Un CRUD automatique

> **RAPPEL**
> CRUD = Create/Read/Update/Delete

Avec le Maker, il est possible de générer une interface d'administration complète pour une entité :

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

La console demandera le nom de l'entité concernée, afin de créer une classe de contrôleurs complète contenant les contrôleurs nécessaires, à la création, visualisation, modification et suppression de données. On peut donc générer un CRUD pour notre entité `Article` et tous les fichiers nécessaires seront créés.

Contrôleurs, Formulaire et vues

Si on regarde les fichiers créés par le Maker, on trouvera une classe de contrôleurs contenant 5 méthodes :

- `index` : liste des articles
- `new` : nouvel article
- `show` : fiche article
- `edit` : modification d'article
- `delete` : suppression d'article

Globalement, dans tous les contrôleurs, on comprend ce qui se passe. Dans le dossier `Form`, on retrouve la définition du formulaire pour un article.

Enfin, dans le dossier `templates`, on trouve toutes nos vues pour chaque contrôleur, chaque action qu'on souhaite effectuer avec un CRUD.

Nous allons nous attarder sur 2 choses : l'attribut `Route` au niveau de la classe de contrôleurs, et le token CSRF dans la méthode `delete` de la même classe.

L'attribut `Route` sur une classe

On l'aura compris assez vite en le voyant au-dessus de la classe de CRUD, mais l'attribut `Route`, utilisé au-dessus d'une classe, permet de définir un **préfixe** pour tous les contrôleurs se trouvant dans la classe.

```php
#[Route('/article/crud')]
class ArticleCrudController extends AbstractController
{
}
```

Ainsi, dans chaque contrôleur (chaque méthode de la classe), on n'aura que la partie d'URL qui se trouvera après ce préfixe.

Par exemple, pour l'index, on a :

```php
#[Route('/', name: 'app_article_crud_index', methods: ['GET'])]
```

Mais si on cherche la route associée en explorant le routes dans la ligne de commande :

```
php bin/console debug:router crud_index
```

Alors dans les informations de la route, on retrouve bien notre préfixe :

```
+--------------+------------------------------------------------------------+
| Property     | Value                                                      |
+--------------+------------------------------------------------------------+
| Route Name   | app_article_crud_index                                     |
| Path         | /article/crud/                                             |
...
```

Le token CSRF

On trouve un nouvel élément dans la méthode `delete` :

```
if ($this->isCsrfTokenValid('delete'.$article->getId(), $request->request->get('_token'))) {
  $articleRepository->remove($article, true);
}
```

S'il y a vérification de token avant de supprimer, c'est que ce token doit avoir été généré précédemment. C'est exactement le fonctionnement d'une potection contre les attaques CSRF.

Principe

Une attaque CSRF (**C**ross **S**ite **R**equest **F**orgery) consiste à forger une requête (POST, surtout, sur des formulaires) depuis l'extérieur de l'application web concernée. Sans passer par un formulaire de login, par exemple, on pourrait tout de même envoyer une requête POST à la cible du formulaire. Pour ce faire, on peut utiliser un autre client (cURL, Postman, une autre application web...) qui ciblerait lui-même l'URL de la cible du formulaire.

Afin d'éviter ça, on va s'assurer que la requête POST est bien passée par le formulaire au préalable :

- Au moment de l'affichage du formulaire, on génère un **token CSRF** qui sera stocké en session
- Quand on va soumettre le formulaire, le token CSRF va donc être posté au serveur
- Le serveur peut alors comparer ce qui lui a été envoyé avec ce qu'il possède en session
- Toute requête qui viendrait d'un autre client ne serait donc pas passée par le formulaire : il n'y aurait aucun token à fournir, ou bien que des tokens invalides

On empêchera donc la réalisation d'une action (ici, une suppression) si le token est invalide ou non fourni.

Si on relance notre serveur, on peut alors accéder aux URL générées pour consulter une liste des articles se trouvant dans notre base de données. Mais si on veut créer ou éditer un article, c'est un peu plus compliqué...le lien vers la catégorie n'est pas pris en charge.

Adaptation du formulaire

La commande du Maker nous a créé un formulaire pour l'entité `Article`, mais n'a pas adapté le type appliquable à chacune des propriétés de notre entité.

Pour la catégorie, ça bloque. Nous devons être plus explicites sur la manière de générer un formulaire.

Nous avons besoin, dans un formulaire d'article, de générer une liste de catégories dans laquelle on peut choisir la catégorie de l'article.

On peut donc chercher un `Type` qui serait adapté à notre lien vers l'entité `Category` dans la liste des Form Types sur le site de Symfony.

Dans la section "Choice Fields", on tombe sur un `EntityType`. En cliquant dessus, on voit qu'il s'agit d'un `ChoiceType` spécial. `ChoiceType`, de base, représente un composant qui rendra un élément `select` ou bien des input `checkbox` ou `radio`, permettant de sélectionner un ou

plusieurs éléments au sein d'une collection.

Le type `EntityType`, lui, est concentré sur les entités Doctrine : il est capable d'aller récupérer une liste d'objets depuis une entité qu'on lui aura spécifiée.

Nous indiquons donc dans notre `ArticleType` que c'est ce qu'on souhaite utiliser. Dans la méthode `buildForm` :

```
$builder
  ->add('title')
  ->add('date_created')
  ->add('content')
  ->add('category', EntityType::class, [
    'class' => Category::class,
    'choice_label' => 'name'
  ]);
```

Si on actualise la page, ça fonctionne. Une liste de catégories est présentée, et on peut sélectionner celle qu'on veut associer à notre article.

> **POUR ALLER PLUS LOIN**
>
> La génération automatique de CRUD nous aide à gagner du temps, mais il existe également des packages (appelés **bundles** dans Symfony) permettant de générer des interfaces d'administration très complètes. La plus populaire est le EasyAdmin Bundle.