Symfony 5- Les formulaires : concepts de base

L'idée de base des formulaires sous Symfony est qu'un formulaire se construit à partir d'un **objet** existant, et ses attributs seront remplis avec les valeurs saisies dans le formulaire présenté à l'internaute. Le rôle du formulaire consiste donc à hydrater l'objet, c'est à dire remplir partiellement ou totalement ses attributs. Ces objets sont généralement utilisés pour modifier des données de la couche Modèle.

Le contrôleur est chargé de :
- créer ou récupérer l'objet de la couche Modèle
- créer le formulaire et l'associer à un objet de la couche Modèle
- passer le formulaire à la vue pour que celle-ci s'occupe de l'affichage des différents champs
- valider les données soumises par l'internaute.

1. Construction d'un formulaire

La construction d'un formulaire sous Symfony peut être réalisée à l'aide du composant **FormBuilder**. Ce composant permet de créer et de configurer des objets **Form**. L'objet Form est l'élément utilisé par le contrôleur, il correspond à l'ensemble des champs du formulaire, sous forme hiérarchique.

Il existe plusieurs façons de créer un formulaire :
- sans utiliser un objet.
- en utilisant un objet.
- en utilisant un constructeur de formulaire créé à partir d'un objet ou d'une entité.

1.1. Construction basique d'un formulaire sans utilisation d'objet

Avant d'utiliser les formulaires, on va installer la fonctionnalité des formulaires :

$ **composer require symfony/form**

Il existe des cas où un formulaire n'a pas obligatoirement besoin d'un objet de la couche modèle pour fonctionner. Par exemple, le formulaire d'authentification, il ne correspond pas forcément à une table en base de données.

Le code suivant illustre un exemple d'utilisation de composant **FormBuilder** sans utilisation d'un objet. Dans ce cas, on indique au composant :
- le nom du champ
- le type du champ
- éventuellement les options du champ (champ requis ou pas, label du champ dans le formulaire….)

L'objet **Form** comporte un certain nombre de méthodes dont les indispensables sont :
- la méthode **handleRequest()** qui prend en paramètre la requête http courante (affichage ou soumission) ;
- la méthode **isSubmitted()** permet de tester si le formulaire a été soumis. Si c'est le cas, la méthode rattache manuellement les données au formulaire ;
- la méthode **isValid()** permet de tester si les valeurs soumises sont valides dans le cas où des contraintes ont été définies par le programmeur ;
- la méthode **getData()** qui permet de récupérer les informations saisies dans les champs du formulaire sous forme de tableau.

```
// Dans le contrôleur ConnexionController - Création d'un formulaire
namespace App\Controller ;
```

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ResetType;

class ConnexionController extends AbstractController {
    public function formToConnect()
    {
        $form = $this->createFormBuilder()
                    ->add('login', TextType::class)
                    ->add('motDePasse', PasswordType::class)
                    ->add('Valider',SubmitType::class)
                    ->add('annuler',ResetType::class)
                    ->getForm();


         $request = Request::createFromGlobals(); // sauf si en paramètre de la
                                                  //méthode,on a Request $request
        $form->handleRequest($request) ;
        // Le visiteur a appuyé sur le bouton Valider
         // et les données saisies sont conformes à la validation des champs
        if ($form->isSubmitted() && $form->isValid()) {

           $data = $form->getData() ;
           // Traitement du formulaire
           return $this->render('connexion/form_to_connect_submit.html.twig',
                           array('data'=>$data));
        }

        // Affichage du formulaire
        return $this->render('connexion/form_to_connect.html.twig',
                ['form'=>$form->createView()]);
    }
}
```

```
{# Dans la vue twig : form_to_connect.html.twig #}
{# … #}
{% block body %}
    {{ form(form) }}
{% endblock %}
```

Si on veut avoir un rendu plus joli, on peut intégrer dans le fichier :

```
{# Dans la vue twig : form_to_connect.html.twig #}
{% block stylesheets %}
    {% form_theme form 'bootstrap_5_layout.html.twig' %}
{% endblock %}
{% block body %}
    {{ form(form) }}
{% endblock %}
```

Et installer : **`composer require twbs/bootstrap`**

Si on veut que ce soit global à tous les formulaires, modifier le fichier **config/packages/twig.yaml** :

```
twig :
    form_themes : ['bootstrap_5_ layout.html.twig']
```

1.2. Construction d'un formulaire à partir d'un objet

Les formulaires ont pour vocation d'enregistrer ou de modifier un objet de la couche modèle. La correspondance des données du formulaire aux différentes propriétés de l'objet de la couche modèle s'appelle « **mapping** ».

L'exemple suivant montre la construction d'un formulaire qui va hydrater (instancier les attributs de) l'objet passé en paramètre au composant FormBuilder. On commence par construire l'objet Artisan. Cette classe peut être générée automatiquement quand on verra Doctrine, on parlera alors **d'entités** (**entity**).

On va construire un formulaire qui va nous permettre de créer un objet de type Secteur (voir contexte GSB).

```
// Dans le contrôleur AccueilController.php
public function newSecteur(Request $request) {
    $secteur = new Secteur();
    $form = $this->createFormBuilder($secteur)
             ->add('libelle', TextType::class)

             ->add('valider', SubmitType::class)  //Symfony ne mappe pas les champs
             ->add('annuler', ResetType::class)  // de type Submit et Reset
             ->getForm();

    $form->handleRequest($request) ;

    if ($form->isSubmitted()) {
        return $this->render('accueil/form_secteur.html.twig',
                             array('secteur'=>$secteur));
    }
    return $this->render('accueil/form_new_secteur.html.twig',
                         array('form'=>$form->createView()));
}
```

Il est important de faire correspondre les types de champ avec les types d'attributs de l'objet.

```
{# Dans la vue twig : form_secteur.html.twig #}

{% block body %}
    {{ form(form) }}
{% endblock %}
```

```
{# Dans la vue twig : form_new_secteur.html.twig #}
{% block body %}
    {{ secteur.id }} : {{ secteur.libelle }}
{% endblock %}
```

Travail à faire :
L'id du secteur est null. Persister l'objet créé avec le formulaire en base de données. Que contient l'id ? Vérifier l'enregistrement en base de données.

2. Personnaliser l'affichage d'un formulaire

La personnalisation de l'affichage du formulaire peut être effectuée dans les vues (ordre d'apparition des champs, labels...) ou lors de la définition du formulaire dans le contrôleur.

2.1. Dans le contrôleur :

```
$this->createFormBuilder([$objet])->add('nomChamp', TypeChamp::class, array('option1' [=> 'Valeur']))
```

La méthode add() du composant FormBuilder permet de définir les champs d'un formulaire. Les arguments de la méthode sont :

- le nom du champ
- le type du champ
- les options (variables) du champ, sous forme de tableau. Les options peuvent être utilisées pour personnaliser les champs du formulaire.

Il est important de faire correspondre les types de champ avec les types d'attributs de l'objet.
Il est possible de mettre des valeurs prédéfinies dans les champs. Il suffit de modifier l'instance de l'objet avant de la passer en argument à la méthode createFormBuilder.

Exemple de syntaxe :

```
$form = $this->createFormBuilder($secteur)
        -> add('libelle',TextType::class,array('label'=>'Libellé','required'=>true))
       ->getForm() ;
```

Exemples d'options

Option	Rôle	Exemple
label	Modifie le libellé du champ	$formBuilder->add('libelle',TextType::class,**array('label'**=> 'Libellé'))
data	Donner une valeur par défaut	$formBuilder->add('libelle',TextType::class,**array('data'**=> 'xxx'))
required	Le champ doit être renseigné Le champ est facultatif	$formBuilder->add('libelle',TextType::class,**array('required'=>true)**) $formBuilder->add('libelle',TextType::class,**array('required'=>false)**)
disabled	Le champ est affiché mais sera ignoré quand on soumet le formulaire	$formBuilder->add('libelle',TextType::class,**array('disabled'=>true)**)
mapped	Le champ n'est pas associé à l'attribut dans le modèle	$formBuilder->add('libelle',TextType::class,**array('mapped'=>false)**)
attr	Ajoute des attributs html au champ	$formBuilder->add('libelle',TextType::class, **array('attr'=>array('class'=>'classe_champ')**)

2.2. Dans la vue :

```
{# ouverture de la balise d'ouverture <form> #}
{{ form_start(form) }}

    <div class="form-error-message"> {{ form_errors(form) }} </div>

    {# Génération du composant input
    <div class="form-control"> {{ form_widget(form.libelle) }} </div>

    {# Autre façon de personnaliser un champ #}
    <div class="form-row">{{ form_row(form.libelle, {'label' :'Libellé'})}}
    </div>


    {{ form_rest(form) }} {# si on ne souhaite pas les afficher :
                             form_rest(form, { render_rest : false }) #}


    <p class="btn btn-primary">
        <button type="submit"> Valider </button>
        <button type="reset"> Annuler </button>
    </p>
```

```
{{ form_end(form) }}
```

Les fonctions utilisées sont :

- ⇨ form_start(form_view) : affiche le formulaire dont le nom est indiqué en 1er paramètre (form).
- ⇨ form_end(form_view, variables) : permet de fermer la fonction form_start(). On indique en 1er le nom du formulaire. On peut également afficher les champs restants du formulaire non personnalisés => {{ form_end(form_view, {'render_rest':true}) }}. Si on met false, tous les champs restants seront ignorés.
- ⇨ form_label(form_view.field, label, variables) : permet de donner un label et éventuellement d'autres attributs au champ indiqué en 1er paramètre.
- ⇨ form_widget(form_view.field, variables) : affiche le composant du champ.
- ⇨ form_row(form_view.field, variables) : affiche le champ du formulaire en incluant, le label, le widget, et les erreurs.
- ⇨ form_rest(form) affiche tous les champs restants qui n'ont pas encore affichés.

Pour aller plus loin => How to Customize Form Rendering (Symfony Docs)

Il est possible également de personnaliser un formulaire à l'aide de thèmes twig de formulaires. Le thème peut être défini à l'intérieur de la vue ou externalisé et ensuite appelé dans la vue. Un thème peut personnaliser plusieurs blocs. Les blocs utilisés sont : form_row, form_errors, form_widget, form_label, form_errors.

Exemple :

```
{# templateForm.html.twig #}
{% block form_row %}
    <div style="margin : 15px 15px ; ">
       <div style="float:left ; width:200px ;"> {{ form_label(form) }} </div>
       <div class="form-control">  {{ form_widget(form) }} </div>
       <div style="color: red ;">  {{ form_errors(form) }} </div>
    </div>
{% endblock %}
```

Utilisation du thème dans la vue du formulaire :

```
{% form_theme form "templateForm.html.twig" %}
{{ form(form) }}
```

3. Types de champs d'un formulaire Sf

Les classes utilisées se trouvent dans le répertoire : \Symfony\Component\Form\Extension\Core\Type

3.1. Champs de type text

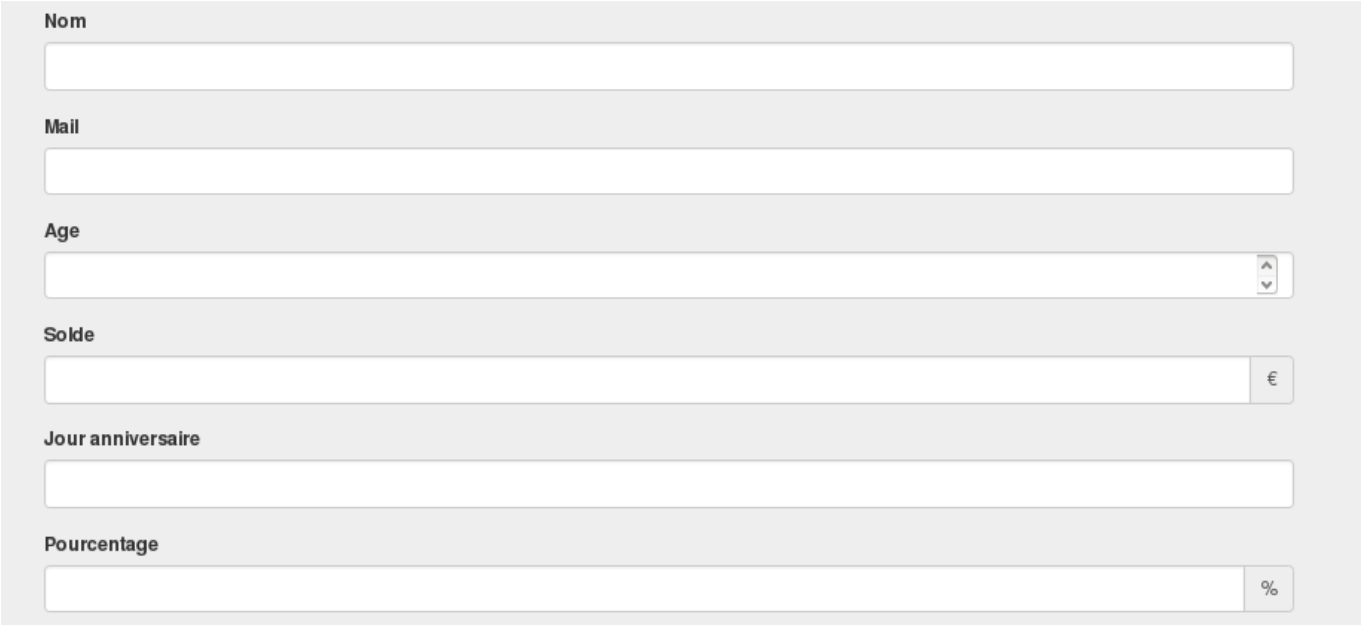
TextType::class	TextareaType::class	EmailType::class	PasswordType::class	SearchType::class
MoneyType::class	NumberType::class	IntegerType::class	PercentType::class	UrlType::class

Travail à faire :

Créer le formulaire correspondant à la capture suivante en respectant les contraintes suivantes :

- ⇨ Nom => supprimer les blancs en début et en fin de saisie dans le widget
- ⇨ Mail => champ obligatoire
- ⇨ Age => champ facultatif. S'il est renseigné, la valeur saisie doit être entre 1 et 120.
- ⇨ Solde => facultatif
- ⇨ Jour anniversaire => libellé : Jour d'anniversaire

⇨ Pourcentage => facultatif.



3.2. Champs de type date/temps

DateTimeType::class	TimeType::class	BirthdayType::class

Travail à faire :
Ajouter au formulaire les champs suivants. La plage de valeurs de la date d'embauche : de 2000 à 2021.

3.3. Champs de choix

ChoiceType

ChoiceType Field (select drop-downs, radio buttons & checkboxes) (Symfony Docs)

La classe ChoiceType permet de générer les composants de sélection. Pour cela, il faut se servir des options 'expanded' et 'multiple'. Par défaut ces options sont à false => **Liste déroulante**.

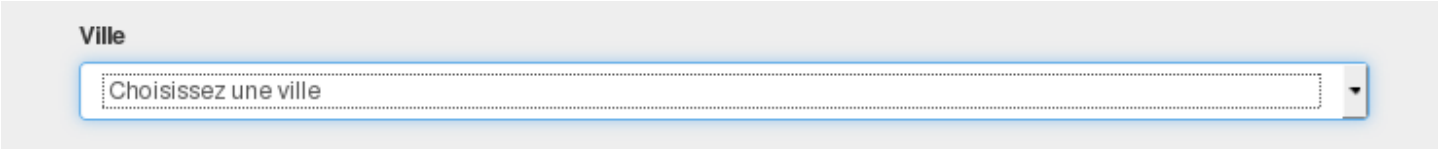
- ⇨ Composant **checkbox** : 'expanded' et 'multiple' à true
  Exemple :

```
->add('saison',ChoiceType ::class,['label'=>'Saison préférée',
            'choices'=>['Automne','Hiver','Printemps','Eté'],
            'expanded'=>true, 'multiple'=>true ] )
```

- ⇨ Composant **radio** : 'expanded' à true
  Exemple :

```
->add('nationalite', choiceType::class,['label'=>Nationalité',
                'choices'=>['Française'=>'f', 'Autre'=>'a'],
                'expanded'=>true ] )
```

Travail à faire :
Tester les composants ci-dessus et ajouter la liste déroulante ci-dessous.



3.4. Autres composants de choix

EntityType::class => liste déroulante construite à partir de valeurs d'une colonne d'une entité
Voir : https://symfony.com/doc/5.4/reference/forms/types/entity.html

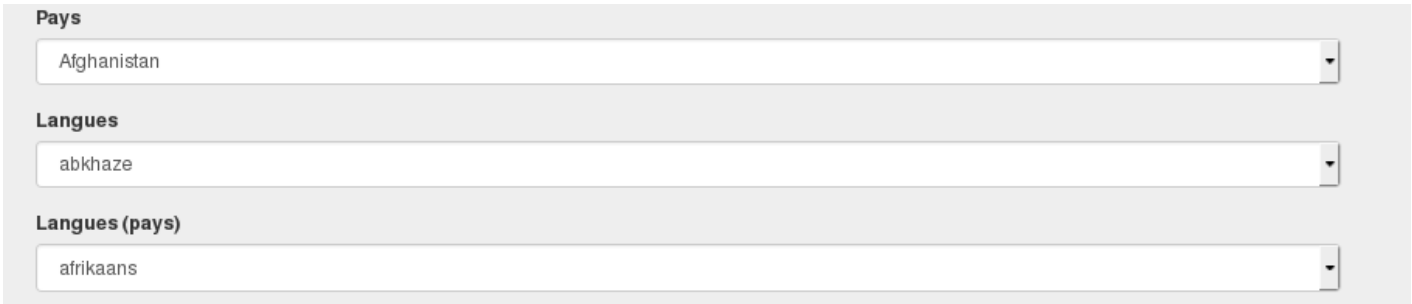
CountryType::class => affiche les pays :
5.4https://symfony.com/doc/5.4/reference/forms/types/country.html

LanguageType::class => affiche les langues :
https://symfony.com/doc/5.4/reference/forms/types/language.html

LocaleType::class => affiche les langues des pays :
https://symfony.com/doc/5.4/reference/forms/types/locale.html

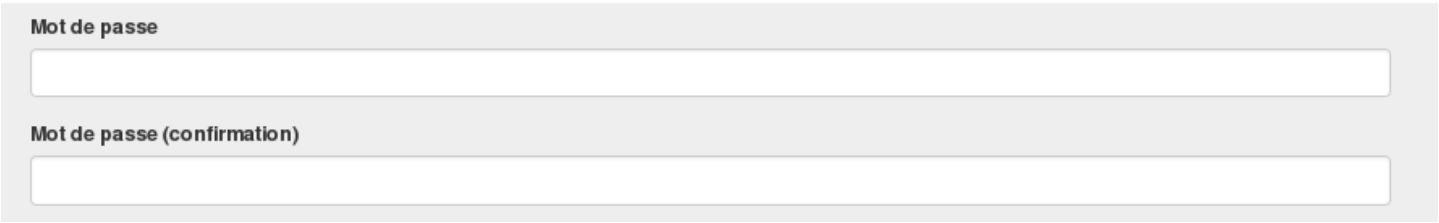
TimezoneType::class => affiche les timezones :
https://symfony.com/doc/5.4/reference/forms/types/timezone.html

Travail à faire :
Ajouter les champs suivants au formulaire.



3.5. Répétition de champs : repeated

Le type **repeated** permet d'avoir deux champs de même type dont le contenu doit être identique. C'est le cas par exemple de la confirmation d'un mot de passe lors de sa création.



3.6. Autres composants

Divers	Multiple	Caché	Boutons
FileType ::class	CollectionType ::class	HiddenType ::class	ButtonType, ResetType, SubmitType ::class

4. Externaliser la définition des formulaires

L'intérêt d'externaliser la définition des formulaires est de pouvoir les réutiliser. Pour cela, il faut déclarer la définition du formulaire dans une classe à part et non plus dans le contrôleur. Cette classe est appelée constructeur de formulaire.
Les constructeurs de formulaires sont définis dans le répertoire **Form** (ce répertoire n'existe pas par défaut, il peut être créé manuellement ou à l'aide d'une commande).
Le nom du fichier correspondant au constructeur de formulaire doit être suffixé de **Type**.
Un constructeur de formulaire crée un formulaire correspondant à une classe ou une entité, c'est à dire que les champs du formulaire sont constitués, par défaut, des attributs de la classe ou de l'entité. On peut supprimer les champs qu'on ne veut pas afficher dans le formulaire. Les entités se trouvent dans le répertoire Entity.

4.1. Définition du formulaire

Prenons l'exemple d'un constructeur qui permet de créer un formulaire de connexion. Le fichier s'appellera ConnexionForm**Type**.php et se trouvera dans le répertoire Form.

Travail à faire :

- ✓ Créer le formulaire type. Pour cela, exécuter la commande : **php bin/console make:form**
- ✓ Le système demande le nom du formulaire type => ConnectionForm**Type**
- ✓ Le système demande le nom de l'entité ou de la classe modèle ou de laisser vide. Laisser vide.
- ✓ Le répertoire Form sera créé ainsi que le squelette de la classe du formulaire type.
- ✓ Compléter le fichier du formulaire type avec les informations : login, mdp et profil.
- ✓ Créer une classe Connection dans le répertoire technique avec les attributs login, mdp et profil.

```
<?php
namespace App\Form ;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ConnectionFormType extends AbstractType {

    public function buildForm(FormBuilderInterface $builder, array $options) {

        $builder
            ->add('login', TextType ::class, array('required'=>true))
            ->add('mdp', PasswordType ::class, array('label'=>'Mot de passe'))
            ->add('profil', ChoiceType ::class,
                array('choices'=>array( 'Artisan'=>'artisan',
                                        'Entrepreneur'=>'entrepreneur'),
```

```
                    'expanded'=> true, 'multiple'=>false))
            ->add('valider', SubmitType ::class)          // à mettre dans le formulaire, bonne pratique
            ->add('annuler',ResetType ::class)           // idem
        ;
    }

    public function configureOptions(OptionsResolver $resolver) {
        $resolver->setDefaults( [ 'data_class' => 'App\technique\Connection'] );
    }

    public function getName() {
            return 'connection';
    }
```

4.2. Utilisation dans le contrôleur

Du côté du contrôleur, la construction du formulaire est réalisée grâce à la méthode **createForm()** et non plus **createFormBuilder().**

Exemple :

```
public function formExterne(Request $request) {
        $connection = new Connection() ;
        $form = $this->createForm(ConnectionFormType::class, $connection);

        $form->handleRequest($request);

        if (($form->isSubmitted() && $form->isValid()) {
            // Traitement du formulaire
        }
     return $this->render('secteur/demo_form_externe.html.twig', array('form'=>$form->createView()));
    }
```

Le formulaire type dont le nom a été indiqué sera créé avec les champs de la classe. Charge à vous par la suite d'indiquer des options si nécessaire et de supprimer les champs qui ne peuvent ou ne doivent pas être initialisés via le formulaire.

5. Génération automatique et accélération du développement

5.1. Générer automatiquement les formulaires types

L'externalisation des formulaires prend tout son intérêt avec l'utilisation des entités doctrine. Les constructeurs de formulaires sont alors générés de manière automatique à l'aide de la commande

```
php bin/console make:form
```

Le formulaire type est créé dans le répertoire src/Form. On peut alors personnaliser les champs avec les différentes options.

5.2. Générer automatiquement les formulaires types

Il est possible de générer automatiquement le contrôleur et les vues liés à une entité à l'aide de CRUD.

Il suffit pour cela de lancer la commande suivante :

```
php bin/console make:crud
```

La commande nécessite au préalable les paquets : twig (si ce n'est pas encore installé), **validator, security-**

csrf et annotations.

Le système vous demande de choisir l'entité souhaitée. Un contrôleur portant le nom de l'entité sera créé ainsi que des formulaires permettent de réaliser toutes les opérations CRUD (Create-Read-Update-Delete) de l'entité concernée.

Travail à faire :

- Réaliser l'application Gsb-Web en respectant le cahier des charges. Pour l'instant, ne vous occupez pas de l'authentification des utilisateurs.