champ suivant :

```
->add('createdAt')
```

Mettre donc en commeantaire la fonction suivante dans le BlogController.php

```
$form = $this-
>createFormBuilder($article) // cela va créer un objet qui est lié à notre article
                    // add() fonction permettant de créer des champs dans un formulaire
                    ->add('title')
                    ->add('content')
                    ->add('image')
                    ->getForm(); // permet d'afficher le rendu final
```

Ajouter à la place :

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

Et faire appel à la classe :

```
use App\Form\ArticleType;
```

Imaginons que nous aurions besoin de créer le même formulaire dans plusieurs méthode de mon controller, nous aurions besoin de dupliquer du code, dupliquer veut dires 'problèmes' (temps de maintenance, erreurs, évolutivité).Maintenant dès que nous avons besoin du formulaire, il nous suffit de l'appeler.

VALIDATION DES ENTITES (contrôles des champs)

Pour controller les champs, symfony va se baser sur les entités, par exemple un champs image ne pourra pas recevoir un prénom et symfony va s'en charger. Pour cela rendons nous dans le dossier Entity puis le fichier Article.php

Ajouter la classe des contraintes suivante :

```
use Symfony\Component\Validator\Constraints as Assert;
```

Ajouter la contrainte assert à la propriété title :

```
/**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(min=10, max=255)
     */
    private $title;
```

Faites un test sur google pour l'insertion d'article, dans l'inspecteur enlever le pattern et entrer un titre moins de 10 caractères , une erreur s'affiche !!

Faire la même chose pour « content » mais sans longeur max

```
/**
     * @ORM\Column(type="text")
     * @Assert\Length(min=10)
     */
    private $content;
```

Ajouter aussi la contrainte pour l'image :

```
/**
     * @ORM\Column(type="string", length=255)
     * @Assert\Url()
     */
    private $image;
```

Pour personnaliser les messages d'erreurs, TWIG possède une fonction :

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

Nous pouvons boucler sur les messages d'erreurs voir doc