Plan :

1. 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
* Préparer et exécuter les plans de tests d’une application
* Préparer et exécuter le déploiement d’une application

1. Projet 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.

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 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.

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.

1. 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.

1. 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.

1. 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 canal 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.

1. 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.

1. 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.

**APPELER MAX POUR CONNAITRE LES TECHNOS UTILISES FRAMEWORKS ET BIBLIO CHOISIES ETC (REACT ADMIN)**

1. Gestion de projet
2. 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 donc comment nous avons planifié les tâches :

1. La base de données
2. Définition des tables et des champs associés
3. Modélisation en MCD, MLD
4. Création de la base de données sur PHPMYADMIN
5. L’API
6. Documentation sur l’environnement NodeJS et du framework Express
7. Définition de l’arborescence de l’API
8. Installation de l’environnement
9. Développement des composants d’accès à la base de données
10. Développement des fichiers “Services” comportant des fonctions utilisant les requêtes SQL
11. Développement des fichiers “Controllers” comportant les différents contrôles et la gestion des erreurs
12. Développement et intégration des middlewares
13. Développement des fichiers de “Routes”
14. Phase de tests
15. L’application mobile
16. Documentation sur le framework Expo et la librairie GiftedChat
17. Maquettage de l’application
18. Installation de l’environnement
19. Définition de l’arborescence de l’application
20. Modélisation de la navigation
21. Développement d’un système de navigation temporaire afin de faciliter l’accès aux différents écrans
22. Développement des écrans suivi d’une phase de test après la création de chaque écran
23. Développement d’un nouveau système de navigation intégrant la road map de l'utilisateur
24. Phase de tests
25. Ajustement et améliorations des écrans
26. Phase de tests
27. Le panel administrateur

**APPELER MAX POUR SAVOIR A PEU PRES SA PLANIFICATION**

1. 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 communiquait 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 :

- 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.

1. 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, chacune assignées à chaque collaborateur.

Cet outil permet aussi de changer le statut de chaque Card du statut “A faire” vers “En cours” ou “Terminé”.

**METTRE UN SCREEN DU TRELLO**

1. 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 effecutions 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.

Sur la fin du projet afin d’effectuer, 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.

1. Spécifications techniques et extraits de codes
2. 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( PENSER A FOUTRE LE MCD en annexe)**, et malgré une relation de cardinalité 1/1 à 0/N nous avons jugé nécessaire de créer une table de liaison users\_messages présente 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.

1. L’API
2. 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.

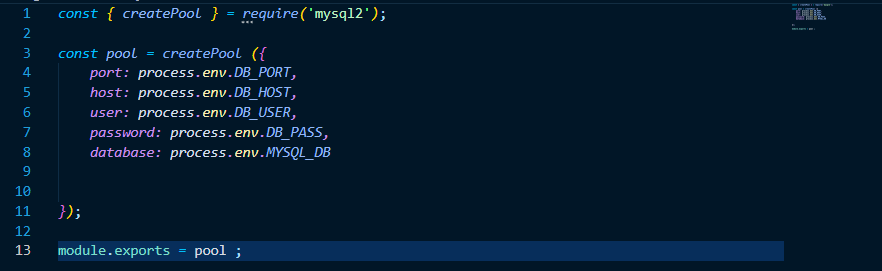
- un sous dossier config contenant le composant d’accès à la base de données

- 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

1. 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. 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)

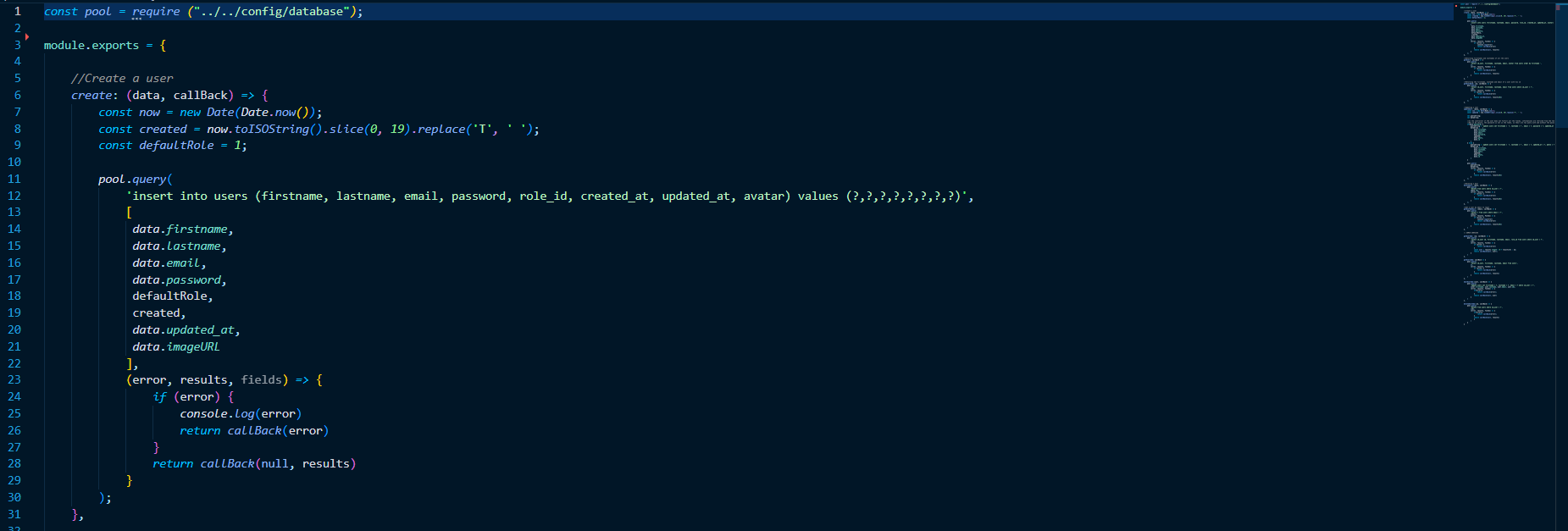
- création de variables supplémentaires nécessaires à l’exécution de la requête

- 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*** 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.

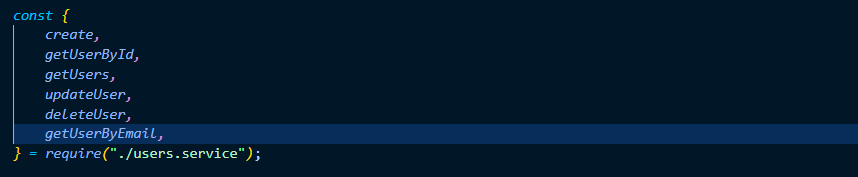
  
 Extrait fichier users.services.js - fonction create permettant de créer un utilisateur

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

1. Structure et fonctions des fichiers “Controllers”

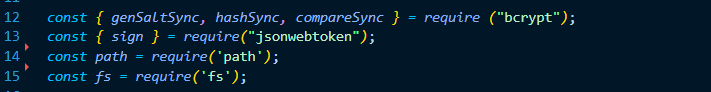
Les fichiers ***.controller*** permettent d’effectuer la ***gestion des erreurs*** et de mettre en place les ***mesures de sécurité*** sur les données entrantes saisies par l’utilisateur et envoyées par le client. La structure est identique pour chacun des fichiers users.controller.js, channel.controller.js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

On importe d’abord les fonctionnalités liées aux contrôles effectués :



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 :

  
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. Pour chaque fonctionnalité de contrôle, on va suivre 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 envoyées 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

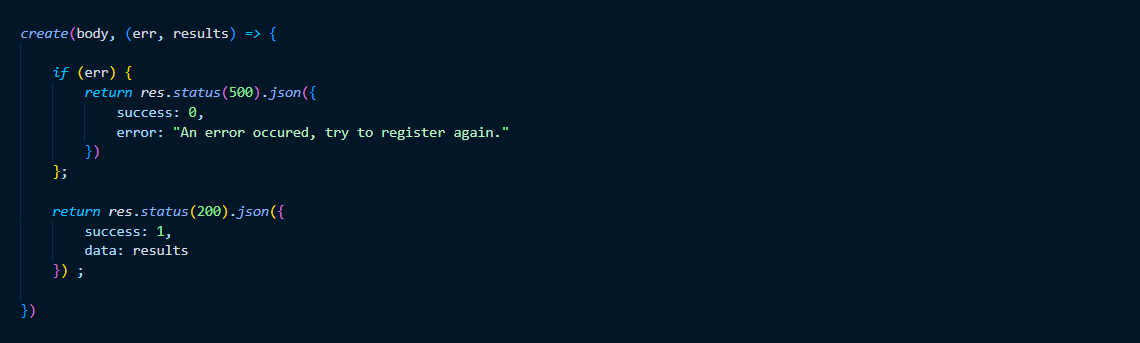


Extrait fichier users.controller.js - fonction de contrôle createUser - contrôles divers

Dans le cas d’une erreur, on retourne un statut de réponse ainsi que deux données success à 0 ainsi que 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).

* 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 retournera un statut 200, success à 1 ainsi que le résultat de la requête qui contient les données envoyées.



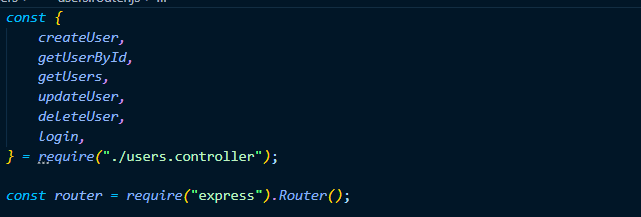
Extrait fichier users.controller.js - fonction de contrôle createUser - appel de la fonction create du fichier users.service.js

1. Structure des fichiers “Routes”

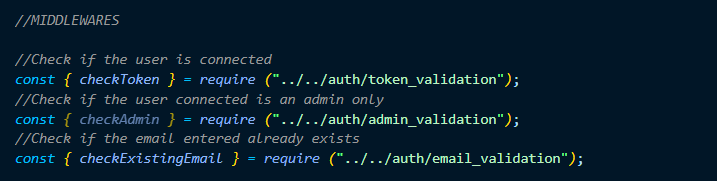
Les fichiers .router permettent de construire les routes de l’API qui seront appelées dans l’application afin d’utiliser les fonctionnalités. Ils sont également répartis selon les tables users, messages et channels.

Voici leur structure :

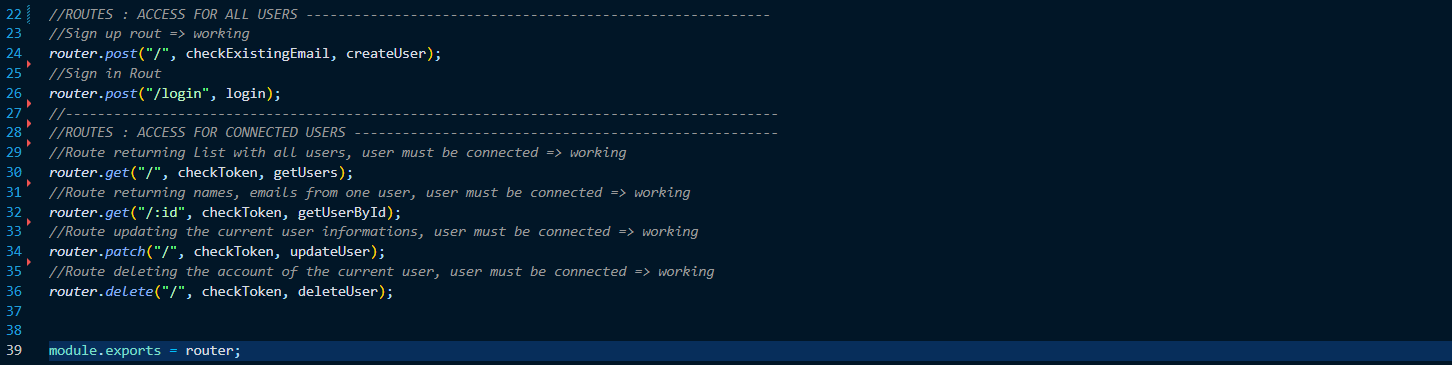
* On importe les fonctions issues des fichiers .controller associés aux routes et aux requêtes SQL à exécuter
* On crée une instance du routeur d’Express, le framework pour NodeJs afin de définir les routes et les actions correspondantes pour gérer les demandes HTTP (get, post, put, delete…)



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

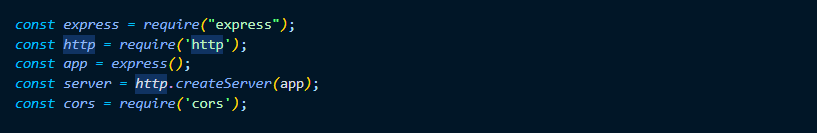


* Enfin on crée les différentes routes spécifiques en utilisant les méthodes .get(), .post(), .patch(), .delete() etc… . On définit pour chacune le chemin d’accès “/”, “/login” par exemple puis les fonctions importées à appeler et à exécuter les unes après les autres. Ces routes seront exportées grâce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



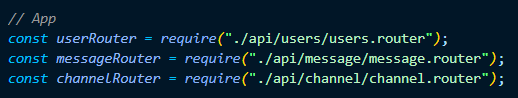
1. Le fichier app.js : serveur et racine des routes

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 dans le point **(PENSER A METTRE LE POINT DANS LEQUEL ON PARLE DES SOCKETS).**

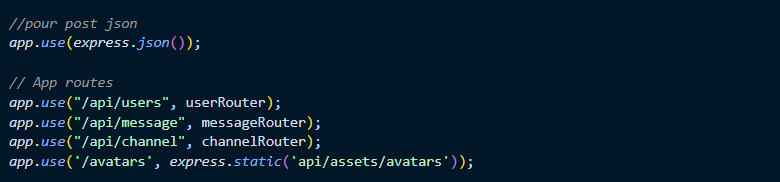




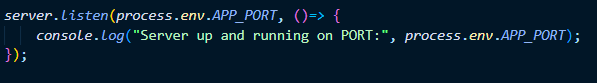
Ces lignes de codes viennent configurer le serveur. On crée une instance du framework Express, du module HTTP de Node.js(http), de l’application Express(app) et on créé le serveur http(server) avec la fonction createServer et on instancie le middleware CORS(Cross Origin Resource Sharing), qui sera activé par la suite afin de pouvoir gérer les requêtes provenant des différentes origines (application, back-office administrateur).



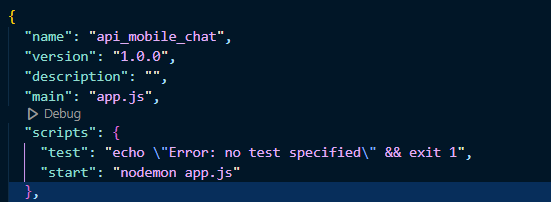
Par la suite on instancie les fonctionnalités des différents fichiers de routes dont le fonctionnement a été explicité dans le point précédent.



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.



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.

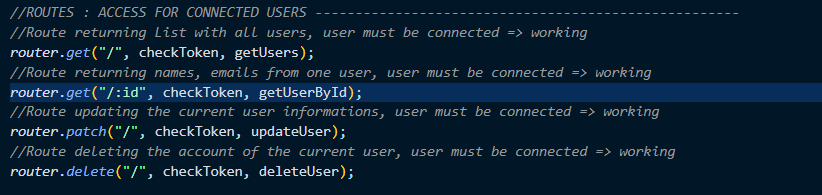


1. Les middlewares

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.

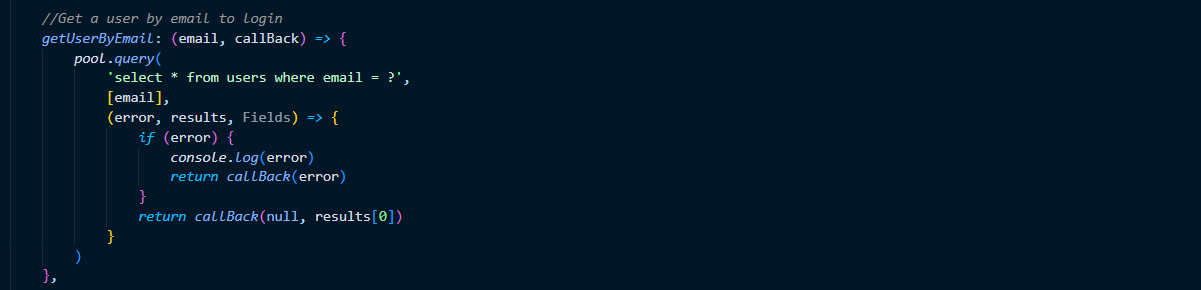
Pour mieux comprendre la logique d’un middleware, je vais reprendre l’exemple de la vérification du token des routes ci-dessous :



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.

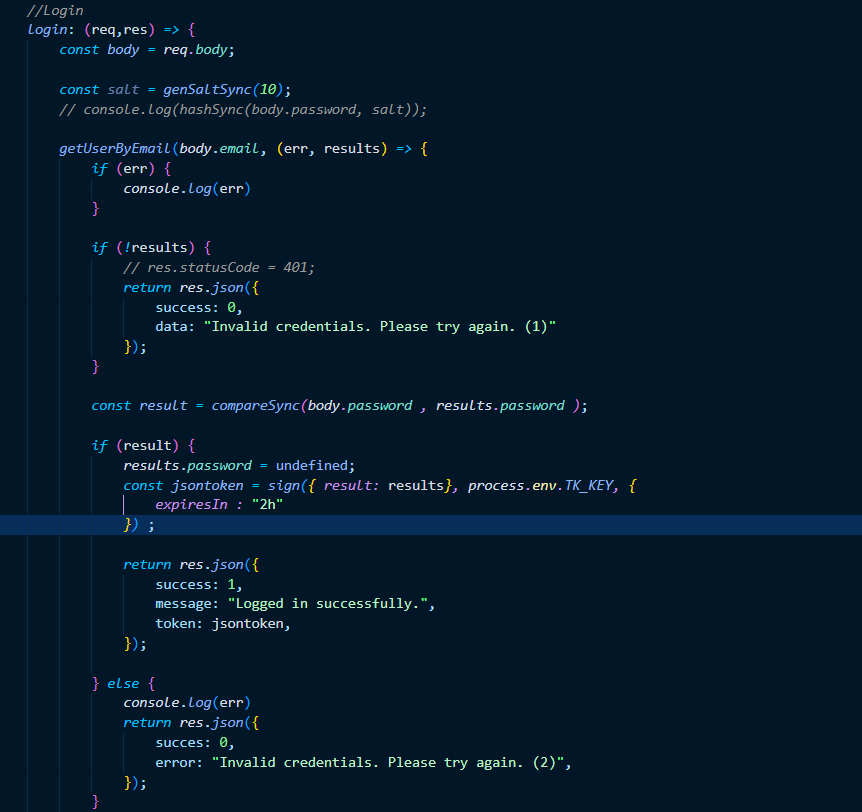
1. La fonction de connexion

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 permet de récupérer l’ensemble des informations d’un utilisateur après qu’il ait rempli le formulaire de connexion.



La route route users/login est appelée dans l’application mobile lorsque l’utilisateur appuie sur le bouton connexion de la page d’accueil de l’application.

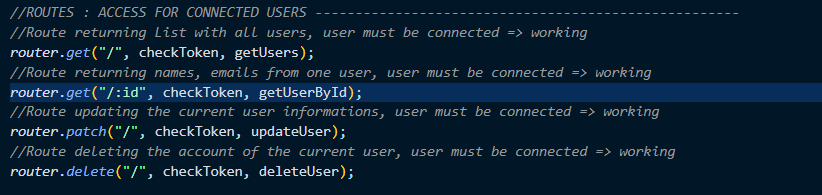


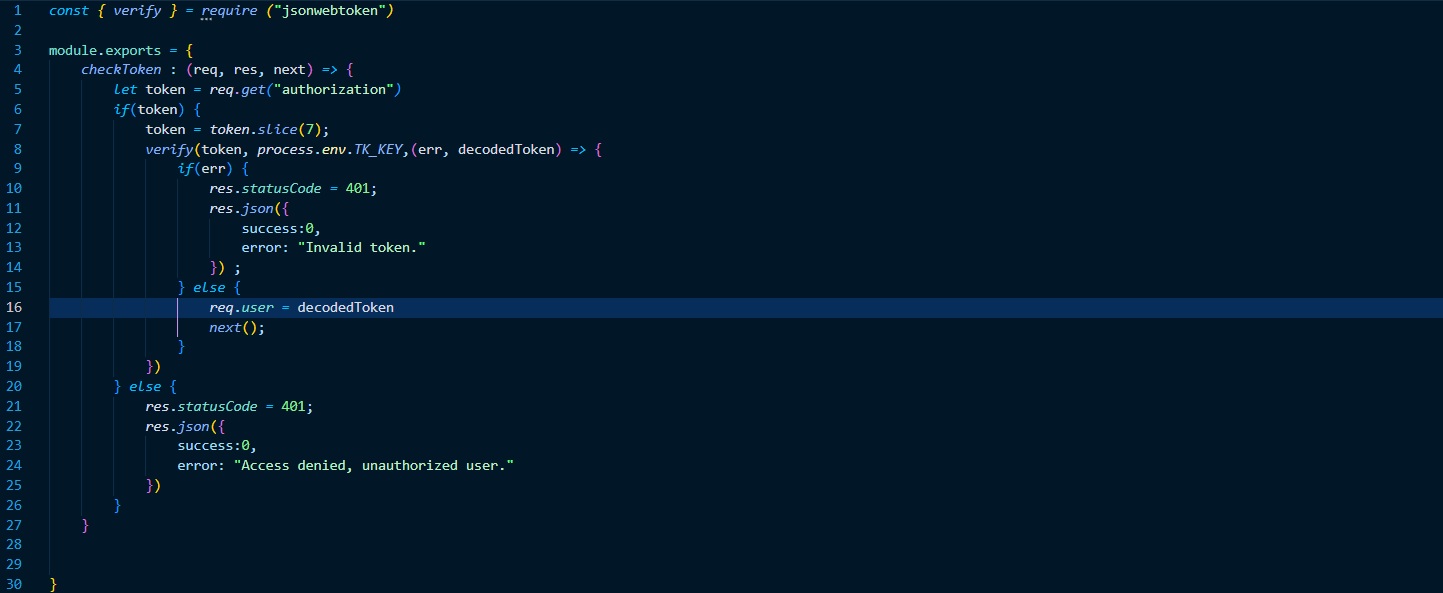
Cette route va donc faire appel à la fonction login du fichier users.controllers.  


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.

1. 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….).



Comment fonctionne donc la vérification du token ?  
  


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.

1. 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

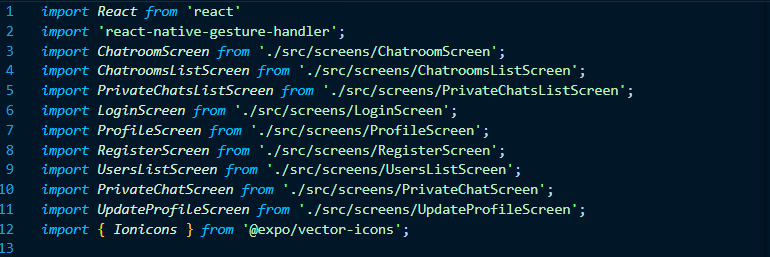
Pour le maquettage, nous avons utilisé l’outil FIGMA. Vous pourrez en retrouver une partie en annexe 6. Nous avons d’abord définit une page de connexion/inscription, puis une navbar et la structure générale de chaque page.

c. Installation de l’environnement

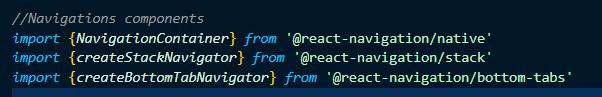
Nous avons développé cette application dans un environnement Node.js et avons également installé React Native, le framework Javascript utilisé pour développer des applications multiplateformes ainsi qu’Expo, qui est un ensemble d’outils qui facilitent le développement d’applications React Native.

d. Structure de la navigation

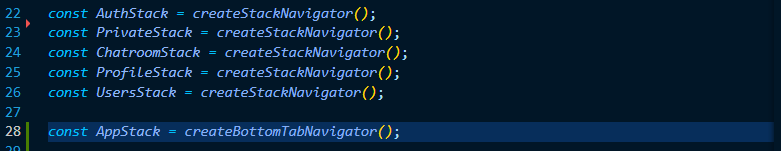
La navigation est gérée dans le fichier App.js.Elle nécessite tout d’abord d’importer les différentes fonctions qui viennent générer les écrans de l’application :



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



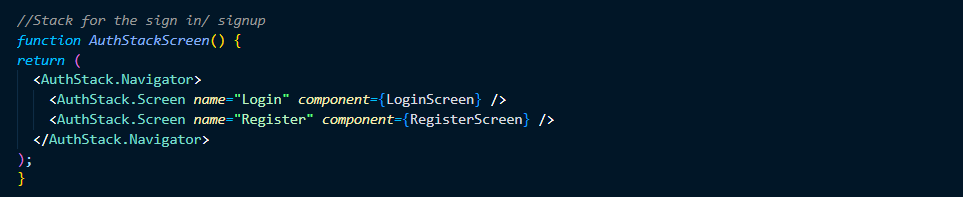
* NavigationContainer, qui va être le contenant racine de la navigation et va la gérer dans sa globalité en fournissant le contexte de navigation à tous les composants.
* createStackNavigator, qui va permettre de générer des “piles” de navigation, en empilant successivement plusieurs écrans les uns avec 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 utilisant, nous donnera accès alors pour chaque bouton à un navigateur empilé (stack navigator) différent.



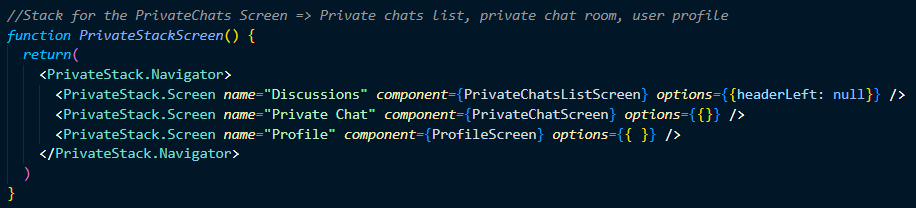
Ensuite nous utilisons la fonction createStackNavigator que nous instancions à des constantes pour créer nos différents stack navigators ainsi que createBottomTabNavigator pour créer la barre de navigation. Nous allons maintenant décortiquer la structure de la navigation en elle même.

1. Les stacks navigators (navigateurs empilés)\*

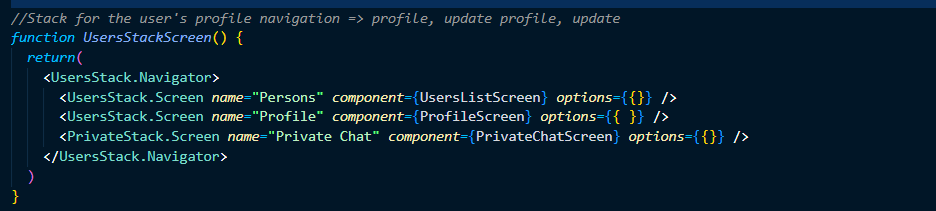
Comme expliqué précédemment, les stacks navigators permettent de créer un empilement dans lequel on pourra naviguer d’un écran à l’autre. Voici quelques exemples :



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 le nom à l’affichage sera 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 rebasculer sur l’écran de connexion.



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.

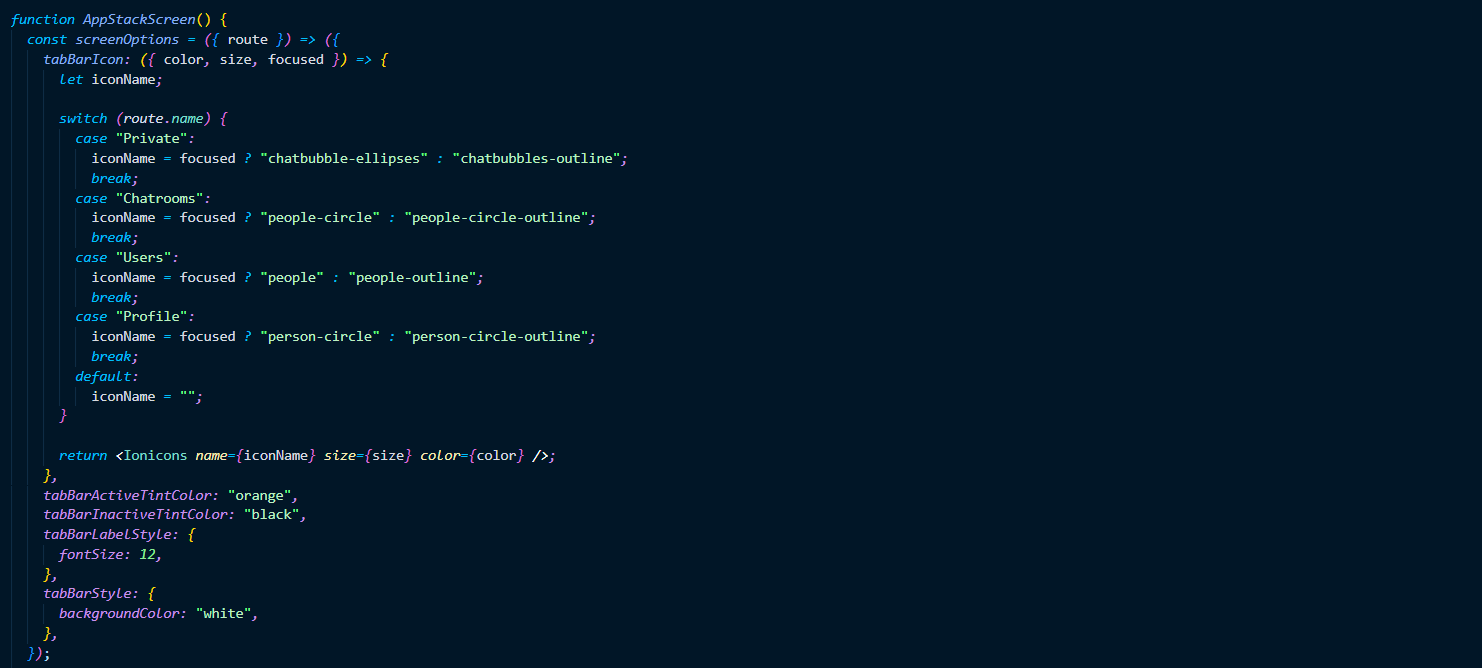


Dans ce dernier exemple, le navigateur UsersStackScreen, le premier écran et le UsersListScreen qui affiche la liste des utilisateurs de l’application. En cliquant sur un utilisateur, on aura accès au profil de l’utilisateur sélectionné, ProfileScreen, et un bouton dans cet écran permettra d’ouvrir la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

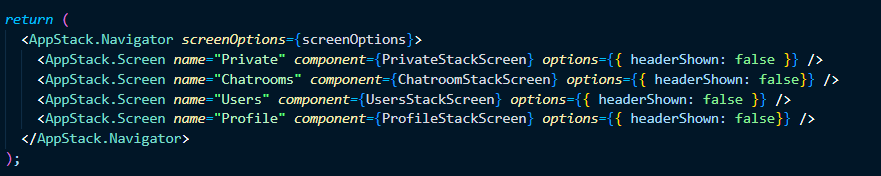
Les deux derniers navigateurs empilés sont le ChatroomStackScreen qui amène l’utilisateur sur l’écran ChatroomsListScreen qui affiche la liste des salons de discussions et à partir duquel il aura accès au ChatroomScreen, l’écran d’un salon de discussion, ainsi que le ProfileStackScreen qui amène d’abord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

1. La barre de navigation

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….) .

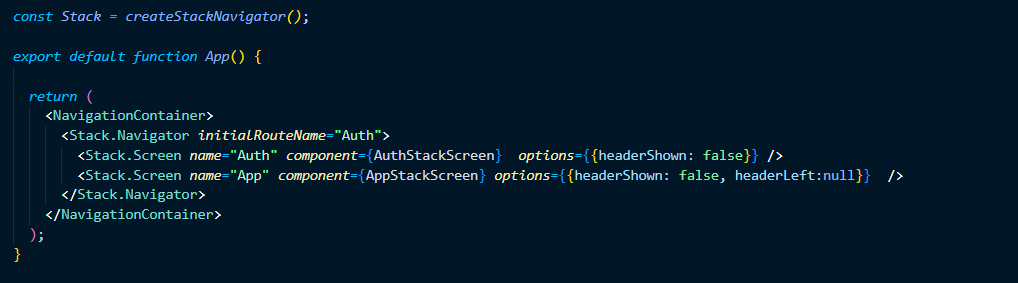


Enfin, la fonction retourne donc les composants de la barre de navigation. On y injecte les différents stacks navigators générés précédemment. Lorsque l’utilisateur clique sur une icône, il aura donc accès aux écrans en fonction du stack navigator associé.

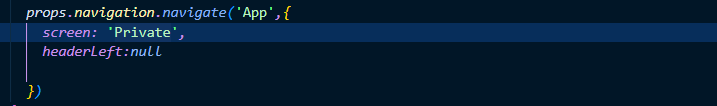


Donc si l’utilisateur clique sur l’icône de l’écran Users dont le composant est le UsersStackNavigator, il sera amené en premier lieu sur la liste des utilisateurs, et aura accès aux écrans suivants qui y sont empilés, c’est-à-dire l’écran du profile utilisateur puis l’écran de discussion privée.

1. La navigation centralisée dans le NavigationContainer



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



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.

FAIRE DEMO DE LA STRUCTURE SIMPLE DE LECRZAN USERSLISTSCREEN PUIS BASCULER SUR LA PARTIE SUIVANTE EN MODE : OUAI DU COUP LE USER EN CHOISI UN, TOMBE SUR SON PROFIL, ET PEUT APPUYER SUR LE BOUTON POUR DISCUTER

g. Le chat : fonctionnement et sockets

1. Le back-office administrateur
2. Jeu d’essai
3. FAIRE UN TEST DE PLUSIEURS FONCTIONNALITES AVEC POSTMAN POUR MONTRER LOUTIL
4. Veille
5. Recherche en anglais
6. Annexes
7. ANNEXE 1 : MCD
8. ANNEXE 2 : MLD
9. ANNEXE 3 : MPD
10. ANNEXE 4 : ABORESCENCE API
11. ANENXE 5 : ABORESCENCE APPLICATION
12. ANNEXE 6 : MAQUETTAGE DE L’APPLICATION