R4.01
Architecture Logicielle

Formulaires, Validation, CRUD

https://symfony.com/doc/current/forms.html
https://symfony.com/doc/current/validation.html

Formulaires – Principes

- Symfony intègre un composant « form builder » qui permet de créer et de manipuler facilement des formulaires HTML que l'on décrit *via* des objets PHP
- Twig contient des méthodes afin de générer le code HTML du formulaire à partir de l'objet PHP qui le décrit
- Un formulaire est toujours associé à une entité PHP (en général une entité Doctrine mais pas forcément)
- Le « mapping » entre les champs du formulaire et l'entité associée est gérée automatiquement (ils doivent avoir le même nom)
- La validation (backend) des données du formulaire est réalisée en appliquant des règles de validation que l'on aura définies, par des annotations, sur l'entité associée au formulaire
- Un formulaire peut être créé :
  - « Dynamiquement », dans un contrôleur, pour être transmis à un *template* qui l'affichera
  - Dans une classe PHP spécifique (ce qui permet de le réutiliser dans différents contrôleurs)

Formulaires - « workflow » de mise en œuvre

1. **Construire** une Formulaire Symfony, dans un contrôleur ou dans une classe PHP dédiée à ce formulaire
2. **Transmettre** et **Rendre** ce formulaire dans un *template* pour que l'utilisateur puisse y saisir de l'information et le soumettre
3. **Traiter** le formulaire pour valider les données qui y ont été saisies et en faire quelque chose (comme par exemple les faire persister en BD)

- **A chaque étape, on manipule toujours un objet PHP de type « formulaire » et l'entité associée** (en général une entité Doctrine) qui contient les données manipulées dans le formulaire.
- **L'affichage du formulaire et son traitement peuvent être faits dans un même contrôleur** (conseillé) ou dans deux contrôleurs séparés.

Exemple de formulaire avec une entité Task

- Exemple : création de formulaire pour manipuler un objet de type **Task**
- Une entité **Task** contient 2 champs : **task** (le texte qui décrit la tâche) et **dueDate** (qui contient la date à laquelle la tâche doit être terminée)

```php
// src/Entity/Task.php
namespace App\Entity;

class Task {

    protected $task;
    protected $dueDate;

    public function getTask(): string {
        return $this->task;
    }
    public function setTask($task): void {
        $this->task = $task;
    }
    public function getDueDate(): ?\DateTime {
        return $this->dueDate;
    }
    public function setDueDate(?\DateTime $dueDate): void {
        $this->dueDate = $dueDate;
    }
}
```

Créer un formulaire dans un contrôleur

- Le formulaire (**$form**) est construit par le « form builder ». Il est associé à une entité (**$task**)
- Ici on ajoute deux champs au formulaire (**task** et **dueDate**) correspondant aux attributs **task** et **dueDate** de la classe **Task**. On ajoute également un bouton « save ».
- Le « FormType » de chaque champ (TextType, DateType, …) détermine comment sera rendu le champ en HTML. Il est optionnel (si absent, il est prédit d'après le type PHP de l'attribut)
- « FormType » peut être un type « champ atomique » ou « formulaire » (=> formulaires imbriqués)

```php
// src/Controller/TaskController.php
namespace App\Controller;
use App\Entity\Task;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
class TaskController extends AbstractController {
    public function new(Request $request) {
        $task = new Task();
        // … on initialise éventuellement l'entité …
        $form = $this->createFormBuilder($task)
            ->add('task',    TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save',    SubmitType::class, ['label' => 'Create Task'])
            ->getForm();
        // …
    }
}
```

- On ajoute au formulaire associé à **$task** les attributs de l'entité (task, duedate)

- Le 2ème paramètre de la méthode **add**, optionnel, permet de spécifier le type du champ HTML qui sera généré (FormType)

- Le 3ème paramètre, optionnel aussi, permet de définir des attributs pour le champ (dans un tableau associatif)

FormTypes fournis par Symfony

Text Fields

- TextType
- TextareaType
- EmailType
- IntegerType
- MoneyType
- NumberType
- PasswordType
- PercentType
- SearchType
- UrlType
- RangeType
- TelType
- ColorType

Choice Fields

- ChoiceType
- EntityType
- CountryType
- LanguageType
- LocaleType
- TimezoneType
- CurrencyType

Date and Time Fields

- DateType
- DateIntervalType
- DateTimeType
- TimeType
- BirthdayType

Other Fields

- CheckboxType
- FileType
- RadioType

Field Groups

- CollectionType
- RepeatedType

Hidden Fields

- HiddenType

Buttons

- ButtonType
- ResetType
- SubmitType

Base Fields

- FormType

Vous pouvez bien sûr définir vos propres « Form Types »

https://symfony.com/doc/current/reference/forms/types.html

Décrire un formulaire dans une Classe (1)

- La commande **`php bin/console make:form`** permet de créer un squelette de classe PHP destinée à décrire un formulaire que l'on veut réutiliser dans plusieurs contrôleurs.
- On créerait ainsi la classe **TaskType** : formulaire pour éditer une entité de type **Task**

```php
// src/Form/Type/TaskType.php
namespace App\Form\Type;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
            ->add('task',    TextType::class)
            ->add('dueDate', DateType::class)
            ->add('save',    SubmitType::class)
        ;
    }
     public function configureOptions(OptionsResolver $resolver) {
        $resolver->setDefaults([
            'data_class' => Task::class,
        ]);
    }
}
```

On précise ici les champs qui doivent figurer dans le formulaire

On précise ici le Type de l'entité associée au formulaire (**Task**)

Décrire un formulaire dans une Classe (2)

- Un fois qu'un type de formulaire a été spécifié dans une classe, on peut l'utiliser dans un contrôleur, en l'associant à une entité (**$task**) :

```php
// src/Controller/TaskController.php
namespace App\Controller;

use App\Form\Type\TaskType;
// ...

class TaskController extends AbstractController {
    public function new() {
        // Créer un objet $task et éventuellement l'initialiser
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createForm(TaskType::class, $task);

        // ...
    }
}
```

Transmettre un formulaire à un *template*

- Le *template* reçoit le formulaire sous la forme d'un objet PHP (ici **myForm**), qui est créé et transmis par le contrôleur
- Cet objet PHP est le résultat de la méthode **createView** que l'on aura appliquée au formulaire.

```php
// src/Controller/TaskController.php
namespace App\Controller;
use App\Entity\Task;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\HttpFoundation\Request;

class TaskController extends AbstractController {
    public function new(Request $request) {
        $task = new Task();
        // ...

        $form = $this->createForm(TaskType::class, $task);

        return $this->renderForm('task/new.html.twig', [
            'myForm' => $form,
            // …
        ]);
    }
}
```

Rendre un formulaire dans un *template* (1)

- Le template reçoit le formulaire sous la forme d'un objet PHP (**myForm**)
- Twig dispose de fonctions (helpers) pour produire le code HTML du formulaire

```
{# templates/task/new.html.twig #}

{{ form(myForm) }}
```

- Le helper Twig **`form`** rend automatiquement chaque champ du formulaire avec un label et un message d'erreur (si il y a eu une erreur de **validation** auparavant)
- Si rien n'a été précisé, la soumission du formulaire sera par défaut traitée par la même action que celle qui a affiché le formulaire
- **On peut *configurer Twig pour appliquer des styles* prédéfinis lors du rendu :**

```
# config/packages/twig.yaml
twig:
    form_themes: ['bootstrap_4_layout.html.twig']
```

Rendre un formulaire dans un *template* (2)

- D'autres helpers sont disponible dans Twig pour rendre le formulaire élément par élément :

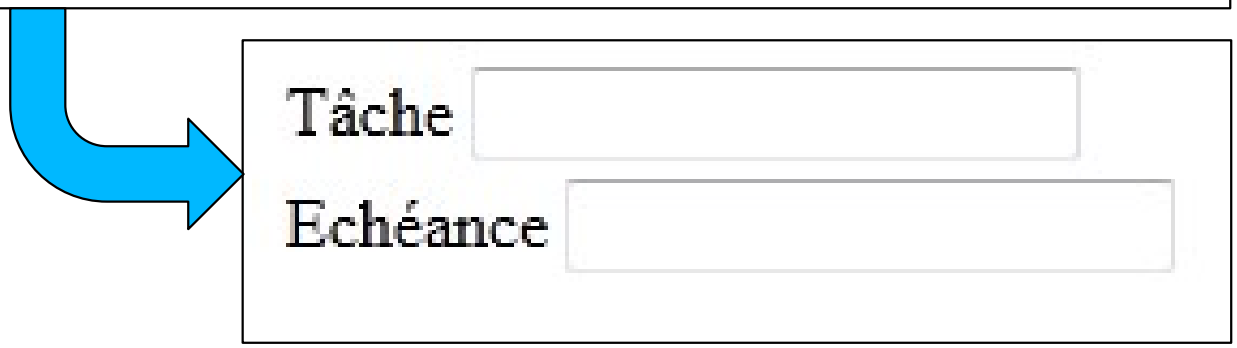
```
{{ form_start(myForm) }}
    <div class="my-custom-class-for-errors">
        {{ form_errors(myForm) }}
    </div>

    <div class="row">
        <div class="col">
            {{ form_row(myForm.task) }}
        </div>
        <div class="col" id="some-custom-id">
            {{ form_row(myForm.dueDate) }}
        </div>
    </div>
{{ form_end(myForm) }}
```

- https://symfony.com/doc/current/form/form_customization.html

Configurer le rendu d'un champ du formulaire

- Chaque **FormType** possède des **options** qui peuvent être utilisées pour configurer le rendu HTML, par exemple :
  - L'option **label** permet de changer l'étiquette qui sera affichée en face d'un champ
  - Le champ **dueDate** est rendu par défaut sous la forme de 3 champs « select » (jour, mois, année).
    Mais un champ de type **DateTimeType** peut être configuré pour être rendu sous la forme d'un unique champ texte grâce à l'option **widget**

```
$form = $this->createFormBuilder($task)
            ->add('task',    TextareaType::class, array('label'  => 'Tâche'))
            ->add('dueDate', DateTimeType::class, array('label'  => 'Echéance',
                                                        'widget' => 'single_text'))
            ->getForm();
```



Soumission du formulaire (1)

- L'étape suivante est de **transmettre** les données saisies par l'utilisateur aux propriétés de l'entité associée au formulaire
- Pour traiter la **soumission** du formulaire, on peut écrire un autre contrôleur ou bien écrire un seul contrôleur qui fait tout selon le scénario suivant :
  - Lors du chargement initial de la page dans votre navigateur, la méthode de la requête est **GET** et le formulaire est simplement créé et rendu
  - Lorsque l'utilisateur soumet le formulaire (méthode **POST**) avec des données non-valides, le formulaire est rendu à nouveau, affichant cette fois toutes les erreurs de validation ;
  - Lorsque l'utilisateur soumet le formulaire (méthode **POST**) avec des données valides, on peut alors manipuler l'entité (par exemple la persister dans la base de données), avant de rediriger l'utilisateur vers une autre page

    (rediriger un utilisateur après une soumission de formulaire réussie empêche l'utilisateur, lors d'un rafraichissement de page, de soumettre involontairement les données une deuxième fois)

Soumission du formulaire (2)

- On peut donc réécrire le contrôleur de la façon suivante pour qu'il gère l'affichage du formulaire ainsi que sa soumission :

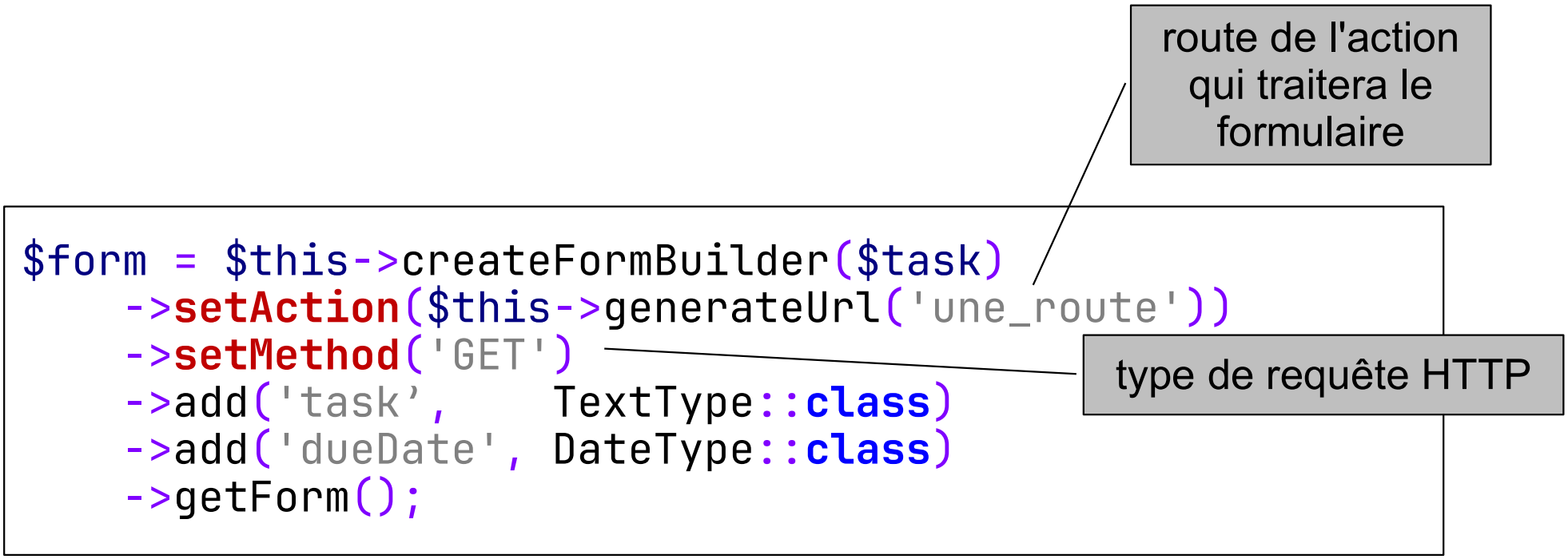
```php
// ...
use Symfony\Component\HttpFoundation\Request;
public function new(Request $request) {
    // Instancier un objet Task et son formulaire associé
    $task = new Task();
    $form = $this->createForm(TaskType::class, $task);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // On peut maintenant manipuler l'entité $task
        //  qui a été peuplée avec des données valides
        // On peut par exemple la sauvegarder
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($task);
        $entityManager->flush();
        return $this->redirectToRoute('task_success');
    }

    // On (ré)affiche le formulaire
    return $this->renderForm('task/new.html.twig',
                             ['myForm' => $form]);
}
```

La méthode **handleRequest** va traiter le formulaire (**$form**) en fonction de la nature de la requête HTTP (**$request**) :

- Si requête **GET** :
  - le formulaire n'est pas soumis, il devra donc être affiché.
  - **isSubmitted** renverra faux.
- Si requête **POST** :
  - Le formulaire est soumis, **isSubmitted** renverra vrai
  - On récupère les données saisies et on peuple l'entité (**$task**) avec ces données.
  - On applique alors les règles de validation de l'entité pour vérifier si les données sont correctes.
  - **isValid** renverra le résultat de cette vérification

Formulaire : Attributs Action et Method

- Quand on crée un formulaire, on peut spécifier l'action qui traitera le formulaire et le type de requête (**GET**/**POST**) avec les attributs **Action** et **Method** :



Pour garantir l'intégrité des données…

Validation

Validation d'Entité (et donc de Formulaire)

- Symfony permet de définir **au niveau des attributs des entités,** des règles de validation (contraintes) qui devront s'appliquer aux données stockées dans les attributs de l'entité.
- Ces règles de validation seront appliquées au moment de la soumission du formulaire (**handleRequest**), lorsque l'entité est peuplée avec les données du formulaire : `if ($form->isValid())`…
- Ces règles sont définies sous forme d'annotations sur les attributs de **l'entité** qui sera ensuite associée au formulaire :

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

class Task {

    #[Assert\NotBlank]
    public $task;

    #[Assert\NotBlank]
    #[Assert\Type("\DateTime")]
    protected $dueDate;
}
```

Validation - Types de contraintes

https://symfony.com/doc/current/validation.html#validation-constraints

Basic Constraints

- NotBlank
- Blank
- NotNull
- IsNull
- IsTrue
- IsFalse
- Type

String Constraints

- Email
- Length
- Url
- Regex
- Ip
- Uuid

Comparison Constraints

- EqualTo
- NotEqualTo
- IdenticalTo
- NotIdenticalTo
- LessThan
- LessThanOrEqual
- GreaterThan
- GreaterThanOrEqual
- Range

Date Constraints

- Date
- DateTime
- Time

Collection Constraints

- Choice
- Collection
- Count
- UniqueEntity
- Language
- Locale
- Country

File Constraints

- File
- Image

Financial and other Number Constraints

- Bic
- CardScheme
- Currency
- Luhn
- Iban
- Isbn
- Issn

Other Constraints

- Callback
- Expression
- All
- UserPassword
- Valid

Validation et Prédiction de Type de Champ

- Si vous avez ajouté des contraintes de validation aux attributs de la classe **Task**, le composant « form builder » peut prédire le type HTML des champs associés.
- Dans cet exemple, il peut deviner grâce aux règles de validation que le champ `task` sera un champ texte (TextType) et que `dueDate` sera un champ date (DateType)
- Lors de la création du formulaire, la « prédiction » sera utilisée si vous omettez le second argument de la méthode **add** ou si vous lui donnez la valeur **null**.
- Si vous passez un tableau d'options en tant que troisième argument à la méthode **add**, ces options seront appliquées au champ HTML.

```php
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, ['widget' => 'single_text'])
        ->getForm();
}
```

Validation d'une entité (hors du contexte « formulaire »)

- Il est possible de vérifier la validité des données d'une entité même si celle-ci n'a pas été peuplée par un formulaire
- Chaque fois que vous construisez une entité, vous pouvez utiliser le service **validator** pour vérifier que les règles de validation d'une entité sont respectées

```php
use App\Entity\Task;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Validator\Validator\ValidatorInterface;

public function author(ValidatorInterface $validator)
{
    $author = new Task();

    // Peupler l'entité Task avec des données
    $task->setTask(...);
    $task->setDueDate(...);

    // Et vérifier que l'entité est bien valide
    $errors = $validator->validate($task);

    // …
}
```

Pour développer + vite…

Génération d'interface CRUD

CRUD - Présentation

- Symfony, en s'inspirant de Ruby on Rails, offre un outil de génération de code très utile, appelé **échafaudage** (**scaffolding**)

- En une seule commande, on peut ajouter à l'application une interface « **CRUD** » qui va permettre de manipuler un type d'entité du modèle

- **CRUD** signifie :
  - **Create** : afficher un formulaire pour saisir une nouvelle entité et la sauvegarder en base
  - **Retrieve** : afficher la liste de toutes les entités et le détail de chacune d'elle
  - **Update** : afficher un formulaire pour modifier une entité existante et la sauvegarder en base
  - **Delete** : supprimer une entité de la base

CRUD - commande

- Pour créer l'interface **CRUD** de l'entité **Categorie**, lancer la commande :

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

- Cette commande va créer dans votre bundle :

  - Une classe PHP, **CategorieType**, représentant un formulaire permettant de créer ou éditer une entité de type Category (fichier **Form/CategorieType.php**)

  - Un contrôleur, **CategorieController**, contenant toutes les actions permettant de réaliser l'interface CRUD sur l'entité Category (fichier **Controller/CategoryController.php**)

  - Dans le répertoire **templates/Categorie**, 6 templates :
    - **index.html.twig** : affiche la **liste** de toutes les entités de type **Categorie**
    - **show.html.twig** : affiche le **détail** d'une entité de type **Categorie**
    - **new.html.twig** : affiche un **formulaire** permettant de **créer** une nouvelle entité **Categorie**
    - **edit.html.twig** : affiche un **formulaire** permettant **d'éditer** une entité **Categorie** existante
    - **_delete_form.html.twig** : formulaire de **suppression**
    - **_form.html.twig** : formulaire de création/éditiond'une entité **Categorie**

CRUD – les routes

Toutes les routes créées par CRUD ont des URL préfixées par **/categorie**

Une route a été créée pour chaque action définie dans le contrôleur :
- **index** : la liste
- **show** : le détail

- **new** : formulaire de création
- **create** : validation du formulaire de création

- **edit** : formulaire de modif.
- **update** : validation du formulaire de modif.

- **delete** : suppression

```php
// src/Controller/CategorieController.php
#[Route('/categorie')]
class CategorieController extends AbstractController {
  #[Route('/', name: 'app_categorie_index', methods: ['GET'])]
  public function index(CategorieRepository $categorieRepository): Response {
      // ...
  }

  #[Route('/new', name: 'app_categorie_new', methods: ['GET', 'POST'])]
  public function new(Request $request,
                      CategorieRepository $categorieRepository): Response {
    // ...
  }

  #[Route('/{id}', name: 'app_categorie_show', methods: ['GET'])]
  public function show(Categorie $categorie): Response {
    // ...
  }

  #[Route('/{id}/edit', name: 'app_categorie_edit', methods: ['GET', 'POST'])]
  public function edit(Request $request, Categorie $categorie,
                       CategorieRepository $categorieRepository): Response {
    // ...
  }

  #[Route('/{id}', name: 'app_categorie_delete', methods: ['POST'])]
  public function delete(Request $request, Categorie $categorie,
                         CategorieRepository $categorieRepository): Response {
    // ...
  }
}
```

CRUD – Le formulaire généré

- Une classe formulaire est généré automatiquement pour l'entité concernée
- Tous les champs (sauf l'identifiant) sont ajoutés au formulaire

```php
namespace App\Form;

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

class CategorieType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options): void {
        $builder
            ->add('libelle')
            ->add('visuel')
            ->add('texte')
        ;
    }

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

CRUD – Retrieve – liste des entités (index)

```php
// src/Controller/CategorieController.php
#[Route('/', name: 'app_categorie_index', methods: ['GET'])]
public function index(CategorieRepository $categorieRepository): Response {
  return $this->render('categorie/index.html.twig', [
                       'categories' => $categorieRepository->findAll(),
  ]);
}
```

```twig
// templates/categorie/index.html.twig
{% for categorie in categories %}
  <tr>
  <td>{{ categorie.id }}</td>
  <td>{{ categorie.libelle }}</td>
  <td>{{ categorie.visuel }}</td>
  <td>{{ categorie.texte }}</td>
  <td>
    <a href="{{ path('app_categorie_show', {'id': categorie.id}) }}">show</a>
    <a href="{{ path('app_categorie_edit', {'id': categorie.id}) }}">edit</a>
  </td>
  </tr>
{% endfor %}
```

Categorie index

Id	Libelle	Visuel	Texte	actions
1	Fruits	images/categories/fruits.jpg	De la passion ou de ton imagination	show edit
2	Légumes	images/categories/legumes.jpg	Plus tu en manges, moins tu en es un	show edit
3	Junk Food	images/categories/junk_food.jpg	Chère et cancérogène, tu es prévenu(e)	show edit
4	Virus	images/categories/corona.jpg	Une opportunité, il faut en profiter	show edit

Create new

CRUD – Retrieve – détail d'une entité (show)

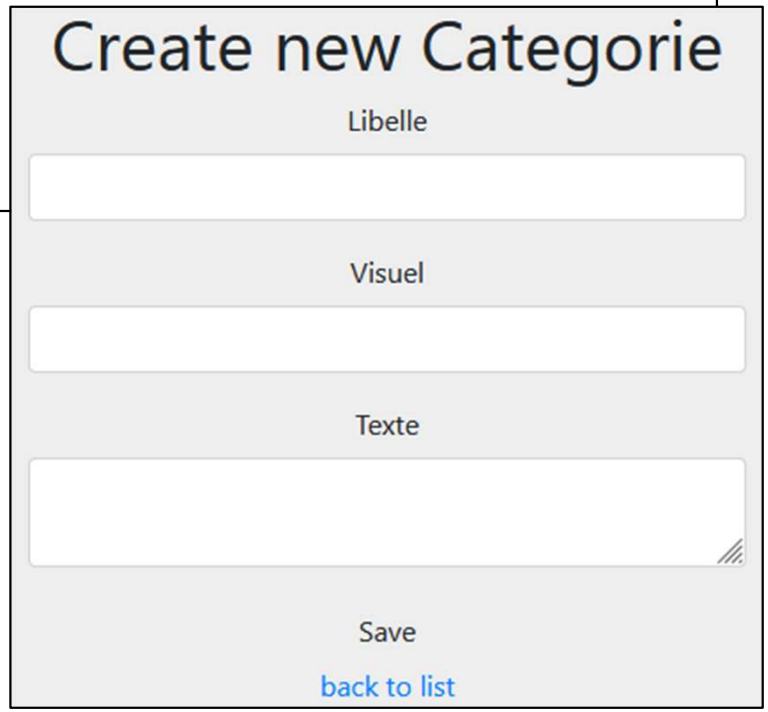
```php
// src/Controller/CategorieController.php
  #[Route('/{id}', name: 'app_categorie_show', methods: ['GET'])]
  public function show(Categorie $categorie): Response {
    return $this->render('categorie/show.html.twig', [
                         'categorie' => $categorie,
    ]);
  }
```

```twig
// templates/categorie/index.html.twig
    <table class="table">
        <tbody>
            <tr>
                <th>Id</th>
                <td>{{ categorie.id }}</td>
            </tr>
            <tr>
                <th>Libelle</th>
                <td>{{ categorie.libelle }}</td>
            </tr>
            <tr>
                <th>Visuel</th>
                <td>{{ categorie.visuel }}</td>
            </tr>
            <tr>
                <th>Texte</th>
                <td>{{ categorie.texte }}</td>
            </tr>
        </tbody>
    </table>
```

CRUD – Create – nouvelle entité (new) (1)

```php
// src/Controller/CategorieController.php
  #[Route('/new', name: 'app_categorie_new', methods: ['GET', 'POST'])]
  public function new(Request $request, CategorieRepository $categorieRepository): Response {
    $categorie = new Categorie();
    $form = $this->createForm(CategorieType::class, $categorie);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
      $categorieRepository->save($categorie, true);
      return $this->redirectToRoute('app_categorie_index', [], Response::HTTP_SEE_OTHER);
    }
    return $this->renderForm('categorie/new.html.twig', [
                             'categorie' => $categorie,
                             'form' => $form,
    ]);
  }
```

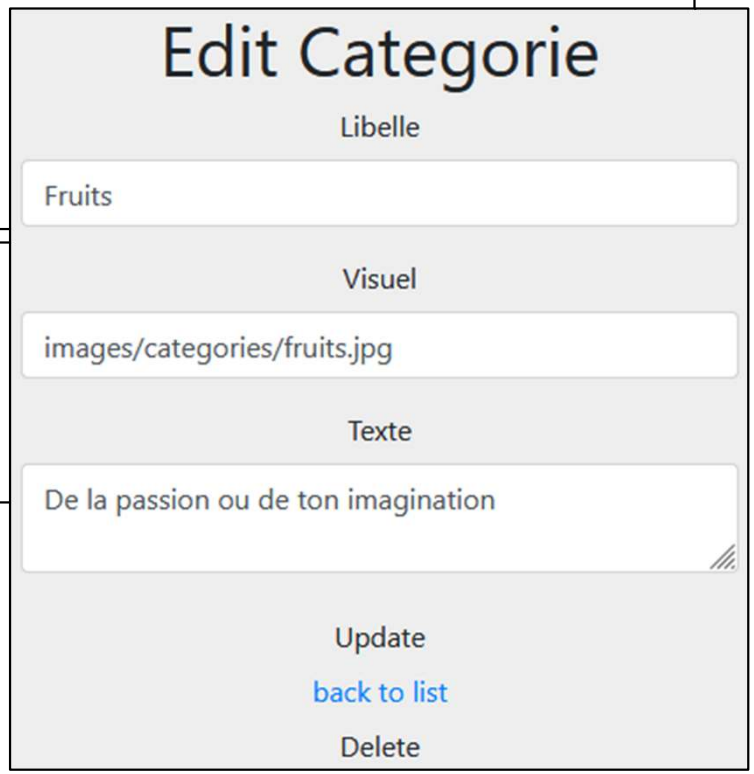
```twig
// templates/categorie/new.html.twig
<h1>Create new Categorie</h1>
{{ include('categorie/_form.html.twig') }}
<a href="{{ path('app_categorie_index') }}">
  back to list
</a>
```



CRUD – Update– édition d'une entité (edit) (1)

```php
// src/Controller/CategorieController.php
  #[Route('/{id}/edit', name: 'app_categorie_edit', methods: ['GET', 'POST'])]
  public function edit(Request $request, Categorie $categorie,
                       CategorieRepository $categorieRepository): Response {
    $form = $this->createForm(CategorieType::class, $categorie);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
      $categorieRepository->save($categorie, true);
      return $this->redirectToRoute('app_categorie_index', [], Response::HTTP_SEE_OTHER);
    }
    return $this->renderForm('categorie/edit.html.twig', [
                             'categorie' => $categorie,
                             'form' => $form,
    ]);
  }
```

```twig
// templates/categorie/edit.html.twig
<h1>Edit Categorie</h1>
{{ include('categorie/_form.html.twig',
           {'button_label': 'Update'}) }}
<a href="{{ path('app_categorie_index') }}">back to list</a>
{{ include('categorie/_delete_form.html.twig') }}
```



CRUD : usage

- Le code généré par la commande CRUD est destiné à être modifié et adapté à vos besoins : c'est donc un **échafaudage** sur lequel le développeur va monter pour bâtir son application plus vite

- On peut bien sûr supprimer une partie des fonctionnalités fabriquées par la commande CRUD si l'on ne s'en sert pas

- **En TP, nous utiliserons cette commande uniquement pour nous aider à coder la fonctionnalité « inscription d'un nouveau client »**

- La commande CRUD est aussi très utile pour fabriquer, (presque) sans coder (approche dite Low Code), le « back-office » d'une application

- Il existe d'autres bundles Symfony pour fabriquer des interfaces CRUD plus complètes, paramétrables et avec une interface plus recherchée :
  - EasyAdminBundle
  - SonataAdminBundle