

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 "**VilleType**".
- Ensuite il nous demande le nom de l'entité auquel le lier. Pour nous ce sera "**Ville**".
- 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 "**Ville**".

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

```
$ville = new Ville();  
$form = $this->createForm(VilleType::class, $ville);  
return $this->render('ville/create.html.twig', [  
    'villeForm' => $form->createView()  
]);
```

- Nous créons une nouvelle Ville.
- 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 :

```
{{form(villeForm)}}
```

Si on tente de voir notre page, il y a de grandes chances qu'on ait l'erreur suivante :

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

Cela vient du fait que nous ayons des relations entre nos entités, Symfony tente d'afficher nos départements en tant que string. Pour corriger cela on va créer une méthode magique dans notre entité **departement**.

Une méthode magique est une méthode qui sera appelée 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 nous pouvons voir notre formulaire. (qui a des champs en trop, mais on corrigera cela plus tard)

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 **"VilleType"** et ajoutons :

```
->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, 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 importante, une mauvaise indentation peut casser votre configuration.

On va ajouter la ligne suivante sous le paramètre twig :

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

puis allons 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 prend par défaut des classes bootstrap pour se mettre en forme automatiquement. Bien sûr on pourrait aussi le personnaliser nous même.

Maintenant il nous faudrait supprimer certains champs inutiles de notre formulaire.

- createdAt
- editedAt
- chefLieu

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(villeForm)}}
<div class="row">
    <div class="col">{{form_row(villeForm.nom)}}</div>
    <div class="col">{{form_row(villeForm.population)}}</div>
</div>
{{form_rest(villeForm)}}
```

- Ici on indique à twig où il 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 :

```
// 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($form->isSubmitted())
{
    dump($ville);
}
```

- **"handleRequest"** indique de prendre les informations fournies par l'objet Request qui je vous le rappelle contient toutes les superglobales dont **\$_POST** et **\$_GET** et les fournira à notre formulaire.
- **"isSubmitted"** permet de vérifier si le formulaire a été soumis (envoyé).

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

Remplaçons maintenant ce dump par :

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

$this->addFlash("success", "Une nouvelle ville a bien été ajoutée");
return $this->redirectToRoute("readVillePage");
```

Rien de nouveau ici, je récupère mon manager et je sauvegarde mon objet qui a été rempli 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<[0-9]+}>', name: 'updateVille')]
public function update(Ville $ville=null, ManagerRegistry $doc, Request $request):
Response
{
    if(!$ville)
    {
        $this->addFlash("danger", "Aucune ville sélectionnée.");
        return $this->redirectToRoute("readVillePage");
    }
    $form = $this->createForm(VilleType::class, $ville);
    $form->handleRequest($request);
    if($form->isSubmitted())
    {
        $em = $doc->getManager();
        $em->persist($ville);
        $em->flush();

        $this->addFlash("success", "Ville édité.");
        return $this->redirectToRoute("readVillePage");
    }
    return $this->render('ville/create.html.twig', [
        'villeForm' => $form->createView()
```

```
]);  
}
```

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 une nouvelle ville, on en récupère une existante, et on remarquera même que par ce fait les champs sont déjà pré-remplis.

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 departement :

```
->add('departement', EntityType::class, [  
    "class"=>Departement::class,  
    "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.

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

```
'query_builder' => function (DepartementRepository $repo) {  
    return $repo->createQueryBuilder('d')  
        ->orderBy('d.nom', 'ASC');  
},
```

Les options sont nombreuses et varies de type en type, 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. (utile pour ajouter des champs supplémentaire)
- 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 **"villes"** la propriété suivante :

"photo 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("photoFile", FileType::class, [
    "label"=>"Photo de cette magnifique ville :",
    "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. 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)
```

```

    {}
    public function uploadFile(UploadedFile $file, string $folder):string|false
    {
        $original = pathinfo($file->getClientOriginalName(), PATHINFO_FILENAME);
        $safe = $this->slugger->slug($original);
        $new = $safe . "-" . uniqid() . "." . $file->guessExtension();
        try{
            $file->move($folder, $new);
        }catch(FileException $e){
            return false;
        }
        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 :

```
ville_directory: "%kernel.project_dir%/public/uploads/villes"
```

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 photos de ville.

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 "**VilleController**" 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 les lignes suivantes:

```

$photo = $form->get("photoFile")->getData();
if($photo)
{
    $dir = $this->getParameter("ville_directory");
}

```

```
$ville->setPhoto($this->uploader->uploadFile($photo, $dir));  
}
```

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

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

TODO: Faire l'affichage twig et supprimer l'ancienne lors de l'upload.

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 "**Ville**"

Ajoutons les lignes suivantes :

```
use Symfony\Component\Validator\Constraints as Assert;  
// au dessus du nom :  
#[Assert\NotBlank(message:"Veuillez renseigner ce champ")]  
#[Assert\Length(min:3, max:50)]  
// au dessus de population :  
#[Assert\NotBlank(message:"Veuillez renseigner ce champ")]  
#[Assert\Regex('/^\d+$/')]  
// et enfin dans les pages utilisant notre formulaire, remplaçons :  
if($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 départements. 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 "**Departement**".

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**.

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 gmail, 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.

"Voir src/Service/Mailer.php"

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 ville un email :

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

Mais si on tente de créer une nouvelle ville, 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.