Formation Symfony 4
3ème séance

Symfony

Retour sur TP

- Créer une entité Categorie liée en OneToMany avec l'entité Movie (crée facilement avec le maker-bundle)
- Au lieu de faire une route pour chaque catégorie, mieux vaut faire une seule route qui prend en paramètre une catégorie

```php
/**
 * @Route("/search/{categorie}", name="movie_index_categorie", methods="GET")
 */
public function indexCategorie($categorie, MovieRepository $movieRepository): Response
{
    $movies = $movieRepository->findByCategorie($categorie);
    return $this->render( view: 'movie/movie.html.twig', ['movies' => $movies]);
}
```

- Au niveau de la vue : {{ movie.categorie.name }} si on veut afficher le nom de la catégorie d'un film

Retour sur TP 2

- On peut créer des requêtes personnalisées depuis le Repository :

```php
public function findByCategorie($value)
{
    return $this->createQueryBuilder( alias: 'm')
        ->innerJoin( join: 'm.categorie',  alias: 'c')
        ->andWhere('c.name = :val')
        ->setParameter( key: 'val', $value)
        ->orderBy( sort: 'm.id',  order: 'ASC')
        //->setMaxResults(10)
        ->getQuery()
        ->getResult()
    ;
}
```

Retour sur TP 3

- Pour qu'un lien amène vers une route, il faut utiliser twig comme cela : {{ path('chemin') }}. On peut aussi y mettre des paramètres.
- Dans notre cas :

```
href="{{ path('movie_index_categorie', {'categorie':'action'}) }}">Action</a>
href="{{ path('movie_index_categorie', {'categorie':'aventure'}) }}">Aventure</a>
href="{{ path('movie_index_categorie', {'categorie':'science-fiction'}) }}">Science-fiction
href="{{ path('movie_index_categorie', {'categorie':'comédie'}) }}">Comédie</a>
href="{{ path('movie_index_categorie', {'categorie':'horreur'}) }}">Horreur</a>
```

Les formulaires

- $ composer require form
- Un formulaire permet à l'utilisateur d'interagir directement avec les données
- Un formulaire se construit sur un objet existant et son objectif est d'**hydrater** cet objet (remplir les attributs de l'objet).
- Les formulaires sont dans **/src/Form/**
- Convention d'écriture : Name**Type**
- $ php bin/console make:form formName (il faut lier le form à l'entité correspondante)

Créer un formulaire avec le builder

- Pour ajouter un champ au builder :

$builder->add('nameChamp', champType::class) avec champType le type du champ et nameChamp le nom de l'attribut de votre Entité

- Par défaut un champ de formulaire doit être requis (vérification côté client). Vous pouvez toujours changer cela en rajoutant en argument l'option :

  array('required' => false)

Les différents types de champs

JUNIORISEP

Texte	Choix	Date et temps	Divers	Multiple
TextType	ChoiceType	DateType	CheckboxType	CollectionT
TextareaType	EntityType	DatetimeType	FileType	RepeatedTyp
EmailType	CountryType	TimeType	RadioType	
IntegerType	LanguageType	BirthdayType		
MoneyType	LocaleType			
NumberType	TimezoneType			
PasswordType	CurrencyType			
PercentType				
SearchType				
UrlType				
RangeType				

Imbrication de formulaires

- Permet d'afficher des champs qui ne sont pas des attributs directs de votre entité (donc d'un autre formulaire)
- EntityNameType => Dans le cas où vous voulez imbriquer un seul formulaire (EntityName = nom de votre entité)
- CollectionType => Dans le cas où vous voulez créer des instances de vos entités, les supprimer, etc.
- EntityType => Dans le cas où vous voulez sélectionner des entités déjà existantes
- FileType => Pour upload un fichier
- ….
- Lire la doc ! https://symfony.com/doc/current/forms.html#content_wrapper

Dans le builder

```php
class MovieType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add( child: 'title',  type: TextType::class)
            ->add( child: 'content',  type: TextareaType::class)
            ->add( child: 'image',  type: ImageType::class, array('required' => false))
            ->add( child: 'categorie',  type: EntityType::class, array(
                'class' => Categorie::class,
                'choice_label' => 'name',
            ))
            ->add( child: 'save',  type: SubmitType::class, ['label' => 'Sauvegarder le film'])
        ;
    }
```

Retourner/valider un formulaire

```php
/**
 * @Route("/newMovie", name="movie_new", methods="GET|POST")
 */
public function new(Request $request): Response
{
    $movie = new Movie();
    $form = $this->createForm( type: MovieType::class, $movie);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($movie);
        $em->flush();

        return $this->redirectToRoute( route: 'movie_index');
    }

    return $this->render( view: 'movie/new.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

On peut rajouter une condition avec isMethod('POST') pour faire plus de vérifications.

L'objet Request

- L'objet Request $request contient les informations de la requête de l'utilisateur. De manière générale, il est accessible via les propriétés publiques suivantes :
  - request => équivalent de $_POST
  - query => équivalent de $_GET
  - cookies => équivalent de $_COOKIE
  - attributes => pour stocker des données additionnelles en paramètres
  - files => équivalent de $_FILES
  - server => équivalent de $_SERVER

Ex : $name = $request->request->get('name'); //nom envoyé en POST

Plus d'infos ici : http://symfony.com/doc/current/components/http_foundation.html#component-foundation-attributes

Méthodes d'affichage du formulaire

- {{ form(form) }} => affiche tout le formulaire
- {{ form_start(form) }} => affiche la balise <form>
- {{ form_end(form) }} => affiche la balise de fermeture </form>
- {{ form_label(form.attribut) }} => affiche le label HTML du champ donné
- {{ form_errors(form.attribut) }} => affiche les erreurs attachées au champ donné en argument
- {{ form_widget(form.attribut) }} => affiche le champ HTML lui-même (balise input)
- {{ form_row(form.attribut) }} => affiche le label, les erreurs et le champ en même temps
- {{ form_rest(form) }} => affiche tous les champs manquants du formulaire

Afficher un formulaire

JUNIORISEP

De manière rapide :

```
{{ form_start(form) }}
    {{ form_widget(form) }}
{{ form_end(form) }}
```

Attention : Par défaut, le formulaire une fois soumis va générer une requête POST et renvoyer vers le controller qui l'a affiché.

Afficher un formulaire

De manière détaillée:

```
{{ form_start(form) }}

    {{ form_label(form.title,"Titre",{'label_attr': {'class': 'col-lg-2 control-label'}}) }}
    {{ form_errors(form.title) }}
    {{ form_widget(form.title, {'attr': {'class': 'col-lg-10'}})}}

    {{ form_label(form.content,"Contenue",{'label_attr': {'class': 'col-lg-6 control-label'}}) }}
    {{ form_errors(form.content) }}
    {{ form_widget(form.content,{'attr': {'class': 'col-lg-10'}}) }}

    {{ form_label(form.image,"Image",{'label_attr': {'class': 'col-lg-12 control-label'}}) }}
    {{ form_errors(form.image) }}
    {{ form_widget(form.image)}}

    {{ form_label(form.categorie,"Catégorie",{'label_attr': {'class': 'control-label'}}) }}
    {{ form_errors(form.categorie) }}
    {{ form_widget(form.categorie) }}

    <div>{{ form_widget(form.save,{'attr': {'class': 'btn btn-success col-lg-2'}}) }}</div>
{{ form_end(form) }}
```

Créer un CRUD automatiquement

- CRUD = Create Read Update Delete
- $ php bin/console make:crud EntityName
- Fournit le Controller avec les méthodes associées, le formulaire ainsi que les vues

- Pratique pour tout générer en un instant, il faut tout de même personnaliser après selon ses besoins

Contraintes de validation

- Les contraintes (vérifications côté serveur) se font au niveau des entités elle-même ou du controller. Elles sont indispensables pour vous assurer que l'utilisateur ne rentre pas n'importe quoi dans le formulaire (en plus des vérifications côté client) ou que des objets soient correctes vis-à-vis des contraintes que vous avez spécifiées.
- $ composer require validator
- Pour voir toutes les contraintes disponibles : https://symfony.com/doc/current/validation.html

Types de contraintes (principales)

Contrainte	Rôle	Options
`NotBlank` `Blank`	La contrainte `NotBlank` vérifie que la valeur soumise n'est ni une chaîne de caractères vide, ni `NULL`. La contrainte `Blank` fait l'inverse.	-
`True` `False`	La contrainte `True` vérifie que la valeur vaut `true`, `1` ou `"1"`. La contrainte `False` vérifie que la valeur vaut `false`, `0` ou `"0"`.	-
`NotNull` `Null`	La contrainte `NotNull` vérifie que la valeur est strictement différente de *null*.	-
`Type`	La contrainte `Type` vérifie que la valeur est bien du type donné en argument.	`type` (option par défaut) : le type duquel doit être la valeur, parmi `array`, `bool`, `int`, `object`, etc.

Contrainte	Rôle	Options
`Email`	La contrainte `Email` vérifie que la valeur est une adresse e-mail valide.	`checkMX` (défaut : `false`) : si défini à `true`, Symfony va vérifier les MX de l'e-mail via la fonction `checkdnsrr`.
`Length`	La contrainte `Length` vérifie que la valeur donnée fait au moins X ou au plus Y caractères de long.	`min` : le nombre de caractères minimum à respecter. `max` : le nombre de catactères maximum à respecter. `minMessage` : le message d'erreur dans le cas où la contrainte minimum n'est pas respectée. `maxMessage` : le message d'erreur dans le cas où la contrainte maximum n'est pas respectée. `charset` (défaut : `UTF-8`) : le charset à utiliser pour calculer la longueur.
`Url`	La contrainte `Url` vérifie que la valeur est une adresse URL valide.	`protocols` (défaut : `array('http', 'https')`) : définit les protocoles considérés comme valides. Si vous voulez accepter les URL en `ftp://`, ajoutez-le à cette option.

`File`	La contrainte `File` vérifie que la valeur est un fichier valide, c'est-à-dire soit une chaine de caractères qui pointe vers un fichier existant, soit une instance de la classe File (ce qui inclut UploadedFile).	`maxSize` : la taille maximale du fichier. Exemple : 1M ou 1k. `mimeTypes` : mimeType(s) que le fichier doit avoir.
`Image`	La contrainte `Image` vérifie que la valeur est valide selon la contrainte précédente File (dont elle hérite les options), sauf que les mimeTypes acceptés sont automatiquement définis comme ceux de fichiers images. Il est également possible de mettre des contraintes sur la hauteur max ou la largeur max de l'image.	`maxSize` : la taille maximale du fichier. Exemple : 1M ou 1k. `minWidth` / `maxWidth` : la largeur minimale et maximale que doit respecter l'image. `minHeight` / `maxHeight` : la hauteur minimale et maximale que doit respecter l'image.

Contraintes sur les nombres :

Contrainte	Rôle	Options
Range	La contrainte `Range` vérifie que la valeur ne dépasse pas X, ou qu'elle dépasse Y.	`min` : la valeur minimum à respecter. `max` : la valeur maximum à respecter. `minMessage` : le message d'erreur dans le cas où la contrainte minimum n'est pas respectée. `maxMessage` : le message d'erreur dans le cas où la contrainte maximum n'est pas respectée. `invalidMessage` : message d'erreur lorsque la valeur n'est pas un nombre.

Contraintes sur les dates :

Contrainte	Rôle	Options
`Date`	La contrainte `Date` vérifie que la valeur est un objet de type `Datetime`, ou une chaine de caractères du type `YYYY-MM-DD`.	-
`Time`	La contrainte `Time` vérifie que la valeur est un objet de type `Datetime`, ou une chaine de caractères du type `HH:MM:SS`.	-
`DateTime`	La contrainte `Datetime` vérifie que la valeur est un objet de type `Datetime`, ou une chaine de caractères du type `YYYY-MM-DD HH:MM:SS`.	-

Exemple d'utilisation

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

```php
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Length(
 *      min=5,
 *      max=30,
 *      minMessage="Le titre est trop court",
 *      maxMessage="Le titre est trop long")
 */
private $title;

/**
 * @ORM\Column(type="text")
 * @Assert\NotBlank()
 */
private $content;
```

Projet : Site web (Suite)