Ecole Nationale Supérieure de Cognitique - https://ensc.bordeaux-inp.fr

Projet Développement Mobile

Communify

Romaine BONNEFON
Thomas HOLSTEIN
Paul LORGUE

Année 2020-2021

Table des matières

Contexte du projet

Objectif du projet

Dans le cadre du module de Développement Mobile de 2ème année à l'ENSC, nous avons réalisé un projet d'application mobile où le but était de présenter des données issues d'une API web externe.

Nous avons choisi d'utiliser l'API Web de Spotify pour notre application. Le but de notre application est de permettre à plusieurs personnes d'ajouter des morceaux à la file d'attente d'un utilisateur Spotify; afin de choisir les prochaines musiques à diffuser sur un seul appareil externe (comme une enceinte). Ainsi chacun peut ajouter sa musique à une file d'attente commune grâce à des salles virtuelles fournies par l'application.

Notre application se découpe en plusieurs parties:

➜ La création de la salle. Un utilisateur possédant un compte Spotify l'associe à l'application.
➜ La connexion à une salle précédemment créée.
➜ Ajout de musiques à la file d'attente sur le compte de l'hôte.

L'API Spotify

L'API Web de Spotify permet d'accéder aux données relatives aux artistes musicaux, albums, pistes du catalogue de données Spotify ainsi que des listes de lecture et musique que l'utilisateur enregistre dans sa bibliothèque. Il est aussi possible de contrôler un compte utilisateur (ex: lancer la lecture d'un morceau, changer le volume, etc.)

Spotify demande une authentification pour pouvoir accéder à son contenu. L'application est identifiée par un identifiant et un code renseigné sur le site de Spotify. Ces identifiants sont nécessaires à chaque requête pour que Spotify identifie ces requêtes comme provenant de notre application. Ils permettent dans un premier temps de récupérer un token(chaîne de caractères) indispensable à la récupération des données, ce token expire au bout d'un certain temps (ici 1h).

Il existe ensuite 2 types d'autorisation:

- ➜ Pour accéder à la plate-forme Spotify et son contenu.
- ➜ Pour accéder et/ou modifier les données d'un utilisateur préalablement connecté.

Nous avons utilisé ces deux types d'autorisations au travers des procédures d'identification, **Client Credentials Flow** et **Authorization Code Flow**.

Avec Client Credentials Flow on demande un token à Spotify à partir des identifiants de l'application permettant ensuite d'effectuer des recherches de titres à partir d'une chaîne de caractères.

Avec Authorization Code Flow on demande à Spotify d'afficher dans un navigateur sa page de connexion permettant à l'utilisateur d'autoriser notre application d'interagir avec son compte et entre autres d'ajouter des pistes à sa file d'attente d'écoute. Une fois l'utilisateur identifié, Spotify renvoie un code à un url indiqué. Ce code permet l'obtention du token.

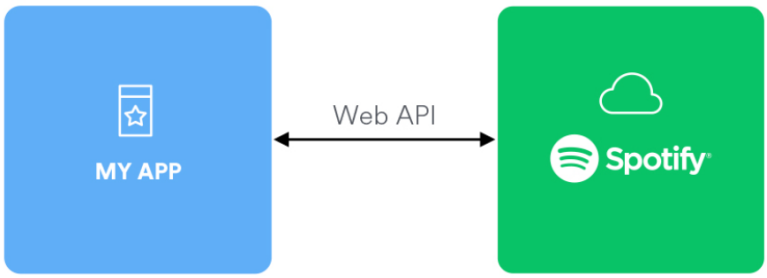
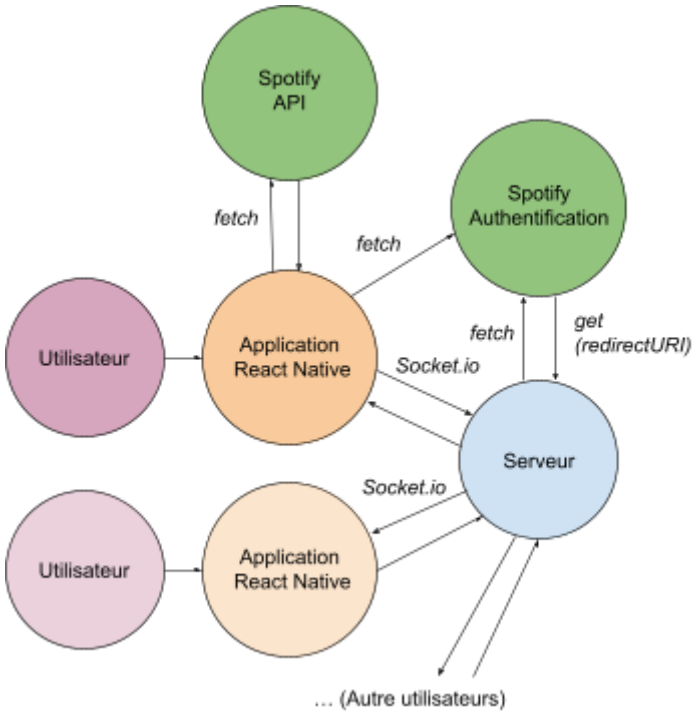


Illustration du fonctionnement d'une API web

Le serveur NodeJS, Express & Socket.io

Le serveur **NodeJS** associé à l'application a deux fonctions. La première étant de gérer les salles des différents utilisateurs afin qu'ils puissent communiquer ensemble (ajout des musiques) et la seconde étant de réceptionner les tokens d'authentification envoyé par l'API Spotify afin de permettre la connexion de nos utilisateurs.



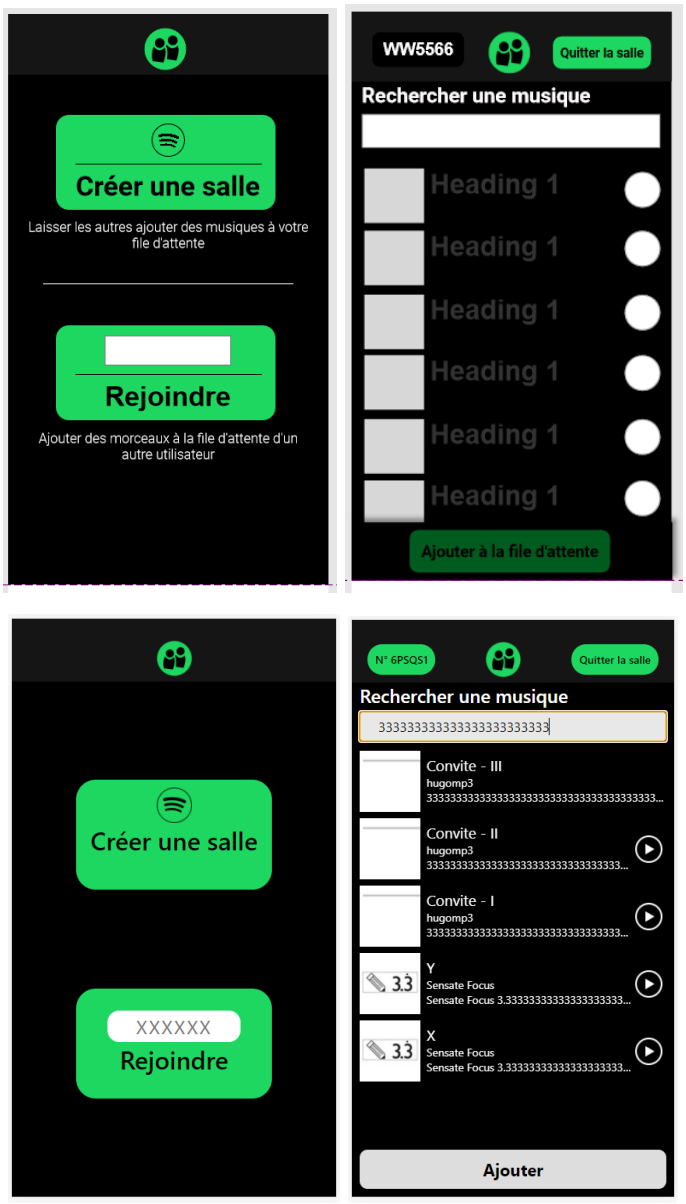
Le schéma ci-dessus représente les différentes communications entre les différentes entités (les utilisateurs et le serveur) et les différentes parties de l'API Spotify.

Nous avons un exemplaire hébergé de ce serveur (via l'hébergeur Projet Herberg) permettant de réellement connecter plusieurs appareils non en local ensemble (comme plusieurs téléphones). Ce dernier a été rédigé en TypeScript, compilé en JavaScript afin d'être exécuté par NodeJS. Il est également possible de le lancer en local (auquel cas il faudra modifier l'adresse du serveur pour localhost dans le service socket.io).

Les différents messages envoyés sont composés d'un titre (nature du message) comme "createNewRoom" ou encore "leaveRoom" suivi d'un corps (data), un objet JavaScript comportant les informations associées. (par exemple le numéro de la salle...)

Maquettes

Afin de partir sur une structure connue et commune, nous avons réalisé les maquettes de nos deux pages, celle d'accueil et celle de recherche de musique (dans une salle). Ces visuels ont peu changé ou un peu simplifiés dans la version développée.



Les maquettes sont les deux images de la ligne du haut et les écrans développées sont ceux de celle du bas.

Organisation de l’application mobile

L’application est composée deux pages utilisant 6 sous composants:
➜ **HomeScreen** : Page d’accueil

➜ **MusicScreen** : Page de recherche de musique et d’ajout à la file d’attente du propriétaire de la salle
(A noter, une page (RoomOption) a été imaginée mais non implémentée).

L’application est composé de 4 services:
➜ **ComunifyService** : Service permettant de communiquer/d’écouter le serveur avec des messages spécifiques à l’application (i.e. ajouter une musique, rejoindre une salle...). Il utilise le service suivant afin de permettre la communication avec le serveur sans avoir à écrire les fonctions socket.io à chaque fois.
A noter, ce service demande de lui fournir des fonctions à activer dans le cas de réception de message du serveur. (ie une réponse de l’authentification, une création de salle...)

➜ **SocketIoService** : Service effectuant la connexion et l’écoute, la communication avec le serveur NodeJS, offrant les fonctionnalités des bases. (Emettre/écouter un message)

➜ **SpotifyAccountService** : Service permettant l’accès aux données utilisateurs à travers l’Authorization Code Flow.

➜ **SpotifyTracksService** : Service permettant l’accès aux données de la plateforme Spotify à travers le Client Credential Flow.

Le serveur quant à lui est présent dans le dossier /server, il est composé d’un fichier TypeScript (utilisé pour sa réalisation) et d’un fichier JavaScript (étant simplement le fichier TypeScript compilé) permettant son exécution.

Exigences techniques

Le tableau ci-dessous énumère les différentes exigences techniques de l'application, exprimées dans la description du projet. Elles ont toutes été remplies.

Désignation	Description	Réalisation
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.	✓

Fonctionnalités

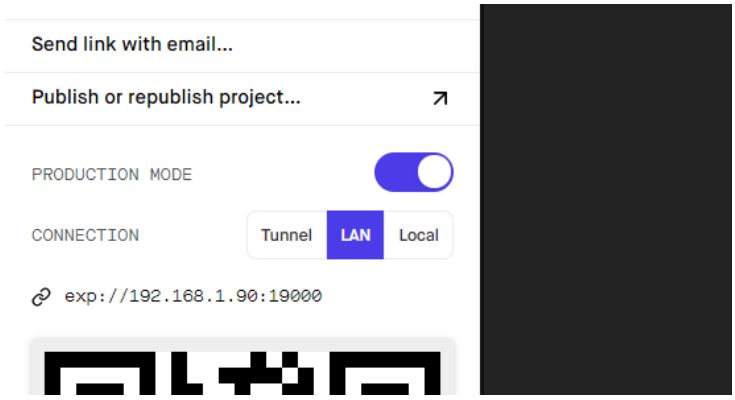
Le tableau ci-dessous énumère les différentes fonctionnalités de l'application.

Désignation	Description
EF_01	L'utilisateur peut créer une salle et identifier aisément celle où il se trouve. Une fois la salle créée, cela génère automatiquement un numéro de salle qui est affiché en haut à gauche.
EF_02	L'utilisateur peut rejoindre une salle qu'il n'a pas lui-même créée à partir du numéro de salle correspondant, ce qui permet également un certain niveau de sécurité.
EF_03	L'utilisateur peut rechercher des musiques grâce à une barre de recherche et à partir du titre de la musique, du nom de l'artiste ou du nom de l'album.
EF_04	Les 10 premiers résultats de recherche s'affichent pendant la saisie. Les autres résultats de recherche sont chargés au fur et à mesure que l'utilisateur arrive en fin de liste.
EF_05	L'utilisateur peut ajouter des musiques à la file d'attente d'écoute du compte Spotify lié à la salle.
EF_06	L'utilisateur peut quitter la salle où il se trouve pour en rejoindre une autre.

Installation du projet

Installation et Exécution

1. Télécharger l'ensemble des fichiers du projet
2. Installer Node.js sur votre ordinateur
3. Ouvrir une fenêtre de commande et se placer dans le répertoire où se trouve le projet
4. Installer les packages node en effectuant la commande : *npm install*
5. Installer Expo en effectuant la commande : *npm install -g expo-cli*
6. Lancer l'application : *expo start*
7. Ouvrir l'application via un smartphone avec l'application Expo Go, via un émulateur/appareil virtuel, ou via le web browser (navigateur)
8. Activer le Mode production d'expo



Switch PRODUCTION MODE activé

PS: Sur **navigateur** quelques warning peuvent apparaître dans la console dû au fait que c'est un navigateur et non un téléphone. En se connectant avec un compte spotify sans être connecté à expo un autre warning peut apparaître. Rien de cela n'empêche la bonne exécution de notre application, mais ils ne sont pas résolvables par nous.

PPS : Lors de l'exécution via **émulateur/téléphone**, expo gère mal l'exécution des scripts en arrière plan, dont la connexion socket.io. De fait, au bout d'une vingtaine de secondes en arrière plan (téléphone éteint, page web de connexion active) l'émulation de l'application expo redémarre (donc revient à la page principale).

PPPS: L'application Spotify sur Ordinateur ne met pas à jour l'affichage de la playlist correctement, les musiques sont bien présentes dedans, jouées mais elles s'affichent grisées.

Choix techniques

Serveur NodeJS

Le choix de réaliser un serveur NodeJs a été premièrement justifié afin de permettre la communication entre les différents clients par l'intermédiaire du serveur. Sans cela, il est impossible de trouver les autres utilisateurs ni de leur envoyer des messages. Socket.io permettant une communication simple et complète, elle est apparue totalement adaptée dans notre cas.

Ce choix a d'autant plus été confirmé lors de la mise en place de l'authentification Spotify, demandant une adresse de redirection pour envoyer les différents tokens de connexion de l'utilisateur. Le serveur NodeJS permet donc de faire tourner socket.io et de réceptionner les requêtes get.

Gestion des événements

Concernant les différentes interactions entre les components/services, différentes fonctions sont déclarées dans le App, et transmises via les props (pour les components) ou dans le constructeur (pour les services). Ces fonctions sont appelées quand un certain événement se produit (un click, un salle est créée/rejointe, ...).

Cette solution convient à notre application car elle est assez petite, mais une solution comme MobX avec des Listener/Observer pour les différents événements seraient possible.

Stockage des informations

Côté client, les informations sont gérées basiquement grâce aux states des différents composants, les informations principales étant dans le component App. Il n'y a pas de persistance de mémoire.

Côté serveur, les différentes salles sont sauvegardées. Une salle est identifiée par un Id (6 chiffres ou lettres), qui lui est unique, et a pour information le propriétaire, les utilisateurs dedans ainsi que les options associées. Ces informations sont stockées dans un objet JavaScript Map, et ne sont pas persistantes.

Gestion de projet

Nous avons séparé ce projet en deux parties qui ont été développées en parallèle:

➜ L'utilisation de l'api Spotify dans React Native (développée par `Paul Lorgue`)
➜ Le serveur Socket.io et sa liaison avec React Native (développé par `Romain Bonnefon`)

Nous avons effectué plusieurs séances de Peer Programming avec `Thomas Holstein` afin partage

Bilan

Difficultés rencontrées

La principale difficulté rencontrée pendant ce projet est l'Authentification.

Le serveur NodeJS et Socket.io a déjà été vu grâce au projet personnel informatique d'un des membres de l'équipe, il s'est ainsi donc fait sans problème.

Pistes d'amélioration

Même si l'application est fonctionnelle, elle n'en demeure pas moins parfaite. Il serait intéressant de tester si, en version exportée, les scripts en arrière-plan s'exécutent bien, ce qui permettrait à Socket.io de rester connecté avec un appareil vérouillé. Si ce n'est pas le cas, cette fonctionnalité serait très importante pour une réelle utilisation.

De plus, notre barre de recherche peut potentiellement être améliorée, permettant d'avoir des résultats moins sensibles à l'orthographe par exemple.

Comme vu précédemment, il serait intéressant d'améliorer notre gestion d'évènement, notamment à la réception de message dans le service utilisant socket.io. Il serait possible d'utiliser MobX par exemple, avec des Listerners/Observers se déclenchant lors de nos différents évènements.