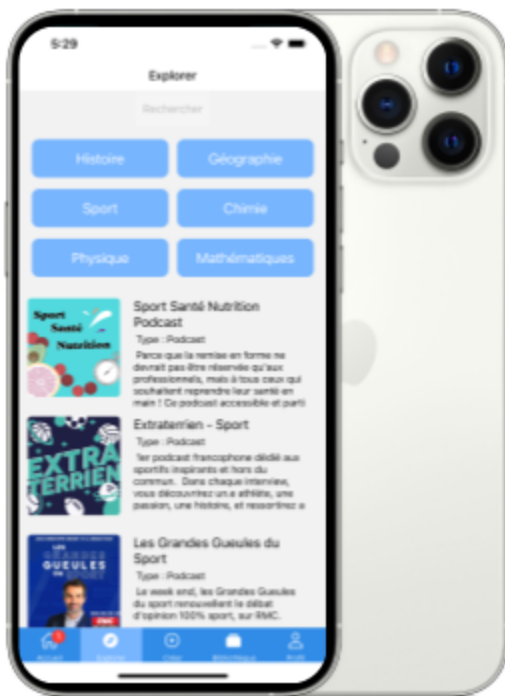


Projet Développement Mobile

Rapport de projet

Développement d'une application pédagogique



Apprendre n'a jamais été aussi simple !

Naviguer à travers des podcasts et vidéos à contenu pédagogique

Pelletreau-Duris Tom

Solan Laura

Vincent Amaury

Sommaire

Présentation	3
Installation	3
Fonctionnalités disponibles	4
Architecture	4
Apis Utilisés	4
Serveur utilisé	8
Diagramme des classes métiers	8
Exigences techniques	11
Organisation du travail	12
Perspectives d'amélioration	13
Conclusion	13

I. Présentation

Dans le cadre du projet de développement mobile, nous avons choisi de développer une application pédagogique. En effet, la fibre éducative et “user-friendly” n’est pas beaucoup développée, les applications ayant tendance à privilégier le temps de présence et l’engagement des utilisateurs. Au contraire, nous voulions mettre en avant des fonctionnalités simples d’accès, avec une interface épurée et une faible sollicitation informationnelle. Une telle application peut donc être appelée pédagogique à la fois par son contenu et sa forme. L’application permet de rassembler du contenus audio (podcast) et vidéos. L’objectif est de pouvoir effectuer des recherches par mots-clés ou en utilisant des filtres et d’obtenir des vidéos et podcasts sur le sujet. Les filtres choisis étant surtout à but éducatifs.

II. Installation

L’application n’étant pas disponible sur les plateformes de téléchargement pour smartphone (AppStore, PlayStore...), vous devez posséder Node.js et Expo sur votre ordinateur pour installer l’application et la tester localement.

Pour récupérer le code de notre application, vous pouvez

- soit vous rendre à cette [adresse](#) et télécharger le dossier contenant le code
- soit directement cloner le dossier (toujours via cette [adresse](#)) depuis Github ou Github Desktop

Une fois le code récupéré, ouvrez votre invite de commande et tapez la commande : **cd \Cheminversledossier\projet-2021-scolan_vincent_duris** pour que votre machine se place dans le dossier contenant le code. Tapez ensuite la commande **npm install** pour installer les librairies manquantes pour ce projet. Enfin effectuez **npm start**. Expo va automatiquement ouvrir une page web dans un de vos navigateurs. Cette page contient un Qr Code avec trois modes disponibles. Choisissez le mode tunnel. Quand l’écran affiche **Tunnel ready** vous pouvez scanner le QrCode directement depuis votre téléphone (téléchargez préalablement Expo) pour lancer l’application.

III. Fonctionnalités disponibles et non disponibles

En blanc vous retrouverez les fonctionnalités disponibles, en gris les fonctionnalités non disponibles.

Code	Fonctionnalité	Description
F_01	Rechercher un Podcast ou une vidéo youtube par mot clé	On peut rechercher et afficher les informations liées au podcast ou à la vidéo
F_02	Lire un Podcast ou une vidéo youtube	Depuis l'application l'utilisateur peut lire un podcast ou une vidéo youtube
F_03	Filtrer les podcats par mot clés	Plusieurs boutons apparaissent afin de pouvoir filtrer les résultats selon des catégories
F_04	Ajouter du contenu	Via l'onglet <i>Créer</i> vous pouvez choisir une image pour l'importer sur l'application. Actuellement, on ne peut pas encore récupérer de vidéos de la bibliothèque de son téléphone ni les associer à son compte. C'est une amélioration qui pourra être faite ultérieurement.
F_05	Accéder à son compte	Depuis l'application on peut accéder à ses données de compte
F_06	Accéder à ses favoris	Pouvoir enregistrer certains contenus multimédias dans une liste de favoris.

IV. Architecture

A. APIs Utilisés

Pour cette application, les données proviennent de deux API :

- [Youtube Data API v3](#)
- [Deezer API](#)

Youtube Data API v3

Cette API est utilisée pour récupérer les données liées aux vidéos Youtube. L'API nécessite une authentification, nous avons donc récupéré une clé d'identification via la plateforme de développement fournie par Google. Cette clé est personnelle et limite le nombre de requêtes possibles.

```
const rootEndpoint = "https://www.googleapis.com/youtube/v3/";  
  
const KEY = "AIzaSyCbTXX3Nch926X-2mawX5peBcLhvfWpCyY";
```

Concernant le code, les deux lignes ci-dessus nous permettent d'avoir l'URL de connexion à l'API et d'être reconnue par celle-ci pour pouvoir effectuer des requêtes. Nous avons rencontré des difficultés pour accéder aux données. En effet, nous avons un problème de CORS policy soit dû au navigateur soit au serveur utilisé pour déployer notre application. Pour palier ce problème nous avons utilisé un intermédiaire, une librairie nommée "axios". Axios est une bibliothèque JavaScript fonctionnant comme un client HTTP. Elle permet de communiquer avec des API en utilisant des requêtes. Son utilisation nous a permis de communiquer avec l'API de youtube beaucoup plus simplement et sans rencontrer d'erreurs.

```
export default axios.create({  
  baseURL: rootEndpoint,  
  params: {  
    part: "snippet",  
    maxResults: 2,  
    key: KEY,  
  },  
});
```

Voici la forme que prend notre requête dans le fichier **youtubedbapi.ts**, il s'agit d'une instance axios qui pourra être rappelée plus tard dans *ExplorerScreen*. Pour effectuer une

requête "axios", on donne simplement les paramètres de recherche : le nombre de résultats souhaité et la clé (ici key) qui permet à Youtube de savoir qui fait la requête.

```
onInput = async (text: string) => {
  const response = await youtubedbapi.get("/search", {
    params: {
      q: text,
    },
  });
  this.setState({
    donnees: response.data.items,
  });
  createYoutubeVideos(response.data.items);
};
```

Ici on exécute la requête en mode asynchrone afin de récupérer les résultats lorsqu'on en aura besoin. Il suffit donc d'utiliser notre **youtubedbapi** auquel on précise la méthode utilisée ici. Aussi, pour effectuer ici une recherche par mot clé, on a plus qu'à ajouter le chemin correspondant d'où l'ajout du **"/search"**. Il prendra aussi en paramètre le texte inscrit par l'utilisateur via la variable **text** pour effectuer la requête avec ce mot clé. Les vidéos obtenues par la requête sont ensuite transformées en composants "YoutubeVideo" afin de pouvoir les manier plus simplement.

Deezer API

Cette Api est utilisée pour récupérer toutes les données liées aux Podcasts. Dès lors que l'utilisateur utilise un filtre (Histoire, Géographie...) ou effectue une recherche via la barre de texte, une requête est envoyée à l'API Deezer et les données sont récupérées.

Dès lors que l'utilisateur appuie sur entrée après avoir écrit son mot clé, la fonction **onInput** liée à cette barre de recherche est appelée. Au travers de cette fonction, on prend en paramètre un texte qui est transmis à la fonction **searchDatabyWord** du service Deezerdbapi.

```
onInput = (text: string) => {
  Deezerdbapi
    .searchDatabyWord(text)
    .then((playlist: Array<Podcast>) => {
      this.setState({ playlist});
    });
};
```

La fonction **searchDatabyWord** utilise elle même la fonction **fetchFromApi** pour récupérer les données (nommées **datas** ici) et les convertir en objet de type **Podcast**. La fonction essentielle est bien évidemment la fonction **fetchFromApi**. En effet, c'est cette dernière qui permet de communiquer avec les données par un **fetch**. Le **fetch** prend en paramètre un **text** qui contient l'url communiqué par l'API deezer (soit <https://api.deezer.com/podcast/n°ID> pour une recherche de podcast par exemple) et retourne un tableau au format **json** nommé **data** récupéré par **jsonResponse.data**.

```
class Deezerdbapi {
  searchDatabyWord(word: string) : Promise<Array<Podcast>> {
    return this.fetchFromApi(`${rootEndpoint}?q=${word.trim()}`).then((datas) =>
      this.createPodcasts(datas));
  }

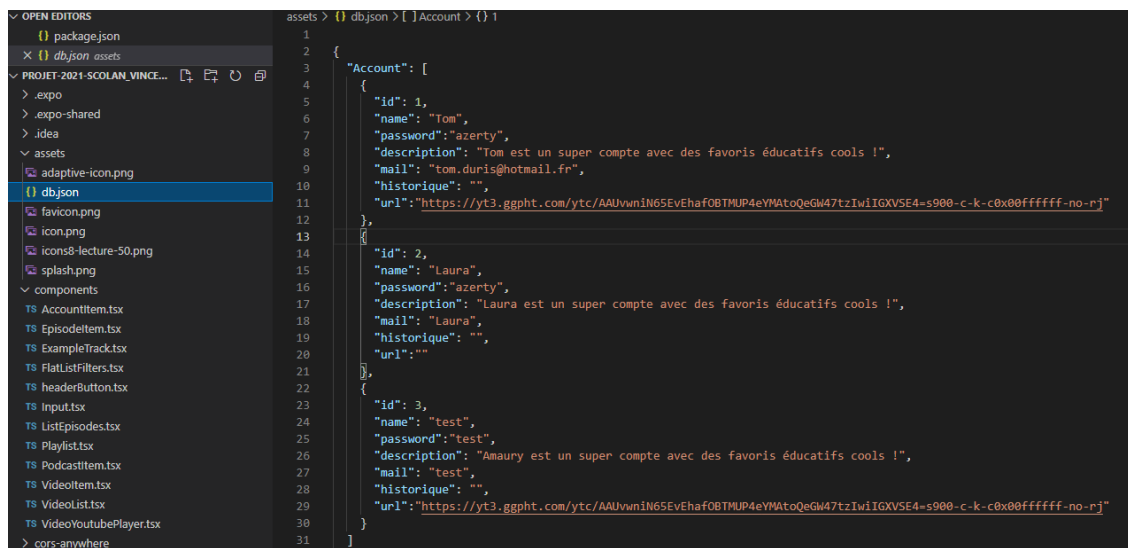
  private fetchFromApi(text: string): Promise<Array<Data>> {
    return (
      fetch(text, requestOptions)
        .then(response => response.json())
        .then((jsonResponse) => jsonResponse.data || [])
        .catch((error) => {console.log('error', error);
        })
    );
  }
}
```

Pour utiliser complètement l'API Deezer et récupérer les données liées aux podcasts, nous avons dû mettre en place différentes fonctions similaires au **fetchFromApi**. En effet, dès lors qu'on récupère le fichier json de données liées à un podcast, on peut récupérer l'url lié à ce podcast. Cependant, un podcast est une playlist d'épisodes et dans ce fichier json, la totalité des urls des épisodes n'est pas spécifiée. Nous récupérons donc l'identifiant du podcast pour récupérer la totalité des urls des épisodes via une fonction nommée **fetchbyIdPodcast**. Une fois exécuté, on peut retrouver toutes les données liées à un épisode.

Note : Deezer est une API dont les données sont protégées, c'est à dire qu'à aucun moment le fichier .mp3 sera renvoyé en interrogeant la base de données. La seule donnée liée à une piste audio est un lien vers Deezer inutilisable pour écouter l'épisode directement via l'application. N'ayant pas trouvé de player fourni directement via une API Deezer, nous avons décidé de proposer à l'utilisateur l'écoute du preview (30 secondes d'extrait) du podcast. Finalement, les podcasts n'ayant pas de previews, nous avons décidé de lancer la même piste audio au format mp3 à chaque épisode. Aussi, pour montrer que cela est possible avec un accès à une lien .mp3, nous avons affiché sur l'accueil le cas d'une musique possédant un preview (d'où la présence de la musique The real slim shady) à l'ouverture de l'application.

B. Stockage des informations utilisateurs

Pour stocker les informations utilisateurs nous avons d'abord essayé d'installer un JSON serveur. Mais nous avons eu beaucoup de bugs avec les "CORS" qui n'autorisaient pas l'accès à la base de données. Après quelques aventures, il a été décidé de simplement accéder à un fichier JSON appelé '**db.json**' et présent dans les **../assets/**.



```

1  assets > {} db.json > [ ] Account > {} 1
2
3  {
4    "Account": [
5      {
6        "id": 1,
7        "name": "Tom",
8        "password": "azerty",
9        "description": "Tom est un super compte avec des favoris éducatifs cools !",
10       "mail": "tom.duris@hotmail.fr",
11       "historique": "",
12       "url": "https://yt3.ggpht.com/ytc/AAUvwmiN6SEvEhaf08TMUP4eYMaToQeGM47tzIwiIGXVSE4=s900-c-k-c0x00ffffff-no-rj"
13     },
14     {
15       "id": 2,
16       "name": "Laura",
17       "password": "azerty",
18       "description": "Laura est un super compte avec des favoris éducatifs cools !",
19       "mail": "Laura",
20       "historique": "",
21       "url": ""
22     },
23     {
24       "id": 3,
25       "name": "test",
26       "password": "test",
27       "description": "Amaury est un super compte avec des favoris éducatifs cools !",
28       "mail": "test",
29       "historique": "",
30       "url": "https://yt3.ggpht.com/ytc/AAUvwmiN6SEvEhaf08TMUP4eYMaToQeGM47tzIwiIGXVSE4=s900-c-k-c0x00ffffff-no-rj"
31     }
32   ]
33 }

```

De cette manière, il était très simple d'accéder aux données json avec un simple :

```
import data from '../assets/db.json'
```

Ensuite, les données sont stockées dans un tableau de **'Account'** :

```
componentDidMount() {
  this.setState({ accounts: data.Account });
}
```

Et il ne reste plus qu'à vérifier l'identité donnée par l'utilisateur et à stocker la variable du compte dans une variable globale. Pour utiliser cette variable globale nous avons utilisé **MobX**. MobX est une librairie de gestion de state (state management) framework-agnostic suivant un pattern de type observer pattern. Ce pattern décrit des valeurs appelées Sujets où chaque sujet va pouvoir être observé par ce que le pattern appelle des Observers. Lorsque les sujets changent de valeurs, les observers sont notifiés et déclenchent les réactions adéquates. C'est justement cette notion d'observer que nous avons utilisé afin de faire communiquer les informations du compte avec tous les autres écrans. On peut voir ci dessous la classe **AppStore** qui permet de définir account :

```
import { makeAutoObservable, action, observable } from "mobx";
import Account from "../services/account";

export class AppStore {

  constructor() {
    makeAutoObservable(this, {
      account: observable
    })
  }

  account : Account;

  @action setAccount = ( account : Account ) => {
    this.account = account;
  }
}
```

De cette manière, il est possible de se connecter avec le mail : **"test"** et le password **"test"**, afin de tester l'application.

C. Architecture et Schématisation

Ce schéma représente les interactions entre l'application et les 2 APIs. L'application interagit directement avec l'API de Deezer. Par contre pour l'API de Youtube un passage par axios est obligatoire pour pouvoir récupérer les informations sur les vidéos.

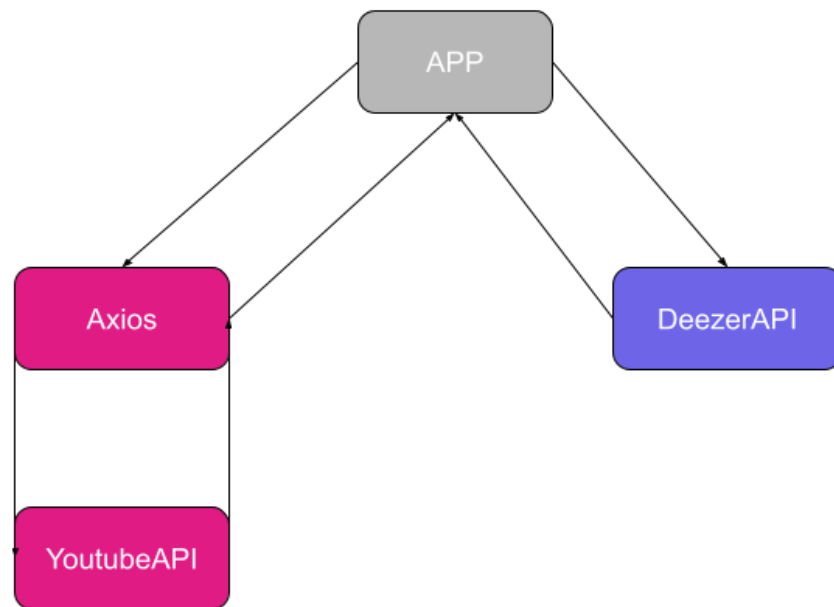


Figure 1 . Architecture de communication entre notre application et les diverses API.

Ci-dessous, un deuxième schéma montre l'organisation des différents composants de notre application. Les principaux composants sont au nombre de 4 :

- Input qui permet de gérer les champs où l'utilisateur peut inscrire ces requêtes.
- FlatListFilters qui permet de filtrer les podcasts selon certains critères
- VideoList qui permet d'afficher la liste des vidéos youtube.
- PlayList qui permet d'afficher les podcasts et les différents épisodes.
- AccountItem qui permet d'afficher les informations d'un compte

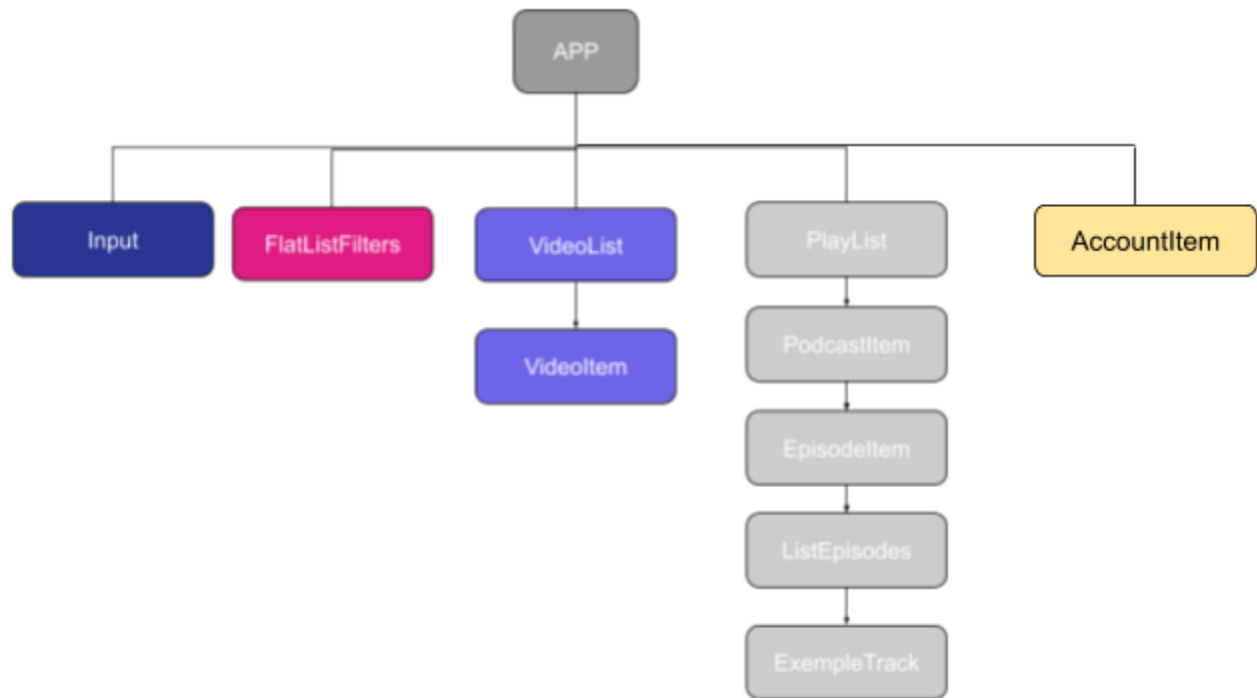


Figure 2. Organisation des composants

Enfin, ce dernier schéma montre les interactions entre les différents écrans depuis l'écran d'accueil "HomeScreen" (en effet, la page de "LoggingScreen" n'est pas directement connectée aux autres mais laisse la place au tabNavigator):

- HomeScreen : page d'accueil lorsqu'on arrive dans l'application
- AddScreen : page qui permet d'ajouter une image
- myAccountScreen : page affichant les données personnelles du compte
- LibraryScreen : page qui affiche la bibliothèque de l'utilisateur
- TrackScreen : page qui permet d'afficher l'exemple de lecture d'un son venant de l'API Deezer.
- ExplorerScreen : page qui permet de rechercher des vidéos youtube et des podcasts soit en mettant un mot clé soit par filtre.

- VideoScreen : page qui apparaît lorsqu'on clique sur une vidéo youtube depuis ExplorerScreen.
- PodcastScreen : page qui apparaît lorsqu'on clique sur un podcast depuis ExplorerScreen.
- EpisodeScreen : page qui apparaît lorsqu'on clique sur un épisode du podcast depuis PodcastScreen.

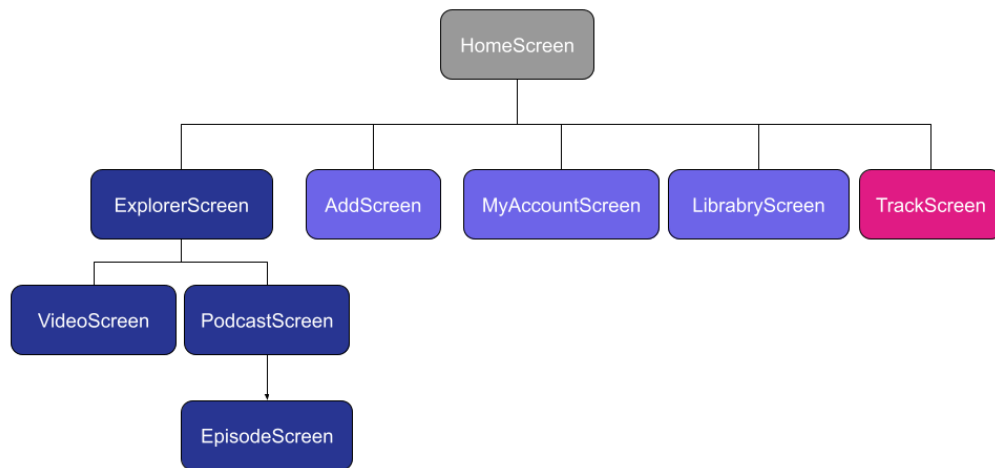


Figure 3 . Organisation des Screens

D. Exigences techniques

L'ensemble des exigences techniques imposées est résumées ci-dessous et a été respectée.

Code	Description	Respectée ?
ET_01	L'application est développée à l'aide d'Expo et basée sur le langage TypeScript.	✓
ET_02	Tous les fichiers sources sont formatés à l'aide de l'outil Prettier.	✓
ET_03	La structure de l'application distingue : - les composants élémentaires (/components) - les fichiers liés à la navigation (/navigation) - la ou les vues principales (/screens) - les classes métier et techniques pour les accès externes (/services)	✓
ET_04	Les props et l'état de tous les composants qui en font usage sont explicitement définis via des interfaces.	✓
ET_05	La granularité des composants est la plus fine possible (autrement dit : un composant trop "gros" doit être découpé en sous-composants plus élémentaires).	✓
ET_06	Les portions de code délicates et/ou importantes sont commentées.	✓

V. Organisation du travail

Pour ce projet, nous formions un groupe de trois personnes. Au départ, nous avons travaillé ensemble à distance lors des créneaux de cours dédiés au projet. Pour réussir à modifier le code à distance ensemble, nous avons utilisé la plateforme GitHub et le logiciel Github Desktop pour faire les différents push et pulls. Plusieurs branches ont été créées localement, pour développer distinctement les différentes fonctions.

Une réunion hebdomadaire était effectuée par semaine et à distance. Pour le sprint final, nous avons la possibilité de travailler ensemble à distance via la plateforme Discord ou de se réunir à l'école.

Concernant la répartition des tâches, nous avons établie dès le début la répartition suivante :

Tâches	Amaury	Tom	Laura
Implémentation Api Youtube	X		
Implémentation Api Deezer			x
Authentification d'un Utilisateur (login + server)		x	

Les screens pages et autres fonctionnalités ont été implémentées par l'ensemble du groupe.

Malgré la mise en place de cette organisation, le projet est livré en retard, ce dont nous avons conscience. En effet, nous avons préféré livrer un projet de meilleur qualité et fonctionnel. Il en reste que nous retenons de faire attention à ce point pour nos futurs projets professionnels et de mieux anticiper la quantité de travail en amont.

VI. Perspectives d'amélioration

Pour la suite de ce projet, les améliorations suivantes peuvent être envisagées :

Améliorations	Explications
Améliorer l'ergonomie et l'efficacité de l'application	Avoir une interface qui soit plus optimale en terme d'expériences utilisateurs en réalisant des tests utilisateurs
Travailler le design de l'onglet Explorer	Mettre en place un affichage horizontal des podcasts et des vidéos.
Enrichir les filtres	Permettre à l'utilisateur d'avoir des filtres plus nombreux, pouvoir sélectionner une recherche spécifique de podcasts ou de vidéos youtube
Améliorer l'ajout de contenu	Faire en sorte que l'utilisateur puisse ajouter

	une vidéo sur l'application et la retrouver facilement (Création de communauté lié à l'application)
Permettre à l'utilisateur de gérer ses favoris	Exemple : sauvegarder des vidéos pour qu'ils puissent facilement les retrouver.
Compléter la page d'accueil	Faire apparaître des vidéos en accord avec les goûts de l'utilisateur (recherche favorite par exemple)

VII. Conclusion

Pour conclure, ce projet nous a permis de nous familiariser avec la technologie React Native et son environnement. Il nous a permis de comprendre comment lier une application à une API déjà existante. C'était également intéressant de pouvoir se confronter à différents problèmes. En effet, ayant choisi 2 APIs au fonctionnement différent, nous avons dû nous adapter et trouver un moyen de réussir à communiquer avec l'API de Youtube. Enfin et surtout, nous avons pu apprendre à développer nos connaissances en développement mobile.