Formulaires et base de données

Envoie de données dans la base

Vérifier que toutes les relations entre vos tables soient bien déclarées. Nous prenons ici comme exemple la table `products` mais peut-être qu'il sera nécessaire de faire ces opérations sur les autres tables de votre base au préalable.

Faisons un insert dans la table `products`.

Concernant cette entité, nous avons déjà un contrôleur permettant l'affichage de la liste des produits

Pour cela, nous allons utiliser la méthode que nous avons créé pour afficher le formulaire d'ajout.

Pour rappel:

```
/**
     * @Route("/new", name="products_new", methods={"POST")
     * @param Request $request
     * @return Response
     */
    public function new(Request $request): Response

    {
        $product = new Products();
        $form = $this->createForm(ProductsType::class, $product);
        $form->handleRequest($request);

          return $this->render('products/new.html.twig', [
                    'product' => $product,
                    'form' => $form->createView(),
                ]);
    }
```

Pour valider l'ajout d'une donnée dans la base, il faut vérifier que le formulaire est soumis et valide :

```
 if ($form->isSubmitted() && $form->isValid()) {

}
```

Si le formulaire est soumis et valide, alors nous allons utiliser l'objet EntityManager de Doctrine. Il nous permet d'envoyer et d'aller chercher des objets dans la base de données :

```
$entityManager = $this->getDoctrine()->getManager();
```

Ensuite nous allons persister notre entité, c'est-à-dire que nous allons la préparer à la sauvegarde des données saisies :

```
 $entityManager->persist($product);
```

Enfin, pour envoyer les données dans la base, nous utilisons la méthode `flush()` :

```
 $entityManager->flush();
```

Et nous redirigeons vers la liste des produits :

```
return $this->redirectToRoute('products_index');
```

Résumé de la méthode d'ajout :

```
/**
     * @Route("/new", name="products_new", methods={"POST")
     * @param Request $request
     * @return Response
     */
    public function new(Request $request): Response

    {
        $product = new Products();
        $form = $this->createForm(ProductsType::class, $product);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($product);
            $entityManager->flush();

            return $this->redirectToRoute('products_index');
        }

        return $this->render('products/new.html.twig', [
            'product' => $product,
            'form' => $form->createView(),
        ]);
    }
```

Il serait intéressant d'afficher un message de confirmation lors de la validation. Nous allons utiliser pour cela la méthode `addflash($type, $message)`. Cette méthode utilise 2 paramètres :

- Le type de message à afficher (`success`, `warning`, ...)
- Le message à afficher

Pour un affichage efficace, nous devons l'utiliser avant la redirection :

```
$this->addFlash(
                    'success',
                    'Produit ajouté avec succès !!'
                );
```

Il nous reste plus qu'à gérer son affichage. Pour cela, rendons-nous dans le fichier `base.html.twig`.

Nous allons en profiter pour styliser ce message en utilisant les alertes de Bootstrap. Insérez ce code avant les balises `{# block body #}` :

```
{% for message in app.flashes('success') %}
    <div class="alert alert-success">
        {{ message }}
    </div>
{% endfor %}
```

Modification

Pour la modification d'un produit, nous partirons du principe que nous pouvons modifier toutes les informations saisies lors de l'ajout. Cela évitera de créer un nouveau formulaire. Si toutefois on devait n'autoriser que certains champs pour la modification, il faudrait créer un autre formulaire comme vu précédemment. Pour effectuer une modification dans la base de données (`UPDATE`), les choses seront assez similaires. Dans un premier nous devons passer l'id du produit dans le lien permettant d'accéder au formulaire de modification :

```
<a href="{{ path('products_edit', {'id': product.id}) }}">edit</a>
```

- `path()` permet d'indiquer le nom de la route vers laquelle nous voulons nous diriger
- `products_edit` est le nom de la route
- `'id'` : nom du paramètre
- `product.id` : valeur du paramètre

`product.id` ici est généré lors de l'affichage du détail d'un produit et / ou lors de l'affichage de la liste de produit.

Créons ensuite une méthode permettant la modification, avec sa route, la création du formulaire et son rendu :

```
/**
     * @Route("/{id}/edit", name="products_edit", methods={"GET","POST"})
     * @param Request $request
     * @param Products $product
     * @return Response
     */
    public function edit(Request $request, Products $product): Response
    {
        $form = $this->createForm(ProductsType::class, $product);
        $form->handleRequest($request);

        return $this->render('products/edit.html.twig', [
            'product' => $product,
            'form' => $form->createView(),
        ]);
    }
```

Ici, nous passons en paramètre de notre méthode l'objet `Products`. On dit que nous injectons l'objet `Products` à la méthode `edit`.

Ce fonctionnement a pour avantage de récupérer directement le produit dont nous avons besoin. Il n'est donc pas nécessaire de récupérer notre produit dans la méthode, Symfony le fais pour nous.

Tout le reste est similaire à ce que nous avons vu jusque-là.

Il nous faut ensuite soumettre le formulaire et vérifier les données saisies. Si vous utilisez le même formulaire que pour l'ajout, tout est déjà en place pour les règles de validations. Si vous utiliser un autre formulaire, il faut bien sûr définir ses règles de validations dans le formType.

```
 if ($form->isSubmitted() && $form->isValid()) {
            $this->getDoctrine()->getManager()->flush();

            return $this->redirectToRoute('products_index');
        }
```

Ici pas besoin de persister notre entité, il suffit juste de faire appel à l'entityManager de Doctrine et de faire un `flush()`. S'en suit une redirection vers la page de votre choix, ici la liste des produits.

Vous pouvez aussi ajouter un message de confirmation pour indiquer à l'utilisateur que la modification a bien été effectué en utilisant `addFlash()`.

Suppression

Pour la suppression la structure reste la même, mais nous utiliserons la méthode `remove()` suivie de la méthode `flush()`.

```
$entityManager->remove($product);
$entityManager->flush();
```

En suivant ce qui a été vu pour l'ajout et la modification, construisez une méthode permettant la suppression d'un produit.

Upload

Ajoutons une photo sur les produits présents dans la base.

Commençons par ajouter un nouveau champ dans la base de données pour y stocker le nom de notre fichier, soit manuellement, soit en utilisant la commande `php bin/console make:entity Products`. Nous stockerons dans ce champ une chaine de caractères de 255 caractères. Appelons ce champ `picture`. Nous ne voulons pas qu'il soit obligatoire dans la base de données, donc nous indiquerons qu'il sera null par défaut.

Il faut ajouter ensuite ce champ dans le constructeur de notre formulaire :

```
->add('picture2', FileType::class, [
                'label' => 'Photo de profil',
                //unmapped => fichier non associé à aucune propriété d'enti
té, validation impossible avec les annotations
                'mapped' => false,
                // pour éviter de recharger la photo lors de l'édition du p
rofil
                'required' => false,
                'constraints' => [
                    new Image([
                        'maxSize' => '2000k',

                        'mimeTypesMessage' => 'Veuillez insérer une photo a
u format jpg, jpeg ou png'
                    ])
                ]
            ])
```

- `FileType::class` : permet d'indiquer que nous uploader un fichier
- `'mapped' => false` : permet de dire que ce champ n'est pas lié à la base de données

- `'required' => false` : nous ne voulons pas que l'upload d'un fichier soit obligatoire
- `'constraints' => [ ]` : définition des contraintes de validation pour ce champ

"Euh ... Pas lié à la base de données ?? et `picture2`, ce n'est pas le nom que nous avons donné à notre champ dans l'entité, non ???"

C'est normal ! Pour l'upload d'un fichier, son stockage dans la base et sur le serveur, nous avons besoin de plusieurs choses :

- le nom du fichier (qui est saisi dans l'input, via l'uploader généré par le navigateur lorsque nous cliquons sur le bouton pour insérer un fichier)
- les informations relatives au fichier (type, taille, chemin d'origine, chemin temporaire, erreur d'upload, etc.)

La propriété définie dans l'entité correspond au nom du fichier, ce que nous allons stocker dans la base.

Le champ que nous construisons ici permet de récupérer les informations du fichier. S'il était relié à la base, nous serions amenés à stocker un tableau d'informations dans la base, 'Symfony' ne serait pas content et nous afficherais une belle erreur !!

Nous avons donc une propriété pour récupérer le nom du fichier, et un champ pour récupérer les informations du fichier.

Nous pouvons maintenant ajouter ce champ sur notre template (`products/edit.html.twig`) :

```
    {{ form_row(form.picture2) }}
```

`picture2` est le champ créé dans le formType, il doit être ajouté dans le template.

Nous pouvons maintenant contrôler notre upload dans `Controller/ProductsController.php`.

Pour l'exemple, le nom du fichier sur le serveur sera l'id du produit suivi de son extension.

Nous commençons par récupérer l'id du produit, puis ce qui a été saisi dans le formulaire :

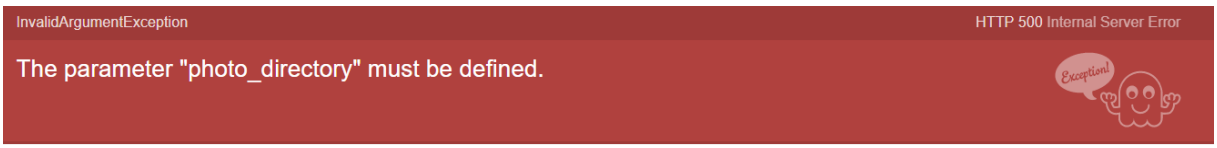
```
/**
 * @Route("/{id}/edit", name="products_edit", methods={"GET","POST"})
 * @param Request $request
 * @param Products $product
 * @return Response
 */
public function edit(Request $request, Products $product): Response
{
    // récupération de l'id du produit
    $idProduct = $product->getId();
    $form = $this->createForm(ProductsType::class, $product);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        // récupération de la saisi sur l'upload
        $pictureFile = $form['picture2']->getData();
        $this->getDoctrine()->getManager()->flush();
        return $this->redirectToRoute('products_index');
```

```
        }
        return $this->render('products/edit.html.twig', [
            'product' => $product,
            'form' => $form->createView(),
            ]);
    }
```

S'il y a une photo, nous allons commencer nos manipulations sur le fichier afin de le sauvegarder sur le serveur :

```
// vérification s'il y a un upload photo
if ($pictureFile) {
    // renommage du fichier
    // nom du fichier + extension
    $newPicture = $idProduct . '.' . $pictureFile->guessExtension();
// assignation de la valeur à la propriété picture à l'aide du setter
    $product->setPicture($newPicture);
    try {
    // déplacement du fichier vers le dossier de destination sur le serveur
        $pictureFile->move(
            $this->getParameter('photo_directory'),
            $newPicture
        );
    } catch (FileException $e) {
      // gestion de l'erreur si le déplacement ne s'est pas effectué
    }
}
```

J'ai une erreur ça ne marche pas !!!



Oui, c'est normal, nous n'avons pas spécifié où on va stocker notre fichier. Enfin à moitié... La méthode `move()` permet le déplacement d'un fichier vers un emplacement spécifique.

`$this->getParameter('photo_directory')` représente cet emplacement. Il ne nous reste plus qu'à le définir. Pour cela, rendons-nous dans le fichier `config/service.yaml` :

```
parameters:
    photo_directory: '%kernel.project_dir%/public/images/produit'
```

Veillez à respecter l'indentation !!!

Vous pouvez maintenant appliquer ce tuto sur tous les formulaires où un upload est nécessaire, adaptez-le en fonction de vos besoins.