

Formulaires en Symfony

la documentation officielle ici : <https://symfony.com/doc/current/forms.html>

Objectifs pédagogiques :

A la fin de cette séance, vous serez en mesure de construire et de gérer des formulaires en Symfony

Introduction

Le composant Form de Symfony est une bibliothèque puissante qui facilite la création, la manipulation et la validation des formulaires dans une application Symfony. Il offre une abstraction efficace des formulaires HTML classiques et permet une gestion centralisée des données de formulaire.

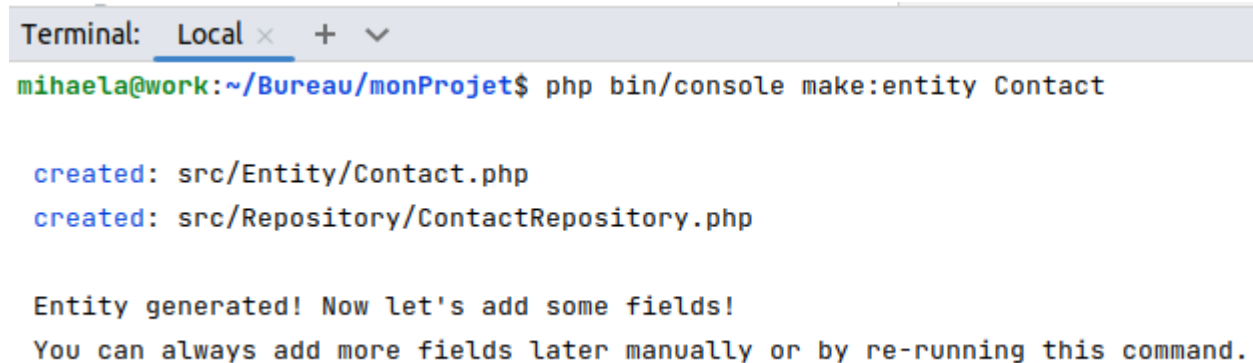
Prérequis

Avant de commencer, assurez-vous d'avoir installé Symfony sur votre machine et d'avoir créé un projet Symfony fonctionnel. Vous pouvez utiliser la commande `composer require symfony/form` pour ajouter le composant Form à votre projet s'il n'existe pas.

Nous allons reprendre le projet Symfony "monProjet" où nous avons créé nos premiers contrôleurs, entités et vues!

Associer un formulaire à une entité

Commençons par créer une entité Contact



```
Terminal: Local x + v
mihaela@work:~/Bureau/monProjet$ php bin/console make:entity Contact

created: src/Entity/Contact.php
created: src/Repository/ContactRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.
```

Maintenant, ajoutons trois propriétés : objet, email et message:

Terminal: Local x + v

created: src/Entity/Contact.php

created: src/Repository/ContactRepository.php

Entity generated! Now let's add some fields!

You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):

> objet

Field type (enter ? to see all types) [string]:

>

Field length [255]:

>

Can this field be null in the database (nullable) (yes/no) [no]:

>

updated: src/Entity/Contact.php

Add another property? Enter the property name (or press <return> to stop adding fields):

> email

Field type (enter ? to see all types) [string]:

>

Field length [255]:

>

Can this field be null in the database (nullable) (yes/no) [no]:

>

updated: src/Entity/Contact.php

Add another property? Enter the property name (or press <return> to stop adding fields):

> message

Field type (enter ? to see all types) [string]:

> ?

Notez le signe d'interrogation à la fin du script : Symfony nous suggérait une propriété de type chaîne de caractères (string), mais nous voudrions avoir un champ qui peut

faire plus de 255 caractères - un équivalent du `textarea` en HTML! Quand vous ne savez pas quels types sont disponibles, vous devez entrer `?` et vous aurez la liste de tous les types disponibles! On va choisir le type `text` pour notre propriété `message` :

Main Types

- * string
- * text
- * boolean
- * integer (or smallint, bigint)
- * float

Relationships/Associations

- * relation (a wizard 🧙 will help you build the relation)
- * ManyToOne
- * OneToMany
- * ManyToMany
- * OneToOne

Array/Object Types

- * array (or simple_array)
- * json
- * object
- * binary
- * blob

Date/Time Types

- * datetime (or datetime_immutable)
- * datetimetz (or datetimetz_immutable)
- * date (or date_immutable)
- * time (or time_immutable)
- * dateinterval

Other Types

- * ascii_string
- * decimal
- * guid

Field type (enter ? to see all types) [string]:

> text

Can this field be null in the database (nullable) (yes/no) [no]:

> yes

Mettons à jour notre base de donnée :

php bin/console d:s:u --force

Créons maintenant un formulaire de contact associé à cette entité qui va nous permettre d'insérer dans la base de données un tout nouveau message

```
mihaela@work:~/Bureau/monProjet$ php bin/console make:form
```

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

```
> ContactForm
```

```
The name of Entity or fully qualified model class name that the new form will be bound to:
```

```
> Contact
```

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

```
Success!
```

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

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

```
mihaela@work:~/Bureau/monProjet$
```

Regardons maintenant ce que le fichier `ContactFormType.php` contient:

```
File Edit View Navigate Code Refactor Run Tools Git Window Help
monProjet > src > Form > ContactFormType.php > Contact

Project
  monProjet ~/Bureau/monProjet
    bin
    config
    migrations
    public
    src
      Controller
      DataFixtures
      Entity
      Form
        ContactFormType.php
      Repository
        Kernel.php
      templates
      tests
      translations
      var
      vendor
      .env
      .env.local
      .env.test
      .gitignore
      composer.json
      composer.lock
      docker-compose.override.yml
      docker-compose.yml
      phpunit.xml.dist
      symfony.lock
    External Libraries
    Scratches and Consoles

security.yaml x index.html.twig x Artist.php x

1 <?php
2
3 namespace App\Form;
4
5 use App\Entity>Contact;
6 use Symfony\Component\Form\AbstractType;
7 use Symfony\Component\Form\FormBuilderInterface;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9
10 class ContactFormType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array $options)
13     {
14         $builder
15             ->add( child: 'objet')
16             ->add( child: 'email')
17             ->add( child: 'message')
18         ;
19     }
20
21     public function configureOptions(OptionsResolver $resolver)
22     {
23         $resolver->setDefaults([
24             'data_class' => Contact::class,
25         ]);
26     }
27 }
28
```

Explication :

Dans cet exemple, nous utilisons la classe `ContactFormType` pour créer notre formulaire de contact. Nous ajoutons trois champs : un champ `objet`, un champ `email` et un champ `message`.

Chaque champ est créé à l'aide de la méthode `add()` de l'objet `FormBuilderInterface`.

La méthode `add()` prend trois arguments : le nom du champ, le type de champ et un tableau d'options facultatives. Les options nous permettent de personnaliser le rendu et le comportement de

chaque champ. Par exemple, nous utilisons l'option 'label' pour spécifier l'étiquette associée à chaque champ.

Ajoutons une ligne supplémentaire à notre formulaire et rendons en même temps le champ `message` optionnel :

```
namespace App\Form;

use App\Entity>Contact;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ContactFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('objet')
            ->add('email')

            //On a rajouté un label et on a rendu le champ optionnel en
            // donnant la valeur false à l'attribut required
            ->add('message', TextareaType::class, [
                'label' => 'Votre message',
                'required' => false
            ])
            ->add('save', SubmitType::class, [
                'label' => 'Envoyer le message'])
        ;
    }

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

Explication : Une des options les plus utilisées sur les formulaires est `required`, qui est par défaut à "true" - c'est-à-dire que tous les champs du formulaire sont obligatoires! Si vous voulez qu'un champ ne le soit pas, il faut passer `required` à `false`.

Nous avons aussi ajouté un champ dont le nom est `save`, le type est `submit` et dans le tableau d'options, nous avons ajouté le label de l'input ('Envoyer le message'). Vous remarquerez la présence d'une classe `SubmitType` : c'est l'équivalent d'un `<input type="submit">` en HTML!

L'une des principales différences entre un formulaire **Symfony** et un formulaire **HTML** traditionnel est que le formulaire **Symfony** est configuré à l'aide de classes **PHP** plutôt qu'avec du code **HTML** brut.

Si dans un formulaire **HTML** nous avons un `<form>` et plusieurs `form fields`, en **Symfony** TOUT est un **FormType**.

Symfony propose un large éventail de types de champ préconstruits, qui correspondent aux différents types de données que vous pouvez utiliser dans un formulaire. Voici quelques-uns des types de champ les plus couramment utilisés :

TextType : Champ de texte simple.

TextareaType : Champ de texte multiligne.

EmailType : Champ pour les adresses e-mail.

IntegerType : Champ pour les nombres entiers.

DateType : Champ pour les dates.

ChoiceType : Champ pour les listes déroulantes ou les cases à cocher.

FileType : Champ pour les fichiers téléchargés.

HiddenType : Champ de caché. **EntityType** : Champ pour sélectionner une entité d'une relation.

Il existe de nombreux autres types de champ disponibles dans **Symfony**. Sachez que vous pouvez également créer vos propres types de champ personnalisés en étendant la classe **AbstractType** et en les configurant selon vos besoins spécifiques.

L'utilisation des types de champ dans **Symfony** permet de simplifier la création et la manipulation des formulaires, car ils gèrent automatiquement le rendu du champ et la conversion des données soumises.

Vous pouvez utiliser les types de champ de **Symfony** en utilisant les classes correspondantes du composant **Form**, telles que `TextType::class`, `EmailType::class`, etc., lors de la construction de votre formulaire.

Configurer les options du formulaire

La méthode `configureOptions` dans un **FormType** sert à spécifier des options supplémentaires pour votre formulaire :

```
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ContactFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // Construction du formulaire...
    }
}
```



```

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults([
        // Options par défaut pour le formulaire
        'data_class' => Contact::class,
    ]);
}
}

```

Explication : Dans la méthode `configureOptions` nous utilisons l'objet `$resolver` pour définir les options par défaut de notre formulaire. Dans ce cas, nous spécifions que l'entité (la classe de données) associée au formulaire est `Contact` (`Contact::class`). Cela indique à `Symfony` quelle classe utiliser pour lier les données du formulaire.

La méthode `configureOptions` permet de spécifier d'autres options pour notre formulaire, telles que des options de rendu, des options de validation ou toute autre option personnalisée que nous souhaitons utiliser dans le formulaire.

N'oubliez pas d'importer la classe `OptionsResolver` depuis le composant `Form` de `Symfony` pour pouvoir utiliser la méthode `configureOptions`.

CSRF token

Le **CSRF** (Cross-Site Request Forgery) est une attaque courante dans laquelle un utilisateur malveillant peut exploiter la confiance d'un utilisateur authentifié pour effectuer des actions non autorisées en son nom. Pour se prémunir contre cette attaque, `Symfony` propose une protection **CSRF** intégrée, qui utilise un **jeton CSRF (CSRF token)** pour vérifier l'origine légitime des soumissions de formulaires.

Lorsque vous utilisez le composant `Form` de `Symfony`, par défaut, un champ de jeton **CSRF** est automatiquement ajouté à vos formulaires. Ce champ contient un jeton unique généré pour chaque soumission de formulaire et est utilisé pour vérifier l'intégrité des données soumises.

Lorsque le formulaire est rendu dans la vue, le champ **CSRF** est généré automatiquement à l'aide d'une balise `<input>` avec un nom et une valeur spécifiques. Lorsque le formulaire est soumis, `Symfony` vérifie que la valeur du jeton **CSRF** correspond à celle attendue, empêchant ainsi les attaques **CSRF**.

Pour générer et vérifier le jeton **CSRF**, `Symfony` utilise le composant `CsrfTokenManager`. Ce composant est configuré par défaut dans le framework `Symfony`, de sorte que vous n'avez pas à vous soucier de sa mise en place. Cependant, vous pouvez vérifier dans votre Contrôleur la présence du champ **CSRF** lors de la soumission du formulaire.

[!NOTE] Par contre, si vous utilisez un simple formulaire **HTML** (comme dans vos projets en **PHP**), il faut créer le token **CSRF** à la main et vérifier sa valeur dans le contrôleur qui gère ce formulaire.

Le formulaire **HTML** :

```

<form action="{{ url('admin_post_delete', { id: post.id }) }}" method="post">
    {# l'argument de la méthode csrf_token() est une chaîne de caractère aléatoire qui sert à
    générer le token #}

```

```

        <input type="hidden" name="token" value="{{ csrf_token('delete-item') }}" />

        <button type="submit">Delete item</button>
    </form>

```

Le contrôleur:

```

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
// ...

public function delete(Request $request): Response
{
    $submittedToken = $request->request->get('token');

    // 'delete-item' est identique à la valeur utilisée dans la vue pour générer le token
    if ($this->isCsrfTokenValid('delete-item', $submittedToken)) {
        // ... le corps de la fonction
    }
}

```

Lorsque le formulaire est soumis, Symfony vérifie automatiquement le jeton CSRF. Si le jeton CSRF n'est pas valide ou manquant, une exception `InvalidCsrfTokenException` sera levée.

Affichage du formulaire

Une fois notre formulaire créé, nous devons le rendre dans notre vue et traiter les données soumises par l'utilisateur. Nous allons créer un contrôleur `ContactController` et à l'intérieur une fonction `index`, qui se déclenche sur la route `/contact`. L'affichage d'abord:

```

<?php

namespace App\Controller;

use App\Form\ContactFormType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ContactController extends AbstractController
{
    #[Route('/contact', name: 'app_contact')]
    public function index(Request $request): Response
    {
        $form = $this->createForm(ContactFormType::class);
        // ...
        // A partir de la version 6.2 de Symfony, on n'est plus obligé d'écrire
        // $form->createView(), il suffit de passer l'instance de FormInterface
        // à la méthode render

        return $this->render('contact/index.html.twig', [
            'form' => $form->createView(),
            'form' => $form
        ]);
    }
}

```

```

    });
}
}

```

Explication : Dans ce contrôleur, nous utilisons la méthode `createForm()` pour créer une instance de notre formulaire `ContactFormType`. Ce formulaire sera affiché à l'aide de la méthode `render()` dans la vue `index.html.twig` qui se trouve dans le répertoire `contact`. On va reprendre ce fichier et on le modifie de la manière suivante :

```

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

{% block body %}

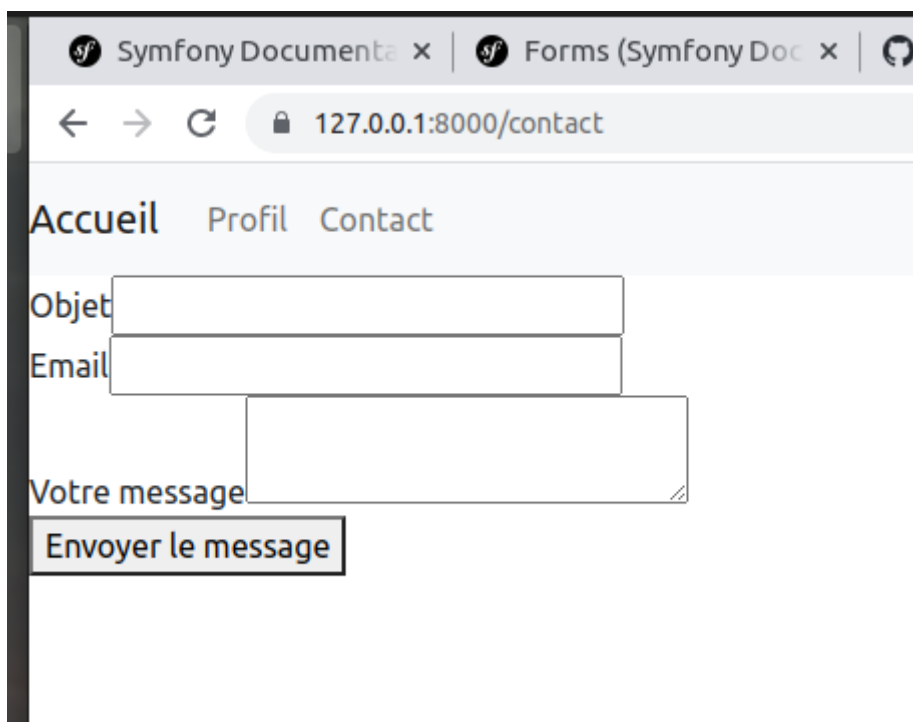
<div>

    {{ form(form) }}

</div>
{% endblock %}

```

La fonction `form()` de Twig suffit pour afficher TOUT le formulaire (que nous lui avons passé en paramètre)!



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/contact'. The page has a navigation bar with links 'Accueil', 'Profil', and 'Contact'. Below the navigation bar, there is a contact form with three input fields: 'Objet', 'Email', and 'Votre message'. The 'Envoyer le message' button is highlighted with a red box.

Voyons le résultat :

Pas très beau sans style, mais les champs qu'on avait sur notre `ContactFormType` sont bien là : un champ text, un champ email et un `textarea`, plus le bouton `submit` ayant comme label `Envoyer votre message`!

On va le reprendre et on va utiliser d'autres fonctions Twig - `form_start()`, `form_end()`, `form_errors()` et `form_row()` pour afficher chaque partie du formulaire individuellement. Le but est de pouvoir ajouter d'autres attributs aux champs du formulaire :

```

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

```

```

{% block body %}

<div class="row text-center">

    {{ form_start(form) }}
    <div class="my-custom-class-for-errors">
        {{ form_errors(form) }}
    </div>

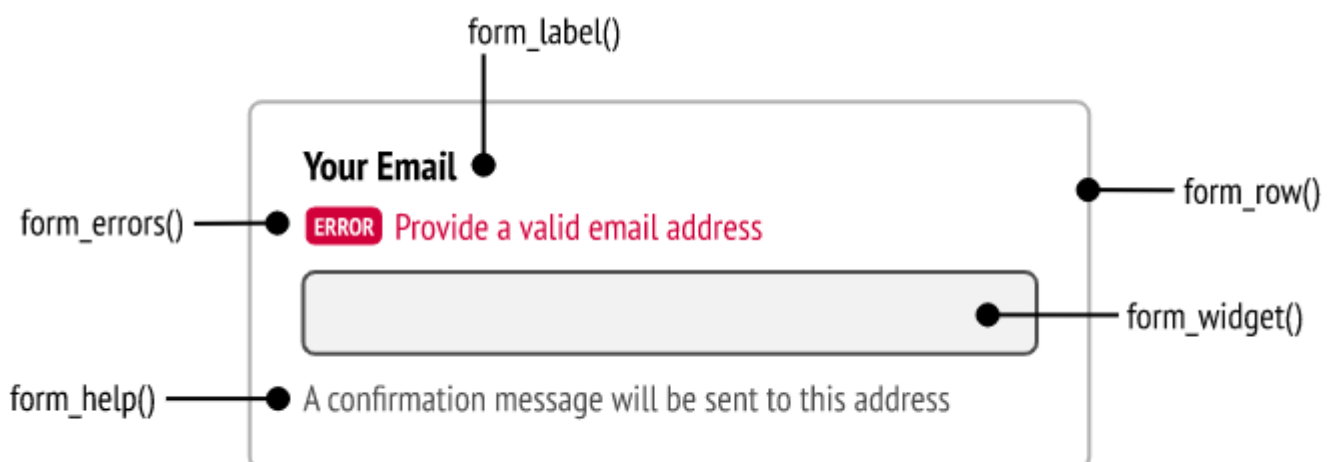
    <div class="row">
        <div class="col-9 mx-auto mb-2 mt-4">
            {{ form_row(form.objet) }}
        </div>
        <div class="col-9 mx-auto mb-2">
            {{ form_row(form.email) }}
        </div>
        <div class="col-9 mx-auto mb-2">
            {{ form_row(form.message) }}
        </div>

    </div>
    {{ form_end(form) }}
</div>
{% endblock %}

```

Explication: Au lieu de se servir de la fonction `form()`, on a utilisé d'autres fonctions pour afficher chaque champ du formulaire individuellement (avec `form_row`). Mais on peut aller encore plus loin et au lieu d'utiliser `form_row()` pour afficher dans un seul bloc l'input + le label + les erreurs éventuelles, on peut utiliser les fonctions `form_label()` pour le label, `form_widget()` pour l'input, `form_help()` pour les messages d'aide à la saisie et `form_errors()` pour les erreurs. Ainsi, on peut les personnaliser comme on veut!

Voici la structure d'un champ de formulaire Symfony :



Maintenant, ajoutons une classe Bootstrap à l'input objet du formulaire et changeons son libellé :

```

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

{% block body %}

<div class="row text-center">

    {{ form_start(form) }}
    <div class="my-custom-class-for-errors">
        {{ form_errors(form) }}
    </div>

    <div class="row">
        ...
        {{ form_label(form.objet, "Mon libellé a été modifié") }}
        {{ form_widget(form.objet, {'attr': {'class': 'btn-light'}}) }}
        {{ form_errors(form.objet) }}
        ...
    </div>
    {{ form_end(form) }}
</div>
{% endblock %}

```

Resultat:

The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/contact'. The page has a light blue navigation bar with links 'Accueil', 'Profil', and 'Contact'. The main content area is white and contains a form. At the top of the form, it says 'Mon libellé a été modifié' next to a text input field. Below that is an 'Email' label next to another text input field. Further down is a 'Votre message' label next to a larger text area. At the bottom right of the form is a button labeled 'Envoyer le message'.

Traitement du formulaire

Revenons à notre contrôleur et regardons les autres fonctions :

```

use App\Entity>Contact;
use App\Form\DemoFormType;
use App\Form>ContactFormType;

```

```

use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class ContactController extends AbstractController
{
    #[Route('/contact', name: 'app_contact')]
    public function index(Request $request, EntityManagerInterface $entityManager): Response
    {
        $form = $this->createForm(ContactFormType::class);
        $form->handleRequest($request);

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

            //on crée une instance de Contact
            $message = new Contact();
            // Traitement des données du formulaire
            $data = $form->getData();
            //on stocke les données récupérées dans la variable $message
            $message = $data;

            $entityManager->persist($message);
            $entityManager->flush();

            // Redirection vers accueil
            return $this->redirectToRoute('app_accueil');
        }

        return $this->render('contact/index.html.twig', [
            'form' => $form->createView(),
            'form' => $form
        ]);
    }
}

```

- ⑩ nous utilisons la méthode `handleRequest()` pour traiter la requête HTTP actuelle et valider les données soumises.
- ⑩ `if ($form->isSubmitted())` - Si le formulaire est soumis et
- ⑩ `&& $form->isValid()` si le formulaire est valide, nous pouvons accéder aux données du formulaire à l'aide de la méthode `getData()`.

À ce stade, nous pouvons effectuer les actions souhaitées, la création d'un nouvel objet de type `Contact ($message)`. Après l'avoir persisté le nouveau message dans la base de données, nous pouvons rediriger l'utilisateur vers une autre page.

Si le formulaire n'est pas soumis ou n'est pas valide, nous rendons le formulaire dans notre vue `index.html.twig` - qui se trouve dans le répertoire `contact` des templates - à l'aide de la méthode `render()`. Nous utilisons également la méthode `createView()` pour générer une représentation du formulaire prête à être affichée dans notre template.

A savoir : Lorsque vous créez un formulaire dans votre contrôleur, Symfony détecte automatiquement la route actuelle et utilise cette route comme action par défaut pour le formulaire. Cela signifie que lorsque le formulaire est soumis, il sera envoyé à la même route où le formulaire a été affiché. Si vous souhaitez spécifier une action différente pour le formulaire, vous pouvez le faire en utilisant l'option `action` définir explicitement une route spécifique :

```
$form = $this->createForm(ContactFormType::class, null, [
    'action' => $this->generateUrl('ma_route_spécifique'),
]);
```

Validation des données

Le composant Form de Symfony intègre une fonctionnalité puissante de validation des données. Il peut valider automatiquement les champs selon des règles prédéfinies (par exemple, vérifier que l'e-mail est valide). Vous pouvez également définir vos propres contraintes de validation personnalisées!

Pour ajouter des contraintes de validation à notre formulaire de contact, nous pouvons utiliser les annotations de validation fournies par Symfony, telles que `#[Assert\Email]` pour vérifier l'adresse e-mail.

Par exemple :

```
use Symfony\Component\Validator\Constraints as Assert;

class Contact
{
    #[Assert\Email(message: "Veuillez saisir une adresse e-mail valide.")]
    private $email;

    // ...
}
```

Ces attributs peuvent être utilisés de manière similaire pour spécifier d'autres contraintes de validation telles que `NotNull`, `Length`, `Regex`, etc. Vous pouvez les combiner et les personnaliser selon vos besoins.

Important! N'oubliez pas d'importer la classe `Assert` depuis le composant `Validator` de Symfony pour pouvoir utiliser ces attributs de validation.

Et si le formulaire n'est associé à aucune entité ?

Si le formulaire n'est pas associé à une entité spécifique, vous pouvez passer `null` en tant que valeur de l'option `data_class`. Cela indique à Symfony que le formulaire n'est pas lié à une classe de données particulière.

Vous pouvez spécifier cela dans la méthode `configureOptions` de votre classe de formulaire :

Exemple de formulaire non lié à une entité

```

class DemoFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('objet')
            ->add('email')
            ->add('message', TextareaType::class, [
                'label' => 'Votre message',
                'required' => false
            ])
            ->add('save', SubmitType::class, [
                'label' => 'Envoyer le message'])
    }

    public function configureOptions(OptionsResolver $resolver): void
    {
        $resolver->setDefaults([
            // Le formulaire n'est associé à aucune entité !!!
            // 'data_class' => Contact::class,
        ]);
    }
}

```

En spécifiant `data_class => null` dans les options par défaut, on indique à Symfony que le formulaire n'est pas lié à une classe spécifique. Cela signifie que les données du formulaire seront renvoyées sous forme de **tableau associatif** plutôt que d'objet.

[!Note] Si vous n'avez pas besoin de lier le formulaire à une classe de données ou si vous ne souhaitez pas traiter les données du formulaire dans un objet spécifique, vous pouvez laisser l'option `data_class` avec la valeur par défaut `null`. Il n'est pas nécessaire de le préciser explicitement.

En spécifiant `null` comme valeur de `data_class`, vous pouvez traiter les données du formulaire directement à partir du tableau renvoyé par la méthode `getData()` du formulaire.

Exemple d'un contrôleur créant un formulaire non lié à une entité

```

#[Route('/contactdemo', name: 'app_contactdemo')]
public function index_demo(Request $request, EntityManagerInterface $entityManager):
Response
{
    // Lors de la création du formulaire, nous pouvons spécifier les valeurs initiales des
champs
    $form = $this->createForm(DemoFormType::class, [
        "objet" => "Entrez un texte !!!",
        "email" => "",
        "message" => "",
    ]);
    $form->handleRequest($request);
}

```



```

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

            // On peut récupérer toutes données du formulaire sous la forme d'un tableau
associatif
            $data = $form->getData();
            dump($data);

            // Ou récupérer les champs un par un
            $nom = $form->get("objet")->getData();
            dd($nom);

            // Envoi de l'e-mail, sauvegarde en base de données, etc.

            // Redirection vers accueil
            return $this->redirectToRoute('app_accueil');
        }

        return $this->render('contact/index.html.twig', [
            'form' => $form
        ]);
    }
}

```

Thèmes pour les formulaires Symfony

Il est possible de préciser un ou plusieurs thèmes de template pour vos formulaires. Vous devez les configurer dans le fichier `config/packages/twig.yaml` :

```

# config/packages/twig.yaml
twig:
    form_themes: ['bootstrap_5_horizontal_layout.html.twig']
    # ...

```

Vous pouvez rajouter plusieurs thèmes dans ce tableau d'optionson, mais attention à l'ordre : chaque thème surpasse le précédent! Pensez à mettre les thèmes les plus importants à la fin de la liste!

Même si la plupart du temps vous appliquerez les thèmes des formulaires au niveau global (pour tous les formulaires existants dans votre projet), vous avez la possibilité d'appliquer un thème au cas par cas. Pour faire cela, vous pouvez utiliser la balise Twig `form_theme` que vous intégrerez dans la page qui affiche le formulaire:

```

{# this form theme will be applied only to the form of this template #}
{% form_theme form 'foundation_5_layout.html.twig' %}

{{ form_start(form) }}
    {# ... #}
{{ form_end(form) }}

```

La documentation Symfony sur les thèmes de formulaire : Styliser un formulaire en Symfony : https://symfony.com/doc/current/form/form_themes.html

DQL / QueryBuilder

Objectifs pédagogiques :

A la fin de cette séance vous serez en mesure d'écrire des requêtes avec Doctrine, en utilisant le Repository d'une entité, le Doctrine Query Langage et le QueryBuilder

Le Repository

Dans les chapitres précédents nous avons déjà créé des entités avec le `maker` de Symfony et nous avons vu qu'une autre classe était créée en même temps que l'entité - il s'agit bien d'un `Repository`. Un `Repository` est une classe dont la responsabilité est de gérer les requêtes sur l'entité correspondante.

Cette classe PHP dispose de plusieurs fonctions par défaut pour retrouver vos objets :

- ⑩ `find($id)` - recherche un objet par son `id`
- ⑩ `findOneBy(['une_ou_plusieurs_propriétés' => 'valeurs_de_ces_propriétés'])` - recherche un objet par une ou plusieurs propriétés
- ⑩ `findBy(['une_ou_plusieurs_propriétés' => 'valeurs_de_ces_propriétés'])` - recherche plusieurs objets en fonction d'une ou plusieurs conditions
- ⑩ `findAll()` - récupère tous les objets (un peu comme un `'SELECT *'` en SQL)

Le repository donne accès au gestionnaire d'entités (`EntityManager`) dont nous avons besoin pour insérer, modifier et supprimer nos objets (les fonctions `persist()` et `flush()` en particulier) !

Exemple d'utilisation d'un Repository dans un contrôleur

Pour comprendre l'utilité d'un repository et la manière dont on peut s'en servir dans un projet Symfony, nous allons reprendre notre `AccueilController` et écrire une fonction qui récupère tous les artistes de la base de données et les afficher sur la page d'accueil, ainsi que le nombre de disques pour chacun:

```
<?php
```

```
namespace App\Controller;
```

```
use App\Repository\ArtistRepository;
use App\Repository\DiscRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
```

```
class AccueilController extends AbstractController
{
```

```
    //On va avoir souvent besoin d'injecter les repositories de nos entités dans les
    contrôleurs et les services
```

```
    //Pour ne pas les injecter dans chaque fonction, on va les instancier UNE SEULE fois dans le
    constructeur de notre contrôleur:
```

```
    //N'oubliez pas d'importer vos repositories (les lignes "use..." en haut de la page)
```

```
    private $artistRepo;
    private $discRepo;
```

```

public function __construct(ArtistRepository $artistRepo, DiscRepository $discRepo)
{
    $this->artistRepo = $artistRepo;
    $this->discRepo = $discRepo;
}

#[Route('/accueil', name: 'app_accueil')]
public function index(): Response
{
    //on appelle la fonction `findAll()` du repository de la classe `Artist` afin de
    récupérer tous les artists de la base de données;

    $artistes = $this->artistRepo->findAll();

    return $this->render('accueil/index.html.twig', [
        'controller_name' => 'AccueilController',
        //on va envoyer à la vue notre variable qui stocke un tableau d'objets $artistes
        (c'est-à-dire tous les artistes trouvés dans la base de données)
        'artistes' => $artistes
    ]);
}
}

```

Reprenons la page `index.html.twig` qui se trouve dans `templates/accueil` et modifions-la de manière à ce qu'elle affiche nos artistes et le nombre de disques pour chaque artiste:

```

{% extends 'base.html.twig' %}
{% block body %}
    <div class="container">
        <div class="row mt-5">
            <div class="col-10 mx-auto">
                <h1>Nombre d'artistes trouvés : {{ artistes | length }}</h1>
                {% for artiste in artistes %}
                    <span class="d-block">
                        {{ artiste.name | upper }} - <b class="btn-primary p-1">{{ artiste.discs |
length }}</b> disques
                    </span>
                {% endfor %}
            </div>
        </div>
    </div>
{% endblock %}

```

Resultat :

Accueil Profil Contact

Nombre d'artistes

QUEENS OF THE STONE AGE - 1 disques

ROLLING STONES - 0 disques

NEIL YOUNG - 3 disques

Notes

- ⑩ Comme les entités Artist et Disc sont reliées par une relation, on peut accéder aux disques d'un artiste directement en écrivant `artist.discs`.

Notes concernant Twig

- ⑩ Pour afficher des variables dans Twig, on utilise les doubles accolades "moustaches" - `{{ artiste.name }}`
- ⑩ Pour écrire des conditions ou des boucles for, on utilise les accolades avec pourcentages - `{% for artist in artists %} ... {% endfor %}`.
- ⑩ Pour afficher la longueur d'un tableau (le nombre de disques par exemple) on utilise un filtre Twig: `length` (les filtres TWIG ont le signe `|` devant)

Mais bien que très utiles, les méthodes proposées par le Repository sont parfois insuffisantes... d'où l'intérêt d'utiliser un langage comme DQL.

Le DQL

DQL (Doctrine Query Language) est le langage utilisé par Doctrine pour écrire des requêtes. Il ressemble au SQL, mais, contrairement au SQL, il s'applique non pas sur une base de données mais sur des objets PHP (les classes de données ou entités - `Entity`).

Attention à ne pas confondre DQL et SQL! Si vous essayez d'utiliser des noms de tables et/ou des noms de colonnes ou de joindre des tables arbitraires dans une requête DQL vous aurez des erreurs, utilisez vos entités et les noms des propriétés de celles-ci!

L'utilisation du DQL va se faire dans une méthode du fichier `Repository` à partir de la méthode `createQuery()` de l'`EntityManager`.

Par exemple, si nous voulions récupérer tous les artistes de notre base de données dont le nom contient 'Neil', en DQL on devrait écrire :

```
// dans la classe ArtistRepository
public function getSomeArtists($name)
{
    //$name est un paramètre qui pour cet exemple a come valeur "Neil";
    $entityManager = $this->getEntityManager(); //on instancie l'entity manager

    $query = $entityManager->createQuery( //on crée la requête
        'SELECT a
        FROM App\Entity\Artist a
        WHERE a.name like :name'
    )->setParameter('name', '%'.$name.'%');

    // retourne un tableau d'objets de type Artist
    return $query->getResult();
}
```

Et voici comment on appelle cette fonction dans le contrôleur en utilisant la classe **ArtistRepository**:

```
class AccueilController extends AbstractController
{
    private $artistRepo;
    private $discRepo;
    private $em;

    public function __construct(ArtistRepository $artistRepo, DiscRepository $discRepo,
EntityManagerInterface $em)
    {
        $this->artistRepo = $artistRepo;
        $this->discRepo = $discRepo;
        $this->em = $em;
    }
    #[Route('/accueil', name: 'app_accueil')]
    public function index(): Response
    {
        //on appelle le repository pour accéder à la fonction
        $artistes = $this->artistRepo->getSomeArtists("Neil");

        //on teste le contenu de la variable $artistes : dd() veut dire Dump and Die
        dd($artistes);

        // ...
    }
}
```

Le resultat de notre dd():

```
AccueilController.php on line 34:
array:1 [▼
  0 => App\Ent...\Artist {#717 ▼
    -id: 3
    -name: "Neil Young"
    -url: "https://neilyoung.warnerartists.net/gb/"
    -discs: Doctrin...\PersistentCollection {#733 ►}
  ]
]
```

Pour approfondir le DQL et les fonctions disponibles, suivez ce lien :
<https://www.doctrine-project.org/projects/doctrine-orm/en/2.15/reference/dql-doctrine-query-language.html>

Le QueryBuilder

Le `QueryBuilder` (constructeur de requêtes) est un outil de Doctrine qui permet de créer des requêtes type DQL. Le but du `QueryBuilder` est de générer DQL dynamiquement, ce qui est utile lorsque nous avons à définir des filtres optionnels, des jointures conditionnelles, etc..

Voici la liste des méthodes disponibles sur la classe `QueryBuilder` :

```
<?php
class QueryBuilder
{
    public function select($select = null);

    public function addSelect($select = null);

    public function delete($delete = null, $alias = null);

    public function update($update = null, $alias = null);

    public function set($key, $value);

    public function from($from, $alias, $indexBy = null);

    public function join($join, $alias, $conditionType = null, $condition = null, $indexBy = null);

    public function innerJoin($join, $alias, $conditionType = null, $condition = null, $indexBy = null);

    public function leftJoin($join, $alias, $conditionType = null, $condition = null, $indexBy = null);

    public function where($where);

    public function andWhere($where);

    public function orWhere($where);

    public function groupBy($groupBy);

    public function addGroupBy($groupBy);
```

```

public function having($having);

public function andHaving($having);

public function orHaving($having);

public function orderBy($sort, $order = null);

public function addOrderBy($sort, $order = null);
}

```

Observez bien cette liste : on retrouve pas mal de mots clés SQL (SELECT, JOIN, UPDATE, GROUP BY, ORDER BY, HAVING etc.).

Doctrine traduira les requêtes que nous voulons exécuter dans la bonne syntaxe SQL en prenant bien sûr en compte le système de base de données que nous utilisons (MySQL, PostgreSQL etc.).

Voyons comment on peut réécrire la requête DQL précédente avec le QueryBuilder :

```

<?php
// dans la classe ArtistRepository
public function getSomeArtists($name)
{
    // $name est un paramètre qui pour cet exemple a comme valeur "Neil";

    $qb = $this->createQueryBuilder('a');
    $qb
        ->andWhere('a.name like :name') // le `placeholder, comme en PDO!
        ->setParameter('name', '%' . $name . '%')
        ->orderBy('a.id', 'ASC')
        ->setMaxResults(10)
        ->getQuery();

    $artists = $qb->getResult();
    return $artists;
}

```

Explications :

- ⑩ D'abord, on instancie le constructeur (en anglais, builder ou factory), c'est lui qui va nous permettre de construire les requêtes.
- ⑩ Une fois le builder instancié, on appelle plusieurs méthodes de construction de requête. Ces méthodes sont généralement nommées de la même manière que les mots clef SQL.
- ⑩ Ensuite, on obtient la requête (getQuery()) et enfin, on retourne le résultat (getResult()).

Appelez cette fonction dans un contrôleur pour regarder le résultat et le DQL généré!

Notes

- ⑩ Le QueryBuilder offre une API puissante et dispose de toute une série de méthodes et d'expressions que vous pouvez retrouver ici : <https://www.doctrine->

project.org/projects/doctrine-orm/en/2.15/reference/query-builder.html#the-querybuilder

- ⑩ Mais le QueryBuilder a ses limites aussi et il arrive que vous soyez parfois obligés de recourir aux requêtes SQL natives (le **Nat i ve SQL**). Consultez cette doc : <https://www.doctrine-project.org/projects/doctrine-orm/en/2.15/reference/native-sql.html#native-sql> pour en savoir plus :
- ⑩ Lisez cette ressource : <https://www.doctrine-project.org/projects/doctrine-orm/en/2.15/reference/best-practices.html#best-practices>. afin de connaître les bonnes pratiques avec Doctrine