Les formulaires avec Symfony

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 "Type".
  Créons le "**MessageType**".
- 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 la vue de 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.

Upload de fichier

Nous savons que lorsqu'on upload un fichier, on se contente d'enregistrer son nom en BDD mais pas le fichier complet qui lui est juste rangé dans un dossier. Donc comment gérer cela? Premièrement ajoutons à nos "**messages**" la propriété suivante :

"**image string 255 null**"

N'oubliez pas de faire la migration.

Ajoutons maintenant le champ suivant dans le formulaire :

```
// Dans les uses :
use Symfony\Component\Validator\Constraints\File;
use Symfony\Component\Form\Extension\Core\Type\FileType;
// dans le builder :
->add("imageFile", FileType::class, [
  "label"=>"Image pour accompagner votre post :",
  "mapped"=>false,
  "required"=>false,
  "constraints"=>[
    new File([
      "maxSize" => "1024k",
      "mimeTypes"=>[
        "image/jpeg",
        "image/png",
        "image/gif"
      ],
      "mimeTypesMessage"=>"Seule les images jpg, png ou gif sont acceptés."
    ])
  ]
])
```

On a indiqué que ce champ n'est pas mappé car ce n'est pas le fichier qu'on veut en BDD mais seulement son nom, ce champ n'ira donc pas remplir automatiquement la propriété "**image**" que l'on vient de créer. Ici constraints permet d'ajouter des contraintes, on pourrait le faire sur n'importe quel champ, mais on verra plus tard que pour les champs mappés, il est plus pratique de le faire directement dans l'entité.

Maintenant il nous reste à traiter l'upload, mais le faire directement dans notre traitement de formulaire serait peu pratique et nous obligerait à le répéter dans l'upload, c'est pour cela que l'on va créer un service : "**src/Service/Uploader.php**"

```
namespace App\Service;

use Symfony\Component\HttpFoundation\File\Exception\FileException;
use Symfony\Component\String\Slugger\SluggerInterface;
use Symfony\Component\HttpFoundation\File\UploadedFile;

class Uploader
{
    public function __construct(private SluggerInterface $slugger)
    {}
    /**
     * Upload un fichier
     *
     * @param UploadedFile $file fichier à téléverser
     * @param string $folder Chemin vers le dossier où téléverser
     * @return string|null
     */
    public function uploadFile(UploadedFile $file, string $folder):string|null
    {
        $original = pathinfo($file->getClientOriginalName(), PATHINFO_FILENAME);
```

```
            $safe = $this->slugger->slug($original);
            $new = $safe ."-".uniqid().".".$file->guessExtension();
            try{
                $file->move($folder, $new);
            }catch(FileException $e){
                return null;
            }
            return $new;
        }
    }
```

On avait déjà vu l'upload de fichier en PHP classique, voici la version symfony. la logique reste cela dit la même, réécrire le nom d'origine pour éviter les doublons et tenter de le déplacer hors de la zone temporaire.

On notera cela dit une nouveauté, jusqu'ici j'appelais les outils dont j'avais besoin dans les paramètres de la méthode où j'en avais besoin, mais il est aussi possible de le faire dans la méthode "**__construct**", Dans ce cas elle sera accessible avec "**$this**" dans toute les méthodes.

Ensuite nous allons ouvrir le fichier suivant : "**config/services.yaml**"
Et nous allons y ajouter la ligne suivante dans parameters :

```
message_directory: "%kernel.project_dir%/public/uploads/messages"
```

Dans ce fichier on peut mettre des paramètres qui seront accessible partout dans notre application. Nous permettant ainsi de n'avoir qu'un seul endroit à modifier si on souhaite le changer.
Ici ce sera la route pour téléverser nos images de message.
Je peux créer les dossiers moi même mais symfony est normalement capable de les créer si ils n'existent pas.

Enfin rendons nous dans le "**MessageController**" et ajoutons un construct :

```
public function __construct(private Uploader $uploader)
{}
```

Sur un projet plus propre j'aurais pu aussi mettre dans le construct l'entity manager que j'appelle dans presque toute les méthodes plutôt que de le répéter à chaque fois.

Enfin ajoutons quand le formulaire est soumis et avant d'enregistrer notre message en BDD les lignes suivantes:

```
$image = $form->get("imageFile")->getData();
if($image)
{
    $dir = $this->getParameter("message_directory");
    $message->setImage($this->uploader->uploadFile($image, $dir));
}
```

- On récupère d'abord les données lié au champ "imageFile".
- Puis si il y en a bien, on récupère le chemin du dossier d'upload.
- Enfin on rempli la propriété Photo de notre ville avec ce que retourne notre service uploader.

On pourrait d'ailleurs copier le même code dans notre "**update**".

Si on souhaite afficher nos images on pourra simplement ajouter à notre liste de message dans le la vue du twig :

```
{% if mess.image %}
  <img src="{{ asset('uploads/messages/' ~ mess.image)}}" alt="Image accompagnant
le message">
{% endif %}
```

Au niveau de notre **update** il faudra penser à supprimer l'ancienne image si elle est présente:

```
$image = $form->get("imageFile")->getData();
if($image)
{
    $oldImage = $message->getImage();
    $dir = $this->getParameter("message_directory");
    $message->setImage($this->uploader->uploadFile($image, $dir));
    if($oldImage)
    {
        unlink($dir."/".$oldImage);
    }
}
```

De même au niveau du delete

```
$dir = $this->getParameter("message_directory");
if($message->getImage() && file_exists($dir."/".$message->getImage() ))
{
  unlink($dir."/".$message->getImage());
}
```

Validation du formulaire

Pour valider les formulaires, nous allons ajouter de nouveaux attribut à nos propriétés dans nos entités.

Il existe tout un tas de contraintes que vous retrouverez dans la documentation :

https://symfony.com/doc/current/validation.html

Mais nous allons en voir certaines ici.

Allons dans notre entité "**Message**"

Ajoutons les lignes suivantes :

```
use Symfony\Component\Validator\Constraints as Assert;
// au dessus de la propriété $content :
#[Assert\NotBlank(message:"Veuillez renseigner ce champ")]
#[Assert\Length(min:3, max:500)]
// Puis dans les pages utilisant notre formulaire, remplaçons :
if($form->isSubmitted())
// par :
if($form->isSubmitted() && $form->isValid())
```

Je vérifie si les champs ne sont pas vide et si ils respectent les conditions imposées.
Puis lorsque je vérifie si mon formulaire est soumis, je vérifie aussi si il est valide.

Si on souhaite désactiver les vérifications HTML pour tester nos vérifications PHP, ajoutons au twig :

```
form_start(form, {'attr': {'novalidate': 'novalidate'}})
```

Résumons tout cela make:crud

On a bien travaillé pour créer ce **CRUD** mais maintenant il nous faut faire **CRUD** de nos categories. Pour cela on est repartie pour utiliser la console mais non pas pour un controller ou un formulaire :

```
symfony console make:crud
```

Effectivement, tout ce qu'on a fait jusqu'ici était là pour comprendre le fonctionnement de Symfony, mais si je veux un **CRUD**, je le demande à Symfony.

Il nous demandera alors quelle entité y est lié, indiquons lui notre "**Category**".
Il nous demande alors le nom du controller, ici celui par défaut nous va.
Nouvellement il prévoit la création de test automatique pour le controller, mais on va s'en passer.

Il nous a donc créé :

- Un Controller.
- Un Formulaire.
- Un template _delete_form.
- Un template _form.
- Un template edit.
- Un template index.
- Un template new.
- Un template show.

Évidement ce sont des templates et un controller généraliste, il nous restera toute les validations, les cas spécifiques, et tout cela à gérer.

Prenons l'exemple du formulaire, on peut lui retirer le **createdAt** et le **editedAt**.

Ou même ajouter les validations suivant à la propriété "**$name**" de Category :

```
use Symfony\Component\Validator\Constraints as Assert;
// Avec le use au début de la page.
#[Assert\Regex('/^[a-zA-Z]+$/')]
#[Assert\NotBlank(message:"Veuillez renseigner ce champ")]
#[Assert\Length(min:3, max:50)]
```

Mais cela nous résume une grande partie du travail.

Envoyer des mails

On va voir comment gérer les mails avec Symfony, si vous utilisez un service externe telle que Amazon SES ou autre, il vous faudra installer un composant différent que vous retrouverez dans la documentation :

https://symfony.com/doc/current/mailer.html

Le premier point va être de paramétrer le DSN dans le fichier "**.env**" ou "**.env.local**".

```
MAILER_DSN=...
```

Nous aurons sûrement plusieurs routes qui utiliserons l'envoi d'email, donc regroupons cela dans un service.

```
namespace App\Service;

use Symfony\Component\Mime\Email;
use Symfony\Component\Mailer\MailerInterface;

class Mailer
{
    public function __construct(Private MailerInterface $mailer)
    {}
    public function sendEmail(
        string $from = "noreply@cours.fr",
        string $to = "nolwenn@cours.fr",
        string $subject = "Message Automatique",
        string $content = "Ne pas répondre à ce message."
    ):void
    {
        $email = (new Email())
            ->from($from)
            ->to($to)
            ->subject($subject)
            ->html($content);
        $this->mailer->send($email);
```

```
    }
}
```

Dans ce service nous appelons le **MailerInterface** qui va nous permettre d'utiliser les fonctionnalités liés aux mails.

Puis nous remplissons simplement notre email avant de l'envoyer.

Pour le tester allons ajouter à notre création de message un email (même si cela aura plus de sens pour l'inscription par exemple):

```
use App\Service\Mailer;
//...
public function create(ManagerRegistry $doc, Request $request, Mailer $mailer)
//...
$mailer->sendEmail();
```

Mais si on tente de créer un nouveau message, aucun mail n'arrive... pourquoi? Simplement car la version webapp de Symfony inclus un paquet qui permet d'envoyer les mails de façon asynchrone. C'est à dire que dans un cas classique, si vous envoyez un mail lourd, alors votre serveur mettra du temps à vous répondre, cela pour qu'il puisse envoyer l'email.

Avec cet envoi asynchrone, Symfony répond tout de suite à l'utilisateur et met son email en file d'attente pour qu'il soit traité en arrière tâche par le serveur.
Pour activer ce service, il faudra faire tourner :

```
symfony console messenger:consume async
# Ou -vv si vous voulez voir le détail des envois.
symfony console messenger:consume async -vv
```

Les messages que l'on a voulu envoyé sont d'ailleurs tant que l'on n'a pas activé cette commande visible dans la BDD.