ATELIER PROFESSIONNEL

MEDIATEK FORMATION

Compte-rendu

Réalisé par
Joseph-Nicolas YAZBEK

BTS SIO-SLAM
2ème année

Table des matières

Mission / Contexte

Le développement rapide des technologies numériques et l'élargissement de l'accès à l'information modifient considérablement les espaces culturels, notamment les médiathèques.

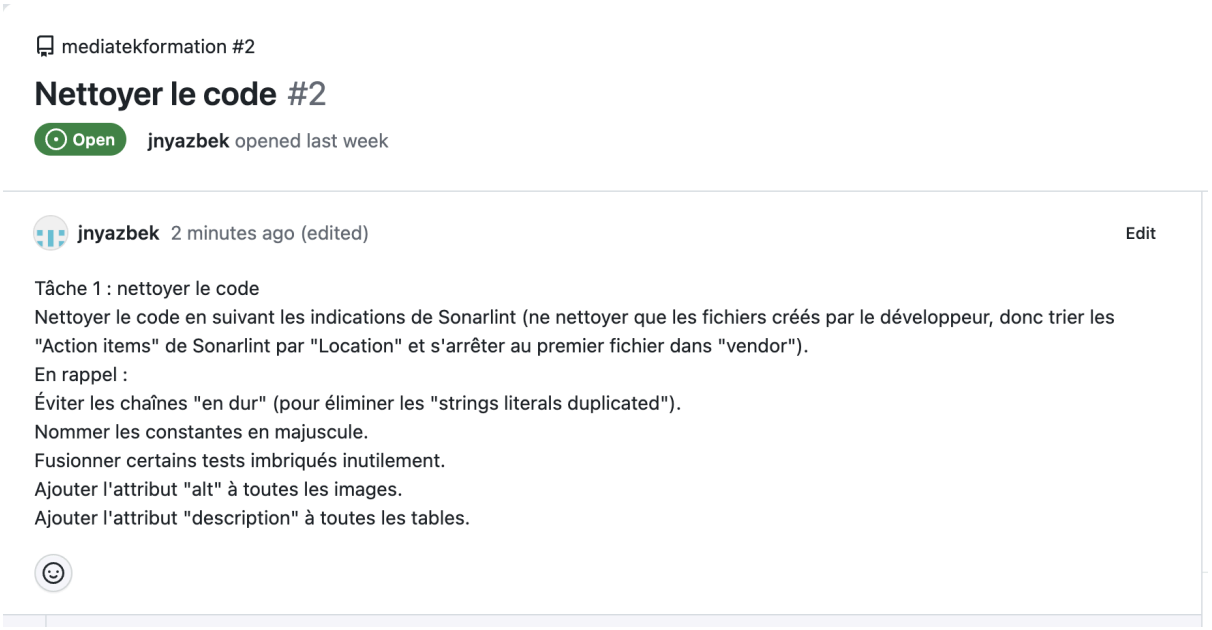
Dans ce contexte dynamique, le réseau de médiathèques viennoises MediaTek86 est à la pointe de l'innovation dans la gestion du patrimoine informatique et l'enrichissement de leurs services numériques.

En collaborant avec l'ESN ITS 86 le projet à abouti à : MediaTek86 qui incarne une vision moderne d'une médiathèque avec en son cœur un accès diversifié à la technologie et à la culture numériques.

Cet atelier professionnel vise à l'utilisation de Symfony pour le développement Web et MySQL pour la gestion de bases de données afin de modifier et d'optimiser les applications Web qui font partie intégrante de la stratégie de MediaTek86.

Mission 1 : nettoyer le code existant et ajouter une fonctionnalité

a) Tâche 1 : nettoyer le code



Temps estimé : 2 heures
Temps réel : 2 heures

Après analyse avec Sonarlint, nous nous retrouvons confrontés à plusieurs erreurs :

1. **Erreur numéro 1 : string literals should not be replicated**

Cela signifie que la même string est utilisée et redéfinie plusieurs fois dans le code. Pour respecter les bonnes pratiques, nous devons définir une constante que nous utiliserons chaque fois que la string sera nécessaire.

PlaylistController avant correction :

```php
public function index(): Response{
    $playlists = $this->playlistRepository->findAllOrderByName('ASC');
    $categories = $this->categorieRepository->findAll();
    return $this->render('pages/playlists.html.twig', [
        'playlists' => $playlists,
        'categories' => $categories
    ]);
}

/**
 * @Route("/playlists/tri/{champ}/{ordre}", name="playlists.sort")
 * @param type $champ
 * @param type $ordre
 * @return Response
 */
public function sort($champ, $ordre): Response{
    switch($champ){
        case "name":
            $playlists = $this->playlistRepository->findAllOrderByName($ordre);
            break;
    }
    $categories = $this->categorieRepository->findAll();
    return $this->render("pages/playlists.html.twig", [
        'playlists' => $playlists,
```

Nous corrigeons le problème en créant une constante contenant la chaîne, puis en la remplaçant dans les méthodes où elle est utilisée :

```php
const PAGE_PLAYLISTS = "pages/playlists.html.twig";

function __construct(PlaylistRepository $playlistRepository,
        CategorieRepository $categorieRepository,
        FormationRepository $formationRespository) {
    $this->playlistRepository = $playlistRepository;
    $this->categorieRepository = $categorieRepository;
    $this->formationRepository = $formationRespository;
}

/**
 * @Route("/playlists", name="playlists")
 * @return Response
 */
public function index(): Response{
    $playlists = $this->playlistRepository->findAllOrderByName('ASC');
    $categories = $this->categorieRepository->findAll();
    foreach($playlists as $playlist){
        $playlist->countFormation = $playlist->getCountFormations();
    }
    return $this->render(self::PAGE_PLAYLISTS, [
        'playlists' => $playlists,
        'categories' => $categories

    ]);
}
```

On fait de même pour la chaîne dans FormationController :

```php
public function index(): Response{
    $formations = $this->formationRepository->findAll();
    $categories = $this->categorieRepository->findAll();
    return $this->render("pages/formations.html.twig", [
        'formations' => $formations,
        'categories' => $categories
    ]);
}

/**
 * @Route("/formations/tri/{champ}/{ordre}/{table}", name="formations.sort")
 * @param type $champ
 * @param type $ordre
 * @param type $table
 * @return Response
 */
public function sort($champ, $ordre, $table=""): Response{
    $formations = $this->formationRepository->findAllOrderBy($champ, $ordre, $table);
    $categories = $this->categorieRepository->findAll();
    return $this->render("pages/formations.html.twig", [
        'formations' => $formations,
        'categories' => $categories
    ]);
}
```

On créé la constante et on remplace la chaîne :

```php
const PAGE_FORMATION = "pages/formations.html.twig";

function __construct(FormationRepository $formationRepository, CategorieRepository $categorieRepository) {
    $this->formationRepository = $formationRepository;
    $this->categorieRepository= $categorieRepository;
}

/**
 * @Route("/formations", name="formations")
 * @return Response
 */
public function index(): Response{
    $formations = $this->formationRepository->findAll();
    $categories = $this->categorieRepository->findAll();
    return $this->render(self::PAGE_FORMATION, [
        'formations' => $formations,
        'categories' => $categories
    ]);
}
```

On suit la même logique avec les string dans les classes PlayListRepository où l'on renseigne des constantes pour les strings 'p.formation' , 'p.id', 'p.name' et pour FormationRepository avec 'f.publishedAt' .
Ces dernières deviennent respectivement : PID, PFORMATION, PNAME et PUBLISHEDAT.

```php
class FormationRepository extends ServiceEntityRepository
{
    /**
     *
     * @const string
     */
    const FPUBLISHEDAT = 'f.publishedAt';
```

```php
class PlaylistRepository extends ServiceEntityRepository
{
    /**
     * accès aux formations de la playlist
     * @const string
     */
    const PFORMATION = 'p.formations';
    /**
     * accès a l'id de la playlist
     * @var string
     */
    const PID = 'p.id';
    /**
     * accès au nom de la playlist
     * @var string
     */
    const PNAME = 'p.name';
```

2. **Erreur numéro 2 : collapsible if statements should be merged**

Dans le package Entity, dans le fichier Playlist, on retrouve une méthode avec des if imbriqués :

```php
public function removeFormation(Formation $formation): self
{
    if ($this->formations->removeElement($formation)) {
        // set the owning side to null (unless already changed)
        if ($formation->getPlaylist() === $this) {
            $formation->setPlaylist(null);
        }
    }

    return $this;
}
```

Ces derniers peuvent facilement être résumés en un seul if en utilisant l'opérateur and "&&" :

```php
public function removeFormation(Formation $formation): self
{
    if ($this->formations->removeElement($formation) && $formation->getPlaylist() === $this ) {
        // set the owning side to null (unless already changed)
            $formation->setPlaylist(null);
    }

    return $this;
}
```

3. **Erreur numéro 3 : une autre correction est à apporter dans PlaylistController**

La fonction sort utilise un switch qui doit avoir un cas par défaut :

```php
/**
 * @Route("/playlists/tri/{champ}/{ordre}", name="playlists.sort")
 * @param type $champ
 * @param type $ordre
 * @return Response
 */
public function sort($champ, $ordre): Response{
    switch($champ){
        case "name":
            $playlists = $this->playlistRepository->findAllOrderByName($ordre);
            break;
    }
    $categories = $this->categorieRepository->findAll();
    return $this->render("pages/playlists.html.twig", [
        'playlists' => $playlists,
        'categories' => $categories
    ]);
}
```

On prévoit donc un cas par défaut utilisant la méthode findAll sans ordre dans le cas par défaut :

```php
public function sort($champ, $ordre): Response{
    switch($champ){
        case "name":
            $playlists = $this->playlistRepository->findAllOrderByName($ordre);
            break;
        default:
            $playlists = $this->playlistRepository->findAll();
            break;
    }
    $categories = $this->categorieRepository->findAll();
    return $this->render($this->pagePlaylist, [
        'playlists' => $playlists,
        'categories' => $categories
    ]);
}
```

4. **Erreur numéro 4 : ‹strong› and ‹em› tags should be used**

Il faut en effet remplacer les balises ‹i› par ‹em› ou ‹strong› en fonction des cas. On retrouve cette correction à effectuer à plusieurs reprises dans la page cgu.html.twig.

En voici les exemples :

```
dite L.C.E.N., il est porté à la connaissance des utilisateurs et visiteurs du site <u><a
.fr</a></u> (ci-après "<i>le site</i>"). L'accès et l'utilisation du Site sont soumis aux p
```

devient :

```
dite L.C.E.N., il est porté à la connaissance des utilisateurs et visiteurs du site <u><a h
.fr</a></u> (ci-après "<em>le site</em>"). L'accès et l'utilisation du Site sont soumis aux
```

et

```
etails>
  <summary>B. Hébergeur du site (ci-après "<i>l'Hébergeur</i>")</summary>
  <address>
```

devient :

```
  <summary>B. Hébergeur du site (ci-après "<strong>l'Hébergeur</strong>")</summary>
  <address>
      <strong>PlanetHoster</strong>
```

5. **Erreur numéro 5 : image tags should have an “alt attribute”**

L'attribut alt sert à attribuer une description à l'image qui peut s'avérer utile si elle venait à ne pas être chargée correctement.
Certaines images dans les pages du projet n'ont pas d'attribut alt renseignés, il faut donc les ajouter.

Exemple dans accueil.html.twig

```
  <img src="{{ formation.picture }}"
       class="card-img-top" style="width:100%;height:auto;">
  </a>
```

```
<img src="{{ formation.picture }}" alt="miniature de la formation",
     class="card-img-top" style="width:100%;height:auto;">
</a>
```

6. **Erreur numéro 6 : ‹table› tags should have a description**

Il faut ajouter des balises ‹caption› afin d'ajouter une description aux tables présentes dans le code.
Il faut donc appliquer cette correction au pages d'accueil, de playlist et de formation :

```
Voici les <strong>deux dernières formations</strong> ajoutées au catalogue :
<table class="table">
  <thead>
    <tr>
      {% for formation in formations %}
        <td>
          <div class="row">
```

```
<table class="table">
    <caption>Table des deux dernières formations </caption>
  <thead>
    <tr>
      {% for formation in formations %}
        <td>
```

```
<table class="table table-striped">
    <thead>
        <tr>
            <th class="text-left align-top" scope="col">
```

```
<table class="table table-striped">
  <caption> Table des formations </caption>
    <thead>
        <tr>
```

b) Tâche 2 : ajouter une fonctionnalité

mediatekformation #3

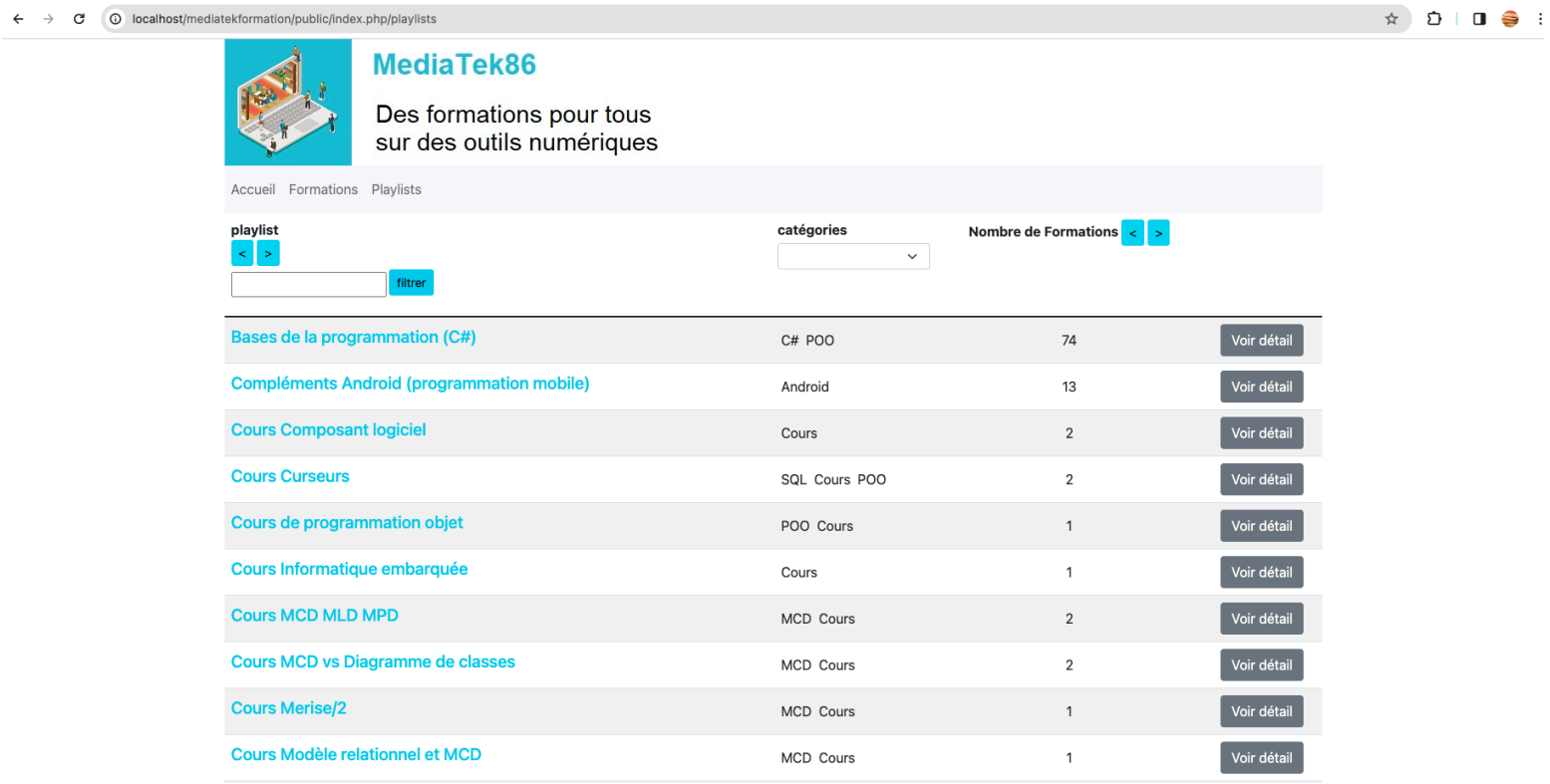
Ajout de la colonne nombre formations par playlist #3

Open **jnyazbek** opened last week

jnyazbek 5 hours ago (edited) Edit

Tâche 2 : ajouter une fonctionnalité
Dans la page des playlists, ajouter une colonne pour afficher le nombre de formations par playlist et permettre le tri croissant et décroissant sur cette colonne. Cette information doit aussi s'afficher dans la page d'une playlist.

L'objectif est de rajouter une colonne indiquant le nombre de formation dans chaque playlist :



Pour cela il faut réaliser les étapes suivantes :

- Ajout d'un header temporaire à la table de la page playlists afin de pouvoir la visualiser sans erreurs. On utilise les balises <thead> </thead.

- Puis, pour gérer l'affichage du nombre de formations par playlist, nous créons une fonction getCountFormation dans la PlaylistRepository :

```php
public function getCountFormations(): int
{
    return $this->formations->count();
}
```

Cette fonction qui permet de récupérer le nombre de formation est ensuite utilisé dans la méthode index du PlaylistController :

```php
/**
 * @Route("/playlists", name="playlists")
 * @return Response
 */
public function index(): Response{
    $playlists = $this->playlistRepository->findAllOrderByName('ASC');
    $categories = $this->categorieRepository->findAll();
    foreach($playlists as $playlist){
        $playlist->countFormation = $playlist->getCountFormations();
    }
    return $this->render($this->PAGE_PLAYLISTS, [
        'playlists' => $playlists,
        'categories' => $categories

    ]);
}
```

En itérant dans les différentes playlist pour appeler la fonction getCountFormation(), On récupère les nombres de formation et on les attribue à chaque playlist.
Dans la partie corps du fichier twig on ajoute au seing de la boucle parcourant les playlists :
‹td class="text-center"›{{ playlists[k].countFormations }}‹/td› afin de rajouter la colonne affichant chaque nombre de formation.

```
<!-- boucle sur les playlists -->
{% if playlists|length > 0 %}
    {% for k in 0..playlists|length-1 %}
        <tr class="align-middle">
            <td>
                <h5 class="text-info">
                    {{ playlists[k].name }}
                </h5>
            </td>
            <td class="text-left">
                {% set categories = playlists[k].categoriesplaylist %}
                {% if categories|length > 0  %}
                    {% for c in 0..categories|length-1 %}
                         {{ categories[c] }}
                    {% endfor %}
                {% endif %}
            </td>
            <td class="text-center">{{ playlists[k].countFormations }}</td>
```

Dans l'en-tête du fichier twig on ajoute également :

‹th class="text-center align-top" scope="col"›Nombre de Formations

‹a href="{{ path('playlists.sort', {champ:'countFormations', ordre:'ASC'}) }}" class="btn btn-info btn-sm active" role="button" aria-pressed="true"›‹‹/a›

‹a href="{{ path('playlists.sort', {champ:'countFormations', ordre:'DESC'}) }}" class="btn btn-info btn-sm active" role="button" aria-pressed="true"›››‹/a›

‹/th›

afin d'avoir l'entête de la colonne mais aussi les boutons permettant de trier par ordre croissant ou décroissant.

Cependant, pour que le tri fonctionne il faut ajouter une fonction dans le PlaylistRepository permettant de récupérer les données de façon ordonnées et d'ajouter un cas au switch de la fonction sort de la PlaylistsController.
On obtient donc :

```
public function findAllOrderedByFormationCount($ordre): array {
return $this->createQueryBuilder('p')
    ->leftJoin('p.formations', 'f')
    ->groupBy('p.id')
    ->orderBy('COUNT(f.id)', $ordre)
    ->getQuery()
    ->getResult();
```

Il faut ensuite prévoir le tri dans la fonction sort :

```php
/**
 * @Route("/playlists/tri/{champ}/{ordre}", name="playlists.sort")
 * @param type $champ
 * @param type $ordre
 * @return Response
 */
public function sort($champ, $ordre): Response{
    switch($champ){
        case "name":
            $playlists = $this->playlistRepository->findAllOrderByName($ordre);
            break;
        case "countFormations":
            $playlists = $this->playlistRepository->findAllOrderedByFormationCount($ordre);
            break;
        default:
            $playlists = $this->playlistRepository->findAll();
            break;
    }
    $categories = $this->categorieRepository->findAll();
    return $this->render($this->PAGE_PLAYLISTS, [
        'playlists' => $playlists,
        'categories' => $categories
    ]);
}
```

Enfin, on ajoute la ligne ‹strong›Nombre de formations : ‹/strong›{{ countFormations }} dans le fichier playlist.html.twig afin que le nombre de formations apparaisse également dans le détail.

Mission 2 : Coder la partie back-office

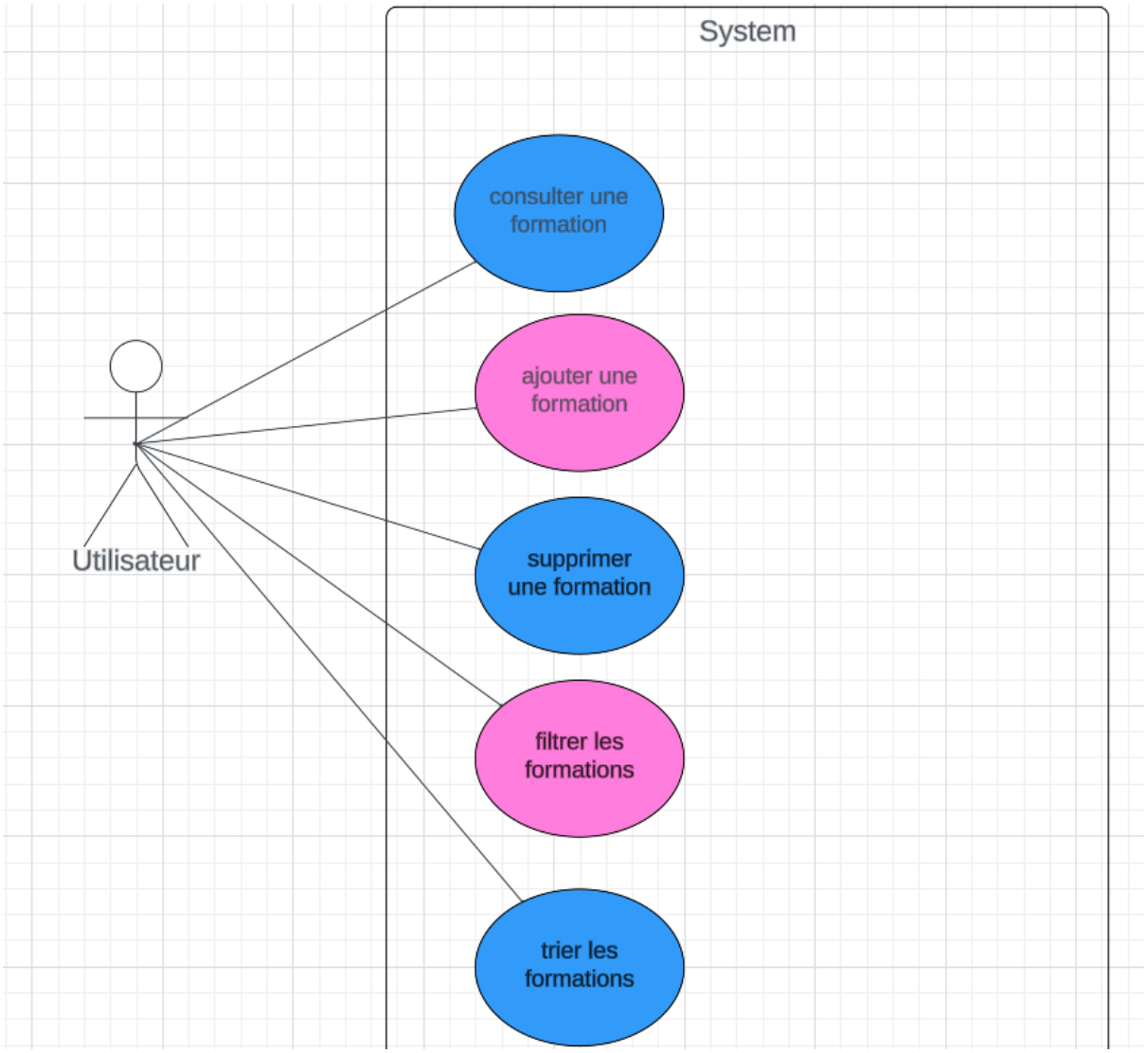
mediatekformation #4

Permettre l'ajout, la modification et la suppression d'une formation #4

Open **jnyazbek** opened last week

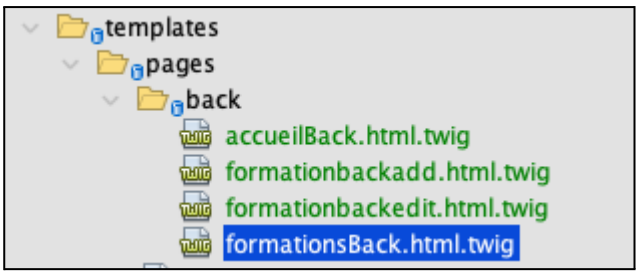
jnyazbek 5 hours ago (edited) Edit

Tâche 1 : gérer les formations
Une page doit permettre de lister les formations et, pour chaque formation, afficher un bouton permettant de la supprimer (après confirmation) et un bouton permettant de la modifier.
Si une formation est supprimée, il faut aussi l'enlever de la playlist où elle se trouvait.
Les mêmes tris et filtres présents dans le front office doivent être présents dans le back office.
Un bouton doit permettre d'accéder au formulaire d'ajout d'une formation. Les saisies doivent être contrôlées. Seul le champ "description" n'est pas obligatoire ainsi que la sélection de catégories (une formation peut n'avoir aucune catégorie). La playlist et la ou les catégories doivent être sélectionnées dans une liste (une seule playlist par formation, plusieurs catégories possibles par formation). La date ne doit pas être saisie mais sélectionnée. Elle ne doit pas être postérieure à la date du jour.
Le clic sur le bouton permettant de modifier une formation doit amener sur le même formulaire, mais cette fois prérempli.



On copie les pages front et on les renomme en ajoutant “back” pour avoir une base

(on pense bien sûr à changer les routes). Ainsi les tri et les filtres resteront fonctionnels.




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

{% block title %}{% endblock %}
{% block stylesheets %}{% endblock %}
{% block top %}
    <div class="container">
        <!-- titre -->
        <div class="text-left">
            <img src="{{ app.request.getBasePath()~'/banniere.jpg' }}", alt="image de la banniere" >
        </div>
        <!-- menu -->
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
            <div class="collapse navbar-collapse" id="navbarSupportedContent">
                <ul class="navbar-nav mr-auto">
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('formationsBack') }}">Formations</a>
                    </li>
                </ul>
            </div>
        </nav>
    </div>
{% endblock %}
```

a) Tâche 1 : gérer les formations

1. Ajouter une formation

On commence avec l'ajout de la colonne avec les boutons actions dans la page formationsBack.html.twig.

On ajoute l'en tête :

```twig
<th class ="text-center align-top" scope="col">
    Action
</th>
```

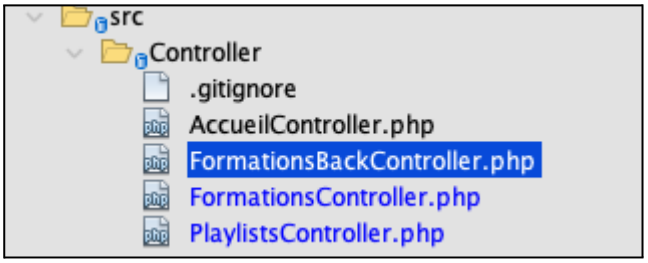
Et les cellules de la colonne contenant les boutons supprimer et modifier dans le "for" qui itère dans les formations :

```twig
<td class ="text-center">
    <button onclick="if(confirm('Êtes-vous sûr de vouloir supprimer cette formation ?')) {
        window.location.href = '{{ path('formationBack_delete', {id:formation.id}) }}'; }" class="btn btn-danger btn-sm">Supprimer</button>
    <a href="{{ path('formationBack_edit', {id:formation.id}) }}" class="btn btn-primary btn-sm">Modifier</a>
</td>
```

On à donc ici deux boutons, l'un permettant la suppression, l'autre la modification de formations. Le bouton supprimer demande confirmation avant de supprimer la formation.

2. Supprimer une formation

On crée le Contoller pour gérer les formations en back : FormationBackController sur le modèle de celui du front, mais on ajoute les fonctions et les routes nécessaires à la suppression.




```php
    /**
     * @Route("/formationsback/{id}", name="formationBack_delete")
     * @param type $id
     * @return Response
     */
    public function deleteOne($id): Response {
    $formation = $this->formationRepository->find($id);
    if ($formation) {
        // Supprime la formation de la playlist si nécessaire
        $playlist = $formation->getPlaylist();
        if ($playlist) {
            $playlist->removeFormation($formation);
        }

        // Supprime la formation de la base de données
        $this->formationRepository->remove($formation, true);
    }

    // Rediriger vers une autre page après la suppression
    return $this->redirectToRoute('formationsBack');
}
```

Ici, deleteOne récupère l'id de la formation, la retire des playlists dont elle fait partie et le cas échéant retire la formation de la base de donnée en utilisant la méthode remove du FormationRepository.

3. Éditer une formation

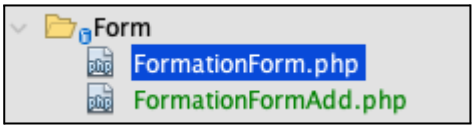
Dans la fonction editFormation, on récupère bien l'id de la formation à modifier pour pré remplir le formulaire.

```php
/**
 * @Route ("/formationsback/edit/{id}", name="formationBack_edit")
 * @param Request $request
 * @param type $id
 * @return Response
 */
public function editFormation(Request $request, $id): Response {
    $formation = $this->formationRepository->find($id);

    if (!$formation) {
        throw $this->createNotFoundException('No formation found for id '.$id);
    }

    $form = $this->createForm(FormationForm::class, $formation);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // Enregistrer les modifications
        $this->getDoctrine()->getManager()->flush();
        error_log("formulaire valide et envoyé");
        // Redirection vers la liste des formations
        return $this->redirectToRoute('formationsBack');
    } else {
        error_log("formulaire non valide ou non envoyé");
    }

    return $this->render('pages/back/formationbackedit.html.twig', [
        'form' => $form->createView(),
        'formation' => $formation // Ajoute la formation au formulaire
    ]);
}
```

La modification se fait à l'aide d'un formulaire que l'on doit créer :

- un avec un fichier php de classe qui en détermine le contenu FormationForm
- un twig qui l'affiche fotmationsbackedit.html.twig

Le formulaire FormationForm qui permet la modification d'une formation :



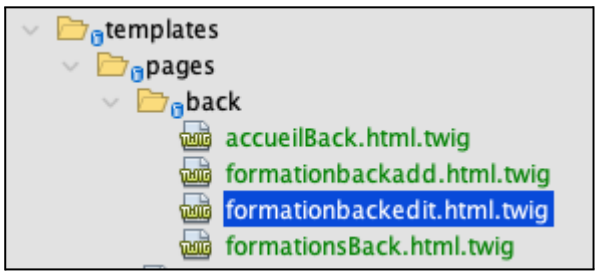
```php
class FormationForm extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
            ->add('title',TextType::class,[
                'label'=>'Titre',
                'attr' => ['maxlength'=> 100],
                'required' => true,
            ])//titre
            ->add('publishedAt', DateTimeType::class, [
                'widget'=>'single_text',
                'label' => 'Date de publication',
                'required'=>true,
                ])//date de publication
            ->add('description', TextareaType::class,[
               'label' => 'Description',
               'attr' => ['rows' => 6],
               'required' => false,
            ])//déscription
            ->add('playlist',EntityType::class, [
                'class' => Playlist::class,
                'choice_label'=> 'name',
                'required' => true,
                ])// sélecteur de playlists
            ->add('categories', EntityType::class, [
                'class' => Categorie::class,
                'choice_label' => 'name',
                'multiple' => true,
                'expanded' => true
            ])// sélecteur de catégories
             ->add('miniature', TextType::class, [
               'label' => 'Miniature URL',
               'attr' => ['maxlength' => 100],
               'required' => false,
            ])
            ->add('picture', TextType::class, [
               'label' => 'Image URL',
               'attr' => ['maxlength' => 100],
               'required' => false,
            ])
            ->add('videoId', TextType::class, [
               'label' => 'Video ID',
               'attr' => ['maxlength' => 11],
               'required' => false,
            ])
            ->add('Enregistrer',SubmitType::class);
    }
    public function configureOptions(OptionsResolver $resolver) {
        $resolver->setDefaults([
            'data_class' => Formation::class,
        ]);
    }
```

Ce formulaire prévoit la modification des caractéristiques d'une formation en proposant des champs de texte à modifier ou des widgets (comme pour la date) afin de faciliter la saisie des informations.

Le formulaire permet également d'indiquer que certains champs ne peuvent être nuls grâce au mot clef 'required' ou encore de préciser les caractéristiques de la saisie attendue comme une longueur maximale.

À la fin du formulaire, on retrouve un bouton Enregistrer qui permet d'envoyer le formulaire et de faire appel à la fonction editFormation qui enregistre les modifications dans la base de donnée si le formulaire est valide.

Il faut à présent afficher le formulaire, nous créons donc le fichier : formationbackedit.html.twig




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

{% block body %}
    <h2>Modifier Formation</h2>
    {{ form_start(form,{'action' : path('formationBack_edit',{'id': formation.id}), 'method' : 'POST'}) }}
       {{ form(form) }}
        {{ form_errors(form) }}
    {{ form_end(form) }}
{% endblock %}
```

On configure l'action qui doit être réalisée lorsque le bouton "Enregistrer" est pressé : on utilise la route menant à la méthode editFormation en passant l'id de la formation en paramètre.

Ensuite, nous affichons tous les champs prévu dans le formulaire en utilisant {{ form(form)}}.

Ajout de formation :
On met en place les mécanisme de l'ajout de la formation :
Dans FormationBackController on ajoute une méthode permettant l'ajout d'une nouvelle formation : AjoutFormation() , cette dernière est similaire à editFormation mais ne prend pas d'id en paramètre, à la place un nouvel objet formation est créé.

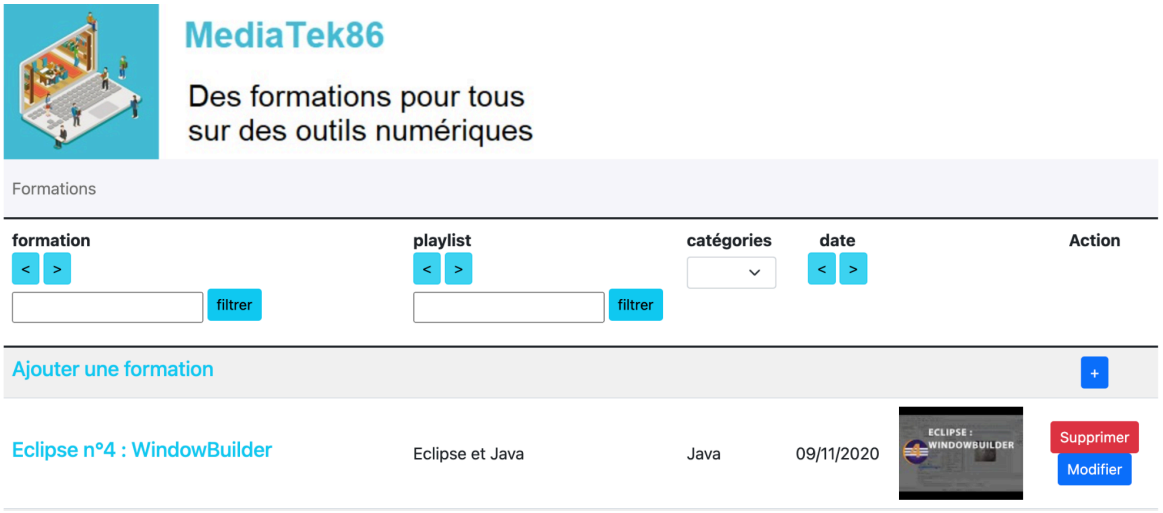
```php
/**
 * @Route ("/formationsback/add/1", name="formationBack_add")
 * @param Request $request
 * @return Response
 */
public function AjoutFormation(Request $request) : Response{
    $formation = new Formation();
    $form = $this->createForm(FormationFormAdd::class, $formation);
    $form->handleRequest($request);

     if ($form->isSubmitted() && $form->isValid()) {
        // Enregistrer les modifications
        $this->getDoctrine()->getManager()->persist($formation);
        $this->getDoctrine()->getManager()->flush();

        // Redirection vers la liste des formations
        return $this->redirectToRoute('formationsBack');
    }

    return $this->render('pages/back/formationbackadd.html.twig', [
        'form' => $form->createView(),
        'formation' => $formation // Ajoute la formation au formulaire
    ]);
}
```

Le fichier prévoyant le formulaire est essentiellement le même, on change simplement le nom du bouton.

Enfin il nous faut créer un bouton permettant l'accès à ce formulaire sur la page formationsback.html.twig :

```twig
<tbody>
    <tr class="align-middle">
            <td>
                <h5 class="text-info">
                    Ajouter une formation
                </h5>
            </td>
            <td>

            </td>
            <td>

            </td>
            <td>

            </td>
            <td>

            </td>
            <td>

            </td>
            <td class="text-center">
                <a href="{{ path('formationBack_add') }}" class="btn btn-primary btn-sm">
                    <span>+</span>
                </a>
            </td>
    </tr>
```

Avant le “for” itérant dans la liste des formation, on ajoute une ligne avec le texte “ajouter une formation” et le bouton “+” renvoyant vers la route permettant l’accès de la fonction AjouteFormation().



b) Tâche 2 : gérer les playlists

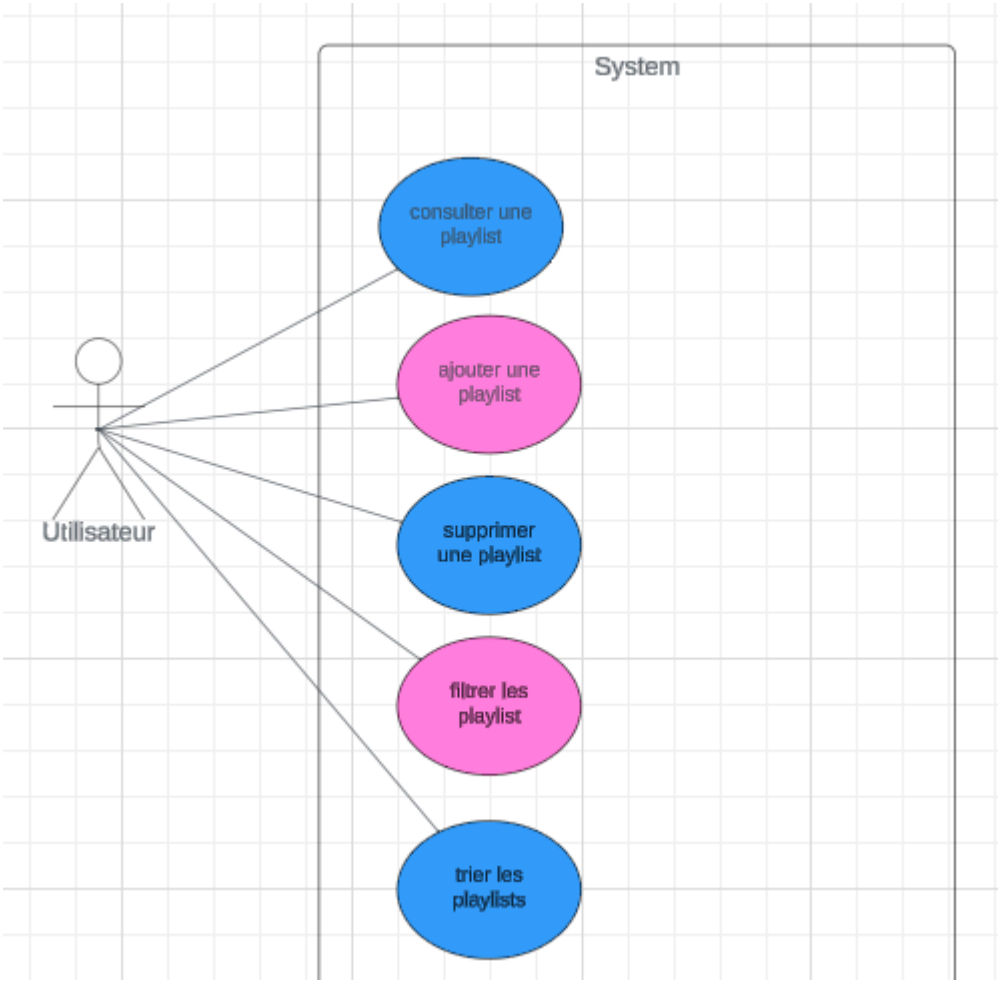
mediatekformation #5

Gérer la suppression, modification et création de playlists #5

Open **jnyazbek** opened 3 days ago

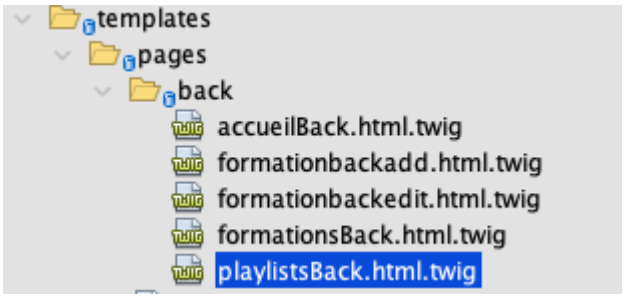
jnyazbek now (edited) Edit

Tâche 2 : gérer les playlists
Une page doit permettre de lister les playlists et, pour chaque playlist, afficher un bouton permettant de la supprimer (après confirmation) et un bouton permettant de la modifier.
La suppression d'une playlist n'est possible que si aucune formation n'est rattachée à elle.
Les mêmes tris et filtres présents dans le front office doivent être présents dans le back office.
Un bouton doit permettre d'accéder au formulaire d'ajout d'une playlist. Les saisies doivent être contrôlées. L'ajout d'une playlist consiste juste à saisir son nom et sa description. Seul le champ name est obligatoire.
Le clic sur le bouton permettant de modifier une playlist doit amener sur le même formulaire, mais cette fois prérempli. Cette fois, la liste des formations de la playlist doit apparaître, mais il ne doit pas être possible d'ajouter ou de supprimer une formation : ce n'est que dans le formulaire de la formation qu'il est possible de préciser sa playlist de rattachement.



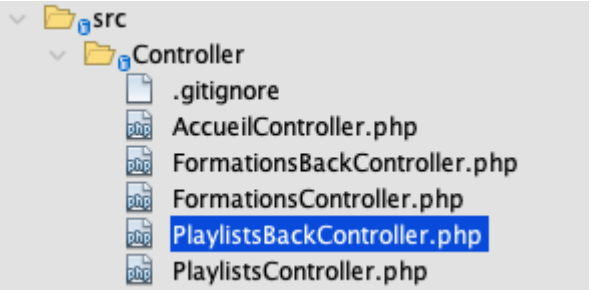
On recommence avec la gestion des playlists.

On crée la page playlistsback.html.twig sur la base de playlists.html.twig.
On change bien sûr l'héritage (on utilise baseback au lieu de basefront) et on change les routes indiquées en ajoutant back à la suite. Ainsi, on obtient directement les filtres et tris fonctionnels.



```twig
{% extends "baseback.html.twig" %}
{% block body %}
    <table class="table table-striped">
      <caption> Table des playlists </caption>
        <thead>
            <tr>
                <th class="text-left align-top" scope="col">
                    playlist<br />
                    <a href="{{ path('playlistsback.sort', {champ:'name', ordre:'ASC'}) }}" class="btn btn-info btn-sm active" role="button" aria-pressed="true"><</a>
                    <a href="{{ path('playlistsback.sort', {champ:'name', ordre:'DESC'}) }}" class="btn btn-info btn-sm active" role="button" aria-pressed="true">></a>
                    <form class="form-inline mt-1" method="POST" action="{{ path('playlistsback.findallcontain', {champ:'name'}) }}">
                        <div class="form-group mr-1 mb-2">
                            <input type="text" class="sm" name="recherche"
                                   value="{% if valeur|default and not table|default %}{{ valeur }}{% endif %}">
                            <input type="hidden" name="_token" value="{{ csrf_token('filtre_name') }}">
                            <button type="submit" class="btn btn-info mb-2 btn-sm">filtrer</button>
                        </div>
                    </form>
                </th>
```

Pour que la page soit fonctionnelle on créé une classe PlaylistsBackController sur le même modèle que PlaylistsController mais en changeant les routes.



1) Supprimer des playlists

On ajoute la méthode permettant la suppression de la playlist :

```php
/**
 * @Route("/playlistsback/{id}", name="playlistsback.delete")
 * @param type $id
 * @return Response
 */
public function deleteOne($id): Response{
    $playlist = $this->playlistRepository->find($id);
    if ($playlist) {
        $playlistFormations = $this->formationRepository->findAllForOnePlaylist($id);
        $countFormations = count($playlistFormations);
        if($countFormations == 0){
            $this->playlistRepository->remove($playlist,true);
            $this->getDoctrine()->getManager()->flush();
            return $this->redirectToRoute('playlistsBack');
        }


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

On récupère l'id de la playlist à supprimer. Si la playlist existe, on utilise la fonction countFormations pour déterminer si des formations sont rattachées à la playlist.
Si aucune formation n'est associée à cette playlist, alors on la supprime de la base de donnée.

Il nous faut maintenant ajouter la colonne et le bouton suppression au fichier playlistsback.html.twig. On ajoute l'en tête :

```
<th class ="text-center align-top" scope="col">
    Action
</th>
```

Et le bouton :

```
<td class ="text-center">
 <button onclick="if(confirm('Êtes-vous sûr de vouloir supprimer cette playlist ? Seules les playlists vides peuvent être supprimées')) {
      window.location.href = '{{ path('playlistsback.delete', {id:playlists[k].id}) }}'; }" class="btn btn-danger btn-sm">Supprimer</button>
```

Presser ce bouton demande confirmation pour la suppression et prévient que seule une playlist vide peut être supprimée.

2) Modifier des playlists

On ajoute au PlaylistBackController la méthode suivante permettant la modifications :

```
 * @Route ("/playlistsback/edit/{id}", name="playlistsback.edit")
 * @param Request $request
 * @param type $id
 * @return Response
 */
public function editPlaylist(Request $request, $id): Response {
    $playlist = $this->playlistRepository->find($id);

    if (!$playlist) {
        throw $this->createNotFoundException('No playlist found for id '.$id);
    }

    $form = $this->createForm(PlaylistForm::class, $playlist);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // Enregistrer les modifications
        $this->getDoctrine()->getManager()->flush();
        error_log("formaulaire valide et envoyé");
        // Redirection vers la liste des formations
        return $this->redirectToRoute('playlistsBack');
    } else {
        error_log("formulaire non valide ou non envoyé");
    }

    return $this->render('pages/back/playlistbackedit.html.twig', [
        'form' => $form->createView(),
        'playlist' => $playlist // Ajoute la playlist au formulaire
    ]);
}
```

Pour fonctionner, cette méthode récupère l'id de la playlist concernée et crée un formulaire permettant sa modification. Si le formulaire est ensuite validé et envoyé, il enregistre les modifications apportées dans la base de données.

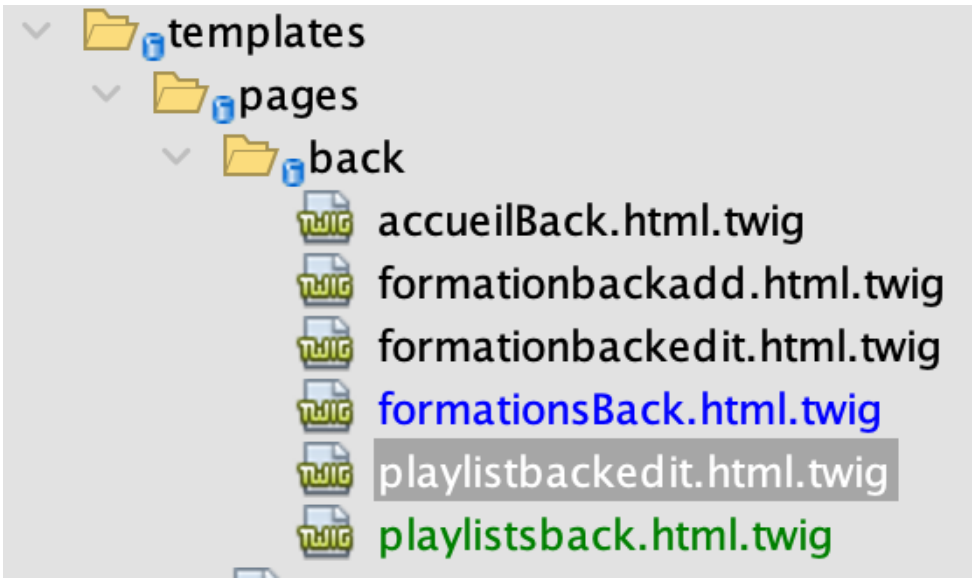
Pour pouvoir fonctionner, nous devons créer un formulaire :


```php
class PlaylistForm extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
            ->add('name',TextType::class,[
                'label'=>'Playlist',
                'attr' => ['maxlength'=> 100],
                'required' => true,
            ])//titre
            ->add('description', TextareaType::class,[
               'label' => 'Description',
               'attr' => ['rows' => 6],
               'required' => false,
            ])//description

            ->add('Enregistrer',SubmitType::class);
    }
    public function configureOptions(OptionsResolver $resolver) {
        $resolver->setDefaults([
            'data_class' => Playlist::class,
        ]);
    }
}
```

Cette classe permet la création d'un formulaire simple permettant la modification du nom et de la description de la playlist.

C'est à ce formulaire que fait référence la méthode createForm() dans editPlaylist().

Pour permettre l'affichage de cette forme on utilise un fichier twig simple playlistbackedit.html.twig :




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

{% block body %}
    <h2>Modifier Playlist</h2>
    {{ form_start(form) }}
        {{ form(form) }}
         {{ form_errors(form) }}
    {{ form_end(form) }}
{% endblock %}
```

Enfin, pour pouvoir afficher tout cela, il faut ajouter le bouton modifier à la page playlistback.html.twig :

```twig
<td class ="text-center">
 <button onclick="if(confirm('Êtes-vous sûr de vouloir supprimer cette playlist ? Seules les playlists vides peuvent être supprimées'
      window.location.href = '{{ path('playlistsback.delete', {id:playlists[k].id}) }}'; }" class="btn btn-danger btn-sm">Supprimer</
   <a href="{{ path('playlistsback.edit', {id:playlists[k].id}) }}" class="btn btn-primary btn-sm">Modifier</a>
</td>
```

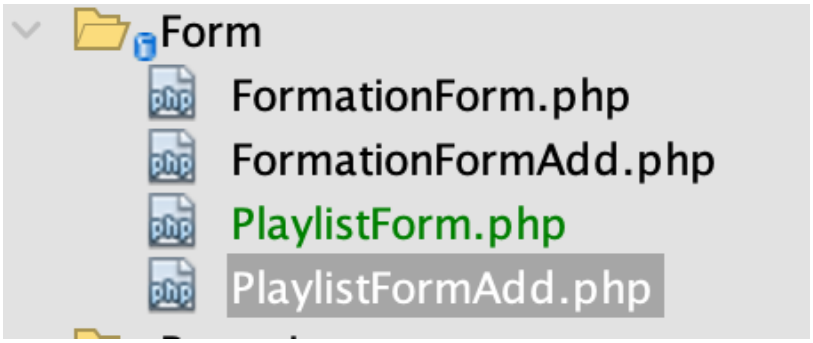
3) Ajouter des playlists

L'ajout de playlist utilise des mécanismes similaires, on crée la méthode dans le PlaylistBackController :

```php
/**
 * @Route ("/playlistback/add/1", name="playlistback.add")
 * @param Request $request
 * @return Response
 */
public function AjoutPlaylist(Request $request) : Response{
    $playlist = new Playlist();
    $form = $this->createForm(PlaylistFormAdd::class, $playlist);
    $form->handleRequest($request);

     if ($form->isSubmitted() && $form->isValid()) {
        // Enregistrer les modifications
        $this->getDoctrine()->getManager()->persist($playlist);
        $this->getDoctrine()->getManager()->flush();

        // Redirection vers la liste des formations
        return $this->redirectToRoute('playlistsBack');
    }

    return $this->render('pages/back/playlistbackadd.html.twig', [
        'form' => $form->createView(),
        'playlist' => $playlist // Ajoute la playlist au formulaire
    ]);
}
```

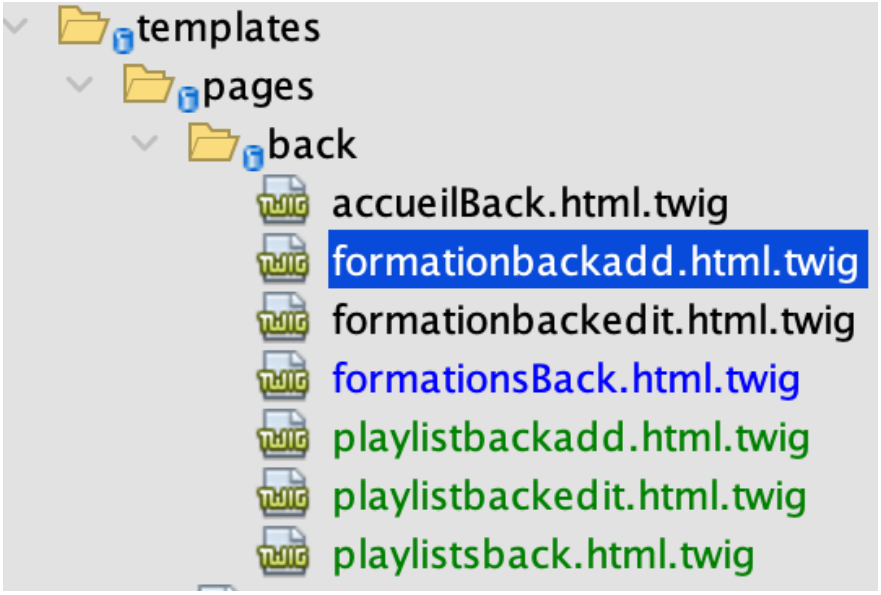
Cette fois-ci, on crée un nouvel objet de type playlist que l'on va remplir à l'aide d'un formulaire. Si le formulaire est validé et envoyé, on l'ajoute à la base de données.

On crée le type de formulaire : PlaylistFromAdd :



```php
class PlaylistFormAdd extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
            ->add('name',TextType::class,[
                'label'=>'Playlist',
                'attr' => ['maxlength'=> 100],
                'required' => true,
            ])//titre
            ->add('description', TextareaType::class,[
               'label' => 'Description',
               'attr' => ['rows' => 6],
               'required' => false,
            ])//description

            ->add('Ajouter',SubmitType::class);
    }
    public function configureOptions(OptionsResolver $resolver) {
        $resolver->setDefaults([
            'data_class' => Playlist::class,
        ]);
    }
}
```

On doit ensuite s'occuper du twig qui permettra son affichage, playlistbackadd.html.twig




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

{% block body %}
    <h2>Ajouter une Playlist</h2>
    {{ form_start(form) }}
        {{ form(form) }}
        {{ form_errors(form) }}
    {{ form_end(form) }}
{% endblock %}
```

L'utilisation de {{form(form)}} permet l'affichage de tous les champs prévus.

Enfin, il nous faut ajouter le bouton dans la page playlistsback.html.twig :

```
<tbody>
    <tr class="align-middle">
        <td>
            <h5 class="text-info">
                Ajouter une playlist
            </h5>
        </td>
        <td>

        </td>
        <td>

        </td>
        <td>

        </td>
        <td>

        </td>
        <td>

        </td>
        <td class="text-center">
            <a href="{{ path('playlistback.add') }}" class="btn btn-primary btn-sm">
                <span>+</span>
            </a>
        </td>
    </tr>
    <!-- boucle sur les playlists -->
```

On ajoute ainsi au-dessus des autres lignes, un intitulé : “Ajouter une playlist” et un bouton + renvoyant vers la méthode d’ajout du PlaylistsBackController.

c) Tâche 3 : gérer les catégories

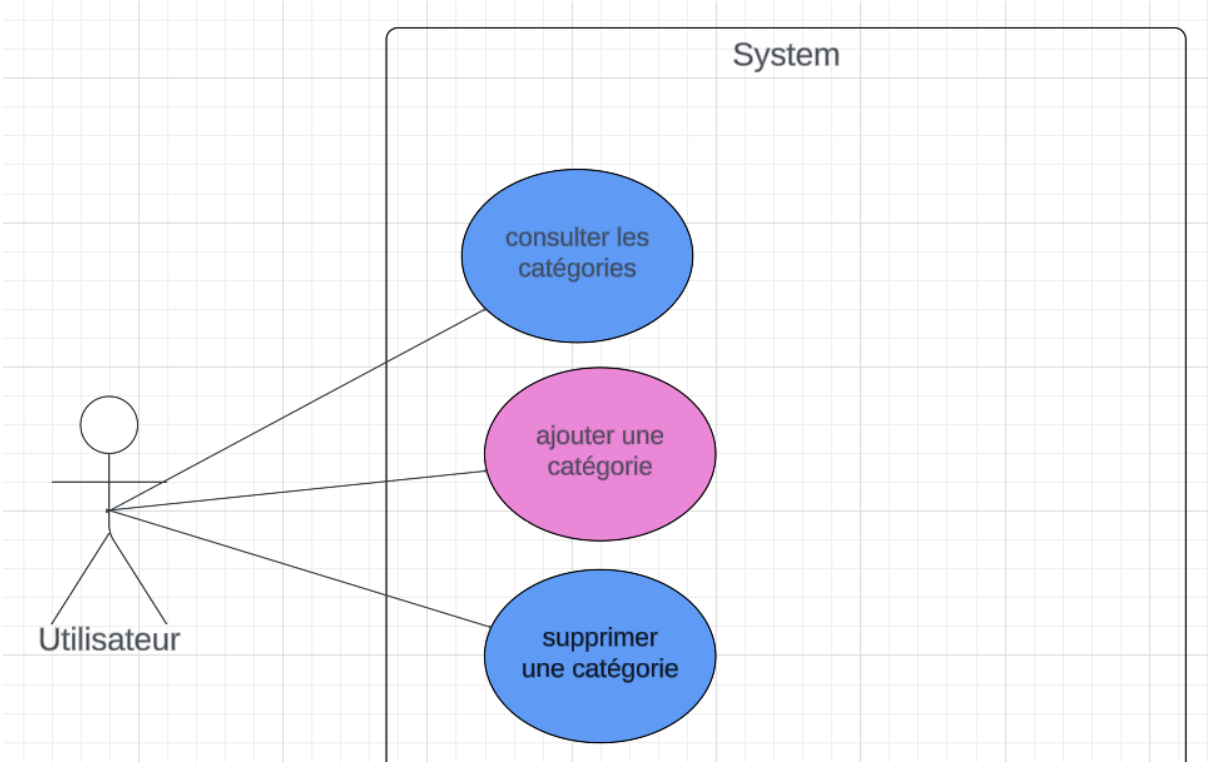
mediatekformation #9

gérer les catégories #9

Open **jnyazbek** opened 1 minute ago

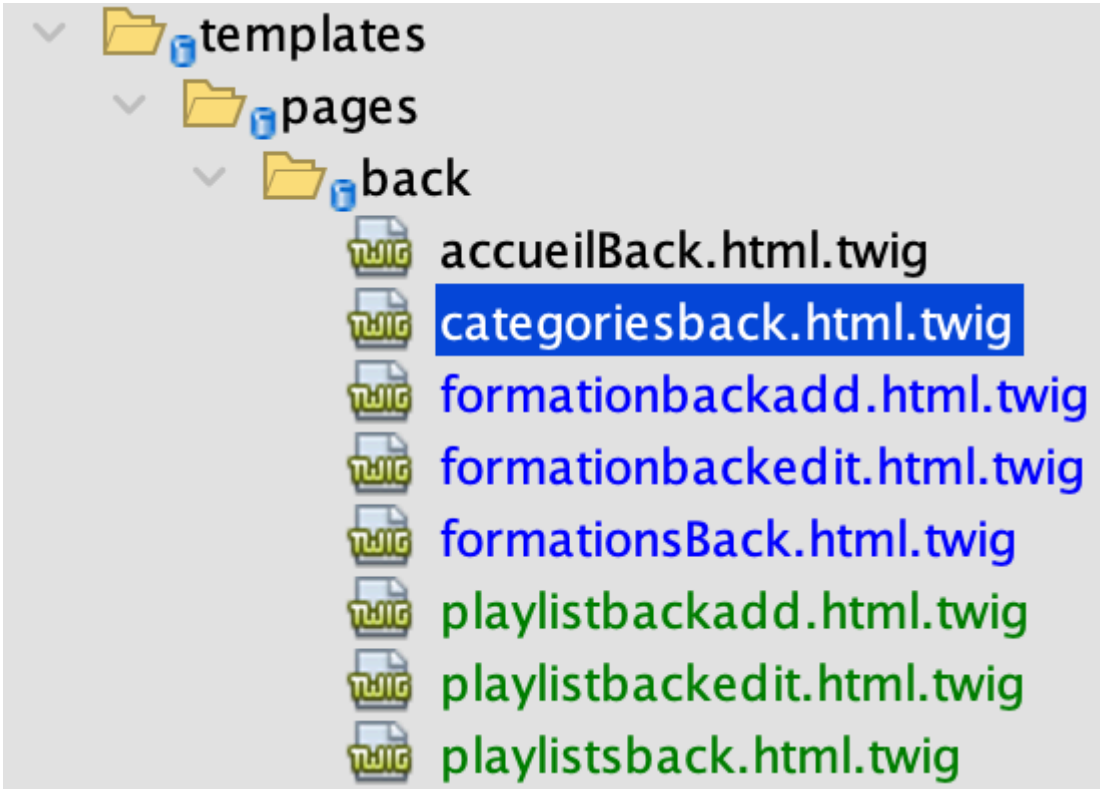
jnyazbek now (edited) Edit

Tâche 3 : gérer les catégories
Une page doit permettre de lister les catégories et, pour chaque catégorie, afficher un bouton permettant de la supprimer.
Attention, une catégorie ne peut être supprimée que si elle n'est rattachée à aucune formation.
Dans la même page, un mini formulaire doit permettre de saisir et d'ajouter directement une nouvelle catégorie, à condition que le nom de la catégorie n'existe pas déjà.



1) Affichage des catégories et des formations associées

Il nous faut premièrement créer le fichier twig : categoriesback.html.twig



```twig
{% extends "baseback.html.twig" %}

{% block body %}
    <h2>Gestion des Catégories</h2>

    <!-- Formulaire pour ajouter une nouvelle catégorie -->
    {{ form_start(form,{'action' : path('categoriesback.add'), 'method' : 'POST'}) }}
        {{ form(form) }}
    {{ form_end(form) }}

    <hr>

    <!-- Liste des catégories avec les formations associées -->
    <table class="table">
        <thead>
            <tr>
                <th>Nom de la Catégorie</th>
                <th>Formations Associées</th>
                <th>Actions</th>
            </tr>
        </thead>
```

On prévoit déjà l'affichage du formulaire d'ajout (sur lequel nous reviendrons plus tard) et on crée les colonnes Catégories, Formation et Action (dans laquelle le bouton "Supprimer" figurera).

Pour l'affichage des donnée on remplit le corps de la page ainsi :

```twig
        <tbody>
            {% for categorie in categories %}
                <tr>
                    <td>{{ categorie.name }}</td>
                    <td>
                        {% for formation in categorie.formations %}
                            {{ formation.title }}{% if not loop.last %}, {% endif %}
                        {% endfor %}
                    </td>
                    <td>
                        <!-- Bouton de suppression conditionnelle -->
                        <button onclick="if(confirm('Êtes-vous sûr de vouloir supprimer cette catégorie ?') && {{ categorie.formations|length == 0 }}) {
                            window.location.href = '{{ path('categoriesback.delete', {id:categorie.id}) }}'; }" class="btn btn-danger btn-sm">Supprimer</button>
                    </td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

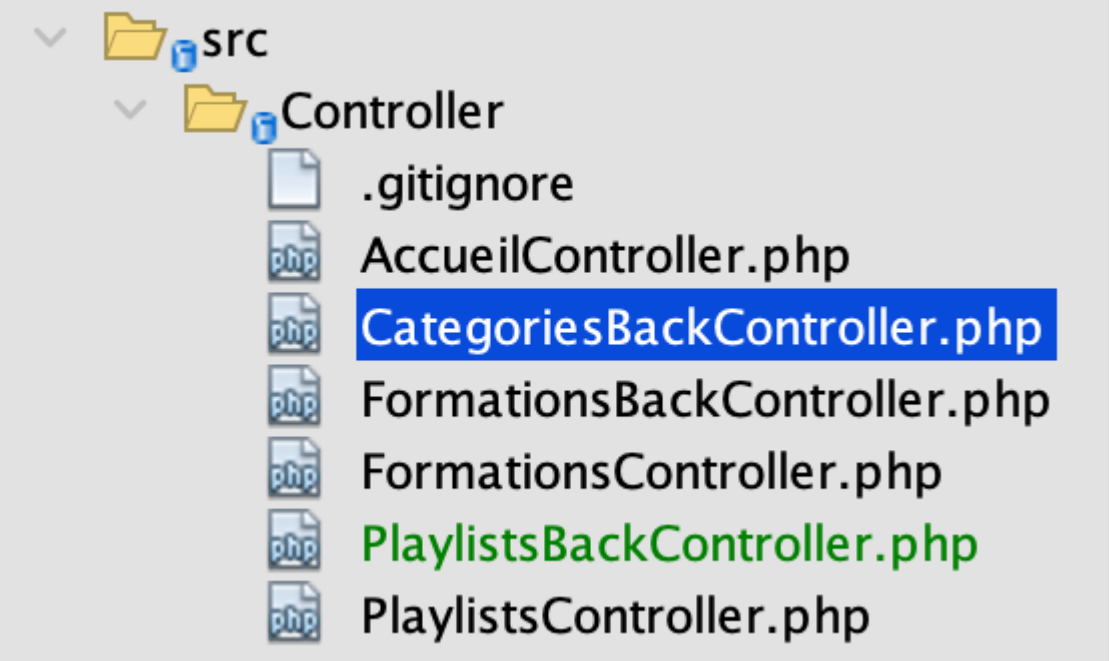
On itère dans la liste des catégories et on affiche leur nom et formations associées dans leurs colonnes respectives.
La partie du code {% if not loop.last %} permet d'ajouter une virgule entre les noms de formations tant que ce n'est pas la dernière de la liste.

Enfin le bouton de suppression est ajouté : il demande confirmation, vérifie qu'aucune formation n'est associée à la catégorie et appelle la route categoriesback.delete.

Ce qui nous amène sur le sujet de la suppression.

2) Suppression des catégories et des formations associées

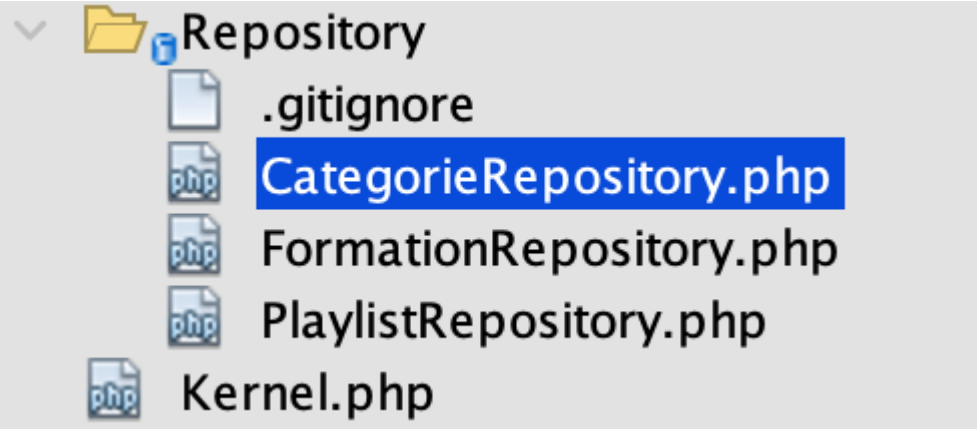
Nous commençons par créer le CategoriesBackController.



Ce contrôleur doit posséder un méthode permettant la suppression de catégories :

```php
/**
 * @Route("/categoriesback/{id}", name="categoriesback.delete")
 * @param type $id
 * @return Response
 */
public function deleteOne($id): Response {
$categorie = $this->categorieRepository->find($id);

if ($categorie && $categorie->getFormations()->isEmpty()) {

        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->remove($categorie);
        $entityManager->flush();
    }
    return $this->redirectToRoute('categoriesBack');
}
```

Cette dernière prend l'id d'une catégorie en paramètre, la recherche dans la base de donnée en utilisant la méthode find() de CategorieRepository.

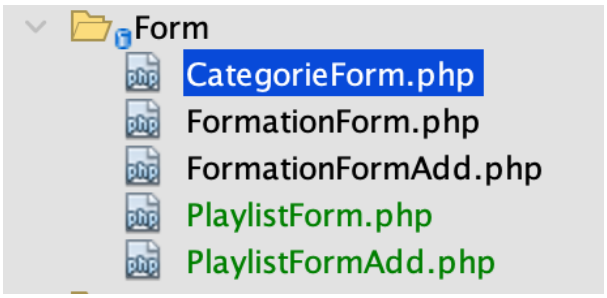


On vérifie ensuite dans le if() que la catégorie existe et qu'aucune formation n'y est rattachée. Lorsque ces conditions sont réunies, nous supprimons la catégorie de la base de données.

3) Ajout des catégories et des formations associées

Nous avons déjà prévu l'affichage du formulaire sur la page categoriesback.html.twig

Nous devons ensuite créer la classe qui dictera le contenu du formulaire.



```php
class CategorieForm extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
            ->add('name',TextType::class,[
                'label'=>'Catégorie',
                'attr' => ['maxlength'=> 100],
                'required' => true,
            ])//titre

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

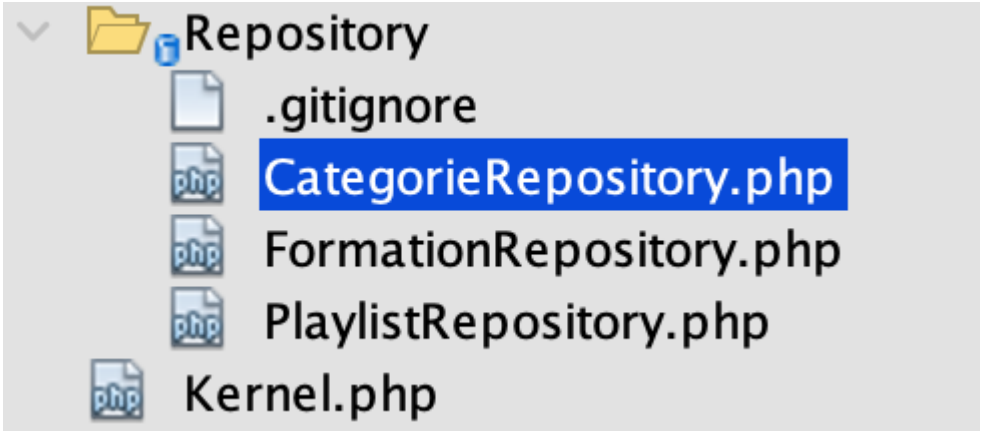
Une fois le formulaire prévu, il faut gérer son interaction avec la base de donnée dans le CategoriesBackController
On écrit dont la méthode AjouteCategorie()

```php
  /**
   * @Route ("/categoriesback/add/1", name="categoriesback.add")
   * @param Request $request
   * @return Response
   */
  public function AjoutCategorie(Request $request) : Response {
   $categorie = new Categorie();
   $formCategorie = $this->createForm(CategorieForm::class, $categorie);
   $formCategorie->handleRequest($request);

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

       $categorieExistante = $this->categorieRepository->findOneByName($categorie->getName());
       if (!$categorieExistante) {
           $entityManager = $this->getDoctrine()->getManager();
           $entityManager->persist($categorie);
           $entityManager->flush();

           return $this->redirectToRoute('categoriesBack');

   }

   return $this->render("pages/back/categoriesback.html.twig", [
       'form' => $formCategorie->createView(),
       'categories' => $categorie
   ]);
  }
```

On initialise un nouvel objet Catégorie, on crée le formulaire et on obtient les informations de la requête.
On récupère les données de la requête POST avec le contenu du formulaire.
Une fois le formulaire envoyé, on doit vérifier qu'aucune catégorie de même nom n'existe.

On a donc besoin d'une méthode dans le CategorieRepository qui prend une string en paramètre et qui retourne une catégorie de la base de données ayant le même nom ou null dans le cas non échéant.

.




```php
public function findOneByName($name): ?Categorie {
 return $this->createQueryBuilder('c')
              ->where('c.name = :name')
              ->setParameter('name', $name)
              ->getQuery()
              ->getOneOrNullResult();
 }
```

? pour nullable.

Si aucune catégorie de même nom n'existe, alors on enregistre la nouvelle catégorie créée dans la base de données.

d) Tâche 4 : ajouter l'accès avec authentification

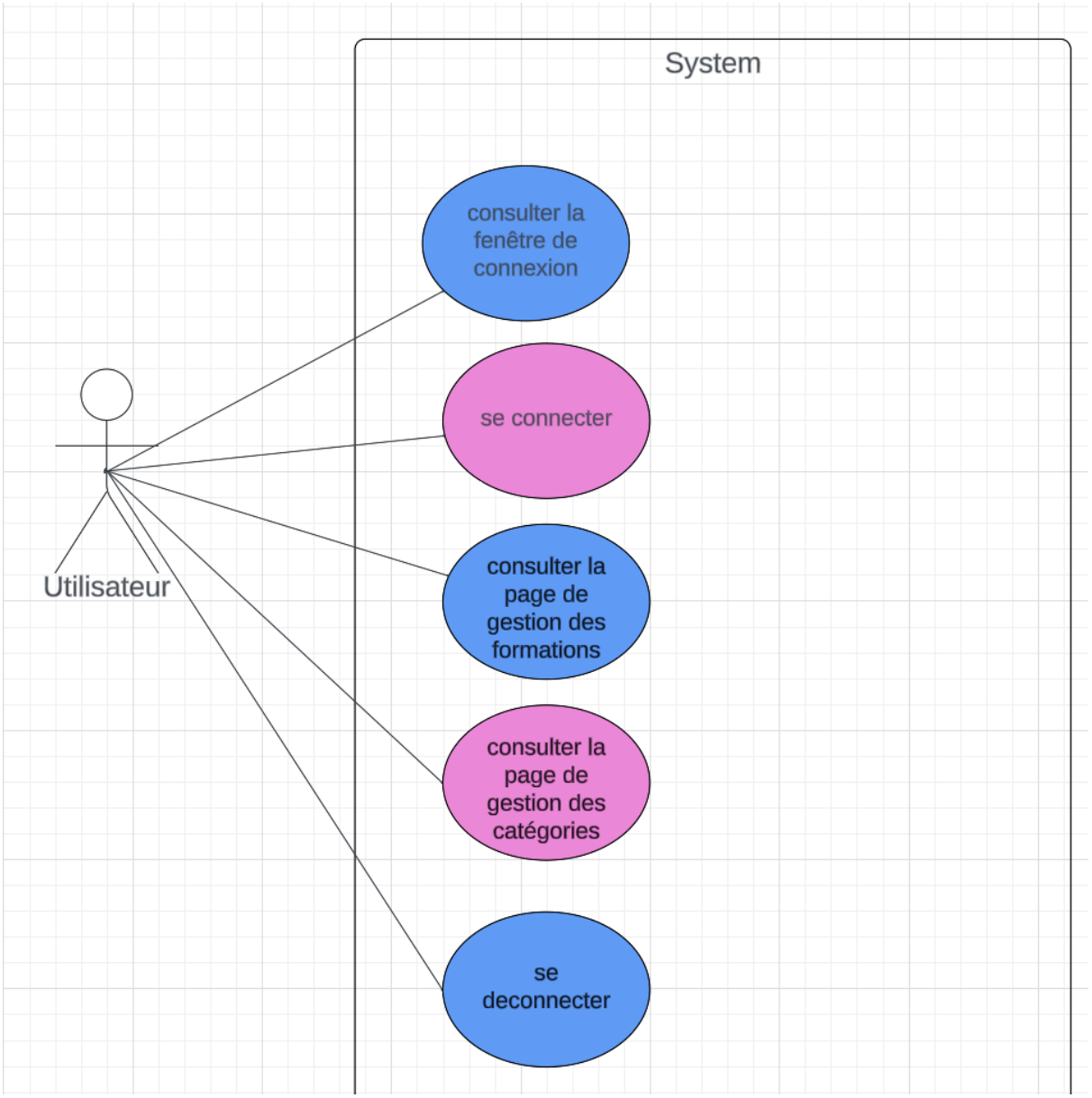
mediatekformation #10

ajouter l'accès avec authentification #10

Open **jnyazbek** opened now

jnyazbek now (edited) Edit

Tâche 4 : ajouter l'accès avec authentification
Le back office ne doit être accessible qu'après authentification : un seul profil administrateur doit avoir le droit d'accès. Pour gérer l'authentification, utiliser Keycloak.
Il doit être possible de se déconnecter, sur toutes les pages (avec un lien de déconnexion).



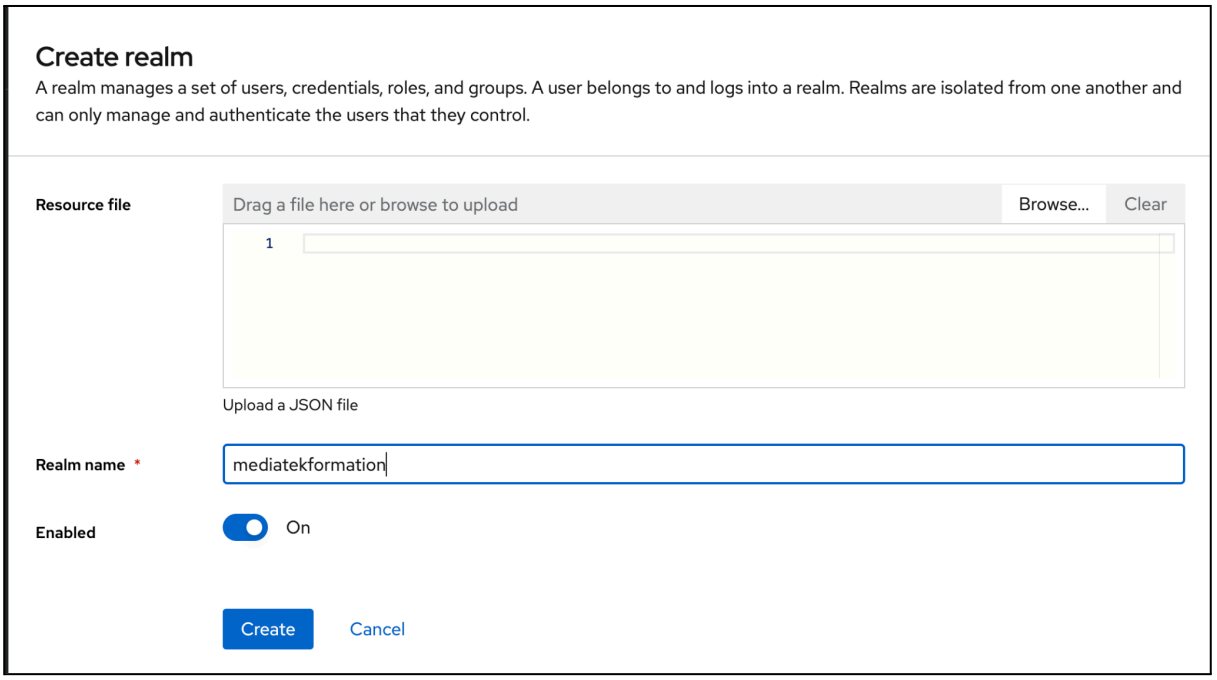
1) Configuration de Keycloak

Après avoir téléchargé Keycloak, nous devons le paramétrer : on se déplace dans le dossier bin de Keycloak dans le terminal et on lance le fichier kc.sh.

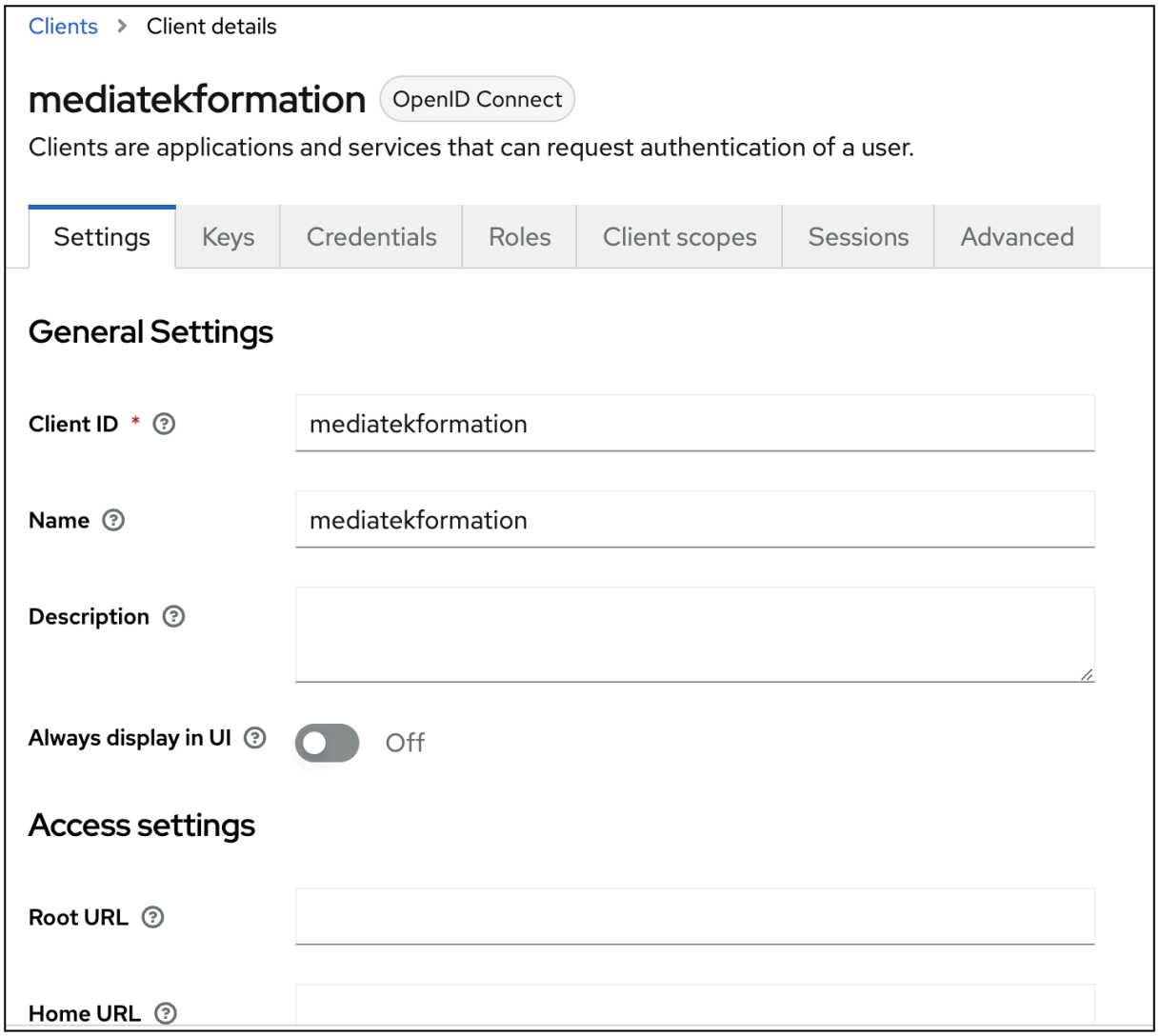
```
joseph-nicolasyazbek@MacBook-Air-de-Joseph-Nicolas bin % ./kc.sh start-dev
Updating the configuration and installing your custom providers, if any. Please wait.
```

Le serveur se lance : nous ouvrons la page "http://localhost:8080", ce qui nous donne accès à la console d'administration de Keycloak.

On commence par créer un nouveau realm que l'on appelle "formation". Dans ce realm, on crée le client mediatekformation.



La configuration du client se fait ainsi :



Access settings

Root URL

Home URL

Valid redirect URIs /*

Add valid redirect URIs

Valid post logout redirect URIs

Add valid post logout redirect URIs

Web origins /*

Add web origins

Admin URL

Capability config

Capability config

Client authentication On

Authorization Off

Authentication flow

- [x] Standard flow
- [x] Direct access grants
- [ ] Implicit flow
- [ ] Service accounts roles
- [ ] OAuth 2.0 Device Authorization Grant
- [ ] OIDC CIBA Grant

Login settings

Login theme — Choose...

Consent required — On

Display client on screen — On

Client consent screen text

Logout settings

Front channel logout — On

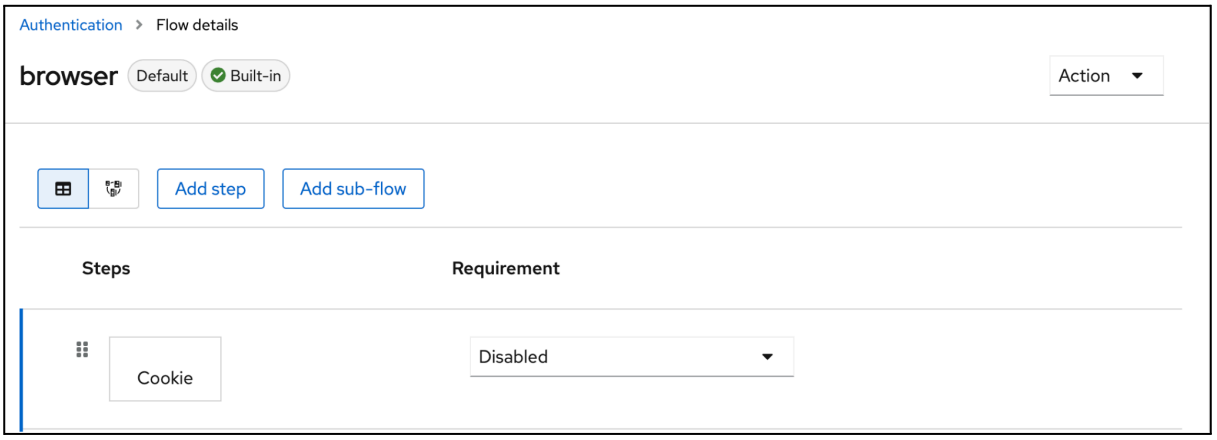
Front-channel logout URL

Backchannel logout URL

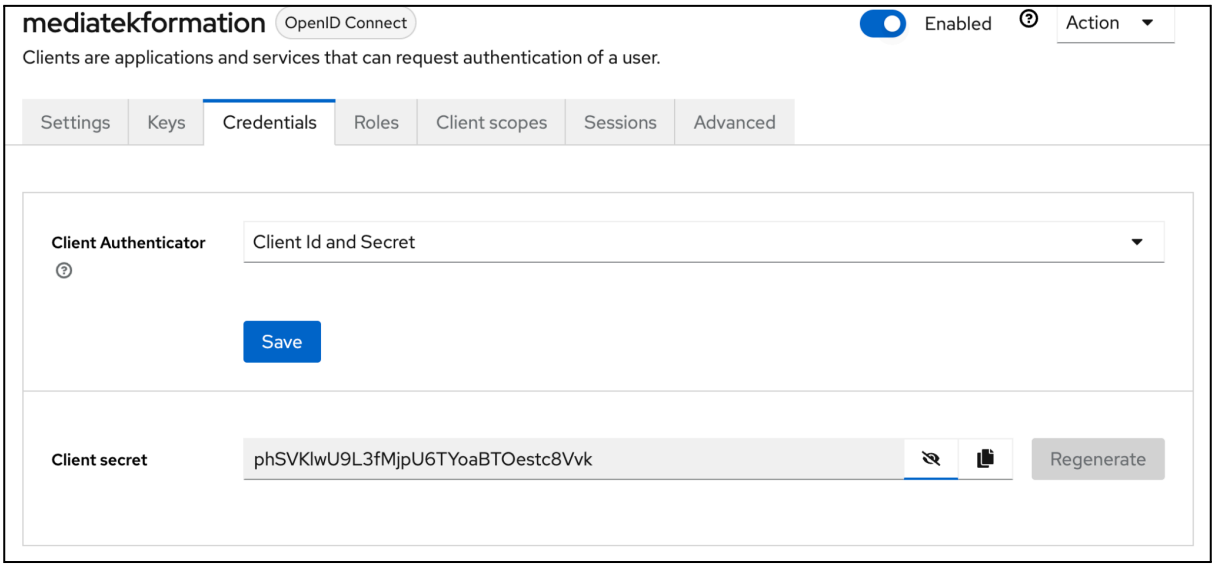
Backchannel logout session required — On

Backchannel logout revoke offline sessions — Off

On désactive également les cookies :



On récupère le client secret pour usage ultérieur :
phSVKlwU9L3fMjpU6TYoaBTOestc8Vvk

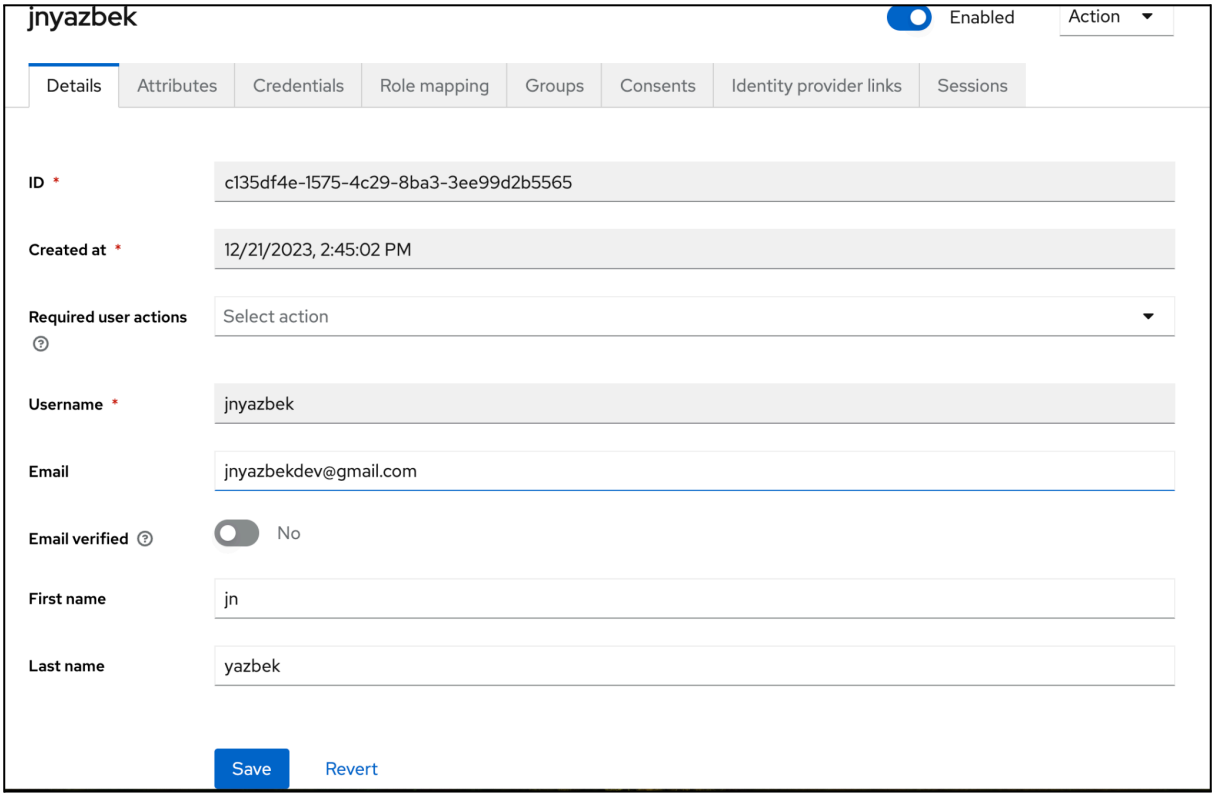


Une fois cela fait, on va enregistrer l'utilisateur dans la base de donnée : il faut pour cela créer une table prévue à cet effet.

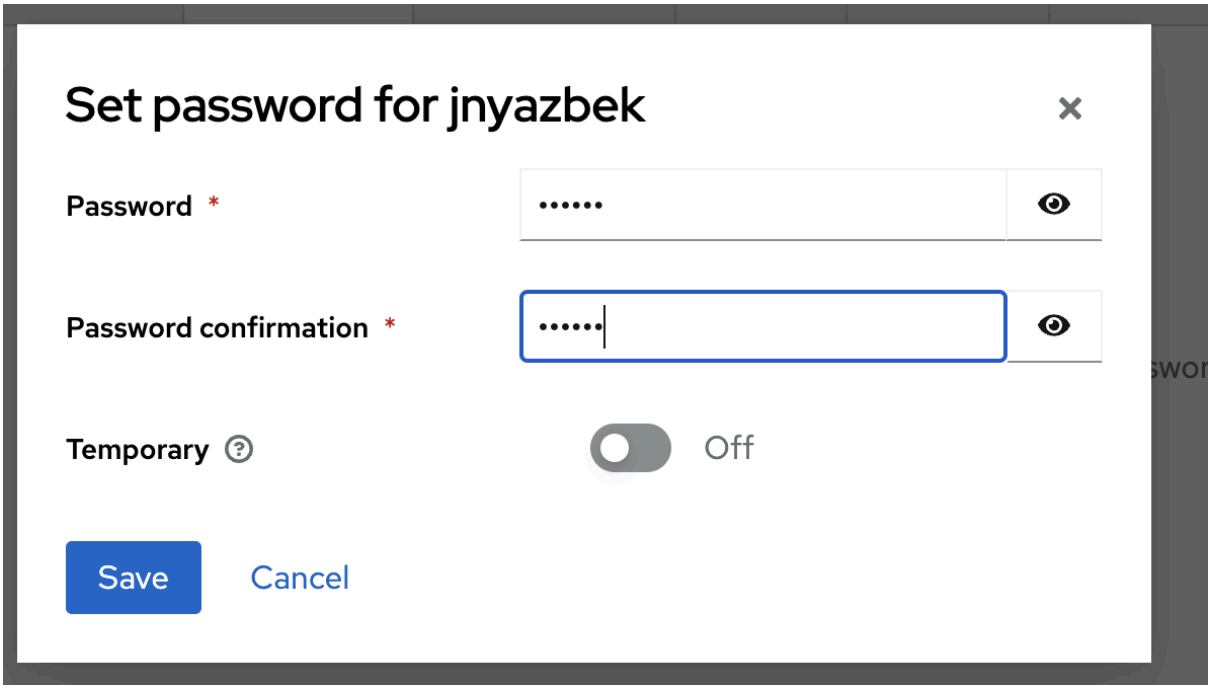
```
joseph-nicolasyazbek@MacBook-Air-de-Joseph-Nicolas bin % cd /Applications/XAMPP/xamppfiles/htdocs/mediatekformation
joseph-nicolasyazbek@MacBook-Air-de-Joseph-Nicolas mediatekformation % php bin/console make:user
```

Avec la commande make:user, nous venons de créer la table.

Il est maintenant temps de créer un nouvel utilisateur :



On attribue également le mot de passe dans l'onglet credentials.



2) Configuration du projet

Dans un premier temps, il nous faut ajouter le client mediatekformation de Keycloak dans notre projet. Pour cela il faut ajouter les lignes suivantes au fichier .env :

KEYCLOAK_SECRET=phSVKIwU9L3fMjpU6TYoaBTOestc8Vvk
KEYCLOAK_CLIENT = mediatek-formation
KEYCLOAK_APP_URL=http://localhost:8080

Ensuite, nous devons créer la table user dans la base de donnée afin de pouvoir y enregistrer ces derniers : on retourne donc dans le dossier mediatekformation et on utilise la commande make:user.

```
joseph-nicolasyazbek@MacBook-Air-de-Joseph-Nicolas bin % cd /Applications/XAMPP/xamppfiles/htdocs/mediatekformation
joseph-nicolasyazbek@MacBook-Air-de-Joseph-Nicolas mediatekformation % php bin/console make:user
```

Une fois la table créée, il nous faut également la paramétrer et prévoir les colonnes et types de données qui vont la composer.
Les id seront générés automatiquement.

On utilise la commande make:entity user et on ajoute le paramètre suivant :

```
joseph-nicolasyazbek@AirdeJohNicolas mediatekformation % php bin/console make:entity user

 Your entity already exists! So let's add some new fields!

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

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

 Field length [255]:
 > 255

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

 updated: src/Entity/User.php
```

On crée ensuite la migration et on l'effectue:

```
joseph-nicolasyazbek@AirdeJohNicolas mediatekformation % php bin/console make:entity user

 Your entity already exists! So let's add some new fields!

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

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

 Field length [255]:
 > 255

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

 updated: src/Entity/User.php
```


```
joseph-nicolasyazbek@AirdeJohNicolas mediatekformation % php bin/console make:entity user

 Your entity already exists! So let's add some new fields!

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

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

 Field length [255]:
 > 255

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

 updated: src/Entity/User.php
```


On utilise ensuite la commande composer require knpuniversity/oauth2-client-bundle 2.10 afin d'ajouter un bundle à notre composer, car ce dernier permettra de créer un fichier permettant la configuration de l'authentification avec Keycloak.

On ajoute egalement le package stevenmaguire/oauth2-keycloak 3.1-with-all-dependencies

On ouvre le fichier "knpu_oauth2_client. yaml" (qui est dans "config > packages") et on le configure en faisant références aux informations saisie dans le fichier .env et celles saisies lors de la création du realm dans KeyCloak :

```yaml
knpu_oauth2_client:
    clients:
        keycloak:
            type: keycloak
            auth_server_url: '%env(KEYCLOAK_APP_URL)%'
            realm: 'formation'
            client_id: '%env(KEYCLOAK_CLIENTID)%'
            client_secret: '%env(KEYCLOAK_CLIENT_SECRET)%'
            redirect_route: 'oauth_check'
            # configure your clients as described here:
                https://github
                .com/knpuniversity/oauth2-client-bundle#configuration
```

Il faut à présent ajouter la route d'authentification au fichier de configuration security.yaml
ainsi que le rôle à avoir pour accéder à ce chemin dans access_control :

```yaml
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory
            form_login :
                login_path : oauth_login

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

            # 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 }
```

3) Création du controller

Il faut en effet créer le Controller qui gérera cette authentification.
Pour cela on passe par le terminal et on utilise la commande :
php bin/console make:controller OAuthController –no-template

```
joseph-nicolasyazbek@AirdeJohNicolas mediatekformation % php bin/console make:controller OAuthController --no-template

 created: src/Controller/OAuthController.php

 Success!

 Next: Open your new controller class and add some pages!
joseph-nicolasyazbek@AirdeJohNicolas mediatekformation %
```

On ouvre le fichier ainsi créé OAuthController.php situé dans src/Controller.
La méthode index, appelée au moment de la demande de connexion, doit recevoir un paramètre (de type ClientRegistry) et retourner un objet de type RedirectResponse.
Cette méthode récupère le client 'keycloak' de l'objet reçu en paramètre et appelle la méthode "redirect" sur cet objet,
On redirige ainsi la requête vers l'authentification gérée par Keycloak.

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;

class OAuthController extends AbstractController
{
    /**
     * @Route("/oauth/login",name="oauth_login")
     * @return RedirectResponse
     */
    public function index(ClientRegistery $clientRegistery): RedirectResponse
    {
        return $clientRegistery->getClient('keycloak')->redirect();
    }

    /**
     *
     * @Route("/oauth/callback", name="oauth_check")
     */
    public function connectCheckAction(Request $request, ClientRegistery $clientRegistery){

    }
}
```

La méthode connectCheckAction va prendre en charge la route de redirection du retour. On va laisser son contenu vide, car la méthode est gérée automatiquement par Symfony.

La méthode index de l'OAuthController appelle la fonction redirect du client keycloak afin de lancer le processus d'authentification.
Keycloak envoie alors un code qu'il nous faut récupérer et échanger contre un access token.

Pour gérer cela, on crée la classe : KeycloakAuthenticator.php.
Cette classe hérite de OAuth2Authenticator et implémente l'interface AuthenticationEntryPointInterface dont on doit ensuite définir les méthodes .

On commence par la création du constructeur qui nous permet de valoriser les propriétés dont nous aurons besoin :

```php
class KeycloakAuthenticator extends OAuth2Authenticator implements AuthenticationEntryPointInterface {

    private $clientRegistery;
    private $entityManager;
    private $router;

    public function __construct(ClientRegistery $clientRegistery,EntityManagerInterface $entityManager, RouterInterface $router) {
        $this->clientRegistery = $clientRegistery ;
        $this->entityManager = $entityManager;
        $this->router = $router;
    }
```

On passe ensuite à la méthode "start" qui permet de démarrer une authentification. Elle envoie donc vers une route temporaire qui est définie dans le OAuthController et donc qui va solliciter Keycloak.

```php
public function start(Request $request, AuthenticationException $authException = null): Response {
    return new RedirectResponse(
            '/oauth/login',
            Response:: HTTP_TEMPORARY_REDIRECT
    );
}
```

La méthode "supports" quant à elle, renvoie true si le système d'authentification doit se déclencher pour l'url donnée. Elle est donc appelée à chaque fois qu'une requête est reçue (une url sollicitée). On vérifie si l'url donnée correspond à 'oauth_check' : dans ce cas la méthode retourne true , la méthode getCredentials() est alors appelée automatiquement :

```php
//appelle getCredentials() si true
public function supports(Request $request): ?bool {
    return $request->attributes->get('_route') === 'oauth_check';
}
```

La méthode authenticate() s'occupe d'abord de récupérer le client correspondant, dans Keycloak, et le token, à partir des informations données.
A partir de ces informations, trois scénarios sont possibles :

- L'utilisateur existe déjà dans la BDD et s'est déjà connecté avec Keycloak. Dans ce cas, on le récupère à l'aide de son ID dans la BDD et la méthode le retourne. Il faut que l'ID Keycloak de l'utilisateur dans la BBD corresponde à celui récupéré avec Keycloak.

- L''utilisateur existe dans la BDD mais ne s'est jamais connecté avec Keycloak. On récupère le username avec keycloak, et on le recherche dans la BDD. Si on l'y trouve, on valorise son idkeycloak, on enregistre les modifications et on retourne l'utilisateur.

- L'utilisateur soit l'utilisateur n'existe pas encore dans la BDD et il faut alors le créer.

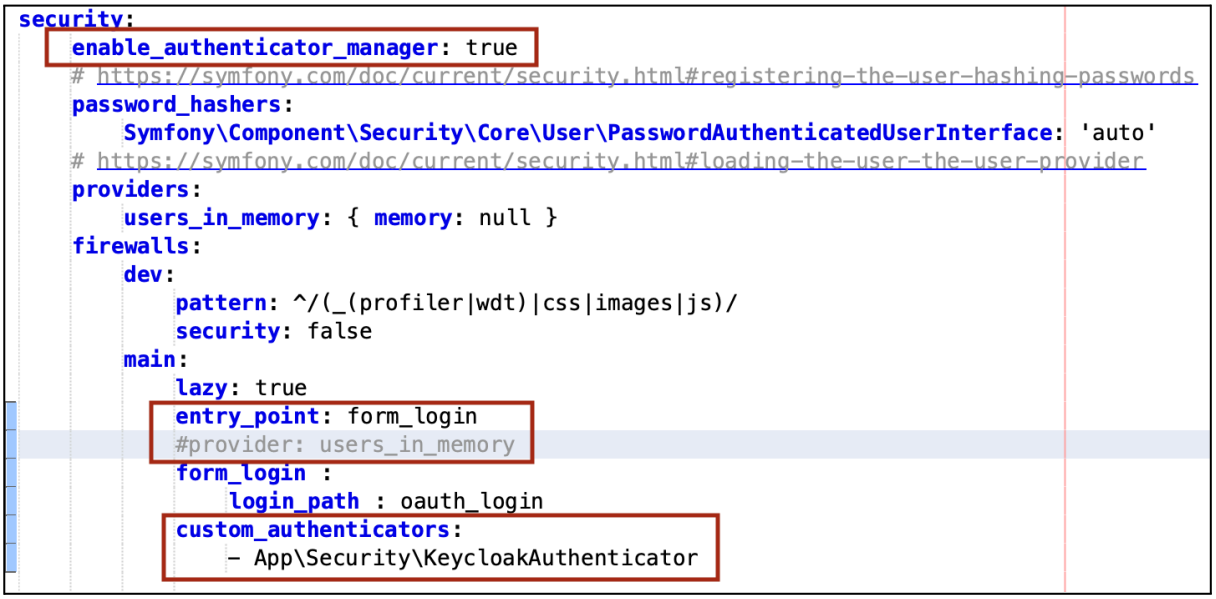
Dans tous les cas, la méthode retourne un "passeport" à partir d'un "badge"généré pour l'utilisateur.

```php
public function authenticate(Request $request): Passport {
    $client = $this->clientRegistery->getClient('keycloak');
    $accessToken = $this->fetchAcessToken($client);

    return new SelfValidatingPassport(
            //récupération de l'utilisateur
            new UserBadge($accessToken->getToken(), function() use ($accessToken,$client){
                /** @var KeycloakUser $keycloakUser$*/
                $keycloakUser = $client->fetchuserFromToken($accessToken);

                //cas où l'utilisateur existe déjà dans la bdd et s'est déjà connecté
                $existingUser = $this->entityManager
                                    ->getRepository(User::class)
                                    ->findOneBy(['idkeycloak'=> $keycloakUser->getId()]);
                if($existingUser){
                    return $existingUser;
                }

                //cas où l'utilisateur existe déjà dans la bdd mais ne s'est jamais connecté
                $username = $keycloakUser->getUsername();
                /** @var User $userInDatabase */
                $userInDatabase = $this->entityManager
                                        ->getRepository(User::class)
                                        ->findOneBy(['username' => $username]);
                if($userInDatabase){
                    $userInDatabase->setKeycloakId($keycloakUser->getId());
                    $this->entityManager->persist($userInDatase);
                    $this->entityManager->flush();
                    return $userInDatase;
                }

                // cas ou l'utilisateur n'existe pas dans la bdd
                $user = new User();
                $user->setKeycloakId($keycloakUser->getId());
                $user->setUsername($keycloakUser->getUsername());
                $user->setPassword('');
                $user->setRoles(['ROLE_ADMIN']);
                $this->entityManager->persist($user);
                $this->entityManager->flush();

                return $user;

            })
    );
```

La méthode "onAuthenticationFailure" s'exécute en cas de problème dans une autre méthode tandis que la méthode "onAuthenticationSuccess" s'exécute quand tout s'est bien passé. On peut alors rediriger vers la route voulue au départ, donc la partie 'admin' du site.

```php
public function onAuthenticationFailure(Request $request, AuthenticationException $exception): ?Response {
    $message = strtr($exception->getMessageKey(), $exception->getMessageData());
    return new Response($message, Response::HTTP_FORBIDDEN);
}

public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response {
    $targetUrl = $this->router->generate('formationsBack');
    return new RedirectResponse($targetUrl);
}
```

La classe étant codée, il nous faut à présent ajouter le chemin de cette dernière dans security.yaml.

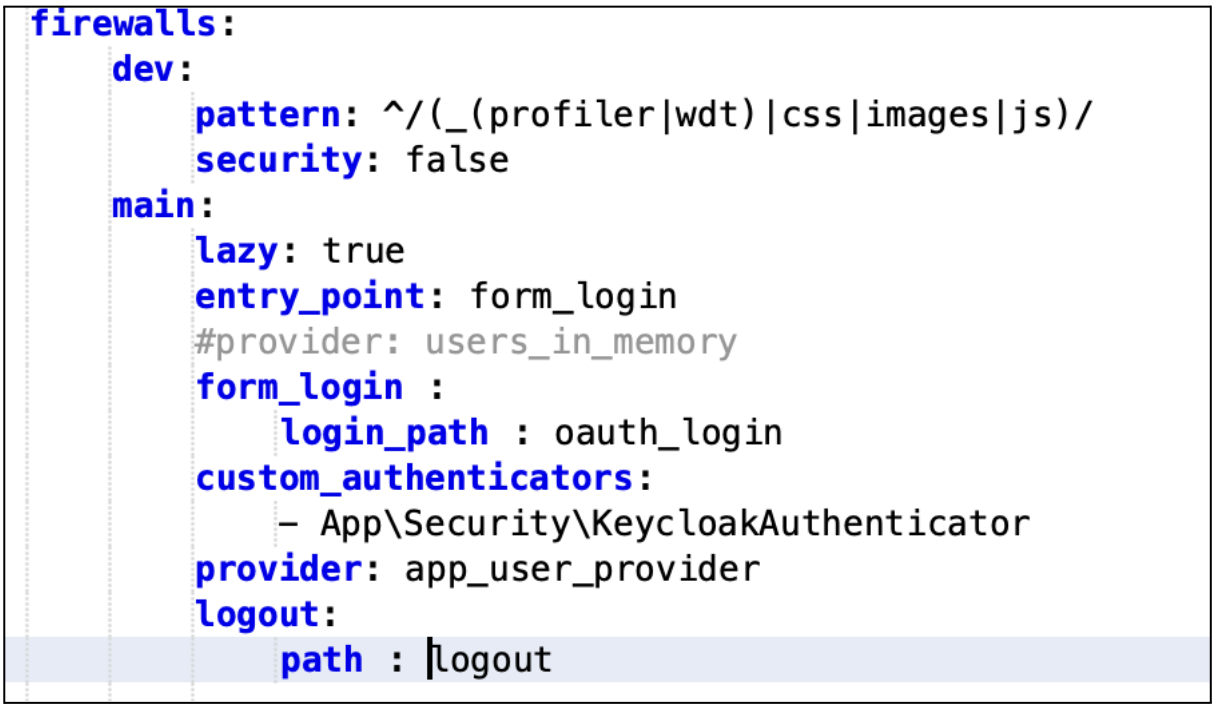


```yaml
security:
    enable_authenticator_manager: true
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            entry_point: form_login
            #provider: users_in_memory
            form_login :
                login_path : oauth_login
            custom_authenticators:
                - App\Security\KeycloakAuthenticator
```

Tout est en place, il est désormais possible d'accéder aux pages de gestion back de mediatekformation.

4) Configuration de la déconnexion

Enfin, il nous faut gérer la déconnexion.
On prépare en premier lieu le firewall dans le fichier security.yaml :



```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        entry_point: form_login
        #provider: users_in_memory
        form_login :
            login_path : oauth_login
        custom_authenticators:
            - App\Security\KeycloakAuthenticator
        provider: app_user_provider
        logout:
            path : logout
```

On ajoute la fonction logout à notre OAuthController :

```php
/**
 * @Route("/logout", name="logout")
 */
public function logout(){

}
```

Enfin, on ajoute un lien de déconnexion dans la navbar de baseback afin qu'il soit possible de se déconnecter depuis toutes les pages de back.

```twig
{% block top %}
    <div class="container">
        <!-- titre -->
        <div class="text-left">
            <img src="{{ app.request.getBasePath()~'/banniere.jpg' }}", alt="image de la banniere" >
        </div>
        <!-- menu -->
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
            <div class="collapse navbar-collapse" id="navbarSupportedContent">
                <ul class="navbar-nav mr-auto">
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('formationsBack') }}">Formations</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('playlistsBack') }}">Playlists</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('categoriesBack') }}">Catégories</a>
                    </li>
                    <li class="nav-item">
                         <a class="nav-link" href="{{ path('logout') }}">Se déconnecter</a>
                    </li>
```

Mission 3 : Tester et documenter

mediatekformation #11

Mission 3 Tâche 1: gérer les tests #11

Open **jnyazbek** opened now

jnyazbek now (edited) Edit

Tâche 1 : gérer les tests (7h)
Tests unitaires :
Contrôler le fonctionnement de la méthode qui retourne la date de parution au format string.
Tests d'intégration sur les règles de validation :
Lors de l'ajout ou de la modification d'une formation, contrôler que la date n'est pas postérieure à aujourd'hui.
Tests d'intégration sur les Repository :
Contrôler toutes les méthodes ajoutées dans les classes Repository (pour cela, créer une BDD de test).
Tests fonctionnels :
Contrôler que la page d'accueil est accessible.
Dans chaque page contenant des listes :
contrôler que les tris fonctionnent (en testant juste le résultat de la première ligne) ;
contrôler que les filtres fonctionnent (en testant le nombre de lignes obtenu et le résultat de la première ligne) ;
contrôler que le clic sur un lien (ou bouton) dans une liste permet d'accéder à la bonne page (en contrôlant l'accès à la page mais aussi le contenu d'un des éléments de la page).
Tests de compatibilité :
Créer un scénario avec Selenium, sur la partie front office, et le jouer sur plusieurs navigateurs pour tester la compatibilité du site.

a) Tâche 2 : générer la documentation

mediatekformation #12

créer la documentation technique #12

Open **jnyazbek** opened now

jnyazbek now (edited) Edit

Tâche 2 : créer la documentation technique (1h)
Contrôler que tous les commentaires normalisés nécessaires à la génération de la documentation technique ont été correctement insérés.
Générer la documentation technique du site complet : front et back office excluant le code automatiquement généré par Symfony (voir l'article "Génération de la documentation technique sous NetBeans" dans le wiki du dépôt).

Afin d'éviter de documenter le code automatiquement généré par symfony, nous créons un fichier phpdoc/xml qui configure le code à documenter :

Applications > XAMPP > xamppfiles > htdocs > mediatekformation > phpdoc.xml

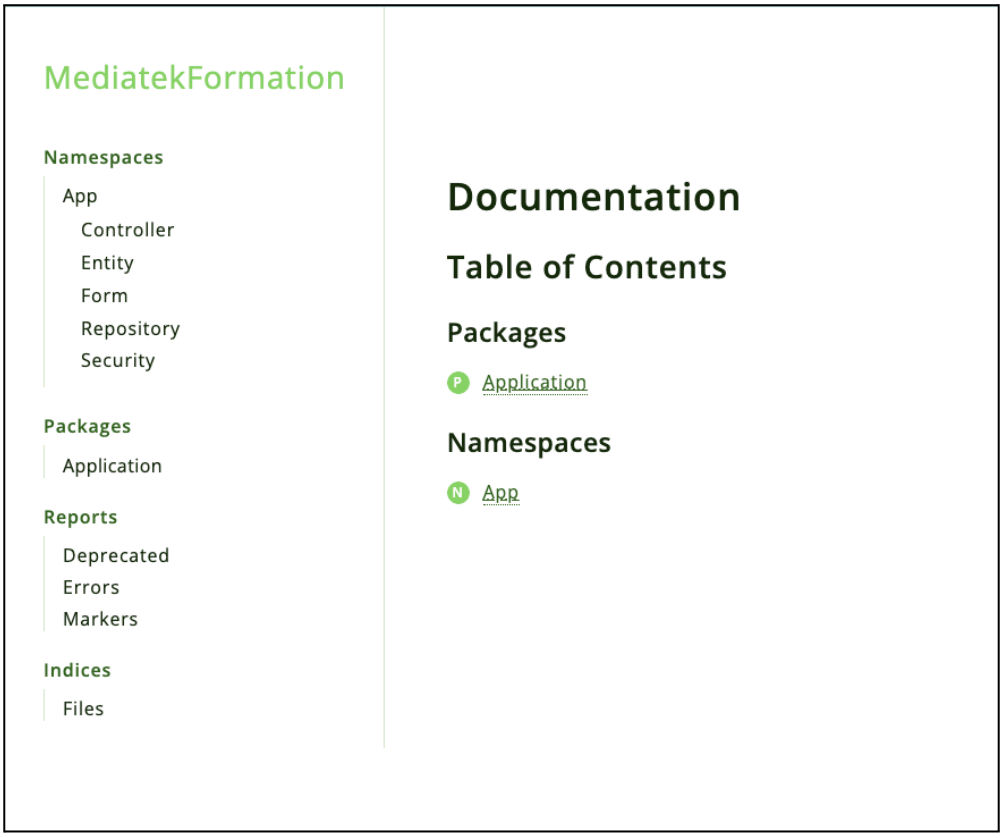
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<phpdocumentor>
    <title>Your Project Title</title>
    <files>
        <directory>src</directory>
        <ignore>vendor/*</ignore>
    </files>
    <transformer>
        <target>/Applications/XAMPP/xamppfiles/htdocs/mediatekformation/Documentation</target>
    </transformer>
    <transformations>
        <template name="clean"/>
    </transformations>
</phpdocumentor>
```

On utilise ensuite la commande :

```
joseph-nicolasyazbek@AirdeJohNicolas mediatekformation % php /Users/joseph-nicolasyazbek/phpDocumentor.phar --cache-folder=/Users/joseph-nicolasyazbek/.phpdoc/cache
phpDocumentor v3.4.3
```

php/Users/joseph-nicolasyazbek/phpDocumentor.phar–cache-folder=/Users/joseph-nicolasyazbek/.phpdoc/cache

La documentation est ainsi créée et nous pouvons la consulter en ouvrant l'index :



b) Tâche 3 : créer la documentation utilisateur

Lien de la vidéo :

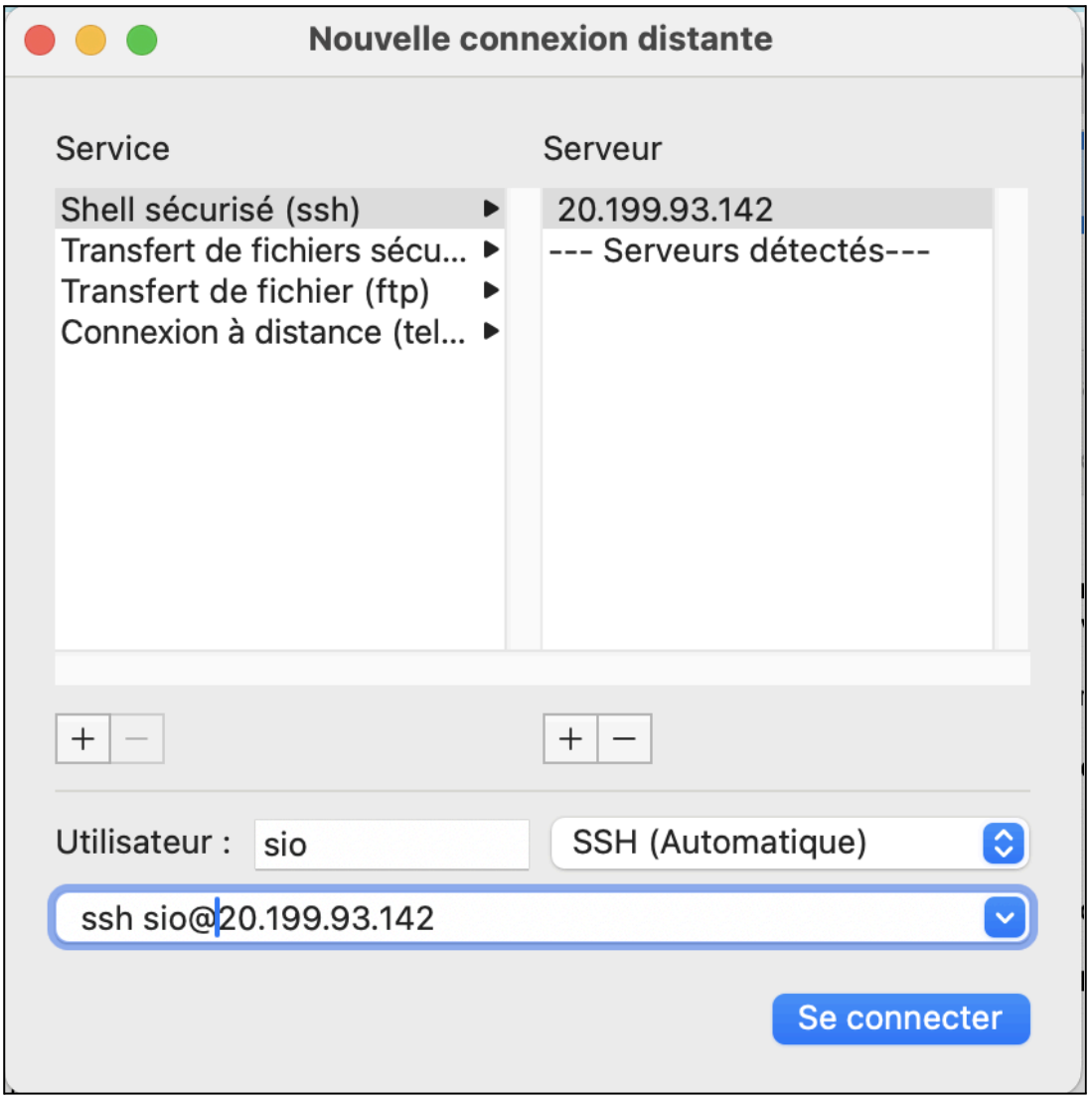
Mission 4 : Déployer le site et gérer le déploiement continu

a) Tâche 1 : déployer le site

1) Configuration de Keycloak HTTPS

À l'aide de microsoft azure, on commence par créer une machine virtuelle sur laquelle nous allons installer KeyCloak.

On lui attribue le dns : monkeycloak1.francecentral.cloudapp.azure.com/



En utilisant le terminal mac on se connecte à la vm pour installer le jdk et l'on crée la variable d'environnement JAVA_HOME.

Une fois java installé nous installons keycloak et apache.
Enfin pour pouvoir accéder au site en HTTPS, il nous faut installer cerbot.

On en profite pour paramétrer le compte et le mdp admin en utilisant les commandes :
export KEYCLOAK_ADMIN=nomutilisateur
export KEYCLOAK_ADMIN_PASSWORD=motdepasse

On peut désormais accéder à Keycloak en utilisant le dns.
Depuis la page keycloak, on crée les mêmes royaumes et utilisateurs que précédemment : formation et mediatekformation.

Il nous faut par la suite modifier notre fichier .env pour qu'il utilise la vm pour l'authentification avec keycloak.
KEYCLOAK_CLIENT_SECRET=LCBU6usVO8byRC0UPcN2E68e5UkPU5UI
KEYCLOAK_APP_URL=https://monkeycloak1.francecentral.cloudapp.azure.com
KEYCLOAK_CLIENTID=mediatekformation

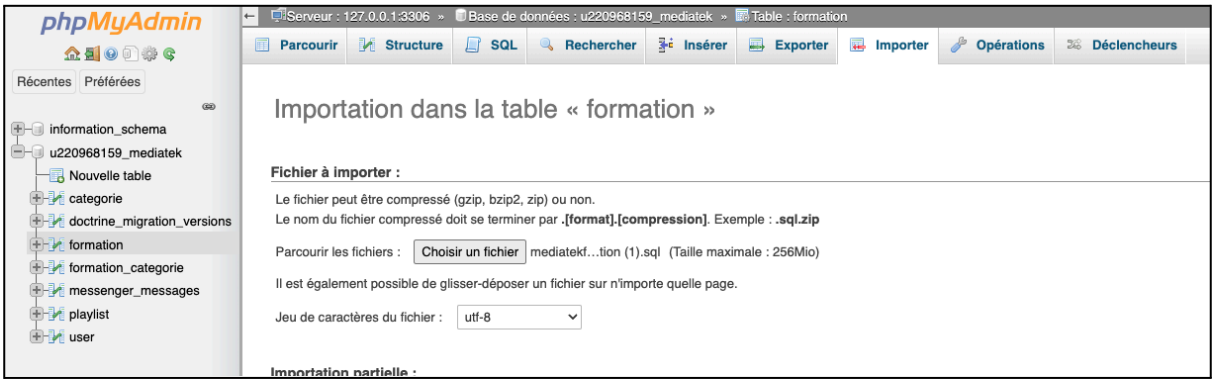
On en profite pour passer l'environnement en production :

```
###> symfony/framework-bundle ###
APP_ENV=prod
APP_SECRET=7abe96acf49eaf2b242b4242bca4d805
APP_DEBUG=1
###< symfony/framework-bundle ###
```

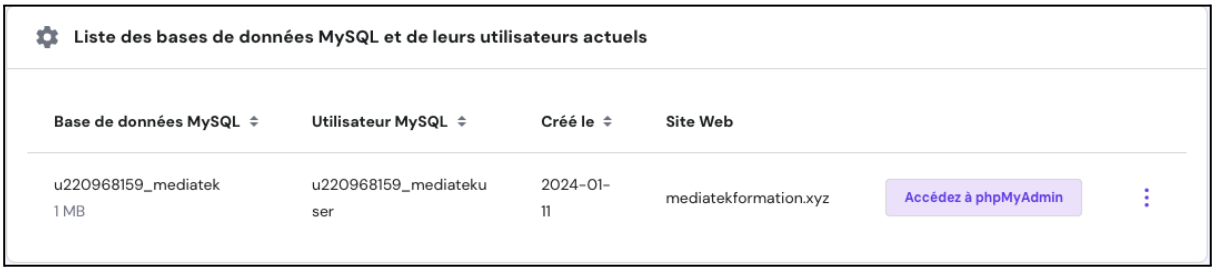
2) Déployer la base de données et le site web

Sur Hostinger (hébergement web), il suffit de créer une nouvelle base de données en lien avec notre site.
Il nous faut acheter un nom de domaine : mediatekformation.xyz.

Après avoir créé la BDD, il convient de réaliser un export de la bdd hébergée en locale puis de l'importer dans celle de notre site.



Afin qu'elle soit accessible depuis notre site il nous suffit de modifier le .env et d'y ajouter les information suivantes :



DATABASE_URL="mysql://u220968159_mediatekuser:mdp****@localhost:3306/u220968159_mediatek"

Afin que l'URL rewriting soit possible, on n'oublie pas d'utiliser la commande : composer require symfony/apache-pack qui génère le fichier .htaccess qui rend la chose possible.

Enfin, après avoir archivé les fichiers du sites, on upload le .zip dans la gestion des fichier de notre hébergeur pour le sy décompresser dans le dossier public_html.

Il ne manque plus qu'à gérer les redirections du nom de domaine afin que l'utilisateur accédant à https://mediatekformation.xyz soit redirigé vers : https://mediatekformation.xyz/public/index.php

b) Tâche 2 : gérer la sauvegarde et la restauration de la base de données

1) Configuration automatique des sauvegardes

Nous allons automatiser les sauvegardes journalières.

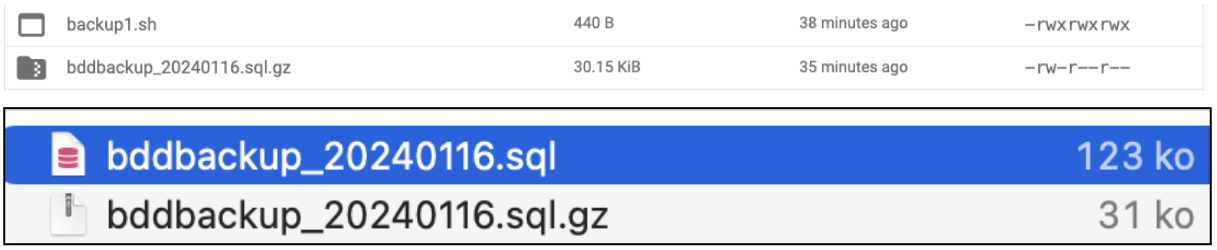
Pour cela, nous devons créer un script qui enregistrera la BDD. Il nous faut d'abord configurer un script en shell utilisant les identifiants de connexion à notre hébergeur et ceux de notre base de données.

Il nous faut ensuite programmer l'exécution journalière de ce script en utilisant la fonctionnalitée "tâche Cron" de Hostinger :

La commande suivante est en fait le chemin d'accès au fichier qui sera exécuté tous les jours grâce aux paramètres visibles à gauche.

2) Restauration manuelle de la BDD

Nous devons récupérer le fichier.gz créée par le script et extraire son contenu pour ensuite l'importer dans la BDD après l'avoir vidée.



c) Tâche 3 : mettre en place le déploiement continu (Github)

L'objectif est de mettre à jour automatiquement le site à chaque github push.
Il nous faut donc créer un workflow sur github qui se matérialise sous la forme d'un fichier .yaml

```
on: push
name: Deploy website on push
jobs:
 web-deploy:
 name: Deploy
 runs-on: ubuntu-latest
 steps:
 - name: Get latest code
 uses: actions/checkout@v2

 - name: Sync files
 uses: SamKirkland/FTP-Deploy-Action@4.3.0
 with:
 server: ftp://153.92.220.162
 server-dir: /home/u220968159/domains/mediatekformation.xyz
/public_html/
 username: u220968159
 password: ${{ secrets.ftp_password }}
```

Une fois ce fichier créé, on effectue un pull pour l'obtenir en local.
Il ne manque qu'à ajouter le mdp du ftp dans les secret de notre repository, pour cela, nous procédons ainsi : settings→secret→action→ add a new secret.

BILAN

Au cours de cet atelier professionnel nous avons pu faire évoluer le site en ajoutant des fonctionnalités permettant de mieux gérer les formations, les playlists et les catégories.

Le site web à été déployé, il est accessible et les administrateurs peuvent le gérer en s'identifiant.

Le site est en déploiement continu ce qui simplifiera son évolution, optimisation et modification.

La mission est remplie est le réseau de médiathèques viennoise s’en trouve amélioré.