BACKOFFICE SYMFONY

Nous allons mettre maintenant en place un BackOffice permettant la gestion complète de notre **BlogSymfony**. Il y a différents moyens de réaliser un BackOffice :

- A l'aide de bibliothèque existante (ex : **easyadmin** / chapitre précédent)
- A la main en scratch

Nous allons voir ici comment réaliser un BackOffice from scratch !

1ère étape :

Nous allons tout d'abord créer un nouveau controller afin de dissocier les différentes parties du site :

php bin/console make :controller AdminController

Vous avez maintenant dans le dossier **src/Controller** un nouveau fichier **AdminController.php**

2ème étape :

Dans le controller **AdminController**, créer la méthode **admin()** permettra d'afficher la page d'accueil du BackOffice :

```
/**
     * @Route("/admin", name="admin")
     */
    public function index()
    {
        return $this->render('admin/index.html.twig', [
        ]);
    }
```

Dans le template **template/admin/index.html.twig**, ajouter les liens suivants :

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

{% block title %}BackOffice{% endblock %}

{% block body %}

    <h1 class="display-4 text-center my-4">BACKOFFICE</h1>

    <p class="text-center">Bienvenue sur votre BackOffice</p>

    <a href="" class="col-md-6 offset-md-3 btn btn-dark mb-
2">GESTION DES ARTICLES</a>

    <a href="" class="col-md-6 offset-md-3 btn btn-info mb-
2">GESTION DES CATEGORIES</a>
```

```
    <a href="" class="col-md-6 offset-md-3 btn btn-primary mb-
2">GESTION DES MEMBRES</a>

    <a href="" class="col-md-6 offset-md-3 btn btn-warning mb-
2">GESTION DES COMMENTAIRES</a>

{% endblock %}
```

Nous avons créé 4 liens qui permettront de gérer les différentes parties du site, la gestion des articles, des catégories, des membres et des commentaires, nous ajouterons des routes dans les liens au fur et à mesure….

Avant de continuer notre backoffice, nous devons définir les rôles accordés aux utilisateurs, c'est-à-dire définir les accès aux différentes parties du site. Par exemple l'administrateur aura bien évidement accès à tout le site, en revanche, un internaute classique connecté sur le site n'aura pas accès au Backoffice.

Il faut créer un nouveau champ dans la BDD :

Nous allons modifier l'entité **User** existante.

php bin/console make:entity User

propriété Roles → type json → not null

Ne pas oublier de créer un fichier de migration et de l'executer en BDD

php bin/console make :migration

php bin/console doctrine :migrations :migrate

Nous allons maintenant modifier le fichier **config/package/security.yaml** :

```
providers:
        users_in_memory: { memory: null }

        in_database:
            entity:
                class: App\Entity\User
                property: email

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy

            provider: in_database

            form_login:
```

```
                login_path: security_login
                check_path: security_login

            logout:
                path: security_logout
                target: blog

    role_hierarchy:
        ROLE_ADMIN: ROLE_USER

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-
authentication

            # https://symfony.com/doc/current/security/impersonating_user.html
            # switch_user: true

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
        - { path: ^/login, roles: ROLE_USER }

    access_denied_url: /blog
```

providers : où se trouve les données de l'utilisateur que Symfony va devoir contrôler au moment de la connexion

firewalls : quelle partie du site nous allons protéger et par quelle moyen ?

main : tout le contenu du site

form_login : contenu du site protégé par un formulaire de connexion

login_path : route vers le formulaire de connexion

check_path : sur quelle route Symfony doit contrôler les données de l'utilisateur en moment de la connexion

access_control : permet d'indiquer à Symfony, quel rôle utilisateur a accès au BackOffice (ROLE_ADMIN) et quel rôle ont les utilisateurs qui se connectent (ROLE_USER)

role_hierarchy : permet d'indiquer à Symfony qu'un utilisateur ROLE_ADMIN, donc un administrateur du site a aussi un ROLE_USER, il peut ainsi accéder à toute les parties du site.

3ème étape : GESTION DES ARTICLES

Nous allons commencer à gérer les articles, il faut pour cela créer une nouvelle méthode, donc une nouvelle route, donc un nouveau Template !

Rendez-vous dans le controller **AdminController** et créer la méthode suivante :

```
/**
     * @Route("/admin/articles", name="admin_articles")
     */
    public function adminArticles(ArticleRepository $repo)
    {
        // On appel getManager afin de récupérer le noms des champs et des col
onnes
        $em = $this->getDoctrine()->getManager();

        // récupération des champs
        $colonnes = $em->getClassMetadata(Article::class)->getFieldNames();

        dump($colonnes);

        $articles = $repo->findAll();

        dump($articles);

        return $this->render('admin/admin_articles.html.twig', [
            'articles' => $articles,
            'colonnes' => $colonnes
        ]);
    }
```

Nous allons maintenant détailler et essayer de comprendre le code…

Pour pouvoir sélectionner des données dans la table **Article**, nous avons besoin de la classe **Repository** de l'entité **Article**, donc nous injections en dépendance, c'est-à-dire en argument de la méthode, la classe **ArticleRepository.**

$**repo** est un objet issu de la classe **ArticleRepository**, cette classe contient des méthodes prédéfinies en Symfony qui permettent de sélectionner des données dans la BDD (requête SELECT SQL)

Nous avons donc exécuté la méthode **findAll**() qui permet de sélectionner l'ensemble de la table **Article**.

```
$em->getClassMetadata(Article::class)->getFieldNames();
```

Nous sélectionnons ici les noms des champs / colonnes de la table **Article** dans la BDD.

Grace à la méthode **render()** nous demandons au controller d'envoyer les données sélectionnées en BDD dans le Template **admin_articles.html.twig** afin de traiter le rendu visuel avec les langages **HTML / TWIG**.

Rendez-vous maintenant dans le template **admin/admin_articles.html.twig**.

```
<table class="table table-bordered text-center">

        <tr>
        {% for data in colonnes %}

            {% if data != 'id' %}
                <th>{{ data }}</th>
            {% endif %}

        {% endfor %}
                <th>Catégorie</th>
                <th>Edit</th>
                <th>Supp</th>
        </tr>

        {% for key, data in articles %}

            <tr>
                <td>{{ data.title }}</td>
                <td>{{ data.content|raw }}</td>
                <td><img src="{{ data.image }}" alt="{{ data.title }}" style="
width: 150px"></td>
                <td>{{ data.createdAt|date("d/m/Y à H:i:s") }}</td>
                <td>{{ data.category }}</td>
                <td><a href="{{ path('admin_edit_article', {'id': data.id }) }
}" class="text-dark"><i class="fas fa-edit fa-1x"></i></a></td>
                <td><a href="{{ path('admin_delete_article', {'id': data.id })
 }}" class="text-dark"><i class="fas fa-trash-alt fa-1x"></i></a></td>
            </tr>
        {% endfor %}

    </table>
```

Nous avons ajouté 2 cellules supplémentaires contenant 2 liens afin de pouvoir modifier et supprimer les articles.

4ème étape :

Nous allons maintenant travailler sur la modification et la suppression des articles. Pour cela, nous allons donc créer une nouvelle méthode associée à 2 routes différentes :

```
/**
     * @Route("/admin/article/new", name="admin_new_article")
     * @Route("/admin/{id}/edit-article", name="admin_edit_article")
     */
    public function editArticle(Article $article = null, Request $request, Ent
ityManagerInterface $manager)
    {
        dump($article);

        if(!$article)
        {
            $article = new Article;
        }

        $form = $this->createForm(ArticleType::class, $article);

        $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid())
        {
            if(!$article->getId())
            {
                $article->setCreatedAt(new \DateTime);
            }

            $manager->persist($article);
            $manager->flush();

            $this-
>addFlash('success', 'Les modifications ont bien été enregistrés !');

            return $this->redirectToRoute('admin_articles');
        }

        return $this->render('admin/edit_article.html.twig', [
            'formEdit' => $form->createView(),
            'editMode' => $article->getId() !== null
        ]);
    }
```

Analysons maintenant cette méthode : **editArticle()**

AJOUT D'UN ARTICLE :

La route **/admin/article/**new permet de créer un nouvel article. Nous faisons appel à la méthode **createForm()** permettant de créer un formulaire à partir de la classe **ArticleType** (make :form), nous

avons créé cette classe dans un chapitre précédent (cf 7-comprendre les formulaires), elle permet de créer un formulaire à partir de l'entité **Article**.

La méthode **handleRequest()** permet d'envoyer chaque données saisies dans le formulaire directement dans les bons setters de l'objet **$article**.

Nous vérifions si le formulaire a bien été soumis (**isSubmitted**) et chaque donnée à bien été envoyé dans les bons setters (**isValid**()), nous préparons ensuite la requête d'insertion (**persist**()) et enfin nous exécutons la requête d'insertion SQL (**flush**())

Nous redirigeons ensuite vers la page d'affichage de l'ensemble des articles dans le BackOffice (**redirectToRoute**()).

MODIFIACTION D'UN ARTICLE :

La route **admin/{id}/edit** permet de modifier un article. C'est une **route paramétrée**, c'est-à-dire que nous devons envoyer un argument de type ID dans l'URL, c'est-à-dire l'ID d'un article.

Nous imposons en argument de la méthode **editArticle**() un argument de type **Article**, c'est une injection de dépendance ! Donc en envoyant l'ID d'un article dans l'URL, Symfony est capable d'aller sélectionner cet article en BDD et de l'envoyer automatiquement en argument de la méthode ! **$article** est donc un objet issu de la classe **Article** !

Exemple : si nous envoyons **l'ID article 2** dans l'URL (ex : **admin/2/edit**), **$article** contient automatiquement les données stockées en BDD de l'article ID 2.

Nous vérifions si le formulaire a bien été soumis (**isSubmitted**) et chaque donnée à bien été envoyé dans les bons setters (**isValid**()), nous préparons ensuite la requête de modification (**persist**()) et enfin nous exécutons la requête d'insertion SQL (**flush**())

persist() et **flush**() sont issue de l'interface **EntityManagerInterface**. C'est via cette interface que nous pouvons manipuler les lignes de la BDD (insertion, modification, suppression).

Nous redirigeons ensuite vers la page d'affichage de l'ensemble des articles dans le BackOffice (**redirectToRoute**()).

5ème étape :

Rendez-vous dans le Template **edit_article.html.twig** afin d'afficher le formulaire **d'insertion / modification** des articles envoyé par le controller.

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

{% block title %}BackOffice | Articles{% endblock %}

{% block body %}

    <h1 class="display-4 text-center my-4">
    {% if editMode %}
        BACKOFFICE | Modification de l'article
    {% else %}
        BACKOFFICE | Poster un nouvel article
    {% endif %}
    </h1>

    {{ form_start(formEdit, {'attr': {'class': 'col-md-7 mx-auto'} }) }}

        {{ form_row(formEdit.title, { 'attr': {'placeholder': "Titre de l'arti
cle"} }) }}

        {{ form_row(formEdit.category) }}

        {{ form_row(formEdit.content, { 'attr': {'placeholder': "Contenu de l'
article", "rows": 15} }) }}

        {{ form_row(formEdit.image, { 'attr': {'placeholder': "URL de l'image"
} }) }}

        <button type="submit" class="btn btn-primary mb-4">
        {% if editMode %}
            Enregistrer les modifications
        {% else %}
            Ajouter l'article
        {% endif %}
        </button>

    {{ form_end(formEdit) }}

{% endblock %}
```

6ème étape :

SUPPRESSION DES ARTICLES :

Nous allons maintenant créer une nouvelle méthode permettant de supprimer un article en BDD.

Dans l'affichage des articles dans le BackOffice, donc dans le Template **admin_articles.html.twig**, pensez à ajouter les routes des liens modification et suppression via la méthode **path()** de TWIG.

```
<td><a href="{{ path('admin_edit_article', {'id': data.id }) }}" class="text-
dark"><i class="fas fa-edit fa-1x"></i></a></td>
                <td><a href="{{ path('admin_delete_article', {'id': data.id })
 }}" class="text-dark"><i class="fas fa-trash-alt fa-1x"></i></a></td>
```

Dans le controller **AdminController**, créer la méthode deleteArticle() :

```
/**
     * @Route("admin/{id}/delete-article", name="admin_delete_article")
     */
    public function deleteArticle(Article $article, EntityManagerInterface $ma
nager)
    {
        $manager->remove($article);
        $manager->flush();

        $this->addFlash('success', "L'article a bien été supprimé !");

        return $this->redirectToRoute('admin_articles');
    }
```

Analysons le code…

La route **admin/{id}/delete-article** est une route paramétrée, c'est-à-dire que nous devons envoyer dans **l'URL** un paramètre de type **ID**, donc l'ID d'un article stocké en **BDD**, l'article que nous voulons supprimer.

Par exemple : Si nous cliquons sur le bouton suppression dans l'affichage des articles dans le Template **admin_articles.html.twig**, nous envoyons l'ID de l'article dans l'URL **admin/2/delete-article**. Nous imposons comme dépendance de notre méthode un objet issu de la classe **Article**.

Donc en envoyant l'ID d'un article dans l'URL, Symfony est capable d'aller sélectionner cet article en BDD et de l'envoyer automatiquement en argument de la méthode ! **$article** est donc un objet issu de la classe **Article** !

Nous exécutons la méthode **remove**() issue de l'interface **EntityManagerInterface**. C'est via cette interface que nous pouvons manipuler les lignes de la BDD (insertion, modification, suppression).La méthode **remove**() permet de préparer la requête de suppression et flush() exécute véritablement la requête SQL.

Nous redirigeons après la suppression vers l'affichage des articles dans le BackOffice (**redirectToRoute**()).

La méthode **deleteArticle**() ne renvoi aucun Template (**render**()), elle permet uniquement de supprimer un article de la BDD.

La gestion des autres tables est sensiblement similaire à la gestion des articles…. A vous de pratiquer et de terminer le BACKOFFICE !! 😉