Formulaires Symfony avancés

Créeons un formulaire pour ajouter une Offre d'Emploi

php bin/console make:form OffreEmploiType OffreEmploi

```
C:\atelier\4D6\chomequipeut>php bin/console make:form OffreEmploiType OffreEmploi
 created: src/Form/OffreEmploiType.php

  Success!

 Next: Add fields to your form and start using it.
 Find the documentation at https://symfony.com/doc/current/forms.html

C:\atelier\4D6\chomequipeut>
```

```php
<?php
# .../Form/OffreEmploiType.php

namespace App\Form;

use App\Entity\Entreprise;
use App\Entity\OffreEmploi;
. . .

class OffreEmploiType extends AbstractType
{
 public function buildForm(FormBuilderInterface $builder, array $options): void
 {
     $builder
            ->add('titre')
            ->add('description')
            ->add('salaireAnnuel')
            ->add('datePublication', null, [
                'widget' => 'single_text',
            ])
            ->add('entreprise', EntityType::class, [
                'class' => Entreprise::class,
                'choice_label' => 'id',
            ])
        ;
    }
. . .
```

Adaptons **EntrepriseController.php** pour traiter ce formulaire :

```php
. . .
#[Route('/ajouterOffreEmploi', name:'ajouterOffreEmploi')]
public function ajouterOffreEmploi(ManagerRegistry $doctrine,
                                   Request $request): Response
{
```

```
    // On crée l'entity
    $offreEmploi = new OffreEmploi;

    // On crée le formulaire avec le OffreEmploiType
    $formExterne = $this->createForm(OffreEmploiType::class, $offreEmploi);

    $formExterne->handleRequest($request);
    // Les étapes suivantes ressemblent aux autres formulaires crées
. . .
```

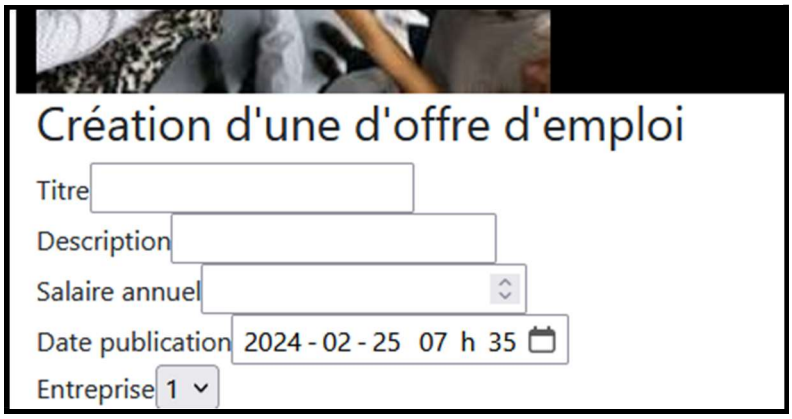
Ajoutons un bouton dans la zone de tests de **accueil.html.twig** pour ouvrir le formulaire

```
<a class='col-3' href="{{ path('ajouterOffreEmploi') }}">
   <button class='btn btn-warning'>Ajouter offre emploi</button>
</a>
```

Ajoutons le template qui affichera le formulaire

```
{# ajouterOffreEmploi.html.twig #}
{% extends('base.html.twig') %}

{% block body %}
<h3>Création d'une d'offre d'emploi</h3>
{{  form(formulaire) }}
{% endblock %}
```



Le comportement par défaut n'est pas tout à fait ce que l'on souhaite. Il faut :

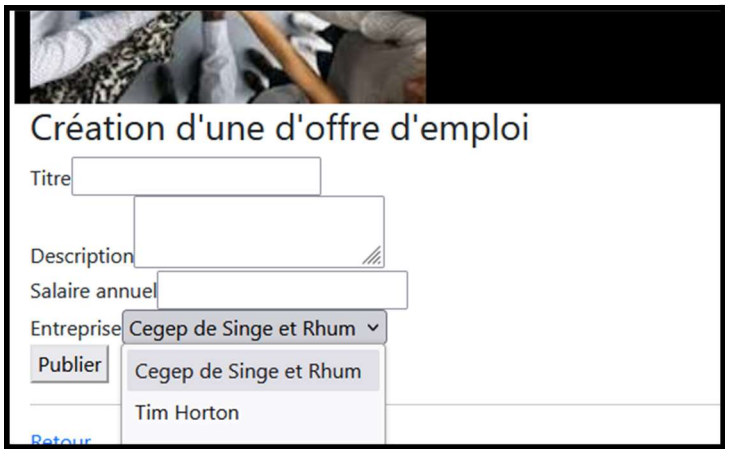
1- Gérer la date de publication par programmation
2- Offrir une liste déroulante avec le nom des entreprises plutôt que leurs id
3- Ajouter un bouton submit

Retournons dans **OffreEmploiType.php** et adaptons le formulaire pour nos trois besoins

```
. . .
use Symfony\Component\Form\Extension\Core\Type\{TextType, NumberType,
```

```
TextareaType, SubmitType};
. . .
public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
        ->add('titre', TextType::class)
        ->add('description', TextareaType::class)
        ->add('salaireAnnuel', NumberType::class)
        ->add('datePublication')
        ->add('entreprise', EntityType::class, [
                'class' => Entreprise::class,
                'choice_label' => 'nom',
                 ])
           ->add('Publier', SubmitType::class);
    }
. . .
```

Symfony ajoutera une liste déroulante des noms d'entreprises :



Modifiez le contrôleur responsable du formulaire d'offre d'emploi ainsi :

```
. . .
#[Route('/ajouterOffreEmploi', name:'ajouterOffreEmploi')]
public function ajouterOffreEmploi(ManagerRegistry $doctrine,
                                   Request $request): Response
{
  // On met le bon fuseau horaire
  ini_set('date.timezone', 'america/new_york');

  // On crée l'entity
  $offreEmploi = new OffreEmploi;
  // On met la date de publication par programmation
  $offreEmploi->setDatePublication(new \DateTime);

  // On crée le formulaire avec le OffreEmploiType
  $formExterne = $this->createForm(OffreEmploiType::class, $offreEmploi);

    $formExterne->handleRequest($request);
. . .
```

Personnalisation d'un formulaire

Jusqu'à maintenant nous avons affiché le formulaire Symfony par défaut :

```
{{  form(formulaire) }}
```

. C'est bien mais c'est insuffisant. Nous voulons contrôler plus finement son contenu. Entre autres nous voulons :

- Masquer certains champs
- Formater l'affichage
- Afficher les erreurs de validations

Modifiez le template **ajouterOffreEmploi.html.twig** ainsi :

```
{# ajouterOffreEmploi.html.twig #}
{% extends 'base.html.twig' %}
{% block body %}
<h3>Création d'une offre d'emploi</h3>
{{ form_start(formulaire)}}
  <div>
    {{ form_label(formulaire.titre, 'Titre') }}
    {{ form_widget(formulaire.titre) }}
    {{ form_errors(formulaire.titre) }}
  </div>
  <div>
    {{ form_label(formulaire.description, 'Compétences recherchées:') }}
    {{ form_widget(formulaire.description)}}
    {{ form_errors(formulaire.description) }}
  </div>
  <div>
   {{ form_label(formulaire.salaireAnnuel, 'Salaire annuel:') }}
   {{ form_widget(formulaire.salaireAnnuel)}}
   {{ form_errors(formulaire.salaireAnnuel) }}
  </div>

{{ form_end(formulaire)}}

<hr>
<a href="{{ path('accueil') }}">retour</a>
{% endBlock %}
```

Le template s'est complexifié. Entre autres nous n'utilisons plus le

```
{{ form(formulaire) }}
```

Nous utilisons maintenant les fonctions :

```
{{ form_start(formulaire)}}
{{ form_label(formulaire.titre, 'Titre') }}
{{ form_widget(formulaire.titre) }}
{{ form_errors(formulaire.titre) }}
{{ form_end(formulaire)}}
```

form_start() indique le début du formulaire(<form>) ainsi que **form_end**() en indique la fin (</form>)

Pour chaque champ du formulaire on utilisera un

- **form_label()**, pour l'étiquette du champ,
- **form_widget()** pour la zone d'input du champ et
- **form_errors**() pour afficher les messages d'erreurs générés lors du `if ($formulaire->isValid())`

De plus **Twig** affichera automatiquement les champs non mentionnés, par exemple le bouton de soumission. Si ce n'est pas ce qu'on veut, on peut empêcher ce comportement en donnant un paramètre facultatif à form_end :

```
{{  form_end(formulaire, {'render_rest': false}) }}
```

Si certains champs ne doivent pas être modifiés par l'utilisateur, on utilisera le :

```
{% do formulaire.nomDuChampANePasAfficher.setRendered %}
```

Note : si vous utilisez un setRendered sur un champ qui ne peut être nullable, il est de votre responsabilité d'initialiser ce champ dans le contrôleur après le **->HandleRequest()** et avant d'appeler le **->persist()** sinon vous frapperez une erreur du genre : Integrity constraint violation : 1048 Le champ 'date' ne peut être vide (null).

Une autre tactique pourrait être d'enlever ce champ du formulaire (ne pas faire un « **->add() »** pour ce champ) dans ce cas vous devrez aussi l'initialiser vous-même avant le **->persist()** mais vous n'aurez pas à le « .setRendered » dans le twig.

Validation et traitement des erreurs

Validation client vs serveur

Les formulaires Web ont deux niveaux de validations : client et serveur. Le niveau client consiste en des validations ergonomiques qui facilitent la vie de l'utilisateur. Elles sont définies dans le HTML et par conséquences très faciles à déjouées et ne constituent pas une barrière de sécurité.

Les validations serveur ne sont visibles que par le serveur et sont le dernier rempart avant les actions d'écriture en BD. Elles sont donc stratégiques et plus importantes.

Validation côté client

Comment ajouter des validations client dans nos formulaires Symfony? On peut le faire à deux endroits :

a) Dans la définition de notre form en lui fournissant un type Symfony. Par exemple je désire que le champ salaire devienne MoneyType et puisse être null:

```
class OffreEmploiType extends AbstractType
{
    . . .
```

```
  public function buildForm(FormBuilderInterface $builder, array $options):
void
    {
        $builder
        ->add('titre', TextType::class)
        ->add('description', TextAreaType::class)
        ->add('salaireAnnuel', MoneyType::class, ['required' => false,
'currency' => 'USD'])
. . .
```

b) Dans nos templates twig on peut ajouter des attributs HTML aux **form_widget().** Par exemple si je veux limiter la taille d'un champ texte je pourrais ajouter **:**

```
{{ form_widget(formulaire.titre, {attr:{'maxlength':'15'}} ) }}
```

En fait, dans un **form_widget,** je peux ajouter tous les attributs HTML nécessaires, incluant des attributs de formatage CSS, par exemple :

```
{{ form_widget(formulaire.titre, {attr:{ 'maxlength': '15', 'class':'col-4
form-control'}} ) }}
```

Validations de niveau serveur.

Si un champ doit respecter certaines contraintes au niveau serveur, il faut les spécifier dans la définition de l'entité. Modifions l'entité **OffreEmploi** pour ajouter six contraintes (lignes en jaune) à ses attributs:

```
<?php
#   . . ./src/Entity/OffreEmploi.php
. . .
use Symfony\Component\Validator\Constraints as Assert;

#[ORM\Entity(repositoryClass: OffreEmploiRepository::class)]
class OffreEmploi
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 100)]
    #[Assert\Length(min:2, minMessage:'deux caractères minimum')]
    #[Assert\Length(max:100, maxMessage:'100 caractères maximum')]
    private ?string $titre = null;

    #[ORM\Column(length: 255)]
    #[Assert\Length(min:2, minMessage:'deux caractères minimum')]
    #[Assert\Length(max:255, maxMessage:'255 caractères maximum')]
    private ?string $description = null;

    #[ORM\Column(name:'salaireAnnuel')]
    #[Assert\LessThan(300000, message:'maximum 300000')]
    #[Assert\GreaterThan(1000, message:'doit dépasser 1000')]
```

```
    private ?int $salaireAnnuel = null;
```

Nous avons ajouté trois types d'annotations valideuses

#[Assert\Length(…)
#[Assert\LessThan(…)
#[Assert\GreaterThan(…)
Il en existe plusieurs autres : https://symfony.com/doc/current/reference/constraints.html

Si l'utilisateur fournit des infos ne respectant pas les contraintes définies dans l'**entity**, l'appel du

```
if ($formulaire->isValid())
```

donnera **false** et on pourra exécuter un **else** pour donner une rétroaction à l'utilisateur :

- L'aviser qu'il a des erreurs
- Réafficher le formulaire avec les champs contenant les données qu'il a déjà entrées (souci d'ergonomie pour ne pas le forcer à refaire la fastidieuse tâche d'entrée de données)
- Indiquer où sont les erreurs
- Indiquer quelles sont les erreurs.

Flashbags

Les flashbags sont des moyens puissants offerts par Symfony pour communiquer de l'information contextuelle à l'utilisateur. Ils fonctionnent de telle sorte que tant qu'ils n'ont pas été affichés ils s'accumulent et lorsqu'ils sont affichés une fois, ils disparaissent. Voici comment les déployer :

Dans le contrôleur, lorsque la ligne

```
if ($formulaire->isValid())
```

est « false », on peut initialiser un **flashbag** pour éventuellement aviser l'utilisateur de l'échec :

```
. . .
$formExterne->handleRequest($request);
if ($formExterne->isSubmitted())
{
   if ($formExterne->isValid())
   {
      . . .
     return $this->redirectToRoute('OffreEmplois');
   }
   else
   {
      $msg = "Au moins une erreur. Veuillez corriger et revalider";
      $this->addFlash('notice', $msg);
   }
   return $this->render('formulaire/ajouterOffreEmploiALaCarte.html.twig',
                  array('formulaire' => $formExterne->createView);
}
```

```
. . .
```

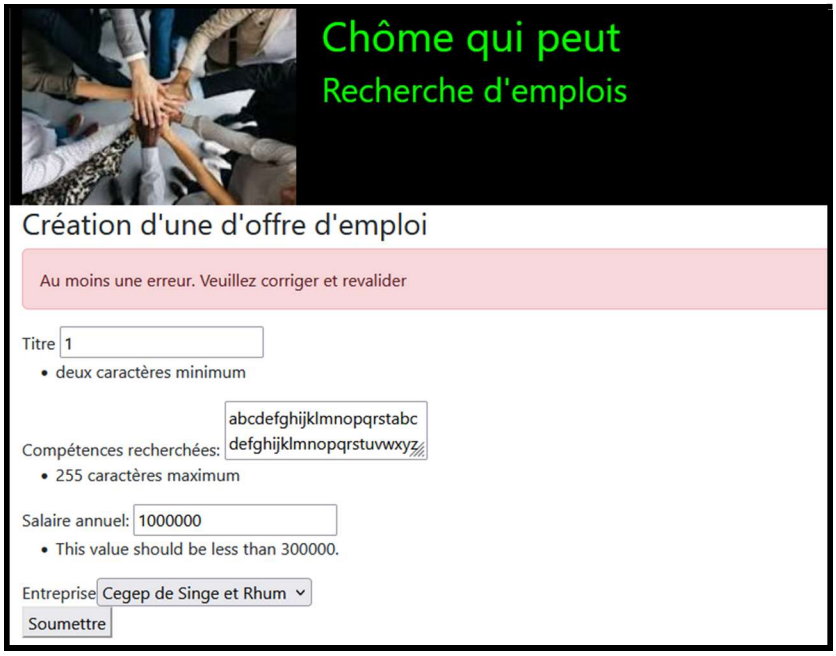
Les twig accèdent aux flashbags dans le but des afficher :

```
{# ajouterOffreEmploi.html.twig #}
. . .
<h3>Création d'une offre d'emploi</h3>

{% if app.flashes('notice') is defined %}
   {% for notice in app.flashes('notice') %}
   <section class='alert alert-danger'>
        {{ notice }}
   </section>
   {% endfor %}
{% endif %}
{{ form_start(formulaire)}}
...
```

Lorsque les contraintes ne sont pas respectées, les champs en erreur seront suivis d'un message expliquant l'erreur. C'est la portion des **form_errors**() dans les templates twig.

Essayez de soumettre votre formulaire en violant volontairement une contrainte définie et voyez la rétroaction qui est maintenant offerte à l'utilisateur : flashbag et messages d'erreur associés au champ fautif :



Avec un peu de CSS vous pouvez dramatiser la présentation des erreurs. Ce formulaire est puissant et ergonomique.