

VOOOV

Projets personnels encadrés, dans le cadre du BTS SIO – Option
SLAM.

Année 2023

Sommaire

Table des matières

Préface du projet.....	5
Identification du besoin	6
Préambule	7
Vue d'ensemble du projet	8
Étapes du projet	9
Début du premier dossier	11
Le design	11
L'UX	11
Tendances	11
Application des tendances au projet	12
Modélisation des données et structure du projet	14
Modélisations des données	14
Base de données	19
Choix de base de données	19
MPD et Script SQL	19
Implémentation des services.....	23
Création du VPS.....	23
Déploiement et hébergement des différents services	23
Sécurisation du serveur	27
Améliorations possibles.....	27
Mise en place d'un service de versioning	28
Utilité du versioning	28
Utilisation de Git.....	28
Utilisation pour le projet	29
Ticketing et tests	30
Le ticketing	30
Phase de conception et de test	30
Phase de production	31
Déploiement de la base de données	32
Procédures stockées	32
Conception, création et implémentation de l'API	33
Conception	33
Créations de l'API	33

Erreurs corrigées	37
Amélioration du code	38
Liste des fonctions de l'API	41
Tests	59
Fin du premier dossier	60
Début du second dossier	61
Conception de l'application mobile (Android/Kotlin)	61
Réalisation de la partie frontend	61
Le dossier « values »	61
Le dossier « layout »	64
Les dossiers complémentaires du frontend dans res	66
Réalisation de la partie Backend	67
Langage Kotlin	67
Manifest et Gradle	67
Création d'une application de test	67
Conception au format MVVM	68
Bibliothèque Retrofit	68
Models	69
Les repository	70
Les ViewModels	70
Fonctionnalités d'application	72
Gestion des threads et de l'asynchrone d'Android	72
Réalisation de la partie identification et inscription	73
La messagerie et les recyclerViews	74
Réalisation de la partie audio et gestion des fichiers	76
Exploration de l'application, Activités et fragments	78
Conception de l'Application Web	82
Les bases du Framework Symfony	82
Organisation et architecture de Symfony	84
La couche Model	84
La couche View	84
La couche Controller	84
L'ORM (Object-Relational Mapping) Doctrine	84
Etapes de construction du projet	85
Création des entités	85
Authentification et inscription	85

Formulaires	86
Controllers.....	86
Vues.....	86
Fonctionnalités poussées	87
Barre de recherche	87
Création des enregistrements	88
Back-office avec easy-admin	89
Déploiement du projet sur le serveur VPS	91
Htaccs	91
Composer	92
Github.....	92
Mode dev et mode prod.....	92
Présentation visuelle de l'application Web	94
Fin du second dossier	97
Post projet.....	98
Evolutions et réflexions au cours du projet	98
Fonctionnalités restant à développer :.....	98
Application Web :	98
Application mobile Android :.....	98
API :	98
Serveur :	99
Design :.....	99
Liste des difficultés rencontrées.....	99

Préface du projet

Le projet présenté par la suite, suit les étapes d'une solution d'information réalisée dans des conditions réelles en mode projet.

Depuis le commencement de ma formation en BTS SIO Option SLAM, j'ai pu approfondir mes nouvelles connaissances en répondant aux besoins de mon entourage.

Les premières réalisations sont des sites web. Ils m'ont permis de maîtriser les bases des principaux langages du web, à savoir le HTML, le CSS, le JavaScript et le PHP, ainsi que la bibliothèque Symfony.

La demande par des clients réels m'a motivé à m'impliquer davantage. Elle m'a aussi forcé à rendre des outils fonctionnels. Cela m'a obligé à pousser mes recherches, jusqu'à trouver des solutions, aux divers problèmes rencontrés.

Les modifications demandées par les clients, m'ont clairement fait apparaître l'importance d'un code propre, lisible, optimisé et maintenable. Un code bien commenté et bien construit permet de gagner du temps en cas de retour sur celui-ci, mais également dans de nouveaux projets.

Lors du projet présenté ici, j'ai pu mettre en œuvre mes connaissances en Java ainsi qu'en Kotlin.

Ce projet scolaire comporte deux projets, faisant office de projets personnels encadrés. Il mettra en œuvre les compétences variées et nécessaires, pour la création d'une solution d'information, et attendues pour la validation du BTS SIO option SLAM.

Identification du besoin

Le projet présenté est né d'une demande concrète. Une personne de mon entourage, m'a contacté pour la réalisation d'un site web. L'objectif du site était de faire la promotion de sa carrière de Voix-Off.

Le projet naît et se nomme Vooov, pour voice-over : Voix-Off en anglais.

Après la réalisation de plusieurs sites web et souhaitant continuer de gagner en compétences durant ma formation, je souhaitais m'essayer à d'autres formes de programmation. Mon but était de réaliser un projet en Java.

L'idée prend forme petit à petit, et après quelques recherches, je n'ai pas pu trouver d'application mobile mettant en lien candidats à la Voix-Off et recruteurs. De plus l'idée m'est apparue que les besoins dans ce domaine pourraient sans doute être étendus et généralisés à d'autres utilisations. Pourquoi pas moins professionnelles, événements familiaux, partage de mémoires, vidéos type Youtube...

Le constat est un certain retour de l'audio, là où le visuel avait pris toute la place. On voit grandir l'utilisation des « Voices » de Facebook prenant une part d'utilisation sur le sms, les appareils tel qu'Alexia de google, répondant à la voix, s'installant dans les foyers et les connectant parfois en domotique. Le nombre d'utilisateurs écoutant, musiques, audiolivres et podcasts, remplaçant la radio, et ce parfois même en voiture, grandit. Simplement, nous utilisons de plus en plus les technologies récentes pour le contenu audio. Les utilisateurs renouent avec ce sens oublié, en lui prêtant même un côté futuriste et amusant.

Le cœur de cible paraît étendu, puisque l'application peut servir à des besoins professionnels ou ludiques. Les utilisateurs peuvent être de tout âge et venir de tout milieu social.

L'idée finale actuelle est donc de réaliser une application sous forme de réseau sociale de type vocal. Celle-ci devra faciliter la mise en lien entre les candidats Voix-Off et les employeurs. Elle devra également permettre le partage de contenu audio entre personnes non professionnelles.

Cette application a pour objectif de rendre possible un échange direct et privé entre utilisateurs. Elle doit également rendre possible une forme de contribution entre utilisateurs, interne à l'application, en échange de services vocaux.

Elle doit être accessible depuis plusieurs plateformes (ordinateurs, tablettes et mobiles) et doit donc permettre de lier les données entre ces différents médias.

Préambule

Une fois le premier besoin identifié, il est nécessaire de mettre en place une forme de pré-architecture abstraite du projet sous forme de brain-storming. Sur ce media en forme nuage, nous plaçons toutes les idées qui peuvent apparaître et qui pourrait paraître cohérentes. Il faut ensuite faire le tri entre les idées essentielles et celles risquant de faire perdre de la consistance, ou encore simplement ayant vocation à apparaître ultérieurement.

Le voici :

Doit :	Choix des contraintes	Idées pour le futur:	Idées mises de côté pour le moment:
Pouvoir être enregistrer depuis un Téléphone	Choix d'une durée maximale d'enregistrement	système de gain sur les jetons. Augmentation de 10% du prix d'un jeton à chaque doublement du nombre total de jetons.	ajout d'images pour les enregistrements. Trop de poids supplémentaire.
Mettre en lien candidats voix-off et recruteurs	format d'enregistrement	Moyen de paiement pour l'achat de jetons. Permet de payer directement depuis le site de manière sécurisée.	
Permettre l'échange entre les utilisateurs		Envoyer des enregistrements audio par messages.	
Etre agréable à regarder		Créer des salles : de réunions, de concerts	
Importer des fichiers audios externes		Créer des conversations de groupes	
Donner la possibilité de choisir une type de voix et un type de contenu		Permettre la retouche des enregistrements	
Permettre la recherche d'enregistrement et d'artiste			

Choix du nom de l'application :
Vooov (pour Voice-over, qui signifie voix-off en anglais)

Par la suite, il faut définir les médias à travers lesquels le projet pourra prendre vie. Il faut définir via quelles méthodes de mise en œuvre cela sera réalisé. A cette étape, on met en lien les compétences de l'équipe et les besoins du projet.

Pour ce projet, dont le but est de permettre un échange entre des utilisateurs, le plus de médias possibles semble être le meilleur moyen d'agrandir la visibilité.

Les compétences disponibles, acquises durant ma formation se compose de langage du web, de Java et de langage C. On peut également ajouter des compétences en gestion de base de données SQL et NoSQL, ainsi que des notions en déploiement, hébergement...

Durant cette expérience, j'ai pu me familiariser avec le langage Kotlin.

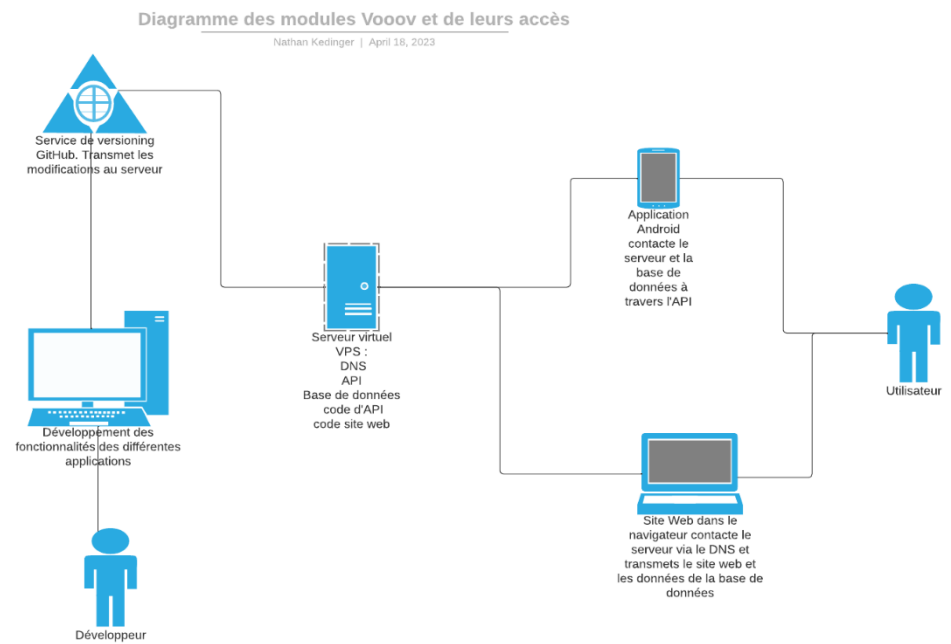
Ces compétences ouvrent donc trois champs :

- Application mobile
- Site web
- Application desktop (application dite lourde)

Afin de mettre en lien les données de ces trois réalisations, il faut ajouter au projet une API (Application Programming Interface). Elle permettra des gérer la base de données de manière uniforme depuis les trois médias. (Une première version de l'application : OFFO a été réalisée via l'API Firebase, qui offre une base de données NoSQL stocké dans le cloud. Afin de répondre à des besoins scolaires, le projet a été modifié en créant une API propre, liée à une base de données MySQL)

Vue d'ensemble du projet

Avant toute chose, voici un schéma réseau réalisé avec Lucidchart. Il permet d'avoir une vue d'ensemble du projet en retraçant les divers modules et leurs interactions les uns avec les autres :



Étapes du projet

Ce projet global se divise en trois. Une fois les médias et les compétences passés en revue, il est nécessaire de définir un ordre d'approche au projet. Le suivant a donc été défini :

« Début du premier dossier »

- **Le design** : Maquettage des différents formats. On crée une identité visuelle commune. Définition de la colorimétrie, de la typographie et du logo. Ici c'est l'outil « Figma » qui est utilisé pour le maquettage de l'ensemble des pages web, application mobile et lourde. Le logo et les icônes sont réalisés via le logiciel « Inkscape ».
- **Modélisation des données** : MCD (Modélisation Conceptuelle des Données) grâce au logiciel « StarUML », MLD (Modélisation Logique des Données) grâce à l'outil « dbdiagram.io ».
- **Script pour la base de données** : MPD (Modélisation Physique des Données) Une fois les besoins définis, il faut retranscrire le script de création de Base de données
- **Implémentation des services** : Afin d'héberger les différentes parties du projet, il faut définir un service d'hébergement. L'API, la base de données, les fichiers de l'application lourde et le site web sont hébergés sur serveur VPS dédié, hébergé par « Hostinger », sous Ubuntu. L'outil « CyberPanel » est utilisé afin de faciliter l'administration du serveur. L'application mobile sera hébergée dans un second temps sur le « Playstore d'Android ».
- **Mise en place d'un service de versioning** : Vous pourrez retrouver l'ensemble des versions de mes repositories sur « Github » à cette adresse : <https://github.com/nathan-kedinger?tab=repositories>.
- **Gestion des tickets** : La gestion des tickets en phase de conception et de test sera réalisée via « Github » et Trello. Une gestion des logs d'erreur et de connexion est également enregistrée directement sur serveur. Au quotidien, j'utilise « Trello » afin de lister les différentes tâches à effectuer. Notamment les tickets à résoudre.
- **Déploiement de la base de données** : Une fois le serveur VPS activé et utilisable, il est temps de déployer la base de données et d'y implanter les scripts créés précédemment. Ici l'outil de gestion de base de données est phpMyAdmin qui utilise MySQL.
- **Conception, création et implémentation de l'API** : L'API créée est une API Rest codée en PHP, permettant une gestion CRUD (Create, Read, Update, Delete) des données via des protocoles http sécurisés. On implémente des processus de test aux différents points de liaisons et on teste le fonctionnement grâce à l'outil « Postman ».

« Fin du premier dossier »

« Début du deuxième dossier »

- **Conception de l'application mobile** : L'application mobile est créée pour les smartphones et tablettes Android et réalisée via l'IDE d'Android Studio fonctionnant sous l'IDEA IntelliJ . Elle sera déployée sur le PlayStore d'Android. Le code source de l'application est écrit dans le langage Kotlin.
- **Conception de l'application web** : L'application web est créée via le Framework Symfony 6 utilisant PHP et réalisé via les IDE Visual Studio Code et PHPStorm. Elle sera déployée sur le serveur VPS de l'API.

« Fin du deuxième dossier »

Mise en place de tests de performances :

Implémentation de services d'intégration continus :

Ouverture et améliorations à venir :

Début du premier dossier

Le design

L'UX

Dans tout projet, l'esthétique joue un rôle clé pour l'UX (expérience utilisateur). L'UX ne définit pas uniquement le design, mais également la maniabilité, la facilité d'utilisation, le contenu, l'ergonomie...

Aujourd'hui, les utilisateurs ont accès à une application pour presque tous les usages de la vie courante. Ils ont l'habitude que cela fonctionne rapidement et ils n'hésiteront pas à quitter l'application s'ils rencontrent des ralentissements ou quelques difficultés. Ce surtout dans les premiers instants d'utilisation.

L'UX tient donc un rôle clé dans le fonctionnement d'une application.

Le rôle du design est d'apporter une identité qui permettra de reconnaître le produit. Il doit également permettre à l'utilisateur un confort visuel. Il doit donc être agréable à regarder, mais également correspondre aux attentes de l'utilisateur, pour le type d'expérience qu'il vient chercher.

Tendances

Ce projet doit répondre à un cœur de cible varié (âge, professionnels/particuliers, sexe...). Il doit donc présenter une esthétique parlant naturellement à tous. Il doit également être dans l'air du temps. Ce projet est basé sur une expérience vocale, auditive et de partage.

Voici une liste, non exhaustive, d'objets de la vie courante répondant à cet imaginaire collectif : Radio, studio d'enregistrement, enceintes, amplis, micros, téléphone, bouches, oreilles, musique, réseaux sociaux, échange, bien-être, écoute... Le groupe d'applications créé plus tard cherchera à s'inspirer de cette liste.

Il est à noter l'aspect psychologique probable vis-à-vis du contenu en création. Par suite d'une recherche d'inspiration, il semble apparaître certains points :

- L'audition peut apporter une certaine forme d'introspection.
- Elle est principalement vue comme liée à la musique et à la parole.
- La musique est source d'émotion. Ces émotions peuvent être fortes, douces, agréables, nostalgiques... (Afin de garder l'attrait des utilisateurs, les aspects positifs sont mis en avant)
- La musique agit sur la concentration.
- La musique est vectrice de calme, mais également de forte énergie.
- La parole se rapporte au langage. Elle transporte le savoir et les informations. Elle permet les échanges sociaux et la communication.
- Le résultat nous porte à croire que dans l'imaginaire visuel collectif, le son est souvent apparenté à un esprit vintage.
- Le son est lié à la créativité.

Cela suppose d'accueillir l'œil dans un milieu calme et rassurant, au potentiel énergisant et créatif. Il est nécessaire de faire appel à des images communes et connues.

Les couleurs prédominantes aperçues lors de cette recherche sont, soit, des couleurs sombres, soit, au contraire, des couleurs très brillantes. La couleur bleue ressort régulièrement (souvent associé à l'apaisement et la vérité. Elle est également liée au bien-être et à la spiritualité, au calme et à la communication). Cela correspond bien à notre sujet. Les formes sont le plus souvent douces et arrondies.

Application des tendances au projet

L'étape suivante consiste à créer une planche de tendance. Elle est composée d'un ensemble d'images correspondant aux objets du quotidien identifiés précédemment. Elle permet de déduire une ligne de couleur et de formes.



Les années 90 sont aujourd'hui de retour à la mode. Cependant, il est important de se concentrer sur le point de vue actuel de ces années. Les utilisateurs se souviennent rarement précisément du design d'il y a 30 ans. L'aspect rassurant d'une œuvre, provient principalement, de la sensation d'évoluer dans un milieu, original, mais familier.

Concernant la police d'écriture, l'application sera composée de deux typographies. L'une servira pour le titre de l'application qui rappelle les vinyles : **MONOTON**. La seconde reste sobre et lisible pour ne pas déranger l'utilisateur : Biryani.

Une fois la trame principale fixé, il est possible de passer à la mise en forme concrète du design. Précéder la mise en production du site par une mise en forme visuelle permet de gagner un temps considérable.

L'outil Figma apporte une prévisualisation précise des futurs écrans. Cet outil permet une création et une modification des visuels extrêmement rapide en comparaison à ce que demande le CSS d'une application web, ou le XML d'une application mobile et desktop.

De plus, le vocabulaire utilisé dans le projet peut d'ores et déjà être constitué. La mise en production n'aura plus qu'à suivre pas à pas le fichier sans réflexion supplémentaire.

NB : Pour certains projets, le vocabulaire est une tâche dédiée à une équipe spécialisée.

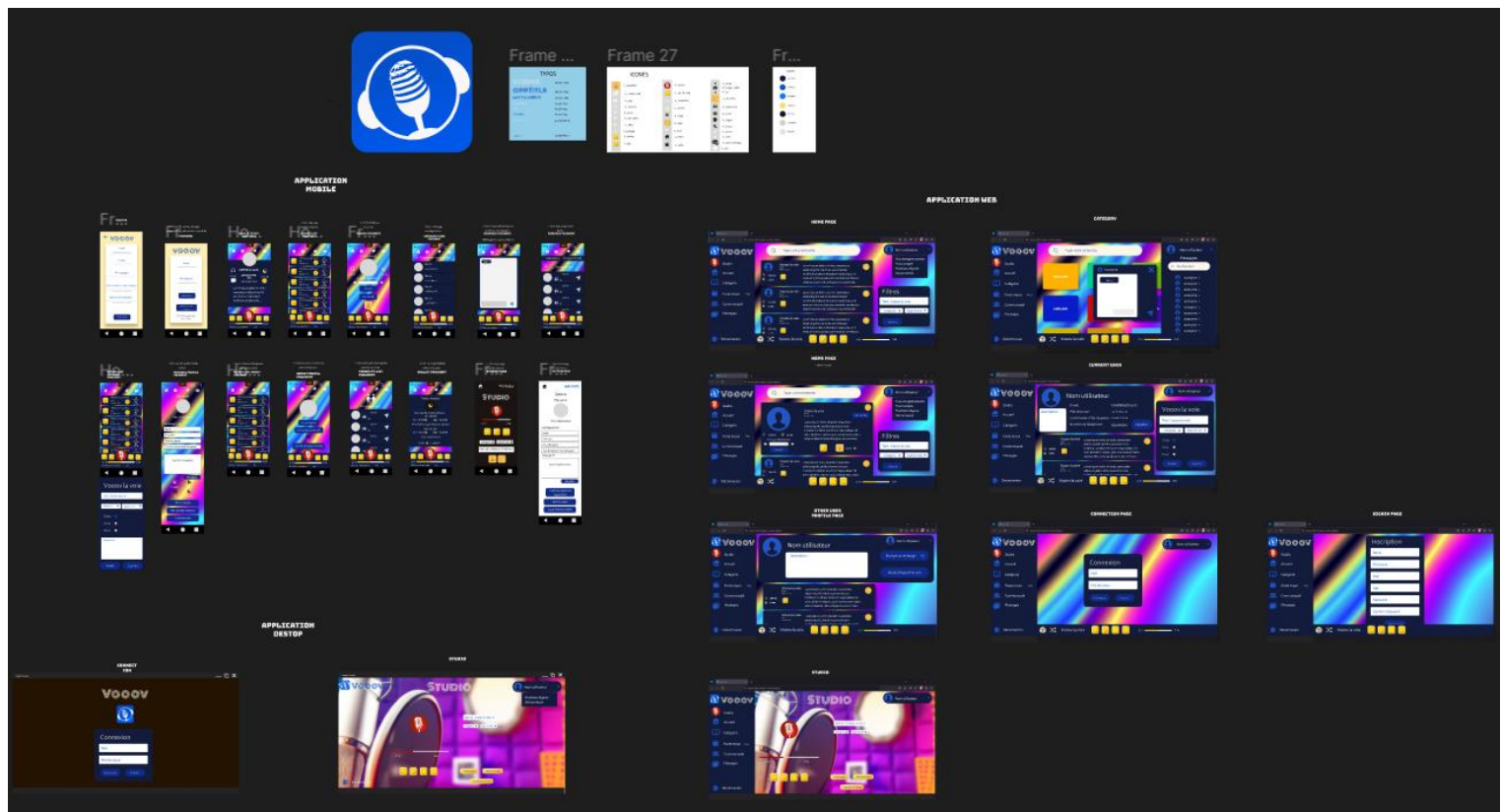
Le visuel général permet également de visualiser plus précisément le projet et les différentes interactions qui auront lieu. Cela nous aide pour la partie suivante : la modélisation des données.

La majorité des icônes est réalisée via l'outil Inkscape.

Vous pouvez retrouver le fichier complet en cliquant sur ce lien :

<https://www.figma.com/file/G2RugGS4NajZlOXKEV5JHs/OFFO?node-id=0%3A1&t=Kkk0Ub5RggEP4woG-1>

En voici également une capture d'écran :



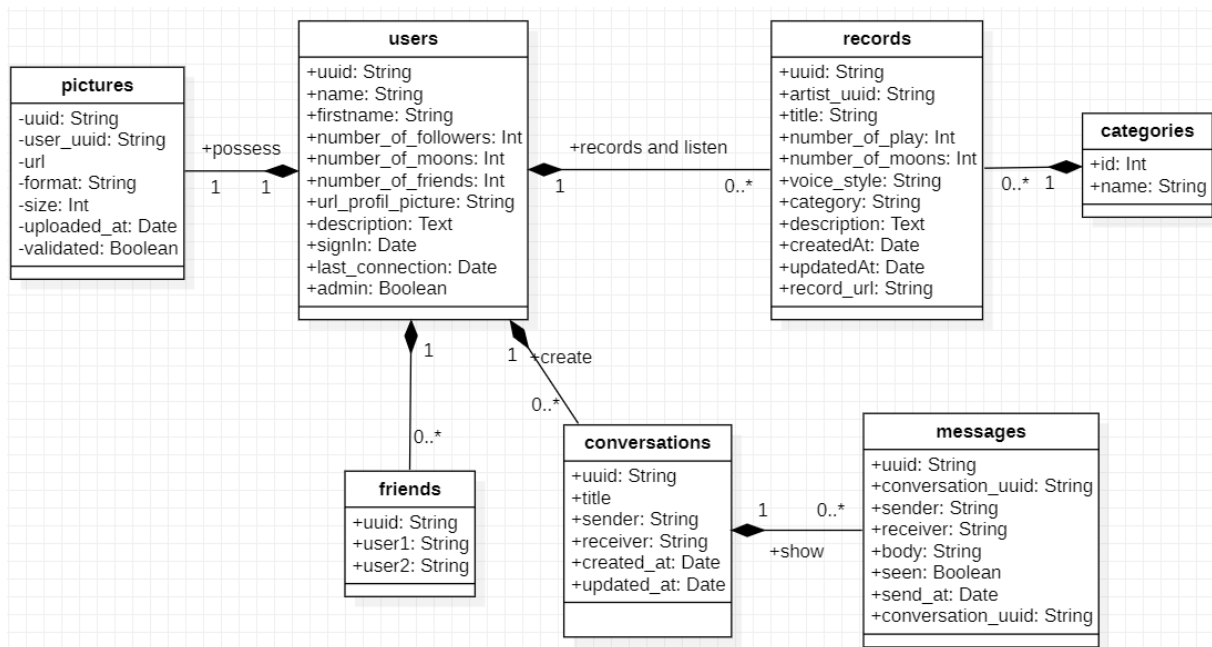
Modélisation des données et structure du projet

Modélisations des données

La modélisation des données consiste en trois phases : MCD (Modélisation Conceptuel des Données), MLD (Modélisation Logique des Données) et le MPD (Modélisation physique des données).

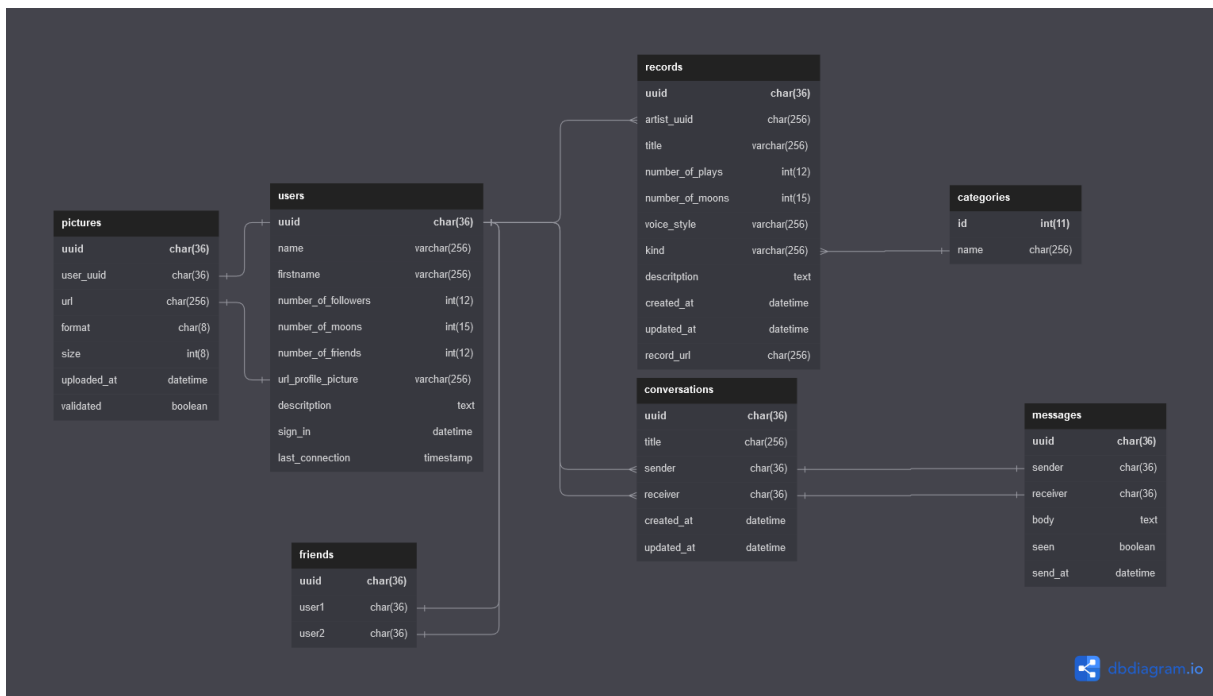
La MCD permet de présenter une vue d'ensemble des données et ainsi, apporter une visibilité sur la finalité de l'application. Elle est réalisée ici via le logiciel StarUML.

La voici :



Vient ensuite la MLD. Celui-ci découle du MCD, en introduisant les relations et les détails contextuels nécessaires à la structure des données. Elle permet de s'approcher de la mise en œuvre finale. Elle met notamment en avant les clés primaires, les clés étrangères, le type de données en base de données et le nombre de caractères par ligne de chaque table (cela permet de connaître le poids d'une ligne et ainsi poids de la table selon la quantité de lignes). Elle est mise en œuvre ici grâce à l'outil dbdiagram.io.

Voici le diagramme en résultant (NB : les clés primaires ne sont pas présentes sur la représentation) :



L'étape de la MPD intervient dans la phase suivante de création et met en œuvre les deux étapes précédentes. Elle sera présentée au prochain point.

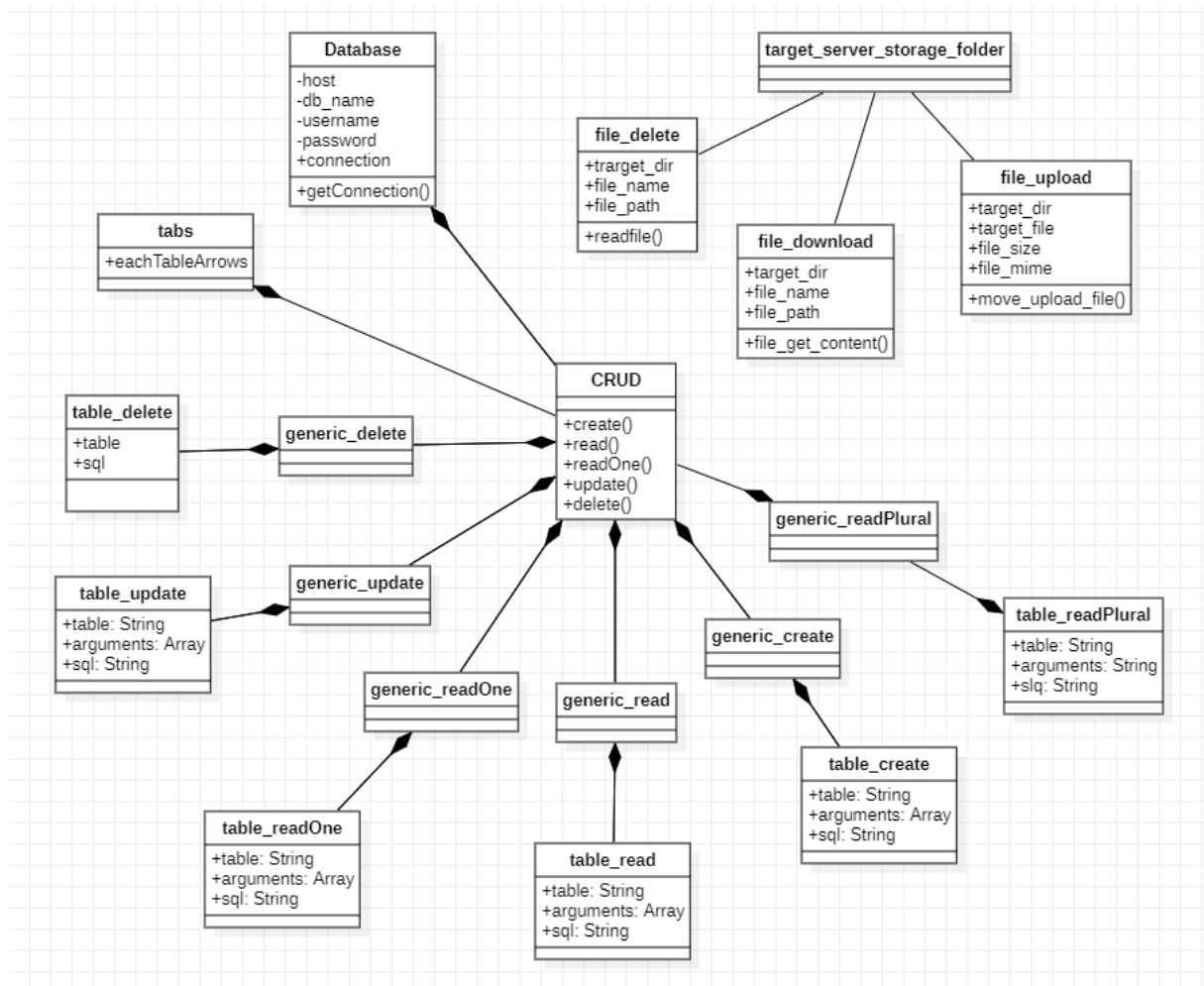
Pour prévoir au mieux les étapes du projet et répartir plus facilement les tâches, il convient de réaliser des diagrammes de classes. Chaque application nécessite son diagramme, y compris l'API. Ceux-ci sont des bases de travail et sont amenés à évoluer et se préciser selon les besoins. Le design en place permet d'y voir plus clair.

L'architecture du projet est constituée de la modélisation des données et des diagrammes de classes. C'est elle qui permet d'anticiper et d'organiser la structure pour la programmation d'une application. Elle continue de se développer au fil de la création et devient de plus en plus précise à mesure que le projet se développe.

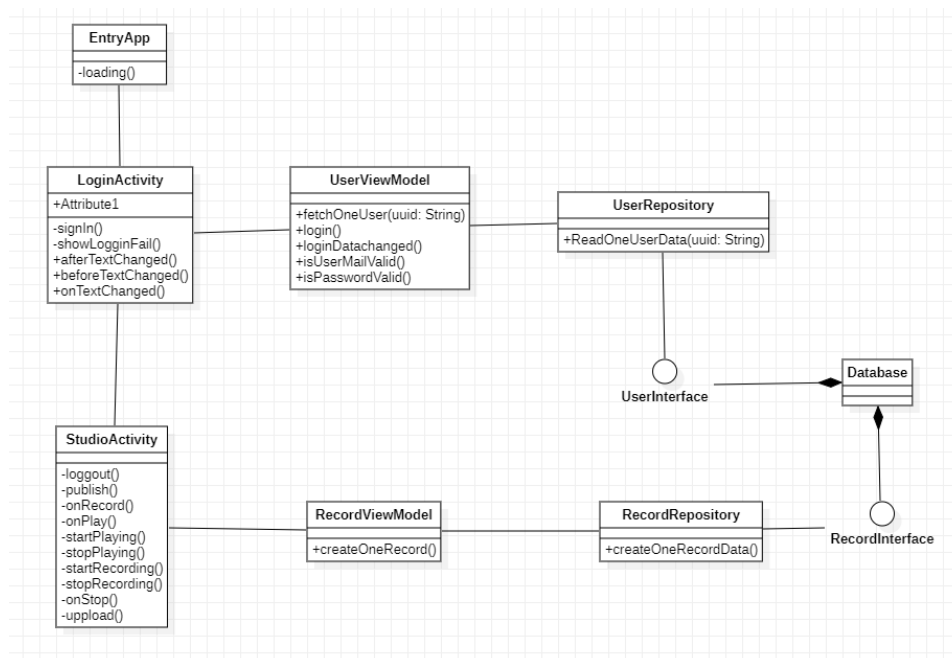
Vous pouvez retrouver les diagrammes en PDF dans le dossier Uml, nommés au nom de chaque application.

En voici également des captures d'écran :

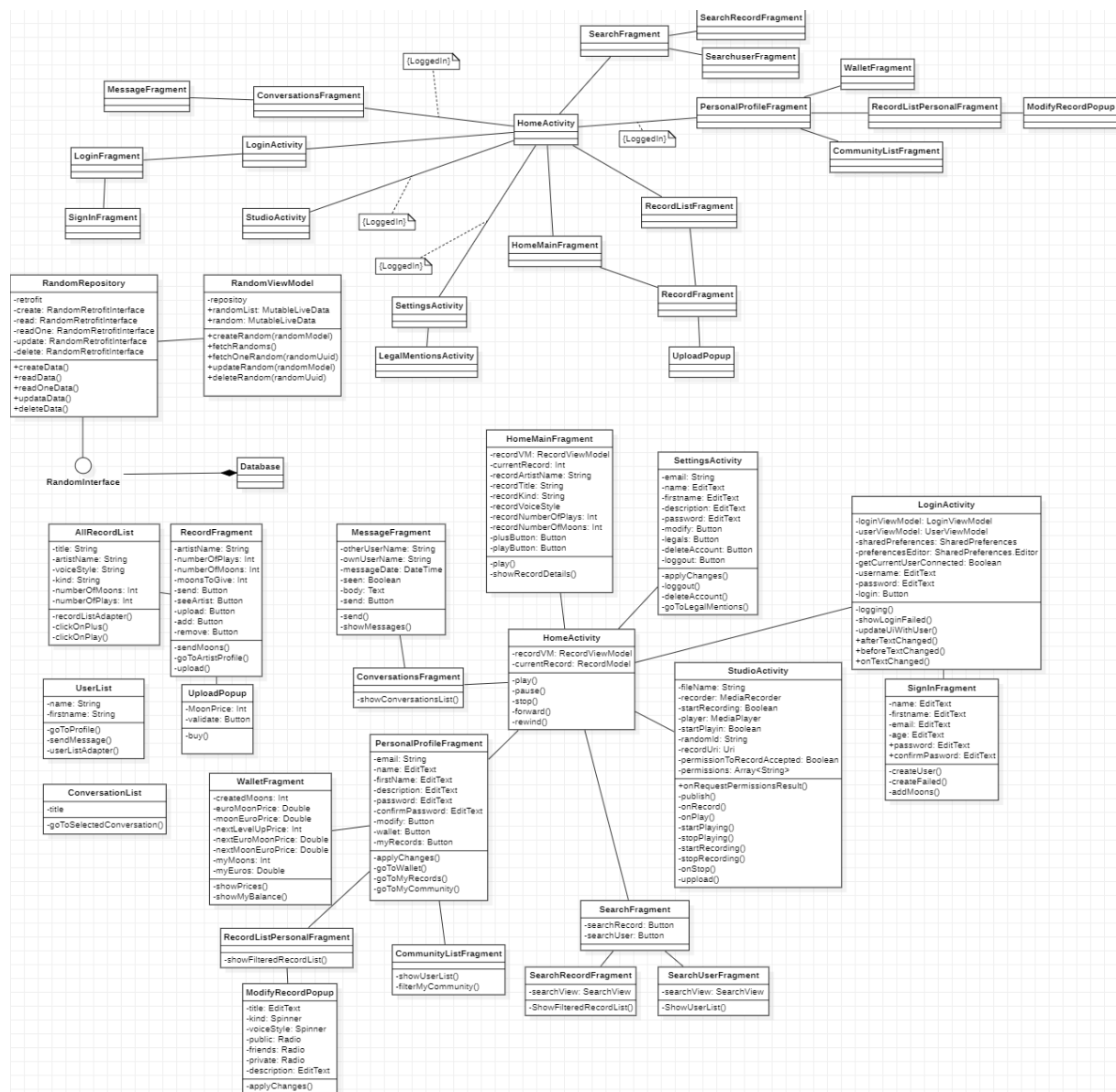
API :



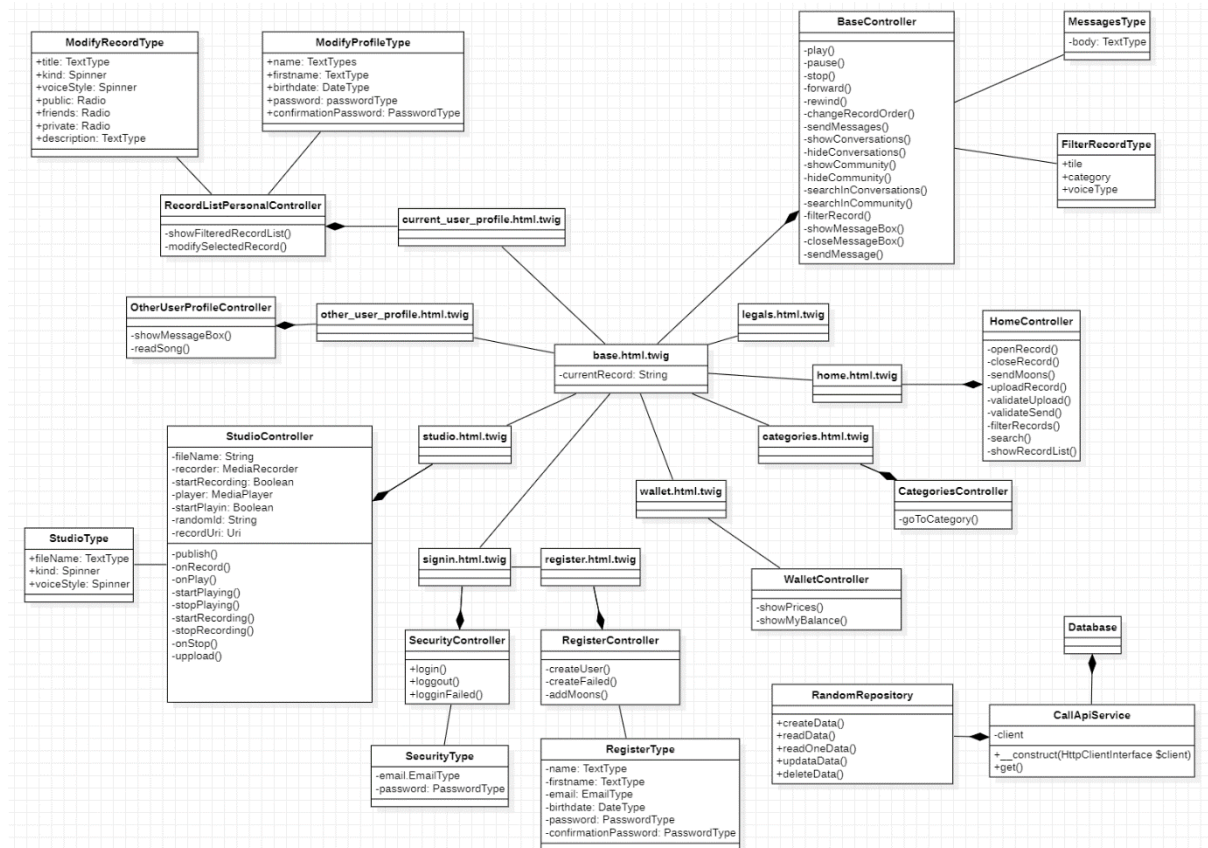
Application lourde :



Application mobile :



Application web :



Base de données

Choix de base de données

Le choix de la base de données doit déjà avoir été pensé en amont. Cependant, nous revenons dessus dans ce chapitre traitant de l'implantation de la base de données. Les bases de données sont des conteneurs d'informations stockés sur serveur. Elles désignent une collection d'informations organisées pour être facilement consultables, gérables et mises à jour. Elles peuvent être très structurées dans des tables, comme c'est le cas pour une base de données relationnelles manipulable via le langage SQL. Elles peuvent également être « non structurées » comme c'est le cas du noSQL. Celle-ci sont souvent organisées en JSON au format clé-valeur.

Chaque base de données peut avoir des avantages selon son utilisation. Certaines ont un prix plus élevé que d'autres. Certaines permettent une organisation plus lisible, d'autres permettent de stocker et d'accéder à des données plus volumineuses plus rapidement. On parle d'équilibre entre les trois vecteurs d'une base de données. Selon le théorème de Brewer, une base de données répartie sur plusieurs serveurs ne peut garantir simultanément cohérence, disponibilité et tolérance au partitionnement. Dans la théorie, chaque type de base de données ne peut posséder que deux de ces trois caractéristiques. C'est donc sur ce facteur qu'il faut s'appuyer lors du choix de la base.

Les bases de données relationnelles sont les plus répandues aujourd'hui. Elles restent simples à maintenir et à faire évoluer. De plus elles répondent aux besoins scolaires et utilisent le langage SQL, tout en proposant une rapidité d'exécution très satisfaisante. Elles permettent l'utilisation d'outils tels que MySQL avec phpMyAdmin, gratuits, pour les gérer. C'est donc une base de données relationnelle SQL qui sera utilisée pour ce projet.

Une fois le type de base de données à utiliser, confirmé, il est nécessaire de choisir le SGBD (Système de gestion de base de données). MySQL est un SGBD open source et gratuit. L'application web écrite en PHP : PHPMyAdmin, permet de gérer cette SGBD. Cet outil fonctionne parfaitement et est compatible avec la plupart des serveurs compatibles à PHP. C'est donc cet outil qui servira pour l'ensemble du projet.

MPD et Script SQL

Une fois le choix du type de base de données validé, le langage nécessaire pour sa manipulation en découle naturellement. C'est ici que commence l'étape du modèle physique de données (MPD). Elle consiste à transformer le résultat de la MLD en langage SQL.

La première étape consiste à créer la base de données via la ligne de commande suivante :

```
CREATE DATABASE IF NOT EXISTS `voovv` DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

La deuxième étape consiste à créer les différentes tables qui ont été mises en lumière lors de la création de l'architecture du projet. Celles-ci sont illustrées par le diagramme de base de données via l'outil dbdiagram.io.

La première table liste les utilisateurs :

```
CREATE TABLE `users` (  
  `uuid` char(36) NOT NULL PRIMARY KEY, /*Les clés primaires ne sont pas auto-  
incrémentées. Le choix d'avoir des uuid permet d'éviter les collisions lors de  
création de nouvelles lignes, mais ne permet pas l'auto-incrémentations en  
SQL*/  
  `name` varchar(256) NOT NULL,  
  `firstname` varchar(256) NOT NULL,  
  `email` varchar(256) NOT NULL,  
  `password` varchar(256) NOT NULL,  
  `phone` varchar(256) NOT NULL,  
  `description` text NOT NULL,  
  `number_of_followers` int(12) NOT NULL,  
  `number_of_moons` int(15) NOT NULL,  
  `number_of_friends` int(12) NOT NULL,  
  `url_profile_picture` varchar(256) NOT NULL,  
  `sign_in` datetime NOT NULL,  
  `last_connection` datetime NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

La deuxième table liste les enregistrements audios :

```
CREATE TABLE `audio_records` (  
  `uuid` char(36) NOT NULL PRIMARY KEY,  
  `artist_uuid` char(36) NOT NULL,  
  `title` varchar(256) NOT NULL,  
  `number_of_plays` int(12) NOT NULL,  
  `number_of_moons` int(15) NOT NULL,  
  `voice_style` varchar(256) NOT NULL,  
  `kind` varchar(256) NOT NULL,  
  `description` text NOT NULL,  
  `created_at` datetime NOT NULL,  
  `updated_at` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  FOREIGN KEY (artist_uuid) REFERENCES users(uuid) ON DELETE CASCADE ON  
UPDATE CASCADE,  
  FOREIGN KEY (kind) REFERENCES categories(name)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

La troisième table liste les messages :

```
CREATE TABLE `messages` (  
  `uuid` char(36) NOT NULL PRIMARY KEY,  
  `sender` char(36) NOT NULL,  
  `receiver` char(36) NOT NULL,  
  `body` text NOT NULL,  
  `seen` boolean NOT NULL,  
  `send_at` datetime NOT NULL,
```

```

    FOREIGN KEY (sender) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (receiver) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

La quatrième table créée liste les conversations :

```

CREATE TABLE `conversations` (
  `uuid` char(36) NOT NULL PRIMARY KEY,
  `sender` char(36) NOT NULL,
  `receiver` char(36) NOT NULL,
  `title` text NOT NULL,
  `created_at` datetime NOT NULL,
  `updated_at` datetime NOT NULL,
  FOREIGN KEY (sender) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (receiver) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

La cinquième table créée récupère les liens d'amitié :

```

CREATE TABLE `friends` (
  `uuid` char(36) NOT NULL PRIMARY KEY,
  `user1` char(36) NOT NULL,
  `user2` char(36) NOT NULL,
  FOREIGN KEY (user1) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (user2) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

La sixième table liste les catégories des enregistrements. Cette table n'est pas vouée aux utilisateurs. Elle comporte un nombre de lignes plus restreint :

```

CREATE TABLE `categories` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` char(36) NOT NULL,
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

Afin de sécuriser l'entrée des données dans la base, nous appliquons des clés étrangères. De cette manière, il est impossible de créer de nouveau lien d'amitié si l'uuid d'un des utilisateurs n'existe pas. De plus la ligne sera supprimée si l'un des utilisateurs de ce lien venait lui-même à l'être.

La septième table créée liste les images de profil utilisateur :

```
CREATE TABLE `pictures` (  
  `uuid` char(36) NOT NULL PRIMARY KEY,  
  `user_uuid` char(36) NOT NULL,  
  `url` varchar(255) NOT NULL,  
  `format` char(5) NOT NULL,  
  `size` int NOT NULL,  
  `uploaded_at` datetime NOT NULL,  
  `validated` boolean NOT NULL,  
  FOREIGN KEY (user_uuid) REFERENCES users(uuid) ON DELETE CASCADE ON UPDATE  
  CASCADE,  
  FOREIGN KEY (url) REFERENCES users(url_profile_picture) ON DELETE CASCADE  
  ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Implémentation des services

Le site web, la base de données et l'API seront stockés sur un serveur VPS et j'utiliserai l'outil « cyber panel » pour la gestion du serveur.

Cependant, pour tout projet, il est préférable de commencer par des tests de mise en place sur un environnement local. Pour se faire, le logiciel XAMPP, permet l'hébergement d'un serveur apache virtuel et l'accès au SGBD MySQL via PHPMyAdmin. Chaque étape testée en locale sera détaillée au moment de sa réalisation.

Création du VPS

La création d'un VPS (Virtual Private Server) auprès d'un hébergeur consiste à partitionner un serveur physique en plusieurs serveurs virtuels indépendants. De cette manière, il est possible de dédier un espace de mémoire définit, ainsi qu'une RAM, une bande passante et un espace de stockage.

Lors de la réalisation de mes précédents projets j'avais opté pour la solution plus simple d'un serveur partagé. Cette fois, c'est l'occasion de gagner en compétences sur un déploiement plus conséquent et complet. De plus, cela apporte une meilleure maîtrise sur les capacités d'utilisation, puisqu'elles peuvent être augmentées avec la demande.

Le serveur utilisé pour ce projet est hébergé par Hostinger. Il tourne sous Linux et se situe au Royaume-Uni. Hostinger met à disposition un outil de gestion de serveur appelé « cyber panel ». Cet outil permet de simplifier la gestion des tâches en proposant une interface semblable à une interface proposée par un OS.

Déploiement et hébergement des différents services

Concernant la gestion de la base de données, Cyber panel donne accès automatiquement à PHPMyAdmin. L'utilisation est donc la même que dans un environnement local. Nous y reviendrons dans la partie qui y est consacrée un peu plus bas.

Le déploiement de site internet demande quelques manœuvres supplémentaires. En effet, il est assez simple de créer un nouveau site de test par exemple. En revanche, pour créer un site web pourvu d'un nom de domaine et validé par un certificat SSL, les étapes sont plus complexes. Pour le moment, nous pouvons créer des sites accessibles depuis internet via l'IP du serveur, suivi du chemin d'accès vers les fichiers.

Voici les étapes à remplir dans cyber panel dans un premier temps :

WEBSITE DETAILS

Select Package Default

Select Owner admin

Test Domain ☐ ⓘ

Domain Name voovv-api.fr

Email nathan.kedinger@gmail.com

Select PHP PHP 8.1

Additional Features

- ☒ SSL
- ☒ DKIM Support
- ☒ open_basedir Protection
- ☒ Create Mail Domain

[Create Website](#)

Cette action aura pour effet de créer un site accessible depuis l'onglet list website. Voici à quoi ressemble sa page :

VOOOV-API.FR - [PREVIEW](#)

All functions related to a particular site.

RESOURCE USAGE

Resource	Usage	Allowed
FTP	0	1000
Databases	1	1000
Disk Usage	2 (MB)	0 (MB)
Bandwidth Usage	0 (MB)	0 (MB)



[STRESS TEST](#) [SET UP SSH/SFTP ACCESS](#) [CLONE/STAGING](#) [MANAGE GIT](#)

VOOOV-API.FR HAS SELF-SIGNED SSL
Your SSL will expire in 3635 days.

Disk Usage

Bandwidth Usage

LOGS

 Access Logs  Error Logs

DOMAINS

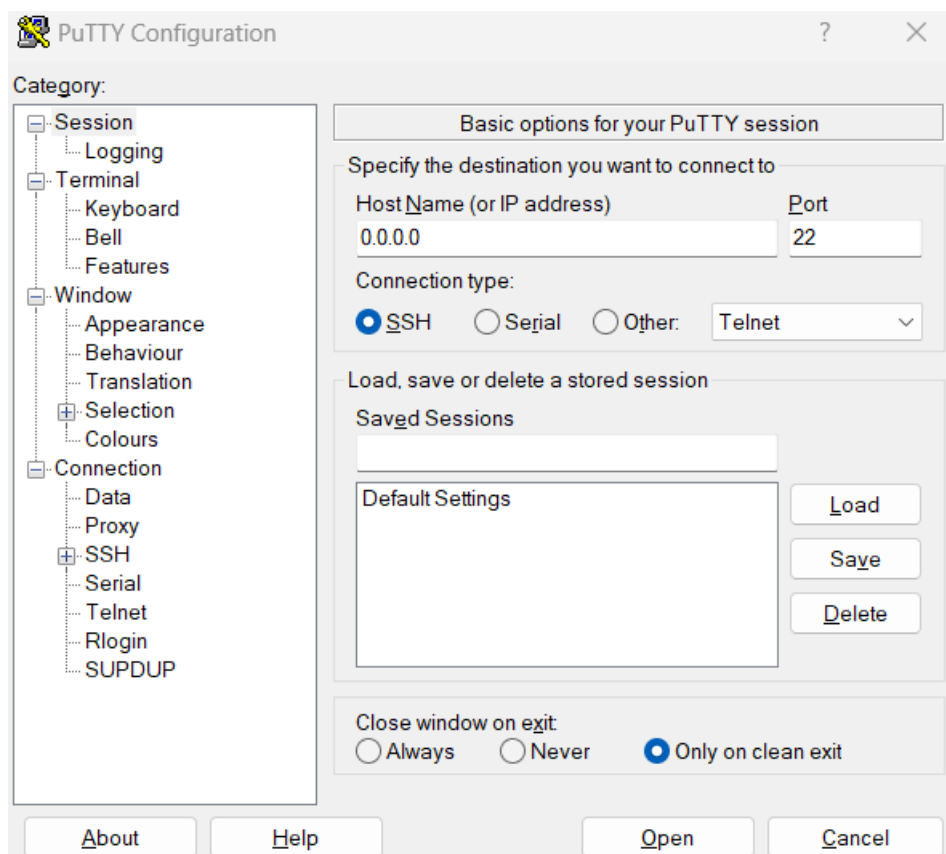
Le problème survenant à la suite de cette action est un certificat SSL auto-signé. Cela pose des problèmes à plusieurs niveaux. Les sites, ne possédant pas de certificats SSL sécurisés, sont répertoriés

par les navigateurs web, comme non sécurisés. Cela affiche une page d'erreur au moment de l'accès à la page.

Un autre problème que cela soulève est qu'Android bloque systématiquement l'accès à des sites non sécurisés, depuis ses applications. Cela nous empêche d'accéder à l'API depuis l'application mobile.

Bien que l'interface Cyberpanel permette un grand nombre d'actions, il reste nécessaire de régler quelques détails directement en ligne de commande. Notamment en ce qui concerne la sécurité. Mais également pour ce détail-ci, la validation du certificat SSL, par exemple. Pour pouvoir y parvenir, il est nécessaire de se connecter au serveur en tant que client. C'est possible grâce à un émulateur de terminal doublé d'un client pour les protocoles SSH. PuTTY convient ici parfaitement.

Afin de se connecter au serveur via le protocole SSH, nous devons entrer l'IP du serveur et son port d'accès :



Un fois la connexion au service effectué, nous devons valider les informations de connexion.

Une fois ceci fait, nous sommes transportés sur ce Terminal de commande :

```
root@vov-server:~  
*****  
Welcome to LiteSpeed One-Click CyberPanel Server.  
To keep this server secure, the firewalld is enabled.  
CyberPanel One-Click Quickstart guide:  
* https://docs.litespeedtech.com/cloud/images/cyberpanel/  
  
In a web browser, you can view:  
* CyberPanel: https://  
* phpMyAdmin: https:// /dataBases/phpMyAdmin  
* Rainloop: https:// /rainloop  
  
On the server:  
* You can get the CyberPanel admin password with the following command:  
  sudo cat .litespeed_password  
* You can get the Mysql cyberpanel user password with the following command:  
  sudo cat .db_password  
  
System Status:  
  Load : 0.00, 0.01, 0.05  
  CPU   : 0.491778%  
  RAM   : 327/2048MB (15.97%)  
  Disk  : 6/40GB (18%)  
  
Your CyberPanel is up to date  
*****  
[root@vov-server ~]#
```

Avant de pouvoir appliquer le certificat à nos sites web, Il est nécessaire de gérer la partie DNS qui n'est pas automatiquement activée. Afin de pouvoir récupérer un certificat SSL fournit par l'hébergeur, les étapes à suivre sont donc les suivantes :

- Acheter un nom de domaine auprès de l'hébergeur. Pour ce projet, ce sera vooov.fr qui sera utilisé. L'API sera également stockée sous ce nom de domaine. Le serveur est de toute façon le même. Cela réduit les coûts de les mutualiser.
- Mettre à jour les enregistrement DNS et de supprimer celui attribué par défaut par l'hébergeur, pour le remplacer par celui pointant vers l'IP du serveur VPS.
- Mettre à jour le serveur : `sh <(curl https://raw.githubusercontent.com/usmannasir/cyberpanel/stable/preUpgrade.sh || wget -O - https://get.acme.sh | sh`
- Imprimer un journal de log d'erreurs supplémentaires qui pourrait encore subsister : `cat /home/cyberpanel/error-logs.txt` (Pas d'erreurs retournées)
- Dernière étape à effectuer dans la console pour appliquer le certificat : `/root/.acme.sh/acme.sh --issue -d yourdomain.com -d`

- ```
www.yourdomain.com --cert-file
/etc/letsencrypt/live/www.rmronsol.com/cert.pem --key-file
/etc/letsencrypt/live/yourdomain.com/privkey.pem --fullchain-file
/etc/letsencrypt/live/yourdomain.com/fullchain.pem -w
/home/yourdomain.com/public_html --server letsencrypt --force --debug
```
- Retourner dans CyberPanel dans la section SSL et cliquer sur le bouton « issue SSL » du domaine concerné.

#### Problème rencontré

Impossible de débloquent le certificat SSL pour l'application ou le site web. Il fallait d'abord acheter un nom de domaine et le faire pointer vers le serveur.

#### Sécurisation du serveur

Le serveur est fonctionnel, le nom de domaine qui hébergera le site web ainsi que l'API est en ligne et disponible. En revanche, j'ai pu remarquer grâce à la connexion via PuTTY, que des centaines de tentatives de connexion au serveur avait lieu entre chacune de mes connexions. Mon accès root semblait être la cible parfaite pour des attaques par force brute depuis des bots lancés au hasard d'internet.

En premier lieu, j'ai doublé la longueur de mon mot de passe.

Dans un second temps, j'ai modifié mon port d'accès 22 qui est le port standard d'accès SSH. De cette façon, les bots automatisés réglés sur le port 22 n'y accèdent plus automatiquement. Pour se faire, Il faut se connecter au SSH via PuTTY puis insérer la ligne de commande suivante :

```
sudo nano /etc/ssh/sshd_config
#Entrer en bas de la liste, la ligne :
Port #le port choisi
#Puis par un redémarrage du serveur :
Service sshd restart
```

NB : Bien penser à modifier l'accès au firewall s'il y en a un. Dans le cas contraire, plus possible de se connecter.

#### Améliorations possibles

La sécurité d'accès au serveur peut encore être améliorée. Pour ce faire, il faudrait commencer par activer les firewalls qui contrôle les adresses IP de connexion. On pourrait ensuite changer le nom de l'utilisateur root par un autre possédant des droits limités selon utilisation. Il faudrait ensuite changer l'accès de connexion de l'utilisateur par clé rsa.

Le déploiement de l'application mobile se fera sur le serveur d'Android. En attendant, elle est déployée directement sur les appareils mobiles en les connectant au pc sur lequel est stockée l'application par un simple build dans l'application Android studio.

## Mise en place d'un service de versioning

### Utilité du versioning

Les services de versioning possèdent plusieurs objectifs :

- Sécuriser les sauvegardes. Ils permettent l'application de la règle 3-2-1 : Trois copies des données, une originale et deux copies, sur deux supports différents, dont une hors site (c'est ici qu'intervient le service de versioning).
- Garder une trace des modifications apportées au code. Il est possible d'accéder aux anciennes versions et d'effectuer un retour en arrière si besoin.
- Permettre une simplification des mises à jour de déploiement.
- Faciliter le travail en équipe en fournissant une version commune à l'ensemble des collaborateurs.
- Git permet le partage et la réutilisation de code entre différents partis (autres développeurs, étudiants, correcteurs d'épreuves de BTS...)

L'outil le plus utilisé pour le versioning est Git. Le versioning consiste à figer le code d'un projet à un instant donné. Pour ce faire, il faut inclure ce projet dans un répertoire (ou repository) Git. A chaque nouvel enregistrement du projet dans Git, cela crée une nouvelle version. Cette version enregistre uniquement les modifications ayant eu lieu depuis la version précédente.

Git a été développé par Linus Torvald, le créateur du noyau Linux. C'est un outil libre et disponible sur tous les systèmes d'exploitation.

### Utilisation de Git

Pour utiliser Git, il est d'abord nécessaire de l'installer. Le plus simple étant généralement d'installer un plugin dans l'IDE utilisé. Aujourd'hui, la plupart en fournissent.

Les étapes de dépôt d'une nouvelle version sur Git consiste à écrire en ligne de commande :

```
git init # On initialise git
git add . # Cette commande ajoute tous les fichiers. C'est pourquoi, si l'on
souhaite empêcher certains fichiers d'être ajoutés au repository il est
judicieux d'ajouter un fichier .gitignore qui contiendra la liste de tous les
fichiers à ne pas transmettre dans un commit. Par ailleurs, il est possible
d'ajouter les fichiers un par un. Pour se faire, remplacer le point par le nom
des fichiers à ajouter.
git commit -m "commentaire obligatoire qui comporte une liste rapide des
modifications apportées"
git push # Cette commande est la commande finale qui envoie la capture du code
à l'instant du commit.
```

Aujourd'hui, il existe principalement deux outils plus larges simplifiant l'usage de Git. Ils proposent grâce à celui-ci, d'autres services facilitant le travail du code, et des développeurs. Ce sont Github, proposé par Microsoft, et GitLab, un logiciel open source.

Chacun de ces outils possède ses qualités propres. Toutefois, la majeure différence réside dans le fait, que GitLab intègre des flux de travail d'intégration continue/ livraison continue et de DevOps. Ce sont des atouts majeurs pour des projets d'équipe.

GitLab semble particulièrement destiné aux équipes de développeurs. Github quant à lui semble suffisant pour un développeur seul. L'avantage de Github vient également de sa notoriété et du grand nombre d'utilisateurs en faisant usage. Les recruteurs sont plus actifs, à ma connaissance, sur Github que sur GitLab.

### Utilisation pour le projet

J'utilise Github depuis le début de ma formation. J'utiliserai Github pour le versioning de ce projet. Cependant, il est tout à fait possible de lier ces deux outils ensemble, et même de migrer de l'un à l'autre selon les besoins.

L'un des points forts du versioning avec Git, est qu'il est possible de lier les répertoires de fichiers des IDE vers le service Git choisi.

Dans l'autre sens, cela permet de lier le répertoire Git au répertoire de fichiers hébergé sur un serveur. De cette façon l'intégration des nouvelles versions sur serveur est très rapide et simple. Nul besoin de copier-coller le code, au risque de produire des erreurs.

CyberPanel propose ce service. Pour se faire, il est nécessaire de se rendre dans la liste des sites web, sélectionner le site puis cliquer sur l'onglet « manage Git ». Choisir, soit d'utiliser un repo existant, ou d'en créer un. Dans tous les cas, il est nécessaire de relier le dépôt à Github par la suite, à l'aide d'une clé de déploiement, à communiquer au site Github. Une fois les vérifications effectuées, il est possible d'importer le dépôt présent sur Github d'un simple clic, sur l'onglet « pull ».

Un détail à noter, toutefois, il est déconseillé de modifier les fichiers directement sur le serveur une fois un répertoire lié. Cela a pour effet de créer un conflit d'écrasement de données et bloque le pull.

Vous pouvez retrouver l'ensemble des réalisations de ces projets cette adresse :

<https://github.com/nathan-kedinger?tab=repositories>

Et notamment la liste des commits effectués pour l'API par exemple ici :

[https://github.com/nathan-kedinger/api\\_vooov/commits/main](https://github.com/nathan-kedinger/api_vooov/commits/main)

## Ticketing et tests

### Le ticketing

Le ticketing est mis en place via des outils de ticketing. On pense par exemple au très notoire GLPI. Un outil de ticketing sert principalement dans un projet passé en phase de production. Cependant, un relevé de logs d'erreur est très important pour un projet en phase de test, également en phase de conception.

En phase de production, il permet directement aux utilisateurs, de faire un retour au service client. Il permet de mémoriser l'ensemble des réclamations, problèmes, erreurs et incidents remontés par les utilisateurs. Il permet d'effectuer un archivage et un suivi de chaque problème. Il permet d'évaluer la priorité d'un ticket, son degré d'avancement ou encore l'état d'avancement du problème.

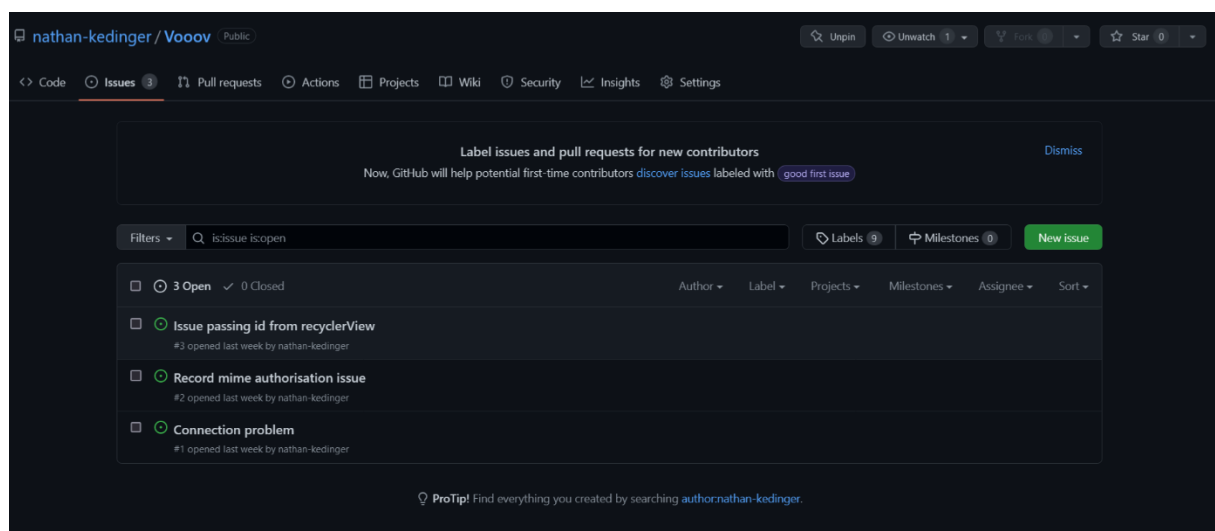
Un système de ticketing permet d'archiver de manière centralisée tous les problèmes rencontrés. De cette façon, l'équipe technique peut visualiser simplement l'ensemble des problèmes à résoudre.

### Phase de conception et de test

Lors de la phase de conception, un outil de ticketing destiné à des équipes de services clients n'a pas de sens.

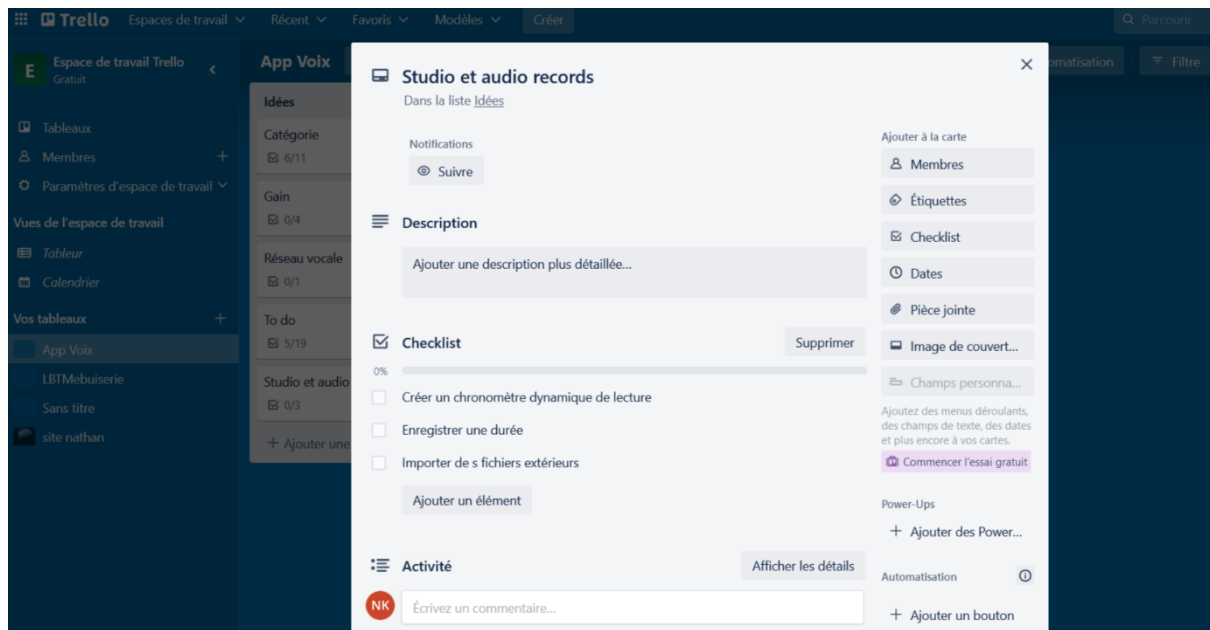
En revanche, il existe bien des outils permettant de remonter et lister les problèmes rencontrés au cours de cette phase. Le premier outil à utiliser par les développeurs découle de la partie précédente : Déclaration d'"Issues" sur le site github. En projet d'équipe, mais également seul, les issues permettent de garder un suivi des problèmes à résoudre.

Voici par exemple quelques issues postées pendant la réalisation de l'application mobile.



L'outil utilisé pour ce projet est également un outil collaboratif. Cet outil se nomme Trello. Il offre une visibilité sur la liste des problèmes rencontrés. Son avantage est son nombre d'utilisation possibles. Il peut également servir à stocker les tâches à effectuer, celles en cours, les idées... Il peut être partagé entre plusieurs acteurs et les tâches effectuées ne sont pas supprimées mais simplement rayées. Cela laisse donc une trace. Il est possible de retourner en arrière si nécessaire...

Voici un exemple de liste de To Do sur Trello :



Les autres outils utiles pour la gestion des erreurs sont directement implantés dans le serveur. Ce sont les fichiers de logs. Dans CyberPanel, deux fichiers de logs sont enregistrés. Le premier est un fichier de logs de connexion. Il répertorie l'ensemble des connexions et interactions ayant eu lieu sur le serveur. Le deuxième est un fichier de logs d'erreur, ayant eu lieu au cours de son utilisation.

Cependant, les erreurs relevées par ces logs restent limitées (principalement des erreurs internes au serveur). Afin de récupérer un maximum d'erreurs dans le code et son utilisation, il est intéressant d'implanter un maximum de « try catch » dans le code source des diverses applications.

Il est possible par la suite de lier ce fichier de logs à des outils ticketing de manière automatisée, en envoyant un ticket par mail à chaque erreur relevée. Cette approche fait sens, puisque les utilisateurs ne remonteront pas systématiquement une erreur survenue lors de l'utilisation de l'application. Ils auront plutôt tendance à stopper son utilisation.

Une gestion de logs automatisée est également un gros avantage lors de la phase de création et de test. Les « try catch » permettent de remonter des erreurs détaillées. Ils permettent également de cibler la recherche d'une erreur particulière lors d'une tentative de débogage du code.

## Phase de production

Lors de la phase de production, l'ensemble des outils de relevé d'erreurs mis en place pour la phase de création et de test reste en place. (Pour les try-catch et autres relevés d'erreurs uniquement. Dans un projet Symfony, il est important de passer le projet en mode production. Cela permet de cacher les erreurs aux utilisateurs). Un formulaire de contact est ajouté dans chaque application, afin de permettre la remontée des erreurs ou des réclamations par les utilisateurs. Ces retours d'erreurs sont gérés par l'outil trac, un logiciel libre et open source.

// En phase de production, le code passera sur GitLab afin de permettre l'intégration continue.

## Déploiement de la base de données

Le déploiement de la base de données est la première implémentation concrète du projet. Celle-ci stockera l'ensemble des données des différentes applications et les mettra en lien. C'est donc nécessairement le point d'entrée logique.

Dans un premier temps, bien que ce ne soit pas obligatoire, il est judicieux de tester l'implantation de la base de données dans un environnement local. Pour ce faire, nous utilisons l'outil XAMPP présenté précédemment. L'outil PHPMyAdmin qu'il propose permet d'entrer des requêtes SQL en direct.

Une fois le script précédemment construit intégré à la base, et fonctionnel, nous testons l'implantation de données dans la base. Les contraintes ajoutées dans les requêtes de créations de table nous obligent à suivre un certain ordre. La première donnée à entrer doit forcément être un utilisateur. L'ensemble des autres tables (excepté la table categories) ont besoin d'un UUID utilisateur pour créer des données.

Voici le script d'ajout d'un utilisateur :

```
INSERT INTO `users` (`uuid`, `name`, `firstname`, `email`, `password`,
`phone`, `description`, `number_of_followers`, `number_of_friends`,
`number_of_moons`, `url_profile_picture`, `sign_in`, `last_connection`) VALUES
('1', 'John', 'Doe', 'hasard@gm.com', 'password', '0606060606', 'a young
person working in IT', 0, 0, 0, 'randomUrl', '2019-08-30 15:34:33', '2021-08-
30 15:36:33')
```

## Procédures stockées

L'utilisation de procédures stockées consiste à créer des fonctions qui viendront être appliquées directement en base de données. Seul l'appel d'exécution de la procédure est envoyé. Cela permet de traiter le code en un seul lot. De cette manière, on réduit considérablement le trafic réseau entre le serveur et le client.

Dans cette application, certaines fonctionnalités devront être appelées plus régulièrement que d'autres. Par exemple, la lecture d'un utilisateur ou d'un ou plusieurs enregistrements audios. Pour cette raison, des procédures stockées seront appliquées à ces requêtes directement sur serveur.

Voici comment en présenter une pour retrouver un utilisateur par son nom :

```
DELIMITER $$
CREATE PROCEDURE find_user_by_name
(IN user_name CHAR(256))
BEGIN
 SELECT * FROM users WHERE name = user_name;
END $$
DELIMITER ;
```

Il ne reste plus qu'à implanter l'appel de cette procédure depuis l'API :

```
"CALL find_user_by_id(" . $userName . ")";
```



## Conception, création et implémentation de l'API

### Conception

Afin de concevoir une API (Application Programming Interface), il est nécessaire d'identifier les besoins auxquels celle-ci doit répondre.

Pour quelle raison utiliser une API ? Dans un projet tel que celui-ci, il est essentiel de pouvoir accéder à la base de données depuis les différents supports utilisés. Une Api permet d'encapsuler le code communiquant avec la base de données. Dans un projet reparté sur plusieurs serveurs différents, c'est la méthode la plus simple et efficace pour permettre la communication des différents acteurs.

Un standard a été défini concernant l'architecture des API. C'est le standard REST, l'abréviation du terme anglais « Representational State Transfer ». Il résulte d'un style d'architecture créé en 2000 par Roy Fielding. Bien qu'il en existe encore d'autres, cohérent pour certaines utilisations, c'est le standard le plus répandu aujourd'hui. Pour qu'une API soit dite REST, elle doit répondre à certaines contraintes. A savoir :

- L'architecture client-serveur : Les serveurs et les clients ne doivent connaître que les URL des ressources, sans dépendances.
- L'absence d'état : Le serveur ne doit pas faire de relations entre les différents appels des clients, ni même d'un même client. Il n'a pas connaissance de l'état du client entre ces transactions.
- La mise en cache des ressources : Le client doit être capable de garder les données en cache pour optimiser les transactions.
- Une interface uniforme : Tous composant qui comprend le protocole HTTP doit pouvoir communiquer avec l'API.
- Un système de couches : Permet d'organiser les différents types de serveurs. Cela permet une répartition des charges et de rendre l'application plus flexible.
- Le code à la demande : l'API peut envoyer du code exécutable au client.

Cette API tâchera donc de respecter le mieux possible ce standard.

Le prochain point consiste à déterminer les données à gérer pour notre application. Nous pouvons ici nous servir du travail réalisé dans les différents diagrammes présentés plus haut.

### Créations de l'API

L'API est codée en PHP via Visual Studio Code avec un suivi du versioning sur GitHub, visible à cette adresse : [https://github.com/nathan-kedinger/api\\_vooov](https://github.com/nathan-kedinger/api_vooov). Vous trouverez à suivre les différentes étapes suivies lors de la réalisation de cette API.

Dans un objectif de test, l'API est d'abord déployée en local, grâce à l'outil XAMPP. J'entame mes tests sur la table « users ».

La première étape consiste à créer un fichier permettant la connexion à la base de données. Créé un fichier spécifique à la connexion permet à la fois d'améliorer la portabilité et d'éviter la redondance de code à chaque appel de la base. Cela sécurise également les identifiants de connexion. En effet, cela

permet de stocker ce fichier à la racine du serveur (en amont du dossier « public\_html »), et donc d'éviter qu'il soit accessible à des utilisateurs externes.

Ce fichier comporte une classe « Database » dans laquelle sont établis des identifiants de connexion. Il comporte également une méthode getConnection() qui permet la connexion via un objet PDO. L'objet PDO permet la sécurisation des données importées dans la base.

La méthode getConnection() peut ensuite être appelée depuis le reste de l'API.

Voici le contenu du fichier « Database » pour une utilisation locale :

```
<?php

class Database{
 private $host = "localhost";
 private $db_name = "api_vooov";
 private $username = "root";
 private $password = "";
 public $connection;

 //getter for connection
 public function getConnection(){
 //closing the connection if exists
 $this->connection = null;

 // try to connect
 try{
 $this->connection = new PDO("mysql:host=" . $this->host . ";
dbname=" . $this->db_name, $this->username, $this->password);
 $this->connection->exec("set names utf8"); // force transaction
in <UTF-8></UTF-8>
 }catch(PDOException $exception){
 echo "Erreur de connexion : " . $exception->getMessage();
 }

 return $this->connection;
 }
}
```

L'étape suivante consiste à créer un fichier CRUD (Create, Read, Update, Delete). Dans un premier temps, les CRUD que j'ai réalisé étaient spécifique à une table. Ils comprenaient chacun 5 méthodes, à savoir :

```
create()
read()
readOne()
update()
delete()
```

Vous pouvez retrouver le fichier précédent ici : [https://github.com/nathan-kedinger/api\\_vooov/blob/main/models/UsersExampleCrud.php](https://github.com/nathan-kedinger/api_vooov/blob/main/models/UsersExampleCrud.php)

L'exemple de la méthode utilisée à l'origine est stocké dans le dossier « exemple ». **Il a été modifié par la suite dans un but d'amélioration. Ceci sera expliqué un peu plus bas.**

Une fois le fichier CRUD créé, il est nécessaire de créer des fichiers permettant l'interaction entre l'API et la base de données. Pour qu'une API soit REST, chaque méthode de requête est appelée par une unique méthode (GET, POST, PUT, DELETE...). En ce sens, il est nécessaire de créer un fichier spécifique pour chaque méthode.

Ces fichiers, permettant l'appel du protocole http, doivent posséder des entêtes (headers). Voici la liste des entêtes utilisées ici (en l'occurrence avec la méthode POST) :

```
// Headers
// Access from any site or device
header("Access-Control-Allow-Origin: *");

// Data format
header("Content-Type: application/json; charset=UTF-8");

// Authorised method
header("Access-Control-Allow-Methods: POST");

// Request lifetime
header("Access-Control-Max-Age: 3600");

// Authorised headers
header("Access-Control-Allow-Headers: Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With");
```

Il est ensuite nécessaire de vérifier que la méthode utilisée lors de l'appel de l'API est correcte :

```
// Verification that used method is correct
if($_SERVER['REQUEST_METHOD'] == 'POST'){
```

Nous appelons les fichiers de connexion et de CRUD (Le fichier CRUD est ici Users). Nous instancions ensuite la Base de données, l'objet CRUD et nous récupérons les datas envoyées par l'utilisateur :

```
// DDB instantiation
$database = new Database();
$db = $database->getConnection();

// Users instantiation
$user = new Users($db);

// Get back sended informations
$datas = json_decode(file_get_contents("php://input"));
```

La partie suivante du fichier permet de récupérer les données transmises par l'utilisateur et de les renvoyer au serveur. Dans un premier temps, le code se présentait sous cette forme :

```
if(!empty($datas->name) && !empty($datas->firstname) && !empty($datas->email) && !empty($datas->phone) && !empty($datas->number_of_followers) && !empty($datas->number_of_moons) && !empty($datas->number_of_friends) && !empty($datas->url_profile_picture) && !empty($datas->description) && !empty($datas->sign_in) && !empty($datas->last_connection)){

 //here we receive datas, we hydrate our object
 $user->name = $datas->name;
 $user->firstname = $datas->firstname;
 $user->email = $datas->email;
 $user->phone = $datas->phone;
 $user->number_of_followers = $datas->number_of_followers;
 $user->number_of_moons = $datas->number_of_moons;
 $user->number_of_friends = $datas->number_of_friends;
 $user->url_profile_picture = $datas->url_profile_picture;
 $user->description = $datas->description;
 $user->sign_in = $datas->sign_in;
 $user->last_connection = $datas->last_connection;

 if($user->create()){
 // Here it worked => code 201
 http_response_code(201);
 echo json_encode(["message" => "The add have been done"]);
 }else{
 // Here it didn't worked => code 503
 http_response_code(503);
 echo json_encode(["message" => "The add haven't been done"]);
 }
}
}else{
 // We catch the error
 http_response_code(405);
 echo json_encode(["message" => "This method isn't authorised"]);
}
```

Les erreurs éventuelles étaient recueillies par un retour http en JSON.

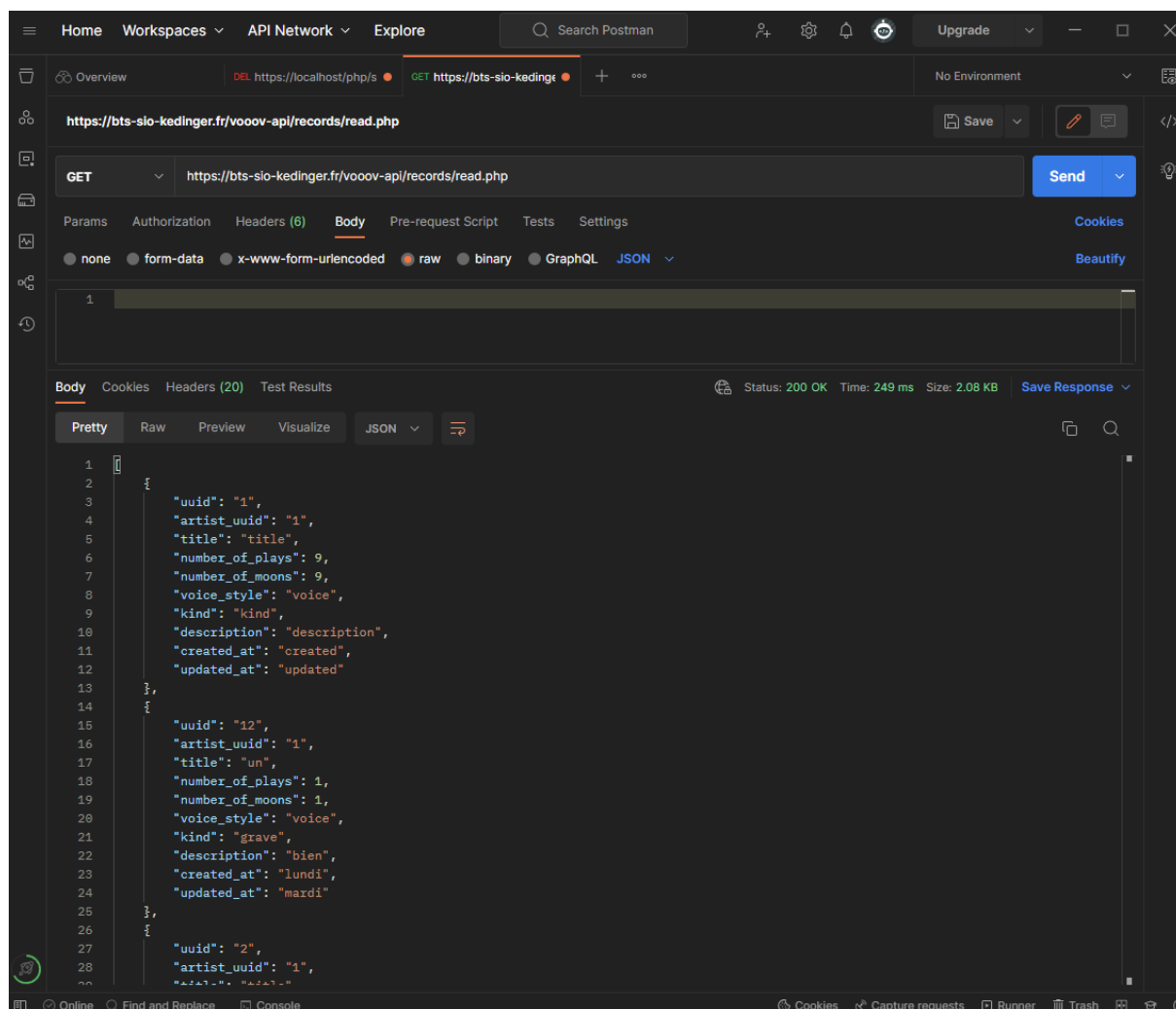
La récupération des erreurs nous permet de les situer plus précisément lors des tests.

Voici donc la méthode fonctionnelle REST pour transmettre des données à un serveur depuis une API. Nous transformons les données de l'utilisateur en format JSON. Le format JSON est un format très portable qui permet de récupérer les données sous forme d'objet.

Cette méthode POST est un exemple de code. Vous pouvez retrouver l'ensemble des méthodes sur le dépôt Github.

Afin de tester le bon fonctionnement de l'API j'utilise l'outil « Postman ». Celui-ci permet de tester l'envoi de requêtes et les réponses du serveur.

Exemple d'une requête GET sur la table « records » :



## Erreurs corrigées

Lors des premiers tests, je récupérais des erreurs sans code. J'avais omis la récupération d'une erreur en laissant un `if` sans `else`. J'ai également pu récupérer les erreurs de syntaxe SQL (qui ne sont pas relevées par l'IDE) grâce à des `try catch` et l'objet PDO, dans le fichier CRUD. Par la suite, j'utilise systématiquement les « `try catch` » qui sont renvoyés dans un fichier de logs sur le serveur.

Une fois chaque méthodes testée et fonctionnelle, on peut lancer le code en test sur serveur.

Sur serveur, il faut implémenter le code produit et penser à modifier les identifiants de connexion à la base. Mis à part ces étapes, le déploiement est relativement similaire à celui en local.

Pour faciliter les mises à jour des évolutions, j'ai lié mon dossier d'API sur serveur à mon repository Github. Cela me permet de travailler sur VS Code et de n'avoir qu'un pull à faire pour récupérer le code côté serveur.

## Amélioration du code

Une fois le CRUD de la table « users » implanté et testé sur serveur, j'ai commencé à écrire celui faisant référence à la table « records ». Au fil de l'adaptation des fichiers, deux points problématiques sont relevés. Le fait de devoir recopier le code dans son ensemble est chronophage et peut amener des erreurs. Le deuxième point apparaît au moment du test du code, à l'apparition d'une erreur dans le fichier Users.php. Elle doit également être corrigée dans le fichier Records.php. Cette API possède plusieurs tables, l'optimisation est donc nécessaire.

Pour optimiser le code, je réfléchis au code qu'il est possible de supprimer.

Tout d'abord, il semble important de n'avoir qu'un seul fichier CRUD. Pour ce faire, il est nécessaire de supprimer toutes les références à des variables définies et les rendre génériques. En ce sens, tous les appels de variables redondantes sont itérés en boucle. Elles sont ensuite appelées depuis le fichier propre de chaque méthode, à chaque table.

Nous créons un fichier « tabs » dans lequel seront passés les arguments pour chaque table, sous forme de tableau. Cela me permet de n'avoir qu'un seul fichier à modifier en cas de modifications dans une table de la base de données. Notamment pour l'ajout ou la suppression d'une colonne.

Exemple de la méthode create() dans le fichier CRUD:

```
/**
 * Creating
 *
 * @param array $arguments the columns to insert in the table
 * @param string $sql the sql query to prepare
 * @return boolean return true if the insertion is successfull, false
 otherwise
 */
public function create($arguments, $sql){

 try{
 // Request preparation
 $query = $this->connection->prepare($sql);

 // Protection from injections
 foreach($arguments as $argument){
 $this->$argument=htmlspecialchars(strip_tags($this-
>$argument));
 }

 // Adding protected datas
 foreach($arguments as $argument){
 $query->bindParam(":". $argument, $this->$argument);
 }

 // Request's execution
 $query->execute();
 // If there are no exceptions, return true
 return true;
 }
```

```

 } catch (PDOException $e) {
 // If there is an exception, print exception's message and return
false
 echo $e->getMessage();
 return false;
 }
 }
}

```

Une fois le fichier CRUD modifié, il est nécessaire de modifier les fichiers contenant les méthodes destinées aux appels http.

Pour améliorer encore la maintenabilité du code, ces fichiers sont séparés en deux parties. Une partie commune à toutes les tables est créée, nommée « generic ».

Toujours dans un souci de réduire le code à modifier (pour une meilleure portabilité) le tableau est appelé et une boucle est lancée sur celui-ci, lui passant ainsi les arguments.

De cette façon, les fichiers « generic » ne changent jamais. S'ils devaient être modifiés, ils ne le seraient qu'une seule fois. Si une erreur est relevée elle sera modifiée à un seul endroit.

Un autre fichier est ensuite créé pour chaque méthode et chaque table avec les informations de la table. On y passe la table visée, le tableau des titres de colonne et sa requête SQL.

Voici un exemple du fichier generic\_create.php :

```

try{
 // Verification that used method is correct
 if($_SERVER['REQUEST_METHOD'] != 'POST'){ // Change with good method
 throw new InvalidArgumentException("Invalid request method. Only POST
is allowed", 405);
 }

 // Including files for config and data access
 include_once '../../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // Records instantiation
 $crudObject = new CRUD($db);

 // Get input data
 $input = file_get_contents("php://input");
 if (!$input = json_decode($input)) {
 throw new InvalidArgumentException("Invalid input data. Must be valid
JSON", 405);
 }

 foreach($arguments as $argument){
 if(isset($datas->$argument)){
 //here we receive datas, we hydrate our object

```

```

 $crudObject->$argument = $datas->$argument;
 }else{
 // We catch the error
 http_response_code(400);
 echo json_encode(["message" => "Arguments doesn't match"]);
 }
}
if($crudObject->create($arguments, $sql)){
 // Here it worked => code 201
 http_response_code(201);
 echo json_encode(["message" => "The add have been done"]);
}else{
 // Here it didn't worked => code 503
 http_response_code(503);
 echo json_encode(["message" => "The add haven't been done"]);
}
} catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage());
}
}

```

Et voici le fichier create.php pour la table users :

```

<?php
 include_once '../tabs/tabs.php';

 $table = "friends"; // Change with the good BDD table name

 // Datas
 $arguments = $tabFriends; // Replace with the good tab

 // SQL request
 $sql = "INSERT INTO " . $table . " SET ". implode(', ',
array_map(function($argument)
 { return $argument . '=: ' . $argument; }, $arguments));

 include_once '../generic_cruds/generic_create.php';

```

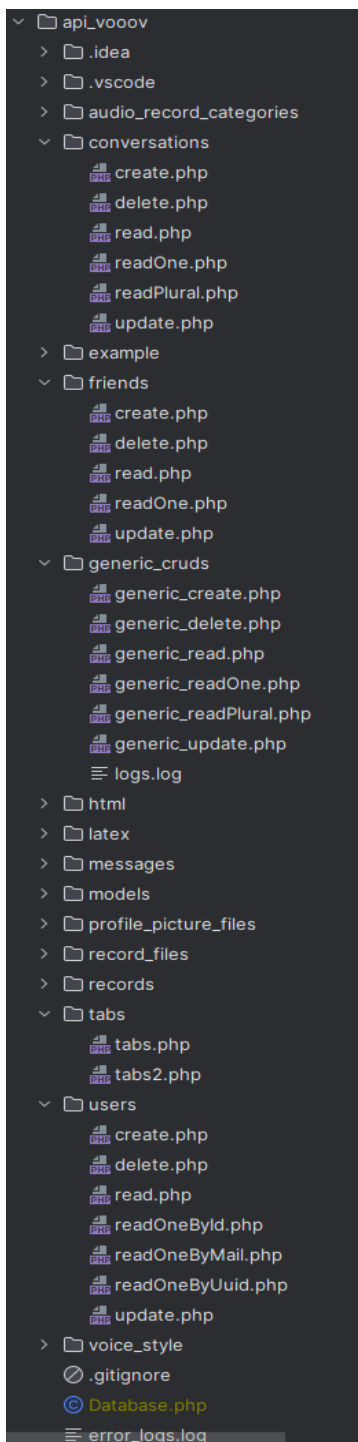
Vous pouvez retrouver le code complet de l'API à cette adresse : [https://github.com/nathan-kedinger/api\\_vooov](https://github.com/nathan-kedinger/api_vooov)



## Liste des fonctions et architecture de l'API

L'API finalisée comprend plusieurs types de fonctions. Celles-ci sont listées dans le diagramme un peu plus haut, mais il peut être intéressant de les lister à nouveau ici et d'expliquer clairement leur fonctionnement.

Le fonctionnement de cette API est basé sur trois couches qui réorientent l'appel d'API vers sa destination dans le but de récupérer les bonnes informations. (Je travaille actuellement à réduire en deux couches.) Premièrement, un dossier est créé par table de base de données. La première couche est le point d'appel. Ce sont les fichiers spécifiques qui sont demandés depuis l'extérieur à l'API. Ces fichiers sont contenus dans les dossiers du nom des tables. Voici l'architecture de l'API :



### Dossiers de tables

Comme nous pouvons le voir ici, il existe un dossier par table. Chacun de ces dossiers de table contient les méthodes utiles. Voici un exemple du code de ces méthodes pour la table conversations :

Fichier create.php :

```
<?php
 include_once '../tabs/tabs.php';

 $table = "conversations"; // Change with the
 good BDD table name

 // Datas
 $arguments = $tabConversations; // Replace
 with the good tab

 // SQL request
 $sql = "INSERT INTO " . $table . " SET ".
 implode(',', array_map(function($argument)
 { return $argument . '=: ' . $argument; },
 $arguments));

 include_once
 '../generic_cruds/generic_create.php';
```

Fichier delete.php :

```
<?php
 // Expected table
 $table = "conversations"; // Change with the
 good BDD table name

 $sql = "DELETE FROM " . $table . " WHERE uuid
 = ?";

 include_once
 '../generic_cruds/generic_delete.php';
```

Fichier read.php :

```
<?php
 include_once '../tabs/tabs.php';

 // Expected table
 $table = "conversations";

 // Datas
 $arguments = $tabConversationsRead; // Replace with the
good tab

 // SQL request
 $sql = "SELECT * FROM " . $table; // It is possible to add
a join after that

 include_once '../generic_cruds/generic_read.php';
```

Fichier readOne.php :

```
<?php
 include_once '../tabs/tabs.php';

 $table = "conversations"; // Change with the good BDD
table name

 $arguments = $tabConversationsRead; // Replace with the
good tab

 $sql = "SELECT ". implode(', ',
array_map(function($argument)
 { return $argument; }, $arguments)) . " FROM " . $table ."
WHERE sender = ? OR receiver = ? ";

 include_once '../generic_cruds/generic_readOne.php';
```

Fichier update.php :

```
<?php
 include_once '../tabs/tabs.php';

 $table = "conversations"; // Change with the good BDD
table name

 // Datas
 $arguments = $tabConversations; // Replace with the good
tab

 // SQL request
```

```

 $sql = "UPDATE " . $table . " SET ". implode(', ',
array_map(function($argument)
 { return $argument . '=: ' . $argument; }, $arguments)) . "
WHERE uuid=:uuid";

 include_once '../generic_cruds/generic_update.php';

```

#### Fichier tab

Le fichier tab regroupe les éléments à transmettre au fichiers points d'entrées pour les requêtes SQL. Il stocke les informations à renvoyer sous forme de tableau. Voici par exemple les tableaux de la table conversation, un tableau de lecture contenant l'ID et un tableau d'écriture :

```

$tabConversationsRead = [
 $id = "id",
 $sender_id = "sender_id",
 $receiver_id = "receiver_id",
 $uuid = "uuid",
 $title = "title",
 $created_at = "created_at",
 $updated_at = "updated_at",
];

$tabConversations = [
 $sender_id = "sender_id",
 $receiver_id = "receiver_id",
 $uuid = "uuid",
 $title = "title",
 $created_at = "created_at",
 $updated_at = "updated_at",
];

```

#### Les fichiers génériques

Ces fichiers sont la continuité des fichiers précédents mais les extraire permettait d'avoir un tronc commun et de séparer les logiques. Je travaille actuellement à insérer un tableau dans ces fonctions référençant la table visée. Cela permettrait d'avoir un tronc commun tout en supprimant l'ensemble des fichiers précédent. Cela permettra d'avoir une maintenabilité encore améliorée.

Les fichiers génériques sont les deuxièmes couches de l'API. Ils traitent la requête http, l'autorisent en fonction des headers. Ils transforment le JSON de la requête en caractères qu'ils renvoient à la troisième couche (Nous la verrons juste après). Ils récupèrent ensuite la réponse de la troisième couche qu'ils transforment de nouveau en JSON, et la renvoient au client. Voici les différents fichiers génériques. Ceux-ci contiennent des outils de récupération d'erreur qui sont envoyés dans les logs et au client.

Fichier generic\_create.php :

```

<?php
// Headers
use public\Database;

```

```

header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: POST");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try{
 /**
 * Script to handle a POST request
 *
 * @throws InvalidArgumentException if the request method
is not POST or if the input data is not valid JSON
 */
 // Verification that used method is correct
 if($_SERVER['REQUEST_METHOD'] != 'POST'){ // Change with
good method
 throw new InvalidArgumentException("Invalid request
method. Only POST is allowed", 405);
 }

 // Including files for config and data access
 include_once '../../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // Records instantiation
 $crudObject = new CRUD($db);

 // Get input data
 $input = file_get_contents("php://input");
 if (!$input = json_decode($input)) {
 throw new InvalidArgumentException("Invalid input
data. Must be valid JSON", 400);
 }

 foreach($arguments as $argument){
 if(isset($input->$argument)){
 //here we receive datas, we hydrate our object
 $crudObject->$argument = $input->$argument;
 }else{
 // We catch the error
 http_response_code(400);
 echo json_encode(["message" => "Arguments
doesn't match"]);
 }
 }

 if($crudObject->create($arguments, $sql)){
 // Here it worked => code 201
 http_response_code(201);
 }
}

```

```

 echo json_encode(["message" => "The add have been
done"]);
 }else{
 // Here it didn't worked => code 503
 http_response_code(503);
 echo json_encode(["message" => "The add haven't
been done"]);
 }

} catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage());
}

```

Fichier generic\_delete.php :

```

<?php
// Headers
use public\Database;

header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: DELETE");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try{
 /**
 * Script to handle a DELETE request
 *
 * @throws InvalidArgumentException if the request method
is not DELETE
 */
 // Verification that used method is correct
 if($_SERVER['REQUEST_METHOD'] != 'DELETE'){
 throw new Exception("Invalid request method. Only POST
is allowed", 405);
 }

 // Including files for config and data access
 include_once '../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // crudObjects instantiation
 $crudObject = new CRUD($db);

```

```

 // Get uuid from url
 $uuid = $_GET['uuid'];

 if(!empty($uuid)){

 $crudObject->uuid = $uuid;

 if($crudObject->delete($sql)){

 http_response_code(200);

 echo json_encode(["message" => "The data have
been deleted"]);

 }else{
 http_response_code(503);
 echo json_encode(["message" => "The data
haven't been deleted"]);
 }
 }else{
 // We catch the error
 http_response_code(403);
 echo json_encode(["message" => "Arguments doesn't
match"]);
 }
 } catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage());
 }
}

```

Fichier generic\_read.php :

```

<?php
// Headers
use public\Database;

header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: GET");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try{
 /**
 * Script to handle a GET request
 *
 * @throws InvalidArgumentException if the request method
is not GET

```

```

 */
 // Verification that used method is correct
 if(isset($_SERVER['REQUEST_METHOD']) &&
$_SERVER['REQUEST_METHOD'] != 'GET'){
 throw new Exception("Invalid request method. Only GET
is allowed", 405);
 }
 // Including files for config and data access
 include_once '../../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // crudObject instantiation
 $crudObject = new CRUD($db);

 // Get datas
 $stmt = $crudObject->read($sql);

 // Add logs to check the SQL query
 error_log("SQL query: " . $sql);

 // Verifying that we have at least one row in database
 if($stmt->rowCount() > 0){
 //initialisation of an associative tab
 $showedDatas = array();

 // $table = DB table
 $data = array();

 while($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
 extract($row);

 foreach($arguments as $argument){
 $data[$argument] = $row[$argument];
 }

 $showedDatas[] = $data;
 }
 http_response_code(200);

 echo json_encode($showedDatas);
 }else{
 // Add a log to check if there are no rows in the
table
 error_log("No rows found in the table.");
 http_response_code(400);
 echo json_encode(["message" => "There is no row in
that table"]);

```

```

 }

} catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage());
}

```

Fichier generic\_readOne.php :

```

<?php
// Headers
use public\Database;

header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: GET");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try{
 /**
 * Script to handle a GET request for one data
 *
 * @throws InvalidArgumentException if the request method
is not GET
 */
 // Verification that used method is correct
 if($_SERVER['REQUEST_METHOD'] != 'GET'){
 throw new Exception("Invalid request method. Only GET
is allowed", 405);
 }

 // Including files for config and data access
 include_once '../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // crudObject instantiation
 $crudObject = new CRUD($db);

 // Get uuid from url Remplacer par oneToGet
 $uuid = $_GET[$theOneToGet];

```



```

 // Verifying that we have at least one crudObject
 if (!empty($uuid)) {

 $crudObject->uuid = $uuid;

 $crudObject->readOne($arguments, $sql);

 $oneShowedData = [];
 foreach ($arguments as $argument){
 $oneShowedData[$argument] = $crudObject-
>$argument;
 }

 http_response_code(200);

 echo json_encode($oneShowedData);

 }else{
 http_response_code(404);
 echo json_encode(array("message" => "This ref
doesn't exists."));
 }

 } catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage(), 0, "logs.txt");
 }
}

```

Fichier generic\_readPlural.php :

```

<?php
 // Headers
 use public\Database;

 header("Access-Control-Allow-Origin: *");
 header("Content-Type: application/json; charset=UTF-8");
 header("Access-Control-Allow-Methods: GET");
 header("Access-Control-Max-Age: 3600");
 header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

 try{
 /**
 * Script to handle a GET request
 *
 * @throws InvalidArgumentException if the request
method is not GET
 */
 // Verification that used method is correct
 }

```

```

 if(isset($_SERVER['REQUEST_METHOD']) &&
$_SERVER['REQUEST_METHOD'] != 'GET'){
 throw new Exception("Invalid request method. Only
GET is allowed", 405);
 }
 // Including files for config and data access
 include_once '../../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // crudObject instantiation
 $crudObject = new CRUD($db);

 // Get datas
 $stmt = $crudObject->readPlural($sql,
[$_GET[$theOneToGet]]);

 // Verifying that we have at least one row in database
 if($stmt->rowCount() > 0){
 //initialisation of an associative tab
 $showedDatas = array();

 // $table = DB table
 $data = array();

 while($row = $stmt->fetch(PDO::FETCH_ASSOC)){
 extract($row);

 foreach($arguments as $argument){
 $data[$argument] = $row[$argument];
 }

 $showedDatas[] = $data;
 }
 http_response_code(200);

 echo json_encode($showedDatas);
 }else{
 http_response_code(400);
 echo json_encode(["message" => "There is no row in
that table"]);
 }
 } catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage());
 }
}

```

Fichier generic\_update.php :

```
<?php
// Headers
use public\Database;

header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: PUT");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try{
 /**
 * Script to handle a PUT request
 *
 * @throws InvalidArgumentException if the request method
 is not PUT or if the input data is not valid JSON
 */
 // Verification that used method is correct
 if($_SERVER['REQUEST_METHOD'] != 'PUT'){ // Change with
good method
 throw new Exception("Invalid request method. Only PUT
is allowed", 405);
 }

 // Including files for config and data access
 include_once '../Database.php';
 include_once '../models/CRUD.php';

 // DDB instantiation
 $database = new Database();
 $db = $database->getConnection();

 // Records instantiation
 $crudObject = new CRUD($db);

 // Get input data
 $input = file_get_contents("php://input");
 if (!$input = json_decode($input)) {
 throw new InvalidArgumentException("Invalid input
data. Must be valid JSON", 405);
 }

 $datas = $input;
 foreach($arguments as $argument){
 if(isset($datas->$argument)){
 //here we receive datas, we hydrate our object
 $crudObject->$argument = $datas->$argument;
 }else{
 // We catch the mistake

```

```

 http_response_code(400);
 echo json_encode(["message" => "Arguments
doesn't match"]);
 }
}
if($crudObject->update($arguments, $sql)){
 // Here it worked => code 201
 http_response_code(201);
 echo json_encode(["message" => "The change have
been done"]);
}else{
 // Here it didn't worked => code 503
 http_response_code(503);
 echo json_encode(["message" => "The change haven't
been done"]);
}
} catch (Exception $e){
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage());
}
}

```

### Couche model

La troisième couche est la couche modèle. Celle-ci s'occupe de l'échange direct avec la base de données et la récupération de données visées. Elle propose des méthodes qui sont appelées par les fichiers génériques selon la requête envoyée par le client. Ces méthodes récupèrent des données et les renvoient à la deuxième couche pour les traduire en JSON. Voici le fichier CRUD.php :

```

<?php
/**
 * CRUD class for handling database operations
 */
class CRUD{
 // Connection
 private $connection;

 // Columns
 public $uuid;

 /**
 * Constructor with $db for db connection
 *
 * @param $db
 */

 public function __construct($db)
 {
 $this->connection = $db;
 }
}

```

```

 /**
 * Creating
 *
 * @param array $arguments the columns to insert in the
table
 * @param string $sql the sql query to prepare
 * @return boolean return true if the insertion is
successfull, false otherwise
 */
 public function create($arguments, $sql){

 try{
 // Request preparation
 $query = $this->connection->prepare($sql);

 // Protection from injections
 foreach($arguments as $argument){
 $this-
>$argument=htmlspecialchars(strip_tags($this->$argument));
 }

 // Adding protected datas
 foreach($arguments as $argument){
 $query->bindParam(":". $argument, $this-
>$argument);
 }

 // Request's execution
 $query->execute();
 // If there are no exceptions, return true
 return true;
 } catch (PDOException $e) {
 // If there is an exception, print exception's
message and return false
 echo $e->getMessage();
 return false;
 }
 }

 /**
 * Reading
 *
 * @param string $sql the sql query to prepare
 * @return object return the query object
 */
 public function read($sql){

 // Request preparation
 $query = $this->connection->prepare($sql);

```

```

 $query->execute();

 //return the result
 return $query;
 }

 /**
 * Reading
 *
 * @param string $sql the sql query to prepare
 * @return object return the query object
 */
 public function readPlural($sql, $params = []){

 // Request preparation
 $query = $this->connection->prepare($sql);

 $query->execute($params);

 //return the result
 return $query;
 }

 /**
 * Reading one
 *
 * @param array $arguments the columns to select in the
table
 * @param string $sql the sql query to prepare
 * @return void
 */
 public function readOne($arguments, $sql){

 $query = $this->connection->prepare($sql);

 $query->bindParam(1, $this->uuid);

 $query->execute();

 $row = $query->fetch(PDO::FETCH_ASSOC);

 foreach($arguments as $argument){
 $this->$argument = $row[$argument];
 }
 }

 /**
 * Update
 *

```

```

 * @param array $arguments the columns to select in the
table
 * @param string $sql the sql query to prepare
 * @return boolean
 *
 */
public function update($arguments, $sql){

 try{
 // Request preparation
 $query = $this->connection->prepare($sql);

 // Protection from injections
 foreach($arguments as $argument){
 $this->
>$argument=htmlspecialchars(strip_tags($this->$argument));
 }

 // Adding protected datas
 foreach($arguments as $argument){
 $query->bindParam(":". $argument, $this->
>$argument);
 }

 // Request's execution
 $query->execute();
 // If there are no exceptions, return true
 return true;
 } catch (PDOException $e) {
 // If there is an exception, print exception's
message and return false
 echo $e->getMessage();
 return false;
 }
}

/**
 * Delete
 *
 * @param string $sql the sql query to prepare
 * @return boolean
 *
 */
public function delete($sql){

 $query = $this->connection->prepare($sql);

 $this->uuid=htmlspecialchars(strip_tags($this->uuid));

 $query->bindParam(1, $this->uuid);

```

```

 if($query->execute()){
 return true;
 }

 return false;
 }
}

```

### *Echanges de fichier image et audio*

L'API comprend également des méthodes permettant le téléchargement en amont et en aval de fichiers, dans le stockage du serveur. Elle permet le téléchargement en amont et en aval de fichiers au format image et au format audio ainsi que leur suppression. Elles sont stockées dans les dossiers « smtg\_files ». Ces dossiers contiennent également un dossier stockant les fichiers échangés. Voici les méthode pour le transfert de fichiers :

Fichier record\_download.php :

```

<?php

// Headers
header("Access-Control-Allow-Origin: *");
header("Content-Type: audio/mpeg");
header("Access-Control-Allow-Methods: GET");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try {
 // Vérification de la méthode HTTP utilisée
 if(!isset($_SERVER['REQUEST_METHOD']) ||
$_SERVER['REQUEST_METHOD'] != 'GET'){
 throw new Exception("Invalid request method. Only GET
is allowed", 405);
 }

 // Validation des paramètres de la requête
 if (!isset($_GET['file']) || empty($_GET['file'])) {
 throw new Exception("Invalid file name. Please provide
a file name", 400);
 }

 // Récupération du fichier demandé
 $target_dir = "records/";
 $file_name = $_GET['file'];
 $file_path = $target_dir . $file_name;

 // Vérification de l'existence du fichier
 if (!file_exists($file_path)) {
 throw new Exception("File not found", 404);
 }
}

```



```

 }

 // Vérification que le fichier est dans le dossier
 autorisé
 $real_path = realpath($file_path);
 $real_target_dir = realpath($target_dir);
 if (strpos($real_path, $real_target_dir) !== 0) {
 throw new Exception("Access denied. File is not within
allowed directory", 403);
 }

 // Envoi du fichier audio demandé
 header('Content-Length: ' . filesize($file_path));
 readfile($file_path);

} catch (Exception $e) {
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage(), 0, '../error_logs.txt');
}

```

Fichier record\_upload.php :

```

<?php
// Headers
header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: POST");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-
Control-Allow-Headers, Authorization, X-Requested-With");

try{
 /**
 * Script to handle a POST request
 *
 * @throws InvalidArgumentException if the request method
is not POST or if the input file is not valid JSON
 */
 // Verification that used method is correct
 if(isset($_SERVER['REQUEST_METHOD']) &&
$_SERVER['REQUEST_METHOD'] !== 'POST'){
 throw new InvalidArgumentException("Invalid request
method. Only POST is allowed", 405);
 }

 $target_dir = "records/";
 $target_file = $target_dir .
basename($_FILES["file"]["name"]);

```

```

 // Check if the file has been uploaded
 if (!isset($_FILES['file']['error']) ||
is_array($_FILES['file']['error'])) {
 throw new InvalidArgumentException("Invalid input
data. Must be valid file", 400);
 }

 switch ($_FILES['file']['error']) {
 case UPLOAD_ERR_OK:
 break;
 case UPLOAD_ERR_NO_FILE:
 throw new InvalidArgumentException("No file
sent", 400);
 case UPLOAD_ERR_INI_SIZE:
 case UPLOAD_ERR_FORM_SIZE:
 throw new InvalidArgumentException("Exceeded
filesize limit", 400);
 default:
 throw new InvalidArgumentException("Unknown
errors", 400);
 }

 /*// Check file mime type
 $file_mime =
mime_content_type($_FILES["file"]["tmp_name"]);
 if ($file_mime != 'audio/mp4' && $file_mime !=
'audio/mpeg' && $file_mime != 'audio/ogg' &&
 $file_mime != 'audio/wav' && $file_mime != 'audio/x-
flac' && $file_mime != 'audio/3gpp'){
 throw new InvalidArgumentException("Invalid input
file. Must be one of .mp4, .mp3, .ogg, .wav, .3gp, or .flac",
408);
 }*/

 // Check file size
 $file_size = $_FILES["file"]["size"];
 if($file_size > 3000000){
 throw new InvalidArgumentException("File is to
big. Max size is 3Mo", 400);
 }

 // Move uploaded file to target directory
 if (move_uploaded_file($_FILES["file"]["tmp_name"],
$target_file)) {
 echo json_encode(["message" => "File uploaded
successfully."]);
 } else {
 echo json_encode(["message" => "There was an error
uploading the file."]);
 }
 }

```

```

} catch (Exception $e) {
 http_response_code($e->getCode());
 echo json_encode(["Message" => $e->getMessage()]);
 error_log($e->getMessage(), 0, '../error_logs.log');
}

```

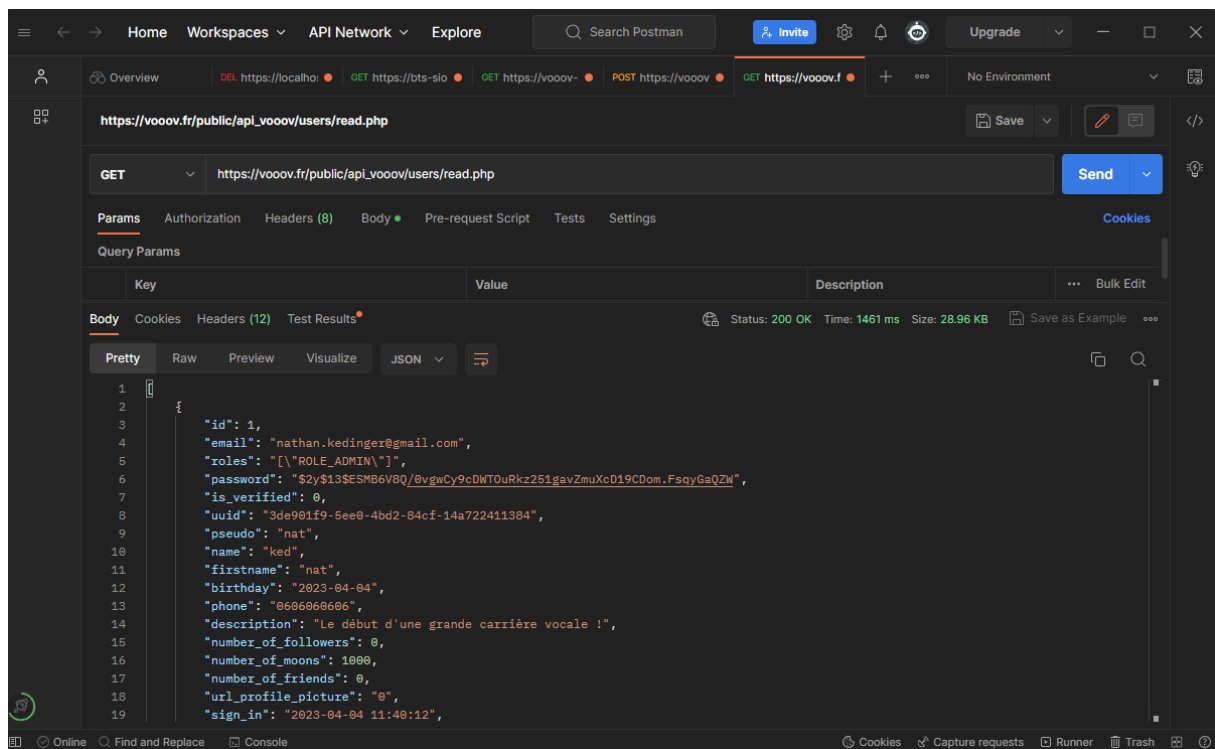
Le dossier API est maintenant stocké directement dans le dossier du site internet vooov.fr sur le serveur. Cela permet un partage de la base de données.

L'objectif à venir concernant cette partie est de regrouper les fichiers de la couche une et deux. L'objectif secondaire sera d'inclure les requêtes API dans le routeur de Symfony afin de stocker l'API dans le dossier source de l'application. Cela permettra de protéger l'API et également d'ajouter une couche d'identifiants par tokens d'accès à l'API.

## Tests

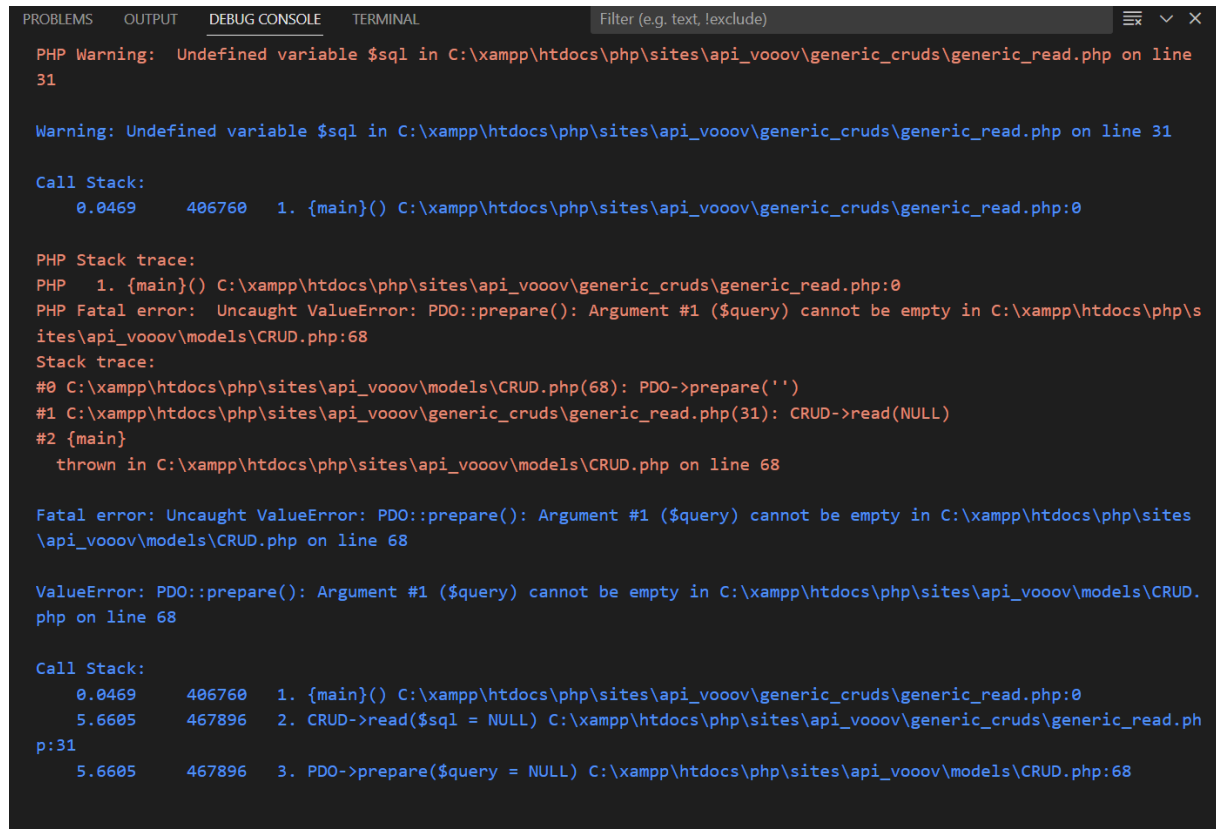
Afin de valider le fonctionnement de l'API, j'utilise deux outils de test. Postman pour les requêtes http et un outil de test unitaire.

Voici un exemple d'appel API testé sur Postman avec les fonctions précédentes. En l'occurrence, il s'agit d'un appel à la première couche de la table users appelant la méthode read. La requête est ensuite transmise à la deuxième couche qui transforme le JSON en caractères puis à la troisième qui récupère les données. Enfin la troisième couche renvoie les données à la deuxième couche. La deuxième couche transforme ces données en JSON et nous les renvoie. Nous récupérerons ici l'ensemble des utilisateurs contenus dans la table users, conformément aux attentes :



Concernant l'outil de test unitaire, c'est PHPUnit qui est utilisé, via Xdebug, dans l'IDE VSCode.

Voici un exemple de résultat de test unitaire comportant des erreurs :



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Filter (e.g. text, !exclude)

PHP Warning: Undefined variable $sql in C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php on line 31

Warning: Undefined variable $sql in C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php on line 31

Call Stack:
 0.0469 406760 1. {main}() C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php:0

PHP Stack trace:
PHP 1. {main}() C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php:0
PHP Fatal error: Uncaught ValueError: PDO::prepare(): Argument #1 ($query) cannot be empty in C:\xampp\htdocs\php\sites\api_vooov\models\CRUD.php:68
Stack trace:
#0 C:\xampp\htdocs\php\sites\api_vooov\models\CRUD.php(68): PDO->prepare('')
#1 C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php(31): CRUD->read(NULL)
#2 {main}
 thrown in C:\xampp\htdocs\php\sites\api_vooov\models\CRUD.php on line 68

Fatal error: Uncaught ValueError: PDO::prepare(): Argument #1 ($query) cannot be empty in C:\xampp\htdocs\php\sites\api_vooov\models\CRUD.php on line 68

ValueError: PDO::prepare(): Argument #1 ($query) cannot be empty in C:\xampp\htdocs\php\sites\api_vooov\models\CRUD.php on line 68

Call Stack:
 0.0469 406760 1. {main}() C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php:0
 5.6605 467896 2. CRUD->read($sql = NULL) C:\xampp\htdocs\php\sites\api_vooov\generic_cruds\generic_read.php:31
 5.6605 467896 3. PDO->prepare($query = NULL) C:\xampp\htdocs\php\sites\api_vooov\models\CRUD.php:68
```

Les erreurs sont causées car le test était effectué sur un script local donc sans passage de paramètres à la méthode. Lorsque Xdebug tourne et que Postman est exécuté avec les bons paramètres, les erreurs disparaissent.

Les tests unitaires permettent de récupérer des informations essentiels et détaillées en cas de problèmes au lancement du code.

Fin du premier dossier

## Début du second dossier

### Conception de l'application mobile (Android/Kotlin)

Une fois les bases du projets réalisées, la conception de l'application finalisée et la structure du backend créée, il est temps de passer à la partie utilisable par les utilisateurs. Voici donc les étapes détaillées de la conception de l'application Vooov.

Comme la plupart des projets d'application, qu'elle soit mobile, web ou lourde, il faut différencier deux parties bien distinctes : La partie frontend et la partie backend. Dans une application Android réalisée sur l'IDE Android Studio, on distingue ces deux parties assez clairement. Le dossier « java » contient la partie backend, alors que le dossier « res » contient l'ensemble des fichiers de la partie frontend. Quelques subtilités seront toutefois abordées concernant les activités.

#### Réalisation de la partie frontend

La partie frontend d'un projet Android est donc représentée par le dossier « res ». Ce dossier contient lui-même plusieurs sous-dossiers que nous détaillerons par la suite. La grande majorité des fichiers que contient le dossier res sont écrits en XML. Le XML est l'acronyme de eXtensible Markup Language. Ce langage permet de décrire des données à l'aide d'un document texte. Il est largement utilisé dans Android Studio. C'est notamment vrai dans la partie mise en page, un peu comme le ferait le HTML dans une application web.

Le nom des dossiers contenus dans le fichier res ont une importance. Ils permettent la cartographie des fichiers par Android Studio. Ils permettent à l'IDE de chercher les ressources aux bons endroits.

#### Le dossier « values »

Le dossier « values » comporte trois fichiers : colors.xml, dims.xml et strings.xml. Il comporte également un sous-dossier thèmes. Ce dossier pourrait en quelque sorte s'apparenter au fichier CSS d'un site web. La conception d'une application Android, notamment la partie frontend, comporte son lot de complexités. Il est nécessaire de gérer les paramètres de résolution d'écran, et celle-ci diffèrent beaucoup plus que pour des écrans d'ordinateur. Il faut également gérer les éventuels changements d'orientations. Le changement d'orientation d'un écran va avoir pour effet d'actualiser la page et supprimera donc les données chargées. Pour nous permettre d'organiser l'UI et remédier à ces divers problèmes, Android Studio nous propose donc quelques solutions.

#### *Fichier string.xml*

Une bonne pratique dans Android Studio, et l'une des premières actions à entreprendre lors du démarrage d'un projet, est d'implémenter le texte de l'application dans un fichier tiers : le fichier « strings.xml ». L'intérêt de cette pratique est pluriel. Lorsque le texte est implémenté en dur dans le code, il doit être écrit une deuxième fois en cas de changement d'orientation du matériel. D'une autre part, en cas de traduction de l'application dans une autre langue, il faudrait reprendre tous les textes

de l'application un par un pour les modifier. C'est pourquoi, le travail de création de la partie frontend de l'application a déjà commencé, avec la création de la maquette sur Figma (voir plus haut dans la partie : [application des tendances au projet](#)). C'est ici qu'une bonne réflexion en amont nous permet de gagner du temps. Nous pouvons entrer le texte de l'application en une seule fois, et nous n'aurons pas à revenir dans le fichier string.xml à chaque création de visuel.

Voici à quoi ressemble le fichier string.xml :

```
<resources>

 <string name="app_name">Vooov</string>

 <string name="base_number">0</string>

 <!-- CONNEXION PAGE -->
 <string name="connexion_page_title">@string/app_name</string>
 <string name="connexion_mail">Email</string>
 <string name="connexion_mail_text_input">Votre email</string>
 <string name="connexion_password">Mot de passe</string>
 <string name="connexion_password_text_input">passe</string>
 <string name="connexion_login">Connexion</string>
 <string name="connexion_continue_without_conexion">Continuer sans me connecter</string>
 <string name="connexion_dont_have_account">Vous n'avez pas de compte?</string>
 <string name="connexion_inscription_link">Inscrivez vous</string>

 <!-- REGISTER PAGE -->
 <string name="register_page_title">@string/app_name</string>
 <string name="register_page_name">Inscription</string>
 <string name="register_mail">Email</string>
 <string name="register_mail_text_input">Entrez votre email</string>
 <string name="register_password">Mot de passe</string>
 <string name="register_password_text_input">Choisissez un mot de passe</string>
 <string name="register_confirm_password">Confirmation du mot de passe</string>
 <string name="register_name">Nom</string>
 <string name="register_name_text_input">Entrez votre nom</string>
```

Ce fichier permet également de reprendre du texte déjà entré s'il devait être utilisé plusieurs fois.

```
<string name="connexion_page_title">@string/app_name</string>
```

Cette ligne fait appel au nom de l'application précisé plus haut. Par exemple, en cas de volonté de changer le nom de l'application, nous n'aurions plus qu'à le modifier à un seul endroit.

Par ailleurs, une bonne organisation de la nomenclature est souhaitable. Cela permet de s'y retrouver lors de la création des différentes pages de l'applications. Ici, on commence par le nom de la page, que l'on fait suivre par le nom de la variable souhaité. L'IDE nous facilitera le travail ensuite. Celui-ci propose les entrées disponibles lorsque nous créerons les différentes pages.

### Colors.xml

Le fichier « colors.xml » contient les couleurs à utiliser dans l'application. Ce fichier permet également de centraliser les variables souhaitées et d'améliorer la portabilité des applications.

Encore une fois, le travail réalisé lors du maquetage de l'application nous permet d'y voir plus clair.

Voici comment se présente ce fichier :

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
 <color name="backgrounds">#131C3D </color>
 <color name="buttons">#102B8A</color>
 <color name="blue_texts">#0A51CA</color>
 <color name="studio_background">#1A1512</color>
 <color name="music_load">#FFDF0D</color>
 <color name="genaral_texts">#F3F3F3</color>
 <color name="black">#000000</color>
 <color name="grey">#DADADA</color>
 <color name="white">#FFFFFF</color>
</resources>

```

### Dimens.xml

Sur le même principe, le fichier « dimens.xml » permet de stocker les dimensions, les marges, le padding...

Voici à quoi il ressemble :

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
 <dimen name="default_margin">20dp</dimen>
 <dimen name="default_padding">30dp</dimen>
 <dimen name="icons_padding">15dp</dimen>
 <dimen name="title_margin">120dp</dimen>
 <dimen name="icons_margin">3dp</dimen>
 <dimen name="text_margin">16dp</dimen>
 <!-- Default screen margins, per the Android Design guidelines. -->
 <dimen name="activity_horizontal_margin">16dp</dimen>
 <dimen name="activity_vertical_margin">16dp</dimen>
</resources>

```

### Themes

Le dossier « themes », quant à lui, permet une gestion plus globale de l'application. Il comprend deux fichiers, l'un pour le thème sombre et l'autre pour le thème clair.

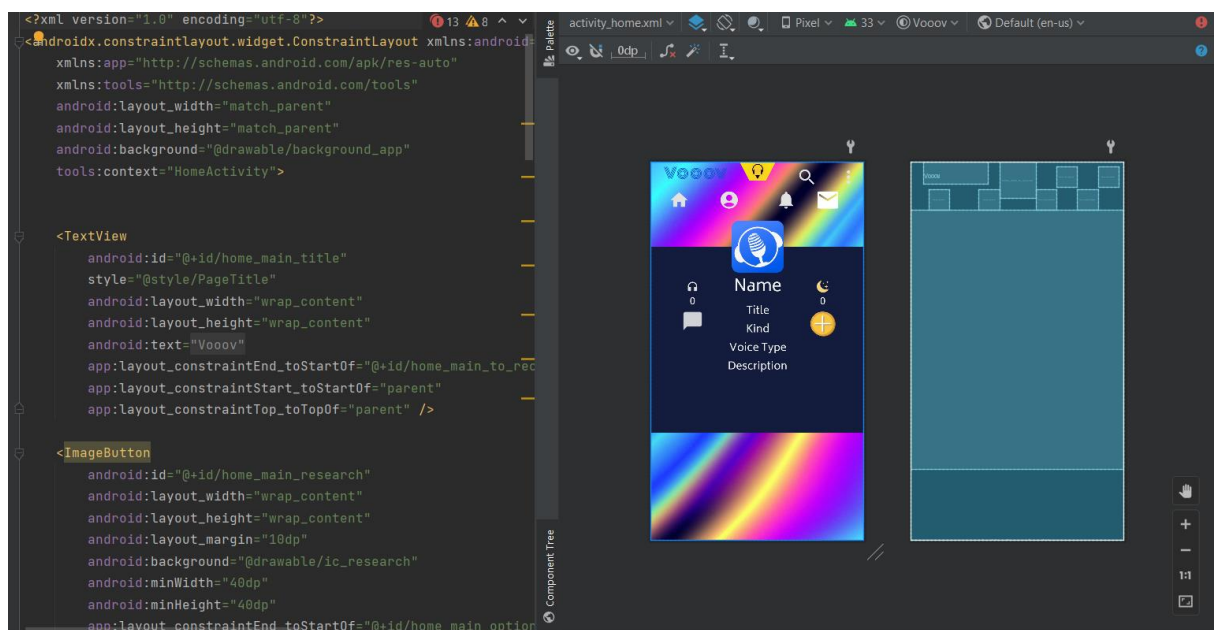
Ce fichier permet de gérer les styles de textes à appliquer dans l'application et la couleur des différents éléments standards. Finalement, c'est le fichier qui s'apparente le plus à un fichier CSS. Contrairement aux autres fichiers du dossier values, qui s'occupe d'apporter un référentiel, celui-ci s'occupe d'avantage d'une forme de mise en page. Ce dossier permet également la création d'une barre de navigation, mais nous n'en feront pas usage à proprement parler dans cette application.

## Le dossier « layout »

Android studio gère les pages des applications à travers ce qui se nomme des activités ou « activity » en anglais. Il est possible d'insérer à l'intérieur de ces pages, un élément plus petit et modulable appelé fragment. Par la suite, il est possible de générer d'autres formes de visuels comme des Popup, qui seront appelés ponctuellement. Il est également possible de créer des items, qui seront appelés dans des listes de défilement. L'ensemble de ces fichiers de mise en page sont stockés dans un dossier nommé « layout ».

L'ensemble de ces types de fichiers, s'ils se différencient par leur utilisation, comporte des similarités dans leur conception. Chacun de ces fichiers est écrit en langage xml pour permettre la mise en page.

Android studio nous propose plusieurs solutions pour mettre en page les différents layouts. Il propose également un aperçu du rendu en temps réel. Voici à quoi ressemble l'interface d'un écran partagé avec la partie code xml et la partie visuelle :



Il est possible soit, de manipuler le xml, soit d'intervenir directement sur la partie visuelle avec les différents outils de palette. J'utilise le xml, cela permet selon moi, une meilleure précision dans les prises de décisions.

Android Studio a beaucoup évolué depuis sa création et continue encore d'évoluer aujourd'hui. De ce fait, les modalités de mise en page ont également évolué. Aujourd'hui, la mise en page qui permet le plus de liberté est le « constraint layout ». Il permet d'accrocher les éléments les uns par rapport aux autres et notamment au cadre parent qui se trouve être les limites extérieures du layout.

Contrairement à une page web, qui sépare véritablement la partie HTML et CSS, la mise en page d'Android est beaucoup plus confondue. Une bonne partie des variables sont externalisées dans le dossier value, mais la grosse partie de la mise en page se fait directement dans les fichiers layouts.

Chaque élément doit posséder ses attributs propres. Il est nécessaire de définir la largeur et la hauteur qu'il doit occuper. Il est préférable d'utiliser les deux variables fournies par Android `match_parent` (occupe la taille totale du parent) et `wrap_content` (occupe la taille de l'élément). Cela permet une meilleure portabilité selon les appareils utilisés.

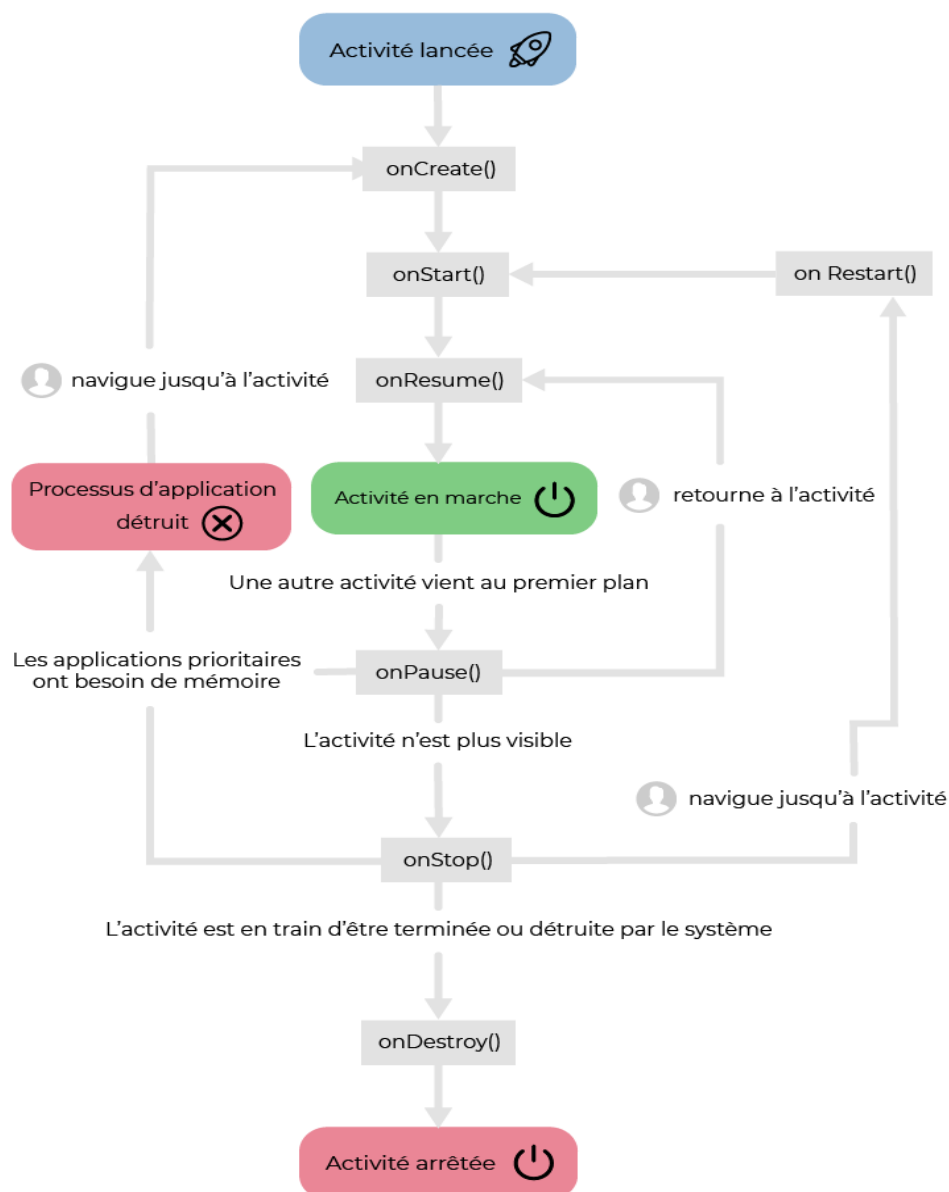


Dans un constraint layout, il est obligatoire de préciser au moins deux points d'accroche à un élément.

Par ailleurs, il est important de donner un id à chaque élément afin de pouvoir y accéder depuis la partie backend et l'utiliser. Cet id doit être unique dans toute l'application.

### Les activités

Une activité, dans le dossier layout, est un conteneur, elle représente un écran. Une application Android doit faire face à plus de contraintes de ressources (batterie, mémoire...) par rapport à une application web ou de bureau. En ce sens il est intéressant de voir qu'Android tente de répondre à ces contraintes en appliquant un cycle de vie aux différents éléments d'une application. En l'occurrence, voici le cycle de vie possible d'une activité :



De cette manière, le développeur peut gérer la durée de vie d'une activité en évitant la superposition de trop d'activités en arrière-plan. Cela a son importance également pour le fonctionnement de

l'application. Dans le cas où un utilisateur revient en arrière sur une activité par exemple. Nous parlerons plus en détail de l'utilisation des cycles de vie d'activités un peu plus loin.

## Les dossiers complémentaires du frontend dans res

En parallèle des principaux dossiers mis à disposition dans le dossier général frontend res, il existe plusieurs dossiers utiles.

### *Drawable*

Le dossier « drawable » contient l'ensemble des ressources visuels de l'application. Il permet de stocker l'ensemble des icônes, le logo de l'application, diverses images... Android propose une large bibliothèque d'icônes standards qu'il est possible d'intégrer dans un projet.

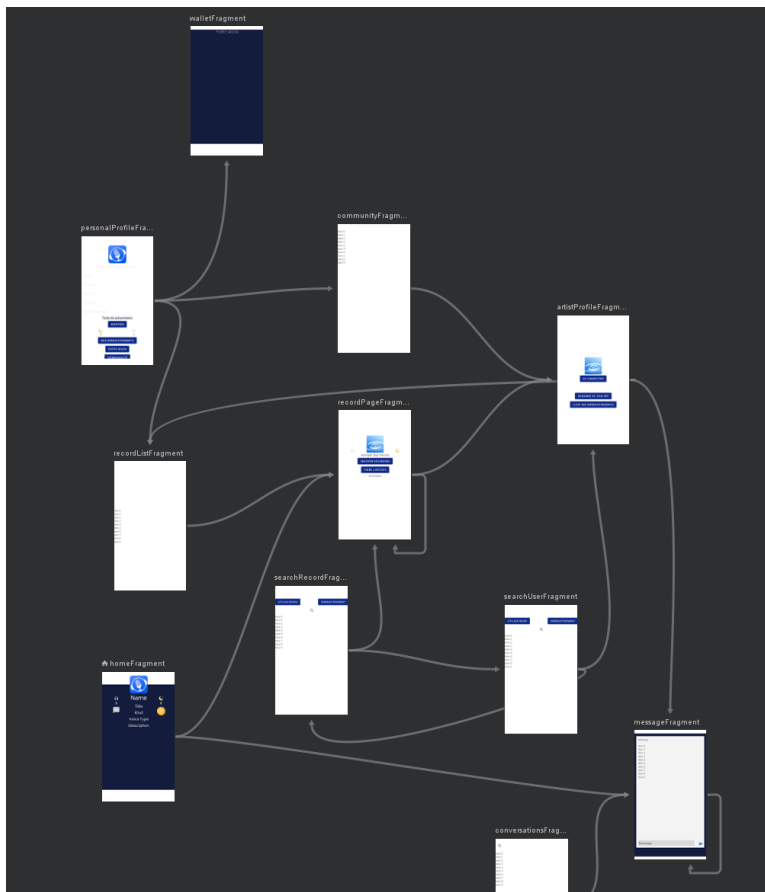
### *Font*

Le dossier « font » permet d'intégrer des polices d'écritures externes au projet.

### *Navigation*

Le dossier « navigation » comporte les fichiers nav\_graph. Ces fichiers sont organisés sous forme de graphe d'ordonnances et permettent de simplifier les diverses interactions entre les fragments. Un fragment hôte est défini et celui-ci accueille le reste des fragments par la suite. Grâce à ces interactions, il est possible de passer des paramètres aux fragments. Il est possible d'utiliser plusieurs navgraphs dans une application.

Voici le fichier nav\_graph de l'application Vooov qui accueille les différents fragments hébergés par l'activité HomeActivity :



## Réalisation de la partie Backend

Le backend d'une application Android permet de gérer l'ensemble des interactions et actions de l'application. Dans ce projet, le langage utilisé pour la partie backend est Kotlin.

### Langage Kotlin

Le langage Kotlin a été choisi pour cette application, à la place du Java. Le but étant de monter en compétence sur un langage en pleine croissance. Par ailleurs, c'est aujourd'hui le langage officiellement soutenu et recommandé par l'entreprise Google, dans le développement d'applications Android. Pour rappel, l'entreprise Google possède la plate-forme Android. Le langage Kotlin est un langage de programmation orienté objet et fonctionnel, avec un typage dynamique qui permet de compiler pour la machine virtuelle Java.

### Manifest et Gradle

Avant d'aborder l'utilisation plus en détail de la conception du backend, il est important de faire cas du fichier `AndroidManifest.xml` et des fichiers `build.gradle`. Ces fichiers ont une grande importance dans la conception d'une application Android.

Le fichier manifest a plusieurs fonctions. C'est dans ce fichier que sont émises les requêtes d'autorisation des différentes permissions du système Android. Pour cette application, quatre autorisations sont demandées :

- `INTERNET` (Pour autoriser l'accès à internet et ainsi récupérer les données du serveur)
- `ACCESS_NETWORK_STATE` (permet d'accéder à l'état du réseau de l'appareil. C'est utile pour vérifier que l'appareil est bien connecté à un réseau avant d'envoyer une requête)
- `RECORD_AUDIO` (permet l'autorisation d'enregistrement audio. L'autorisation doit être confirmée dans nouveau dans l'activité concernée pour être accepté par google)
- `STORAGE` (Cela permet de stocker des fichiers sur l'appareil, en l'occurrence, des fichiers temporaires d'enregistrement audio).

Le fichier manifest permet également de cartographier l'ensemble des activités. Il sert aussi à désigner l'activité responsable d'accueillir l'utilisateur à son arrivée dans l'application.

Les fichiers gradle quant à eux, permettent de gérer les dépendances de l'application. C'est en quelque sorte l'équivalent de composer pour Symfony. Dans ces fichiers sont stockés les noms d'accès aux diverses bibliothèques, et plugins, dont l'application a besoin pour fonctionner. Ces fichiers s'occupent également de gérer les versions d'Android prises en charge par l'application. Gradle s'occupe de tout ce qui n'est pas construit par le développeur de l'application, pour que celle-ci fonctionne.

### Création d'une application de test

Dans le but de tester le fonctionnement de la structure générale, et en particulier l'échange de données avec le serveur grâce à l'API, il était important de réaliser des tests préliminaires. En ce sens,

j'ai créé une application de test en Kotlin également. Celle-ci fait usage des principales méthodes de l'API.

Vous pouvez retrouver le code de cette application dans ce repository sur Github :

<https://github.com/nathan-kedinger/TestApi>

Dans cette application, les tests ont été réalisés via la bibliothèque Retrofit. Nous détaillerons plus en détail par la suite cette bibliothèque.

## Conception au format MVVM

La réalisation de cette application a essayé de suivre au mieux le standard de programmation MVVM : Model View ViewModel. Ce standard est un modèle de conception architecturale d'interface utilisateur. Il permet de découpler l'interface utilisateur et le code ne lui étant pas associé. C'est-à-dire que les différentes couches applicatives sont séparées.

La couche Model contient les données liées à la logique métier et est stocké dans les repositories. C'est cette couche qui viendra interroger l'API pour récupérer les données à fournir au client, ici l'application.

La couche View est la couche qui transmet la vue aux utilisateurs. C'est l'interface graphique. On peut l'apparenter à la partie frontend.

Et enfin la couche ViewModel est chargée de transformer et organiser les modèles métiers avec la vue. C'est cette couche qui permet le lien et les interactions entre les données récupérées et l'interface graphique.

## Bibliothèque Retrofit

Retrofit est une bibliothèque open-source pour le développement d'applications Android. Elle permet de simplifier la communication avec des API REST. Elle fournit un moyen facile et efficace de communiquer avec des serveurs distants en utilisant des appels HTTP pour récupérer ou envoyer des données.

Pour utiliser Retrofit, il est nécessaire en premier lieu de définir une interface spécifiant les appels d'API. La bibliothèque se charge des détails de connexion et de communication, tel que l'encodage et le décodage des données, la gestion des erreurs... Retrofit prend en charge les fichiers JSON que nous utilisons avec notre API.

Cette bibliothèque permet de simplifier grandement les échanges http avec le serveur par rapport à la méthode native d'Android.

Voici un exemple d'interface de Retrofit du fichier RecordRetrofitInterface :

```

import ...

@nathan-kedinger
interface RecordsRetrofitInterface {
 @nathan-kedinger
 @POST("records/create.php")
 suspend fun postRecord(@Body data: RecordModel): Response<RecordModel>

 @nathan-kedinger
 @GET("records/read.php")
 suspend fun getRecords(): Response<MutableList<RecordModel>>

 @nathan-kedinger
 @GET("records/readOne.php")
 suspend fun getOneRecord(@Query("id") id: Int): Response<RecordModel>

 @nathan-kedinger
 @PUT("records/update.php/{uuid}")
 suspend fun updateRecord(@Path("uuid") uuid: String?, @Body record: RecordModel): Response<RecordModel>

 @nathan-kedinger
 @DELETE("records/delete.php")
 suspend fun deleteRecord(@Query("uuid") uuid: String): Response<RecordModel>

 @nathan-kedinger
 @Multipart
 @POST("record_files/record_upload.php")
 suspend fun uploadRecordFile(@Part("name") filename: RequestBody,
 @Part("type") mimeType: RequestBody,
 @Part("size") fileSize: RequestBody,
 @Part file: MultipartBody.Part): Response<ResponseBody>

 @nathan-kedinger
 @GET("record_files/record_download.php")
 suspend fun downloadRecordFile(@Query("file") filename: String): Response<ResponseBody>

 @nathan-kedinger
 @DELETE("record_files/delete.php")
 suspend fun deleteRecordFile(@Query("uuid") uuid: String): Response<RecordModel>
}

```

## Models

Afin de récupérer et stocker les objets JSON envoyés de la base de données et du serveur par l'API, il est nécessaire de créer des objets model. Ces objets sont calqués sur les tables de la base de données.

Voici un exemple de model de l'application. Celui-ci permet de récupérer les informations d'un enregistrement audio :

```

@nathan-kedinger
data class RecordModel(
 val uuid: String = "null",
 val id: Int? = null,
 val artist_uuid: String? = "Doe",
 val title: String = "Jon",
 val length: Int = 0,
 val number_of_plays: Int = 0,
 var number_of_moons: Int = 0,
 val voice_style: String = "",
 var kind: String = "",
 val description: String = "",
 var created_at: String = "",
 val updated_at: String = "",
)

```

## Les repository

Une fois le model créé et les requête http définies dans l'interface Retrofit, la part model de l'architecture MVVM entre en jeu. Ce sont les « repositories » qui ont pour mission d'envoyer une demande à l'interface Retrofit. Cela permet de récupérer les données et de les transmettre ensuite à l'application en les stockant dans un objet adéquat. Voici un exemple de Repository de l'application Vooov :

```
nathan-kedinger
class RecordRepository {

 val retrofit = Retrofit.Builder()// Construction du client Retrofit
 .baseUrl("https://vooov-api.fr/")
 .addConverterFactory(GsonConverterFactory.create())
 .build()
 private val create: RecordsRetrofitInterface = retrofit.create(RecordsRetrofitInterface::class.java)

 nathan-kedinger
 suspend fun createRecordData(record: RecordModel): Response<RecordModel> {
 return try{
 create.postRecord(record)
 } catch (e: Exception) {
 throw IOException("Error creating record",e)
 }
 }
}
```

Dans cet exemple, un appel est lancé vers l'interface Retrofit en lui fournissant l'url de l'api. Retrofit converti le fichier JSON reçu grâce à la librairie GsonConverterFactory et le transmet au repository.

La méthode createRecordData contient en paramètre le model dans lequel va être stocké l'objet reçu.

Grâce au return elle renvoi un Response<Model> qui sera fourni par la suite au ViewModel.

On note le mot clé « suspend ». Les application Android fonctionne sous le principe de thread qui sont des processus. Le processus principal est lancé et compilé au lancement d'une activité. Cela pose problème dès lors que certaines actions prennent plus de temps que l'exécution du thread. Par exemple, ici, la récupération de données auprès du serveur a de très grande chance d'être plus lente que le lancement d'une activité.

Le mot clé suspend est créer par l'entreprise IntelliJ dans Kotlin, pour forcer une méthode à se lancer, uniquement, sous condition d'être inséré dans une coroutine. Nous détaillerons plus tard les coroutines mais pour faire simple elle simplifie le callback de Java et permettent de lancer un processus dans un thread parallèle qui ne prendra effet que lorsque toutes les données auront été libérées.

L'ensemble est pris dans un try catch qui renvoi une erreur en cas de problème dans la récupération des données.

## Les ViewModels

Les « ViewModels » sont les classes Kotlin qui servent de jointure entre les repositories et les Activities. Ces classes servent de classes utilitaires. Elles vont récupérer les données envoyées par les repositories et les exploiter si besoin. Cela permet de bien séparer chaque couche. Voici un exemple d'une méthode de ViewModels:

```

nathan-kedinger
class ConversationsViewModel : ViewModel() {
 private val repository = ConversationsRepository()

 nathan-kedinger
 suspend fun createConversation(contactUuid: String, selfUuid: String) {
 val randomUuid = UUID.randomUUID().toString()
 val user1: Response<UserModel> = UserRepository().readOneUserData(selfUuid)
 val user2: Response<UserModel> = UserRepository().readOneUserData(contactUuid)

 val conversation = ConversationsModel(
 randomUuid,
 selfUuid,
 contactUuid,
 title: "${user1.body()?.pseudo} ${user2.body()?.pseudo}",
 Date().toString(),
 Date().toString()
)
 val response = repository.createConversationData(conversation)
 if (response.isSuccessful) {
 Log.i(ContentValues.TAG, msg: "Conversation créée avec succès")
 } else {
 Log.e(ContentValues.TAG, msg: "Erreur lors de la création de la conversation")
 Log.i(ContentValues.TAG, msg: "$conversation")
 Log.i(ContentValues.TAG, msg: "${response.code()} Message: ${response.errorBody()?.string()}")
 }
 }
}

```

Cette méthode a pour but de créer une nouvelle conversation. Dans cette méthode, on récupère les UUID de l'utilisateur courant et celui de l'utilisateur avec lequel la conversation va être créée. On crée ensuite un UUID aléatoire qui sera l'UUID de la conversation. On récupère ensuite les informations des utilisateurs et en particulier leurs pseudos. Une fois l'ensemble des données récupérées, on les intègre dans un model, en l'occurrence le model de conversation. On implante également la date du moment dans la ligne date de création et de modification. Ce sont les mêmes à ce moment.

Une fois l'ensemble réalisé l'objet créé est envoyé au repository. Celui-ci se chargera de le transmettre à Retrofit, qui le transmettra à l'API, qui enfin, le transmettra à la base de données.

Si la requête est fructueuse, le serveur renvoi une réponse positive. Dans le cas contraire, nous récupérons la réponse négative et les erreurs grâce au divers Logs transmis au Logcat d'Android Studio.

Le Logcat permet de suivre les évolutions de l'application en cours d'utilisation, à travers des logs, qui vont du log d'erreur au simple log d'information.

## Fonctionnalités d'application

Maintenant que nous avons pu découvrir les bases du fonctionnement de la partie backend de l'application, nous allons nous intéresser aux différentes fonctionnalités de l'application en elle-même.

### Gestion des threads et de l'asynchrone d'Android

Avant d'entrer dans le vif du sujet, il est important d'apporter un peu de clarté sur l'une des problématiques évoquées plus haut. Celle-ci sera rencontrée à plusieurs reprises au cours de la partie suivante. La problématique dont il est question est propre au fonctionnement d'Android. Ce sont les threads.

Dans une application Android, l'interface utilisateur est gérée par un thread appelé thread principal (ou "main thread" en anglais). Ce thread est responsable de la gestion des événements d'interface utilisateur tels que les clics de boutons et les gestes tactiles, ainsi que de l'affichage de la mise à jour de l'interface utilisateur en réponse à ces événements.

Cependant, si une application effectue une tâche longue ou gourmande en ressources, comme le téléchargement d'un fichier volumineux ou le traitement d'une image, cela peut bloquer le thread principal et rendre l'interface utilisateur inutilisable, jusqu'à ce que la tâche soit terminée.

C'est là que les threads entrent en jeu. En créant un thread séparé pour effectuer des tâches en arrière-plan, une application peut libérer le thread principal pour continuer à gérer l'interface utilisateur sans interruption. Une fois la tâche terminée, le thread en arrière-plan peut envoyer un message au thread principal pour mettre à jour l'interface utilisateur avec les résultats.

En résumé, les threads sont utilisés dans Android pour effectuer des tâches en arrière-plan afin de libérer le thread principal pour gérer l'interface utilisateur et éviter ainsi les blocages et les temps de réponse lents.

Pour permettre la gestion des threads, les créateurs du langage Kotlin : IntelliJ, ont créé un procédé permettant de simplifier leur utilisation. Ce procédé est appelé coroutine et remplace en quelque sorte le Callback de Java. Afin d'éviter l'appel des fonctions en dehors d'une coroutine, on ajoute le mot clé suspend devant celle-ci. Voici un exemple d'utilisation d'un appel à une méthode en asynchrone :



```

// Load all messages for the conversation
CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
 messageViewModel.showAllConversationMessages(conversationUuid)
}

// Observe changes in the messageList LiveData and update the RecyclerView
messageViewModel.messageList.observe(viewLifecycleOwner, Observer { messageList ->
 if(messageList != null){
 // Sort messages by send time and set the adapter for the RecyclerView
 val sortedMessages = messageList.sortedBy { it.send_at }
 recycler.adapter = MessageAdapter(
 context: this@MessageFragment,
 sortedMessages,
 R.layout.item_message,
 viewLifecycleOwner,
 userViewModel,
)
 recycler.scrollToPosition(position: messageList.size-1)
 }
})

```

Dans cette méthode, on appelle tout d'abord une méthode dans une coroutine pour récupérer une donnée de manière asynchrone. Par la suite on utilise un observer qui récupère la valeur assignée à une variable définie dans le ModelView visé. De cette manière, on peut utiliser les données récupérées. En l'occurrence, on récupère ici l'ensemble des messages d'une conversation et on les affiche dans un RecyclerView (liste dynamique) trié par date en forçant un scroll jusqu'en bas de la liste. Nous reviendrons un peu plus bas sur la question des listes et des RecyclerViews.

## Réalisation de la partie identification et inscription

L'inscription et l'identification sont souvent les premières choses que l'on réalise dans une application. Cependant, dans cette application, l'inscription n'est pas obligatoire. Dans l'idée de « fidéliser » les nouveaux utilisateurs, il semble important de garder une certaine simplicité dans l'utilisation de l'application. L'utilisateur peut ensuite faire le choix ou non de s'inscrire si le contenu lui a plu et qu'il souhaite interagir avec celui-ci.

L'utilisateur non connecté peut accéder à la page d'accueil, à la liste des enregistrements audios et à l'écoute de ceux-ci. En revanche, lorsqu'il tentera d'accéder à la messagerie, au studio ou à une quelconque action nécessitant d'être identifié, il se verra afficher un popup lui proposant de se connecter.

Dans le cas où cet utilisateur n'a pas de compte, il lui sera permis, sur la page de connexion, de se diriger vers une page d'inscription.

L'intérêt de permettre aux utilisateurs la connexion est de pouvoir leur permettre d'interagir avec l'application. Il est donc nécessaire de gérer le maintien de la session ouverte. La méthode la plus simple pour permettre à un utilisateur de garder sa session ouverte est de stocker ses informations dans un fichier propre à l'application sur l'appareil de l'utilisateur. (Uniquement les informations essentiels et non sensibles, il y a toujours un risque qu'une application tierce puisse accéder à ces données) Pour permettre le stockage de ces données, il est nécessaire d'utiliser les « shared

préférences ». Elles permettent de stocker une valeur qui sera associée à une clé. Il sera possible de récupérer cette valeur par la suite en transmettant la clé à l'application.

Voici une partie du code pour y parvenir, écrit dans une classe utilitaire, la classe `CurrentUser` :

```
//allow to register a string value in sharedPreferences
fun saveString(key: String, value: String) =
sharedPreferences.edit().putString(key, value).apply()
//allow to register a value in sharedPreferences. here if user is connected
or not
fun saveConnection(key: String, value: Boolean) =
sharedPreferences.edit().putBoolean(key, value).apply()
//allow to get back a string value from sharedPreferences
fun readString(key: String) = sharedPreferences.getString(key, null)
```

## La messagerie et les `recyclerViews`

La messagerie est au cœur de notre application puisque celle-ci permet aux utilisateurs d'échanger entre eux. Cette messagerie doit par exemple permettre à un utilisateur, en recherche d'une voix off, pour ces vidéos YouTube, présentant son association, de demander à un autre utilisateur, de lui lire un texte prédéfini, après avoir écouté la voix de celui-ci, dans la liste des enregistrements.

Cette application a un but social et doit donc permettre l'échange.

Afin de permettre aux utilisateurs de se retrouver dans leurs messages, des conversations ont été créées dans un premier temps. Celles-ci regrouperont l'ensemble des messages communs à deux utilisateurs.

Les conversations comme les messages doivent être affichées sous forme de liste pour pouvoir être accessibles simplement aux utilisateurs. Une liste comportant de simples textes peut être relativement légère. Cependant, certaines listes peuvent être plus lourdes et longues. Par exemple, si celles-ci comportent des photos. Gardons bien en tête les limites des appareils mobiles.

Dans un souci d'optimisation de mémoire, Android a créé un outil permettant de gérer les listes de manière dynamique tout en allégeant au maximum le processus. Cet outil mis à disposition des développeurs Android s'appelle un « `recyclerView` ». Les `recyclerViews` ont un fonctionnement relativement complexe. Plutôt que de charger une liste complète d'éléments, ils vont charger uniquement les éléments à afficher à l'écran (plus quelques-uns avant et après). Ils viendront charger les autres éléments au fur et à mesure que l'utilisateur se déplace dans la liste. Dans le même temps, les éléments sortants de l'écran sont détruits. De cette façon, seul le nombre d'éléments affichable à l'écran seront chargés, à plus ou moins deux éléments. Les quelques éléments chargés en avance dans la liste permettent d'éviter les ralentissements pendant l'utilisation de l'application.

Afin de mettre en place des listes dynamiques, il est nécessaire de respecter un certain standard de création. Dans un premier temps, il est nécessaire de créer un item. Voici par exemple un item de liste d'enregistrement audio :



Il est ensuite nécessaire d'implémenter un recyclerview dans le layout où l'on souhaite accéder à notre liste.

Une fois les questions frontend réglées, il faut aborder la partie Kotlin. Il est donc nécessaire maintenant de créer un fichier « adapter ». Ce fichier est relativement standard. Il comporte trois méthodes :

- onCreateViewHolder : Cette méthode s'occupe de créer la vue
- onBindViewHolder : Cette méthode s'occupe de lier la partie visuel aux éléments de la liste et gère la partie utilitaire. C'est dans cette méthode que peuvent être gérés les cliques par exemple.
- getItemCount : Cette méthode traite du nombre d'éléments que l'on souhaite avoir dans notre liste. En règle générale, la fonction `smtgList.size` est utilisée. Elle permet de récupérer le nombre d'éléments présents dans la liste et de s'y adapter.

Il est également possible d'ajouter une inner classe afin d'alléger la taille de la méthode `onBindViewHolder`. Elle s'occupera de lier les éléments de layout, laissant ainsi la méthode `onBindViewHolder` s'occuper de la partie utilitaire.

Une fois le fichier « adapter » terminé, il faut maintenant intégrer le recyclerview dans l'activité choisie.

Voici le code d'implémentation du recyclerview de conversations

```
// Conversations recyclerview
val sortedConversations = conversationsUserList.sortedBy { it.updated_at }
recycler.adapter = ConversationsAdapter(
 this@ConversationsFragment,
 sortedConversations,
 R.layout.item_conversation,
 findNavController(),
 viewLifecycleOwner,
 conversationViewModel,
 userUuid
)
```

Dans ce code on peut apercevoir les différents paramètres qui peuvent être passé à un recyclerview. Le premier paramètre est le « context ». Dans Android, les éléments doivent être assortis à un contexte. Le contexte dans Android est un élément clé. Il fournit un accès aux ressources et les informations nécessaires pour que l'application fonctionne correctement. En l'occurrence, le contexte est le fragment dans lequel se trouve le recyclerview.

Le deuxième argument fournit la liste de données à lier et afficher. Ici, la liste est triée par date.

Le troisième argument lie le layout de l'item à la liste.

Les arguments suivants sont des arguments optionnels. Ils sont fournis au `recyclerView`, ne pouvant être appelés directement depuis lui, pour cause d'absence de contexte propre.

## Réalisation de la partie audio et gestion des fichiers

### Enregistrement

La réalisation de la partie audio nous amène à la question de la gestion des fichiers dans une application Android et à leur utilisation.

Lorsque nous enregistrons un son via l'application, nous faisons appel à la bibliothèque `MediaRecorder`.

Voici comment celle-ci s'utilise :

```
// Create a new instance of MediaRecorder with the current context as
parameter
recorder = MediaRecorder(this).apply {
 // Set the audio source to default
 setAudioSource(MediaRecorder.AudioSource.DEFAULT)
 // Set the output format to MPEG-4
 setOutputFormat(MediaRecorder.OutputFormat.MPEG_4)
 // Set the output file path
 setOutputFile(fileFullPath)
 // Set the audio encoder to HE_AAC
 setAudioEncoder(MediaRecorder.AudioEncoder.HE_AAC)
 binding.studioMeterReading
 binding.studioMeterTotalTime

 try {
 prepare()
 } catch (e: IOException) {
 Log.e(LOG_TAG, "prepare() failed")
 }

 start()
}
```

On fait appel à une méthode tiers pour lancer et stopper l'enregistrement :

```
private fun onRecord(start: Boolean) = if (start) {
 startRecording()
} else {
 stopRecording()
}
```

Voici à quoi ressemble le chemin absolu du fichier transmis au media player :

```
fileFullPath = "${filesDir.absolutePath}/${randomId}.mp4"
```

De cette manière, nous enregistrons le record dans un dossier temporaire. Il est ensuite possible de le transmettre en base de données grâce à la méthode `uploadRecord`. Nous enregistrons également les

informations du record en base de données pour pouvoir retrouver le fichier plus tard grâce à son nom :

```
// Call the createRecord function of the repo object with the record
parameter
repo.createRecord(record)

// Launch a coroutine with the main dispatcher
CoroutineScope(Dispatchers.Main).launch {
 // Switch to the IO dispatcher
 withContext(Dispatchers.IO) {
 // Call the uploadRecord function of the RecordRepository object
 with the fileFullPath and randomId parameters
 RecordRepository().uploadRecord(fileFullPath, randomId)
 }
}
```

### *Lecture d'un fichier*

Il est ensuite possible de lire un fichier enregistré en le téléchargeant dans l'autre sens.

Cette fois, nous utilisons la méthode MediaPlayer. Voici la méthode en question :

```
suspend private fun startPlaying(currentRecordUuid: String, context:
Context) {
 val audioRecord = repo.downloadAudioRecord("$currentRecordUuid.mp4")
 if (audioRecord != null) {
 // Save the downloaded audio file to a local file
 val file = File(context.filesDir, "$currentRecordUuid.mp4")
 file.writeBytes(audioRecord)
 Log.e("LOG_TAG", "le fichier existe-t-il? ${file.exists()}")
 Log.e("LOG_TAG", "$file")

 CoroutineScope(Dispatchers.Main).launch {
 // Initialize the media player with the local file path
 player = MediaPlayer().apply {
 try {
 setDataSource(file.absolutePath)
 setOnPreparedListener {
 start()
 }
 prepareAsync()
 } catch (e: Exception) {
 throw IOException("Error reading record", e)
 }
 }
 }
 }
}
```

Dans cette fonction, nous téléchargeons le record depuis le serveur. Nous stockons de nouveau le fichier dans un fichier temporaire puis nous lisons son contenu avec MediaPlayer.

## Exploration de l'application, Activités et fragments

Avant d'explorer l'application de façon plus dynamique, il est nécessaire d'aborder la question des Activities et des Fragments. Ces deux entités composent la couche View du modèle MVVM. Ce sont des entités liant la partie frontend et la partie backend. Chaque activity et chaque fragment fait référence à un fichier du layout.

Notre application est constituée de cinq activités. L'une des activités, `HomeActivity`, héberge un `host_fragment`. Il s'occupe de porter l'ensemble des autres fragments de l'application. De cette manière, l'activité principale contient la barre haute qui reste fixe. Le fragment du bas, contenant le bloc de lecture des records, reste également en place, où que l'on aille entre les fragments. Cela permet d'avoir un socle à notre application. L'activité principale, elle, héberge douze fragments.

Les quatre autres activités sont : le studio, il doit permettre de se consacrer à sa tâche et ne doit pas se faire superposer deux enregistrements ; la page de settings, celle-ci a également une vocation plus « sérieuse » ; et enfin les deux activités de connexion et d'inscriptions qui doivent également être séparées du reste des activités de l'application.

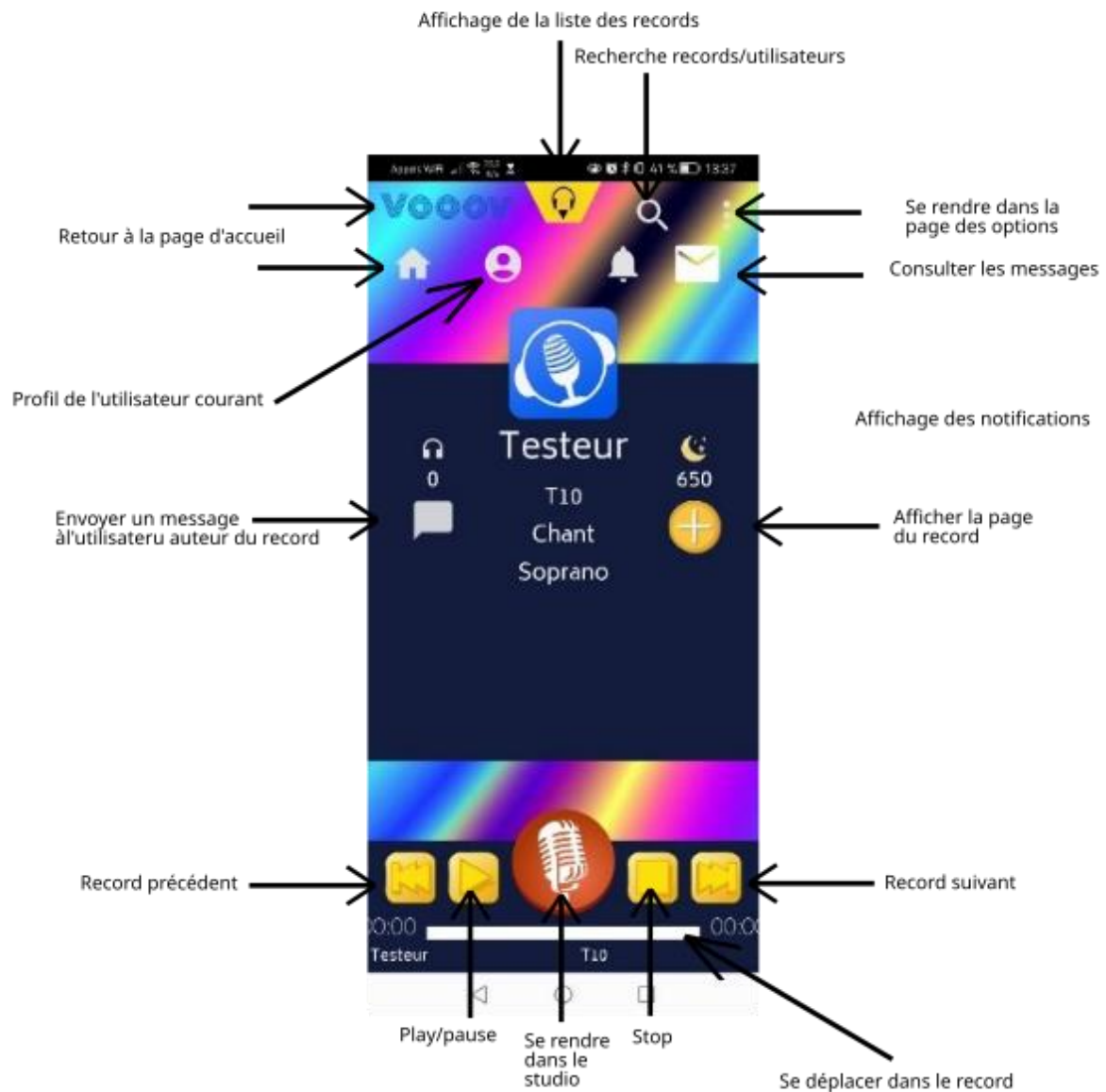
Maintenant que nous avons pu voir comment l'application était conçue, nous allons la rencontrer plus en détail.

Cette application est donc disponible sur mobile pour des utilisateurs Android. (Vous pouvez la télécharger sur votre mobile en mode développeur via Android Studio pour la tester si vous le souhaitez. Elle n'est pas encore tout à fait terminée cependant)

Voici donc étape par étape les interactions que peut avoir un utilisateur avec l'application :

Dans un premier temps, celui-ci arrive sur l'interface d'accueil. Il a ici plusieurs possibilités que nous détaillons dans l'image suivante :

la voici :



Depuis cette page, l'utilisateur a donc la possibilité de se rendre dans plusieurs endroits différents de l'application. Prenons l'exemple du studio :



Dans cette activité, l'utilisateur a la possibilité de créer un enregistrement. Il peut attribuer un type de voix, le style d'enregistrement et lui attribuer un nom qui sera visible une fois publié. Il peut ensuite publier l'enregistrement.

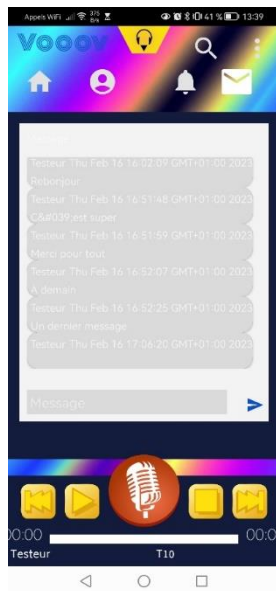


Ce fragment correspond au profil de l'utilisateur courant. Il peut consulter ses informations et modifier sa description. Celle que les utilisateurs pourront lire en se rendant sur son profil. Il peut également accéder un peu plus bas à ses enregistrements et son stock de moons disponible dans le fragment wallet.



Ce fragment correspond à la page d'un enregistrement. Un utilisateur peut décider d'attribuer des moons à l'enregistrement. Il peut se rendre sur la page de l'artiste et peut tout simplement consulter les informations sur l'enregistrement : le nombre d'écoutes, le nombre de moons reçues, le titre...





Ce fragment contient une conversation. Les messages y sont ordonnés par liste dans l'ordre de réception. L'utilisateur a la possibilité d'écrire un nouveau message et de l'envoyer. (Des modifications, notamment visuelles, sont à apporter)



Ce fragment contient la liste de l'ensemble des enregistrements. C'est également à cette liste que ressemble les listes de recherches et les listes d'enregistrements des différents utilisateurs. Au clic sur un item l'enregistrement doit être lancé dans le fragment du bas (la solution est en cours de recherche).

Au clic sur le bouton +, l'utilisateur est dirigé vers la page de l'enregistrement.

## Conception de l'Application Web

L'application web correspond en partie à l'image de l'application mobile transposée au web. Elle se démarque par l'absence de quelques fonctionnalités, notamment technique, en rapport avec les enregistrements et la lecture des audios. En revanche, elle propose un backoffice permettant aux administrateurs d'intervenir sur la modification des différentes entités, sans accès direct à la base de données.

Le code source est écrit en PHP et j'ai eu l'occasion de m'aider du Framework puissant qu'est Symfony dans sa sixième version. A l'occasion de la création du site web, j'ai pu me rendre compte qu'il était important de connaître parfaitement le fonctionnement de l'ORM utilisé pour créer une base de données en amont.

Dans le cadre de ce projet, créer des bases de données grâce au langage SQL, m'a permis d'approfondir mes compétences du langage. Cependant, à la réalisation de l'application web, j'ai été dans l'obligation de revoir quelques détails de la base de données et quelques approches de l'application mobile.

### Les bases du Framework Symfony

Symfony est un outil Open-source puissant et en permanente évolution. Il permet de rentrer très en détail dans l'aspect technique mais permet également de gagner du temps par rapport à une application construite entièrement « from scratch ». Ce Framework permet également d'apporter une certaine rigueur et cohérence dans l'architecture des projets. Il apporte également une forte sécurité puisque l'ensemble des fonctionnalités sont testé par un grand nombre de développeurs.

La première étape de mise en place d'un projet Symfony est l'installation de Symfony sur le poste de travail. Le plus simple pour réaliser cette action est d'abord d'installer l'outil Composer. Celui-ci est téléchargeable via le site : <https://getcomposer.org/download/>. Il suffit ensuite de choisir en fonction de ses besoins de configuration et de l'installer. Cet outil permet un gain de temps important dans la réalisation d'une application Symfony via l'utilisation des lignes de commandes. Celles-ci permettent d'activer des fonctions intégrées dans le Framework et ainsi de réaliser automatiquement les tâches les plus couramment utilisées. (A mon sens, c'est ce qui manque à Java Spring Boot) Il suffit pour les utiliser de les entrer dans le terminal de l'IDE choisit. Voici une liste des commandes que j'ai utilisé durant la conception de vooov.fr.

RACCOURCIS TERMINAL :

**Créer un projet** : `composer create-project symfony/skeleton nom_de_projet`

**Lancer le serveur** : `symfony serve`

**Créer un Controller** : `symfony console make:controller`

**Créer une classe User** : `symfony console make:user`

**Créer et configurer une base de donnée** : `symfony console doctrine:database:create`

**Créer une Entity** : symfony console make:entity

**Ajouter une jointure dans une entity** : Field type -> relation

**Créer un fichier de migration (requête SQL)** : symfony console make:migration

**Créer un fichier de migration après avoir modifié une entity manuellement** : symfony console doctrine:migrations:diff

**Appliquer la migration** : symfony console doctrine:migration:migrate

**Créer un formulaire** : symfony console make:form

**Créer un formulaire d'inscription** : *symfony* console make:registration-form

**Créer un formulaire de connexion** : symfony console make:auth

**Créer un backoffice** : symfony console make:admin:dashboard

**Créer des entités à manager pour le backoffice** : symfony console make:admin:crud

**Mapper des entités** : symfony console make:admin:crud

**Permet d'afficher tous les services de symfony** (il est possible de filtrer): Symfony console debug:autowiring

**Permet de supprimer tous les bundles en cas de problème (commandes du terminal)** : composer remove symfony/maker-bundle --dev

**Permet d'installer à nouveau les bundles (répare les bundles cassés entre temps)** : composer require symfony/maker-bundle --dev

## Organisation et architecture de Symfony

Symfony répond à une organisation MVC (Model View Controller). Cela permet de séparer la logique métier, la présentation et le contrôle des données. C'est une forme d'architecture très présente dans l'informatique. La représentation de celle-ci tend cependant à reculer avec l'arrivée de certaine architecture tel que MVVM utilisé dans les applications mobiles Android notamment. Je me suis permis d'ajouter quelques approches MVVM dans mon application. Cela permet une séparation des différentes couches encore plus clair et donc une bonne maintenabilité du code. L'isolation et la réduction de taille des fonctions permettent une efficacité accrue concernant les tests unitaires.

### La couche Model

Le modèle est responsable de la gestion des données et de la logique métier. Dans Symfony, les modèles sont souvent représentés par les entités et les repositories. Les entités sont des classes PHP qui définissent la structure des données et leurs relations, tandis que les repositories sont des classes qui contiennent des requêtes pour interagir avec la base de données.

### La couche View

La vue est responsable de la présentation des données à l'utilisateur. Dans Symfony, les vues sont représentées par les fichiers de template Twig. Twig est un moteur de template flexible et puissant utilisé par Symfony pour afficher les données et générer du contenu HTML. Il permet l'utilisation de fonctions et d'appels de variables directement dans ses fichiers.

### La couche Controller

Les contrôleurs sont responsables de la gestion des requêtes entrantes, de la manipulation des données du modèle et de la sélection de la vue appropriée pour afficher les données. Dans Symfony, les contrôleurs sont des classes PHP qui définissent des méthodes pour gérer différentes routes et requêtes.

### L'ORM (Object-Relational Mapping) Doctrine

L'ORM Doctrine est une bibliothèque qui facilite l'interaction entre le code PHP et la base de données. Doctrine fournit une couche d'abstraction, cela permet de fonctionner avec différentes bases de données sans avoir à modifier le code. Elle a un grand pouvoir de facilitation de l'organisation avec un mapping efficace entre entités et tables de la base. Elle permet également de faciliter l'écriture des requêtes, surtout les plus complexes. Elle permet également de gérer les migrations des schémas de bases de données sans avoir à toucher directement à la base en ligne de commande ou via phpMyAdmin par exemple. Finalement, elle permet de simplifier grandement la manipulation des données tout en rendant les échanges sécurisés.

## Etapas de construction du projet

### Création des entités

La structure du projet est déjà bien avancée et pensée à ce stade. Il suffit finalement de suivre les étapes suivantes dans l'ordre en intégrant les données déjà imaginées durant la conception de l'application mobile. C'est ici que cela se complique. Logiquement, on commence par copier les entités de la base de données existante. Or, je me heurte à deux problèmes.

#### *ID et UUID*

J'ai préféré utiliser les UUID pour mon application mobile plutôt que le ID. Or, nativement la création d'une entité par Symfony crée automatiquement une clé primaire auto-incrémentée sous la forme d'un id. Il est possible de modifier les entités afin de résoudre ce problème et de faire pointer la clé primaire vers les UUID. Cependant, cela m'a fait apparaître que les id simple peuvent avoir un intérêt à utiliser en parallèle des UUID. Les id permettent un classement chronologique (même si les date de créations peuvent également être utilisées), elles permettent un classement et une manipulation plus simple à l'utilisation et une visibilité plus claire que les UUID. Les deux couplés assurent une certitude de n'avoir jamais deux entités identifiées sous la même valeur, quelle que soit le nombre simultané de création.

#### *Clés étrangères*

Le deuxième problème que j'ai rencontré se trouve au niveau des clés étrangères qui lient les entités entre elles. Dans l'application mobile, j'avais lié mes colonnes de tables entre elles par les UUID. Symfony, à l'aide de son ORM Doctrine emploie un procédé différent, mais finalement plus clair et efficace. Doctrine lie une colonne d'entité directement à l'id d'une autre entité. Cela permet de récupérer un objet et de travailler directement dessus.

#### *Adaptation et mise en place*

Une fois les problèmes constatés, il faut penser aux solutions possibles. Le choix résidait entre modifier les entités dans Symfony ou modifier le code de l'application mobile. J'ai opté pour la conservation du model apporté par Symfony, celui-ci est robuste. Toutefois, la possibilité de donner un rôle plus important aux UUID est toujours présente puisque ni les id, ni les UUID n'ont été effacés. En parallèle, il était nécessaire de modifier l'application mobile.

Dans l'application Symfony, l'ordre le plus simple est donc de commencer par créer la base de données, puis l'ensemble des entités. On gère ensuite en une seule fois les migrations avec Doctrine (Il y a évidemment quelques retouches).

### Authentification et inscription

Symfony met à disposition des outils très efficaces pour la mise en place des authentifications et d'inscription des utilisateurs. Avec quelques lignes de commande, contrôleurs et entité sont créés. Symfony s'occupe de la gestion du hachage des mots de passe avec l'outil le plus puissant disponible. Ici c'est Argon2. Celui-ci est également celui utilisé dans l'application mobile et permet donc une interopérabilité entre les applications.

## Formulaires

Symfony dispose d'un outil efficace qui permet la manipulation des formulaires et simplifie la gestion des échanges de données entre utilisateurs et serveur.

## Controllers

Une fois les Entités créées, il faut maintenant créer les contrôleurs, qui se chargeront de router les utilisateurs entre les différentes pages. Ils permettront également la gestion des données des models.

## Vues

Viens ensuite la mise en place de la partie visuelle. Nous utilisons à ce stade différents outils, tel que Bootstrap et Twig. Le dossier réalisé avec Figma en première partie prend également tout son sens et permet un gain de temps en termes de prise de décisions.

Symfony permet l'utilisation d'un template de base. Celui-ci pourra être appelé ensuite par tous les autres template et permet de gagner du temps sur la formation des pages. Il est possible d'intégrer tous les visuels communs à l'ensemble des pages dans ce template.

## Fonctionnalités poussées

Une fois la structure de l'application mis en place, il faut maintenant s'intéresser aux diverses fonctionnalités poussées permettant la réalisation d'actions par les utilisateurs.

### Barre de recherche

Dans une application où le but est l'échange de services entre utilisateurs, il est important de permettre à ceux-ci de pouvoir chercher ce qu'ils souhaitent échanger. En ce sens, les barres de recherches et filtres de recherches sont essentiels.

Pour gérer les recherches, il est nécessaire de faire appel aux données de la base de données. Symfony nous permet d'utiliser les requêtes de Doctrine à travers les repositories.

Voici un exemple de requête que Doctrine permet de réaliser :

```
/**
 * @return AudioRecords[] Returns an array of AudioRecords
 objects
 */
public function findByTitle($title): array
{
 $query = $this->createQueryBuilder('a')
 ->join('a.artist', 'user')
 ->join('a.categories', 'categories')
 ->join('a.voice_style', 'voice')
 ->where('a.title LIKE :title')
 ->orWhere('user.pseudo LIKE :pseudo')
 ->orWhere('categories.name LIKE :categories')
 ->orWhere('voice.voice_style LIKE :voice')
 ->setParameters([
 'title' => "%{$title}%",
 'pseudo' => "%{$title}%",
 'categories' => "%" . ucfirst($title) . "%",
 'voice' => "%" . ucfirst($title) . "%",
])
 ->orderBy('a.id', 'DESC')
 ->getQuery();

 return $query->getResult();
}
```

Ici, Doctrine nous permet de rechercher dans une table à travers trois colonnes qui sont elles-mêmes jointes par clé étrangère à d'autres tables. Cette requête permet de trouver un enregistrement par son titre, le nom de l'artiste, le type de voix ou encore la catégorie de l'enregistrement.

## Création des enregistrements

Dans l'application qui vous parviendra le jour de l'examen, les fonctionnalités d'enregistrement et de lecture ne devraient pas être disponibles.

Cependant, les fonctions de création d'enregistrement des informations liées aux enregistrements vocaux sont fonctionnelles. Ceux-ci nécessitent l'utilisation de formulaires pour le dépôt de ces informations.

Voici un exemple de code qui illustrera les contrôleurs et en particulier le StudioController.php

```
<?php

namespace App\Controller;

use App\Classes\ConversationsClass;
use App\Entity\AudioRecords;
use App\Entity\Users;
use App\Form\AudioRecordType;
use DateTime;
use Doctrine\ORM\EntityManagerInterface;
use Ramsey\Uuid\Nonstandard\Uuid;
use
Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Bundle\SecurityBundle\Security;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Core\User\UserInterface;

class StudioController extends AbstractController
{
 /**
 * @param Request $request
 * @param EntityManagerInterface $em
 * @param Security $security
 * @param ConversationsClass $conversations
 * @return Response
 */
 #[Route('studio', name: 'app_studio')]
 public function index(Request $request,
EntityManagerInterface $em, Security $security,
ConversationsClass $conversations): Response
 {
 // Conversations logic implementation
 $currentUser = $security->getUser();
 if (!$currentUser instanceof Users) {
 $currentUserConversations = "";
 } else {
 $currentUserConversations = $conversations-
>findCurrentUserConversations($currentUser->getId(), $em);
 }
 }
}
```



```

 if ($security->isGranted('IS_AUTHENTICATED_FULLY')) {
 $user = $security->getUser();
 } else {
 $user = null;
 }
 $record = new AudioRecords();
 $form = $this->createForm(AudioRecordType::class,
$record);
 $form->handleRequest($request);
 $uuid = Uuid::uuid4();
 $uuid_string = $uuid->toString();
 $actualeDate = new DateTime('now');
 $actualeDate_string = $actualeDate->format('d/m/y');

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

 $record->setUuid($uuid_string);
 $record->setArtistId($user);
 $record->setLength(0);
 $record->setNumberOfPlays(0);
 $record->setNumberOfMoons(0);
 $record->setCreatedAt($actualeDate_string);
 $record->setUpdatedAt($actualeDate_string);

 $em->persist($record);
 $em->flush();

 $this->addFlash('notice', 'L\'enregistrement a
bien été créé.');
```

```

 }

```

```

 return $this->render('studio/index.html.twig', [
 'audioRecordForm' => $form->createView(),
 'conversations' => $currentUserConversations,
]);
 }
}

```

## Back-office avec easy-admin

Enfin, Symfony, à l'aide de l'ajout d'une bibliothèque, nous permet la création simple d'un back-office. Ce back-office permet l'ajout des valeurs dans la base de données pour des valeurs charnières qui peuvent avoir besoin d'évoluer. Cela permet à des utilisateurs de modifier certaines données sans avoir à manipuler directement la base de données. C'est utilisé ici par exemple pour modifier les types de voix ou les types d'enregistrements. Cependant, il peut permettre la gestions des utilisateurs, l'ajout de photos... La bibliothèque utilisée ici est easy-admin.

L'intérêt et la force principale de cet outil est donné par Symfony, qui permet l'administration de rôle différents selon les utilisateurs. Par la suite, un utilisateur selon son statut ne pourra accéder qu'aux pages dont il a l'autorisation d'accès.

Ces fonctionnalités se règlent dans le fichier `security.yaml`, situé dans le dossier `package`. C'est ce dossier qui comprend les fichiers permettant d'administrer l'utilisation des fonctionnalités de Symfony.

Voici le code permettant d'attribuer les accès selon les rôles :

```
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: ^/compte, roles: ROLE_USER }
```

Dans ce backoffice, il est nécessaire de créer de nouvelle table, que nous créerons grâce aux commandes doctrines et migration.

Nous avons ensuite besoin de mapper les nouvelles entités créées.

Nous avons la possibilité de créer des dépendances entre des classes, entre `category` et `products` par exemple.

## Déploiement du projet sur le serveur VPS

Le déploiement du site web sur serveur VPS nécessite quelques manipulations.

### Htaccess

Le premier fichier à mettre en place sur le serveur est le « .htaccess ». C'est un fichier de configuration utilisé par le serveur web Apache pour définir des directives et des règles de réécriture d'URL spécifiques à un répertoire. C'est lui qui va permettre de filtrer les URLs entrés par les utilisateurs selon certaines règles définies dans celui-ci. Il fait office de premier point de routing et renvoie l'utilisateur vers le fichier index. Cela permet d'éviter aux utilisateurs d'accéder aux fichiers autres que ceux contenus dans le dossier public.

Voici le fichier .htaccess du site internet :

```
<IfModule mod_rewrite.c>

RewriteEngine On

RewriteCond %{THE_REQUEST} /public/([^\s?]*) [NC]

RewriteRule ^ %1 [L,NE,R=302]

RewriteRule
!\.(js|gif|JPG|JPEG|PNG|jpeg|jpg|png|webmanifest|webp|css|txt|svg|woff|woff2|ttf|map|ico)$
public/index.php [L]

RewriteCond %{REQUEST_URI} !^/public/

RewriteRule ^(.*)$ public/$1 [L]

</IfModule>
```

Voici une explication ligne par ligne :

- **<IfModule mod\_rewrite.c>** : Vérifie si le module mod\_rewrite est activé sur le serveur Apache. Si c'est le cas, le code entre les balises <IfModule> et </IfModule> sera exécuté.
- **RewriteEngine On** : Active le moteur de réécriture pour ce fichier .htaccess.
- **RewriteCond %{THE\_REQUEST} /public/([^\s?]\*) [NC]** : Cette condition de réécriture vérifie si la requête originale (%{THE\_REQUEST}) contient un chemin commençant par /public/. Les parenthèses ( ) capturent la partie restante du chemin après /public/. [NC] signifie "No Case" (sans tenir compte de la casse), ce qui signifie que la correspondance est insensible à la casse.
- **RewriteRule ^ %1 [L,NE,R=302]** : Si la condition précédente est satisfaite, cette règle redirige l'utilisateur vers l'URL capturée (représentée par %1) avec un code de statut HTTP 302 (redirection temporaire). [L] indique que c'est la dernière règle à appliquer si

elle correspond, et [NE] empêche l'échappement des caractères spéciaux dans l'URL de substitution.

- **RewriteRule**  
`!\.(js|gif|JPG|JPEG|PNG|jpeg|jpg|png|webmanifest|webp|css|txt|svg|woff|woff2|ttf|map|ico)$public/index.php [L]` : Si l'URL demandée ne se termine pas par l'une des extensions de fichier spécifiées (par exemple, .js, .gif, .jpg, etc.), elle est réécrite pour pointer vers `public/index.php`. [L] signifie que c'est la dernière règle à appliquer si elle correspond. De cette façon
- **RewriteCond** `%{REQUEST_URI} !^/public/` : Cette condition de réécriture vérifie si l'URI de la requête (`%{REQUEST_URI}`) ne commence pas par `/public/`.
- **RewriteRule** `^(.*)$ public/$1 [L]` : Si la condition précédente est satisfaite, cette règle réécrit l'URL en ajoutant `/public/` au début du chemin. [L] indique que c'est la dernière règle à appliquer si elle correspond.
- `</IfModule>` : Ferme la balise `<IfModule>` ouverte à la première ligne.

## Composer

Dans le cadre de l'implantation d'un site développé à l'aide de Symfony, tel que celui-ci, il est nécessaire d'installer Composer sur le serveur. Composer permettra par la suite d'effectuer les migrations de Doctrine. Pour installer Composer, il est nécessaire d'avoir un serveur à jour et une version de PHP en adéquation entre les différentes parties. L'installation de Composer et des différents composants se fait à l'aide d'un terminal SSH et nous utilisons ici puTTY.

## Github

L'utilisation du versionning est essentiel pour le déploiement. Pour ce site internet je continue d'utiliser GitHub. Il n'est toujours pas question de déploiement continu en CI/CD. Cependant, l'utilisation des déploiements via les commandes git permet un gain de temps considérable. Cela permet également de minimiser les erreurs puisqu'il n'est pas nécessaire de transférer l'ensemble des fichiers manuellement, ou une partie, au risque d'en oublier. Le déploiement sur le serveur VPS fonctionne de la même manière que pour le déploiement de l'API.

## Mode dev et mode prod

Le déploiement du site sur le web signifie le passer en mode production. Symfony dispose de deux modes, le mode production et le mode développement. Le mode développement permet de récupérer les erreurs et le détail de celles-ci. C'est très utile pour le débogage d'une application. Cependant, cela donne toutes les informations du contenu de l'application. Il est dangereux de laisser ces informations accessibles à tous.

Pour passer le site du statut de développement à production, il est nécessaire de vider le cache (informations souvent utilisées nécessitant des calculs coûteux) et d'indiquer à Symfony cette transition. Cela peut se faire via cette ligne de commande avant le déploiement sur serveur :

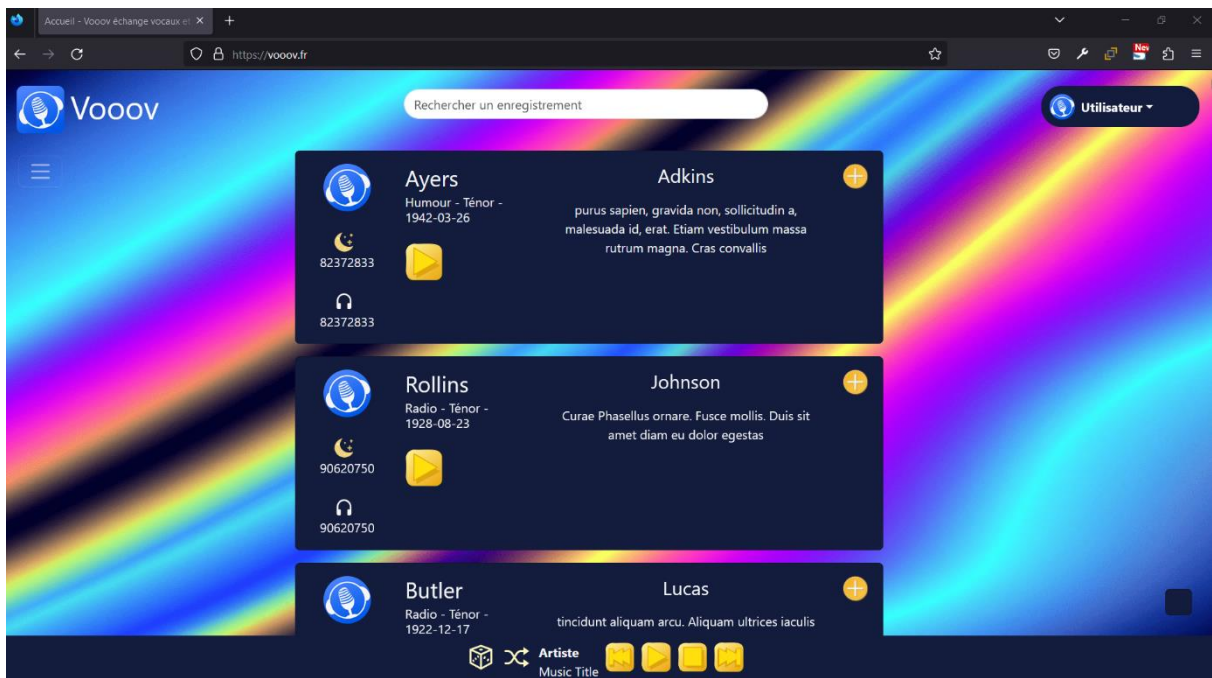
```
php bin/console cache:clear --env=prod
```

Cette commande aura également l'intérêt de faire ressortir d'éventuelles erreurs.

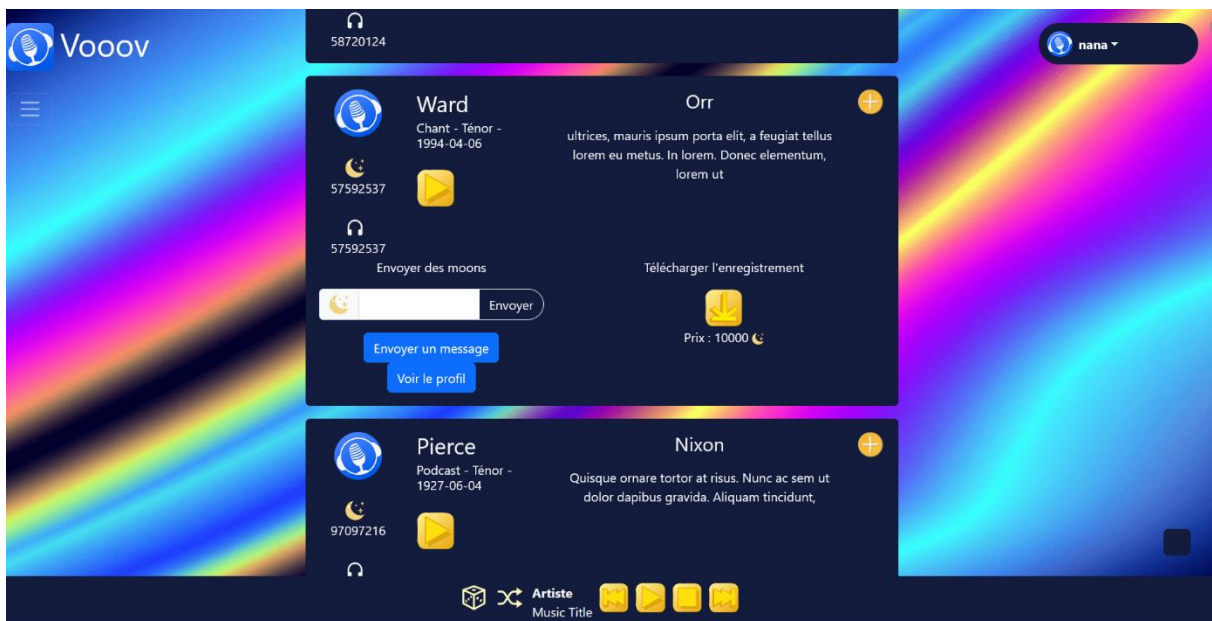
Symfony met à disposition un fichier qui référence l'ensemble des accès, tel que les outils de mailing, mais également les identifiants d'accès à la base de données. Ces accès à la base de données sont différents s'ils sont en local ou sur serveur. Afin de ne pas avoir à modifier systématiquement ce dossier, Symfony met à disposition un fichier `.env.local`. Cela s'avère particulièrement utile pour du travail en équipe. Le fichier `.env.local` contient les informations sur machine en local et lorsque le site est passé en production, Symfony récupère les informations du fichier `.env` stocké sur serveur.

## Présentation visuelle de l'application Web

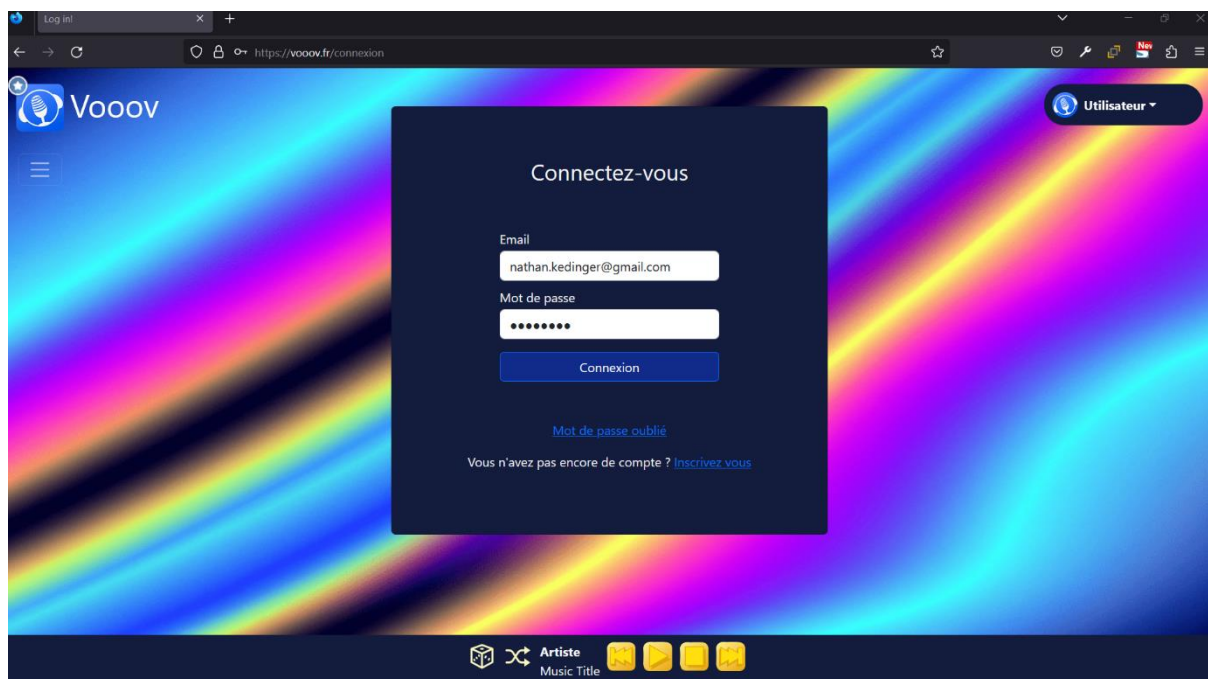
Page d'accueil du site : Permet de visualiser les derniers enregistrements publiés. Possibilité d'effectuer des recherches à travers les enregistrements via la barre de recherche.



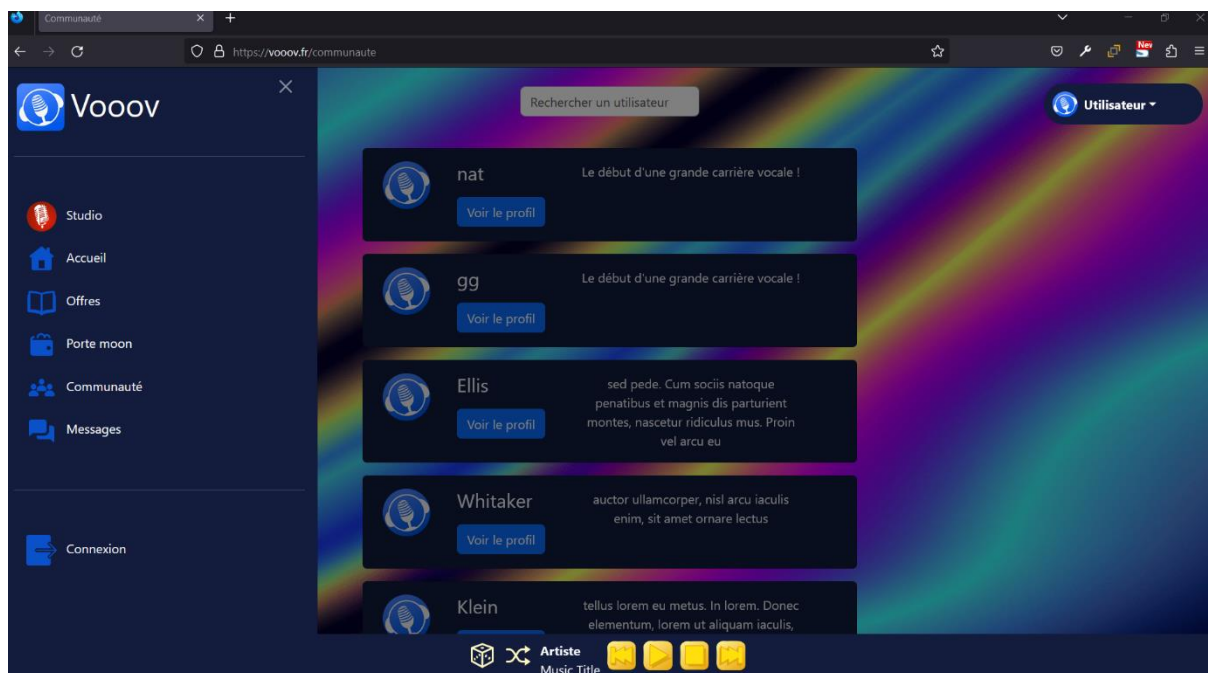
Affichage des détails d'un enregistrement : Permet de voir le prix en moons de téléchargement de l'enregistrement, de donner des moons pour l'enregistrement, d'envoyer un message à l'utilisateur propriétaire de l'enregistrement et de voir son profil.



Page de connexion : Permet à l'utilisateur de se connecter, pour avoir accès aux fonctionnalités de partage avec les autres utilisateurs et accès au studio.

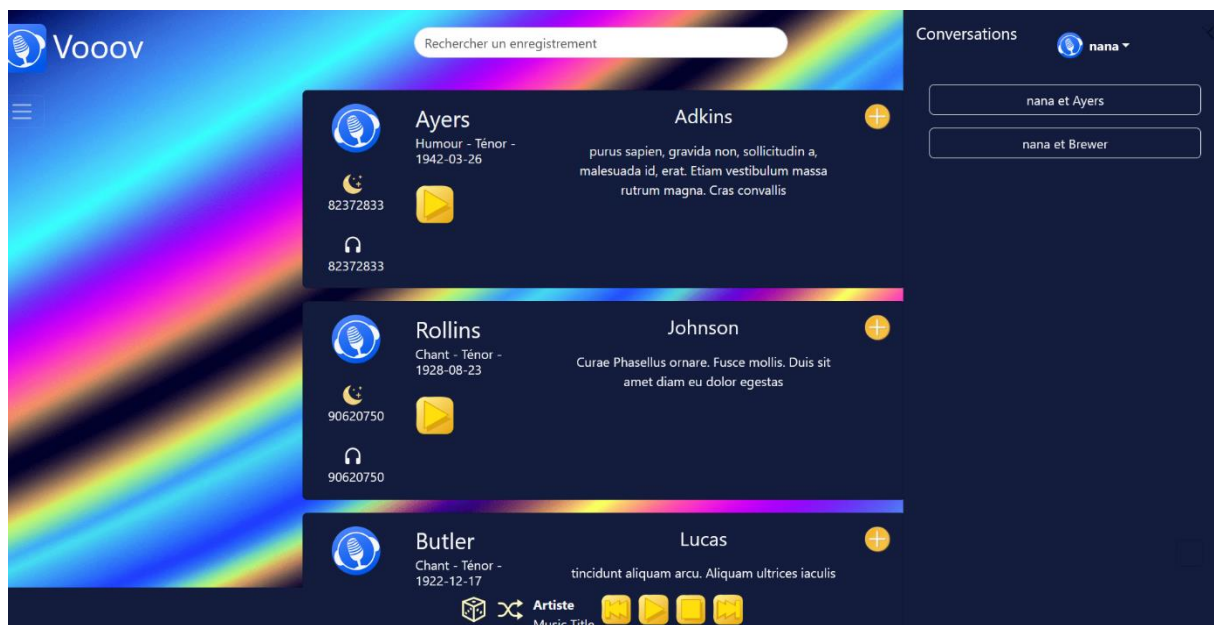


Page de recherche d'utilisateurs et menu ouvert : possibilité de rechercher un utilisateur et d'accéder à son profil. Menu latéral permettant de se rendre sur les différentes pages du site, en fonction de la connexion ou non de l'utilisateur.

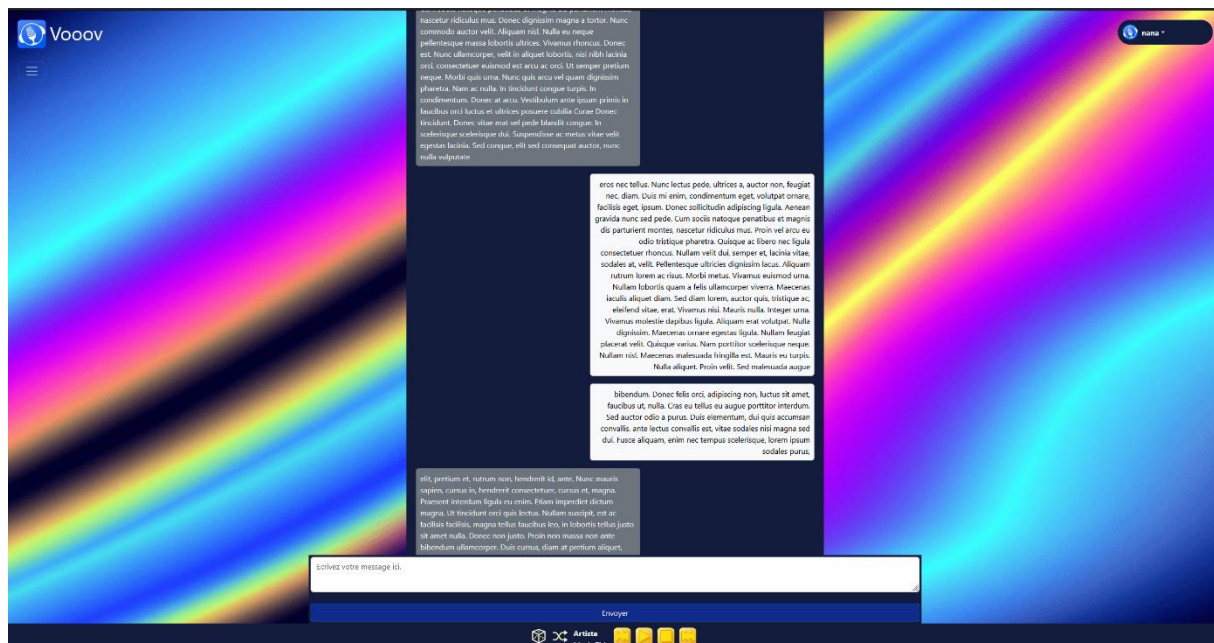




Menu latéral droit messagerie : Permet à l'utilisateur de visualiser l'ensemble de ses conversations

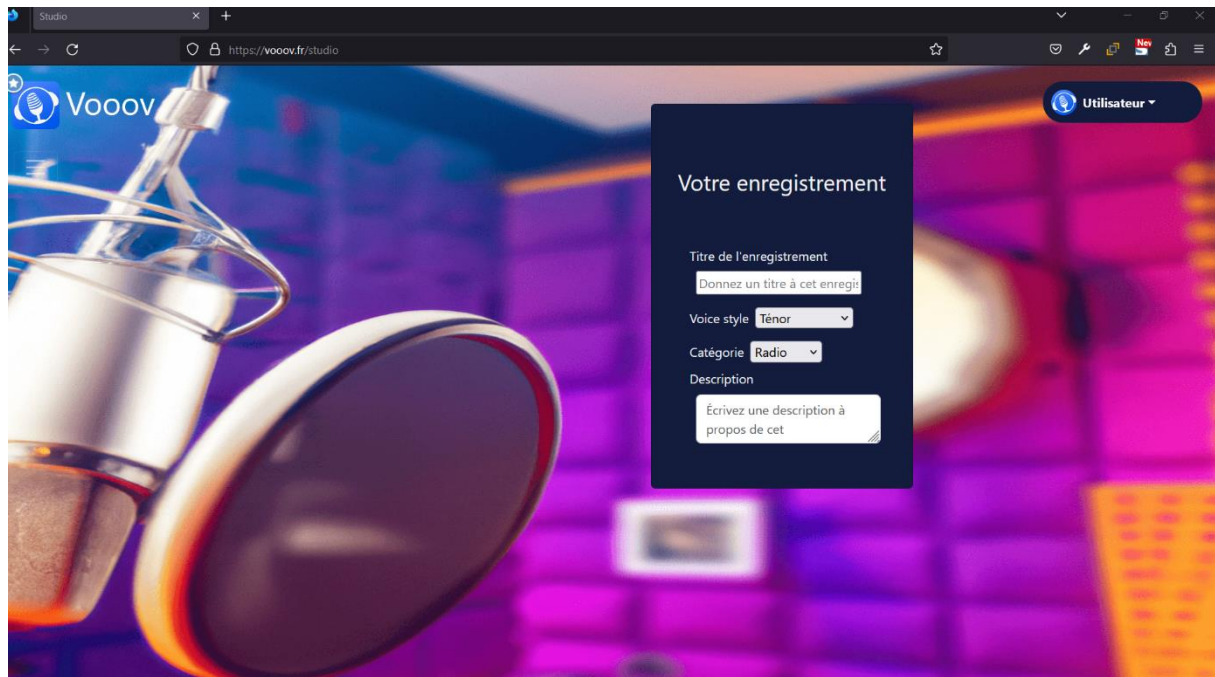


Conversation : Permet à l'utilisateur d'accéder aux messages envoyés avec un autre utilisateur.  
Permet également d'envoyer de nouveaux messages.

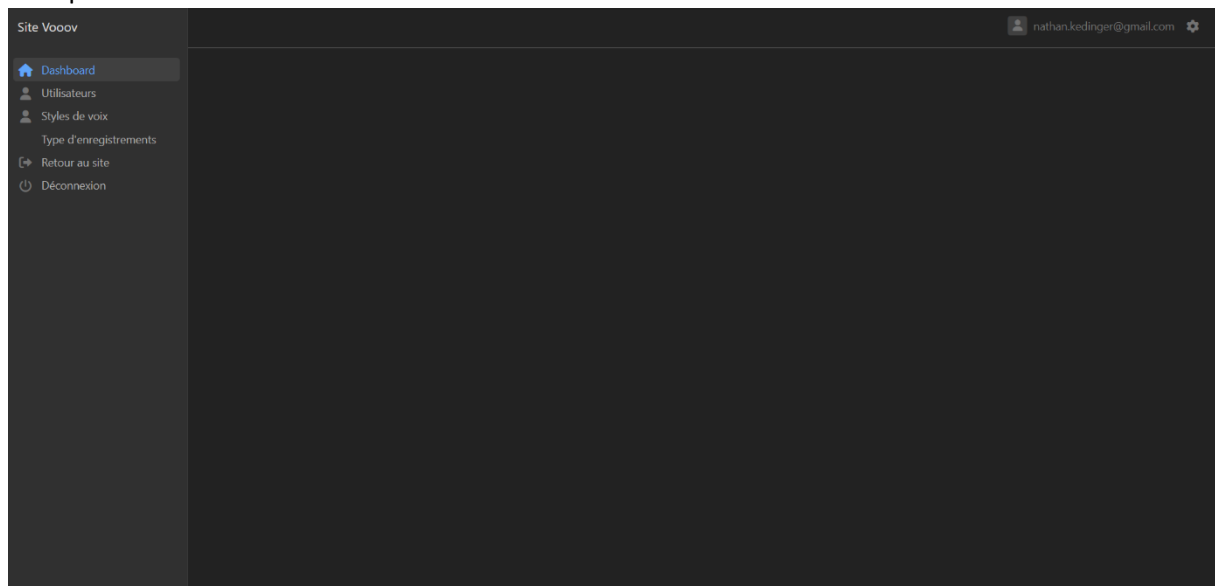




Studio d'enregistrement : Permet de créer des enregistrements audios et de donner du détail sur les enregistrements. (Permet seulement d'enregistrer les informations pour l'instant)



Back-office : Ce back-office permet à l'utilisateur ayant des droits administrateurs de modifier les entités utilisateurs, type de voix et types d'enregistrement, directement depuis le back-office, sans manipuler manuellement la base de données.



Fin du second dossier

# Post projet

## Evolutions et réflexions au cours du projet

Lors de l'utilisation d'un ORM il semble plus cohérent de commencer par la création de la base de données via l'ORM que manuellement. Dans le cas contraire, il est nécessaire de parfaitement connaître les réactions de l'ORM.

L'API est intégrée directement au code source du site internet. Par mesure de sécurité Les fichiers de l'API doivent encore être d'avantage centralisés. De plus, ils doivent de préférence passer par l'indexation de Symfony. Cela permettra de filtrer encore d'avantage les accès. Ils sont pour le moment accessibles en Public ce qui constitue une faille de sécurité très importante.

## Fonctionnalités restant à développer :

### Application Web :

- Mettre en place les commentaires sur les enregistrements.
- Mettre en place l'enregistrement et la lecture audio.
- Intégrer les moons et portes-moons.
- Implémenter un moyen d'ajouts de moons : paiement ? publicités ?...

### Application mobile Android :

- Page de chargement d'accueil (cacher les latences de chargement des datas)
- Bouton de suppression de compte
- Création de l'espace setting pour modifier un compte
- Modification des images de profil utilisateur
- Gestion des notifications
- Notification de nouveau message
- Modifier le format des dates
- Permettre le téléchargement de fichiers audio de source extérieur de manière sécurisé
- Régler les bugs (Connexion avec mail au lieu d'Uuid, actualisation du fragment bas au clique sur un enregistrement dans la liste...)
- Donner un accès au téléchargement d'un record en contrepartie de moons
- Gestion des relations (amis, followers...)
- Permettre de rendre des record privé, public...
- Implémenter un moyen d'ajouts de moons : paiement ? publicités ?...
- Amélioration générale de la lecture des enregistrements, du défilement, de l'algorithme d'aléatoire...
- Centraliser les CRUD complètement

### API :

- Extraire l'API du dossier public et l'intégrer dans le fonctionnement de Symfony

Serveur :

- Sécuriser le serveur : Précision du Firewall, changement des accès ssh.

Design :

- Travailler sur le détail général du design
- Amélioration des UI
- Amélioration des boutons

Et encore beaucoup d'autres améliorations...

Liste des difficultés rencontrées