SOMMAIRE

INTRODUCTION

Je suis Thomas Dellacase, et ce dossier de projet concerne le projet ChatApp réalisé par Maxime Hadj, Thomas Serdjebi et moi même. Le projet ChatApp est une application de messagerie instantanée développée en couches.

Je suis actuellement étudiant en préparation du titre de Développeur et Concepteur d'Applications à la Plateforme_ à Marseille. Anciennement comptable et animé par une envie de reconversion professionnelle dans un métier plus technique et plus ingénieux, j'ai obtenu mon titre de Développeur Web et Web Mobile suite à une année d'étude en initial à la Plateforme_.

Au cours de ma première année, j'ai pu acquérir les compétences de bases d'un développeur web, qui, couplées à mes compétences en comptabilité m'ont permis de décrocher une alternance au sein de l'entreprise Ligno à Aix pour préparer le titre de Concepteur Développeur d'Applications.

Afin de couvrir l'ensemble des compétences visées par le titre de Conception Développeur d'Applications, j'ai choisi de présenter un projet scolaire, le projet ChatApp, qui répond entièrement aux compétences attendues pour l'obtention du titre.

I. Liste des compétences du référentiel couvertes le projet

Le projet de l'application ChatApp couvre les compétences suivantes :

- 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
- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
- Concevoir une Application
- Développer des composants métiers
- Construire une application organisée en couches
- Développer une application mobile
- Développer une interface utilisateur de type desktop
- Préparer et exécuter les plans de tests d'une application
- Préparer et exécuter le déploiement d'une application

II. Project Summary

The ChatApp Project is a **mobile chat application** allowing its users to sign up by filling a form with their profile information, sign in, update their information and to talk with the other users with an **instant messaging service**. Indeed, they can talk with other users privately or in several chat rooms. The ChatApp Project can be divided into 4 parts given that it is a layered application.

First of all, let's introduce the **mysql database** which is hosted by **PHPMYADMIN**. The database was first modeled in different schemas : Entity Relationship Model, Logical Data Model and Physical Data Model. It contains **4 main tables (channels, messages, roles, users) and a join table users_messages** to make a link between the messages table and the users table.

Then, its **API** which contains all the **functions needed by the APP to interact with our database**. This API was developed in **Javascript inside a NodeJS environment**. Inside you can find three main folders : **users, chatrooms, and messages**. Inside of each folder, you can find three main type files : **services**, with all the SQL requests that will be used, **controllers** for the errors, security and data handling and finally **routes** with the routes to call to use the API.

Also, the third part is **the application itself used by the users**, which **makes calls to the API functionalities to work**. The application was developed with **React and React Native** and also with the **framework** named **Expo**. This application is divided into several screens and uses stack navigation.

Finally, an **administration back-office web interface** which allows administrators to handle users, chat rooms and to moderate the messages which are sent. It also **makes calls to the API** to **interact with the database and it was developed in ReactJS**.

We were 3 students from the Plateforme_ School to develop this application : Maxime Hadj, Thomas Delacase and I. We are all worky-study students in different companies as developers and we have decided to combine our skills to carry out this project.

III. Cahier des charges et spécifications fonctionnelles

Le cahier des charges spécifie une **application mobile divisée en 4 parties** : une base de données, une API, une application mobile et un back-office administrateur, ainsi qu'une base de données.

A. La base de données

La base de données se devait d'être modélisée en suivant le **MCD** et le **MLD**. Elle devait à minima contenir les **tables** suivantes : **Users, Messages, Channels, Roles**. Sa gestion est en **MYSQL** est elle est hébergée sur **PHPMYADMIN**, un outil de gestion pour les systèmes de gestion de base de données MySQL et MariaDB réalisée principalement en PHP.

B. L'API

L'API a dû être réalisée à l'aide du **framework ExpressJs** qui est utilisé pour la **création d'API et d'applications mobiles**, prenant en charge les détails essentiels du back-end comme la **gestion des erreurs et le routage**. Il est relativement **simple à utiliser** puisqu'il se base sur le langage **Javascript** et permet d'utiliser un **seul langage pour le front-end et le back-end**. Il est régulièrement révisé et mis à jour. Nous l'avons installé sous un **environnement NodeJS**.

L'API devait comporter **plusieurs routes** :

- **routes accessibles à tous** : inscription, connexion, avec une méthode d'identification JSON WEB TOKEN
- **routes accessibles aux utilisateurs connectés** : liste des utilisateurs, détails d'un utilisateur, modification de ses informations, envoie de messages dans un salon de discussion ou en discussion privée
- **routes accessibles aux administrateurs uniquement** : modification, suppression d'un utilisateur, suppression d'un message, création/modification/suppression d'un salon de discussion

… ainsi que **plusieurs middlewares** :

- vérification du token de l'utilisateur qui est connecté
- vérification du statut d'administrateur de l'utilisateur
- vérification de si émail existant pour l'inscription des utilisateurs afin d'éviter des doublons

… et enfin la **mise en place de websockets** permettant d'ouvrir un c**anal de communication interactif permanent entre l'application et le serveur**, permettant l'actualisation de l'application en temps réel, notamment pour les fonctionnalités comme les messages.

C. L'App Mobile

L'application mobile devait tout d'abord être **maquettée sur Figma** afin d'avoir un aperçu des **différents écrans** qui la composent ainsi que de sa **barre de navigation**. Elle devait se composer de différents écran :

- inscription
- connexion
- liste des utilisateurs
- détails d'un utilisateur
- profil
- liste des salons de discussions
- salon de discussion
- discussion privée

En ce qui concerne **ses fonctionnalités**, elles corroborent avec les **routes développées dans l'API** : connexion, inscription, affichage de son profil ou de celui d'un autre utilisateur, modification de ses informations personnelles, messagerie instantanée privée et salons de discussions.

Nous avons choisi le **framework Expo** qui permet la **facilitation** pour la **création** et le **déploiement d'applications mobiles avec React Native**, embarquant de nombreuses librairies et outils utiles pour React Native. Ses principaux avantages réside également dans le fait qu'il **gère la mise à jour de ces librairies**, permet de **tester les applications directement sur mobile** en ayant installé l'application Expo Go et facilite la publication des applications sur les stores Apple et Google.

D. Le panel administrateur

Le **panel administrateur** devait être une interface sous format d'**application web**, où il aurait la possibilité de supprimer/modifier des utilisateurs, des messages, soit administrer l'application dans son ensemble.

Nous avons choisi d'utiliser **React** et **React Admin** afin de rester sur des technologies similaires à celles utilisées dans l'API et dans l'application mobile. React Admin a facilité la création d'une interface administrateur.

IV. Gestion de projet

A. Planification des tâches

Nous avons établi une **planification des tâches** en fonction des **besoins de chaque partie du projet**. L'**API** nécessite que la **base de données soit déjà en place**. L'**application mobile** et l'**interface administrateur web** nécessitent que l'**API soit en place**. Voici la façon dont nous avons planifié les tâches :

a. La base de données

1. Définition des tables et des champs associés
2. Modélisation en MCD, MLD
3. Création de la base de données sur PHPMYADMIN

b. L'API

1. Documentation sur l'environnement NodeJS et du framework Express
2. Définition de l'arborescence de l'API
3. Installation de l'environnement
4. Développement des composants d'accès à la base de données
5. Développement des fichiers "Services" comportant des fonctions utilisant les requêtes SQL
6. Développement des fichiers "Controllers" comportant les différents contrôles et la gestion des erreurs
7. Développement et intégration des middlewares
8. Développement des fichiers de "Routes"
9. Phase de test

c. L'application mobile

1. Documentation sur le framework Expo et la librairie GiftedChat
2. Maquettage de l'application
3. Installation de l'environnement
4. Définition de l'arborescence de l'application
5. Modélisation de la navigation
6. Développement d'un système de navigation temporaire afin de faciliter l'accès aux différents écrans
7. Développement des écrans suivi d'une phase de test après la création de chaque écran
8. Développement d'un nouveau système de navigation intégrant la road map de l'utilisateur
9. Phase de tests
10. Ajustement et améliorations des écrans
11. Phase de tests

d. Le panel administrateur

1. Documentation sur la bibliothèque React Admin
2. Installation de l'environnement
3. Définition de l'arborescence du panel administrateur
4. Création de l'authProvider
5. Création du fichier principal de gestion App.js
6. Création des composants d'affichages
7. Création des composants de formulaire
8. Tests

B. Répartition des tâches

Pour la répartition des tâches, nous avons décidé de **travailler simultanément sur certaines tâches et séparément pour d'autres**.

Au sujet de la **base de données**, sa modélisation et sa création, nous avons **effectué les tâches ensemble**. Idem en ce qui concerne la **documentation**, les **installations** d'environnement et le **maquettage**, que ce soit pour l'API, l'application mobile, ou l'interface administrateur.

En ce qui concerne l'**API**, pour la **création des fichiers "Services", "Controllers" ou "Routes"**, nous nous sommes **répartis le travail en fonction des tables de base de données** qui allaient être impactées :

- Un collaborateur a développé les fichiers services, controllers, routes relatifs aux **utilisateurs**
- Un collaborateur a développé les fichiers services, controllers, routes relatifs aux **messages**
- Un collaborateur a développé les fichiers services, controllers, routes relatifs aux **salons de discussions**
- Nous nous sommes **répartis le développements des différents middlewares**
- Chacun a effectué ses **tests après développement** puis c**ommuniquait dessus** avec les autres collaborateurs.

A propos de l'**application mobile** nous avons travaillé ensemble du maquettage en passant par l'installation de l'environnement jusqu'au développement du premier système temporaire de navigation.
Ensuite, nous nous sommes **répartis les écrans à développer, toujours dans une logique de trame que suis l'utilisateur à partir de la navigation** :

- Un collaborateur a développé les écrans d'inscription/connexion/profil/modification du profil
- Un collaborateur a développé les des écrans de la liste des utilisateurs, la liste des conversations privées, et l'écran de discussion privée
- Un s'est occupé des écrans de la liste des espaces de discussion, de l'écran d'espace de discussion ainsi que de la navigation finale

Nous faisions **un point ensemble** tous les deux jours et réalisions des **tests globaux** ensemble afin de voir les **corrections à apporter**.

C. Outil de suivi

Afin d'effectuer un suivi de nos tâches, nous avons utilisé l'**outil en ligne et collaboratif Trello**, permettant de créer un **tableau des tâches à effectuer**. En effet, nous avons créé des "**Cards**" dans lesquelles nous avons établi des **checklists des sous-tâches à effectuer**. Nous les avons assigné à l'oral mais l'outil permettait de les assigner. Cet outil permet aussi de changer le statut de chaque Card du statut "**A faire**" vers "**En cours**" ou "**Terminé**".

Voici quelques captures d'écrans afin de vous montrer le Trello mis en place pour l'API au cours du projet :

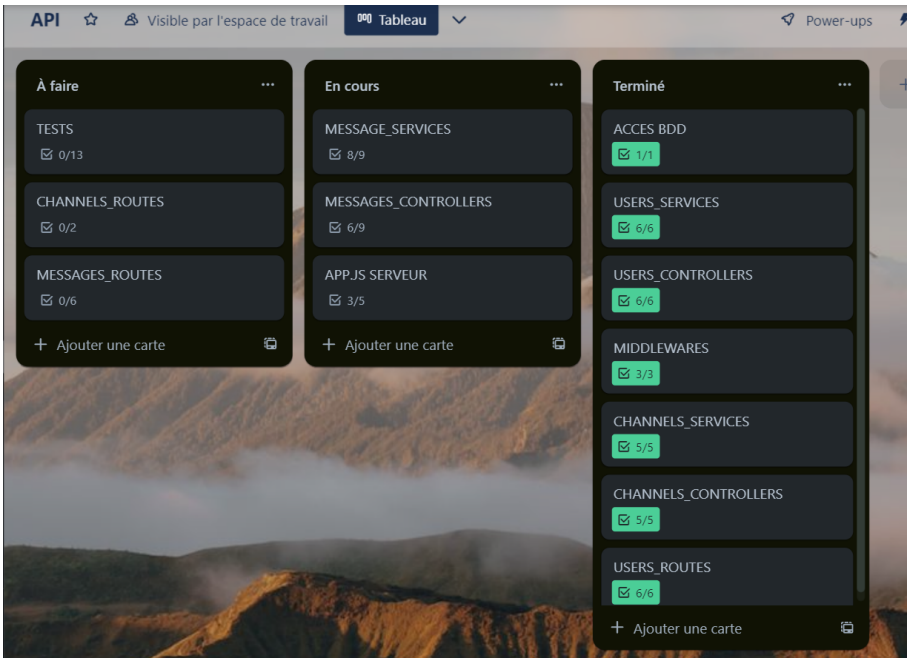
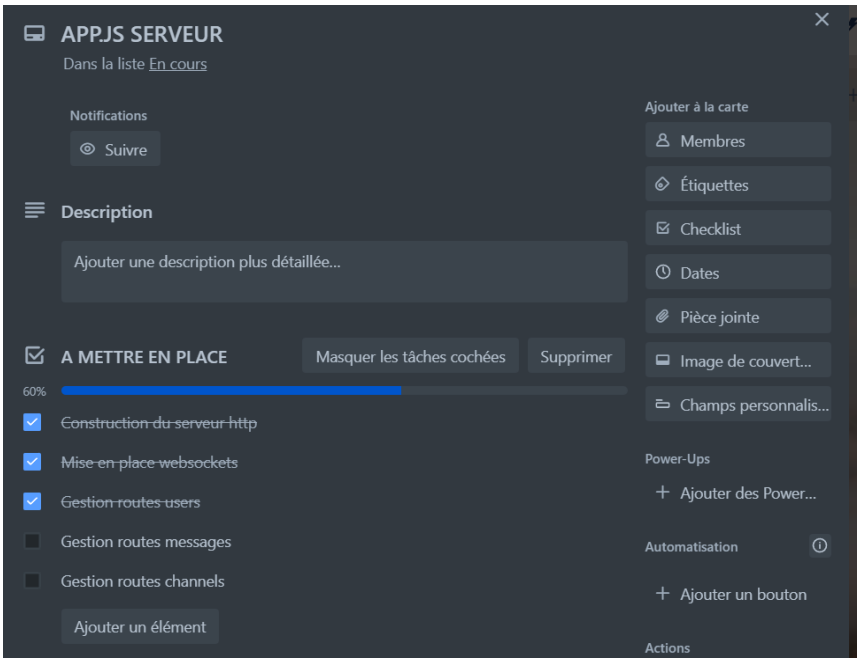


Tableau de gestion des tâches pour la conception de l'API avec les "Cards"



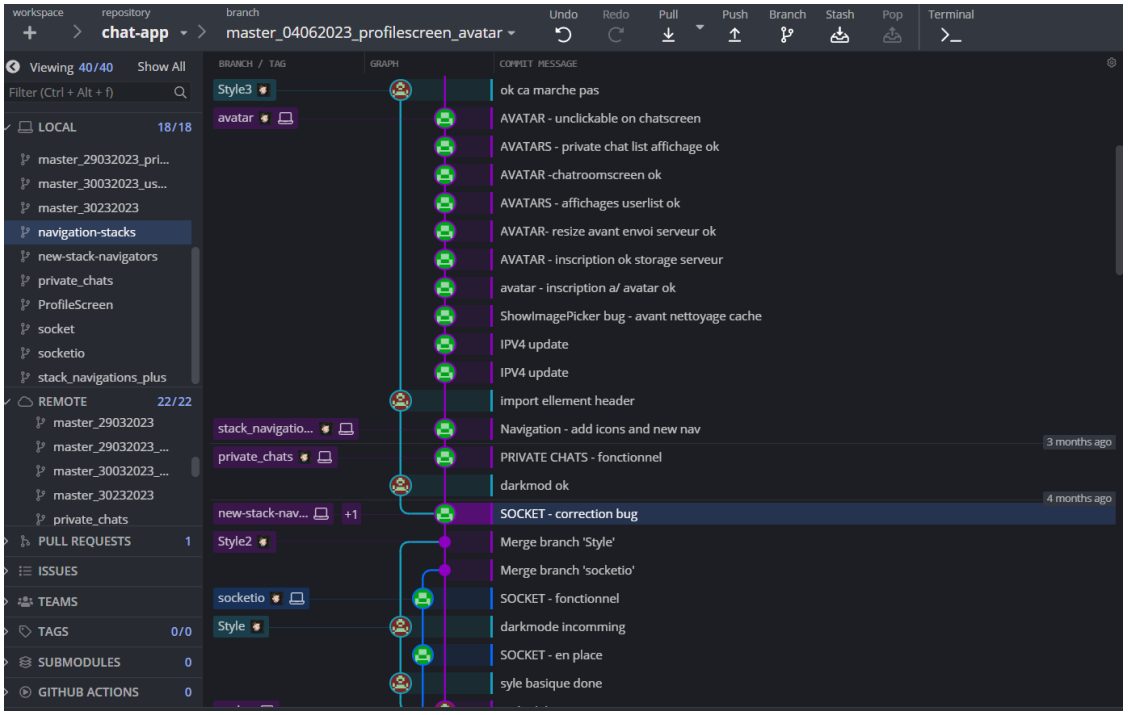
Contenu de la Card "APP.JS SERVEUR" en cours

D. Versionning

Pour la gestion du versionning, nous avons utilisé **Git**. En effet, nos **repository** sont stockés sur **Github**. Nous avons créé une **branche à chaque développement** d'une **fonctionnalité** / d'un **fichier** / d'un **écran** liés par la trame logique que suit l'utilisateur.

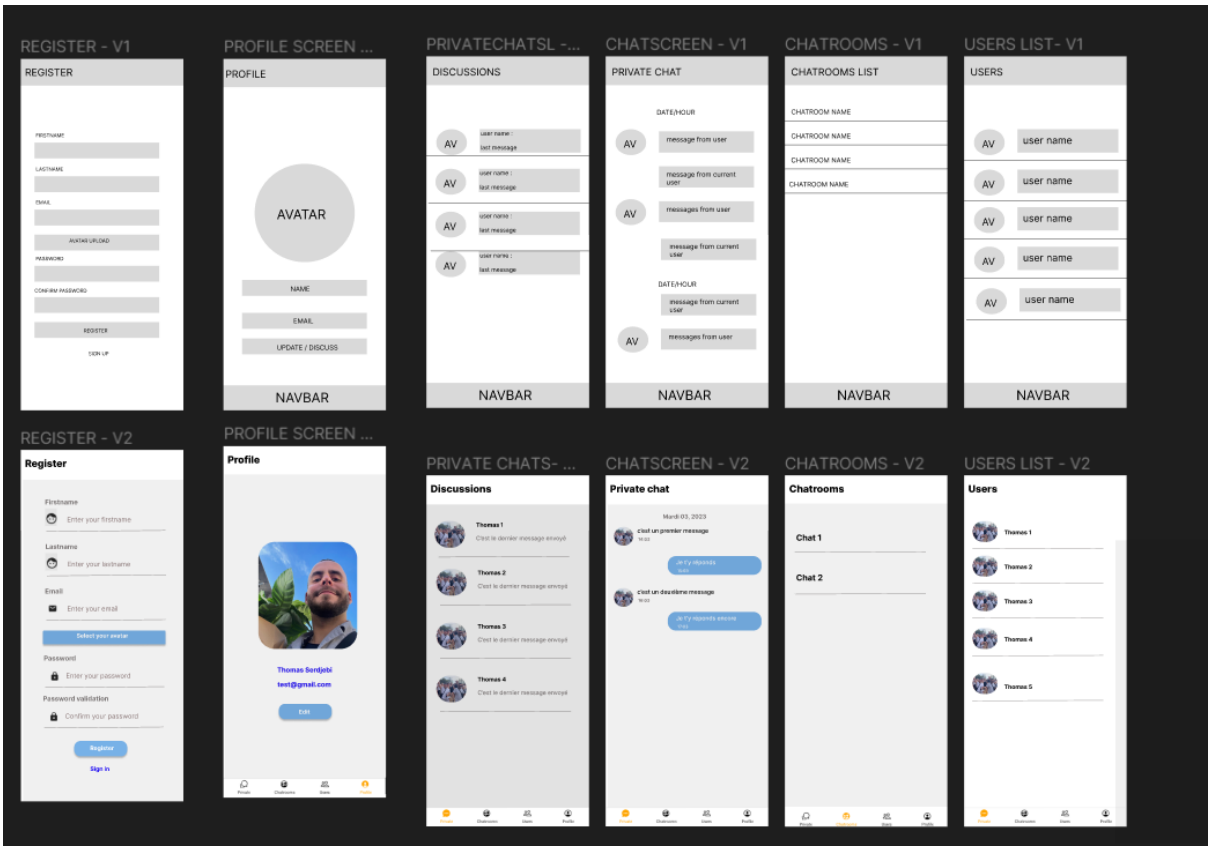
Nous effectuons **plusieurs "commits"** avec des sur les **branches spécifiques** puis, après avoir achevé un développement nous effectuions des tests, apportions les corrections et enfin nous **fusionnons la branche spécifique à la branche principale**.

Quand la majorité des développements étaient aboutis et afin d'apporter les dernières modifications suite à des tests globaux, nous avons changé de méthode et créé des branches avec le nom des écrans ainsi que la date de modification. Quand tous les tests étaient terminés, nous fusionnions ces branches spécifiques datées à la branche principale.



E. Maquettage

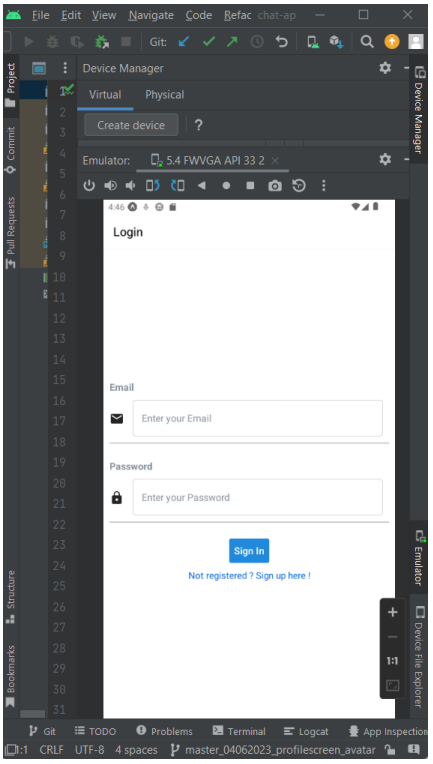
Le maquettage de l'application a été effectué avec l'outil Figma. Nous avons d'abord établi un wireframe basse fidélité avec seulement les blocs clés des différents écrans. Par la suite, nous avons établi un wireframe détaillé de chaque écran : style des formulaires, style de la topbar, couleurs et taille des différentes typographies, barre de navigation en bas de l'écran, forme et affichage des avatars ainsi que des messages.



F. Déploiement

Nous avons développé notre application à l'aide d'**Expo**. Le **déploiement d'une application** est un **processus** qui permet de **rendre l'application mobile disponible pour les utilisateurs**, et Expo est justement un **ensemble d'outils** qui facilite le développement d'applications en utilisant notamment React Native.

Nous avons donc installé **Node.js** et **Expo CLI** dans notre projet développé avec Visual Studio Code, puis créé le projet grâce à la **commande expo init** dans le terminal qui vient **structurer le projet de base** avec les fichiers et dossiers préconfigurés. Au cours du développement, nous avons utilisé la **commande 'expo start'** qui permet de **démarrer le serveur de développement Expo** et permet de **tester l'application** via notre **émulateur Android Studio**(permettant de générer une image de différents appareils de différentes tailles iPhone, Android, tablettes…) ou **directement sur notre téléphone** en scannant le **QR code** généré dans le terminal et ouvrant notre application dans l'application mobile Expo Go.



Expo offre aussi la possibilité de **publier l'application** et de la rendre disponible via le service d'hébergement d'Expo, de la distribuer par la génération de fichiers binaires ou de les soumettre aux stores respectifs.

V. Spécification techniques et extraits de code

A. La base de données

La base de données se compose **au départ de 4 tables** : users, messages, channels, roles. En ayant établi le **MCD** présent en annexe 1, et malgré **une relation de cardinalité 1/1 à 0/N** nous avons jugé nécessaire de créer une **table de liaison users_messages** présenté dans le **MLD** présent en annexe 2 pour les **messages privés** car la relation entre les deux tables contient des **attributs supplémentaires** qui permettent de définir par quel utilisateur est envoyé le message et à quel utilisateur il est destiné.

B. L'API

a. Structure générale

Vous pourrez trouver un **schéma de l'arborescence de l'API** en annexe 4. Hormi les fichiers générés par l'installation de l'environnement et des différentes librairies, on peut distinguer les dossiers et fichiers principaux :
- **un fichier app.js** qui appelle les différents fichiers de **route**, les **librairies** utilisées, les **websockets** mis en place.
- un sous **dossier config** contenant le **composant d'accès à la base de données**
- un sous **dossier api** contenant le sous **dossier assets** avec les différentes **images** utilisées (icônes, avatars..) ainsi que les trois sous **dossiers channel**, **message**, **users** contenant les **fichiers** "**Service**", "**Controllers**" et "**Routes**" spécifiques aux tables de la base de données avec lesquels l'API va interagir.
- un sous **dossier auth** contenant les **middlewares** relatifs à l'**authentification** des utilisateurs

b. Composants d'accès à la base de données

Le fichier **database.js** permet l'**accès à la base de données**. En effet, il utilise la fonction **createPool** du module **mysql12** afin de définir la **constante pool** qui permettra l'accès à la base de données. Les informations sont stockées dans **un fichier .env** qui n'est pas publié sur github lors du push des branches. Enfin la dernière ligne du fichier permet d'**exporter la constante** afin de l'utiliser dans les **fichiers Services** qui contiennent les **requêtes SQL**.

```
1  const { createPool } = require('mysql2');
2
3  const pool = createPool ({
4      port: process.env.DB_PORT,
5      host: process.env.DB_HOST,
6      user: process.env.DB_USER,
7      password: process.env.DB_PASS,
8      database: process.env.MYSQL_DB
9
10
11 });
12
13 module.exports = pool ;
```

Extrait fichierdatabas.js - const pool accès à la base de données

c. Structure et fonctions des fichiers “Services”

Les fichiers **.services** ont tous une **structure identique**. En effet, on en trouve un pour les **messages**, un pour les **utilisateurs** et un pour les **messages**. Chacun de ces fichiers va fournir différentes **fonctionnalités**.

En premier lieu, on appelle la **constante pool** contenue dans le fichier **database.js** qui permettra l’accès à la base de données. Ensuite le **module.exports** va permettre d’exporter toutes ces **fonctionnalités** qui seront appelées dans les **fichiers .controller**.

Ensuite, à l’intérieur des **accolades du modules.exports**, on définit donc **chaque fonctionnalité** de la façon suivante :

- **nom de la fonction et ses paramètres** (dans l’exemple ci-dessous data et callBack)
- création de **variables supplémentaires** nécessaires à l’exécution de la requête
- la **méthode pool.query contenant la requête SQL** liée à la **fonctionnalité** désirée, qui prend plusieurs paramètres :
  - **la requête SQL**
  - les paramètres qui vont permettre de **remplacer les valeurs dans la requête SQL** et sont extraits initialement du **paramètre data** ou des **autres variables supplémentaires** créées comme la date de création
  - **la fonction de rappel dit callBack** est utilisée pour retourner le résultat ou **gérer les erreurs**. On constate que s’il n’y a pas d’erreurs,seul le résultat est retourné, s’il y a une erreur, l’erreur est retournée.

```
const pool = require ("../../config/database");

module.exports = {

    //Create a user
    create: (data, callBack) => {
        const now = new Date(Date.now());
        const created = now.toISOString().slice(0, 19).replace('T', ' ');
        const defaultRole = 1;

        pool.query(
            'insert into users (firstname, lastname, email, password, role_id, created_at, updated_at, avatar) values (?,?,?,?,?,?,?,?)',
            [
             data.firstname,
             data.lastname,
             data.email,
             data.password,
             defaultRole,
             created,
             data.updated_at,
             data.imageURL
            ],
            (error, results, fields) => {
                if (error) {
                    console.log(error)
                    return callBack(error)
                }
                return callBack(null, results)
            }
        );
    },
```

Extrait fichier users.services.js - fonction create permettant de créer un utilisateur

Ainsi sont structurés les fichiers **.services**. **Les fonctions** sont **séparées** par des **virgules et** pourront donc être **exportées et utilisées** dans les **fichiers .controllers**.

d. Structure et fonctions des fichiers “Controllers”

Les fichiers ***.controller*** permettent d’effectuer la ***gestion des erreurs*** et de mettre en place les **mesures de sécurité sur les données entrantes saisies par l’utilisateur** et envoyées par le client. La **structure est identique pour chacun des fichiers** users.controller.js, channel.controller.js et message.controller.js.

- On **importe** d’abord les **fonctionnalités** liées aux contrôles effectués issu des fichiers **.services.js** :

```
const {
    create,
    getUserById,
    getUsers,
    updateUser,
    deleteUser,
    getUserByEmail,
} = require("./users.service");
```

Extrait fichier users.controller.js - import des fonctions du fichier users.service.js

- Ensuite on **importe** les **modules** et les **fonctions spécifiques** à partir de **différents packages** qui serviront dans nos **contrôles** et nos **exécutions de code** :

```
12  const { genSaltSync, hashSync, compareSync } = require ("bcrypt");
13  const { sign } = require("jsonwebtoken");
14  const path = require('path');
15  const fs = require('fs');
```

Extrait fichier users.controller.js - import des modules et fonctionnalités des packages

Dans un autre temps, le **module.exports** va aussi **contenir** toutes les **fonctionnalités de contrôle** qui vont être exécutées à l’appel de l’API via son nom de route.

```
module.exports = {

    //Creating a user
    createUser: (req, res) => {

        const body = req.body;

        //Vérifying all fields are fulfilled
        if (!body.firstname || !body.lastname|| !body.email || !body.password) {
            return res.status(500).json({
                success: 0,
                error: "You must enter your firstname, lastname, email and password."
            })
        } else {

            const namesRegex = "^[a-zA-ZÀ-ÿ -]+$"

            //Verifying firstname & lastname without numbers or special caracters except accents and "-"
            if (!body.firstname.match(namesRegex) || !body.lastname.match(namesRegex)){
                return res.status(500).json({
                    success: 0,
                    error: "Your firstname and lastname can only contain letters, accents, blank space and '-'."
                })
            }

            //Verifying email format
            if (!body.email.match(/^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$/)) {
                return res.status(500).json({
                    success: 0,
                    error: "Your email is incorrect."
                })
            }

            body.firstname = body.firstname.charAt(0).toUpperCase() + body.firstname.slice(1).toLowerCase();
            body.lastname = body.lastname.charAt(0).toUpperCase() + body.lastname.slice(1).toLowerCase();
            const salt = genSaltSync(10);
            body.password = hashSync(body.password, salt);
```

Extrait fichier users.controller.js - fonction de contrôle createUser appelée sur la route POST /api/users

Pour chaque **fonctionnalité de contrôle**, on suivra les étapes suivantes :

- **définition du nom de la fonction et ses paramètres** (req, qui contient l'objet de requête HTTP envoyée par le client et res, l'objet de réponse HTTP)
- **récupération du corps de la requête** qu'on va assigner à une ou plusieurs variables (const body = req.body)
- **application des contrôles sur les données entrantes** qui peuvent être par exemple :
  - contrôle des champs vides ou non renseignés
  - mise en place de regex pour vérifier la conformité des données saisie set envoyées et lutter contre les injections SQL
  - hachage du mot de passe
  - formatage des données

Dans le cas d'une **erreur**, on retourne un **statut de réponse** ainsi que deux données : **success à 0** et **error** qui permettra d'**afficher le message d'erreur** dans l'application. S'il n'y a **pas d'erreur**, on peut **appeler alors la fonction associée du fichier .service** en lui spécifiant les **paramètres de corps de données (body) ainsi que le tableau (err, results).**

```
create(body, (err, results) => {

    if (err) {
        return res.status(500).json({
            success: 0,
            error: "An error occured, try to register again."
        })
    };

    return res.status(200).json({
        success: 1,
        data: results
    }) ;

})
```

Extrait fichier users.controller.js - fonction de contrôle createUser - appel de la fonction create du fichier users.service.js

Si une **erreur** se produit lors de l'**exécution de la requête**, un **statut 500** est retourné avec une variable **success à 0 et un message d'erreur**. Si la requête est **bien exécutée**, cela retourne un **statut 200, success à 1 ainsi que le résultat de la requête** qui contient les données envoyées.

e. Structure des fichiers "Routes"

Les fichiers .router permettent de **construire les routes de l'API** qui seront **appelées dans l'application** afin d'utiliser les **fonctionnalités**. Ils sont également répartis selon les tables users, messages et channels. Voici leur structure :

- On **importe les fonctions issues des fichiers .controller** associés aux **routes** et aux **requêtes SQL** à exécuter
- On crée une **instance du routeur d'Express**, le framework pour NodeJs afin de **définir les routes et les actions correspondantes** pour gérer les demandes HTTP (get, post, put, delete…)

```
const {
    createUser,
    getUserById,
    getUsers,
    updateUser,
    deleteUser,
    login,
} = require("./users.controller");

const router = require("express").Router();
```

Extrait fichier users.router.js - import des fonctions de contrôle du fichier users.controller.js

- .On importe également les middlewares qui vont exécuter des fonctions intermédiaires, dont nous détaillerons le fonctionnement dans le point g. de cette section

```
//MIDDLEWARES

//Check if the user is connected
const { checkToken } = require ("../../auth/token_validation");
//Check if the user connected is an admin only
const { checkAdmin } = require ("../../auth/admin_validation");
//Check if the email entered already exists
const { checkExistingEmail } = require ("../../auth/email_validation");
```

Extrait fichier users.router.js - import des middlewares d'authentification

- Enfin on crée les **différentes routes spécifiques** en utilisant les **méthodes .get(), .post(), .patch(), .delete()** etc… . On définit pour chacune le **chemin d'accès "/", "/login"** par exemple **puis les fonctions importées à appeler** et à **exécuter les unes après les autres**. Ces routes seront **exportées grâce au module.exports**

```
22  //ROUTES : ACCESS FOR ALL USERS ----------------------------------------------------------
23  //Sign up rout => working
24  router.post("/", checkExistingEmail, createUser);
25  //Sign in Rout
26  router.post("/login", login);
27  //---------------------------------------------------------------------------------------
28  //ROUTES : ACCESS FOR CONNECTED USERS ----------------------------------------------------
29  //Route returning List with all users, user must be connected => working
30  router.get("/", checkToken, getUsers);
31  //Route returning names, emails from one user, user must be connected => working
32  router.get("/:id", checkToken, getUserById);
33  //Route updating the current user informations, user must be connected => working
34  router.patch("/", checkToken, updateUser);
35  //Route deleting the account of the current user, user must be connected => working
36  router.delete("/", checkToken, deleteUser);
37
38
39  module.exports = router;
```

Extrait fichier users.router.js - créations des routes

f. Le fichier app.js : serveur et racine des routes

Le fichier **app.js** permet de configurer le serveur Express en lui-même avec sa **gestion des routes** et d'autres **fonctionnalités** comme la mise en place des **sockets**, dont le fonctionnement sera développé ultérieurement dans ce dossier.

```
const express = require("express");
const http = require('http');
const app = express();
const server = http.createServer(app);
const cors = require('cors');
```

```
app.use(cors());
```

Extrait fichier app.js - configuration du serveur

Ces lignes de codes viennent configurer le serveur. On crée une **instance du framework Express, du module HTTP de Node.js(http), de l'application Express(app)** et on créé le **serveur http(server)** avec la fonction **createServer** et on **instancie le middleware CORS**(Cross Origin Resource Sharing), qui sera activé par la suite afin de pouvoir gérer les requêtes provenant des différentes origines (application, back-office administrateur). Par la suite **on instancie les fonctionnalités des différents fichiers de routes** dont le fonctionnement a été explicité dans le point précédent.

```
// App
const userRouter = require("./api/users/users.router");
const messageRouter = require("./api/message/message.router");
const channelRouter = require("./api/channel/channel.router");
```

Extrait fichier app.js - instanciation des modules de routes

On active également le **middleware express.json** permettant de **gérer les données JSON** et les requêtes **POST**. On définit les **routes associées aux routeurs spécifiques** pour les différents **endpoints** de l'API. Pour la dernière ligne de code, on **configure Express** pour **servir les fichiers statiques** enregistrés dans le sous répertoire api/assets/avatars.

```
//pour post json
app.use(express.json());

// App routes
app.use("/api/users", userRouter);
app.use("/api/message", messageRouter);
app.use("/api/channel", channelRouter);
app.use('/avatars', express.static('api/assets/avatars'));
```

Extrait fichier app.js - définitions des routes

Enfin on utilise la **méthode listen pour démarrer le serveur HTTP et écouter les requêtes entrantes** sur le port spécifié. En utilisant la commande **"npm start"**, les fonctions du fichier app.js sont exécutées et donc le **serveur démarre**, comme paramétré dans le **package.json**.

```
server.listen(process.env.APP_PORT, ()=> {
    console.log("Server up and running on PORT:", process.env.APP_PORT);
});
```

```
{
  "name": "api_mobile_chat",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon app.js"
  },
```

Extrait fichier package.json

g. Les middlewares

Les middlewares sont des **fonctions intermédiaires** qui vont permettre de "**pré-traiter" les requêtes** envoyées par le client, et notamment exécuter des **fonctionnalités** qui peuvent être **communes** aux différentes routes, comme re-traiter les données envoyées avant exécution ou vérifier l'**authentification** d'un utilisateur par exemple.

On peut considérer que les **fichiers .controllers** sont des **middlewares**, puisqu'il viennent effectuer la **gestion des erreurs**, reformater les données parfois, avant d'exécuter ou non la requête SQL.

Pour mieux comprendre **la logique d'un middleware**, je vais reprendre l'exemple de la **vérification du token des routes** ci-dessous :

```
//ROUTES : ACCESS FOR CONNECTED USERS ----------------------------------------------------
//Route returning List with all users, user must be connected => working
router.get("/", checkToken, getUsers);
//Route returning names, emails from one user, user must be connected => working
router.get("/:id", checkToken, getUserById);
//Route updating the current user informations, user must be connected => working
router.patch("/", checkToken, updateUser);
//Route deleting the account of the current user, user must be connected => working
router.delete("/", checkToken, deleteUser);
```

Extrait fichier app.js - routes contenant le middleware checkToken

Certaines routes du projet sont accessibles à tous les utilisateurs connectés, comme la route "users/" et la route"users/login", qui sont les deux premières de la capture d'écran précédente. Cependant, **les autres routes sont uniquement disponibles pour les utilisateurs connectés**. On peut constater l'**appel du middleware "checkToken"** avant d'**appeler la fonction du fichier .controller** qui, sans présence d'erreur, va elle-même **appeler la fonction d'exécution de la requête sql définie dans le fichier .service**.

1. La fonction de connexion

Lorsqu'un utilisateur se connecte, un "token" (jeton) d'authentification est généré, et voici comment. En premier lieu, **la fonction getUserByEmail** dans le **fichier users.service.js** permet de **récupérer l'ensemble des informations d'un utilisateur** après qu'il ait rempli le **formulaire de connexion**.

```
//Get a user by email to login
getUserByEmail: (email, callBack) => {
    pool.query(
        'select * from users where email = ?',
        [email],
        (error, results, Fields) => {
            if (error) {
                console.log(error)
                return callBack(error)
            }
            return callBack(null, results[0])
        }
    )
},
```

Extrait fichier users.service.js - fonction getUserByEmail

La **route POST users/login** est appelée dans l'application mobile lorsque l'utilisateur appuie sur le **bouton connexion** de la page d'accueil de l'application. Cette route va donc faire appel à la **fonction login du fichier users.controllers**.

```
//Login
login: (req,res) => {
    const body = req.body;

    const salt = genSaltSync(10);
    // console.log(hashSync(body.password, salt));

    getUserByEmail(body.email, (err, results) => {
        if (err) {
            console.log(err)
        }

        if (!results) {
            // res.statusCode = 401;
            return res.json({
                success: 0,
                data: "Invalid credentials. Please try again. (1)"
            });
        }

        const result = compareSync(body.password , results.password );

        if (result) {
            results.password = undefined;
            const jsontoken = sign({ result: results}, process.env.TK_KEY, {
                expiresIn : "2h"
            }) ;

            return res.json({
                success: 1,
                message: "Logged in successfully.",
                token: jsontoken,
            });

        } else {
            console.log(err)
            return res.json({
                succes: 0,
                error: "Invalid credentials. Please try again. (2)",
            });
        }
```

Extrait fichier users.controller.js - fonction login appelant la fonction getUserByEmail du fichier users.service.js

Les **données envoyées par l'utilisateur** sont **stockées** dans la **constante body**, et qui contient donc l'**email** et le **mot de passe**. Elle appelle la **fonction getUserByEmail** à partir du mail saisi par l'utilisateur. Si un **résultat est trouvé**, les **données de l'utilisateur** sont donc **récupérées** dans le **tableau results**. Ensuite on vient **comparer le mot de passe saisi** avec le **mot de passe crypté** enregistré dans les **données utilisateur** à l'aide de la **fonction compareSync** de la **bibliothèque bcrypt** ayant servi à crypter le mot de passe.Si **les mots de passe correspondent**, alors on **supprime le mot de passe du tableau** des informations de l'utilisateur, et on vient **générer le token** avec la **fonction sign** de la bibliothèque **jsonwebtoken**. On y **injecte les informations de l'utilisateur** (excepté le mot de passe) ainsi qu'une **clé secrète**.

2. Le middleware token_validation

La **fonction checkToken** contenue dans ce middleware est appelée sur toutes les routes qui **nécessitent pour l'utilisateur d'être connecté** afin d'accéder aux fonctionnalités liées aux écrans de l'application(récupération de la liste des utilisateurs, accès aux salons de discussions….).

```
//ROUTES : ACCESS FOR CONNECTED USERS ----------------------------------------------------
//Route returning List with all users, user must be connected => working
router.get("/", checkToken, getUsers);
//Route returning names, emails from one user, user must be connected => working
router.get("/:id", checkToken, getUserById);
//Route updating the current user informations, user must be connected => working
router.patch("/", checkToken, updateUser);
//Route deleting the account of the current user, user must be connected => working
router.delete("/", checkToken, deleteUser);
```

Extrait fichier app.js - routes nécessitant que le token de l'utilisateur soit vérifié

Comment fonctionne donc la vérification du token ?

```
const { verify } = require ("jsonwebtoken")

module.exports = {
    checkToken : (req, res, next) => {
        let token = req.get("authorization")
        if(token) {
            token = token.slice(7);
            verify(token, process.env.TK_KEY,(err, decodedToken) => {
                if(err) {
                    res.statusCode = 401;
                    res.json({
                        success:0,
                        error: "Invalid token."
                    }) ;
                } else {
                    req.user = decodedToken
                    next();
                }
            })
        } else {
            res.statusCode = 401;
            res.json({
                success:0,
                error: "Access denied, unauthorized user."
            })
        }
    }

}
```

Le **token** est récupéré dans la valeur "**authorization**" de l'**entête de la requête HTTP**. S'il n'y a **pas de valeur, l'accès est refusé**. Si un token est **bien récupéré**, on vient alors **vérifier son authenticité** à l'aide de la **fonction verify** de la bibliothèque **jsonwebtoken** qui prend en **paramètre** le **token** et la **clé secrète** et va retourner soit une erreur, soit le **token décodé**, c'est-à-dire **les informations de l'utilisateur**. Si l'authentification du token est valide, alors on **injecte les informations de l'utilisateur dans l'objet de requête HTTP req à l'index user.** Ces informations pourront être utilisées dans les écrans/fonctions suivantes qui prennent req en paramètre, et on pourra donc exécuter la fonction suivante.

Extrait fichier app.js - routes nécessitant que le token de l'utilisateur soit vérifié

C. L'application mobile

a. Architecture générale

Vous pourrez trouver un **schéma de l'arborescence de l'application** mobile en annexe 5. Hormi les fichiers générés par l'installation de l'environnement et des différentes librairies, on peut distinguer les dossiers et fichiers principaux :
- **un fichier App.js** qui vient configurer la **navigation** dans l'application
- **un sous dossier src/screens** qui contient l'ensemble des **écrans de l'application**
- **un sous dossier assets** qui contient les images et icônes présentes dans l'application

b. Maquettage

Pour le **maquettage**, nous avons utilisé l'outil **FIGMA**. Vous pourrez en retrouver une partie en annexe 6. Nous avons d'abord défini **une page de connexion/inscription**, puis une **navbar** et la **structure générale** de chaque page.

c. Installation de l'environnement

Nous avons développé cette application dans un **environnement Node.js** et avons également installé **React Native**, le **framework Javascript** utilisé pour développer des applications multiplateformes ainsi qu'**Expo**, qui est un ensemble d'outils qui facilitent le développement d'applications React Native.

d. Structure de la navigation

La navigation est gérée dans le **fichier App.js**. Elle nécessite tout d'abord d'**importer les différentes fonctions** qui viennent générer les écrans de l'application :

```
1   import React from 'react'
2   import 'react-native-gesture-handler';
3   import ChatroomScreen from './src/screens/ChatroomScreen';
4   import ChatroomsListScreen from './src/screens/ChatroomsListScreen';
5   import PrivateChatsListScreen from './src/screens/PrivateChatsListScreen';
6   import LoginScreen from './src/screens/LoginScreen';
7   import ProfileScreen from './src/screens/ProfileScreen';
8   import RegisterScreen from './src/screens/RegisterScreen';
9   import UsersListScreen from './src/screens/UsersListScreen';
10  import PrivateChatScreen from './src/screens/PrivateChatScreen';
11  import UpdateProfileScreen from './src/screens/UpdateProfileScreen';
12  import { Ionicons } from '@expo/vector-icons';
13
```

Extrait fichier App.js - imports des fonctions qui génèrent les écrans

Ensuite nous importons les **composants de navigation** fournis par les **bibliothèques de React Native** :

```
//Navigations components
import {NavigationContainer} from '@react-navigation/native'
import {createStackNavigator} from '@react-navigation/stack'
import {createBottomTabNavigator} from '@react-navigation/bottom-tabs'
```

Extrait fichier App.js - imports des composants de React Native

- **NavigationContainer**, qui va être le **contenant racine de la navigation** et va la gérer dans sa **globalité** en fournissant le **contexte de navigation** à tous les composants.
- **createStackNavigator**, qui va permettre de **générer des "piles" de navigation**, en empilant successivement plusieurs écrans les uns avec les autres, en suivant la trame logique du parcours utilisateur.
- **createBottomTabNavigator,** qui va permettre de **générer une barre de navigation** située en bas de l'écran, et qui dans notre cas, nous donnera **accès alors pour chaque bouton** à un **navigateur empilé (stack navigator) différent**.

Enfin nous utilisons **la fonction createStackNavigator** que nous instancions à des constantes pour créer nos différents stack navigators ainsi que createBottomTabNavigator pour créer la barre de navigation. Nous allons maintenant décortiquer la structure de la navigation en elle-même.

```
22  const AuthStack = createStackNavigator();
23  const PrivateStack = createStackNavigator();
24  const ChatroomStack = createStackNavigator();
25  const ProfileStack = createStackNavigator();
26  const UsersStack = createStackNavigator();
27
28  const AppStack = createBottomTabNavigator();
```

Extrait fichier App.js - constante instanciant des stackNavigator et un bottomNavigator

1. Les stacks navigators (navigateurs empilés)

Comme expliqué précédemment, les stacks navigators permettent de créer un empilement dans lequel on pourra naviguer d'un écran à l'autre. Voici quelques exemples :

```
//Stack for the sign in/ signup
function AuthStackScreen() {
return (
  <AuthStack.Navigator>
    <AuthStack.Screen name="Login" component={LoginScreen} />
    <AuthStack.Screen name="Register" component={RegisterScreen} />
  </AuthStack.Navigator>
);
}
```

Extrait fichier App.js - navigateur empilé AuthStackScreen

Dans ce premier exemple, on **génère un stack navigator** pour l'**authentification**. La **constante AuthStack** a été créée plus haut. On indique donc que ce stack navigator va **contenir les deux écrans LoginScreen et RegisterScreen** dont les noms à l'affichage seront respectivement **Login** et **Register**. Ainsi dans ce navigateur on peut **passer de l'écran de connexion à l'écran d'inscription** et revenir en arrière pour basculer sur l'écran de connexion.

```
//Stack for the PrivateChats Screen => Private chats list, private chat room, user profile
function PrivateStackScreen() {
  return(
    <PrivateStack.Navigator>
      <PrivateStack.Screen name="Discussions" component={PrivateChatsListScreen} options={{headerLeft: null}} />
      <PrivateStack.Screen name="Private Chat" component={PrivateChatScreen} options={{}} />
      <PrivateStack.Screen name="Profile" component={ProfileScreen} options={{ }} />
    </PrivateStack.Navigator>
  )
}
```

Extrait fichier App.js - navigateur empilé PrivateStackScreen

Dans ce deuxième exemple, on **génère un stack navigator PrivateStackScreen** dont le premier écran sera la **liste des discussions privées de l'utilisateur**. Ainsi, dans ce navigateur, l'utilisateur pourra **cliquer sur une conversation** pour accéder à l'**écran de la discussion privée** avec un utilisateur (**PrivateStackScreen**), et dans cet écran, s'il **clique sur le nom de l'utilisateur**, il aura accès à **son profil**. Il pourra donc également naviguer en arrière s'il le souhaite.

```
//Stack for the user's profile navigation => profile, update profile, update
function UsersStackScreen() {
  return(
    <UsersStack.Navigator>
      <UsersStack.Screen name="Persons" component={UsersListScreen} options={{}} />
      <UsersStack.Screen name="Profile" component={ProfileScreen} options={{ }} />
      <PrivateStack.Screen name="Private Chat" component={PrivateChatScreen} options={{}} />
    </UsersStack.Navigator>
  )
}
```

Extrait fichier App.js - navigateur empilé UsersStackScreen

Dans ce dernier exemple, le **navigateur UsersStackScreen**, le premier écran et le **UsersListScreen** qui affiche la **liste des utilisateurs de l'application**. En **cliquant sur un utilisateur**, on aura accès au **profil de l'utilisateur sélectionné**, ProfileScreen, et un bouton dans cet écran permettra d'**ouvrir la conversation privée** avec cet utilisateur.

Les deux derniers navigateurs empilés sont le **ChatroomStackScreen** qui amène l'utilisateur sur l'écran **ChatroomsListScreen** qui affiche la liste des salons de discussions et à partir duquel il aura accès au **ChatroomScreen**, l'écran d'un salon de discussion, ainsi que le **ProfileStackScreen** qui amène d'abord l'utilisateur sur son **profil** et lui permettra aussi d'accéder à l'écran de modification de ses informations.

2. La barre de navigation

```
function AppStackScreen() {
  const screenOptions = ({ route }) => ({
    tabBarIcon: ({ color, size, focused }) => {
      let iconName;

      switch (route.name) {
        case "Private":
          iconName = focused ? "chatbubble-ellipses" : "chatbubbles-outline";
          break;
        case "Chatrooms":
          iconName = focused ? "people-circle" : "people-circle-outline";
          break;
        case "Users":
          iconName = focused ? "people" : "people-outline";
          break;
        case "Profile":
          iconName = focused ? "person-circle" : "person-circle-outline";
          break;
        default:
          iconName = "";
      }

      return <Ionicons name={iconName} size={size} color={color} />;
    },
    tabBarActiveTintColor: "orange",
    tabBarInactiveTintColor: "black",
    tabBarLabelStyle: {
      fontSize: 12,
    },
    tabBarStyle: {
      backgroundColor: "white",
    },
  });
```

La barre de navigation est générée par la **fonction AppStackScreen**. On y définit d'abord la **constante screenOptions** qui va permettre de gérer les options d'affichage de la barre en fonction du nom de la page où on se situe (icône, couleur en fonction du focus, taille, couleur de fond….) .

Extrait fichier App.js - barre de navigation AppStackScreen

Enfin, la **fonction AppStackScreen** retourne donc les **composants de la barre de navigation**. On y injecte les différents **stacks navigators** générés précédemment. Lorsque l'utilisateur **clique** sur une **icône**, il aura donc **accès aux écrans** en fonction du stack navigator associé.

```
return (
  <AppStack.Navigator screenOptions={screenOptions}>
    <AppStack.Screen name="Private" component={PrivateStackScreen} options={{ headerShown: false }} />
    <AppStack.Screen name="Chatrooms" component={ChatroomStackScreen} options={{ headerShown: false}} />
    <AppStack.Screen name="Users" component={UsersStackScreen} options={{ headerShown: false }} />
    <AppStack.Screen name="Profile" component={ProfileStackScreen} options={{ headerShown: false}} />
  </AppStack.Navigator>
);
```

Extrait fichier App.js - fonction AppStackScreen

Donc si l'utilisateur clique sur l'**icône de l'écran Users** dont le **composant** est le **UsersStackNavigator**, il sera amené en premier lieu sur la liste des utilisateurs, et aura accès aux écrans suivants qui y sont empilés, c'est-à-dire l'écran du profile utilisateur puis l'écran de discussion privée.

3. La navigation centralisée dans le NavigationContainer

Finalement, la **fonction principale App** de l'écran vient **centraliser la navigation** dans le **NavigationContainer**. Dans le navigateur principal, on accède tout d'abord au **stack navigateur Auth pour la connexion/inscription**.

```
const Stack = createStackNavigator();

export default function App() {

  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Auth">
        <Stack.Screen name="Auth" component={AuthStackScreen}  options={{headerShown: false}} />
        <Stack.Screen name="App" component={AppStackScreen} options={{headerShown: false, headerLeft:null}}  />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

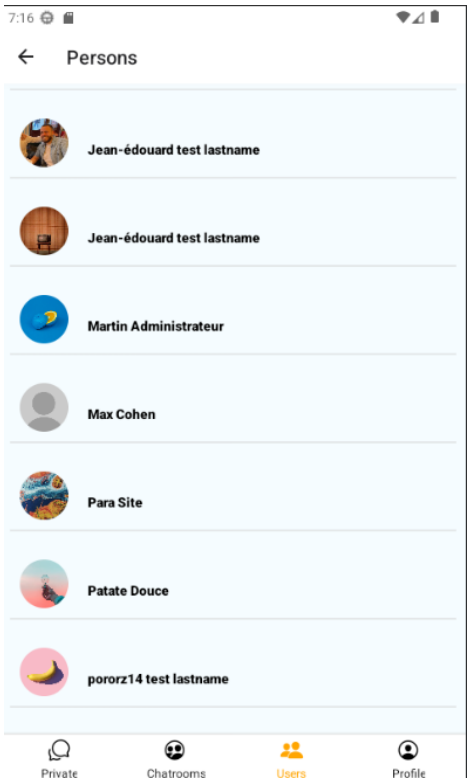
Extrait fichier App.js - fonction principale App

Une fois l'**utilisateur connecté**, il sera redirigé vers le **stack App**, avec donc la **barre de navigation**. La redirection après connexion redirige l'utilisateur par défaut vers la liste de ses discussions privées.

```
props.navigation.navigate('App',{
  screen: 'Private',
  headerLeft:null

})
```

Extrait fichier LoginScreen.js - redirection vers les discussions privées après connexion

e. Structure des screens



Les **"screens" (écrans)** sont des **composants** de l'application. Dans notre projet, ils sont générés par une **fonction fléchée** qui prend en **paramètres les "props"**, des données transmises de composants parents à composants enfants, et **retourne un élément React**.

Prenons l'exemple de la **structure du fichier UsersListScreen.js**, retournant l'ensemble des utilisateurs de l'application dans l'écran comme sur la capture d'écran ci à gauche.

Ecran Persons accessible via le bouton Users de l'application
Liste des utilisateurs

1. Import des modules et composants.

```
import React, { useState, useEffect} from 'react';
import { View, Text, ScrollView, StyleSheet, StatusBar,  TouchableOpacity,  } from 'react-native';
import { Image, FlatList } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';
import jwtDecode from 'jwt-decode';
```

Extrait fichier UsersListScreen.js - imports des divers modules et composants

On **importe** d'abord les **modules** et **composants** depuis les **bibliothèques React Native** et d'autres afin de construire cet écran. Le **useState** permet de **définir l'état des variables users** et **loading**. La **FlatList** permettra de générer la **liste des utilisateurs**, l'**image** de charger l'**avatar** des utilisateurs, l'**asyncStorage** permettra de récupérer le **token de l'utilisateur** stocké à sa connexion afin d'exécuter l'**appel à notre API**, et le **jwtDecode** permettra de **décoder le token** afin de récupérer les informations de l'utilisateur.

2. Définition du composant UsersListScreen

```
const UsersListScreen = (props) =>{

    const [users, setUsers] = useState([]);
    const [loading, setLoading] = useState(true);
```

Ensuite, nous définissons le **composant UsersListScreen** qui va contenir tous les **éléments** et **fonctions** qui vont **générer** cet **écran**. Les composants screens sont définis comme des fonctions prenant en paramètres les props. On initialise immédiatement la variable users qui contient la liste des informations de tous les utilisateurs et la variable loading qui permettra d'afficher que la page se charge en cas de lenteur, avec le hook useState.

3. La fonction getAllUsers

La **fonction getAllUsers** est une **fonction asynchrone** qui vient **appeler l'API** afin de **récupérer** la **liste des utilisateurs** et l'**affecter** à la variable **processedUsers**. Elle est **contenue** dans le **composant UsersListScreen**. Voici sa structure :

3.1. Définition des constantes

Dans un premier temps, nous définissons les **différentes constantes** qui vont nous être utiles. La fonction getAllUsers est **asynchrone** puisqu'il sera nécessaire de **récupérer le token** de l'utilisateur grâce à l'**asyncStorage**.

```
const unknownAvatar = 'https://icon-library.com/images/unknown-person-icon/unknown-person-icon-4.jpg'
const userToken = await AsyncStorage.getItem('user_token');
const decodedToken = jwtDecode(userToken)
const idUser = decodedToken.result.id_user
```

Extrait fichier UsersListScreen.js - création des constantes

La **constante unknownAvatar** permettra d'**afficher une image** représentant un **individu inconnu** si l'utilisateur n'a pas souhaité s'inscrire en enregistrant un avatar. La **const userToken** se voit affecter le **token de l'utilisateur** qui a été stocké grâce à l'**AsyncStorage** sous le nom d'item '**user_token**'. La **constante decodedToken** vient **décoder le token** et donc contient le tableau **d'informations de l'utilisateur** actuellement **connecté**. Pour plus de simplicité, nous avons isolé l'id du user connecté, vous verrez par la suite pourquoi.

3.2. Appel de l'API

Dans un second temps nous utilisons la **fonction fetch** qui prend en **paramètre** la **route de l'API** que nous désirons utilise, la méthode '**GET**' car nous souhaitons **récupérer** des **informations** ainsi que les **headers** et notamment l'**Authorization** qui est le **token** enregistré dans la variable **userToken**. La réponse sera ensuite chargée au **format json** et retraitée par la suite.

```
fetch('http://192.168.0.14:3000/api/users', {
    method: 'GET',
    headers:{ Authorization: 'Bearer ' + userToken },
})
.then(response => response.json())
.then(response =>{
```

Extrait fichier UsersListScreen.js - appel de la route api/users et paramètres

3.3. Traitement de la réponse

Le **traitement de la réponse retraite la réponse pour lui donner un format spécifique :**

- si l'utilisateur courant est présent dans la liste retournée par l'API, on le supprime de cette liste
- si l'utilisateur n'a pas d'avatar, alors il faut remplacer la donnée avatar par la constante unknownAvatar qui permettra d'afficher une icône d'une personne inconnue
- si l'utilisateur à un avatar, il faudra modifier le chemin d'accès afin qu'il soit fonctionnel pour que ce dernier soit afficher

```
const processedUsers = response.data.map(user => {


  if (user.id_user === idUser){
    return null;
  }

  if (!user.avatar) {
    return {
      ...user,
      avatar: unknownAvatar
    }
  } else {
    return {
      ...user,
      avatar: user.avatar.replace("localhost", "192.168.0.14"),
    }
  }
}).filter(Boolean);
```

On définit donc la **constante processedUsers** en utilisant la **méthode .map** qui va permettre de **parcourir** le **tableau de réponse** retourné par l'API pour **chaque utilisateur**. C'est ici que l'on vient **comparer l'id de chaque user avec l'id du user actuellement connecté** afin de le supprimer de la liste pour qu'il ne voit pas son profil s'afficher en **retournant null** si les **deux ids sont identiques**. S'il n'y a **pas d'avatar**, on retourne l'ensemble des informations de l'utilisateur excepté l'**avatar** qui va prendre la **valeur de unknownAvatar**.

Extrait fichier UsersListScreen.js - const processedUsers retraitant la réponse de l'API

En revanche, si les informations contiennent le **chemin d'accès vers l'avatar**, on le **transforme** en remplaçant le localhost par l'adresse **IPV4 utilisée par l'API**. Enfin on utilise la **méthode .filter** pour **supprimer les éléments nulls** de la constante processedUsers.

3.4. Définition de l'état des variables users et loading

Après retraitement de la réponse, on **définit l'état des variables users** et **loading** en utilisant le **useState**. La **variable users** contient désormais le **tableau retraité de la réponse contenu** dans la **constante processedUsers** et **loading est set à false** afin de pouvoir effacer l'affichage du Loading…. et afficher la liste des utilisateurs.

```
setUsers(processedUsers);
setLoading(false);
```

Extrait fichier UsersListScreen.js - définition de l'état de users et loading

3.5. Gestion des erreurs

En cas d'erreur, on va afficher le message d'erreur dans la console :

```
  .catch(function(error) {
      console.log('There has been a problem with your fetch operation: ' + error.message);
  })
}
```

Extrait fichier UsersListScreen.js - catch des erreurs potentielles en appelant l'API

4. Le Hook useEffect

Le **Hook useEffect** est utilisé pour **exécuter des fonctions** au moment du **rendu initial de la page**. Dans notre cas, ce hook va donc **appeler la fonction asynchrone getAllUsers** à l'intérieur de la **constante UsersListScreen**.

```
useEffect(() => {
    getAllUsers();
}, []);
```

Extrait fichier UsersListScreen.js - useEffect appelant la fonction getAllUsers

5. Le style

Le style est généré dans la **constante styles** qui vient utiliser la **méthode create** du **module StyleSheet** de **React Native**. On y définit le **style des différents éléments** qui vont être retournés : le conteneur principal, le conteneur d'un utilisateur, le style du nom ainsi que celui de l'avatar.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#F5FCFF',
  },
  userContainer: {
    flex: 1,
    flexDirection:'row',
    margin: 10,
    padding: 10,
    borderBottomWidth: 1,
    borderColor: '#d3d3d3',
  },
  userName: {
    fontSize: 14,
    marginLeft: 20,
    paddingTop:'5%',
    fontWeight:'900'
  },
  avatar: {
    width: 50,
    height: 50,
    borderRadius: 25,
    marginColor: '#black'
  },
});
```

Extrait fichier UsersListScreen.js - définition du style des composants dans la constante styles

6. Rendu des composants

```
return (
  <View style={styles.container}>

    <FlatList
      data={users}
      showsVerticalScrollIndicator={false}
      showsHorizontalScrollIndicator={false}
      renderItem={({ item }) => (
        <TouchableOpacity
          style={styles.userContainer}
          onPress={() => {
          props.navigation.navigate('Profile', {
            id_user: item.id_user,
          });
        }}>
        <Image
          style={styles.avatar}
          source={{uri: item.avatar}}
        />
        <Text
          style={styles.userName}
        >
          {item.firstname} {item.lastname}
        </Text>
        </TouchableOpacity>
      )}
      keyExtractor={item => item.id_user.toString()}
    />
  </View>
);
```

La constante **UsersListScreen** vient retourner un **ensemble de composants imbriqués** les uns dans les autres :

- un **composant View** auquel on applique le **style** défini auparavant et qui contient tous les autres
- le **composant FlatList** qui va **générer la liste des utilisateurs** et qui prend plusieurs **paramètres** comme le **paramètre data**, auquel on affecte le **tableau users**, le **renderItem** qui va définir pour **chaque users** des **paramètres d'affichage** de la liste, et le **keyExtractor** qui retourne pour chaque élément de la liste une valeur unique qui représente la clé de l'élément.

.Extrait fichier UsersListScreen.js - rendu des composants

- **le composant TouchableOpacity** contenu dans la **FlatList** qui va rendre chaque **item cliquable** vers le **profil de l'utilisateur** via les **props de navigation** vers l'écran Profile et qui envoie en **paramètre dans la navigation l'id du user** sur lequel on clique
- **le composant Image** contenu dans le **TouchableOpacity** auquel on va pouvoir appliquer le style et donner **la source des avatars des utilisateurs**
- Le **composant Text** contenu dans **TouchableOpacity** pour le prénom et le nom de l'utilisateur.

7. Export du screen

Enfin, **on exporte la constante UsersListScreen** puisqu'elle sera **importée dans le fichier App.js** et utilisée dans les différents **stacks navigators**.

```
export default UsersListScreen;
```

Extrait fichier UsersListScreen.js - export du composant

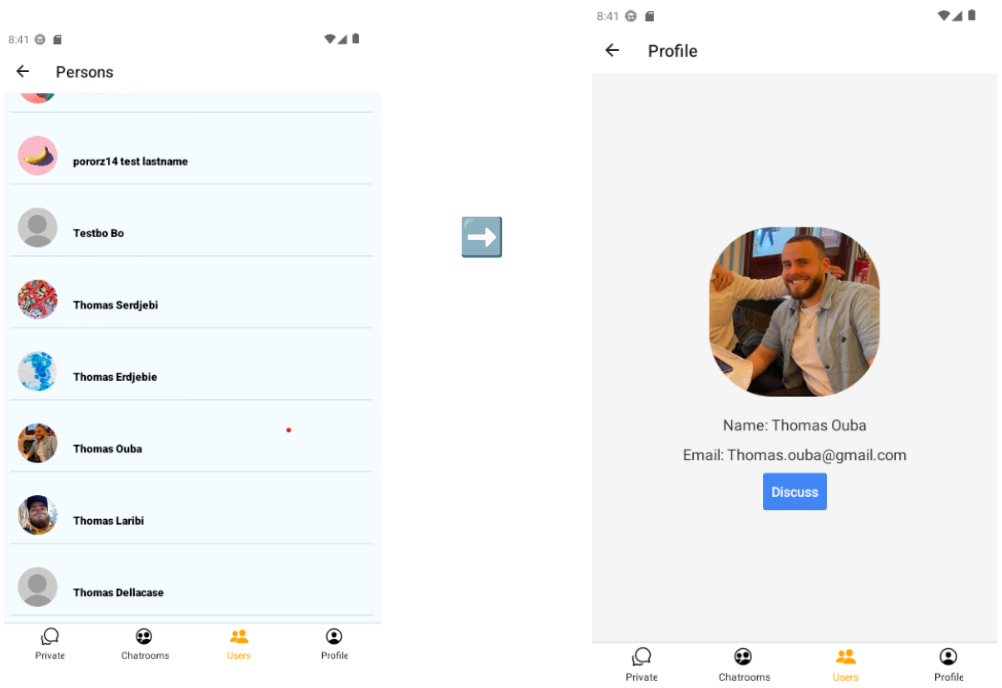
8. De la liste des utilisateurs vers le chat

Lorsqu'on clique sur la liste des utilisateurs dans la barre de navigation, le stack navigator suivant est appelé :

```
//Stack for the user's profile navigation => profile, update profile, update
function UsersStackScreen() {
  return(
    <UsersStack.Navigator>
      <UsersStack.Screen name="Persons" component={UsersListScreen} options={{}} />
      <UsersStack.Screen name="Profile" component={ProfileScreen} options={{ }} />
      <PrivateStack.Screen name="Private Chat" component={PrivateChatScreen} options={{}} />
    </UsersStack.Navigator>
  )
}
```

Extrait fichier App.js - Navigateur Empilé UsersStackScreen (icône Users dans la barre de navigation)

Si on clique sur un utilisateur, voici la transition vers l'écran Profile qui s'opère :

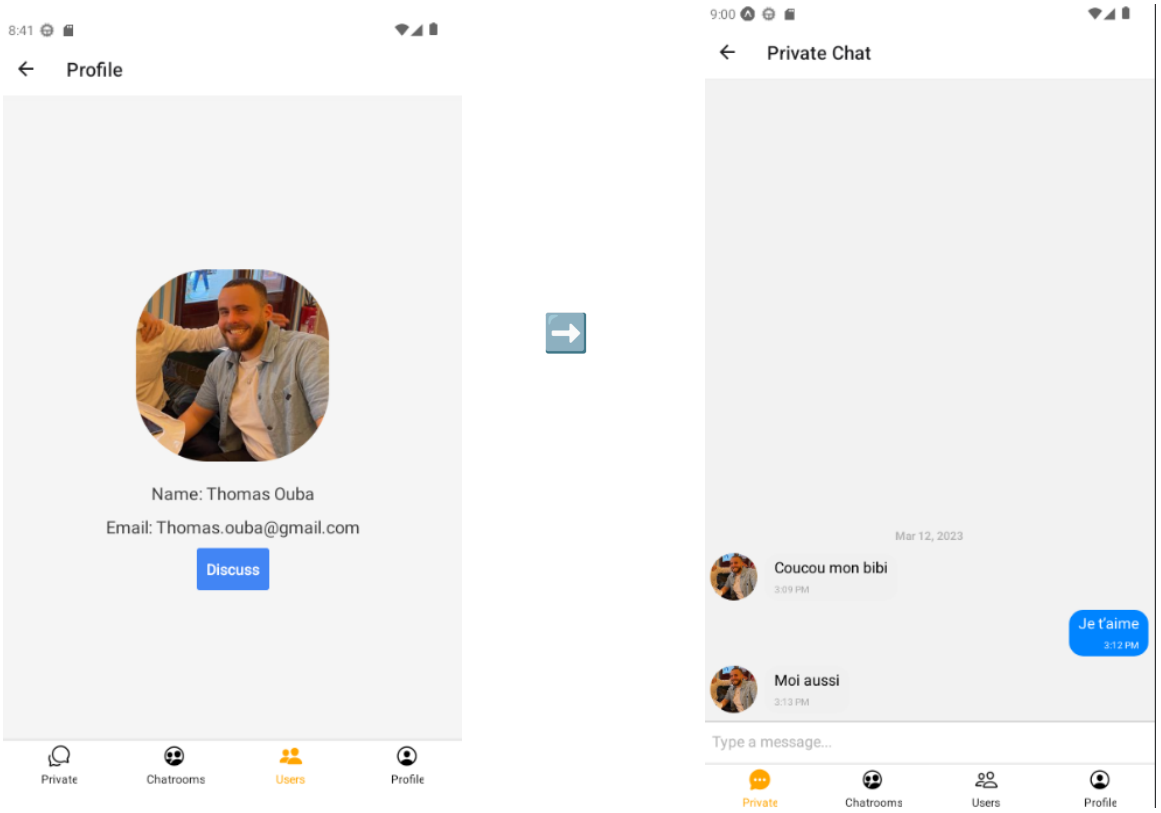


On constate l'existence du **bouton Discuss** structuré comme suit dans le **ProfileScreen** :

```
{isCurrentUser === false ? (
    <Button
        title="Discuss"
        onPress={() =>
            props.navigation.navigate('Private', {
                screen: 'Private Chat',
                params: {
                  id_user: user.id_user,
                  token: token
                }
            })
        }
        buttonStyle={styles.button}
    />
) : (
```

Extrait fichier ProfileScreen.js - Bouton de navigation vers l'écran de discussion privée

Ce bouton nous mène donc au chat privé avec l'utilisateur Thomas Ouba, généré par le composant **PrivateChatScreen** dont la navigation envoie en **paramètres l'id de l'utilisateur** avec qui nous allons discuter ainsi que le **token de l'utilisateur actuellement connecté**.



g. Le chat : fonctionnement et websockets

Le chat est une des principales fonctionnalités de notre projet. En effet, on distingue les **chats dans les salons de discussion** et les **chats privés**, qui sont tous deux **instantanés**, sans avoir à rafraîchir la page. Dans notre cas nous allons expliquer **comment s'articule la fonctionnalité du chat entre l'application client et le serveur**.

1. La mise en place des websockets

Les sockets sont un **mécanisme de communication** en **temps réel** entre un **client** et un **serveur**, permettant de transférer les données de manière **asynchrone** et **instantanée**. Ils doivent donc être mis en place à la fois chez le client et dans le serveur.

- Dans notre **API** les sockets sont mis en place de la manière suivante :

```
const app = express();
const server = http.createServer(app);
const socketIO = require('socket.io');
const io = socketIO(server);
```

Extrait fichier app.js de l'API - mise en place des websockets

Après avoir créé l'**instance d'express dans la constante app** et créé le **serveur http** avec la méthode **createServer** contenu dans la **constante server**, on importe le **module Socket.IO** de Javascript qui facilite la **gestion des sockets en temps réel**. On initialise enfin une **instance** de **socketIO** dans la **constante io** qui prend en paramètre le **serveur** créé précédemment.

- Dans notre **application**, dans les écrans qui nécessitent l'utilisation de sockets, leur mise en place de la manière suivante :

```
import io from 'socket.io-client';
const socket = io.connect('http://192.168.0.14:3000')
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - mise en place des websockets

On **importe la bibliothèque io de SocketIO** et on utilise la **méthode connect** dans la constante socket qui prend en paramètre l'**adresse du serveur et son port**.

2. L'envoi des messages et l'émission de sockets

Le **PrivateChatScreen** contient la **fonction sendMessagesInDb** qui permet d'envoyer des messages. Cette fonction vient appeler **successivement deux routes** :

- la **première route de l'API** appelée **(api/message)** vient **enregistrer le message** en base de données avec le texte, la date de création, et l'id du user qui envoie le message dans la **table Messages** dont on peut voir la **requête de l'API** associée ci-après :

```
//Sending messages in database
const sendMessagesInDb =  async(text) => {
  fetch('http://192.168.0.14:3000/api/message', {
    method:'POST',
    headers: { 'Authorization': 'Bearer ' + token,
               'Content-Type': 'application/json'
             },
    body: JSON.stringify({
      text: text,
    })
  })
  .then(data => data.json())
  .then(data =>  {
      if(data.error) {
        Alert.alert(data.error)
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 1)
Appel de la première route localhost:3000/api/message/

```
create: (data, callBack)=>{
    const now = new Date(Date.now());
    const created = now.toISOString().slice(0,19).replace('T', ' ');
    pool.query(
        'insert into messages (text, date, id_user, channel_id)  values (?, ?, ?, ?)',
        [
            data.text,
            created,
            data.id,
            data.channel_id
        ],
        (error, result, fields)=>{
            if(error){
                return callBack(error);
            }
            return callBack(null, result)
        }
    );
},
```

Extrait fichier messages.services.js - fonction create permettant l'insertion d'un message dans la table messages de la base de données - appelée par la route localhost:3000/api/message/

- la **seconde route appelée**, et seulement en cas de **succès de la première requête**, va enregistrer dans la **table de liaisons users_messages** les informations telles que l'id du message qui vient d'être créé, l'id du user qui envoie le message et l'id du user à qui il est destiné avec la **requête de l'API** associée ci-après :

```
85            } else if (data.succes == 1) {
86              fetch('http://192.168.0.14:3000/api/message/private', {
87                method:'POST',
88                headers: { 'Authorization': 'Bearer ' + token,
89                           'Content-Type': 'application/json'
90                         },
91                body: JSON.stringify({
92                  id_user_from: userId,
93                  id_user_to: towardUserId,
94                  id_message: data.data.insertId,
95                })
96              })
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 2)
Appel de la seconde route localhost:3000/api/message/private en cas de succès de la 1ère requête

```
    createPrivateMessage: (body, callBack)=>{
        pool.query(
            'insert into users_messages (id_user_from, id_user_to, id_message)  values (?, ?, ?)',
            [
                body.id_user_from,
                body.id_user_to,
                body.id_message
            ],
            (error, result, fields)=>{
                if(error){
                    return callBack(error);
                }
                return callBack(null, result)
            }
        );
    },
```

Extrait fichier messages.services.js - fonction createPrivateMessage permettant l'insertion des données d'un message dans la table de liaison users_messages appelée par la route localhost:3000/api/message/private

Si cette dernière **requête renvoie un succès**, c'est à ce moment là qu'intervient l'**envoie de socket** :

```
  })
})
.then(data => data.json())
.then(data =>  {
    if(data.error) {
      Alert.alert(data.error)
    } else if (data.succes == 1) {
      socket.emit('private message', {
      text: text,
      name: userName,
      id: '${socket.id}${Math.random()}',
        socketID: socket.id
      })
    }
})
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 3)
Suite au succès de la second requête, émission de l'évènement socket 'private message'

En effet, grâce à la **méthode socket.emit** le socket émet l'**événement 'private message'** qui est envoyé au serveur avec plusieurs informations telles que le contenu du texte, le nom de l'utilisateur et l'id du socket. Comment le serveur réceptionne-t-il alors l'émission du socket private message ?

3. La fonction de réception des messages et la réception de sockets

Dans le **fichier app.js de notre API** on crée le **gestionnaire d'événement** qui va **écouter les connexion au serveur Socket.IO**

```
io.on('connection', (socket) => {
  console.log(`Connecté au client ${socket.id}`);

  socket.on('message', (data) => {
    console.log('message')
    io.emit('messageResponse', data);
  });
  socket.on('private message', (data) => {
    console.log('private message')
    io.emit('privateMessageResponse', data);
  });

  socket.on('refresh chatlist', (data) => {
    io.emit('refresh chatlist', data);
  });
});
```

En effet, grâce à la **méthode socket.on** on peut définir quels **évènements** doivent être **écoutés par le serveur**. La ligne *socket.ion('private message')* va donc écouter cet évènement. A la suite de l'écoute de l'événement, le serveur va donc lui-même **émettre** au client l'événement **privateMessageResponse** qui contient les data qui lui ont été envoyées au préalable.

Extrait fichier App.js de l'API - gestionnaire d'évènement socket.IO

```
apiUrl = apiMessage + towardUserId

fetch(apiUrl, {
  method: 'GET',
  headers:{ Authorization: 'Bearer ' + token },
})
.then(response => response.json())
.then(response => {
  let array = response.data
  setMessages(
    array.map(data =>
      ({
      _id : data.id_message,
      createdAt: data.date,
      text: data.text,
      user: {
        _id: data.id_user_from,
        name: data.firstname + ' ' + data.lastname,
        avatar: data.avatar_from ? data.avatar_from.replace("localhost", "192.168.0.14") : unknownAvatar
      },
    })
    )
  )
}).catch(function(error) {
  console.log('There has been a problem with your fetch operation: ' + error.message);
});
```

Dans notre **client**, la fonction **getMessagesFromDb** va faire appel à la route de l'API **récupérant les messages privés entre deux utilisateurs**. Après avoir récupéré les informations désirées, elle va les **re-traiter** afin de pouvoir **structurer les données selon notre souhait.**

Extrait fichier PrivateChatScreen.js de l'application - fonction getMessageFromDb

```
// Call fetching previous messages with layoutEffect
useLayoutEffect(() => {
  socket.on('privateMessageResponse', (data) =>
  getMessagesFromDb(),
  setMessages([...messages, data]),

  );
}, [socket, messages]);
```

Extrait fichier PrivateChatScreen.js de l'application - useLayoutEffect

Dans le **useLayoutEffect** ci-dessus, un hook similaire au useEffect mais exécuté de manière **synchrone** après les **mutations du DOM**, on utilise la **méthode socket.on** qui écoute **l'événement privateMessageResponse** envoyé par le **serveur**. A ce moment-là, on lance la **fonction getMessagesFromDb** et on utilise le **hook useState** définit plus haut sur la **constante messages** afin d'affecter à la constante messages les informations récupérées en base de données.

```
const [messages, setMessages] = useState([]);
```

Extrait fichier PrivateChatScreen.js de l'application - useState sur la variable messages

Pour **résumer le fonctionnement des websockets** dans le chat, il faut les **mettre en place** à la fois dans l'**API côté serveur** ainsi que dans l'**application côté client**. **Côté serveur**, il faut penser à **paramétrer les évènements qui vont être écoutés**. Ensuite **côté client**, lorsque la fonction d'**insertion des messages** envoyés en base de données renvoie un **succès**, on envoie au serveur un **événement** qui sera **écouté**, et va **émettre** une **réponse** au **client** stipulant qu'il a bien reçu l'information. A ce moment-là, on exécute la **fonction de récupération des messages** entre les deux utilisateurs en base de données pour les afficher dans le GiftedChat.

4. Le GiftedChat

GiftedChat est une **bibliothèque React Native** facilitant l'**implémentation de fonctionnalités de chat dans les applications**. Elle fournit des composants préconstruits et personnalisables pour la création d'interfaces utilisateur de chat. Le composant fourni GiftedChat est au cœur de cette bibliothèque.

```
return (
    <GiftedChat
        messages={messages}
        showAvatarForEveryMessage={true}
        onSend={messages => onSend(messages)}
        user={{
          _id: userId,
          name: userName,
          avatar: userAvatar,
        }}
        renderAvatar={props => {
          const id  = props.currentMessage.user._id;
          return (
            <TouchableOpacity onPress={() => avatarPressed(id)}>
            <Image
              style={styles.avatar}
              source={{ uri: props.currentMessage.user.avatar }}
            />
          </TouchableOpacity>
          )
        }}
    />
);
```

Extrait fichier PrivateChatScreen.js de l'application - rendu du composant PrivateChatScreen

Le **composant GiftedChat** prend en paramètres :

- les **messages** initialisées dans la variable grâce au **useState** et au **setMessages**

```
<GiftedChat
    messages={messages}
```

- le **onSend** qui va appeler la fonction d'**insertion des messages** en base de données lorsque l'utilisateur va envoyer un message

```
  onSend={messages => onSend(messages)}
```

```
//Call sending messages in db and then messages from db to reload the discussion
const onSend = useCallback((messages = []) => {
  const { _id, createdAt, text, user,} = messages[0]
  sendMessagesInDb(messages[0].text)
}, []);
```

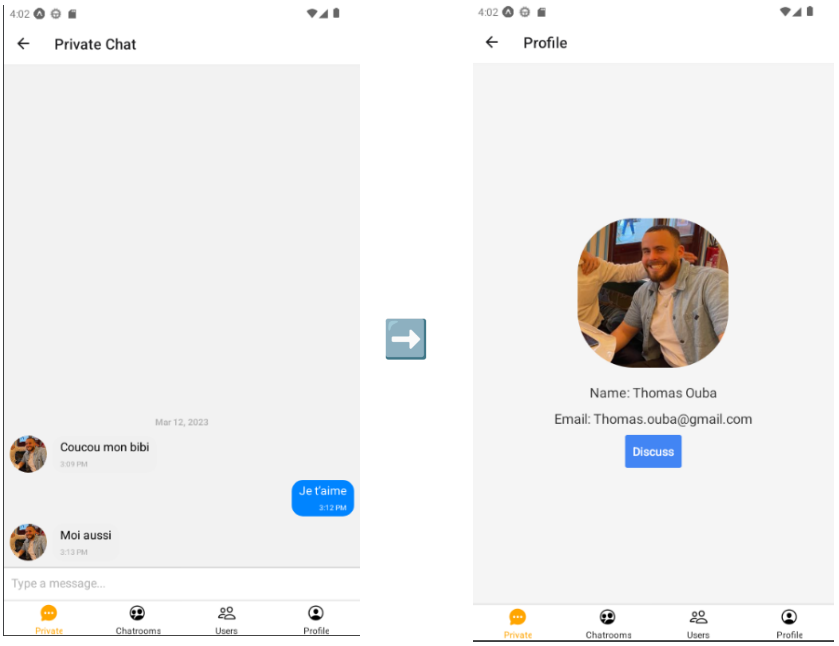
- les **informations** associées à l'utilisateur qui envoie le message dans le paramètre user

```
user={{
  _id: userId,
  name: userName,
  avatar: userAvatar,
}}
```

- la **méthode renderAvatar** permet de rendre l'**avatar cliquable**, de lui **appliquer du style**. Le fait de rendre l'avatar cliquable permet à l'utilisateur d'aller sur le profil de la personne avec qui il est en discussion en appelant la fonction **avatarPressed** qui prend en **paramètre l'id** de l'**utilisateur visé** via les **props de GiftedChat**.

```
renderAvatar={props => {
  const id  = props.currentMessage.user._id;
  return (
    <TouchableOpacity onPress={() => avatarPressed(id)}>
    <Image
      style={styles.avatar}
      source={{ uri: props.currentMessage.user.avatar }}
    />
  </TouchableOpacity>
  )
}}
```

```
const avatarPressed = (id) => {
  props.navigation.navigate('Profile', {
    id_user: id
  });
}
```



D. Le panel administrateur

a. Structure générale

La structure du panel administrateur est simple. En effet, grâce à **React Admin** qui facilite la création d'**interface administrateur**, cette **couche du projet** se compose de peu de choses :

- un **fichier App.js** important les différentes **librairies utilisées**, **composants utilisés** et qui vient configurer la **connexion** à notre **API** à l'aide d'un **data provider**, ainsi que l'affichage du tableau administrateur
- un fichier **authProvider** permettant la **connexion au panel administrateur** et de vérifier le rôle Administrateur de l'utilisateur
- un fichier **index.js** qui permettra l'**affichage du composant principal App dans le navigateur**
- des **composants personnalisés** permettant d'afficher, créer, modifier ou supprimer des éléments de la base de données

b. Structure du fichier App.js

Le fichier App.js implémente React Admin. Sa structure est assez simple.

1. Import des bibliothèques

En premier lieu, on effectue l'**import des bibliothèques** qui nous seront utiles :

```
import React from 'react';
import { Admin, Resource, fetchUtils } from 'react-admin';
import simpleRestProvider from 'ra-data-simple-rest';
```

Extrait fichier App.js de l'interface administrateur - import des bibliothèques

2. Import des composants

Dans un second temps, nous importons les différents **composants personnalisés** qui permettront d'**afficher les listes dans les tableaux de bords et/ou d'éditer les informations**, ainsi que l'authProvider qui va permettre de gérer l'authentification de l'utilisateur :

```
import ChanList from './components/ChanList';
import MessageList from './components/MessageList';
import UserList from './components/UserList';
import ChanEdit from './components/ChanEdit';
import MessageEdit from './components/MessageEdit';
import UserEdit from './components/UserEdit';
import { authProvider } from './authProvider';
```

Extrait fichier App.js de l'interface administrateur - import des composants personnalisés

3. Gestion des requêtes HTTP

```
const httpClient = (url, options = {}) => {
  if (!options.headers) {
      options.headers = new Headers({ Accept: 'application/json' });
  }
  const token = localStorage.getItem('token');
  options.headers.set('Authorization', `Bearer ${token}`);
  return fetchUtils.fetchJson(url, options);
};
```

Extrait fichier App.js de l'interface administrateur - const httpClient

La **fonction httpClient** prend **deux paramètres** : **url** (l'URL de la requête HTTP) et **options** qui sont définies comme un objet vide. Si les headers ne sont pas créés, alors elle créé un nouvel objet **Headers** spécifiant que l'on souhaite recevoir des réponses au **format JSON**. La fonction récupère le jeton d'authentification (token) à partir du stockage local (localStorage) en utilisant la clé 'token' et configure l'ente^te authorization avec ce token. Enfin, la fonction utilise la **méthode fetchJson** de **fetchUtils** (une dépendance de react-admin) pour **effectuer la requête HTTP avec l'URL** et les **options** fournies. Elle renvoie le **résultat de la requête**, qui est une **promesse** retournant les **données** au **format JSON**. Elle est utilisée en tant que **fournisseur de données** (dataProvider) dans le composant Admin de l'application React Admin pour gérer les requêtes HTTP vers l'API REST.

4. Rendu du composant

Le rendu crée une interface d'administration avec une connexion à l'API, un fournisseur d'authentification personnalisé, et affiche trois ressources (channels, messages, et users) avec des composants spécifiques pour la liste et l'édition des données de chaque ressource.

```
function App() {
  return (
    <Admin dataProvider={simpleRestProvider('http://localhost:3000/api/bo', httpClient)} authProvider={authProvider}>
      <Resource name='channels' list={ChanList} create='' edit={ChanEdit} delete='' />
      <Resource name='messages' list={MessageList} create='' edit={MessageEdit} delete='' />
      <Resource name='users' list={UserList} create='' edit={UserEdit} delete='' />
    </Admin>
  )
}

export default App
```

Extrait fichier App.js de l'interface administrateur - rendu fonction App

c. Structure des composants

1. Import des modules et des composants

```
import React from 'react';
// import AsyncStorage from '@react-native-async-storage/async-storage';
import { List,
         Datagrid,
         TextField,
         EditButton,
         DeleteButton,
         useGetList
         } from 'react-admin';
```

Extrait fichier UserList.js de l'interface administrateur - import des composants et modules

Le code importe des modules et des composants nécessaires. Il importe notamment les composants List, Datagrid, TextField, EditButton, DeleteButton et useGetList de la bibliothèque react-admin.

2. La fonction composant

Cette **fonction composant** qui prend des "**props**" en **paramètre**. Tous les composants en comportent une permettant d'afficher et d'utiliser les fonctionnalités souhaitées. Ce composant représente la **liste des utilisateurs dans l'interface d'administration**. Elle fait appel au hook **userGetList** de react-admin pour récupérer les **données des utilisateurs** en appelant l'API via deux paramètres : le nom de la ressources qui est 'users" et les options comme la pagination et le tri.

```
const UserList = (props) => {
  const { data, isLoading } = useGetList('users', {
    pagination: { page: 1, perPage: 10 },
    sort: { field: 'id', order: 'asc'},
  });
  return (
    <List {...props}>
      <Datagrid>
        <TextField source='id_user' />
        <TextField source='firstname' />
        <TextField source='lastname' />
        <TextField source='email' />
        <EditButton />
        <DeleteButton />
      </Datagrid>
    </List>
  )
}
```

List et Datagrid sont des composants de react-admin permettant d'afficher les données des utilisateurs sous forme de liste avec des colonnes. Le composant List reçoit les props props du composant parent et le composant Datagrid est utilisé comme un enfant de List pour afficher les données dans un tableau. Les composants TextField affichent les valeurs des propriétés utilisateurs. Les deux boutons Edit et Delete sont également des composants de React Admin.

Extrait fichier UserList.js de l'interface administrateur - fonction composant UserList

Enfin comme pour tous les composants React, ce dernier est exporté afin d'être utilisé dans le fichier App.js.

```
export default UserList
```

VI. Jeux d'essai

Laissez-moi vous présenter plusieurs jeux d'essai successifs sur les requêtes développées dans l'API. Dans la continuité de notre exemple de développement de la fonctionnalité de conversation privées, je vais vous montrer les jeux d'essai des **fonctions create**, **createPrivateMessage** et **getPrivateMessages** de l'API appelées dans la **fonction sendMessagesInDb** et **getMessagesFromDb()** du **composant PrivateChatScreen** citées dans la partie développant le f**onctionnement du chat et des websockets**.

Pour rappel, la **fonction create de l'API** est appelée dans la **fonction sendMessagesInDb** après que l'utilisateur ait écrit et appuyé sur le **bouton envoyé** dans l'écran de conversation privée **PrivateChatScreen** :

```
//Call sending messages in db and then messages from db to reload the discussion
const onSend = useCallback((messages = []) => {
  const { _id, createdAt, text, user,} = messages[0]
  sendMessagesInDb(messages[0].text)
}, []);
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction onSend, appelée lorsque l'utilisateur envoie un message, prenant en paramètre le message envoyé dans GiftedChat, appelant la fonction suivante sendMessagesInDb

```
//Sending messages in database
const sendMessagesInDb =  async(text) => {
  fetch('http://192.168.0.14:3000/api/message', {
    method:'POST',
    headers: { 'Authorization': 'Bearer ' + token,
               'Content-Type': 'application/json'
             },
    body: JSON.stringify({
      text: text,
    })
  })
  .then(data => data.json())
  .then(data =>  {
      if(data.error) {
        Alert.alert(data.error)
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 1)
Appel de la première route localhost:3000/api/message/

Les données sont envoyées avec la **méthode POST**. En **headers**, on spécifie le **token** de l'utilisateur récupéré plus haut dans le composant via les **paramètres de routes** envoyées en navigation vers cet écran :

```
const token = props.route.params.token
```

Le **corps du message** est contenu dans **la variable text** spécifiée en **paramètre** de la fonction **sendMessagesInDb** et récupérée à l'aide de **GiftedChat**. Avant d'arriver à l'exécution de la requête, voici les étapes suivies :

- la **route** appelée dans le fichier app.js de l'API :

```
app.use("/api/message", messageRouter);
```

- dans le même fichier app.js, le **composant messageRouter** est importé :

```
const messageRouter = require("./api/message/message.router");
```

- dans le fichier **message.router**, voici la **route "POST"** qui est appelée :

```
//Rout to send a message
router.post("/", checkToken, createMessage);
```

- dans le **middleware checkToken**, le **token** de l'utilisateur est **vérifié** et les **informations de l'utilisateur** sont **stockées** dans le **corps de requête req.user** avant de passer à la fonction **createMessage** du fichier **message.controller.js** :

```
            req.user = decodedToken
            next();
```

```
module.exports = {
    createMessage: (req, res) =>{
        const body = req.body;
        if(!body.text){
            return res.status(500).json({
                succes: 0,
                error: "Your message cannot be empty."
            })


        }

        body.id =  req.user.result.id_user

        create(body, (err, results)=>{

            if(err){
                console.log(err)
                return res.status(500).json({
                    succes: 0,
                    error: "An error occured"
                })
            };

            return res.status(200).json({
                succes:1,
                data: results
            })
        })
    },
```

On vérifie donc d'abord que le corps de requête req.body.text n'est pas vide. Ensuite on affecte l'id de l'utilisateur créant le message au body.id depuis le corps de requête req.user généré précédemment. Ensuite on lance donc la fonction create du fichier messages.services.js importé message.controller.js.

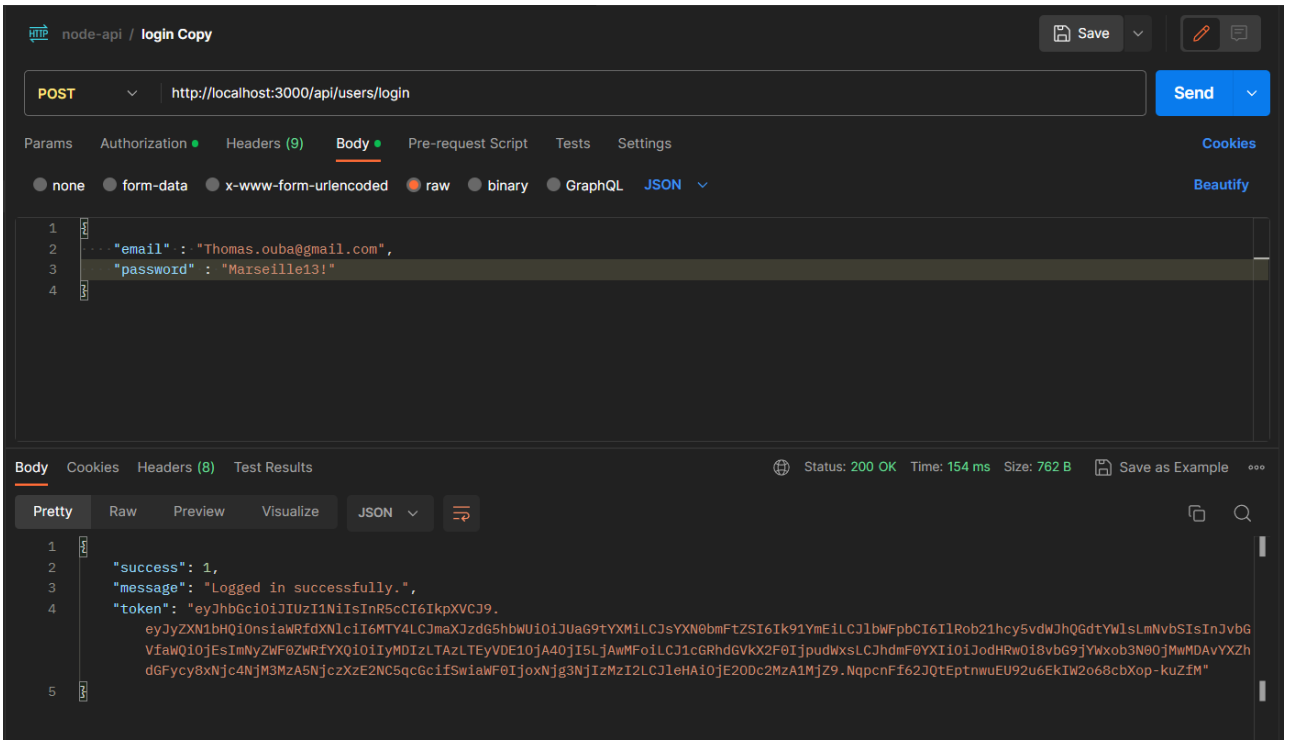
```
create: (data, callBack)=>{
    const now = new Date(Date.now());
    const created = now.toISOString().slice(0,19).replace('T', ' ');
    pool.query(
        'insert into messages (text, date, id_user, channel_id)  values (?, ?, ?, ?)',
        [
            data.text,
            created,
            data.id,
            data.channel_id
        ],
        (error, result, fields)=>{
            if(error){
                return callBack(error);
            }
            return callBack(null, result)
        }
    );
},
```

Extrait fichier messages.services.js - fonction create permettant l'insertion d'un message dans la table messages de la base de données - appelée par la route localhost:3000/api/message/

La **variable data** correspond donc à la **variable body du corps de requête**. On va donc **insérer** dans la **table messages** de la base de données le **texte**, la **date** généré dans la constante created, l'**id** de l'utilisateur et potentiellement l'**id du channel** où est envoyé le message, cependant dans notre cas, puisqu'il s'agit d'une messagerie privée, ce champ de la table restera vide.

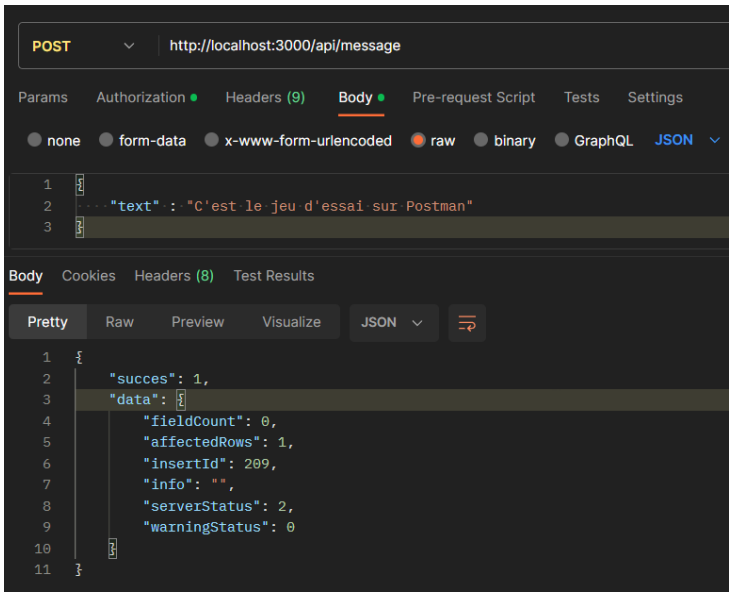
Voici un test effectué dans l'**outil POSTMAN** en plusieurs étapes :

- connexion de l'utilisateur : on envoie une combinaison email/password qui sont corrects, on s'attend donc à recevoir un success à 1 ainsi qu'un token qui sera généré.

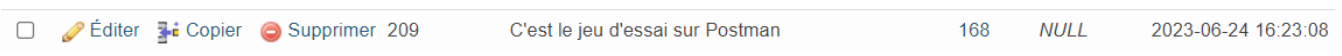


Success à 1 est retourné, on a bien le message stipulant que la connexion s'est déroulée avec **succès** et le **token est généré**. On l'enregistre dans l'**onglet Authorization** en spécifiant que le type est bien un **Bearer Token**.

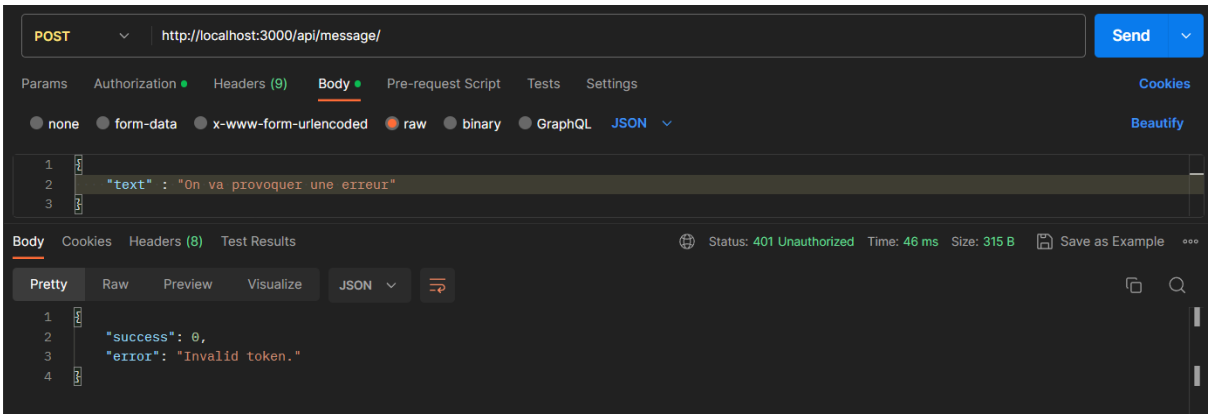
- Appelons maintenant la **route de création d'un message dans POSTMAN** :



Étant donné que l'**id de l'utilisateur** et **la date de création** sont affectées à des **variables** avant l'**appel de la fonction create**, et ce grâce au middleware checkToken qui décode le token et affecte les informations utilisateurs à des variables, il est seulement nécessaire de **saisir le contenu de la variable text** dans le **corps de requête**. On constate bien que l'**envoi des données avec l'id de l'utilisateur s'est effectué avec succès**. L'id de l'utilisateur avec lequel j'étais connecté est le 168. Dans la réponse de la requête on constate que l'**id du message inséré** en base de données est le **209**. Constatez ci-dessous **l'enregistrement dans la table messages** de notre base de données :



Testons la même route sans que l'utilisateur ne se connecte et donc sans enregistrer le token dans le header Authorization : on devrait obtenir maintenant un success à 0 et une erreur "Invalid Token".



Le success est retourné à 0 est l'erreur retournée est "Invalid Token" comme spécifié dans le middleware checkToken :

```
verify(token, process.env.TK_KEY,(err, decodedToken) => {
    if(err) {
        res.statusCode = 401;
        res.json({
            success:0,
            error: "Invalid token."
        }) ;
```

Dans la **seconde partie de notre fonction sendMessagesInDb** , si la première requête ci-dessus renvoie un **succès**, alors on va appeler la **route de l'API** permettant d'enregistrer ce message dans **la table de liaison users_messages** :

```
85          } else if (data.succes == 1) {
86            fetch('http://192.168.0.14:3000/api/message/private', {
87              method:'POST',
88              headers: { 'Authorization': 'Bearer ' + token,
89                         'Content-Type': 'application/json'
90                       },
91              body: JSON.stringify({
92                id_user_from: userId,
93                id_user_to: towardUserId,
94                id_message: data.data.insertId,
95              })
96            })
```

Extrait fichier PrivateChatScreen.js de l'application ChatApp - fonction sendMessagesInDb (partie 2)
Appel de la seconde route localhost:3000/api/message/private en cas de succès de la 1ère requête

On envoie donc toujours une **requête POST** qui prend en **headers** le **token** de l'utilisateur et en **corps de requête** :

- l'**id de l'utilisateur** qui **envoie le message (id_user_from)** affecté précédemment à la **variable userId** comme ceci :

```
const token = props.route.params.token
const decodedToken = jwt_decode(token)
const userId = decodedToken.result.id_user
```

- l'**id de l'utilisateur** qui **reçoit le message (id_user_to)** affecté à la variable **towardUserId**. En effet, l'**id du user** avec lequel on engage une conversation est **envoyé en paramètre de route** lorsqu'on clique sur le **bouton discuss** depuis l'**écran ProfileScreen** ou alors lorsqu'on clique dans l'**écran de la liste des conversations privées de l'utilisateur**

```
onPress={() => {
  props.navigation.navigate('Private Chat', {
    id_user: item.id_user_to == userId ? item.id_user_from : item.id_user_to,
    token: token
  });
}}
```

Extrait fichier PrivateChatListScreen.js - évènement onPress déclenché lorsqu'on clique sur une conversation privée avec un utilisateur dans l'écran, dirigeant l'utilisateur vers l'écran PrivateChatScreen avec l'id de l'utilisateur avec qui a lieu la conversation en paramètre ainsi que le token de l'utilisateur courant

```
<Button
    title="Discuss"
    onPress={() =>
        props.navigation.navigate('Private', {
            screen: 'Private Chat',
            params: {
              id_user: user.id_user,
              token: token
            }
        })
    }
    buttonStyle={styles.button}
/>
```

Extrait fichier ProfileScreen.js - Bouton Discuss déclenchant la navigation vers l'écran de conversation privée PrivateChatScreen avec l'id de l'utilisateur avec qui a lieu la conversation en paramètre ainsi que le token de l'utilisateur courant

```
const towardUserId = props.route.params.id_user
```

- On envoie également l'**id du message précédemment créé** et contenu dans la **réponse data** renvoyée par le **serveur (data.data.insertId)**

```
body: JSON.stringify({
  id_user_from: userId,
  id_user_to: towardUserId,
  id_message: data.data.insertId,
})
```

```
{
    "succes": 1,
    "data": {
        "fieldCount": 0,
        "affectedRows": 1,
        "insertId": 209,
        "info": "",
        "serverStatus": 2,
        "warningStatus": 0
    }
}
```

- dans le fichier **message.router**, voici la **route "POST"** qui est appelée :

```
//Rout to send a message
router.post("/", checkToken, createMessage);
```

Comme précédemment, les **informations du user** vont **être récupérées** dans le **token** grâce au **middleware checkToken** et **envoyées** dans l'**objet de requête req.user** avant l'**exécution** de la **fonction createMessage** du fichier **messages.controller.js** :

```
createPrivateMessage: (req, res) =>{
    const body = req.body;
    if(!body.id_user_from){
        return res.status(500).json({
            succes: 0,
            error: "Your message is not sent from an existing user."
        })

    }

    if(!body.id_user_to){
        return res.status(500).json({
            succes: 0,
            error: "Your message is not sent to an existing user."
        })
    }

    if(!body.id_message){
        return res.status(500).json({
            succes: 0,
            error: "No message to sent."
        })
    }

    createPrivateMessage(body, (err, results)=>{

        if(err){
            console.log(err)
            return res.status(500).json({
                succes: 0,
                error: "An error occured"
            })
        };
        return res.status(200).json({
            succes:1,
            data: results
        })
    })
},
```

On vérifie à nouveau que le **corps de requête** contient bien toutes les **informations requises** :

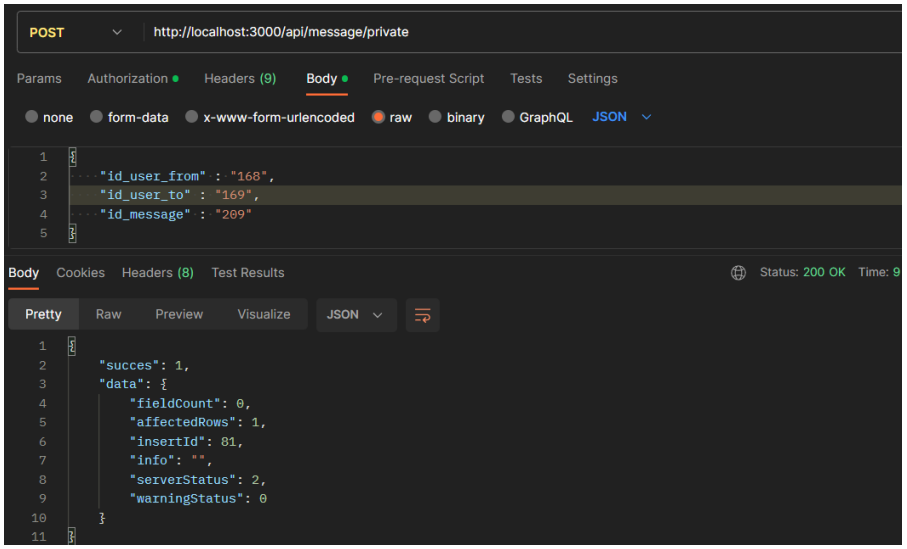
- l'id_user_to
- l'id_user_from
- l'id du message

Ensuite on appelle donc la requête contenue dans la fonction createPrivateMessage du fichier message.services.js ci-après.

```
createPrivateMessage: (body, callBack)=>{
    pool.query(
        'insert into users_messages (id_user_from, id_user_to, id_message)  values (?, ?, ?)',
        [
            body.id_user_from,
            body.id_user_to,
            body.id_message
        ],
        (error, result, fields)=>{
            if(error){
                return callBack(error);
            }
            return callBack(null, result)
        }
    );
},
```

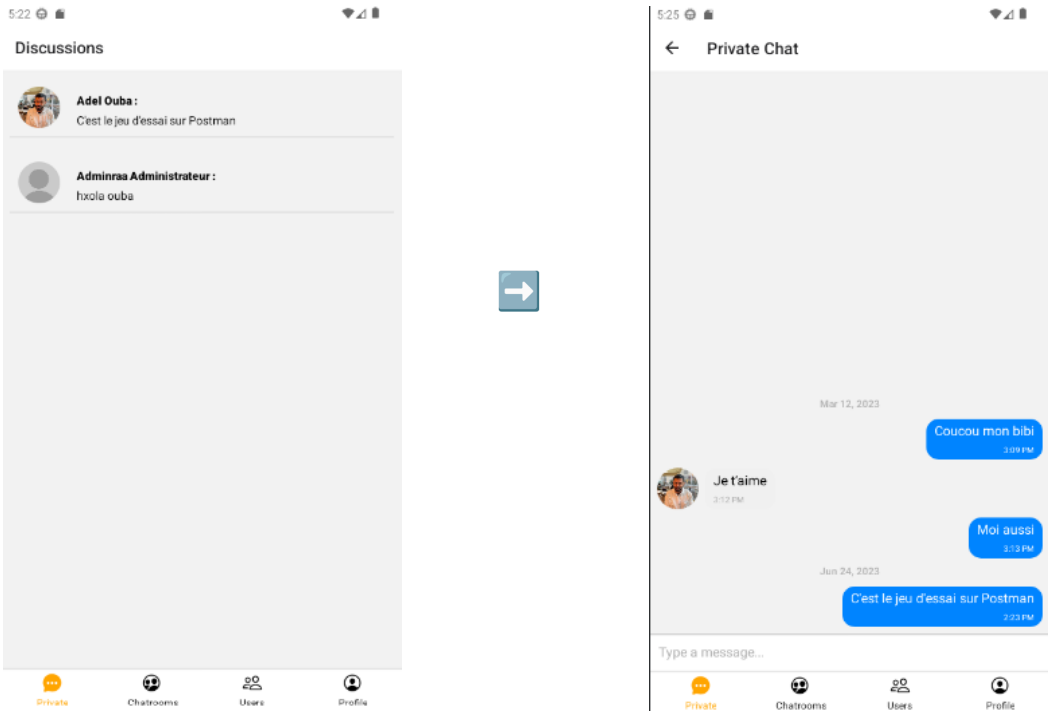
Extrait fichier messages.services.js - fonction createPrivateMessage permettant l'insertion des données d'un message dans la table de liaison users_messages appelée par la route localhost:3000/api/message/private

Réalisons donc le test dans **POSTMAN** sachant que l'id du message précédemment créé est le 209. Imaginons que l'on envoie le message à l'utilisateur ayant l'id 169.

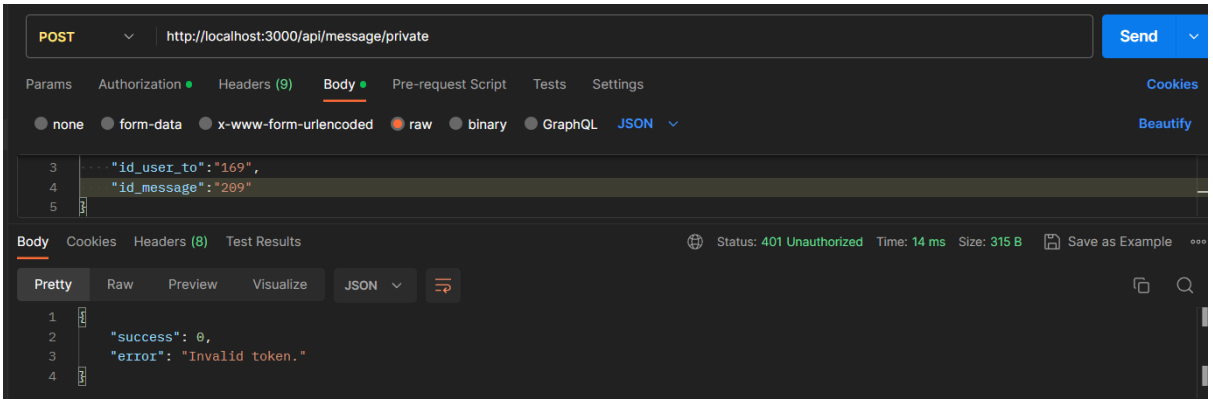


La requête renvoie un succès et spécifie que l'id créé dans la table **users_messages** est le 81. Voici donc l'enregistrement dans la table **users_messages** de la base de données :

Enfin, voici dans le résultat dans l'application mobile depuis le **compte de l'utilisateur 168** :

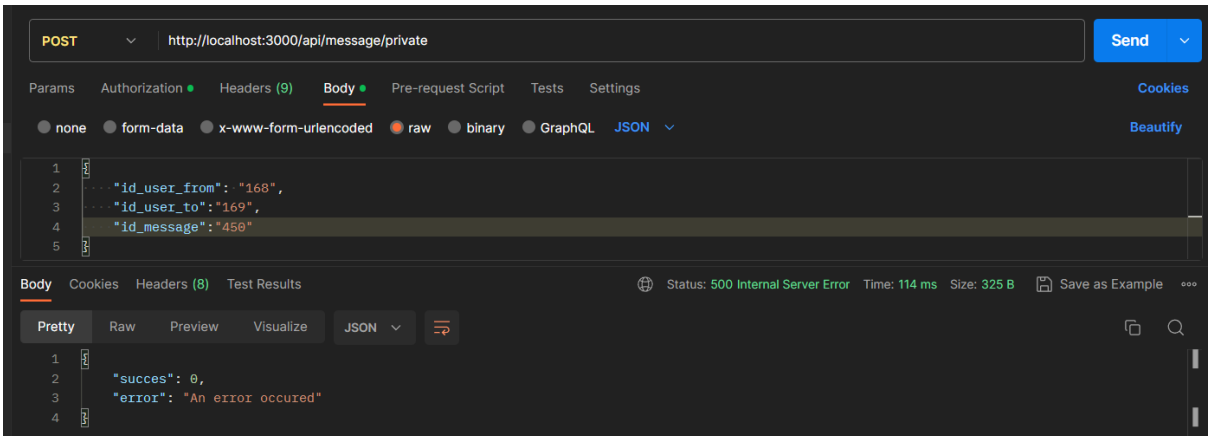


Testons à nouveau cette requête sans que l'utilisateur ne se soit connecté et sans token :



Le success est retourné à 0 est l'erreur retournée est "Invalid Token" comme spécifié dans le middleware checkToken.

Testons encore cette route après que l'utilisateur se soit connecté et que le token soit généré et enregistré, mais en utilisant un id de message inexistant :



Le success est retourné à 0 est l'erreur retournée est "Invalid Token" comme spécifié dans le middleware checkToken. Il en serait de même en saisissant des id users incorrects.

Pour terminer sur les jeux d'essai, faisons un bref test de la **fonction** qui vient **récupérer les messages privés** entre deux utilisateurs, appelée par la **route localhost:3000/api/message/private/:id**

```
getPrivateMessages:(id_user_1, id_user_2, callBack)  => {
    pool.query(
        `SELECT users_from.firstname AS firstname_from,
        users_from.lastname AS lastname_from,
        users_to.firstname AS firstname_to,
        users_to.lastname AS lastname_to,
        users_to.avatar AS avatar_to,
        users_from.avatar AS avatar_from,
        users_messages.*,
        messages.*
        FROM users_messages
        JOIN users AS users_from ON users_from.id_user = users_messages.id_user_from
        JOIN users AS users_to ON users_to.id_user = users_messages.id_user_to
        JOIN messages ON messages.id_message = users_messages.id_message
        WHERE ( id_user_from = ? and id_user_to = ? )
        OR  ( id_user_to = ? and id_user_from = ? )
        ORDER BY messages.id_message DESC`,
        [id_user_1, id_user_2, id_user_1, id_user_2],
        (error, results) => {
            if (error) {
                return callBack(error)
            }
            return callBack(null, results)
        }
    )
```

Extrait fichier message.service.js de l'API - fonction getPrivateMessages permettant de récupérer les messages privés entre deux utilisateurs

On envoie en **paramètre de route l'id de l'utilisateur** avec qui on discute, l'**id de l'utilisateur courant** est récupéré via le **middleware checkToken** comme dans les exemples précédents.

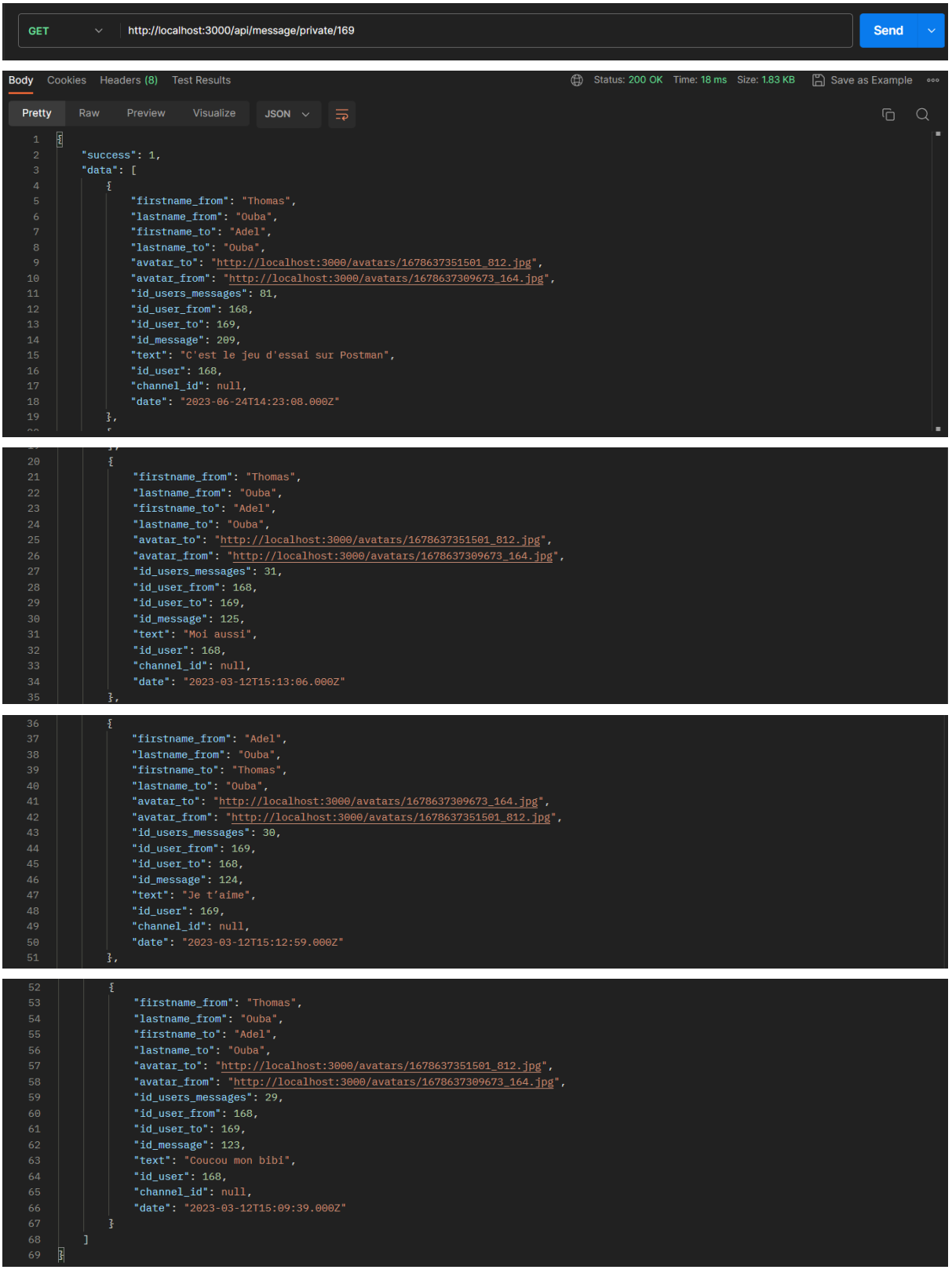
```
const apiMessage = 'http://192.168.0.14:3000/api/message/private/'

const towardUserId = props.route.params.id_user

apiUrl = apiMessage + towardUserId
```

Extrait fichier PrivateChatScreen.js de l'application - constitution de la route appelée pour récupérer les messages dans la base de données

Dans la requête ci-dessus, on récupère les **messages** depuis la **table users_messages** en faisant des **jointures** avec les **tables users** et **messages**, où les **ids des deux utilisateurs** remplissent soit le champ **id_user_from** soit **id_user_to** grâce aux **conditions WHERE et OR** :

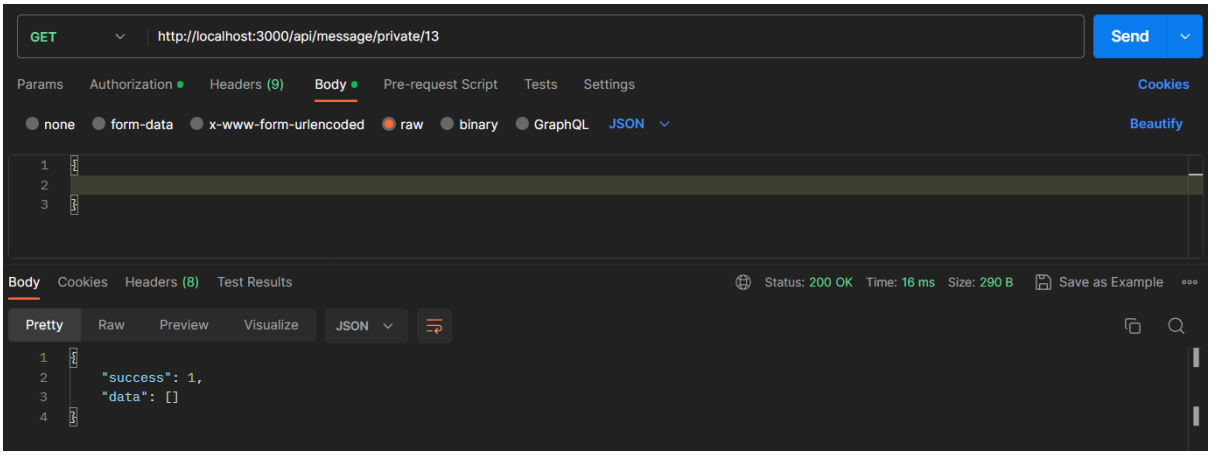
```
WHERE ( id_user_from = ? and id_user_to = ? )
OR  ( id_user_to = ? and id_user_from = ? )
ORDER BY messages.id_message DESC`,
[id_user_1, id_user_2, id_user_1, id_user_2],
```

Dans notre jeu d'essai précédent, la conversation privée entre l'**utilisateur 168** et **169** comporte **4 messages**, **3 de l'utilisateur 168 (Thomas Ouba)** et **1 message de l'utilisateur 169 (Adel Ouba)**. Lançons le test sur POSTMAN en s'étant au préalable **connecté avec l'utilisateur 168** et inscrit **son token en Authorization** :



En conclusion nous récupérons bien les 4 messages envoyés entre les deux utilisateurs.

Cependant, tentons de récupérer des messages entre deux utilisateurs qui ne s'en sont jamais envoyés comme entre les utilisateurs 168 (actuellement connecté) et 13 :



La requête est effectuée avec succès (success : 1) et renvoie un tableau vide comme prévu.

VII. Sécurité et veille anglophone

A. La sécurité dans une application en couches

Dans une **application développée en couche** comme notre projet ChatApp, la **sécurité** doit être gérée à la fois **côté client et côté serveur**. En effet, afin de lutter contre les **attaques** de types **injections SQL** et/ou **XSS Cross Site Scripting**, les contrôles effectués doivent être en place sur tous les plans. Ces attaques sont effectuées en entrant dans des **formulaires** des balises et requêtes SQL dans les formulaires afin de pouvoir atteindre de multiple façon le contenu des bases de données, ou bien en injectant du script Javascript.

a. Sécurisation de la connexion à la base de données côté serveur

Le fichier **.env** contient les données d'accès à la base de données. Par définition, ce fichier ne sera **pas publié sur le repository gitHub**, ce qui assure de pouvoir publier son projet tout en évitant que les accès de la base de données ne le soit également.

```
1  APP_PORT=3000
2  DB_PORT=3306
3  DB_HOST=127.0.0.1
4  DB_USER=root
5  DB_PASS=
6  MYSQL_DB=api-mobile-chat
7  TK_KEY=qwe1234
```

Extrait fichier .env de l'API - constantes de connexion à la base de données

b. Sécurisation du système de connexion utilisateur via un JsonWebToken

Comme vous pouvez le voir ci-dessus, il existe une **constante TK_KEY** dans le fichier .env. Cette **clé unique** est utilisée à la fois à la **génération du token**, à son **authentification** ainsi qu'à son **décodage**. On parle de système d'**authentification forte** lors de l'utilisation de JsonWebToken. Celles ci présente plusieurs avantages :

- une sécurité renforcée à l'aide de la clé du token
- stockage des informations de l'utilisateur dans le token
- authentification renforcée permettant de vérifier l'identité de l'utilisateur lors de chaque requête ultérieure
- le format JSON qui permet leur expansibilité puisque c'est un format largement utilisé et supporté
- possibilité d'inclure une date d'expiration

Ce token est utilisé dans plusieurs couches de notre projet :

- à la connexion de l'utilisateur le token est généré par l'API
- à la connexion de l'utilisateur, le token est stocké par l'application mobile
- le token de l'utilisateur est envoyé dans la plupart des routes de l'API en header et est vérifié par le middleware checkToken

c. Sécurisation des formulaires côté client

Les **données entrantes** saisies par l'utilisateur dans les formulaires sont à la fois vérifiées **côté client et côté serveur**. Prenons l'exemple de l'inscription d'un utilisateur :

- on utilise la **fonction sanitizeHtml** qui en supprimant ou en échappant les balises, attributs et scripts potentiellement dangereux du code HTML évite les attaques XSS et injections SQL

```
const sanitizedFirstname = sanitizeHtml(firstname)
const sanitizedLastname = sanitizeHtml(lastname)
const sanitizedEmail = sanitizeHtml(email)
const sanitizedPassword = sanitizeHtml(password)
const sanitizedPasswordValidation = sanitizeHtml(password_validation)
```

Extrait fichier RegisterScreen de l'application - constantes du formulaire d'inscription

- on vérifie la **cohérence de la saisie de l'utilisateur** dans le champ attendu à l'aide des **regexs** : les champs relatifs aux noms et prénoms ne doivent pas contenir de caractères spéciaux hors lettres accentuées, un email doit respecter un certain format :

```
const namesRegex = "^[a-zA-ZÀ-ÿ -]+$"

//Verifying firstname & lastname without numbers or special caracters except accents and "-"
if (!sanitizedFirstname.match(namesRegex) || !sanitizedLastname.match(namesRegex)){
  alert("Your firstname and lastname can only contain letters, accents, blank space and '-'.")
  return;
}

//Verifying email format
if (!sanitizedEmail.match(/^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$/)) {
  alert("Your email is not correct.")
  return;
}
```

Extrait fichier RegisterScreen de l'application - contrôle des champs email, firstname, lastname

d. Sécurisation des formulaires côté serveur

Côté serveur nous effectuons les **mêmes vérifications** de **cohérence** de saisie de l'utilisateur à l'aide de regex :

```
const namesRegex = "^[a-zA-ZÀ-ÿ -]+$"

//Verifying firstname & lastname without numbers or special caracters except accents and "-"
if (!body.firstname.match(namesRegex) || !body.lastname.match(namesRegex)){
    return res.status(500).json({
        success: 0,
        error: "Your firstname and lastname can only contain letters, accents, blank space and '-'."
    })
}

//Verifying email format
if (!body.email.match(/^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$/)) {
    return res.status(500).json({
        success: 0,
        error: "Your email is incorrect."
    })
}
```

Extrait fichier usersControllers.js - contrôle des champs email, firstname, lastname

Les **requêtes** exécutées sont également **sécurisées**. En effet,les requêtes sont construites avec des **paramètres de substitutions (placesholders)** qui permettent de **prévenir les attaques** de type injections SQL, les valeurs étant traitées en tant que **données brutes** et non pas concaténées dans la requête SQL :

```
pool.query(
    'insert into users (firstname, lastname, email, password, role_id, created_at, updated_at, avatar) values (?,?,?,?,?,?,?,?)',
    [
     data.firstname,
     data.lastname,
     data.email,
     data.password,
     defaultRole,
     created,
     data.updated_at,
     data.imageURL
    ],
```

Extrait fichier usersServices.js - contrôle des champs email, firstname, lastname

B. Veille et documentation anglophone

La réalisation de ce projet a nécessité une **veille** et de la **documentation régulière**, principalement en anglais. En premier lieu, nous avons dû consulter la **documentation de NodeJS** (pour la création de l'API notamment), puis, celles de **React** et **React Native** disponibles sur **React Blog** et React **Native Blog**. Nous nous sommes également abonnés à **React Status** une **newsletter** hebdomadaire qui nous permettaient de suivre les **nouvelles fonctionnalités** et **tutoriels** ainsi que **Medium**, une plateforme dédiée aux développeurs qui partagent des articles, tutoriels et notamment en React et React Native. Nous nous sommes également documentés sur toutes les **bibliothèques** et **frameworks utilisés** : Express, Expo, React Admin, JWT, websockets, et bien d'autres.

Pour faire le lien avec la rubrique précédente, nous avons effectué des **recherches pour prévenir des injections SQL côté serveur**. Nous avons donc tapé dans notre **moteur de recherche** les **mots clés** "prevent injections sql react nodeJs" et avons trouvé un article du site **PlanetScale** qui dépeint les différentes méthodes possibles. Voici ci-dessous un extrait et sa traduction :

- **Extrait de l'article How to prevent sql injection attacks in NodeJs du site PlanetScale.com**

Use placeholders

Therefore, you should never accept raw input from a user and input it directly into your query string. Instead, you should use placeholders (`?`) (or parametrized queries) which would look like this (notice the `?` as the placeholder):

Javascript

```
const query = 'SELECT * FROM Repository WHERE TAG = ? AND public = 1'
const [rows] = await connection.query(query, [userQuery])
```

By using placeholders, the malicious SQL will be escaped and treated as a raw string, not as actual SQL code. The end result query would look like this:

SQL

```
SELECT * FROM Repository WHERE TAG = `javascript';--` AND public = 1;
```

Thanks to using placeholders, the malicious SQL is not run and instead, is treated as a search query as intended.

Input validation

In addition to using placeholders, you can add logic in your applications to prevent invalid user input. Let's stick with the example of querying public repositories by tag. For demo purposes, you can assume that you should not have a tag that includes special characters or numbers. In other words, tags should only use capital and lowercase letters (A-Z, a-z).

This means you can add logic to your application to validate that user input matches the correct formatting (no numbers and no special characters). To do this, you can create a regex pattern to match the user input. If it doesn't match, return an error.

Javascript

```
app.get('/repositories/:userQuery', async (req, res) => {

   const {userQuery} = req.params;
   const onlyLettersPattern = /^[A-Za-z]+$/;

   if(!userQuery.match(onlyLettersPattern)){
     return res.status(400).json({ err: "No special characters and no numbers, please!"})
   }

   ...
 });
```

Now the code doesn't even get to the SQL part unless a valid input is passed. You can apply this method with any sort of validation that is relevant to your data. For example, if you allow the user to query by an **`id`** property which should be a number, you can throw an error if the input isn't a valid number.

Javascript

```
 app.get('/repositories/:id', async (req, res) => {
   const {id} = req.params;

   if(isNaN(Number(id))) {
     return res.status(400).json({ err: "Numbers only, please!"})
   }
...
```

- **Traduction de l'article :**

Utilisez des paramètres de substitutions plutôt que directement les variables

En conséquence, vous ne devriez jamais accepter les champs saisis par l'utilisateur et les insérer directement dans votre requête. Vous devriez plutôt utiliser des "placeholders" (paramètres de substitutions) qui ressembleraient à cela (remarquez le ? en tant que placeholder):

Javascript

```
const query = 'SELECT * FROM Repository WHERE TAG = ? AND public = 1'
const [rows] = await connection.query(query, [userQuery])
```

En utilisant des placeholders, le SQL malin sera échappé et traité comme une chaîne de caractère et non comme du code SQL. Le résultat final ressemblera à cela :

```SQL
SELECT * FROM Repository WHERE TAG = `javascript';--` AND public = 1;
```

Grâce à l'utilisation de placeholders, le SQL malin ne sera pas exécuté et traité comme une requête de recherche comme prévu.

Validation de saisie

En plus de l'utilisation de placeholders, vous pouvez ajouter de la logique dans vos applications pour prévenir des saisies invalides de l'utilisateur. Afin de le démontrer, on peut supposer que vous ne devriez pas avoir de balise contenant des caractères spéciaux ou des chiffres. En d'autres termes, vous devriez seulement utiliser des lettres majuscules et minuscules (A-Z, a-z).

Cela signifie que vous pouvez ajouter de la logique à votre application pour valider les saisies de l'utilisateur afin qu'elle corresponde au bon format (pas chiffres ni de caractères spéciaux). Pour faire cela, vous pouvez créer un modèle de regex pour valider la saisie de l'utilisateur. Si la saisie ne respecte pas le regex, elle retourne une erreur.

```Javascript
app.get('/repositories/:userQuery', async (req, res) => {

   const {userQuery} = req.params;
   const validTags = ["javascript", "html", "css"];

   if(!validTags.includes(userQuery)){
     return res.status(400).json({err: "Valid tags only, please!"});
   }

   ...
 });
```

Maintenant, le code n'arrivera même pas jusqu'à l'exécution de la requête SQL à moins que la saisie utilisateur soit valide. Vous pouvez appliquer cette méthode à toute sorte de validation qui est cohérente avec vos données. Par exemple, si vous autorisez l'utilisateur d'exécuter une requête avec une propriété id qui devrait être un nombre, vous pouvez renvoyer une erreur si la saisie n'est pas un nombre valide.

VIII. Conclusion

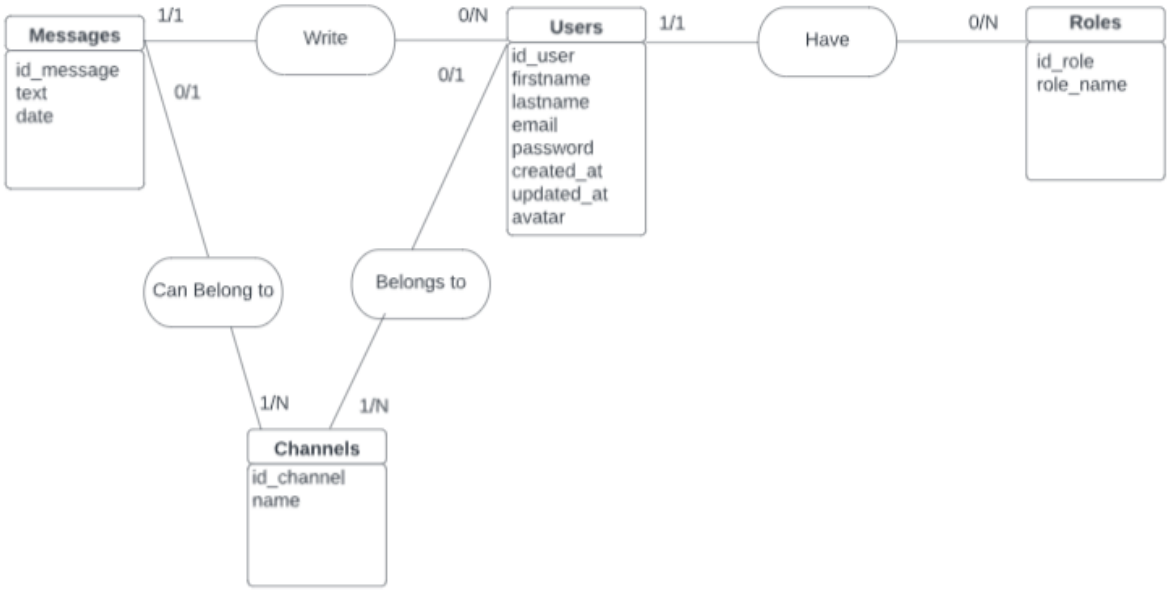
En conclusion, le projet ChatApp nous a permis de développer l'ensemble des compétences nécessaires à la création d'une application en couches : API, application et back-office administrateur. A travers ce projet, nous sommes sortis de notre zone de confort, qui était la création de projet entièrement web.

Couplé à la découverte du L4G, langage de développement que j'utilise dans mon entreprise et au développement en PHP sur un framework maison dans l'application web WMS, je me sens désormais capable d'appréhender les autres technologies de développement web.

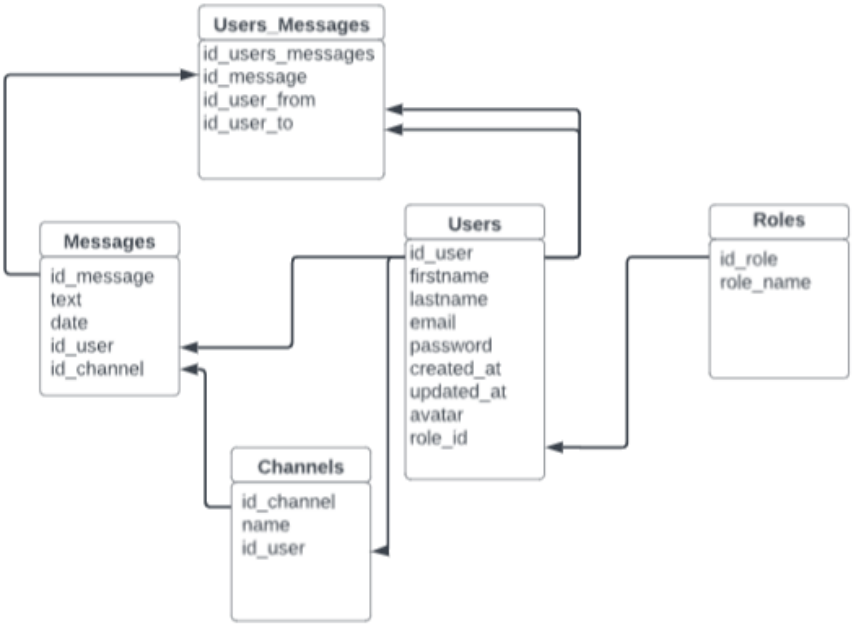
Nous avons finalement découvert de nouvelles technologies, de nouveaux langages, de nouveaux moyens et élargi le champ des possibilités dans nos futures carrières de développeurs.

IX. Annexes

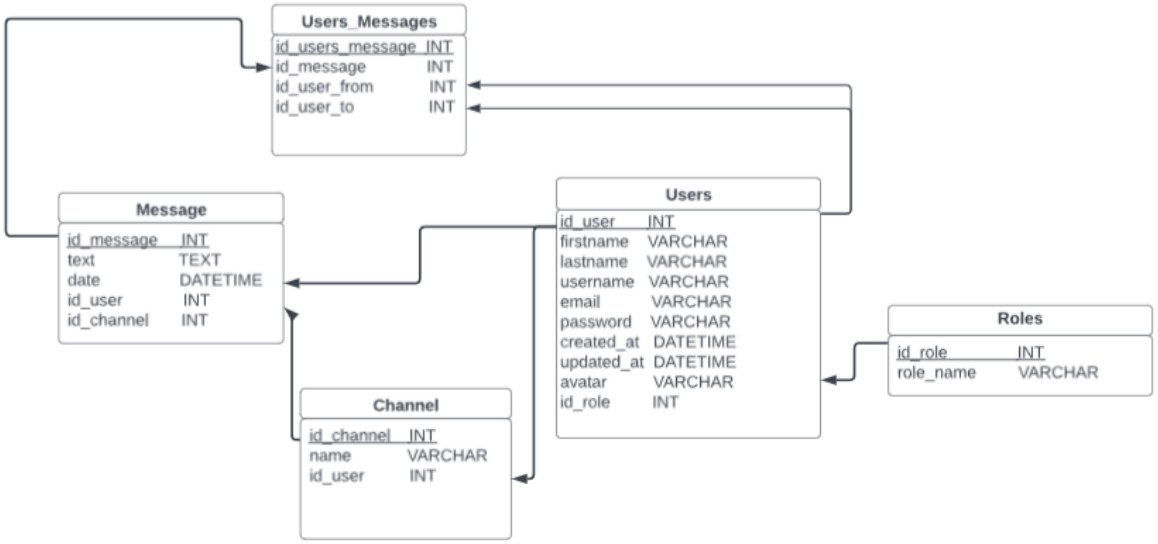
ANNEXE 1 : Modèle Conceptuel de données



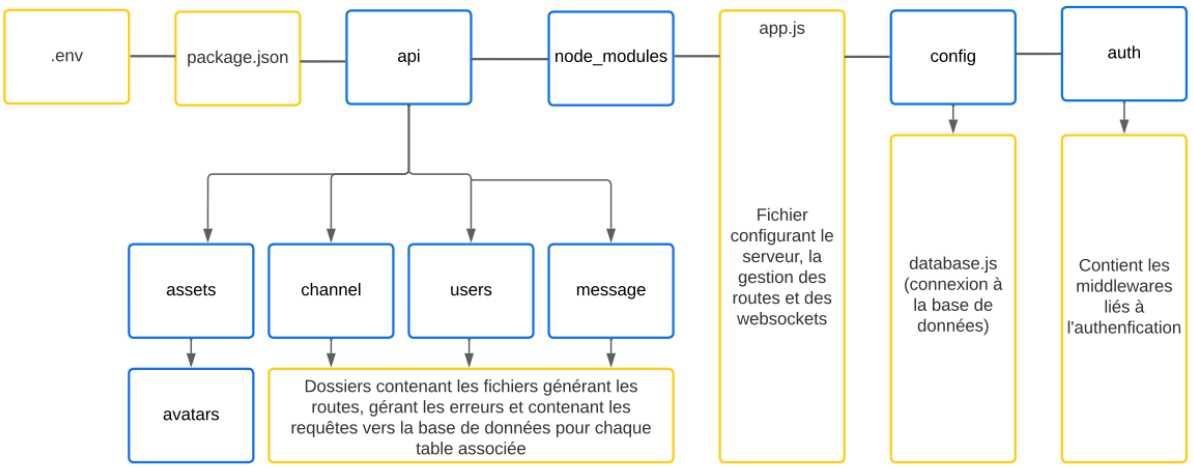
ANNEXE 2 : Modèle Logique de données



ANNEXE 3 : Modèle Physique de données



ANNEXE 4 : Arborescence API



ANNEXE 5 : Arborescence Application Mobile