

Prise en main de Symfony 4.0

Les formulaires

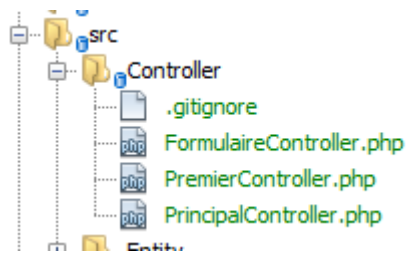
L'essentiel

Environnement :

- ✓ Serveur apache : celui de wamp (préféré à celui intégré à symfony)
- ✓ Base de données : mysql en local (wamp)
- ✓ IDE : netbeans
- ✓ virtualHost : premierProjet40.Sym
- ✓ On continue notre premier projet
- ✓ Les bundles form et validator ont été chargés

Partie 1 : PREMIER EXEMPLE

Vous allez créer un nouveau contrôleur appelé FormulaireController



```
/**
 * @Route("/exempleformulaire", name="exempleformulaire")
 */
public function exempleFormulaire(Request $request)
{
    //Création d'un employe
    $employe= new Employe();
    // $employe->setNom("Dupont");
    // $employe->setSalaire(10000);

    $form = $this->createFormBuilder($employe)
        ->add('nom', TextType::class)
        ->add('salaire', NumberType::class)
        ->add('Enregistrer', SubmitType::class, array('label' => 'Créer'))
        ->getForm();

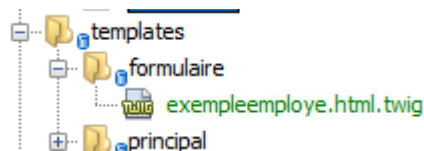
    return $this->render('formulaire/exempleemploye.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

La création d'un formulaire requiert relativement peu de code car les objets de formulaire Symfony sont construits avec un "constructeur de formulaire". Le but du constructeur de formulaire est de vous permettre d'écrire des "recettes" de forme simple et de faire tout le poids de la construction du formulaire.

Dans cet exemple, vous avez ajouté deux champs à votre formulaire - task et dueDate - correspondant aux propriétés task et dueDate de la classe Task. Vous avez également affecté un "type" (par exemple TextType et DateType), représenté par son nom de classe complet. Entre autres, il détermine quel tag de formulaire HTML est rendu pour ce champ.

Enfin, vous avez ajouté un bouton d'envoi avec une étiquette personnalisée pour soumettre le formulaire au serveur.

Symfony est livré avec de nombreux types intégrés qui seront discutés sous peu (voir Types de champs intégrés).



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

{% block title %}Employe{% endblock %}
{% block body %}
    {{ form_start(form) }}
    {{ form_widget(form) }}
    {{ form_end(form) }}
{% endblock %}
```

- ✓ **form_start (écran) :** Rend la balise de début du formulaire, y compris l'attribut enctype correct lors de l'utilisation des téléchargements de fichiers.
- ✓ **form_widget (formulaire) :** Rend tous les champs, ce qui inclut l'élément de champ lui-même, une étiquette et tous les messages d'erreur de validation pour le champ. On verra plus loin qu'on peut détailler l'affichage de chaque champ du formulaire.
- ✓ **form_end (formulaire) :** Rend l'étiquette de fin du formulaire et tous les champs qui n'ont pas encore été rendus, dans le cas où vous avez rendu chaque champ vous-même. Ceci est utile pour rendre les champs cachés et tirer parti de la protection CSRF automatique.

Voici le résultat :

The screenshot shows a web browser window with the title 'Employe'. The address bar shows the URL 'premierprojet40.sym/exempleformulaire'. The form contains two text input fields: 'Nom' and 'Salaire'. Below the 'Salaire' field is a 'Créer' button.

Saisissez des valeurs :

The screenshot shows the same form as before, but now the 'Nom' field contains the text 'Bondeau' and the 'Salaire' field contains the text '32000'. The 'Créer' button is still present.

Et soumettez le formulaire

Cet exemple suppose que vous soumettez le formulaire dans une requête "POST" et dans la même URL que celle dans laquelle il a été affiché.

On verra plus tard comment modifier la méthode de requête et l'URL cible du formulaire.

Par contre ouvrez la barre de debug en bas, et allez voir dans le menu Request/Response : on voit bien que le formulaire appelé, c'est-à-dire lui-même a bien reçu les valeurs en post :

The screenshot shows the Symfony Dev Toolbar with the 'Request / Response' tab selected. The 'Request' sub-tab is active. Under 'GET Parameters', it says 'No GET parameters'. Under 'POST Parameters', there is a table showing the submitted data:

Key	Value
form	[<ul style="list-style-type: none"> "nom" => "Bondeau" "salaire" => "32000" "Enregistrer" => ""]

Le système de formulaire ne peut accéder à la valeur des propriétés privées ou protégées d'une classe qu'à la seule condition qu'elles aient un getter et un setter.

Pour une propriété booléenne, vous pouvez utiliser une méthode "is" ou "has" (par exemple isActive () ou hasRendu ()) à la place d'un getter (par exemple getActif () ou getRendu ()).

Gestion des soumissions de formulaire



Par défaut, le retour de formulaire est géré dans la méthode du contrôleur qui l'a généré (appelé). La méthode reçoit une requête (request) de type POST.

On va donc rajouter dans la méthode du contrôleur le code qui va gérer le retour du formulaire.

Modifiez la méthode exemple employe de votre contrôleur pour obtenir ceci :

```
        ->getForm();

// code à rajouter ici
$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()) {
    $employe = $form->getData();
    // code de gestion des données reçues
    return $this->redirectToRoute('employe_ok', array("nom" => $employe->getNom()));
}
//fin du code à rajouter
return $this->render('formulaire/exempleemploye.html.twig', array(
```

Ce contrôleur suit un modèle commun pour la gestion des formulaires et a trois chemins possibles:

- ✓ Lors du chargement initial de la page dans un navigateur, le formulaire est créé et rendu. `handleRequest()` reconnaît que le formulaire n'a pas été soumis et ne fait rien. `isSubmitted()` renvoie la valeur `false` si le formulaire n'a pas été envoyé.
- ✓ Lorsque l'utilisateur soumet le formulaire, `handleRequest()` le reconnaît et écrit immédiatement les données soumises dans les propriétés `nom` et `salaire` de l'objet `$employe`. Ensuite, cet objet est validé. Si elle est invalide (on verra plus loin), `isValid()` renvoie `false` et le formulaire est rendu à nouveau, mais maintenant avec des erreurs de validation;
- ✓ Lorsque l'utilisateur soumet le formulaire avec des données valides, les données soumises sont de nouveau écrites dans le formulaire, mais cette fois, `isValid()` renvoie `true`. Vous avez maintenant la possibilité d'effectuer certaines actions en utilisant l'objet `$employe` (par exemple en le persistant dans la base de données) avant de rediriger l'utilisateur vers une autre page.

Amusons nous...

Vous allez persister votre nouvel employé dans la bd.... Rajoutez le code suivant dans votre contrôleur :

```
use Symfony\Bridge\Doctrine\RegistryInterface;
```

Pour pouvoir travailler avec les méthodes de doctrine

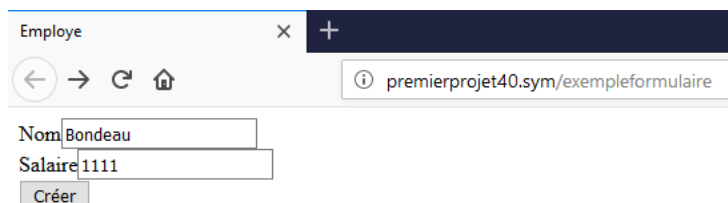
```
public function exempleFormulaire(Request $request, RegistryInterface $doctrine) {
```

Pour pouvoir utiliser doctrine en tant que service.

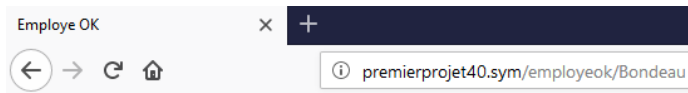
```
if ($form->isSubmitted() && $form->isValid()) {
    $employe = $form->getData();
    $e = $doctrine->getManager();
    $e->persist($employe);
    $e->flush();
}
```

La méthode persist va bufferiser la command de persistance,
La méthode flush va persister les commandes bufferisées.

Testez l'URL, saisissez des valeurs :

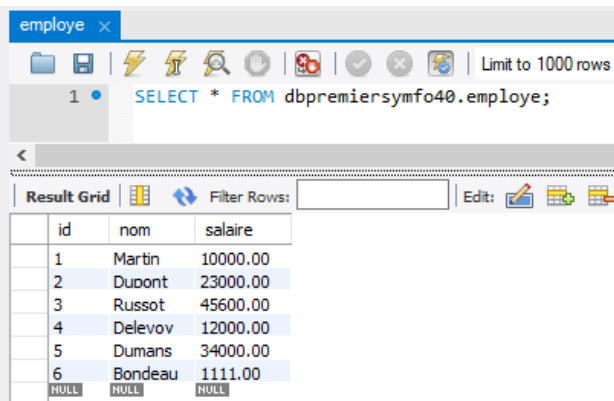


Et vous arriverez au formulaire suivant :



Vous avez saisi Bondeau

Vérifiez dans la BD :



	id	nom	salaire
1	1	Martin	10000.00
2	2	Dupont	23000.00
3	3	Russot	45600.00
4	4	Delevov	12000.00
5	5	Dumans	34000.00
6	6	Bondeau	1111.00

Notre objet \$employe a bien été persisté.

Partie 2 : OUI MAIS ...VALIDATION DU FORMULAIRE

Dans la section précédente on a exploité les données sans qu'elle soient validées. Par exemple le salaire doit être supérieur à 10000.

Appeler la méthode `$form->isValid()` revient à s'assurer que les données affectées à l'objet `$employe` sont valides (respectent les contraintes d'intégrité)

Vérifiez que le bundle validator est bien chargé :

Vendor/symfony/validator

Sinon, chargez-le avec la commande *composer require validator*

La validation est effectuée en ajoutant un ensemble de règles (appelées contraintes) à une classe. Pour voir cela dans l'action, ajoutez des contraintes de validation.

Grâce à HTML5, de nombreux navigateurs peuvent appliquer nativement certaines contraintes de validation côté client. La validation la plus courante est activée en affichant un attribut requis dans les champs obligatoires. Pour les navigateurs prenant en charge HTML5, un message du navigateur natif s'affiche si l'utilisateur tente de soumettre le formulaire avec ce champ vide.

Les formulaires tirent pleinement parti de cette nouvelle fonctionnalité en ajoutant des attributs HTML sensibles qui déclenchent la validation. La validation côté client peut cependant être désactivée en ajoutant l'attribut `novalidate` à la balise de formulaire ou `formnovalidate` à la balise `submit`.

Ceci est particulièrement utile lorsque vous souhaitez tester vos contraintes de validation côté serveur, mais que votre navigateur les empêche, par exemple, de soumettre des champs vides.

Vous allez donc modifier l'entity `Employe` :

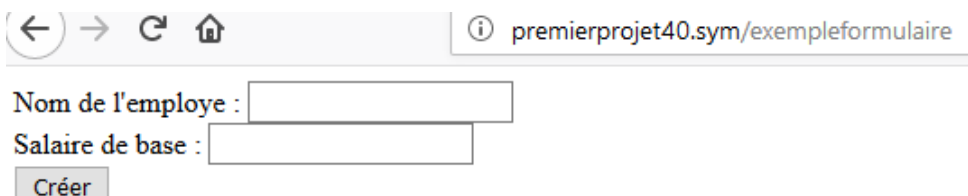
Par exemple, on va s'assurer ici que le nom n'est pas vide et que le salaire est bien un numérique supérieur ou égal à 10000

```
private $id;

/**
 * @ORM\Column(type="string", length=50)
 * @Assert\NotBlank()
 */
private $nom;

/**
 * @ORM\Column(type="decimal", precision=9, scale=2)
 * @Assert\NotBlank()
 * @Assert\Type(type="float", message = "La valeur {{ value }} doit être de type {{ type }}")
 * @Assert\Range(
 *     min = 10000,
 *     minMessage = "Le salaire doit au moins être égal à 10000"
 * )
 */
private $salaire;
```

Testez l'url : <http://premierprojet40.sym/exempleformulaire>



premierprojet40.sym/exempleformulaire

Nom de l'employé :

Salaire de base :

Essayez de soumettre le formulaire sans rien saisir :

Nom de l'employé :

Salaire de base :

Veuillez compléter ce champ.

Saisissez un salaire inférieur à 10000 :

Nom de l'employé :

Salaire de base :

- Le salaire doit au moins être égal à 10000

... Bon ce n'est pas beau, mais on pourra faire mieux avec les styles !!!



Assurez-vous que les données ont été persistées dans la base de données lorsque le formulaire est valide !!!

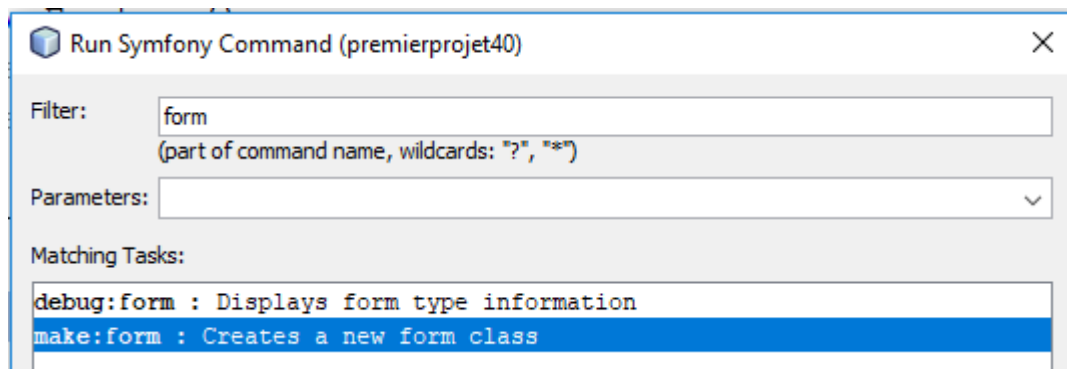
Partie 3 : TRAVAILLER AVEC DES CLASSES DE TYPE FORM

On vient de créer un formulaire directement dans un contrôleur.

On va maintenant mettre en place un formulaire indépendant du contrôleur, ce qui va le rendre indépendant et donc qui pourra être appelé depuis n'importe quel endroit de l'application.

On va passer par la commande `make:form`

La classe s'appellera **EmployeeType** Le nom du formulaire sera toujours *nomEntityType*



Répondez aux 2 questions posées :

- ✓ Le nom du formulaire
- ✓ Le nom de l'entity rattachée

```
"C:\wamp64\bin\php\php7.1.9\php.exe" "T:\Wampsites\CoursSymfony\premierprojet40\bin\console" "--ansi" "make:form"
```

```
The name of the form class (e.g. AgreeableChefType):
```

```
> EmployeeType
```

```
The name of Entity or custom model class that the new form will be bound to (empty for none):
```

```
> Employee
```

```
created: src/Form/EmployeeType.php
```

Success!

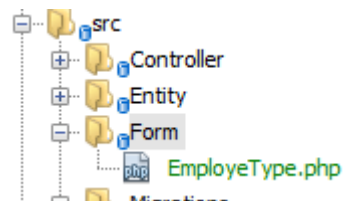
```
Next: Add fields to your form and start using it.
```

```
Find the documentation at https://symfony.com/doc/current/forms.html
```

```
Done.
```

```
|
```

Un nouveau dossier (Form) et un nouveau fichier formulaire ont été créés.



Le fichier contient la classe EmployeeType :

```
<?php

namespace App\Form;

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

class EmployeeType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nom')
            ->add('salaire')
        ;
    }

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

Remarquez la méthode buildForm

Pour utiliser le formulaire, vous allez rajouter l'action suivante dans le contrôleur

FormulaireController :

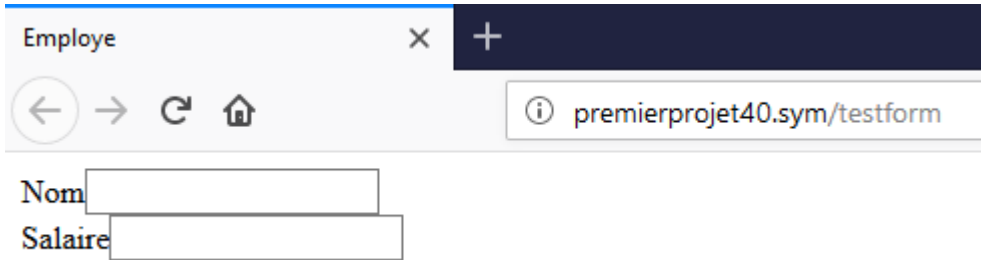
Commencez par rajouter l'espace de noms :

```
use App\Form\EmployeeType;

public function testForm(Request $request, RegistryInterface $doctrine) {
    $employee = new Employee();
    $form = $this->createForm(EmployeeType::class, $employee);
    return $this->render('formulaire/exempleemployee.html.twig', array(
        'form' => $form->createView()));
}
```

Et testez avec l'URL : <http://premierprojet40.sym/testform>

Vous obtenez ceci :



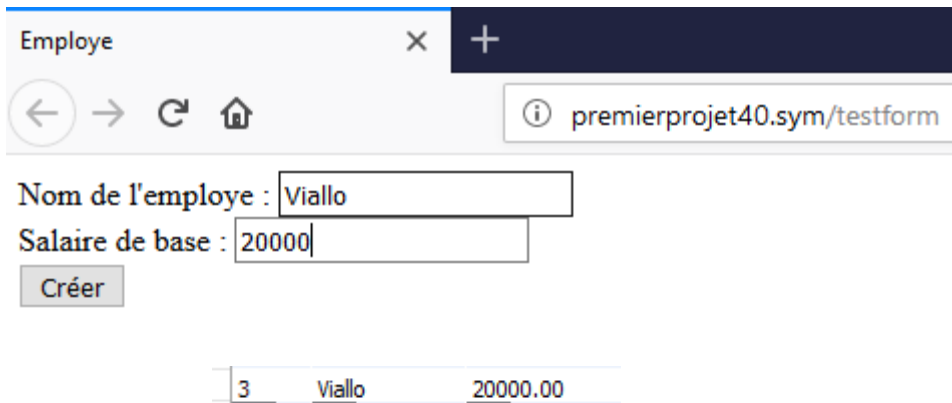
The screenshot shows a web browser window with a tab titled 'Employee'. The address bar shows 'premierprojet40.sym/testform'. The form has two input fields: 'Nom' and 'Salaire'.

Je vous laisse modifier le formulaire pour obtenir ceci : <http://premierprojet40.sym/testform>



The screenshot shows the modified form. The labels are 'Nom de l'employe' and 'Salaire de base'. There is a 'Créer' button at the bottom left.

Et créer l'employe Viallo qui gagne 2000



The screenshot shows the form with the values 'Viallo' and '20000' entered. Below the form, a table snippet is visible:

3	Viallo	20000.00
---	--------	----------

Chaque formulaire doit connaître le nom de la classe qui contient les données sous-jacentes (par exemple l'entity `Employee`).
Habituellement, cela correspond au deuxième argument de la méthode `createForm()` (c'est-à-dire `$employe`).
Plus tard, lorsque l'on fera des sous-formulaires, cela ne sera plus suffisant.
Il est donc recommandé d'appliquer les règles de bonnes pratiques et de spécifier explicitement l'option `data_class` pour obtenir ceci :

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

Vous remarquerez qu'en passant par la commande `make:form`, la méthode est ajoutée par défaut. Vous la rajouterez si vous créez manuellement votre Form.

Faire apparaître des champs dans le formulaire qui ne sont pas des attributs de la classe :

Lorsque le formulaire est mappé sur les attributs de la classe, une exception sera levée pour chaque champ du formulaire qui n'existent pas sur l'objet mappé.

Dans les cas où vous avez besoin de champs supplémentaires dans le formulaire (par exemple: une case à cocher «Êtes-vous d'accord avec ces termes») qui ne sera pas mappée à l'objet sous-jacent, vous devez définir l'option mappée sur `false`:

On peut accéder dans le contrôleur à ces données par l'instruction :

```
$form->get('nomChamp')->getData();
```

On peut aussi directement modifier la valeur du champ:

```
$form->get(nomChamp)->setData("valeur");
```

Vous allez rajouter une case à cocher demandant si l'employé a agréé les termes du contrat :

```
public function buildForm(FormBuilderInterface $builder, array $options) {
    $builder
        ->add('nom', TextType::class, array('label' => "Nom de l'employé : "))
        ->add('salaire', NumberType::class, array('label' => 'Salaire de base : '))
        ->add('agreer', CheckboxType::class, array('mapped' => false,
                                                'label'=>"Agréez-vous ces termes",
                                                'required' => false))
        ->add('save', SubmitType::class, array('label' => 'Créer'));
}
```

Vous remarquerez que vous avez l'option `required` qui vaut `false` ce qui permet à l'utilisateur de ne pas Saisir la case à cocher s'il ne le veut pas.



Attention, vous aurez sûrement des choses à rajouter en tête de classe (use ???)

Et dans le contrôleur, vous allez tester la valeur de la case à cocher lors du retour du formulaire :

```
$e->flush();
$mess= "en attente d'agrément";
if ($form->get('agreer')->getData()) {
    $mess="il a agréé";
}
return $this->redirectToRoute('employee_ok', array("nom" => $employee->getNom(),
                                                "message"=>$mess));
```

Quand la redirection se fera, l'url aura la forma suivante :

<http://premierprojet40.sym/employeeok/Viallo?message=il%20a%20agr%C3%A9%C3%A9>

Vous remarquez que le message passe en GET :

`employeeok/Viallo?message=il%20a%20agr%C3%A9%C3%A9`

Il faudra donc modifier le formulaire employeOK.html.twig pour

- ✓ Tester si la variable message existe en GET
- ✓ Afficher sa valeur

Vous allez donc rajouter le code suivant à votre formulaire employeOK.html.twig :

```
<H1>Vous avez saisi {{ nom }} </H1>
{% if app.request.query.get("message") is defined %}
    <h2>{{app.request.query.get("message")}}</h2>
{% endif %}
```

Il ne reste plus qu'à tester :

Et après avoir cliqué sur le bouton créer :

Si on teste sans cocher la case :

Vous avez saisi Viallo

en attente d'agrément

... Allez vérifier si les enregistrements ont été persistés dans la base de données !

Lors de la création de formulaires, gardez à l'esprit que :



- ✓ le premier objectif d'un formulaire est de traduire les données d'un objet (ex : employe) en un formulaire HTML afin que l'utilisateur puisse modifier ces données.
- ✓ Le second objectif d'un formulaire est de prendre les données soumises par l'utilisateur et de les ré-appliquer à l'objet.

Partie 4 : CHANGER L'ACTION D'UN FORMULAIRE

Vous l'avez vu, par défaut, un formulaire sera envoyé via une requête HTTP POST à la même URL sous laquelle le formulaire a été rendu.

Parfois, il sera nécessaire de ne pas revenir sur la même action que celle qui a généré le formulaire.

Il existe 2 façons de faire

Par le FormBuilder, vous pouvez utiliser `setAction ()` et `setMethod ()`:

```
$form = $this->createForm(EmployeType::class, $employe, array( 'action' => $this->generateUrl('un_employe'),
                                                             'method' => 'POST'));
```

Directement dans le formulaire :

```
{% block body %}
    {{ form_start(form, {'action': path('un_employe'), 'method': 'POST'}) }}
{% endblock %}
```

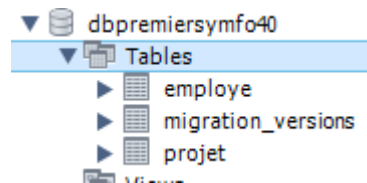
Partie 5 : SOUS-FORMULAIRES

Souvent, un formulaire devra intégrer un sous formulaire.
Par exemple, un employé pourra travailler sur un projet.

Créez avec la commande `make:entity` une entité `Projet` qui contient les champs :

- ✓ `NomProjet` : chaîne longueur 80 not null
- ✓ `Datedebut` : date not null

Mettez la base de données à jour, vous devriez avoir ceci :



Rajoutez l'attribut `projet` dans la classe `Employe` :

```
/**
 * @Assert\Type(type="App\Entity\Projet")
 * @Assert\Valid()
 */
private $projet;
```

Ainsi que les getter/setter :

```
public function getProjet()
{
    return $this->projet;
}

public function setProjet(Projet $projet = null)
{
    $this->projet = $projet;
}
```

Puis générez un formulaire type avec la commande `make:form`

Voici le code généré :

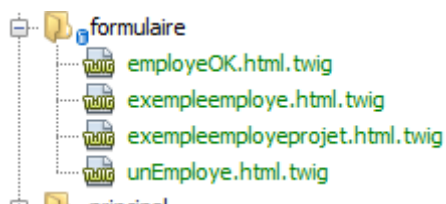
```
namespace App\Form;

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

class ProjetType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nomProjet')
            ->add('dateDebut')
        ;
    }

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

Par copier/coller, créez le formulaire exempleemployeprojet.html.twig :



Et rajoutez-y le code suivant :

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

{% block title %}Employe{% endblock %}
{% block body %}
    {{ form_start(form) }}
    {{ form_row(form.nom) }}
    {{ form_row(form.salaire) }}
    <p>Projet</p>
    {{ form_row(form.projet) }}
    {{ form_end(form) }}
{% endblock %}
```

Créez-vous une nouvelle action dans le contrôleur FormulaireController :

```
public function sousForm(Request $request) {
    $employe = new Employe();
    $form = $this->createForm(EmployeType::class, $employe);

    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $employe = $form->getData();
        $mess = "en attente d'agrément";
        if ($form->get('agreer')->getData()) {
            $mess = "il a agréé";
        }
        return $this->redirectToRoute('employe_ok', array("nom" => $employe->getNom(),
            "message" => $mess));
    }

    return $this->render('formulaire/exempleemployeprojet.html.twig', array(
        'form' => $form->createView()));
}
```

Vous remarquerez que l'on a supprimé les actions de mise à jour de la BD, on les retrouvera plus tard dans doctrine



Testez l'URL : <http://premierprojet40.sym/sousform>
Et vous obtiendrez ceci :

Nom de l'employé :

Salaire de base :

Agréez-vous ces termes ☐

Projet

Nom projet

Date debut

Jan ▼ 1 ▼ 2013 ▼

Créer



Exercice:

Modifiez le formulaire employeOK.html.twig pour afficher le nom du projet si il existe. Vous aurez également à modifier l'appel à ce formulaire.

Vous pouvez également intégrer une collection de formulaires dans un formulaire. Ceci est fait en utilisant le type de champ de collection.

Exemple :

← ⓘ symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerCoursTheme

Libellé du cours :

Nombre de jours :

☐ Développement web

☐ Développement mobile

☐ Développement système

☐ Système et réseau

☐ Sécurité informatique

Enregistrer



Exercice:

Mettez vos formulaires en forma grâce à bootstrap !!!

<https://symfony.com/doc/current/form/bootstrap4.html>

Bravo pour votre travail, il ne reste plus maintenant qu'à tout revoir dans les détails

