Dossier de projet : développeur web & web mobile

# CHUU CHAT

Laura Savickaite

Coding School 2

La Plateforme

# Table des matières

**No table of contents entries found.**

# Présentation du projet

Dans le cadre de notre scolarité et de l'obtention du titre de concepteur/développeur d'application web, nous devions réaliser une application mobile de messagerie instantanée, communément appelée chat.

Depuis quelques années déjà, un genre particulier de musique s'est popularisé, ramenant avec lui de nombreux admirateurs tout autour du globe. En effet, la Kpop (musique pop coréenne) faisant écho à la Hallyu Wave (vague coréenne) s'est répandue au-delà de ses frontières géographiques. Basé sur un principe d'idols et de groupes, il y a un véritable aspect social derrière tout ceci : entre fan wars ("guerre de groupes" menés par leurs fans) et partage de la musique, c'est cet aspect "liant" que nous avons voulu exploiter, justifiant ainsi la création d'un chat.

L'intention de Chuu était de réaliser une plateforme qui regrouperait cette communauté, les Kpop fans (aussi nommés stans), et lui permettrait de communiquer tout en continuant de s'épanouir.

Le chat est divisé en plusieurs "rooms" correspondant à des groupes de Kpop ainsi qu'une room globale qui servirait de levier à l'intégration. L'utilisateur peut choisir dans quelles rooms il veut s'inscrire, et ainsi discuter de son groupe favoris sans entraîner de disputes, ce qui peut arriver sur d'autres plateformes comme Twitter par exemple. Les perturbateurs sont bannis d'une room par l'administrateur pour éviter le phénomène de trolls connu sur internet.

Pour ce faire nous avons utilisé Node JS pour la construction du back ainsi qu'une base de données en sql. Concernant le front, c'est avec React Native que nous l'avons réalisé avec l'aide de Socket.io pour le côté instantané du chat.

# Présentation du projet en anglais

For a couple of years now, a particular genre of music has become more and more popular amongst people, getting recognition from all over the globe. Indeed, Kpop (korean pop music) has played a huge part in the Hallyu Wave (a soft power concept in Korea) which pushed korean culture outside of its geographical borders. Based on idols and boygroups/girlgroups, we can see an underlying social aspect : between what are called fan wars and the shared love of music, we can refer to it as an almost virtual social gathering that we wanted to dive deep into. And thus, it inspired the creation of Chuu chat.

The intention behind Chuu was to create a platform where people with similar taste in music could meet and communicate with each other.

The chat is divided into different “rooms” corresponding to Kpop groups and a single “common” room that would help people to present and integrate themselves. The user can choose which rooms they want to get into in order to chat about their favorite group with other fans without creating any discord, which can usually be found in other platforms such as Twitter for example. The disruptors are banned from the room by the admin in order to prevent any “trolls”, a common phenomenon on the internet. So the goal was to keep the best in this culture without the fan wars (which consist of pitting groups one against the other) that can be really toxic and pointless at times.

When it comes to the technologies we used, we mainly focused on Javascript as it is a modern and flexible language. We used NodeJS for the back-end as well as a database in SQL. For the front-end development, we used React Native and Socket.io for the interactive, instant part of the chat.

# Compétences couvertes par le projet

Voici les compétences couvertes par le projet :

- maquetter une application
- développer des composants d'accès aux données
- développer la partie front-end d'une interface utilisateur web
- développer la partie back-end d'une interface utilisateur web
- concevoir une base de données
- mettre en place une base de données
- développer des composants dans le langage d'une base de données
- concevoir une application
- développer des composants métier
- construire une application organisée en couche
- développer une application mobile
- préparer et exécuter les plans de test d'une application
- préparer et exécuter le déploiement d'une application

# Description, organisation & cahier des charges

## Description de l'existant

Les fans de Kpop se réunissent généralement sur différentes plateformes telles que Twitter, Tumblr ou bien Discord, mais il n'y a pas d'application spécifique concernant le sujet avec des espaces réservés pour différents groupes.

## Les utilisateurs du projet

Le projet se distingue d'autres applications de messagerie par le fait qu'il s'adresse à un public non général mais spécifique. Il répond à un besoin social d'échanger, communiquer etc… sur un sujet, une passion, commun. Les principaux utilisateurs seraient donc des fans de Kpop - et ce, malgré le fait qu'il y ait "deux" parties.
En effet, la partie web répondra aux besoins d'un administrateur : créer une room, bannir un utilisateur d'une room... Tandis que la partie application mobile, elle, sera pour les simples utilisateurs : créer un compte, éditer son compte, poster un message...

## Les fonctionnalités du projet

L'application mobile contient plusieurs pages qui sont accessibles via une tab-bar - barre de navigation.

*Page de connexion* : Elle sert de page “d’accueil” - lorsque l’utilisateur lance l’application, il est invité à se connecter à l’aide d’un nom d’utilisateur et d’un mot de passe. S’il ne possède pas de compte sur le chat, il peut alors s’inscrire.
L’idée étant de hermétiser l’application dès le départ.

*Page d’inscription* : Sur celle-ci, l’utilisateur qui n’a pas de compte peut en créer un afin de participer à la vie de l’application. Il aura besoin d’un pseudo et d’un mot de passe (qu’il devra confirmer également).

*Page “feed”* : Une fois connecté, l’utilisateur est dirigé vers une page recensant l’ensemble des rooms auxquelles il appartient (ainsi que le dernier message envoyé dessus). Il y a également un onglet pour filtrer seulement la chatroom globale et la différencier des autres rooms plus “privées”. L’utilisateur peut cliquer sur une des rooms pour y voir les messages.

*Page messages* : Une fois cliqué sur une discussion, l’utilisateur va pouvoir voir tous les messages de la room. Il peut également poster un message lui-même.

*Page “find new rooms”* : Il existe également une page permettant à l’utilisateur de voir toutes les rooms dans lesquelles il n’appartient pas encore. Il peut ainsi en cliquant dessus s’y inscrire s’il le souhaite. S’il est banni d’une room, il ne peut pas s’y inscrire.

*Page profil* : Sur celle-ci, l’utilisateur peut voir ses informations personnelles. Il peut changer son avatar, son pseudo et son mot de passe.

L’application web est exclusivement accessible par l’administrateur afin de pouvoir administrer l’application et ses utilisateurs.

*Page index* : Une simple page introductrice.

*Page users* : L'admin peut updater les rôles des utilisateurs (donc les bannir) et leur pseudo.

*Page rooms* : L'admin peut supprimer une room et en créer une et updater un nom.

Nous avons décidé que l'administrateur n'ait pas d'impact sur les messages - ils reposent sur la responsabilité de l'utilisateur.

# Conception technique du projet

## Choix de langages de développement

Le langage majeur qui a influencé notre projet a été le Javascript. Moderne et flexible, il nous a permis de construire un back-end ainsi qu'un front-end avec les bons frameworks.
Le choix a été relativement rapide pour plusieurs raisons : aucun site qui se veut moderne et dynamique n'échappe à Javascript - cela nous permettait de monter en compétence sur un langage qui, sur le plan professionnel, n'est plus une option. Parce qu'il a également plusieurs ressources et fonctionnalités natives, Javascript semblait plus intéressant à utiliser.

De plus, personnellement, c'est un langage que j'ai beaucoup apprécié et travaillé durant mon année d'alternance; il me semblait donc logique d'avoir une certaine continuité dans mon apprentissage.

## Technologie back-end

Suivant cette logique, nous avons décidé d'utiliser NodeJS comme environnement de développement pour la création de notre API.
Pour framework, on a choisi ExpressJS. C'est le framework le plus populaire et, étant libre d'accès, il permet d'être plus appréhendable avec à disposition plusieurs outils et fonctionnalités. Il a été un support avec notre contrainte temporelle mais également une surcouche gardant les performances optimales et relativement rapides. Nous avions également rajouté des dépendances telles que bcrypt, socket.io etc…

## Technologie front-end

Aussi, notre framework de choix pour le front a été React Native - ayant utilisé du React dans nos alternances respectives, il était intéressant de tester React Native. Il permet également de développer un code qui serait lu aussi bien par les IOS (évitant ainsi de se former sur du Swift par exemple) que les Android.


# Environnement de travail

D'autres outils ont été employés pour la réalisation de ce projet, parmi eux :

- Visual Studio Code (VSCode) comme éditeur de code.
- Thunder Client, une extension REST API pour VSCode.
- Github pour le versioning.
- NPM pour l'installation des "packets" et Nodemon pour le lancement du serveur.
- Trello pour l'organisation de nos tickets. C'est un outil collaboratif.
- LucidChart pour le maquettage de nos bases de données.
- Figma pour les maquettes graphiques (maquettes low-fidelity et high-fidelity, ainsi que la charte graphique) de l'application. C'est un outil collaboratif.
- Expo pour la simulation de notre application sur nos téléphones mobiles.

# Organisation

Le projet étant réalisé de manière “saccadée” entre des semaines en alternance et en groupe, il était nécessaire de trouver une organisation qui puisse être non seulement maintenue mais qui fasse également sens pour tout le monde.

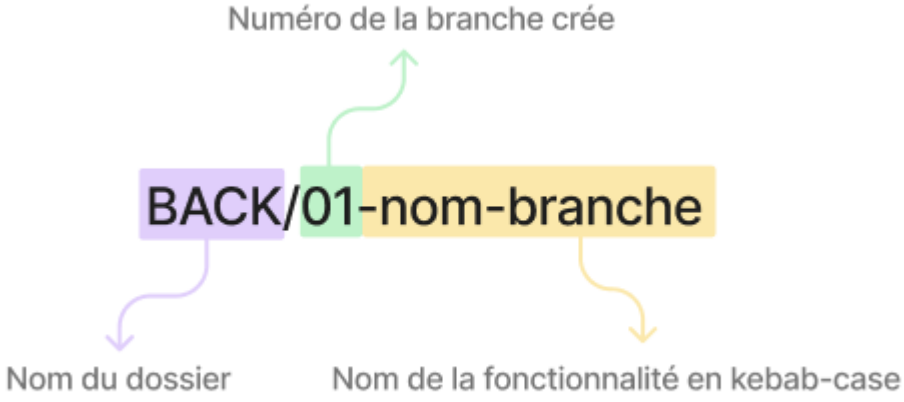
Il existe plusieurs méthodes de conception de logiciels comme par exemple l’UML (‘unifying modeling language’) qui peut être utilisé pour modéliser les architectures logicielles, les processus métier, les interactions entre les composants, etc. L'un des avantages de l'utilisation d'UML est qu'il est un langage standardisé, ce qui signifie qu'il est compréhensible pour un large éventail de professionnels de l'informatique. Cependant, sa complexité peut également être un inconvénient, car il peut être difficile à apprendre et à mettre en œuvre pour les petites équipes de développement. Ou bien encore, le développement en cascade qui est une approche séquentielle, dans laquelle les activités de conception, de développement, de test et de maintenance sont organisées en phases. Chaque phase doit être achevée avant de passer à la suivante.

Pour notre projet, nous avons décidé d’essayer d’implémenter la méthode Agile SCRUM puisque cette dernière était déjà principalement appliquée dans les alternances respectives. C’est un cadre de gestion de projet qui divise les projets en plusieurs phases dynamiques appelées “sprints”. Après chaque sprint, on parlait des éléments qui pouvaient être améliorés pour le prochain, les éléments bloquants et les éléments ralentisseurs. On discutait également de ce qu’il faudrait faire pour le sprint suivant - notamment s’occuper de la priorisation des différentes tâches.
Le but était d’avoir une certaine flexibilité sur le projet afin qu’il s’adapte aux différents besoins, aux potentiels changements et à l’équipe. Il y eut plusieurs réunions où par exemple, une envie qu’on avait au départ n’était plus faisable au niveau du temps imparti - dans ce cas, on devait réadapter notre planning et nos tickets.

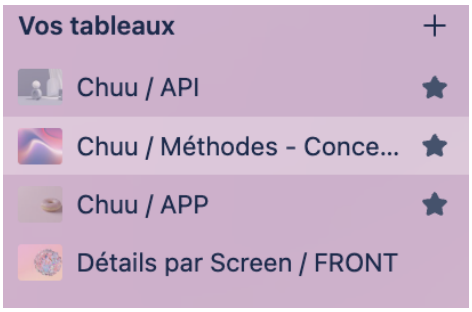
Nous avions également mis en place des “daily” : des petites réunions qui ne devaient pas dépasser quinze minutes où chacun parlait de ce qu’il avait fait la veille et ce qu’il

planifiait de faire dans la journée. Cela permettait d'avoir une vue d'ensemble sur la progression du projet et relever les potentiels problèmes sur lesquels un développeur pouvait coincer. Cela construisait également une compréhension commune du produit avec un partage d'informations.

- **GIT** : Afin d'éviter le plus possible de potentiels merge conflicts problématiques, nous avons décidé de mettre en place une gestion spécifique des branches. En effet, nous avions une branche par ticket. Une fois une fonctionnalité complètement développée, on "mergeait" la branche correspondante vers une branche de "pré-production" - il y a eu plusieurs branches "pré-production" qui concordent avec le numéro du sprint en cours. Nous avons essayé de développer le projet sous des "sprints" afin d'avoir une certaine agilité - il a été néanmoins un peu compliqué pour nous de déterminer les fins des sprints et certains bavaient sur le temps des autres. Enfin nous avions une branche dite de production (la branche "main") qui recevait principalement les merges considérés comme "finaux" - en d'autres termes, lorsque les différents merges sur un sprint fonctionnaient ensemble, on "mergeait" sur main.
  Il y avait une nomenclature spécifique pour les branches ainsi que pour les commits.
  Branches : il devait y avoir la notion *FRONT* ou *BACK - le numéro du ticket - nom donné à la fonctionnalité/au fix.*
  Commit : *[FRONT* ou *BACK/numéro du ticket] ce qui a été fait dans le commit.*




**Trello** : Le tableau Trello était divisé en trois parties : une “centralisante” qui contenait des informations importantes quant au développement commun (des notions de knowledge sur le projet - par exemple, si quelqu’un changeait quelque-chose qui pourrait éventuellement influencer les autres, il fallait le marquer dans ce tableau). Ensuite, on avait deux tableaux : un pour le back, un autre pour le front. Les deux étaient organisés de la même sorte : il y avait différentes catégories suivant les étapes de développement de chaque ticket. A sa création, un ticket se trouvait dans la catégorie “à faire”, il possédait un numéro spécifique ainsi qu’un nom (dans le back, la couleur du ticket était différente selon l’accessibilité de la route). Ensuite quand il était pris, le développeur se l’assignait, rajoutait son prénom dans le titre et le déplaçait dans la catégorie “en cours”. Quand il était “mergé” sur un sprint, il était alors passé en “recette”. Une fois testé, il pouvait enfin finir son cycle de vie dans les “terminés”.



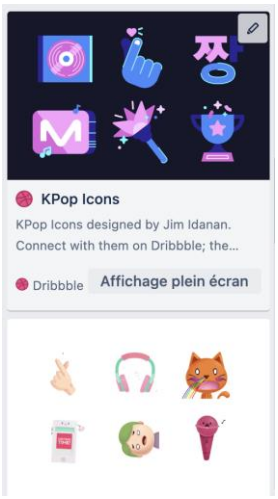


# Conception visuelle du projet

De par sa vocation, on voulait pour Chuu chat une présence visuelle assez dynamique et reflétant la jeunesse. L'inspiration première était les moyens de communications des générations plus jeunes : les emojis. En effet, ces derniers sont un bon moyen de refléter une certaine émotion et même la convier à l'interlocuteur - en créant/personnalisant nos propres emojis, on voulait inspirer une certaine familiarité : un endroit "fun", amical et où il fait bon d'être. Il y a également des inspirations du pixel art et des esthétismes des années 90/début 2000 qui semblent refaire surface depuis quelques années.

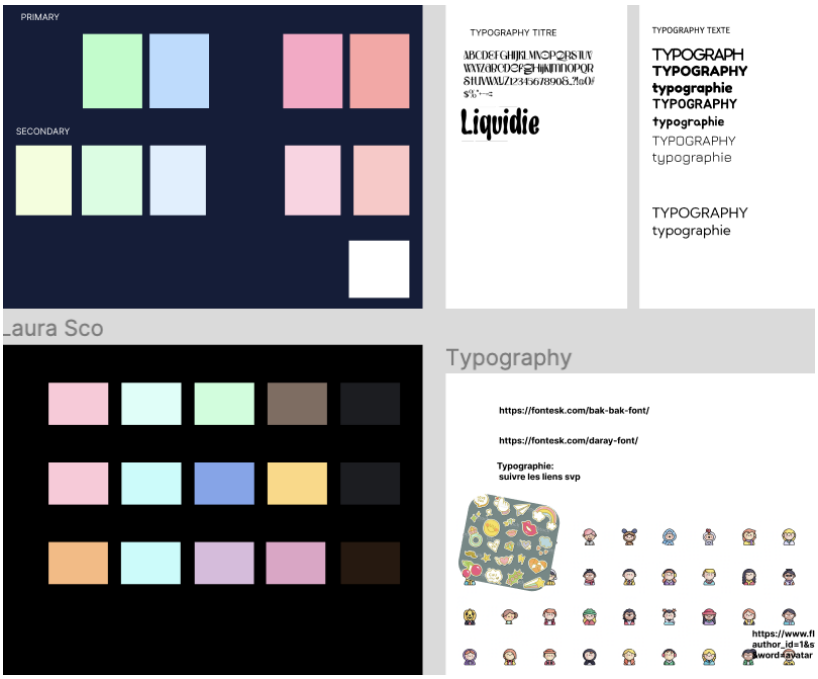

## Moodboard

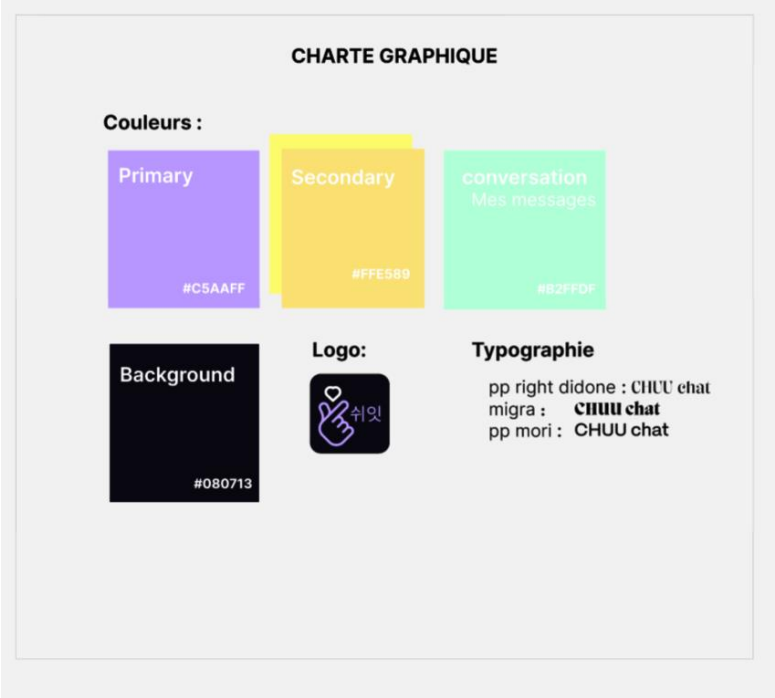
Pour guider la conception visuelle et trouver des inspirations, nous avions réalisé une petite collection d'images - un moodboard.




## Charte graphique

Nous avons également réalisé une charte graphique pour nous guider dans la conception des maquettes. Nous y avons répertorié les couleurs que nous allions utiliser mais également les inspirations typographiques.





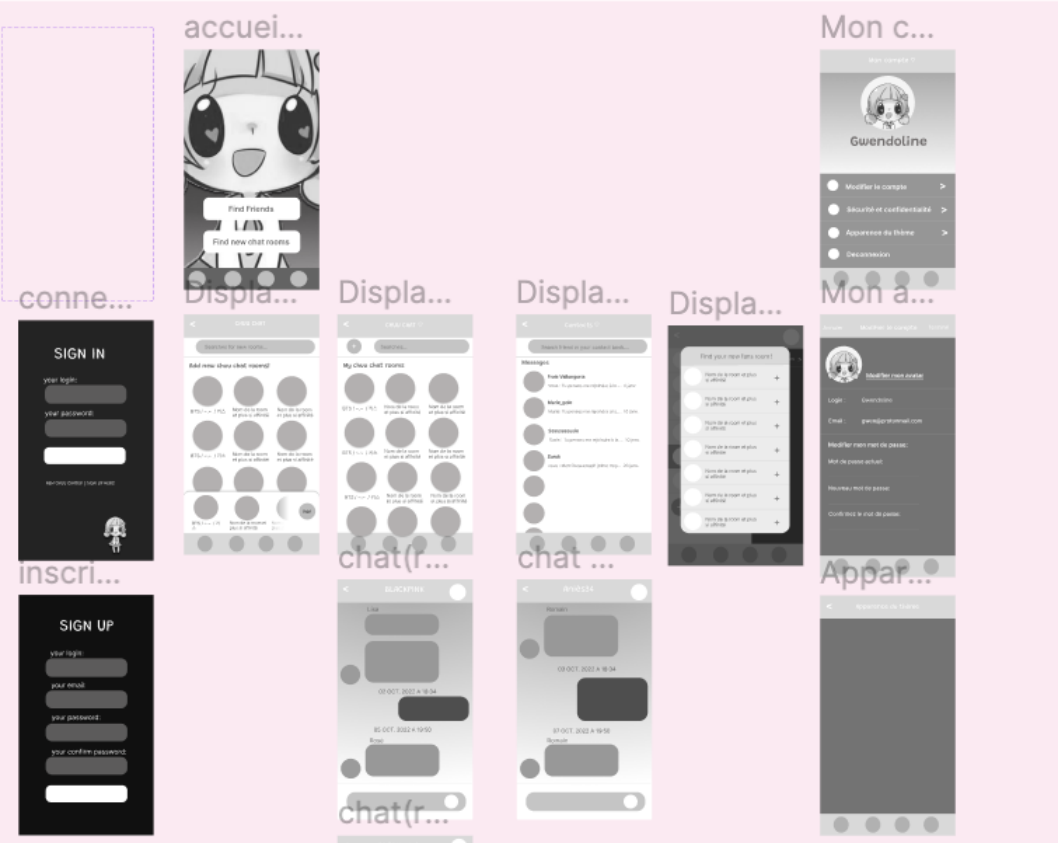

**Wireframe**

Avec toutes ces données récoltées, nous devions former le “squelette” de notre application, c’est-à-dire le wireframe aussi dit la maquette “low fidelity” (“low-fi”). Il permet la visualisation complète de la structure de l’application. Il ne doit pas être détaillé mais doit comprendre l’ensemble des éléments, leur agencement et l’articulation des uns avec les autres.

C’est une étape importante de la conception UX (user experience) qui permet de réfléchir sur le côté instinctif de l’application et de son accessibilité. Ainsi que d’avoir une première vue d'ensemble sur les différents composants à réaliser, ceux qui seront réutilisables par d’autres plus grands etc…c’est utile si l’on code en React.

Nous l’avons réalisé avec Figma mais il est tout à fait possible de le faire à l’aide d’un crayon sur une feuille de papier.

Étant une application mobile, la conception a été, de manière logique, mobile first (ça aurait été le cas également si l’application pouvait être web).




**Maquette high-fidelity**

Lorsque la maquette low-fidelity est validée, la réalisation de la maquette high-fidelity peut commencer. Cette étape consiste à merger l'UI (user interaction) et l'UX ensemble - afin que cette maquette puisse ressembler le plus possible au produit final. Elle permet un appuie aux développeurs et un guide notamment pour la partie "front".

La maquette finale devrait également comprendre un design system graphique. Ce dernier a pour objectif de réunir tous les éléments visuels, graphiques etc (couleurs, typographie, espacements entre les éléments…) dans un même système standardisé. Ce dernier peut, bien évidemment, évoluer mais il permet aux développeurs de construire des normes stylistiques à réutiliser dans tout le site/l'application afin d'avoir une homogénéité. Par exemple à l'aide de la mise en place de mixins.
Nous les avons réalisés à l'aide de Figma.

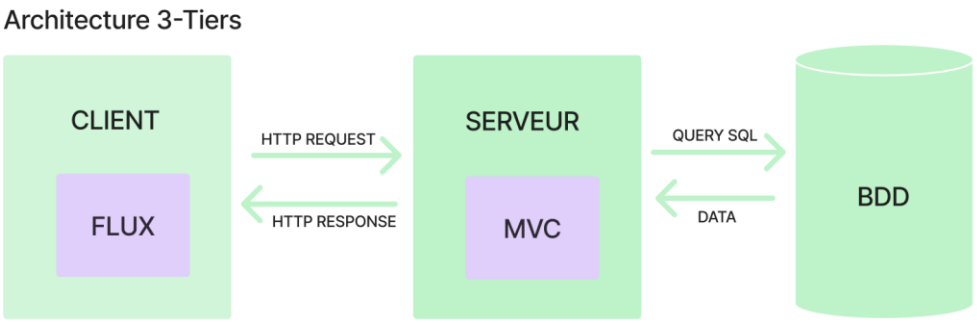

# Conception serveur/back du projet

La conception du back et notamment de notre système d'information a été réalisée en suivant la méthode Merise. Cette dernière est basée sur la séparation des données et traitements à effectuer en plusieurs modèles.

Voici un aperçu des données jugées comme pertinentes pour la représentation des besoins et donc la construction de notre base de données :

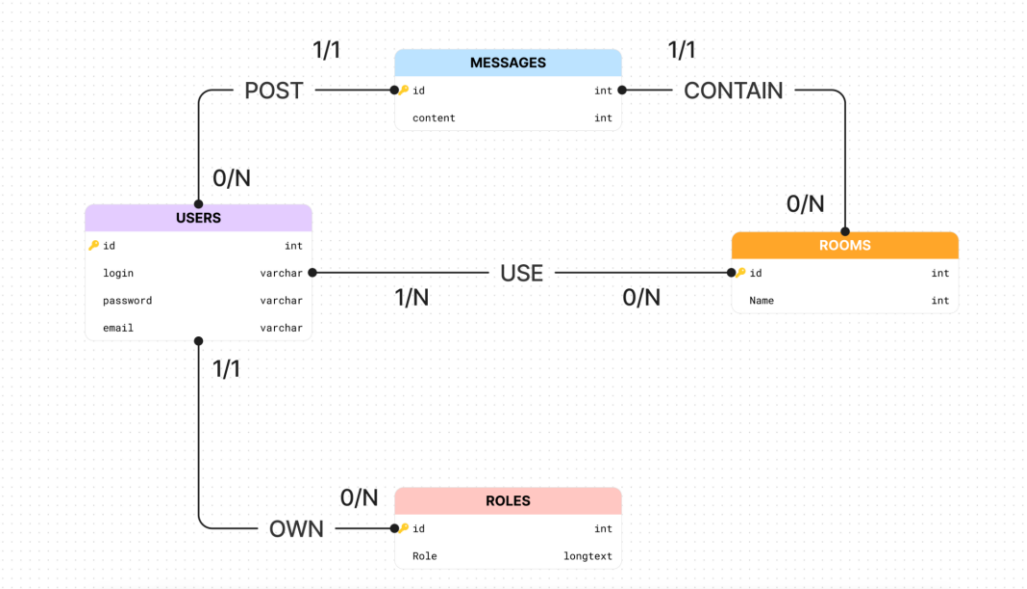
- les *Users* : ils sont le cœur de notre application, un utilisateur peut s'inscrire et donc se connecter. Il a un login, un mot de passe, un rôle et une liste de groupchats auxquels il participe.
- les *Groupchats* : ils doivent pouvoir recevoir et distribuer les messages des utilisateurs.
- les *Messages* : appartiennent à un.e utilisateur/trice, ont une date d'envoi et un contenu.
- les *Rôles*

Nous avons également puisé une inspiration pour la construction de l'API dans le modèle client-serveur. Cette méthode implique la séparation de l'application en deux parties distinctes - le client et le serveur. Le client communique avec le serveur pour envoyer et recevoir des messages. Le serveur fournit des ressources et des services à plusieurs clients, qui peuvent être situés sur des ordinateurs différents. Les clients communiquent avec le serveur en utilisant un protocole standard, tel que HTTP. Le serveur gère les données, l'authentification, la sécurité et les transactions. Le client gère l'interface utilisateur et l'affichage des données.
Cependant nous avons décidé de faire une architecture 3-tiers, toujours basée sur cette inspiration mais en ajoutant le facteur base de données. Le serveur se charge du traitement de la donnée récupérée de la bdd et également des requêtes du côté client.




1) **MCD**

Le MCD (modèle conceptuel de données) permet une représentation rapide de notre système d'information. D'un seul coup d'œil, il est possible de voir quelles sont les possibles futures tables majeures car en effet, le MCD se compose des informations principales (il n'y a donc pas de tables de liaison) donc d'entités et de leur relation entre elles grâce à des cardinalités et un verbe à l'infinitif traduisant ces liens.

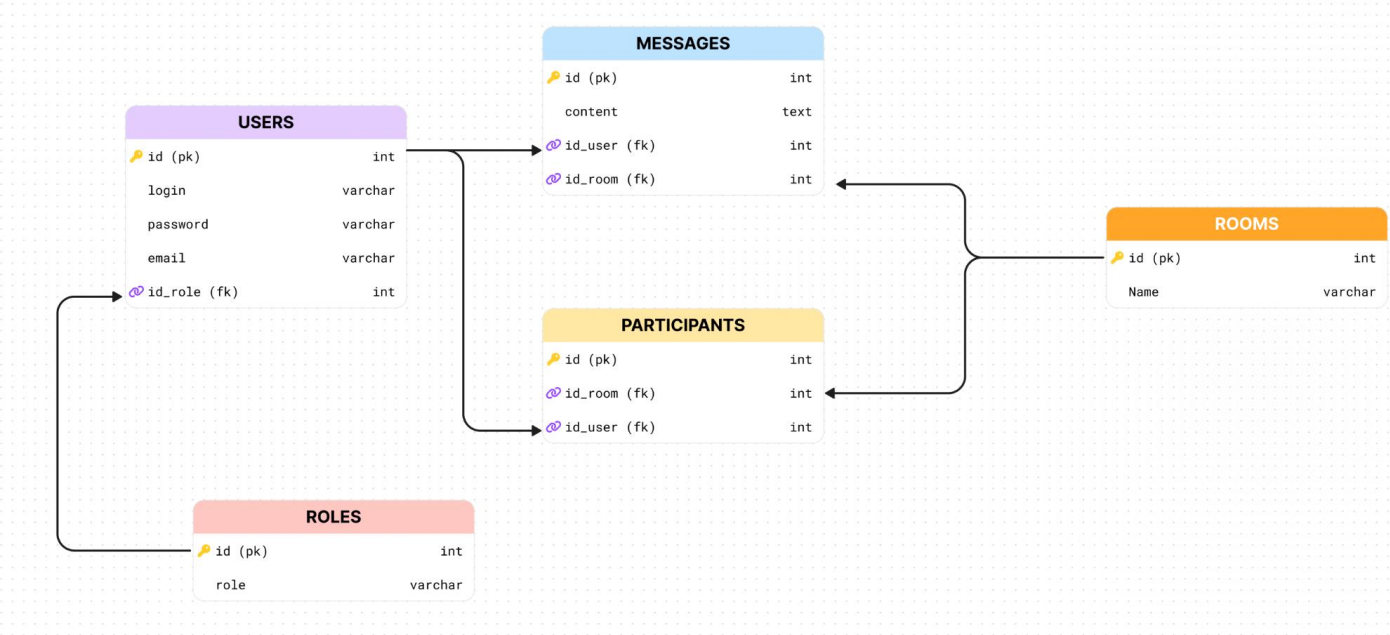



Dans le modèle ci-dessus, un utilisateur peut utiliser 0 à plusieurs chatrooms tandis qu'une chatroom peut exister sans utilisateurs comme avec plusieurs. Un

utilisateur n'est pas obligé de poster un message comme il peut en poster plusieurs tandis qu'un message est obligatoirement et uniquement posté par un utilisateur etc..

### 2) **MLD**

Le MLD (modèle logique des données) précise davantage le MCD. Les entités deviennent tables et d'autres sont ajoutées : les tables de liaison avec leur foreign keys (une clé correspondant à la primary key d'une autre table). Les primary keys qui identifient les tables sont également ajoutés. Le MLD est voué à évoluer au fur et à mesure du projet mais il représente déjà globalement l'organisation des tables, c'est-à-dire que c'est à ce stade qu'on peut avoir un aperçu général de notre base de données : combien de tables elle aura, quelles sont les principales etc…

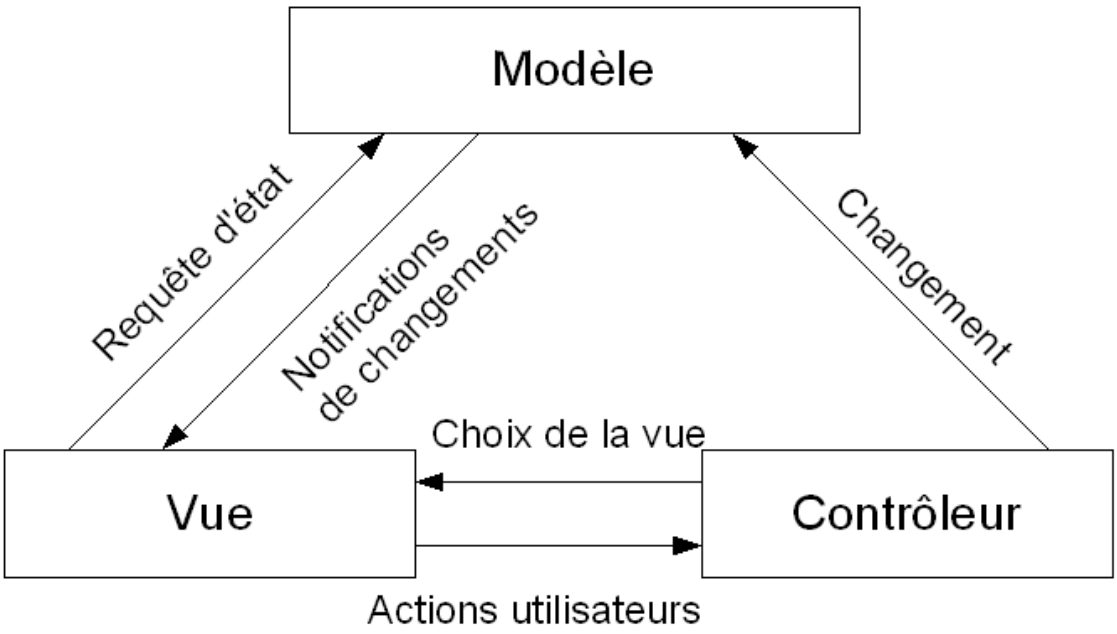



Dans le modèle ci-dessus, nous avons mis en place une table de liaison entre les utilisateurs et les chatrooms sous le nom de participants (la relation étant de nature N/N) avec les id des rooms mais également ceux des utilisateurs.

Un MPD (modèle physique de données) aurait pu également être ajouté, seulement nous ne l'avons pas réalisé. C'est le modèle final lorsqu'on est certain que nos tables ne changeront plus. Il est également celui qui est présenté.

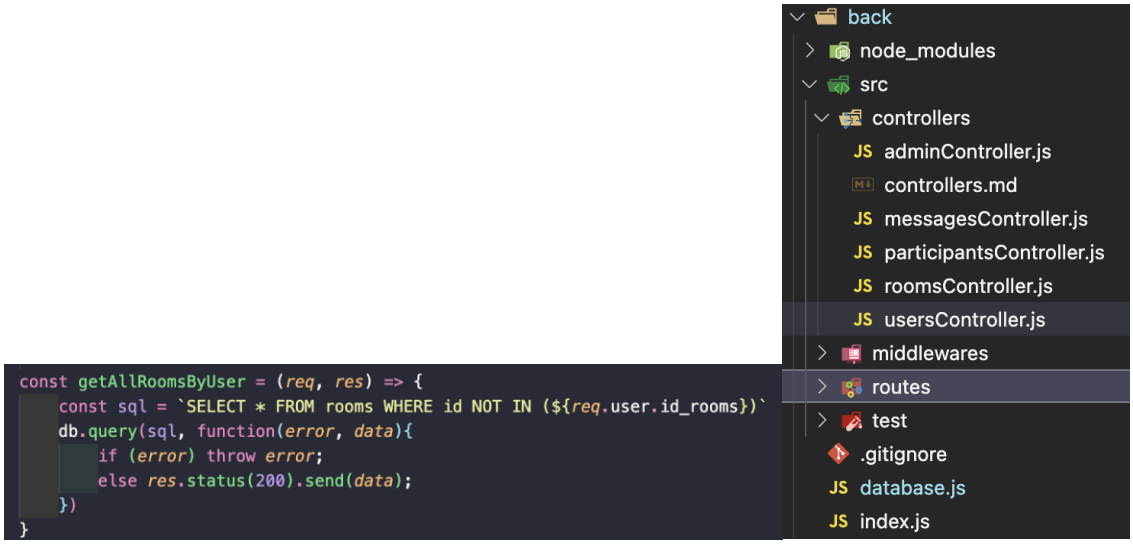
# MVC

Le MVC (model, view, controller) est un design pattern permettant de structurer la manière de coder. Le modèle (côté serveur) détient les données, il permet de les récupérer de la base de données et de les gérer afin d'être envoyés vers le controller. Le controller, lui, traite ces données et les manipulent, servant ainsi d'intermédiaire entre le modèle et la view. Il fait référence à la logique du code. Une fois ces données maniées, le controler peut laisser place à la view (côté client) qui va présenter ces données à l'utilisateur.




Source : https://rosedienglab.defarsci.org/

Nous n'avons pas suivi ce design à la lettre. En effet, alors que nous avons bien un controller, nous avons trouvé cela plus facile de ne pas le séparer du model et donc avoir un seul dossier. La view est, quant à elle, gérée par le front.




Avoir un model et un controller, aurait signifié découper cette petite fonction et en extraire la requête sql pour la placer dans le model. Cela nous semblait un peu moins pratique pour une API de notre gabarit - nous avons donc préféré tout garder dans un même fichier.

# API REST

Basé sur le principe du protocole HTTP (hypertext transfer protocol), qui est un protocole de communication client-serveur basé sur une architecture de type requête (client)/réponse (serveur), une API REST (Representational State Transfer) permet à différents systèmes de communiquer entre eux en utilisant des méthodes HTTP standardisées comme GET (récupérer une donnée), POST (enregistrer une donnée), PUT (mettre à jour l'intégralité des informations d'une donnée), PATCH (mettre à jour partiellement une donnée) et DELETE (supprimer une donnée). Les données échangées entre les systèmes sont souvent formatées en JSON ou en XML. Les API REST sont généralement considérées comme plus légères et plus flexibles que les API SOAP.

Une API SOAP (Simple Object Access Protocol) est une API basée sur un protocole de transport plus lourd que HTTP, souvent basé sur XML et nécessitant des en-têtes

spécifiques. Elles sont souvent utilisées dans des environnements d'entreprise où la sécurité et la fiabilité sont primordiales.

Notre back était donc composé des dossiers suivants :

- node modules
- src
    - controlers
    - middlewares
    - routes

# Conception front du projet

Le projet étant réalisé en React, nous avons voulu utiliser les naturelles dispositions du langage pour moduler notre front.

## Composants

Une architecture basée sur les composants a été réalisée, puisque particulièrement populaire avec React, afin de créer des composants indépendants et réutilisables.

Les caractéristiques principales étaient d'avoir des composants qui puissent être :

- autonomes : une modification apportée à un composant n'influence pas ses compères.
- réutilisables

- remplaçables : par d'autres composants plus fonctionnels par exemple, dans le futur de l'application.
- maintenables

Chaque composant devait réaliser une tâche uniquement. Quelques composants, qu'on appelait composants génériques, étaient également codés pour servir d'autres composants (exemple, des boutons etc…).
Dans React Native, les architectures basées sur les composants sont généralement organisées en utilisant des dossiers pour regrouper les fichiers liés à chaque composant. Les dossiers peuvent contenir des fichiers de code source, des fichiers de test, des fichiers de style et des fichiers de documentation pour chaque composant.
Dans notre cas, nous avions le code source ainsi que son style dans le même fichier.

**Expo**

Afin de simuler notre application, nous avons utilisé Expo. Il suffisait d'être connecté sur le même réseau, à la fois sur le mobile et l'ordinateur, pour que cela fonctionne.

# Fonctionnement de l'API

Le fonctionnement de l'API est basé sur une architecture de type requête/réponse. Le client envoie une requête à mon API via une url, dite requête HTTP (composées d'une ligne de commande, d'en-têtes et d'un corps de message) - cette dernière est analysée par le routeur qui répertorie toutes les requêtes possibles. En fonction de la route, un model/controler est appelé qui se chargera de récupérer ou d'envoyer une donnée.
Le serveur renvoie une réponse sous forme de JSON et également un statut.

Les différents statuts utilisés dans ce projet sont :
- 200 : OK
Indique que la requête a réussi
- 201 : CREATED
Indique que la requête a réussi et une ressource a été créé
- 400 : BAD REQUEST
Indique que le serveur ne peux pas comprendre la requête a cause d'une mauvaise syntaxe
- 401 : UNAUTHORIZED
Indique que la requête n'a pas été effectuée car il manque des informations d'authentification
- 403 : FORBIDDEN
Indique que le serveur a compris la requête mais ne l'autorise pas
- 404 : NOT FOUND
Indique que le serveur n'a pas trouvé la ressource demandée
- 500 : INTERNAL SERVER ERROR
Indique que le serveur a rencontré un problème.


**Routing**

Le routeur utilisé est celui d'express. Il se forme simplement d'un URI et d'une requête (get, post, etc) : "app.requête(route, fonction)". "App" est une instance d'express, la "requête" est la requête http, la route se présente sous forme d'uri et la fonction est le controler qui traite la donnée et renvoie une réponse au client.

```
//Users route
app.use('/users', users);

//Participants route
app.use('/participants', participants);

// Verify route
app.use('/connected', signIn, users )

//Admin route
app.use('/admin', [signIn, isAdmin], admin)

//Message route
app.use('/chat', chat);

//Rooms route
app.use('/rooms', rooms);
```

Nous avons dans un premier temps, un index.js qui recense l'ensemble des routes principales. La première ligne indique donc à l'application que pour toutes les routes ayant un URI commençant par "/users", il faudra aller chercher la suite dans le dossier users.

```
var users = require('./src/routes/users');
```

Ce dernier comporte l'entièreté des routes, requêtes possibles, offertes pour la partie "users". Cela nous évitait également de devoir préciser /user à chaque route users.

```
//[BACK/03-get-all-users]: Une route qui
router.get('/', getUsers);
```

Cette requête commence par une instance du routeur d'express que nous appelons plus haut dans le fichier.

```
var {
    registerUsers,
    authUsers,
    connectedUser,
    addUserToRoom,
    getUsers,
    getUserDetails,
    getUserRole,
    updateUser,
    refreshToken,
    getAllFromUsers
} = require('../controllers/usersController')

var router = express.Router();
```

La route permet donc de “get”, d’obtenir des données dès qu’on demandait /users (ici tous les utilisateurs). Lorsque cette requête est lancée, le routeur va trouver le controller correspondant avec la fonction indiquée, ici getUsers.

Il est important de noter que la manière dont une requête est réalisée de la sorte : http://localhost:3000/admin/users/1/update/role
Nous avons d’abord le port et l’URI qui suit.

```
const getUsers = (req, res) => {
    const sql = 'SELECT `login` FROM users'
    db.query(sql, function (error, data) {
        if (error) throw error;
        else res.send(data);
    })
}
```

**Middleware**

Le middleware est un logiciel qui permet une connectivité entre au moins deux applications ou des composants d'application. Dans notre projet, il prend la forme d’une fonction qui se dote également d’un rôle précis, fournit un service précis. Se situant entre deux couches logicielles, dans ExpressJS on le retrouve généralement entre la requête et la réponse. De base, le framework possède quelques middleware mais il est possible d’en créer au besoin de l’application. D’ordre général, il peut configurer et contrôler les connexions et les intégrations, gérer le trafic de manière dynamique via des systèmes distribués et sécuriser les connexions et le transfert de données. C’est principalement cette dernière fonctionnalité qu’on a utilisé pour notre projet.

Un middleware peut être appliqué à une unique route ou à plusieurs.

```
//[BACK/07] get the details from 1 user
router.get('/details/:userId', signIn, getUserDetails);
```

Ici, le middleware en question est signIn - avant de se diriger vers le controller attendu, la requête va être mise “en suspens” pour que des données nécessaires soient vérifiées.

On ne veut pas que n’importe qui ait accès aux détails d’un utilisateur, on veut seulement que les membres inscrits et connectés puissent le faire. SignIn nous permet de le faire en vérifiant le token de l’utilisateur (s’il y a bien un token).

```
const mySecret = "mysecret";
const decoded1 = jwt.verify(tokenToUse, mySecret);
req.user = decoded1;
const decoded2 = jwt.decode(tokenRefresh)
var now = new Date().getTime() / 1000;
```

Dans le cas où ce n’est pas le cas, la requête se verra automatiquement échouée.

```
} catch (err) {
    // console.log('auth.js : ', err);
    return res.status(401).send(err);
}
```

Cela permet également de rajouter une couche de sécurité et de pouvoir rediriger les utilisateurs de l’application vers les services disponibles selon leur statut.

**Controler**

Le rôle classique du controler est de gérer les réponses renvoyées par le model, les traiter et les envoyer sur la view. Il fait office d'intermédiaire entre les deux pôles, ainsi qu'entre le serveur et le client.

Dans notre application, nous avons décidé de nous passer de model et de tout rassembler dans le controler pour une raison de simplicité principalement. Alors que le model s'occupe de réaliser les demandes à la base de données d'ordinaire, nous avons intégré ceci dans le controller.

```
const getAllFromUsers = (req, res) => {
    const sql = 'SELECT * FROM users'
    db.query(sql, function (error, data) {
        if (error) throw error;
        else res.send(data);
    })
}
```

Aussi ce dernier détient et envoie non seulement la requête sql mais également s'occupe de la traiter s'il est nécessaire.

La plupart de nos fonctions sont similaires à celle présentée au-dessus, à savoir qu'il n'y a qu'une seule requête et un traitement minime (justifiant encore une fois notre décision d'utiliser que le controller). Il arrive que le traitement doit être plus spécifique comme ci-dessous.

```
const addUserToRoom = (req, res) => {

    const verifyRoles = `SELECT role FROM  users INNER JOIN roles
    ON roles.id = users.id_role WHERE users.id = ?`
    db.query(verifyRoles, [req.user.id], function (error, data) {

        if (data.length !== 0) {
            if (data[0].role !== "ban") {

                const verifyParticipation = `SELECT id_room FROM participants WHERE id_user = ? AND id_room = ?`
                db.query(verifyParticipation, [req.user.id, req.params.idRoom], function (error, dataIdRoom) {
                    if (dataIdRoom[0] == undefined) {
                        var insertUser = insertToRoom(req.body.id_room, req.user.id);
                        res.status(200).send({ message: 'Request succeed.' })

                    } else res.status(400).send({ message: 'The id user ' + [req.user.id] + ' is already related to t
                })
            } else res.status(400).send({ message: 'You were ban of this room.' })
        }
    })
}
```

Dans cette partie, un utilisateur peut s'ajouter à une room seulement s'il n'est pas banni de celle-ci. Dans un MVC classique, tout le traitement de la data aurait été dans le controller seulement.

**Sécurité**

Nous avons vu précédemment comment les middlewares permettaient une première sécurisation. En effet, ils permettent, principalement, dans notre application de vérifier les accès.

Il y a également une mise en place de rôles. En effet, il y en a trois disponibles : l'utilisateur lambda, l'administrateur et l'utilisateur banni - selon l'attribution de ce dernier, l'utilisateur aura différentes possibilités et droits au sein de l'application. Cela se présente sous forme d'un id_rôle qui est de 1 pour un utilisateur lambda, de 2 pour l'admin et de 0 pour un banni.

Il faut savoir que dans le cadre de l'administrateur, les deux points évoqués sont intimement liés :

```
//Admin route
app.use('/admin', [signIn, isAdmin], admin)
```

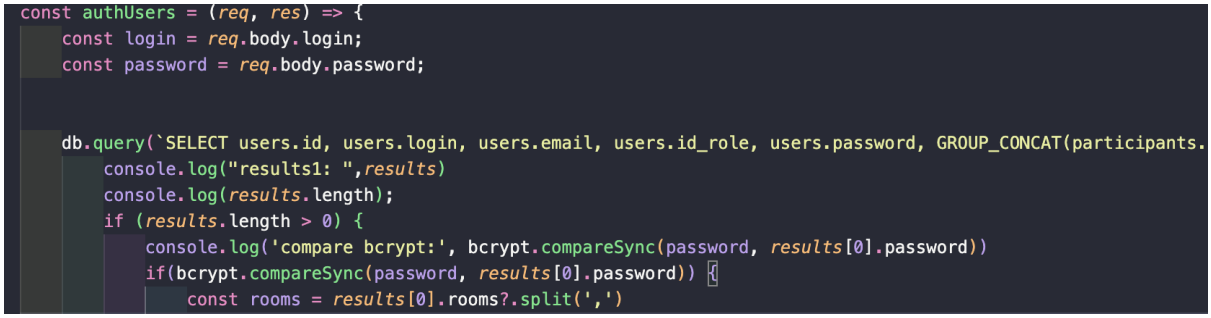
Effectivement, pour ce dernier, nous avons mis en place un middleware spécifique.

```
exports.isAdmin = (req, res, next) => {
    console.log(req.body)
    console.log('req id role admin', req.body.id_role_admin)

    if (parseInt(req.body.id_role_admin) === 2) {
        return next();
    }else {
        return res.status(400).json({ message: "You are not an administrator" });
    }
}
```

Ici on vérifie l'id du rôle.

Évidemment, de manière classique nous avons les vérifications au niveau de la connexion : login et mot de passe correspondants.



```
const authUsers = (req, res) => {
    const login = req.body.login;
    const password = req.body.password;


    db.query(`SELECT users.id, users.login, users.email, users.id_role, users.password, GROUP_CONCAT(participants.
        console.log("results1: ",results)
        console.log(results.length);
        if (results.length > 0) {
            console.log('compare bcrypt:', bcrypt.compareSync(password, results[0].password))
            if(bcrypt.compareSync(password, results[0].password)) {
                const rooms = results[0].rooms?.split(',')
```

Mais il y a également, la mise en place de bcrypt pour le password notamment. Bcrypt permet le hachage du mot de passe en base de données : au lieu d'avoir le mot de passe directement rentré dedans, il est transformé pour éviter la fraude par force brute.

```
jennie      $2b$10$7tMGWzG2VhbwZW8HAfDFr.phidwVk44pgFA3RqDxSZo...
```

Aussi, il est important de le déchiffrer lors de la vérification du password lors du login.

Enfin, la nouvelle instance de sécurité emmenée est le JWT aussi dit Json Web Token. Le Json Web Token (JWT) est un standard ouvert (protocole de communication) qui échange des informations entre deux parties sous forme d'objet JSON. L'information est signée de manière digitale via un algorithme HMAC ou RSA.
Le jeton fourni est composé de trois parties :

- Un en-tête (header), utilisé pour décrire le jeton. Il s'agit d'un objet JSON.
- Une charge utile (payload) qui représente les informations embarquées dans le jeton. Il s'agit également d'un objet JSON.
- Une signature numérique.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

On peut le décoder grâce au site jwt.io.
Ce jeton est généré lors de la connexion de l'utilisateur : ce dernier entre ses informations et une demande est envoyée au serveur afin de fournir le token. Ce token

sera alors vérifié lors des actions réalisées par l'utilisateur : il contient ses renseignements (son id, login, son rôle, ses rooms) et permet de s'assurer que l'utilisateur a des droits d'accès sur certaines rooms etc.
Il a également une durée de vie - nous étions donc obligés de le régénérer.

```
const tokenToUse = req.headers.token1;
const tokenRefresh = req.headers.refreshtoken;
```

Généralement le token et son refresh se trouvent dans le header de la requête.

```
const mySecret = "mysecret";
const decoded1 = jwt.verify(tokenToUse, mySecret);
req.user = decoded1;
var now = new Date().getTime() / 1000;
```

Une fois récupérés, on vérifie le token (const decoded1 - s'il y a une erreur on atterrit dans le catch directement ) puis on s'occupe principalement du refreshToken : voir s'il a expiré ou pas.

```
//verification avec la date actuelle, si l'expiration
const decoded2 = jwt.verify(tokenRefresh, mySecret)
var now = new Date().getTime() / 1000;

//mettrev decoded2.iat et pas .exp
if (now > decoded2.exp) {
    /* expired */
    //le token est disponible ds le scope grace au callback ds refreshToken du usersController
    return refreshToken(decoded1.id, token => {

        // console.log('first')
        res.status(417).send(token)
    });
}

next();
```

Dans le cas où le refresh est périmé, on relance un nouveau refreshToken.

Le token a été une nouvelle problématique avec laquelle on devait jouer - il y a eu malheureusement beaucoup de problèmes pour le mettre en place et l'appliquer correctement. Cependant, voici quelques pistes d'amélioration qu'avec le recul, nous allons pouvoir mettre en place afin d'utiliser le JWT de manière un peu plus correcte :

- Si un utilisateur effectuait une action qui nécessitait une mise à jour de l'état du compte, comme la suppression, le blocage ou la suspension du compte, nous pourrions invalider le JWT associé à cet utilisateur en maintenant une liste

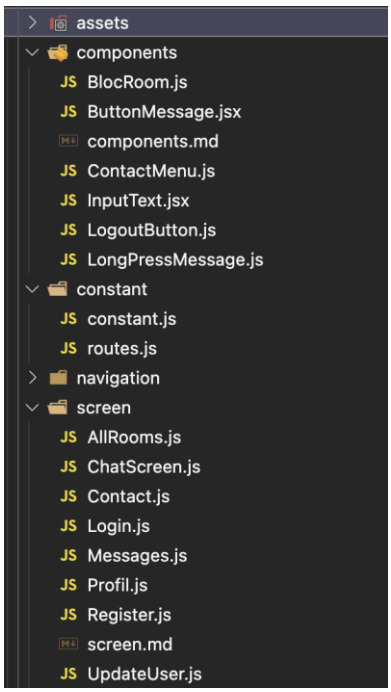
blanche ou une liste noire des JWT valides/invalides. Ainsi, à chaque requête, nous vérifierons si le JWT fourni est toujours valide.

- Pour limiter la fenêtre d'exposition en cas de changement d'état du compte ou de modification des autorisations, nous pourrions définir des durées de validité plus courtes pour les JWT. Cela nécessiterait une régénération plus fréquente des JWT, mais offrirait une meilleure réactivité en cas de changements importants.

# Fonctionnement du front

**Arborescence**

Dans le dossier src, nous avions divisé le travail en plusieurs sous-dossiers :



Notre dossier components avait les composants réutilisables. Navigation comprenait la navigation de l'application et le screen les "pages".

**Composants**

Notre front a été pensé en React, ce qui induit donc l'utilisation de composants. Un composant est un bout de code indépendant et réutilisable. Il marche généralement comme une fonction. Il existe des composants en class mais aussi en fonctions - dans le cadre de notre application, nous avons utilisé les derniers, étant recommandés puisque préférés dernièrement dans le développement. Nous avons donc divisé nos composants en deux catégories : la première comprenait les composants réutilisables (tels que des boutons etc…), la seconde comprenait les composants plus grands (les screens) qui utilisaient donc les premiers et transmettaient les informations nécessaires via des props.

```
export default function BlocRoom({name, room, id, pressRoom, deletePress, specialClass, touchable, tb, disabled, m

    const blocRoomPress = () => {
        // console.log('room id', room.id)
        pressRoom({'id':room.id, 'name':name})
    }

    const deleteRoom = () => {
        deletePress(id)
    }

    return (
        //ajouter dans le press qu'on peut aller dans les messages de la room également si c'est notre room
            <TouchableOpacity
                style={styles.container}
                onPress={() => {deletePress ? deleteRoom() : blocRoomPress()} }
                // disabled={}
            >
                {/* Image */}
                <View style={styles.tinyIcon}>
                    <Image
                    style={styles.img}
                    source={{uri: 'https://64.media.tumblr.com/6b9e50e7237cf04fd44b6f56ce75e848/04f19f3b5d21afac-b
                    />
                </View>
                <Text style={styles.name}>{name}</Text>
            </TouchableOpacity>
    )
}
```

Par exemple, ici nous avons un composant réutilisable. Il s'occupait de former les petites bulles de présentation des rooms. Ce composant était réutilisé dans la même page selon des states différents (il a été cliqué ou non).

```
<ScrollView style={styles.bg}>
    <View style={styles.tabs}>

        <TouchableOpacity onPress={() => underlined(2)}>
            <Text style={underline === 2 ? styles.selected : styles.notSelected}>More bands</Text>
        </TouchableOpacity>
        <TouchableOpacity onPress={() => underlined(1)}>
            <Text style={underline === 1 ? styles.selected : styles.notSelected}>My chuu bands</Text>
        </TouchableOpacity>
    </View>
    <View style={styles.container}>
        {
            rooms.length >= 1 ?
            rooms.map((room) =>
                <BlocRoom
                    key={room.id}
                    name={room.name}
                    room={room}
                    tab={underline}
                    pressRoom={setNewRooms}
                    // specialClass={moreRooms.find(moreRoom => moreRoom.id === room.id) ? true : false
                    // touchable={touchable}
                    tb = {false}
                    disabled = {disabled}
                    moreRooms = {moreRooms}
                />
            )
            :
            <Text>No rooms.</Text>
        }
    </View>
```

Ici par exemple, nous avons un “composant” qui est réellement une page entière. On peut d’ailleurs remarquer que BlocRoom y est utilisé, il est instancié plus haut par import.

```
import BlocRoom from '../components/BlocRoom';
```

Il y a, néanmoins, une différence notable entre React JS et React Native. En effet, le React que je mentionne tout au long du dossier est React Native. React Native est utilisé pour la création d’applications mobiles qui s’adaptent aussi bien en IOS qu’en android tandis que React JS est plutôt utilisé pour les applications web qui nécessitent donc un browser. La raison étant que React JS se sert de Javascript, CSS et HTML pour construire les interfaces lorsque React Native use en plus des composants natifs comme on peut le voir sur les captures d’écran du dessus : l’un des plus basiques est le composant <View> qui est un conteneur capable de supporter les flexbox, style, contrôle d’accessibilité etc.

Aussi, un détail important à noter est que le style de nos composants était rédigé sur le même fichier et non sur un fichier css dédié. Pour ce faire, on avait créé une constante styles qui était au final une fonction avec un retour d’objet que l’on parcourait en donnant la clef souhaitée (classname) à notre composant natif.

```
const styles = StyleSheet.create({
    container: {
        flex: 1,
        marginTop: 52,
        alignItems: 'center'
    },
    boxTitle: {
        marginTop: 90,
    },
    boxForm: {
        marginTop: 60,

    },
```

```
<Text style={styles.title}>
```

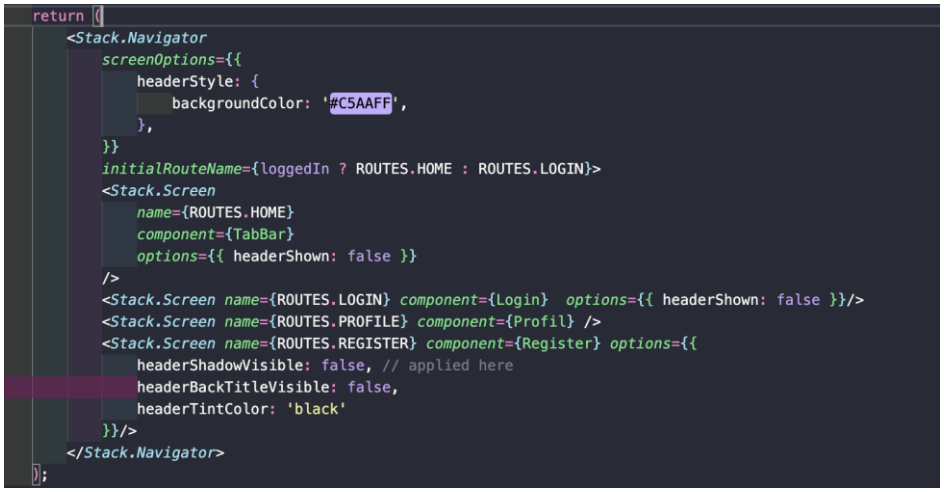
# Développement

Un fois la conception terminée et les bases posées, chacun pouvait alors commencer à développer selon les tickets attribués.
Dans cette partie je vais me concentrer principalement sur le code et son explication.

**Extrait de code**

*a) Navigation*

Pour la navigation, nous avons utilisé React Navigation, notamment Stack qui ajoute un effet de “stacking” entre les pages (installation via npm).



```
return (
    <Stack.Navigator
        screenOptions={{
            headerStyle: {
                backgroundColor: '#C5AAFF',
            },
        }}
        initialRouteName={loggedIn ? ROUTES.HOME : ROUTES.LOGIN}>
        <Stack.Screen
            name={ROUTES.HOME}
            component={TabBar}
            options={{ headerShown: false }}
        />
        <Stack.Screen name={ROUTES.LOGIN} component={Login}  options={{ headerShown: false }}/>
        <Stack.Screen name={ROUTES.PROFILE} component={Profil} />
        <Stack.Screen name={ROUTES.REGISTER} component={Register} options={{
            headerShadowVisible: false, // applied here
            headerBackTitleVisible: false,
            headerTintColor: 'black'
        }}/>
    </Stack.Navigator>
);
```

La manière dont le navigateur marche est principalement grâce à des noms et des composants (screens) associés.
Le stack.navigator est le composant de base où les screens sont répertoriés. Le notre prend en props initialRouteName qui permet de choisir un nom à la route rendered en premier. On a également un screenOptions qui permet de personnaliser le navigateur. Ensuite, s'empilent les stack.screens : chacun a un nom et un composant associé - on peut également y ajouter des options comme ici en disant qu'on ne veut pas que le header soit visible. En effet, pour certaines pages, il ne nous était pas utile.
C'est le navigateur à son état le plus basique qui permet une passation entre les pages selon les "press" de l'utilisateur : c'est celui qui permet par exemple dans la page "messages" de cliquer sur une room utilisée par l'utilisateur et d'atterrir sur la page des messages de cette room.

Cependant, notre navigateur a également une tab-bar située sur le bas pour une accessibilité directe avec le pouce de l'utilisateur. Aussi, nous l'avons codé de la sorte :

```
<Tab.Navigator {...{screenOptions}} >
    <Tab.Screen name={ROUTES.CHATROOMS} component={ChatRoomStack} options={{
            tabBarIcon: ({focused}) => (
                <View style={{alignItems: 'center', justifyContent: 'center',}}>
                <Image
                    source={require('../../assets/icons/chuu-purpl.png')}
                    resizeMode='contain'
                    style={{
                        width: 35,
                        height: 35,
                        tintColor: focused ? '#B2FFDF' : '#ADADAD'
                    }}
                />
                <Text style={{color: focused ? '#B2FFDF' : '#ADADAD', fontSize: 10}}>
                    rooms
                </Text>
            </View>
            ),
            headerShown:false
        }}
    />
```

De la tab-bar, l'utilisateur pouvait avoir accès à ses rooms, ses messages ainsi qu'à son profil. Le concept est le même : un navigator contient des screens avec un nom qui renvoient à des composants (screens) particuliers.

### *b) Socket.io*

Afin de mimer la messagerie instantanée, nous avons mis en place Socket.io. Il dispose d'une connexion bidirectionnelle signifiant que le serveur peut pousser des informations (ici messages) aux clients connectés dans la chatroom.

Il faut commencer par installer socket.io via npm dans le projet.

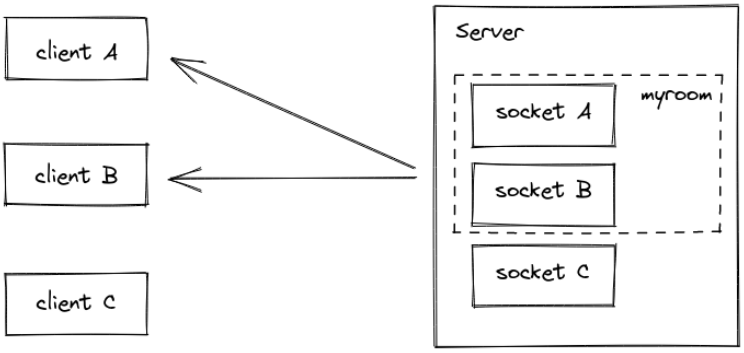
Par la suite, nous avons dû faire une connexion du côté serveur.

```
const io = require('socket.io')(server)
```

```
io.on('connection', (socket) => {
    console.log('a user connected');
    socket.on('disconnect', () => {
      console.log('user disconnected');
    });
    socket.on('joinIn', (id_room) => {
      socket.join(id_room);
    })
  });

  server.listen(port, () => {
    console.log(`Socket.IO server running on port :${port}`);
});
```

Cette connexion de socket io se réalise à chacune des rooms qui sont identifiées par leur id grâce à ".join".




Pour que le serveur connaisse les id des rooms, on appose à socket un “évènement” nommé “joinIn” - dans l’exemple au-dessus, il est écouté via le “.on”.

Cet événement est émis du côté client dans notre composant ChatRoom qui détient l’ensemble des id des rooms. C’est également dans ce composant qu’on appelle celui des Messages à qui on fait passer la connexion socket en props.

```
import { io } from 'socket.io-client';
```

///

```
const [socket, setSocket] = useState(io("http://localhost:3000"));

useEffect(() => {
    socket.emit('joinIn', route.params.id_room)
```

```
<Messages style={{ height: '40%' }} idRoom={route.params.id_room} socket={socket} />
```

Puis, dans le composant Message, on écoutait l’évènement “newMessage”.

```
    props.socket.on('newMessage', message => setMessages(messages => [...messages, message]));
```

Ce dernier était émis depuis notre controler de messages dans la fonction qui s’occupait d’envoyer un message dans la base de données.

```
    io.to(parseInt(req.params.roomId)).emit('newMessage', message)
```

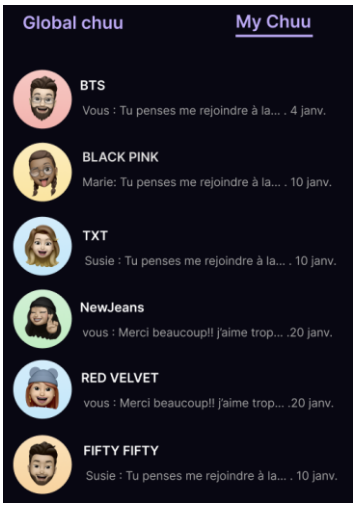
Etant donné que nos messages étaient un useState, une fois qu’on récupérait l’entièreté des messages, on avait un setter qui en faisait un array qu’on pouvait parcourir et donc afficher sur l’écran par la suite.

```
getMessages(data => {
    setMessages(data);
    console.log('getMessages: ',setMessages);
})
```

Aussi, on disait à socket que sous l'évènement "newMessage", il devait re-set un nouveau tableau avec en plus l'information "fraîchement" envoyée. Ce qui faisait que lorsqu'on "map"-pait notre array de messages, on avait toujours également le dernier message envoyé dans la chatroom.
Socket s'occupait donc d'éviter un refresh qu'on aurait dû avoir en temps normal - de cette manière il avait un accès au côté serveur et envoyait en instantané à tous les clients participants à la room.

### *c) Contact page : get answers from the API*

Semblable à un jeu d'essai, je développe dans cette sous-partie une fonctionnalité de notre application. Lorsqu'un utilisateur se connecte à l'application, s'il participe à des rooms, il est redirigé vers une page qui répertorie toutes ses chatrooms et le dernier message envoyé dans chacune de ces rooms.




Pour ce faire, dans le screen contact.js, nous appelions l'API et faisions une requête http (ci-dessous "url") au serveur depuis le client.

```
axios({
        method: 'get',
        url:`${API + uri}`,
        headers: {
            'Content-Type' : 'application/json',
            token1: token,
            refreshtoken: refresh
        }
    }).then((response) => {
        setContacts(response.data)
    })
```

L'uri était le chemin que devait prendre le routeur pour retrouver le controler correspondant.

```
uri = "/rooms/contact"
```

Tandis que l'API était une constante qu'on importait.

Le routeur allait donc à l'index.js qui répertorier l'ensemble des "grandes routes".

```
//Rooms route
app.use('/rooms', rooms);
```

Ensuite, il retrouvait la route correspondant aux rooms.

```
var rooms = require('./src/routes/rooms');
```

Et dans cette dernière, il retrouvait enfin le controler demandé.

```
// route get dernier message, d'un chat name etc...dans lequel le participant participe
router.get('/contact', signIn, displayRoomsAndChat);
```

Dans roomsController.js, on trouvait donc :

```
// get dernier message, d'un chat name etc...dans lequel le participant est
const displayRoomsAndChat = (req, res) => {
    const sql = `SELECT rooms.id, rooms.name, messages.content FROM messages CROSS JOIN rooms ON messages.id_room = ro
    db.query(sql, function(error, data){
        if (error) throw error;
        else res.status(200).send(data);
    })
}
```

S'il y avait une erreur, elle était traitée. Dans le cas contraire, le serveur renvoyait une réponse positive et les informations demandées par le côté client.

Du côté client, nos contacts étaient un useState alors on "set"-tait (setContacts) la réponse fournie par l'API dans notre state.

```
    contacts.length > 0 ?
    contacts.map((contact) =>
        <ContactMenu
            key={contact.id}
            contact={contact}
            onPress={() => navigation.navigate(ROUTES.MESSAGES, {id_room: contact.id, room_na
        />
    )
    :
    <View style={styles.container2}>
        <Text style={styles.text}>
            You are not subscribed to any rooms.
        </Text>
        <TouchableOpacity style={styles.cta}>
                <Text style={styles.textCta}> Get a room </Text>
        </TouchableOpacity>
    </View>
```

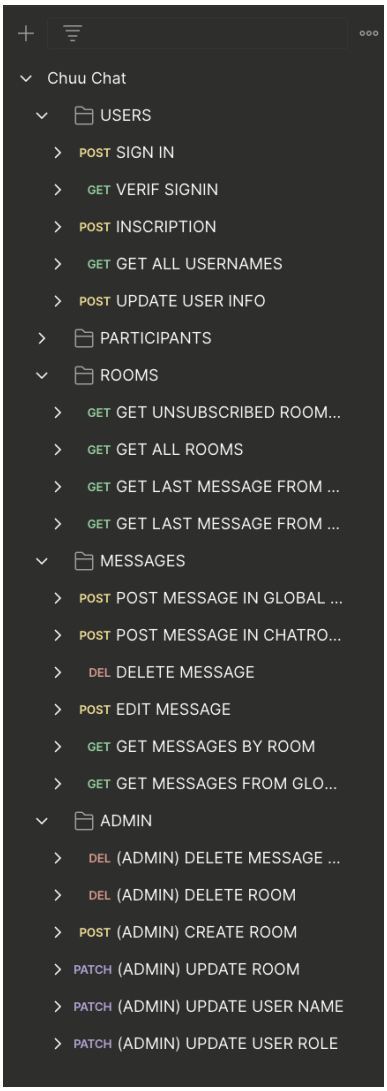
Si la réponse était “vide”, donc que l’array contacts se trouvait vide (l’utilisateur ne participait à aucune room), on envoyait une view simple avec un texte lui indiquant qu’il n’avait aucune chatroom.

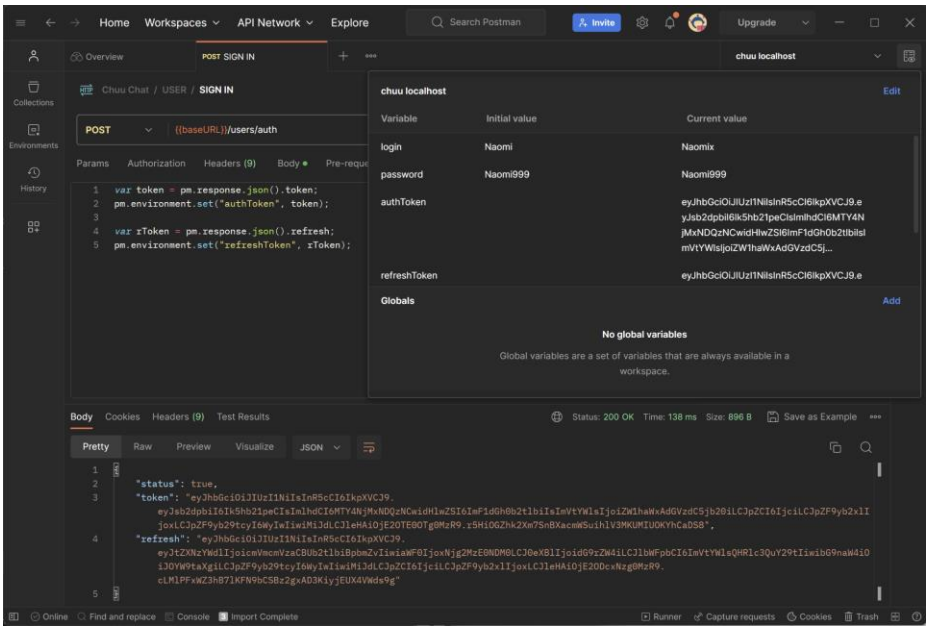
**Tests**

Les tests en développement permettent de vérifier que les fonctionnalités développées marchent correctement et que s’il y a des erreurs, ces dernières sont traitées via le code. Il existe plusieurs tests qui peuvent être aussi bien manuels qu’automatisés :

- les *tests unitaires* : consistent à tester de manière individuelle et isolée chaque fonction du projet en comparant le résultat effectif et le résultat attendu en fonction d'un jeu de données test prédéfinies.
- les *tests fonctionnels* : consistent à tester une fonctionnalité complète afin de vérifier un scénario d'utilisation de bout en bout (ex = connexion d'un utilisateur à l'application).
- les *tests de non régression* : consistent à spécifiquement vérifier que la version en cours n’a pas généré de régression par rapport à la version précédente.

Pour un projet aussi minime, nos tests n’étaient pas forcément les plus développés. Néanmoins, nous avons utilisé Postman pour tester notre API et ses requêtes manuellement.
Dans un premier temps, nous avons créé la collection Postman recensant l’ensemble des appels possibles vers notre API.




Collection Postman de l'API
Scripts d'enregistrement des tokens dans l'environnement

Afin de fluidifier le processus de test, nous avons créé 2 environnement Postman contenant le nom de domaine de l'API ainsi que les informations de l'utilisateur connecté. Cela permet alors de passer d'un utilisateur à un autre sans avoir à modifier la collection.

L'outil "Tests" de Postman nous a ainsi permis de créer un script qui enregistre les tokens de connexion de l'utilisateur dans les variables de l'environnement. Ce script est donc configuré pour être automatiquement exécuté après chaque appel de la requête de connexion.

Une fois tout cela mis en place, il a fallu lancer chaque requête une à une, puis comparer le résultat de la requête (response + contrôle des données en base) au résultat attendu, puis le consigner dans le cahier de recettes.

# Conception de la partie administration

Lorsque l'ensemble du projet est réalisé pour être utilisé en tant qu'application mobile, la partie administration, elle, est plutôt une application web.

C'est à partir de cette dernière que l'administrateur aura une influence sur l'application mobile.

Pour avoir un accès à la partie administrateur, ce dernier doit déjà s'identifier en entrant le login et le password. Lorsque ces derniers étaient vérifiés, on s'occupait alors de vérifier le rôle de l'utilisateur. En effet, un rôle "2" était synonyme d'être administrateur et donc pouvoir entrer dans la partie web.

```
if(roleAdmin[0].id_role === 2) {
    // store the token in local storage
    sessionStorage.setItem('token', json.token);
    sessionStorage.setItem('refresh_token', json.refresh);
    sessionStorage.setItem('id_admin', roleAdmin[0].id_role);
    window.location.href = front + "/app-mobile-chat-admin/admin/adminIndex.php";
}
```

L'administrateur était alors redirigé vers son index. Dans le cas contraire, il n'y avait pas accès.

On stockait également les tokens et le rôle de l'administrateur dans une sorte de session afin de pouvoir les utiliser plus facilement lors des différentes requêtes.

```
const bodyData = JSON.stringify(data)
const token = Token.get();
const refresh = refreshToken.get()
```

```
fetch(API + `/admin/users/${idUser}/update`, {
        method: 'PATCH',
        mode: 'cors',
        body: bodyData,
        headers: {
            'token1': token,
            'refreshtoken': refresh,
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Credentials': 'true'
        },
    })
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

Puisqu'en effet, l'identité de l'administrateur était vérifiée à chaque requête du côté client par un middleware au niveau serveur.

```
//Admin route
app.use('/admin', [signIn, isAdmin], admin)
```

La connexion entre la web app et notre serveur était validée via les CORS (cross-origin resource sharing). En effet, notre client web app avait un port différent que celui du serveur de l'application.
Le CORS est un mécanisme HTTP-header qui permet à un serveur d'indiquer quelle origine (domaines ou ports) autre que la sienne peut accéder à ses ressources.

```
const cors = require('cors');
const corsOptions ={
    origin:['http://localhost:3000','http://localhost:8888','http://localhost:8881', 'http://localhost'],
    credentials:true,            //access-control-allow-credentials:true
    optionSuccessStatus:200
}
app.use(cors(corsOptions));
```

# Problématiques rencontrées

Plusieurs spécificités techniques ont posé problème au cours du développement mais également la conception de l'application. Il y a eu bien sûr l'implémentation des sockets.io (et notamment comment dispatcher correctement les éléments : où écouter l'évènement etc…avec la bonne room), la navigation côté front aussi et les spécificités de React Native.
Il y a eu également, lors du développement du côté administrateur, les CORS qui pouvaient nous bloquer avec ce genre d'erreur :

```
Access to XMLHttpRequest at 'http://localhost:5000/global_config'   step1:1
from origin 'http://localhost:8080' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No 'Access-
Control-Allow-Origin' header is present on the requested resource.
```

Il suffisait d'ajouter son port dans les autorisations.

Un autre souci était plutôt d'ordre social - ou comment coordonner une équipe de quatre personnes avec différentes alternances au cours d'une année segmentée. Bien que nous ayons mis en place la technique de l'agilité, il y a eu des petits ics et inconvénients qui ont joué un rôle défavorable quant à notre avancée. Que ce soit le désistement d'un membre, la baisse de motivation, ou encore des différends au niveau de l'organisation, il a fallu adapter l'avancement du projet et les différentes techniques employées selon le groupe.

# Recherches anglophones et documentation

La connaissance et la maîtrise de l'anglais est un outil fondamental pour un développeur web. Pour plusieurs raisons mais la principale étant que la plupart des outils qu'il utilisera seront toujours d'abord en anglais lors de leur sortie, ce qui signifie que les mises à jour le seront aussi ainsi que les instructions.
Une autre raison est pour les recherches. Beaucoup seront à faire en anglais pour trouver les dernières façons de faire et celles plus détaillées.
Que ce soit des vidéos anglaises ou simples recherches sur mon navigateur, j'ai pour ma part principalement tout réalisé en langue anglaise.
Aussi sur ce projet, de nouvelles compétences ont dû être mises en jeu - il a fallu également se renseigner sur de nouveaux outils.
Par exemple, les sockets dont la documentation socket.io est en anglais :

## How it works

The bidirectional channel between the Socket.IO server (Node.js) and the Socket.IO client (browser, Node.js, or another programming language) is established with a WebSocket connection whenever possible, and will use HTTP long-polling as fallback.

The Socket.IO codebase is split into two distinct layers:

- the low-level plumbing: what we call Engine.IO, the engine inside Socket.IO
- the high-level API: Socket.IO itself

D'autre part, Stackoverflow est une source majeure pour un développeur. Sur cette plateforme, il peut non seulement poser des questions, présenter ses problèmes mais aussi répondre et recevoir lui-même des réponses. C'est donc un outil important et qui est majoritairement en anglais.

# Conclusion

Ce projet est un résumé global de mon année et de toutes les choses que j'ai pu apprendre. Même les difficultés rencontrées m'ont permis de m'améliorer sur des compétences primordiales en tant que développeur mais également de travailler l'amont : la conception.

J'ai pu réaliser, avec mon équipe, une veille technologique sur différents principes comme React Native, NodeJS etc... J'ai pu également pousser une vision plus professionnelle, c'est-à-dire qu'on a tenté avec notre groupe de nous rapprocher le plus possible de ce qui pourrait être potentiellement fait dans nos entreprises d'alternance respectives et de trouver un pont entre les deux mondes.

Bien que pouvant être grandement amélioré, jongler entre ce projet et les différentes demandes du monde professionnel dans notre alternance a été un challenge de taille que nous avons surmontés grâce aux techniques de conception employées mais également grâce à notre charte de travail établie au préalable. Il était également intéressant de travailler des aspects que personnellement, je ne touchais pas dans mon entreprise : effectivement, dans cette dernière j'ai eu l'occasion de plus travailler le front alors qu'avec cette application mobile, j'ai pu toucher globalement à l'entièreté du travail.