



PROJET MEMPA

ANGULAR - NODE.JS

L3 MIAGE

Anthony Rodrigues
Lucas Garlaschi
Maxime Veschembes

Sommaire

I. Front-end : Angular

1. Liste des composants	4
2. Liste des services	4
3. Les Routes	5
4. Schémas pour chaque page de l'application	6
5. Utilisation de FormBuilder	8
6. Utilisation de Guards	8

II. Back-end : Node.js

1. Le principe d'une API	9
2. Stockage des données dans une ArrayList	9
3. Les routes	9

Introduction

Dans le cadre de notre formation en développement web, nous avons été chargés de réaliser un projet d'application de playlists musicales. Ce projet nous a offert l'opportunité de mettre en pratique les connaissances acquises lors des cours et de travailler ensemble sur une application concrète et fonctionnelle.

L'objectif principal de cette application était de permettre aux utilisateurs de créer et gérer des playlists de morceaux de musique. Pour cela, nous avons développé une interface utilisateur intuitive et réactive, ainsi qu'un back-end.

Nous avons utilisé le framework Angular pour le développement front-end et Node.js pour le back-end. Leur combinaison nous a permis de créer une application web facile à maintenir.

Pour réaliser ce projet, nous avons partagé nos compétences et nos idées pour concevoir et mettre en œuvre les différentes fonctionnalités de l'application. Ce travail d'équipe nous a permis de mieux comprendre l'importance de la communication et de la coordination dans un projet de développement web.

Cette introduction présente un aperçu général de notre projet. La documentation technique qui suit détaille les aspects techniques de l'application, notamment l'architecture, les fonctionnalités et les choix technologique.

I. Front-end : Angular

1. Liste des composants

AppComponent: Composant principal de l'application qui englobe les autres composants.

NavbarComponent: Composant pour la barre de navigation en haut de la page.

AccueilComponent: Composant pour la page d'accueil.

PlaylistListComponent: Composant pour afficher la liste des playlists.

CreerPlaylistComponent: Composant pour créer une nouvelle playlist.

PlaylistVoirComponent: Composant pour voir les détails d'une playlist.

ConnexionComponent: Composant qui permet la connexion d'un utilisateur

MonCompteComponent: Composant qui affiche les playlists de l'utilisateur connecté

2. Liste des services

1. PlaylistListService

PlaylistListService est un service qui gère les opérations liées aux playlists, telles que la récupération, la création, la modification et la suppression de playlists et de morceaux de musique.

Il utilise **HttpClient** pour envoyer des requêtes HTTP au serveur Node.js et interagir avec les routes définies pour les différentes opérations.

```
public getPlayListById(id : Number) : Observable<IPlaylist> {
    let queryParams = new HttpParams().set("idPlaylist", id.toString());
    return this.http.get<IPlaylist>(this.PLAYLIST_API_URL_HTTP + "getplaylistbyid/", {params:queryParams})
}
```

Par exemple, la méthode `getPlayListById` envoie une requête GET à l'URL `/getplaylistbyid` avec l'ID de la playlist en tant que paramètre. Le serveur Node.js renvoie les détails de la playlist correspondante, qui sont ensuite mappés à l'interface `IPlaylist` et renvoyés comme un Observable.

2. UsersService

`UsersServies` est un service qui gère les opérations liées au utilisateurs, telles que la connexion et déconnexion.

Ce service contient une propriété publique `estConnectee` de type `BehaviorSubject<boolean>` avec la valeur initiale définie par l'appel à `this.utilisateurConnectee()`. `BehaviorSubject` est un type d'observable qui émet toujours la dernière valeur émise à tous les nouveaux abonnés. Cet attribut nous permet de savoir si l'utilisateur est connecté.

```

    seConnecter(pseudo: string, password: string): Observable<boolean> {
      return this.http.post<any>(this.PLAYLIST_API_URL_HTTP + "connexion", { pseudo, password }).pipe(
        tap(response => {
          if (response.success) {
            localStorage.setItem('utilisateurCourrant', JSON.stringify({ id: response.user.id, pseudo,
              password }));
            this.estConnectee.next(true);
          } else {
            this.estConnectee.next(false);
          }
        }),
        map(response => response.success)
      );
    }
  }

```

L'opérateur **tap** permet de réaliser une action sans modifier la réponse de la requête, et ici il vérifie si la réponse a la propriété **success** définie sur **true**. Si c'est le cas, il stocke les informations de l'utilisateur dans le stockage local (localStorage) et met à jour la valeur de l'observable **estConnectee** à **true**. Sinon, il met à jour la valeur de **estConnectee** à **false**.

3. Les Routes

1. **{path: '', component: AccueilComponent, canActivate: [ConnexionGuard]}**: Cette route définit le chemin d'accès pour la page d'accueil de l'application. Elle est associée au composant **AccueilComponent**. Elle utilise le canActivate pour vérifier si l'utilisateur est connecté avant de lui permettre d'accéder à cette page.
2. **{path: 'creer', component: CreerPlaylistComponent, canActivate: [ConnexionGuard]}**: Cette route définit le chemin d'accès pour la page de création de playlist. Elle est associée au composant **CreerPlaylistComponent**. Elle utilise également le canActivate pour vérifier si l'utilisateur est connecté avant de lui permettre d'accéder à cette page.
3. **{path: 'voir/:id', component: PlaylistVoirComponent , canActivate: [ConnexionGuard]}**: Cette route définit le chemin d'accès pour la page de détails d'une playlist. Elle est associée au composant **PlaylistVoirComponent**. Elle utilise également le canActivate pour vérifier si l'utilisateur est connecté avant de lui permettre d'accéder à cette page. Cette route contient un paramètre id dans l'URL qui est utilisé pour afficher les détails de la playlist correspondante.
4. **{path: 'monCompte', component: MonCompteComponent, canActivate: [ConnexionGuard]}**: Cette route définit le chemin d'accès pour la page de compte de l'utilisateur. Elle est associée au composant **MonCompteComponent**. Elle utilise également le canActivate pour vérifier si l'utilisateur est connecté avant de lui permettre d'accéder à cette page. Cette route permet à l'utilisateur de voir les playlists qu'il a publiées.
5. **{path: 'connexion', component: ConnexionComponent}**: Cette route définit le chemin d'accès pour la page de connexion de l'utilisateur. Elle est associée au composant **ConnexionComponent**. Cette route permet à l'utilisateur de se connecter à l'application en entrant son pseudo et son mot de passe. Il n'y a pas de restriction pour accéder à cette page.

4. Schémas pour chaque page de l'application

1. **Connexion:** La page de connexion affiche le formulaire de connexion

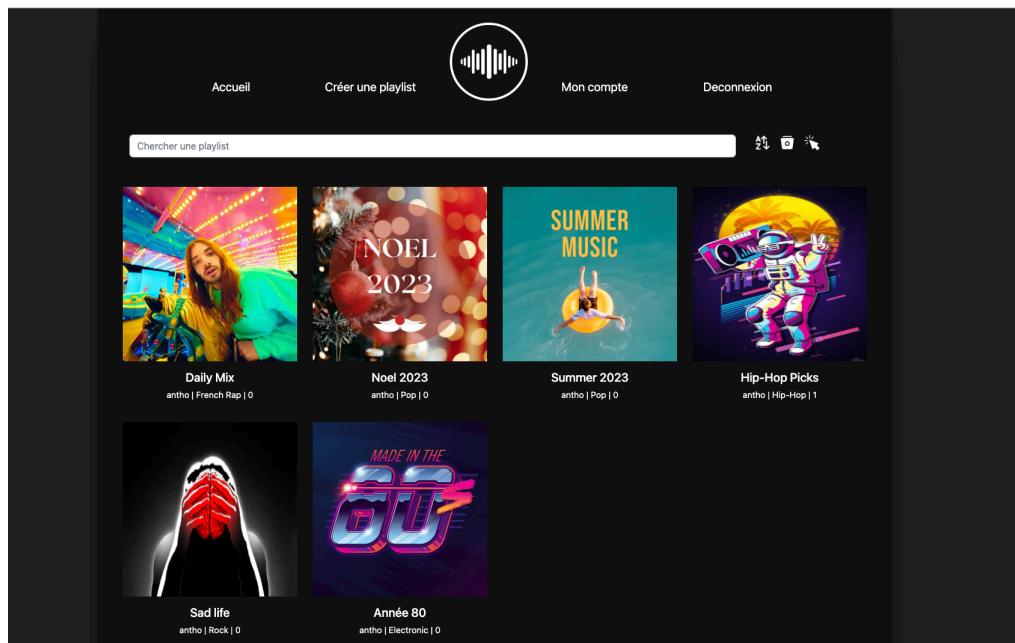
Utilisateur exemple : test / test

Pseudo

Password

Se connecter

2. **Accueil:** La page d'accueil affiche la liste des playlists disponibles et permet à l'utilisateur de naviguer vers la page de création d'une nouvelle playlist.



3. **Créer une playlist:** Cette page contient un formulaire pour créer une nouvelle playlist. L'utilisateur peut entrer le nom de la playlist, le nom du créateur, le style de musique et télécharger une photo de couverture.

Accueil Crée une playlist Mon compte Déconnexion

Crée une playlist

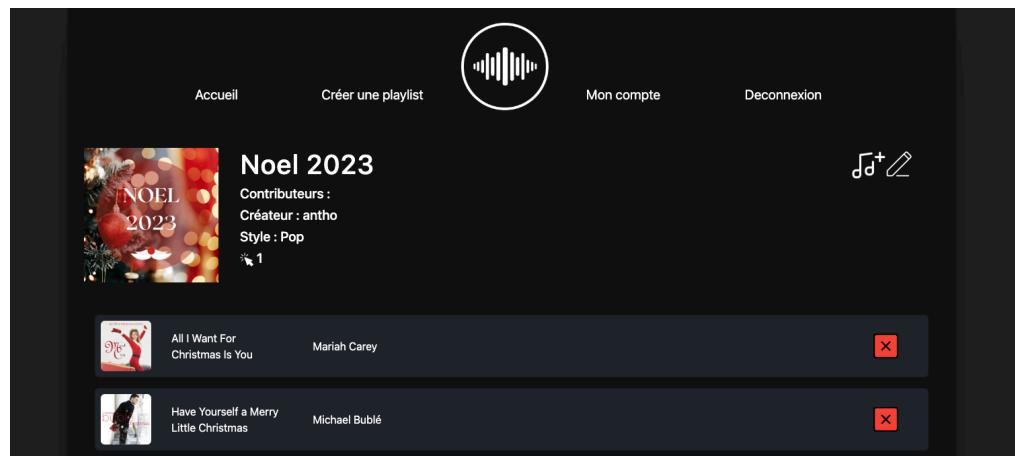
Nom

Url de l'image

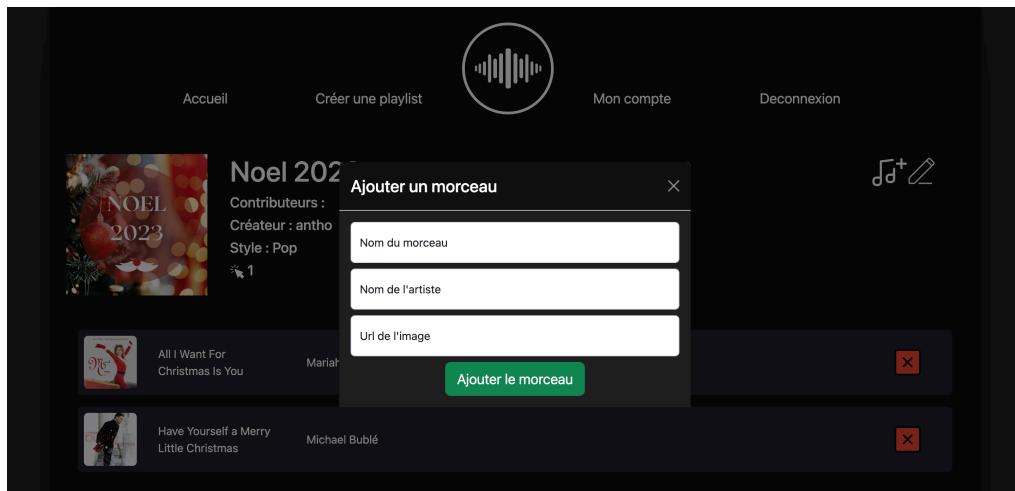
Style

Crée la playlist

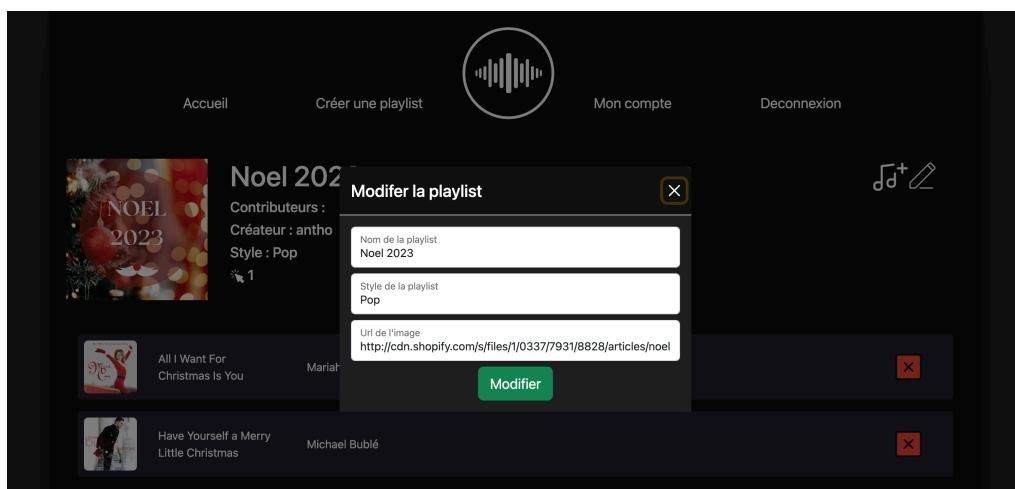
4. **Voir une playlist:** Cette page affiche les détails d'une playlist, y compris la liste des morceaux de musique. L'utilisateur peut ajouter de nouveaux morceaux, modifier les détails de la playlist ou supprimer des morceaux.



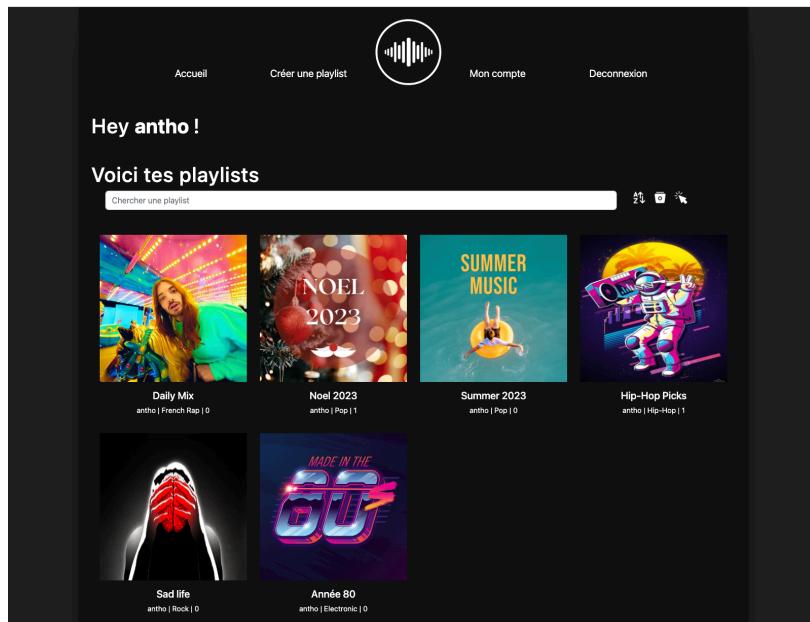
Si nous cliquons sur l'icône : nous pouvons ajouter un morceau à notre playlist.



Si nous cliquons sur l'icône : nous pouvons modifier les informations de notre playlist.



5. **Mon compte:** Cette page affiche les playlists de l'utilisateur connecté et propose également le tri, en fonction du nom, du style de musique, du nombre de clic, ainsi qu'un tri par ordre alphabétique.



5. Utilisation de FormBuilder

FormBuilder est utilisé dans le composant **CreerPlaylistComponent** pour faciliter la création et la validation du formulaire de création de playlist. Il permet de créer des groupes de contrôles et d'appliquer des validateurs pour assurer la validité des données soumises.

```
this.playListCreationForm = this.fb.group({
  nomPlayList: ['', [Validators.required, Validators.minLength(1), Validators.maxLength(15)]],
  urlCouverture: ['', Validators.required],
  nomCreateur: ['', [Validators.required, Validators.minLength(1), Validators.maxLength(15)]],
  nomStyle: ['', [Validators.required, Validators.minLength(1), Validators.maxLength(15)]],
});
```

6. Utilisation de Guards

Le garde **ConnexionGuard** est utilisé dans plusieurs routes de l'application pour vérifier si l'utilisateur est connecté avant de lui permettre d'accéder à une page spécifique. Il permet de protéger les pages sensibles de l'application et de rediriger les utilisateurs non autorisés vers la page de connexion.

```
canActivate(): Observable<boolean | UrlTree> {
  return this.userService.estConnectee.asObservable().pipe(
    map((estConnectee: boolean) => {
      return estConnectee ? true : this.router.parseUrl('/connexion');
    })
  );
}
```

II. Back-end : Node.js

1. Le principe d'une API

Une API (Application Programming Interface) est un ensemble de règles et de spécifications qui permettent aux applications de communiquer entre elles. Dans notre projet, nous avons créé une API REST à l'aide de Node.js et du framework Express. L'API expose des points de terminaison (ou routes) pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les données stockées côté serveur.

2. Stockage des données dans une ArrayList

Dans notre application, nous stockons les données des playlists dans une ArrayList (un tableau JavaScript) nommée playlists. Chaque élément de ce tableau représente un objet Playlist. Lorsqu'une nouvelle playlist est créée, elle est ajoutée à la fin de cette liste. Les opérations CRUD sont effectuées sur cette liste pour gérer les playlists et les morceaux qu'elles contiennent.

3. Les routes

Voici la liste des routes définies dans notre application et leur fonction :

1. POST /creerplaylist/ : Crée une nouvelle playlist et l'ajoute à l'ArrayList.
2. POST /ajoutermorceau/ : Ajoute un morceau à une playlist existante.
3. GET /ajouterclic/ : Incrémente le compteur de clics d'une playlist.
4. GET /getallplaylists : Récupère la liste de toutes les playlists.
5. GET /getplaylistbyid : Récupère une playlist spécifique en fonction de son ID.
6. DELETE /deleteplaylistmorceau : Supprime un morceau d'une playlist en fonction de l'ID de la playlist et de l'ID du morceau.
7. PUT /modifierplaylist/:id : Modifie les informations d'une playlist en fonction de son ID.
8. GET /getmorceauxbyidplaylist : Récupère la liste des morceaux d'une playlist en fonction de l'ID de la playlist.
9. GET /createtestvalues : Crée des données de test (playlists et morceaux) pour faciliter le développement et les tests.
10. POST /getplaylistsbyuserid : Permet d'obtenir toutes les playlists d'un utilisateur en fonction de son id
11. POST /connexion : Permet de tester le pseudo et le mot de passe pour autoriser ou non la connexion de l'utilisateur