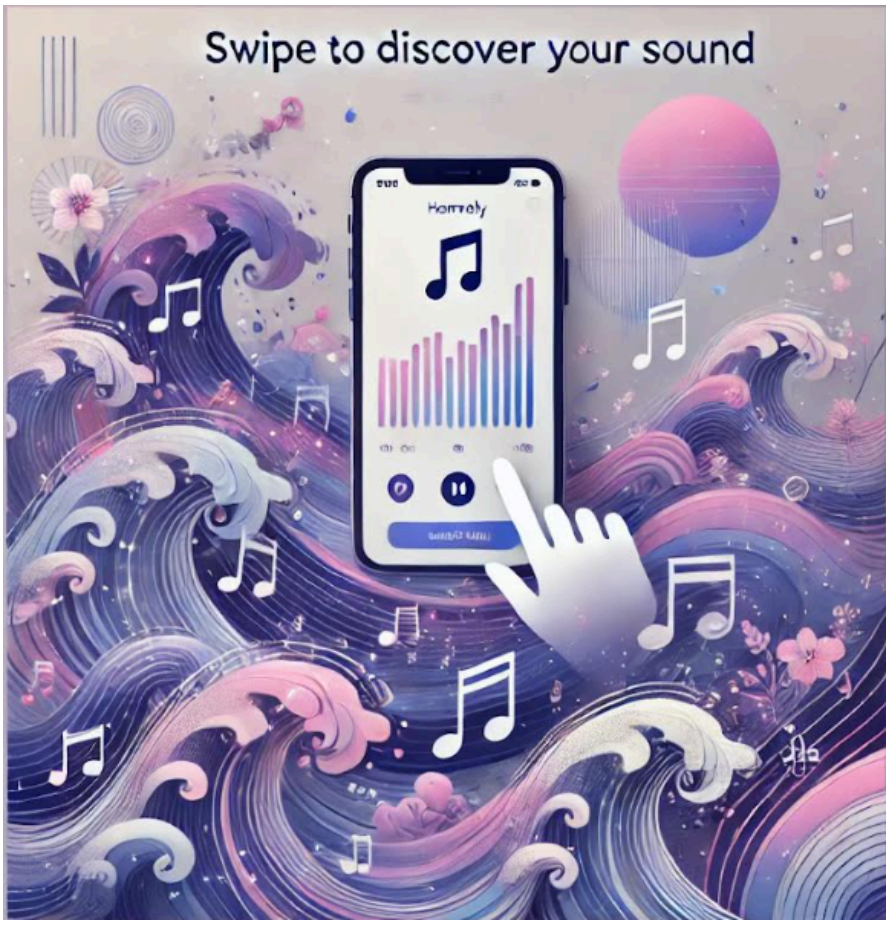
HONORINE Kylian
GRONDIN Angelique

SAE

Application Harmely



Sommaire

Introduction

Dans le cadre de notre SAE, nous avons développé une application innovante pour la découverte musicale, en utilisant l'API Spotify. L'objectif de cette application était de permettre aux utilisateurs d'explorer de nouvelles musiques tout en travaillant nos compétences en développement mobile. Ce projet nous a également permis de renforcer nos connaissances en JavaScript, React Native, Firebase, ainsi que dans l'intégration d'API tierces. En parallèle, nous avons travaillé sur l'expérience utilisateur, la gestion des données en temps réel et l'interaction entre utilisateurs grâce à un système de chat. Cette SAE nous a donné l'opportunité de mettre en œuvre des technologies modernes et de mieux comprendre les défis liés au développement d'applications.

Nous avons aussi conçu pour cette application un potentiel logo pour cette application pour laquelle on s'est investi.

Le thème du Japon a été choisi arbitrairement pour son esthétique. Les notes de musique dans le ciel rappellent le thème central de notre application, tout en évoquant des ailes, symbole d'envoie de partage et de communication. La palette de mauve et bleu crée une ambiance relaxante, offrant une expérience apaisante et immersive à l'utilisateur. Les fleurs ajoutent une touche de sérénité, tandis que l'intégration de **La Grande Vague de Kanagawa** (qui est aussi présent sur notre affiche de "pub" au début de notre compte rendu) invite à un voyage musical empreint de calme et de diversité.

Étape Préliminaire : Installation et Configuration de l'Environnement

Avant de commencer le développement, nous avons configuré un environnement de travail basé sur Node.js et gérer les dépendances nécessaires à l'application.

1. Installation de Node.js et npm :

- **Pourquoi** : Node.js est essentiel pour gérer les packages nécessaires au projet et exécuter les commandes liées à React Native.

- **Étapes** :
  - Télécharger et installer Node.js

Dépendances principales :

- `expo` : Framework principal pour créer notre application React Native.
- `react-native` : Permet de développer des applications mobiles en JavaScript.
- `npm` et `install` : Gestion des bibliothèques et de leurs versions.

Vérifier l'installation avec :

```
node -v
npm -v
```

2. Création des fichiers JSON (package.json) :

- **Pourquoi** : Le fichier `package.json` est la base pour gérer les dépendances et scripts nécessaires au projet.
- **Contenu** : Il contient les informations du projet (nom, version) et liste les dépendances comme React, Firebase, Expo, etc.

Commande pour initialiser le projet :

```
npm init
```

3. Installation de React Native et Expo :

- React Native est le framework de base pour construire l'application, tandis qu'Expo facilite le développement en fournissant des outils prêts à l'emploi.

Commandes :

```
npm install -g expo-cli
expo init nom_du_projet
```

Étape 1 : Authentification (AuthScreen.js)

Nous avons implémenté un système d'authentification pour permettre à chaque utilisateur d'accéder à son propre espace personnalisé. Cela garantit la sécurité des données et une meilleure gestion des préférences (titres aimés, non aimés, etc.). En utilisant Firebase Authentication, nous avons pu offrir des fonctionnalités de connexion et d'inscription en ligne. De plus, nous avons traduit les messages d'erreur pour rendre l'interface utilisateur plus compréhensible et adaptée à notre public cible.

Installation et configuration :

- Installation de Firebase et configuration de l'application via un fichier centralisé `firebaseConfig.js.

```
npm install firebase
```

- Intégration d'une librairie pour le chargement de polices (`expo-font`) et d'une Splash Screen (`expo-splash-screen`).

Dépendances utilisées pour la mise en place d'un utilisateur :

- `@expo/vector-icons` : Pour intégrer des icônes dans les boutons et menus.
- `expo-linear-gradient` : Pour les effets de dégradés sur les arrière-plans et les éléments visuels.
- `expo-font` : Permet de charger et utiliser des polices personnalisées.
- `expo-splash-screen` : Gère l'écran de chargement avant l'affichage de l'application.

Dépendances utilisées pour la gestion d'authentification sur la firebase :

- `firebase` : Fournit l'authentification et la base de données en temps réel.
- `@react-native-async-storage/async-storage` : Gère la persistance des données de connexion localement.

Ce qui a été installé et configuré :

Installation de Firebase :

```
npm install firebase
```

- Création d'un projet Firebase, récupération des clés de configuration et initialisation via le fichier `firebaseConfig.js`.
- Ajout de **expo-font** et **expo-splash-screen** pour améliorer le rendu visuel lors du chargement des polices personnalisées.

```
export default function AuthScreen({ navigation }) {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [message, setMessage] = useState('');
  const [isSignUpMode, setIsSignUpMode] = useState(false); // Mode par défaut : connexion
  const [fontsLoaded, setFontsLoaded] = useState(false);

  const loadFonts = async () => {
    await Font.loadAsync({
      'CustomFont': require('../assets/fonts/Megrim-Regular.ttf'), // Chemin vers la police téléchargée
    });
    setFontsLoaded(true);
  };

  useEffect(() => {
    const prepare = async () => {
      try {
        await SplashScreen.preventAutoHideAsync();
        await loadFonts();
      } catch (e) {
        console.warn(e);
      } finally {
        setFontsLoaded(true);
        await SplashScreen.hideAsync();
      }
    };

    prepare();
  }, []);

  // Fonction pour traduire les messages d'erreur en français
  const traduireMessageErreur = (errorCode) => {
    switch (errorCode) {
      case 'auth/invalid-email':
        return 'Adresse e-mail invalide.';
      case 'auth/user-not-found':
        return 'Utilisateur non trouvé.';
      case 'auth/wrong-password':
```

Nous avons conçu cet écran pour permettre aux utilisateurs de se connecter ou de s'inscrire facilement. La fonction `loadFonts` charge une police personnalisée pour améliorer l'esthétique, tandis que `prepare` gère le splash screen pour offrir une transition fluide. Enfin, une fonction traduit les messages d'erreur de Firebase en français pour une meilleure expérience utilisateur. Ce travail optimise à la fois l'accessibilité et la convivialité de l'application.

```
  const handleLogin = async () => {
    try {
      await signInWithEmailAndPassword(auth, email, password);
      setMessage('Connexion réussie !');
      navigation.navigate('MusicDiscovery'); // Redirige vers la page principale après connexion
    } catch (error) {
      setMessage(traduireMessageErreur(error.code));
    }
  };

  const handleSignUp = async () => {
    try {
      await createUserWithEmailAndPassword(auth, email, password);
      setMessage('Compte créé et connecté !');
      const user = userCredential.user; // Récupération de l'utilisateur
      const uid = user.uid; // Récupération de l'UID automatiquement généré par Firebase
      navigation.navigate('MusicDiscovery'); // Redirige vers la page principale après création du compte
    } catch (error) {
      setMessage(traduireMessageErreur(error.code));
    }
  };

  // Basculer entre le mode inscription et connexion
  const toggleMode = () => {
    setIsSignUpMode(!isSignUpMode);
    setMessage(''); // Réinitialiser le message
  };

  if (!fontsLoaded) {
    return null; // Affiche un écran vide jusqu'à ce que la police soit chargée
  }
```

Nous avons créé `handleLogin` pour connecter les utilisateurs et `handleSignUp` pour leur inscription via Firebase. Ces fonctions redirigent vers l'écran principal après validation et affichent des messages d'erreur clairs en cas de problème.

```
<View style={styles.buttonContainer}>
  {isSignUpMode ? (
    <>
      <TouchableOpacity onPress={handleSignUp}>
        <Text style={styles.buttonText}>Créer un compte</Text>
      </TouchableOpacity>
      <TouchableOpacity onPress={toggleMode} style={{ marginTop: 10 }}>
        <Text style={styles.switchText}>Vous avez déjà un compte ? Se connecter</Text>
      </TouchableOpacity>
    </>
  ) : (
    <>
      <TouchableOpacity onPress={handleLogin}>
        <Text style={styles.buttonText}>Se connecter</Text>
      </TouchableOpacity>
      <TouchableOpacity onPress={toggleMode} style={{ marginTop: 10 }}>
        <Text style={styles.switchText}>Pas de compte ? Créer un compte</Text>
      </TouchableOpacity>
    </>
  )}
</View>
```

L'interface permet de basculer entre les modes connexion et inscription de manière fluide. Les boutons intuitifs simplifient la navigation tout en réinitialisant les messages d'erreur.

Étape 2 : Navigation Principale (Navigation.js)

Nous avons structuré notre application à l'aide de `react-navigation` pour que les utilisateurs puissent naviguer facilement entre les différents écrans. Cette approche nous a permis d'organiser les fonctionnalités principales de manière logique et intuitive. Nous avons utilisé une navigation stack pour empiler les écrans et permettre une navigation fluide.

Installation et configuration :

- Mise en place de `react-navigation` et des dépendances nécessaires.
- Configuration des écrans avec des routes claires (Auth Screen, Swipe Screen, ChatScreen, etc.).

Dépendances utilisées :

- `@react-navigation/native` et `@react-navigation/stack` : Gèrent la navigation.
- `react-native-screens` : Améliore la performance des transitions entre écrans.

Ce qui a été installé et configuré :

- Installation de `react-navigation` et ses dépendances :

```
npm install @react-navigation/native react-native-screens
react-native-safe-area-context react-native-gesture-handler
react-native-reanimated react-native-svg

npm install @react-navigation/stack
```

```
Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > JS Navigation.js > ...
1  import React from 'react';
2  import { NavigationContainer } from '@react-navigation/native';
3  import { createStackNavigator } from '@react-navigation/stack';
4  import AuthScreen from './AuthScreen'; // Écran d'authentification
5  import SwipeScreen from './SwipeScreen'; // Écran de swipe (remplace MusicDiscoveryScreen par SwipeScreen)
6  import ChatScreen from './ChatScreen'; // Écran de chat
7  import SearchScreen from './SearchScreen';
8
9  const Stack = createStackNavigator();
10
11 export default function AppNavigation() {
12   return (
13     <NavigationContainer>
14       <Stack.Navigator initialRouteName="Auth">
15         <Stack.Screen name="Auth" component={AuthScreen} options={{ headerShown: false }} />
16         <Stack.Screen name="Swipe" component={SwipeScreen} options={{ headerShown: false }} /> {/* Écran de swipe */}
17         <Stack.Screen name="Chat" component={ChatScreen} options={{ title: 'Discussion' }} /> {/* Écran de chat */}
18         <Stack.Screen name="Search" component={SearchScreen} options={{ title: "Recherche" }} /> {/* Écran de recherche */}
19       </Stack.Navigator>
20     </NavigationContainer>
21   );
22 }
23
```

Voici le menu principal et les différents boutons qui sont accessibles.

Étape 3 : Système de Swipe (SwipeScreen.js)

L'une des fonctionnalités principales de notre application est le swipe pour aimer ou non des morceaux, inspiré du modèle de Tinder. Nous avons choisi cette approche ludique pour engager les utilisateurs dans la découverte musicale. Nous avons utilisé `PanResponder` pour gérer les mouvements de swipe, et intégré `expo-av` pour permettre la lecture des extraits musicaux.

```
const panResponder = useRef(
  PanResponder.create({
    onMoveShouldSetPanResponder: (evt, gestureState) => Math.abs(gestureState.dx) > 20,
    onPanResponderMove: (event, gestureState) => {
      pan.setValue({ x: gestureState.dx, y: gestureState.dy });
    },
    onPanResponderRelease: (event, gestureState) => {
      const direction = gestureState.dx > 0 ? 'right' : 'left';
      const threshold = 120;

      if (Math.abs(gestureState.dx) > threshold) {
        handleSwipe(direction);
      } else {
        Animated.spring(pan, {
          toValue: { x: 0, y: 0 },
          useNativeDriver: true,
        }).start();
      }
    },
  })
).current;
```

Ce code implémente un système de gestion des gestes via PanResponder, permettant de détecter et d'interpréter les mouvements de glissement (swipes) de l'utilisateur. Il distingue les directions ("droite" ou "gauche") selon la distance du geste et déclenche une animation ou une action correspondante.

Installation et configuration :

- Configuration de l'API Spotify avec les clés de développeur (`clientID` et `clientSecret`).
- Installation de `expo-av` pour la lecture des extraits musicaux.

Dépendances utilisées :

- `react-native-gesture-handler` : Pour gérer les swipes.
- `react-native-deck-swiper` : Simplifie la création d'un composant swipe.
- `react-native-linear-gradient` : Utilisé pour l'arrière-plan des cartes musicales.

Ce qui a été installé et configuré :

Installation de `expo-av` pour la gestion de l'audio :
npm `install expo-av`

- Intégration de l'API Spotify dans `SpotifyService.js` (voir Étape 7).

```
10  const SwipeScreen = ({ navigation }) => {
11    const [hostIpAddress, setHostIpAddress] = useState('192.168.1.17'); // Adresse IP statique par défaut
12    const [tracks, setTracks] = useState([]);
13    const [currentTrack, setCurrentTrack] = useState(null);
14    const [sound, setSound] = useState(null);
15    const [isPlaying, setIsPlaying] = useState(false);
16    const [genreLock, setGenreLock] = useState(false);
17    const [likedTracks, setLikedTracks] = useState([]);
18    const [dislikedTracks, setDislikedTracks] = useState([]);
19    const [menuVisible, setMenuVisible] = useState(false);
20    const pan = useRef(new Animated.ValueXY()).current;
21
22    useEffect(() => {
23      // Appel au serveur avec une IP statique
24      const fetchHostIpAddress = async () => {
25        try {
26          const response = await fetch('http://192.168.1.17:3000/get-ip'); // IP statique
27          const data = await response.json();
28          setHostIpAddress(data.ipAddress);
29          console.log('Adresse IP de l\'hôte détectée :', data.ipAddress);
30        } catch (error) {
31          console.error('Erreur lors de la récupération de l\'adresse IP :', error);
32        }
33      };
```

Nous avons implémenté un système pour récupérer l'adresse IP de l'hôte via une requête HTTP. Cela nous permet d'adapter dynamiquement l'application à l'environnement réseau tout en définissant une adresse IP statique par défaut.

```
SwipeScreen.js
Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > SwipeScreen.js > SwipeScreen
10  const SwipeScreen = ({ navigation }) => {
43    }, []);
44
45    const fetchTracks = async () => {
46      await getAccessToken();
47      const newTracks = await getRandomTracksByGenre('hip-hop', 'french'); // Genre par défaut
48      setTracks(newTracks);
49      if (newTracks.length > 0) {
50        setCurrentTrack(newTracks[0]);
51      }
52    };
```

Nous avons créé une fonction pour récupérer des morceaux aléatoires via l'API Spotify. Les morceaux sont filtrés par genre (par défaut, "hip-hop" et "french"), et le premier morceau est défini comme actuellement en lecture.

```
Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > SwipeScreen.js > SwipeScreen
10  const SwipeScreen = ({ navigation }) => {
71    const handleSwipe = async (direction) => {
85      if (direction === 'right') {
86        setLikedTracks((prev) => [...prev, trackDetails]);
87      } else if (direction === 'left') {
88        setDislikedTracks((prev) => [...prev, trackDetails]);
89      }
90
91      const toValue = direction === 'right' ? { x: width, y: 0 } : { x: -width, y: 0 };
92
```

Nous avons développé une fonction pour gérer les interactions de swipe. Si un utilisateur swipe à droite, le morceau est ajouté à la liste des titres aimés, et s'il swipe à gauche, il est ajouté à la liste des titres non aimés. Cela nous permet de capturer les préférences musicales des utilisateurs.

```
return (
  <ImageBackground source={require('../assets/Choses_Violettes.jpg')} style={styles.container}>
    <TouchableOpacity onPress={toggleMenu} style={styles.menuIcon}>
      <MaterialCommunityIcons name="menu" size={40} color="#fff" />
    </TouchableOpacity>

    {/* Icône de loupe redirige vers SearchScreen */}
    <TouchableOpacity onPress={() => navigation.navigate('Search')} style={styles.searchIcon}>
      <MaterialCommunityIcons name="magnify" size={30} color="#fff" />
    </TouchableOpacity>

    {menuVisible && (
      <View style={styles.menuContainer}>
        <TouchableOpacity onPress={() => navigation.navigate('LikedTracks', { likedTracks })} style={styles.menuOption}>
          <MaterialCommunityIcons name="heart" size={30} color="red" />
          <Text style={styles.menuText}>Titres aimés</Text>
        </TouchableOpacity>
        <TouchableOpacity onPress={() => navigation.navigate('DislikedTracks', { dislikedTracks })} style={styles.menuOption}>
          <MaterialCommunityIcons name="heart-broken" size={30} color="gray" />
          <Text style={styles.menuText}>Titres dislikés</Text>
        </TouchableOpacity>
      </View>
    )}

    <View style={styles.swipeArea}>{renderCard()}</View>

    <View style={styles.bottomMenu}>
      <TouchableOpacity style={styles.iconContainer}>
        <MaterialCommunityIcons name="account" size={30} color="#fff" />
      </TouchableOpacity>
      <TouchableOpacity style={styles.iconContainer}>
        <MaterialCommunityIcons name="share-variant" size={30} color="#fff" />
      </TouchableOpacity>
```

Nous avons intégré des options de navigation pour accéder aux différents écrans comme la recherche, les titres aimés, et les titres non aimés. Un menu interactif permet également de gérer les actions principales, comme partager ou explorer d'autres morceaux, tout en offrant une interface claire et accessible.

Étape 4 : Chat en Temps Réel (ChatScreen.js)

Nous avons développé un système de chat en temps réel pour permettre aux utilisateurs de communiquer et de partager leurs découvertes musicales. Cette fonctionnalité repose sur Firebase Firestore, qui nous a permis de stocker et de synchroniser les messages en temps réel.

Installation et configuration :

- Configuration de Firestore pour stocker les messages.
- Mise en place de la synchronisation en temps réel via `onSnapshot`.

Dépendances utilisées :

- `pusher-js` et `socket.io-client` : Gèrent la communication en temps réel.

```jsx
const ChatScreen = ({ route }) => {
  // Récupérer le paramètre "chatId" envoyé depuis SwipeScreen
  const { chatId } = route.params;
  const [messages, setMessages] = useState([]);
  const [newMessage, setNewMessage] = useState('');

  useEffect(() => {
    // Charger les messages pour ce chat en utilisant le chatId
    const q = query(collection(firestore, 'chats', chatId, 'messages'), orderBy('createdAt', 'asc'));
    const unsubscribe = onSnapshot(q, (snapshot) => {
      const messagesList = snapshot.docs.map((doc) => ({
        id: doc.id,
        ...doc.data(),
      }));
      setMessages(messagesList);
    });

    return () => unsubscribe();
  }, [chatId]);

  const handleSend = async () => {
    if (newMessage.trim()) {
      await addDoc(collection(firestore, 'chats', chatId, 'messages'), {
        text: newMessage,
        createdAt: new Date(),
        userId: auth.currentUser.uid,  // Exemples d'informations liées à l'utilisateur
      });

      setNewMessage('');  // Réinitialiser le champ de saisie
    }
  };

  const renderItem = ({ item }) => {
    const isOwnMessage = item.userId === auth.currentUser.uid;
    return (
      <View style={[styles.messageContainer, isOwnMessage ? styles.ownMessage : styles.otherMessage]}>
        <Text style={styles.messageText}>{item.text}</Text>
      </View>
    );
  };
```

Cet écran synchronise les messages en temps réel via Firestore grâce au `chatId`. La fonction `handleSend` permet d'envoyer des messages avec des informations sur l'utilisateur et l'heure, tandis que `renderItem` distingue visuellement les messages envoyés et reçus.

Étape 5 : Recherche de Musiques (SearchScreen.js)

Nous avons ajouté une fonctionnalité de recherche pour que les utilisateurs puissent explorer des morceaux spécifiques via l'API Spotify. La recherche est triée par pertinence et popularité, et les résultats incluent des informations détaillées (titre, artiste, album).

Installation et configuration :

- Utilisation de l'API Spotify pour récupérer les résultats.
- Mise en place d'un affichage clair et réactif pour les résultats de recherche.

Dépendances utilisées :

- `expo-network` : Vérifie l'état de la connexion internet avant les requêtes.
- `buffer` : Gère les données encodées lors des requêtes à l'API.

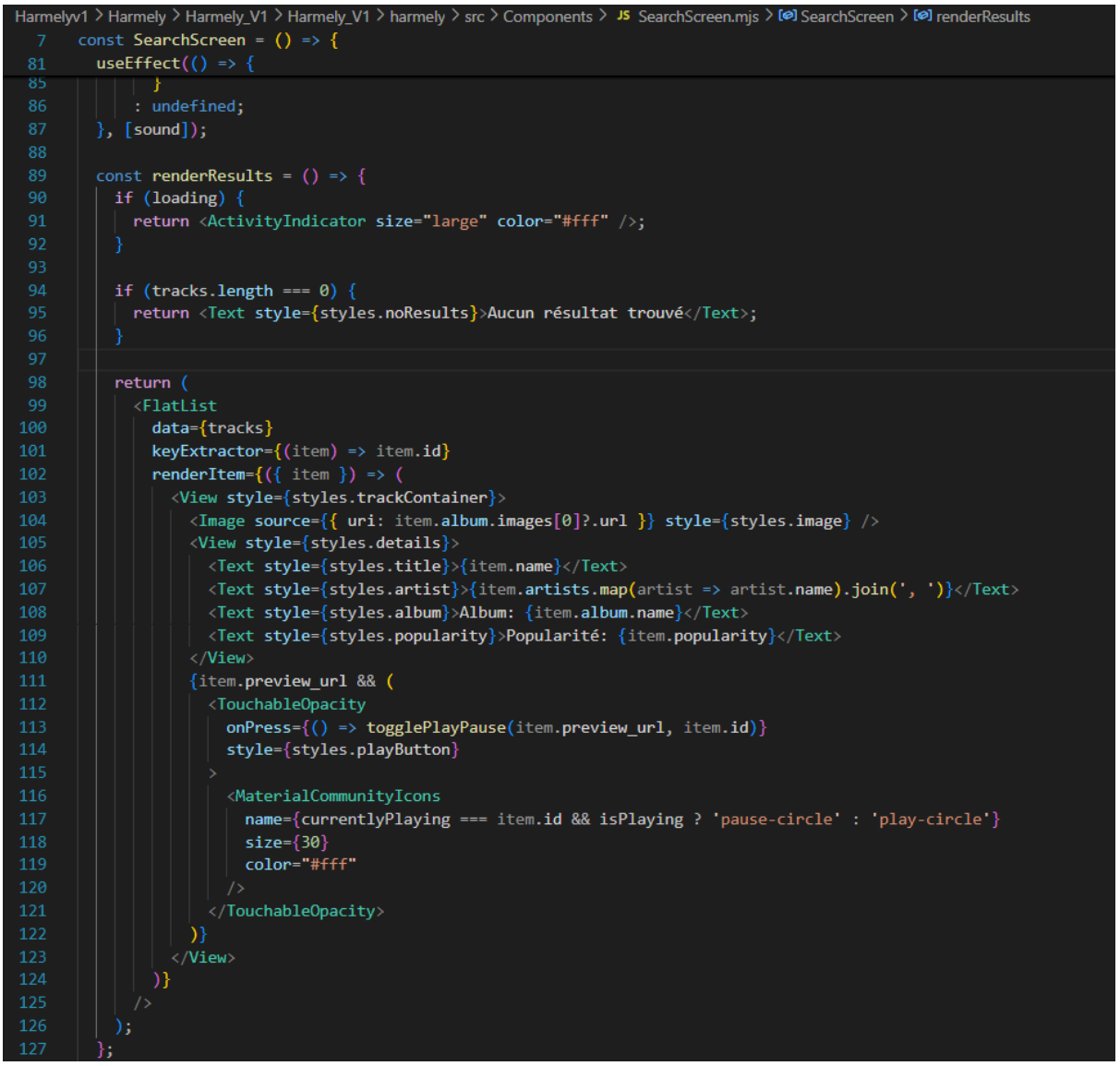
```
// Fonction pour normaliser une chaîne (supprimer accents et caractères spéciaux)
const normalizeString = (str) =>
  str
    .normalize('NFD')
    .replace(/[\u0300-\u036f]/g, '') // Supprimer les accents
    .toLowerCase();

const handleSearch = async () => {
  setLoading(true);
  try {
    await getAccessToken();
    const results = await searchTracks(query); // Utilisation de searchTracks
    const filteredResults = results.filter(track => track.preview_url); // Filtrer uniquement les musiques avec un extrait

    // Trier les résultats par similarité avec la requête
    const normalizedQuery = normalizeString(query);
    const sortedResults = filteredResults
      .map((track) => {
        const normalizedTrackName = normalizeString(track.name);
        const normalizedArtistNames = normalizeString(
          track.artists.map((artist) => artist.name).join(' ')
        );

        // Score basé sur la présence dans le titre ou l'artiste
        const relevance =
          normalizedTrackName.includes(normalizedQuery) +
          normalizedArtistNames.includes(normalizedQuery);
        return { ...track, relevance };
      })
      .sort((a, b) => b.relevance - a.relevance || b.popularity - a.popularity) // Trier par pertinence puis par popularité
      .slice(0, 20); // Limiter à 20 résultats
```

Nous avons implémenté une recherche musicale avancée. La fonction `normalizeString` standardise les chaînes pour une correspondance précise, et `handleSearch` récupère les morceaux via l'API Spotify, filtre les résultats pour inclure uniquement les extraits disponibles et les trie par pertinence et popularité. Seuls les 20 résultats les plus pertinents sont affichés.

Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > Components > SearchScreen.mjs > SearchScreen > renderResults

```
 7   const SearchScreen = () => {
22     const handleSearch = async () => {
31         const sortedResults = filteredResults
45           .slice(0, 20); // Limiter à 20 résultats
46
47         setTracks(sortedResults);
48       } catch (error) {
49         console.error('Erreur lors de la recherche de musique', error);
50       } finally {
51         setLoading(false);
52       }
53     };
54
55     const togglePlayPause = async (previewUrl, trackId) => {
56       if (currentlyPlaying === trackId && sound) {
57         if (isPlaying) {
58           await sound.pauseAsync();
59           setIsPlaying(false);
60         } else {
61           await sound.playAsync();
62           setIsPlaying(true);
63         }
64       } else {
65         if (sound) {
66           await sound.unloadAsync();
67           setSound(null);
68           setIsPlaying(false);
69         }
70
71         if (previewUrl) {
72           const { sound: newSound } = await Audio.Sound.createAsync({ uri: previewUrl });
73           setSound(newSound);
74           setCurrentlyPlaying(trackId);
75           setIsPlaying(true);
76           await newSound.playAsync();
77         }
78       }
79     };
80
81     useEffect(() => {
82       return sound
83         ? () => {
84             sound.unloadAsync();
85           }
86         : undefined;
```

Nous avons ajouté une fonctionnalité pour gérer la lecture et la mise en pause des extraits musicaux. La fonction `togglePlayPause` contrôle la lecture selon l'état actuel : elle peut mettre en pause, reprendre ou changer de morceau. Elle utilise la bibliothèque **expo-av** pour charger, jouer ou arrêter l'audio en fonction de l'extrait sélectionné.



Nous avons conçu une fonction pour afficher les résultats de recherche sous forme de liste. Si aucun résultat n'est disponible, un message informatif est affiché. Chaque élément inclut des détails comme le titre, l'artiste et l'album, ainsi qu'un bouton de lecture pour écouter un extrait. L'état de chargement est également géré pour améliorer l'expérience utilisateur.

Étape 6 : Historique des Titres Aimés et Non Aimés (LikedTracksScreen.js & DislikedTracksScreen.js)

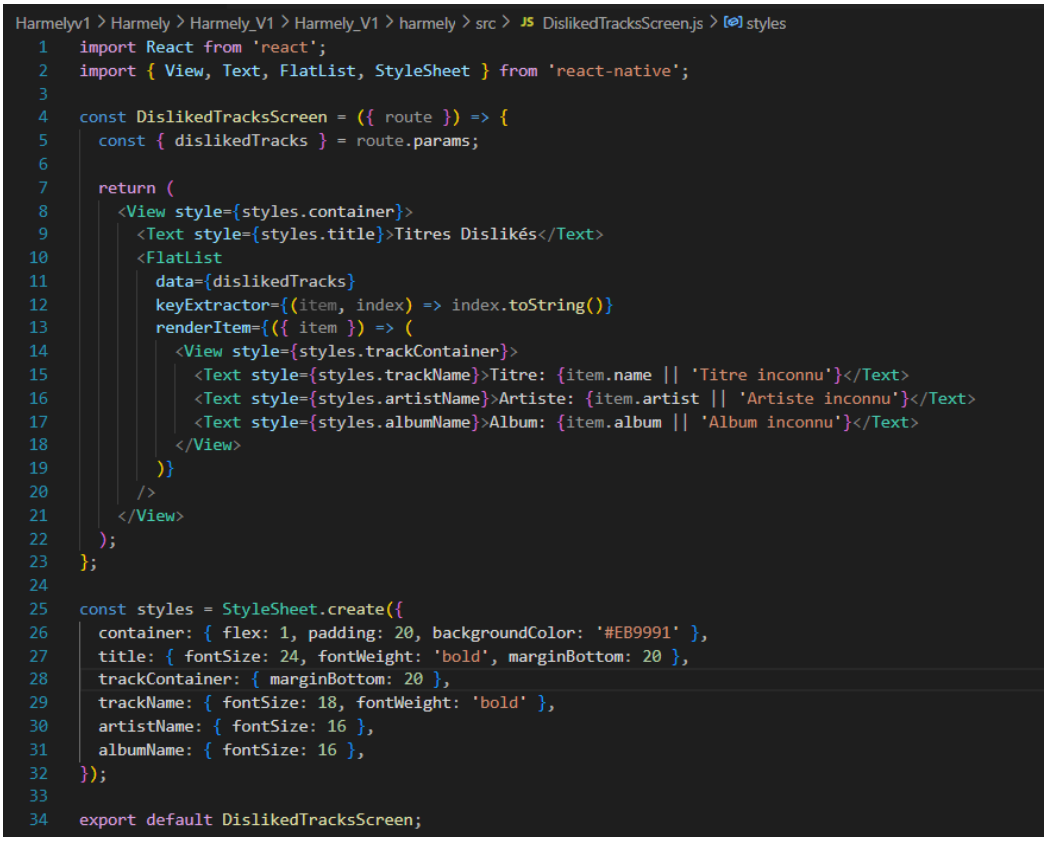
Nous avons conçu deux écrans pour afficher les morceaux aimés ou non aimés. Ces écrans permettent à l'utilisateur de retrouver ses préférences musicales.

Installation et configuration :

- Aucun package supplémentaire requis, les fonctionnalités sont développées en natif avec React Native.

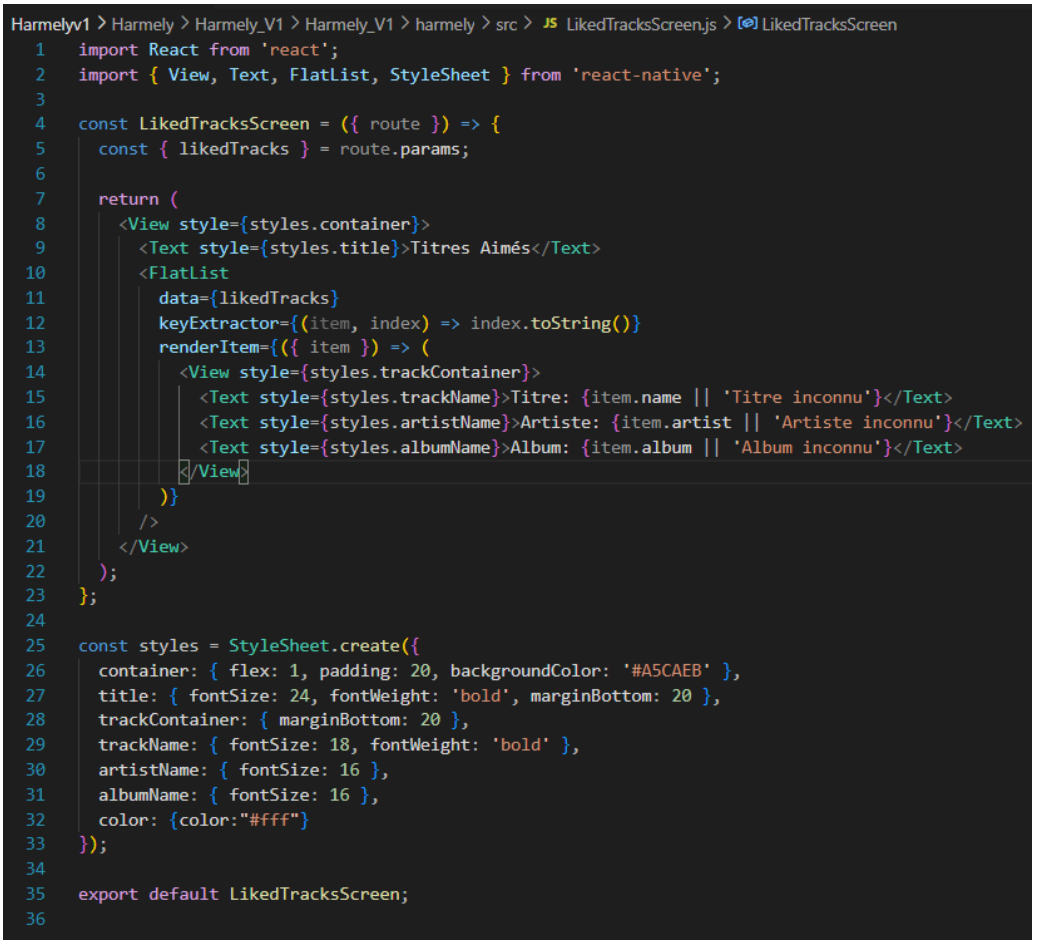
Dépendances utilisées :

- `react-native-flatlist` : Affiche les listes dynamiques des morceaux aimés et non aimés.



Nous avons développé cet écran pour afficher la liste des titres non aimés. Grâce à `FlatList`, chaque titre est affiché avec son nom, son artiste et son album, ou une mention

"inconnu" si l'information est absente. Cet affichage utilise un style personnalisé pour rendre les informations claires et lisibles.



Cet écran permet de visualiser les titres aimés par l'utilisateur. À l'aide de `FlatList`, nous affichons chaque morceau avec son titre, son artiste et son album, ou un texte "inconnu" si des données manquent. L'interface est stylisée pour garantir une lisibilité optimale et une expérience utilisateur agréable.

Étape 7 : Intégration avec l'API Spotify (SpotifyService.js)

Nous avons configuré un service centralisé pour interagir avec l'API Spotify, ce qui nous a permis de récupérer des morceaux, des recommandations et des recherches.

Installation et configuration :

- Configuration des tokens API dans Spotify Developer Console.
- Installation des bibliothèques nécessaires (`spotify-web-api-node`, `react-native-base64`).

Dépendances utilisées :

- `spotify-web-api-node` : Gère les appels à l'API Spotify.
- `axios` : Permet de faire des requêtes HTTP vers l'API Spotify.

Ce qui a été installé et configuré :

Installation de `spotify-web-api-node` :

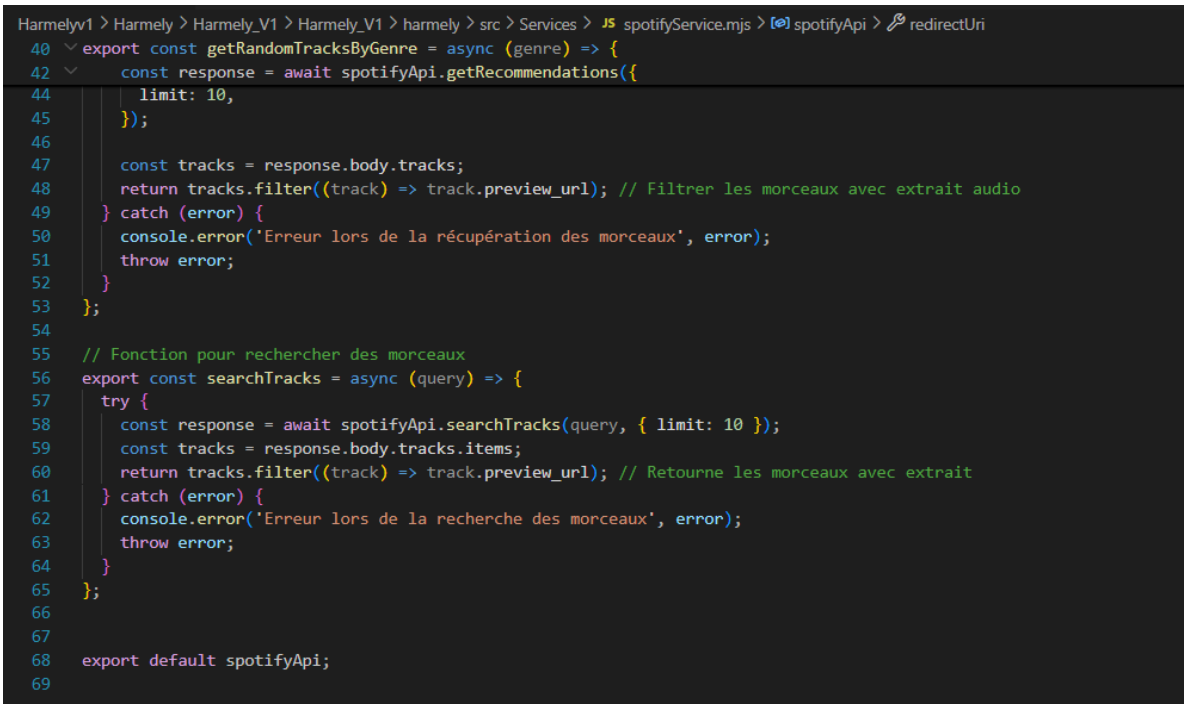
```
npm install spotify-web-api-node react-native-base64
```

- Configuration des clés API dans Spotify Developer Console.

Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > Services > spotifyService.mjs > spotifyApi > redirectUri

```
import SpotifyWebApi from 'spotify-web-api-node';
import base64 from 'react-native-base64';

// Configuration de l'API Spotify
const spotifyApi = new SpotifyWebApi({
  clientId: '9b0805e5f2ec44969cb8641a8d3d52da',
  clientSecret: '5e0196f3d89b4778bea373daef5d0962',
  redirectUri: 'http://localhost:3000/callback',
});

// Fonction pour obtenir un jeton d'accès
export const getAccessToken = async () => {
  try {
    const encodedCredentials = base64.encode('9b0805e5f2ec44969cb8641a8d3d52da:5e0196f3d89b4778bea373daef5d0962');
    const response = await fetch('https://accounts.spotify.com/api/token', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
        'Authorization': `Basic ${encodedCredentials}`,
      },
      body: new URLSearchParams({
        grant_type: 'client_credentials',
      }).toString(),
    });

    const data = await response.json();
    if (data.access_token) {
      spotifyApi.setAccessToken(data.access_token);
      return data.access_token;
    } else {
      throw new Error('Jeton d\'accès manquant dans la réponse de l\'API');
    }
  } catch (error) {
    console.error('Erreur lors de l\'obtention du jeton d\'accès', error);
    throw error;
  }
};

// Fonction pour obtenir des morceaux par genre
export const getRandomTracksByGenre = async (genre) => {
  try {
    const response = await spotifyApi.getRecommendations({
      seed_genres: [genre],
      limit: 10,
    });
```

Nous avons configuré l'API Spotify pour permettre l'accès aux données musicales. La fonction `getAccessToken` génère un jeton d'accès nécessaire pour authentifier nos requêtes. La fonction `getRandomTracksByGenre` utilise ce jeton pour récupérer des morceaux aléatoires selon un genre défini, offrant ainsi une expérience personnalisée à l'utilisateur.



```
Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > Services > spotifyService.mjs > spotifyApi > redirectUri
40  export const getRandomTracksByGenre = async (genre) => {
42      const response = await spotifyApi.getRecommendations({
44        limit: 10,
45      });
46
47      const tracks = response.body.tracks;
48      return tracks.filter((track) => track.preview_url); // Filtrer les morceaux avec extrait audio
49    } catch (error) {
50      console.error('Erreur lors de la récupération des morceaux', error);
51      throw error;
52    }
53  };
54
55  // Fonction pour rechercher des morceaux
56  export const searchTracks = async (query) => {
57    try {
58      const response = await spotifyApi.searchTracks(query, { limit: 10 });
59      const tracks = response.body.tracks.items;
60      return tracks.filter((track) => track.preview_url); // Retourne les morceaux avec extrait
61    } catch (error) {
62      console.error('Erreur lors de la recherche des morceaux', error);
63      throw error;
64    }
65  };
66
67
68  export default spotifyApi;
69
```

Nous avons implémenté deux fonctions principales : `getRandomTracksByGenre` pour récupérer des morceaux aléatoires par genre, et `searchTracks` pour rechercher des morceaux basés sur une requête utilisateur. Ces deux fonctions filtrent les résultats pour inclure uniquement les morceaux disposant d'extraits audio, garantissant une expérience cohérente et enrichissante pour l'utilisateur.

Étape 8 : Configuration Firebase (firebaseConfig.js)

Firebase est au cœur de notre application pour l'authentification et le stockage des données. Nous avons configuré Firebase pour fonctionner avec React Native et garantir une expérience fluide.

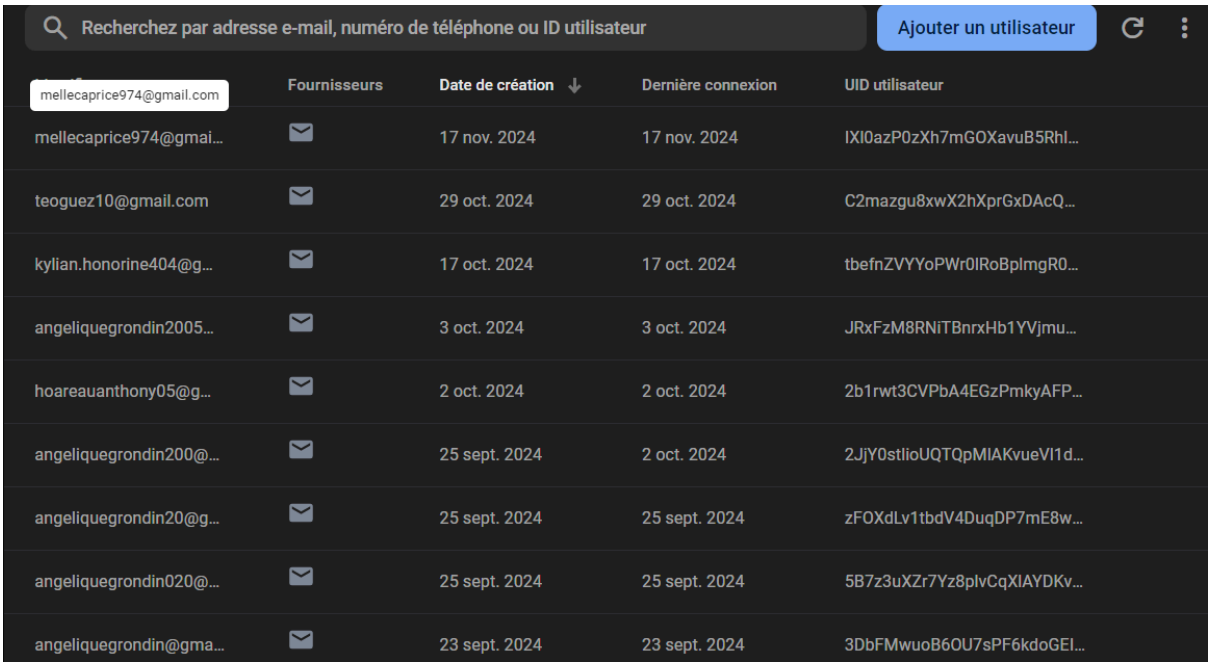
Installation et configuration :

- Création d'un projet Firebase et récupération des clés de configuration.
- Configuration de la persistance des données avec `AsyncStorage`.

Harmelyv1 > Harmely > Harmely_V1 > Harmely_V1 > harmely > src > JS firebaseConfig.js > ...

```js
// src/firebaseConfig.js
import { initializeApp } from "firebase/app";
import { getAuth, initializeAuth, getReactNativePersistence } from "firebase/auth";
import { getFirestore } from 'firebase/firestore';
import AsyncStorage from '@react-native-async-storage/async-storage'; // Importer AsyncStorage

const firebaseConfig = {
  apiKey: "AIzaSyBLrk6PcmBEVQHmUexjGSNOZZ37Sf95rtM",
  authDomain: "harmely-18ea3.firebaseapp.com",
  projectId: "harmely-18ea3",
  storageBucket: "harmely-18ea3.appspot.com",
  messagingSenderId: "424517468426",
  appId: "1:424517468426:web:9a52280525a62781287af3",
  measurementId: "G-9ML8DNSDFP"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
const firestore = getFirestore(app);

// Configurer l'authentification avec persistance
const auth = initializeAuth(app, {
  persistence: getReactNativePersistence(AsyncStorage)
});

// Fonction pour récupérer l'utilisateur actuel et son UID
const getCurrentUser = () => {
  return new Promise((resolve, reject) => {
    onAuthStateChanged(auth, (user) => {
      if (user) {
        // L'utilisateur est connecté
        resolve(user); // Retourne l'utilisateur, avec l'UID accessible via user.uid
      } else {
        // Aucun utilisateur n'est connecté
        resolve(null);
      }
    }, reject);
  });
};

export { auth, firestore, getCurrentUser };

```

Nous avons configuré Firebase pour gérer l'authentification et la base de données Firestore de l'application. La méthode `initializeAuth` est utilisée pour permettre une persistance des sessions utilisateur, assurant une expérience continue. La fonction `getCurrentUser` récupère les informations de l'utilisateur actuellement connecté, facilitant la personnalisation et la gestion des données utilisateur.



Voici notre firebase, nous pouvons observer les différents tests de connexion par différentes personnes. Firebase permet aussi de générer des UID pour chaque utilisateur qui se connecte.

Liste des dépendances

```
"dependencies": {
  "@expo/metro-runtime": "^4.0.0",
  "@expo/vector-icons": "^14.0.3",
  "@react-native-async-storage/async-storage": "^1.23.1",
  "@react-native-community/slider": "^4.5.5",
  "@react-navigation/native": "^6.1.18",
  "@react-navigation/native-stack": "^6.11.0",
  "@react-navigation/stack": "^6.4.1",
  "axios": "^1.7.7",
  "buffer": "^6.0.3",
  "dotenv": "^16.4.5",
  "expo": "^52.0.7",
  "expo-av": "^15.0.1",
  "expo-font": "^13.0.1",
  "expo-linear-gradient": "^14.0.1",
  "expo-network": "^7.0.0",
  "expo-notifications": "^0.29.8",
  "expo-splash-screen": "^0.29.11",
  "firebase": "^10.13.1",
  "install": "^0.13.0",
  "link": "^2.1.1",
  "npm": "^10.8.3",
  "os": "^0.1.2",
  "pusher": "^5.2.0",
  "pusher-js": "^8.4.0-rc2",
  "react-native": "^0.76.2",
  "react-native-base64": "^0.2.1",
  "react-native-deck-swiper": "^2.0.17",
  "react-native-gesture-handler": "^2.20.2",
  "react-native-linear-gradient": "^2.8.3",
  "react-native-network-info": "^5.2.1",
  "react-native-safe-area-context": "^4.12.0",
  "react-native-screens": "^4.1.0",
  "react-native-snap-carousel": "^1.3.1",
  "react-native-vector-icons": "^10.2.0",
  "react-native-web": "^0.19.13",
  "socket.io-client": "^4.8.1",
  "spotify-web-api-node": "^5.0.2"
},
"devDependencies": {
  "@react-native-community/cli": "^15.1.2"
}
}
```

Difficultés Rencontrées

1. Problèmes techniques lors de l'installation des dépendances :
    - Problème Certaines dépendances, comme `react-native-reanimated`, nécessitent une configuration supplémentaire.
    - Solution : Nous avons suivi la documentation officielle pour résoudre les erreurs de compatibilité.

2. Alternance des machines de travail :
    - Problème : Certains membres ne disposaient pas de PC personnel performant.
    - Solution : Nous avons organisé des sessions de travail et utilisé des plateformes comme GitHub pour synchroniser notre travail.

3. Gestion des tokens Spotify :

- Problème : La génération des tokens d’accès nécessitait une bonne compréhension des permissions API.
- Solution : Nous avons effectué plusieurs tests et ajustements dans Spotify Developer Console.

Conclusion

Cette SAE nous a permis de travailler de manière approfondie sur plusieurs aspects du développement mobile. Nous avons appris à manipuler des API tierces, à configurer des services backend comme Firebase et à optimiser l’expérience utilisateur grâce à React Native. Les défis techniques nous ont permis de renforcer nos compétences en résolution de problèmes et en collaboration. Ce projet a été une occasion enrichissante d'appliquer nos connaissances tout en développant une application innovante et fonctionnelle.