Les formulaires

Construction du formulaire

Côté back-end

Pour générer les formulaires, Symfony utilise des classes permettant de les 'construire'.

Ouvrons le fichier créé (`ProductsType.php`) et observons son contenu. Vous devriez avoir quelque chose comme ceci :

```
<?php

namespace App\Form;

use App\Entity\Products;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ProductsType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            ->add('ProductName')
            ->add('SupplierId')
            ->add('CategoryId')
            ->add('QuantityPerUnit')
            ->add('UnitPrice')
            ->add('UnitsInStock')
            ->add('UnitsOnOrder')
            ->add('RedorderLevel')
            ->add('Discontinued')
            ->add('SupplierID')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Products::class,
        ]);
    }
}
```

Sa structure est la même que toutes les autres classe sous Symfony. On retrouve 2 méthodes à l'intérieur :

- `buildForm()` : on y trouvera les différents qui composeront notre formulaire
- `configureOptions()` : c'est ici que nous pourrons établir une liste d'option sur le rendu ou le contrôle du formulaire.

Dans notre cas, nous avons un champ par propriété se trouvant dans la table `products`. Il est possible d'avoir des champs qui ne soient en liaison avec une table (notamment dans le cas d'upload d'images).

Par défaut, Symfony prend le type de champ qui se trouve dans l'entité. C'est-à-dire, si le champ dans l'entité est de type integer, Symfony va générer un `input` de type `number`.

Allons dans notre contrôleur pour générer une vue. Nous allons y créer une vue permettant d'afficher notre formulaire. Commençons par créer une méthode :

```
 public function new(Request $request): Response

    {

    }
```

On va demander à Symfony de "créer" notre formulaire, et que des données vont y être lues (doc) :

```
 public function new(Request $request): Response

    {
// création du formulaire
  $form = $this->createForm(ProductsType::class, $product);
// lecture du formulaire
 $form->handleRequest($request);
    }
```

Enfin, il ne reste plus qu'à faire un rendu de notre vue, en incluant notre template :

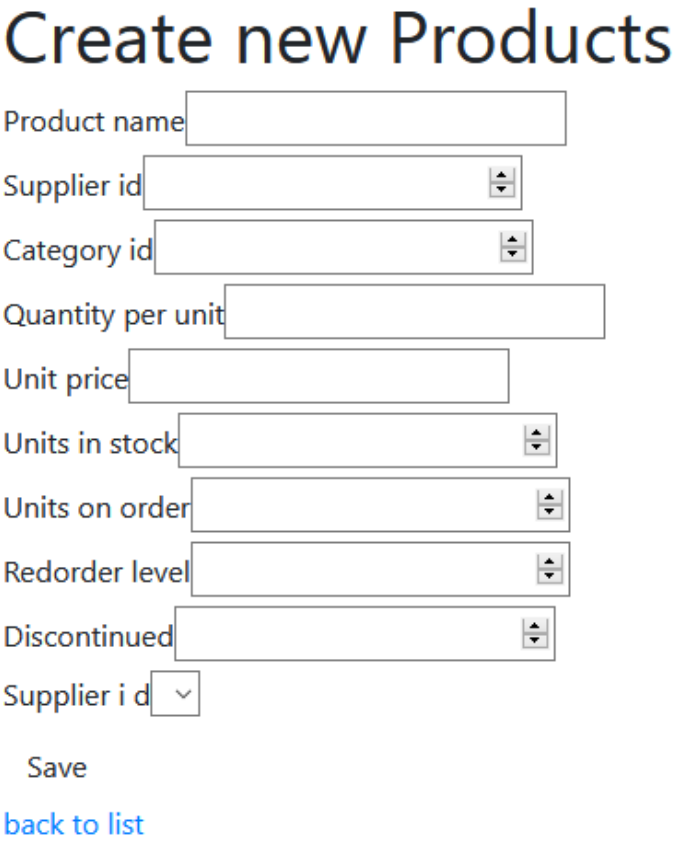
```
 public function new(Request $request): Response

    {
// création du formulaire
  $form = $this->createForm(ProductsType::class, $product);
// lecture du formulaire
 $form->handleRequest($request);
  return $this->render('products/new.html.twig', [
            'product' => $product,
            'form' => $form->createView(),
    }
```

Définissons une route pour cette méthode et prfitons-e pour mettre un peu de doc sur cette méthode :

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

En appelant notre route dans l'url, nous obtenons :



Nous allons rendre ce formulaire un peu plus propre et plus proche du rendu que nous devrions avoir...

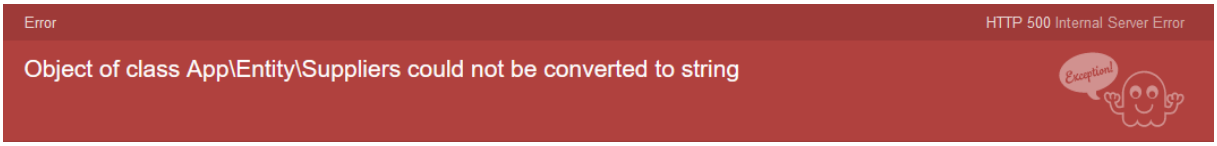
Avant d'aller plus loin, vérifiez que les tables `suppliers` et `products` soient alimentées.

Pour avoir un rendu propre grâce à Bootstrap, nous allons nous rendre dans le fichier `config/packages/twig.yaml`, et ajouter la ligne suivante :

```
form_themes: ['bootstrap_4_layout.html.twig']
```

Attention : gardez l'indentation afin d'éviter d'éventuelles erreurs !!!

Vous pourriez vous retrouver face à cette erreur :

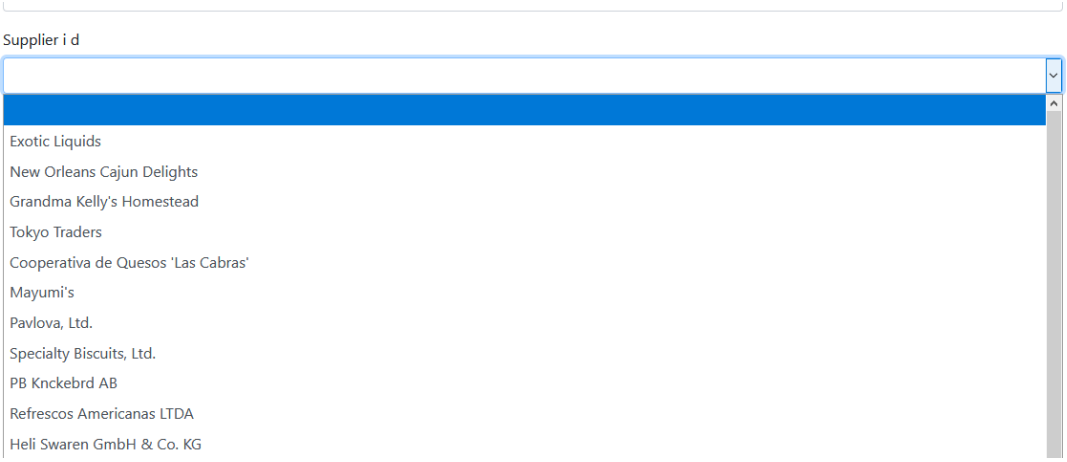


C'est normal : Les tables `products` et `suppliers` étant liées, Symfony récupère automatiquement les données de la table `suppliers` pour les afficher dans une liste déroulante (voir plus haut, Symfony génère le formulaire en fonction de la configuraiton de l'entité). Sans actions de notre part, Symfony récupère un objet, au lieu de récupérer une chaine de caractères.

Pour cela, nous devons nous rendre dans l'entité `suppliers`, et nous allons y ajouter les lignes suivantes :

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

Ceci nous permettra de récupérer directement les noms des fournisseurs et de les afficher dans une liste déroulante.



Plutôt que d'être obligé de faire une requête supplémentaire pour afficher les fournisseurs comme nous le faisions jusque-là, en configurant correctement les entités, Symfony fait tout, tout seul.

Passons maintenant à la personnalisation des `inputs`.

A noter qu'ils existent plusieurs façons de configurer les `inputs` et les `labels`. Nous pouvons le faire sur > le template, mais pour garder un template propre et clair, nous le ferons sur le formType.

Dans le fichier `ProductsType.php` chaque `->add('nom_input')` représente un input de notre formulaire. Pour le personnaliser, nous allons passer en paramètre de cette méthode un tableau d'option qui nous permettra de changer l'intitulé du label, ajouter un placeholder, ajouter des contraintes de validations, et bien d'autres choses encores.

Prenons en exemple l'input `ProductName` :

```
->add('ProductName')
```

Ajoutons-lui un label plus propre et plus clair, et un placeholder :

```
->add('ProductName', TextType::class, [
                'label' => 'Nom du produit',
                'attr' => [
                    'placeholder' => 'Produit',
                ],
            ])
```

Occupons-nous maintenant des contraintes de validations :

```
->add('ProductName', TextType::class, [
                'label' => 'Nom du produit',
                'attr' => [
                    'placeholder' => 'Produit',
                ],
                'constraints' => [
                    new Regex([
                        'pattern' => '/^[A-Za-zéèàçâêûîôäëüïö\_\-\s]+$/',
                        'message' => 'Caratère(s) non valide(s)'
                    ]),
                'help' => 'Vous devez rentrer le nom du produit ici',
                ]
            ])
```

Vous remarquerez que pour que les contraintes de validation fonctionnent, vous devez effectuer un import de classe sur le `ProductsType` :

```
use Symfony\Component\Validator\Constraints\Regex;
```

Il existe d'autres options pour configurer vos champs de saisie, et d'autres contraintes de validation !!! Par exemple, on pourrait afficher un message d'aide à l'utilisateur pour lui donner des indications sur les données qui doivent être saisies :

```
->add('ProductName', TextType::class, [
                'label' => 'Nom du produit',
                'help' => 'Indiquez ici le nom complet du produit',
                'attr' => [
                    'placeholder' => 'Produit',
                ],
                'constraints' => [
                    new Regex([
                        'pattern' => '/^[A-Za-zéèàçâêûîôäëüïö\_\-\s]+$/',
                        'message' => 'Caratère(s) non valide(s)'
                    ]),
                ]
            ])
```

Pour en savoir plus, rendez-vous sur la documentation officielle.

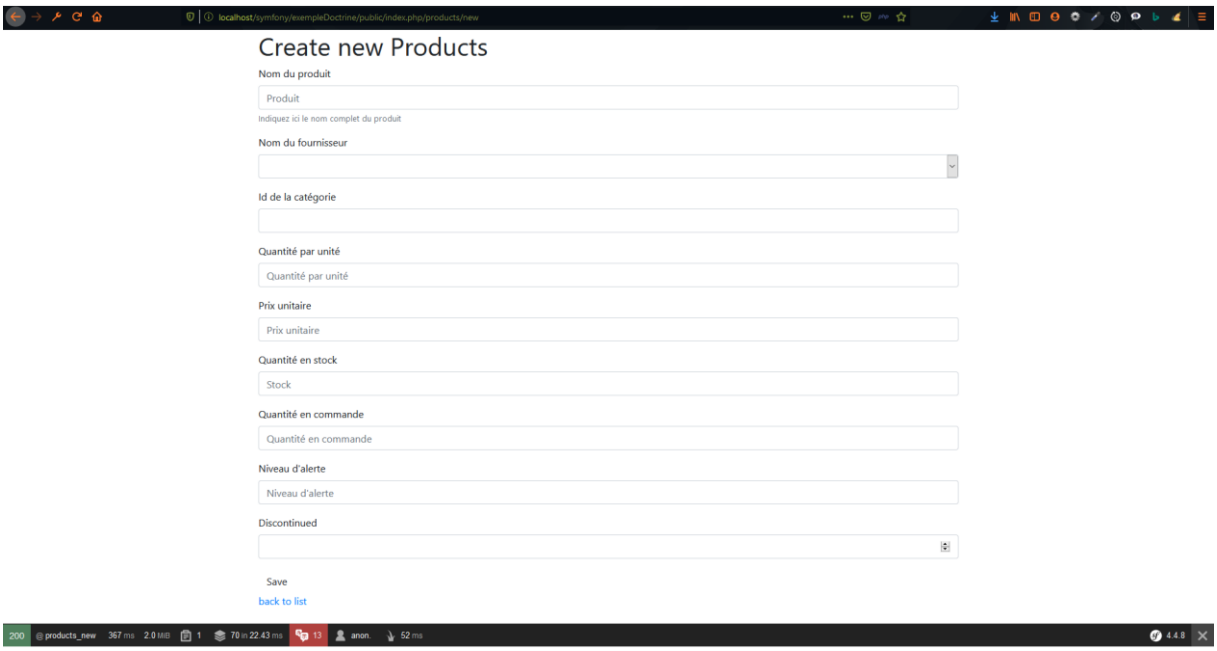
Procédez de la même manière pour les autres champs du formulaire d'ajout de produits.

Notre formulaire est maintenant lisible, et sécurisé. Mais certains champs pourraient être trop grand pour les données qui seraient saisies à l'intérieur. Rendons tout ça un peu joli ...

Côté Front-end

Personnalisation du rendu

À ce stade, vous devriez avoir un formulaire ayant plus ou moins ce visuel :



Passons à sa mise en forme et à la découverte de la structure de son template. En ouvrant le fichier `templates/products/new.html.twig`, on observe ... pas grand choses :

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

{% block title %}New Products{% endblock %}

{% block body %}
    <div class="container">
        <h1>Create new Products</h1>

        {{ include('products/_form.html.twig') }}

        <a href="{{ path('products_index') }}">back to list</a>
    </div>

{% endblock %}
```

Pas de formulaire, mais un `include` d'un fichier `_form.html.twig`. Il s'agit en fait du fichier qui contient notre formulaire. Mais pourquoi le mettre dans un fichier séparé ? Tout simplement parce que ce formulaire ne sert pas que pour l'ajout, il sert aussi pour la modification. Plutôt que de faire plusieurs formulaires identiques pour des actions différentes lors de la génération du CRUD, Symfony faire créer un seul formulaire qui sera réutilisable selon les besoins.

Ouvrons donc le fichier `_form.html.twig` :

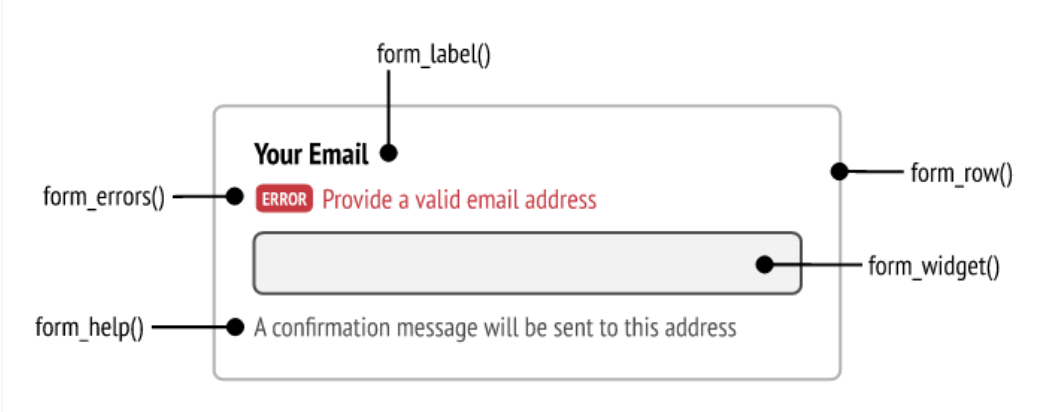
```
{{ form_start(form) }}
    {{ form_widget(form) }}
    <button class="btn">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Voici donc la structure d'un formulaire par Symfony :

- `{{ form_start(form) }}` correspond à la balise ouvrante `<form>`
- `{{ form_widget(form) }}` contient tous les champs de notre formulaire
- `<button class="btn">{{ button_label|default('Save') }}</button>` correspond au bouton de validation
- `{{ form_end(form) }}` correspond à la balise fermante `</form>`

Pour personnaliser le rendu du formulaire, il faut prendre connaissance de la structure d'un champ par Symfony.

Voici un schéma qui reprend tous les composants d'un champ :



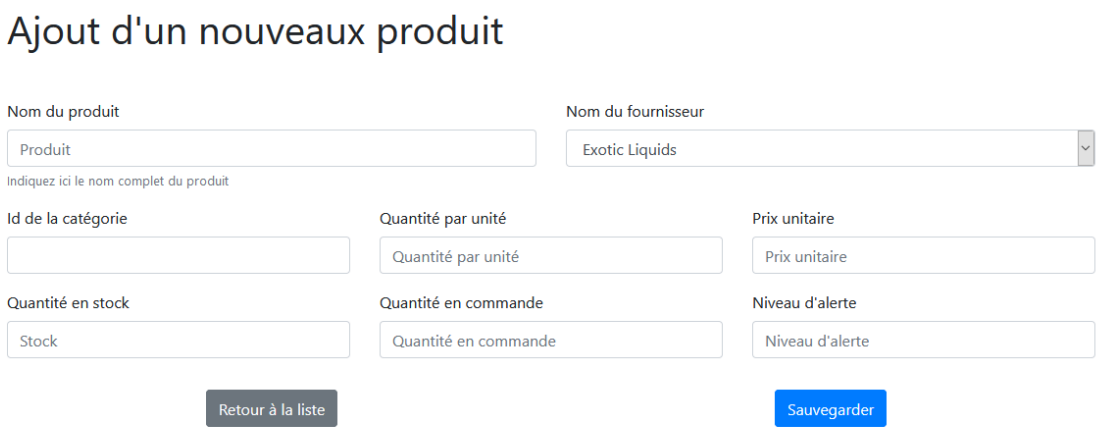
- `form_row()` reprend tous les éléments d'un champs
- `form_label()` indique le label de l'input
- `form_widget()` reprend l'input

- `form_errors()` servira à l'affichage des messages d'erreurs
- `form_help()` affiche les messages d'indications

Ainsi pour afficher un champ en particulier il faudrait procéder de la manière suivante :

```
{{ form_start(form) }}
   {{ form_row(form.ProductName) }}
<button class="btn">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Utilisez la grille de Bootstrap pour mettre en forme le formulaire en suivant ce modèle :



Messages d'erreur de Symfony

Si on souhaite tester la validation du formulaire en ne soumettant que des valeurs vide, on se rend compte que la validation du navigateur, dûe aux types des champs, prend le dessus sur Symfony.

Pour y remédier, nous devons ajouter un attribut `novalidate` sur le `{{ form_start(form) }}` :

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

Validez le formulaire en faisant des erreurs et observez le résultat : les erreurs générées par Symfony sont maintenant visibles. Les messages d'erreurs sont personnalisables lors de la définition des contraintes de validations.

Validation : méthode alternative

Comme pour beaucoup d'autres composants, Symfony propose différentes façons de faire une validation de formulaire. La méthode que l'on vient de voir est intéressant si on doit faire plusieurs formulaires reliés à une même entité, avec différents champs à chaque fois (tous les champs de l'entité ne sont pas sur un formulaire). On a une organisation propre à chaque formulaire.

Toutefois, si nous n'avons qu'un seul formulaire ou, si dans le cas où nous avons plusieurs formulaires, ils sont identiques et ont le même fonctionnement, il est inutile de faire plusieurs formType. De ce fait, pour alléger le formType, nous pouvons mettre les contraintes de validations dans l'entité.

Pour l'exemple, et pour éviter d'effacer ce que nous venons de faire, nous ferons un CRUD sur la table `suppliers` (`php bin/console make:crud Suppliers`).

Rendez-vous dans le fichier `Entity/Suppliers.php`. Pour faire notre validation ici, nous allons utiliser les **annotations**.

Les contraintes de validations seront donc ici déclarées sur les propriétés de notre entité :

```
/**
 * @ORM\Column(type="string", length=40)
 */
 private $CompanyName;
```

Rendons d'abord ce champ obligatoire, avec un message d'erreur :

```
/**      * @ORM\Column(type="string", length=40)
* @Assert\NotBlank(
*     message="Veuillez renseigner le nom du fournisseur"
* )
*/
private $CompanyName;
```

Ajoutons-y une regex :

```
/**      * @ORM\Column(type="string", length=40)
* @Assert\NotBlank(
*     message="Veuillez renseigner le nom du fournisseur"
* )
* @Assert\Regex(
*     pattern="/^[\s\w\#\_\-éèàçâêîôûùäaëïüö]+$/",
*     message="Caratère(s) non valide(s)"
* )
*/
private $CompanyName;
```

N'oubliez pas de faire les imports lorsque vous utiliser de nouveaux composants

Faites de même pour les autres champs du formulaire, et profites-en pour le rendre ergonomique. Testez ensuite le bon fonctionnement des contraintes de validations.