Les formulaires avec Symfony 1

ATTENTION: à partir d'ici le cours n'a pas été mis à jour depuis 2024

Jusqu'ici on manipulais nos entités via nos routes, mais dans un cas normal, on utilise des formulaires. Là aussi Symfony sait faire de la magie.

Créer un formulaire

On va pouvoir créer un formulaire capable de irriguer l'objet auquel il est lié. (Remplir automatiquement les champs) Pour cela, appelons de nouveau notre ami le maker :

```
symfony console make:form
```

- Il va alors nous demander de le nommer, la convention veut qu'on le nomme avec le nom de l'entité liée et le mot "Form". (Attention, sur les anciennes versions, c'était "Type") Créons le "**MessageForm**".
- Ensuite il nous demande le nom de l'entité auquel le lier. Pour nous ce sera "**Message**".
- Il nous indique alors avoir créé dans le dossier "**src**", un dossier "**Form**" avec le fichier de notre formulaire.

Dans ce nouveau fichier nous allons trouver :

- Le namespace.
- Les uses.
- La classe et sa classe abstraite lui apportant de nouveaux outils.
- Une méthode buildForm qui va nous permettre de construire notre formulaire.
- Une méthode configureOptions.

Arrêtons nous un instant sur la méthode configureOptions, on verra qu'elle paramètre le formulaire pour être lié à notre entité.

Puis voyons le buildForm, lui appelle un builder et lui demande d'ajouter les différentes propriétés de notre objet "**Message**".

laissons cela de côté un instant et rendons nous dans notre "**MessageController**" dans notre méthode "createMessage" on va retirer le contenu précédent pour mettre à la place :

```
$message = new Message();
$form = $this->createForm(MessageType::class, $message);
return $this->render('message/create.html.twig', [
    'messageForm' => $form
]);
```

- Nous créons un nouveau Message.
- Puis on crée un nouveau formulaire en lui indiquant son type et à quelle entité il est lié.

- Enfin nous appelons une nouvelle vue à laquelle on donne une variable contenant notre formulaire.

Puis nous allons créer ce nouveau fichier twig et lui indiquer :

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

{% block body %}
    {{form(messageForm)}}
{% endblock %}
```

Si on tente de voir notre page, notre formulaire est affiché.

Dans les anciennes versions, une erreur était provoqué du type :

"**Object of class App\Entity\Category could not be converted to string**"

Cela vient du fait que nous ayons des relations entre nos entités, Symfony tente d'afficher nos categories en tant que string. Pour corriger cela on aurait typiquement créé une méthode magique dans notre entité **Category**.

Une méthode magique est une méthode qui sera appelé automatiquement par Symfony quand il en aura besoin.

```
public function __toString()
{
  return $this->nom;
}
```

On indique à Symfony que quand il doit transformer cet objet en string, il faut utiliser le nom de celui ci.

Maintenant symfony comprend directement le problème et le résout en indiquant en paramètre du builder quel champ doit être utilisé pour l'affichage de la catégorie, cela grâce à l'option "**choice_label**".

Par défaut il est indiqué d'afficher l'**id**, mais pour un meilleur confort de l'utilisateur, changeons cela par "**name**";

Il nous manque aussi un bouton de soumission. On pourrait l'ajouter à la main dans notre formulaire, mais laissons plutôt symfony faire. Revenons à notre "**MessageType**" et ajoutons :

```
// En haut de la page :
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
// à la suite de notre builder :
->add("Envoyer", SubmitType::class)
```

La méthode **add** peut en effet prendre un second paramètre permettant de lui préciser le type du champ.

Comme on a pu le voir, par défaut, si on ne précise pas de type, Symfony donne des types à nos champs qui semblent correspondre au type de la propriété.

Designer un formulaire

Mais avant de continuer à faire cela, notre formulaire n'est guère jolie à voir. Nous pourrions passer du temps à faire du CSS, mais je vais plutôt vous présenter une nouvelle option:

on va s'amuser avec un peu avec bootstrap. Allons dans le fichier suivant : "**config/packages/twig.yaml**"

Les fichiers .yaml sont des fichiers de configuration, attention, comme en SASS, l'indentation est extrêmement important, une mauvais indentation peut casser votre configuration.

On va ajouter la ligne suivante sous le paramètre twig :

```
form_themes: ['bootstrap_5_layout.html.twig']
```

Si on actualise maintenant, nous ne verrons aucune différence et pourtant il y en a. Symfony a ajouté des classes et des divs selon un template défini dans le fichier indiqué précédement. Et celles-ci correspondent à des classes bootstrap.

Allons donc ajouter le CDN de bootstrap dans notre "**base.html.twig**". Rien ne vous oblige à utiliser bootstrap avec Symfony, mais pour les cours c'est bien pratique de ne pas passer du temps sur le design.

Maintenant si on regarde à nouveau notre formulaire, il se met en forme automatiquement. On verra plus tard qu'il y a mieux à faire si on ne veut pas utiliser un CDN et plutôt l'intégrer directement au projet.

Maintenant il nous faudrait supprimer certains champs inutiles de notre formulaire qui sont censé se mettre automatiquement.

- createdAt
- editedAt

Pour cela deux solutions, soit si on veut les supprimer que dans un controller précis directement dans celui ci avec la méthode :

```
$form->remove("createdAt");
```

Soit et c'est ce que l'on va faire ici, directement dans le constructeur de formulaire en retirant les "**add()**" correspondant.

C'est bien beau d'afficher notre formulaire d'un coup, mais que faire si je souhaite designer un peu celui ci en affichant un champ à un endroit et un autre ailleurs, voir deux côte à côte. Pour cela on peut aussi choisir d'afficher notre champs petit à petit. Revenons à notre vue et remplaçons ce que nous avons écrit par :

```
{{form_start(messageForm)}}
<div class="row">
```

```
    <div class="col">{{form_row(messageForm.content)}}</div>
    <div class="col">{{form_row(messageForm.category)}}</div>
</div>
{{form_rest(messageForm)}}
```

- Ici on indique à twig où le formulaire commence avec "**form_start**"
- Puis on lui indique d'ajouter les lignes une à une avec "**form_row**"
- Puis on lui dit d'afficher le reste du formulaire avec "**form_rest**"

Traiter un formulaire

On s'est bien amusé avec le design de notre formulaire mais maintenant il faudrait que lors de son envoi il soit traité. Pour cela rendons nous dans notre contrôleur et ajoutons :

```
// Dans la liste des classes à utiliser
use Symfony\Component\HttpFoundation\Request;
// On ajoute l'objet Request
public function create(ManagerRegistry $doc, Request $request): Response
// puis dans la méthode après avoir créé le formulaire :
$form->handleRequest($request);
if($form->isSubmitted())
{
  dd($message);
}
```

- "**handleRequest**" indique de prendre les informations fourni par l'objet Request qui je vous le rappel contient toute les superglobals dont **$_POST** et **$_GET** et les fournir à notre formulaire.
- "**isSubmitted**" permet de vérifier si le formulaire a été soumit (envoyé).

Si on regarde le dump de notre nouvelle objet "Message", on verra qu'il a été irrigué, les informations qui ont été rentré dans notre formulaire sont placé dans notre objet.

Remplaçons maintenant ce dump par :

```
$em = $doc->getManager();
$em->persist($message);
$em->flush();

$this->addFlash("success", "Un nouveau message a bien été ajouté");
return $this->redirectToRoute("app_message_read");
```

Rien de nouveau ici, je récupère mon manager et je sauvegarde mon objet qui a été remplit par le formulaire. J'ajoute un message flash et je redirige ailleurs.

Et pour gérer l'édition, ce n'est pas plus compliqué retournons sur notre page d'édition et modifions la:

```
#[Route('/update/{id<\d+>}', name: 'updateVille')]
public function update(Ville $ville=null, ManagerRegistry $doc, Request $request):
Response
{
 if(!$message)
  {
    $this->addFlash("danger", "Aucun message sélectionné.");
    return $this->redirectToRoute("app_message_read");
  }
  $form = $this->createForm(MessageType::class, $message);
  $form->handleRequest($request);

  if($form->isSubmitted())
  {
    $em = $doc->getManager();
    $em->persist($message);
    $em->flush();

    $this->addFlash("success", "Message édité.");
    return $this->redirectToRoute("app_message_read");
  }
  return $this->render('message/create.html.twig', [
    'messageForm' => $form
  ]);
}
```

Si on y regarde de plus près, c'est quasiment un copié collé du create. On a retiré les paramètres de la route, et changé un peu les messages, mais c'est tout.

Au lieu de créer un nouveau message, on en récupère un existant, et on remarquera même que par ce fait les champs sont déjà pré-remplis.

Créons un lien pour aller sur notre page à côté de notre lien de suppression.

```
<a href="{{ path("app_message_update", {id: mess.id})}}">modifier</a>
```

Changer les types des champs du formulaire

On a vu avec le bouton submit que les champs de nos formulaire peuvent prendre en second argument des types. Pour cela on pourra voir la documentation de Symfony :

https://symfony.com/doc/current/reference/forms/types.html

Ces types peuvent aussi prendre des options supplémentaires, changeons par exemple notre Category :

```
->add('category', EntityType::class, [
  'class' => Category::class,
  'choice_label' => 'name',
```

```
    "expanded"=>false,
    "multiple"=>false
  ])
```

Ici on ne verra pas de différence car ce sont les paramètres que Symfony avait placé par défaut, mais amusez vous à changer les booleans et vous verrez que vous aurez des listes ouverte, des radio ou encore des checkbox.

Attention que cela corresponde toujours aux relations que vous avez choisi. Ici mettre les radios ne poserait pas de problème. mais la liste ouverte ou les checkbox provoquerait des erreurs. Car on a bien indiqué que nos messages ne peuvent avoir qu'une seule catégorie.

On pourrait aussi choisir de n'afficher que certaines de nos categories ou alors les afficher par ordre alphabétique en changeant la requête sql par défaut :

```
// Dans les uses
use App\Repository\CategoryRepository;
// Dans les options de notre ajout des categories
'query_builder' => function (CategoryRepository $repo) {
  return $repo->createQueryBuilder('c')
              ->orderBy('c.nanme', 'ASC');
},
```

Les options sont nombreuses et varies selon les types d'input utilisés, mais certaines sont en commun à tous les types comme :

- label qui permet de changer le label du champ.
- required qui permet d'indiquer si le champ est requis.
- mapped qui permet d'indiquer si le champ est lié à l'objet. (Par défaut ils sont tous mapped sauf le submit)
- attr pour ajouter des classes ou autre attribut.
- Et bien d'autres.

On va d'ailleurs en tester certains maintenant.