Les formulaires avec Symfony 2

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;
```

```php
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 :

```yaml
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 :

```php
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.