DOSSIER DE PROJET PROFESSIONNEL

La Plateforme

Réalisation d’une application mobile de chat

-

Lucas Verdier

Introduction

Je m'appelle Lucas Verdier, j'ai obtenu mon diplôme de "Développeur web et web mobile" en 2022. Cette année, je poursuis ma formation au sein de La Plateforme pour obtenir le diplôme de "Concepteur Développeur d'Application". J'effectue mon année en alternance dans la société ViaXoft à Marseille. Une entreprise qui produit des applications web qui accompagnent les entreprises du tourisme dans leur métier.

Présentation du projet en Anglais

Compétences couvertes par le projet

Le projet présenté couvre les compétences du REAC suivantes :

- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement.
- Concevoir une application.
- Développer des composants métier.
- Construire une application organisée en couches.
- Développer une application mobile.
- Concevoir une base de données.
- Mettre en place une base de données.
- Développer des composants dans le langage d'une base de données.
- 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.

ORGANISATION & CAHIER DES CHARGES

Les utilisateurs du projet

Ce projet se décompose en deux parties, destinées chacune à un type d'utilisateurs différents : une partie application mobile et une partie interface web admin.

L'application mobile est destinée à toute personne souhaitant utiliser une application de chat et inscrite sur l'application. Les utilisateurs devront impérativement s'inscrire, et auront ensuite la possibilité d'échanger avec les autres utilisateurs de l'application.

L'interface web Admin est destinée aux administrateurs de l'application ARCO. Les utilisateurs ayant le rôle "admin" auront accès à un dashboard permettant de gérer les utilisateurs et les chatroom (CRUD).

Les fonctionnalités attendues

Application mobile

Page d'accueil :

Lorsque l'utilisateur lance l'application, s'il n'est pas authentifié, il arrive sur une page avec deux boutons, un pour se connecter et un pour s'inscrire. Si l'utilisateur est déjà authentifié (token d'authentification toujours actif) il arrivera directement sur une page listant les différentes chatroom.

Page “Channels”:

Une page qui répertorie toutes les chatrooms disponibles. Il est possible de cliquer sur chacune d’entre elles afin d’avoir accès aux messages de la chatroom.

Page “InChannel”:

Une page qui représente l’intérieur d’un channel, avec tous les messages, l’utilisateur qui a envoyé le message et la data où le message a été envoyé. D’ici, il est possible d'interagir avec les utilisateurs présents dans la room. Au moyen d’une text area l’utilisateur peut rédiger son message et l’envoyer.

Page “Profile”:

Une page où sont affichées les informations de l’utilisateur, sur cette même page il est également possible de modifier ses informations.

La navigation sur l’application se fait grâce à un footer (bottom-navigation-bar), où les pages sont représentées par une icône. La bottom-navigation est toujours affichée à partir du moment où l’utilisateur est authentifié.

Application Web

Contexte technique

L’application mobile est développée dans le but d’être disponible sur IOS et Android.
L’application web sera accessible sur tous les navigateurs.

CONCEPTION DU PROJET

Choix de développement

Choix des langages :

Pour la réalisation du projet, j'ai utilisé JavaScript pour le back-end et le front-end.

Et NodeJs comme environnement de développement pour exécuter du code JavaScript côté serveur.

J'ai fait ces choix car JavaScript est un langage dynamique et polyvalent qui peut être utilisé à la fois côté serveur et à la fois côté client et permet ainsi de créer une application avec un seul langage.
De plus JavaScript possède une très large communauté et ainsi beaucoup de librairies sont disponibles pour répondre aux besoins de l'application.
Dernièrement, JS est un langage de programmation directement interprété par le navigateur et ne nécessite pas d'installation sur la machine de l'utilisateur.

Framework Back-end:

ExpressJS est un framework populaire pour NodeJs, il apporte peu de surcouche et laisse ainsi une liberté d'action et de très bonne performances. Il n'impose pas d'architecture et s'adapte donc parfaitement aux besoins de l'application.
De plus, ExpressJS utilise un système de middleware qui permet de gérer facilement l'authentification par le biais de tokens par exemple.
Dernièrement, étant très populaire, ExpressJS bénéficie d'une large communauté, il est donc très facile de se documenter ou bien trouver des solutions aux problèmes rencontrés.

Libraire front :

React JS est une libraire moderne (2016) développée par Facebook qui a beaucoup de succès. Son approche sous forme de composants réutilisables permet une large personnalisation sans surcharger le code. Il est plus facile à maintenir car chaque composant à ses propres propriétés. Aujourd'hui, REACT JS s'impose comme une des librairie JS

les plus utilisées aujourd'hui, il est donc très facile également de se documenter.

Nous utilisons ReactJS pour l’application web et React Native pour l’application mobile.

React Native est très similaire à React JS dans son fonctionnement (composants) et écriture et permet de développer des applications multi plateformes (IOS et Android).

Autres outils utilisés :

- IDE: IntellIJ.
- Postman pour tester les routes API.
- GitHub pour collaborer en équipe et le versioning de notre code.
- Trello pour l'organisation du projet et la gestion des tickets.
- Figma pour maquetter notre application.
- Expo pour émuler notre application sur téléphone.

ORGANISATION DU PROJET

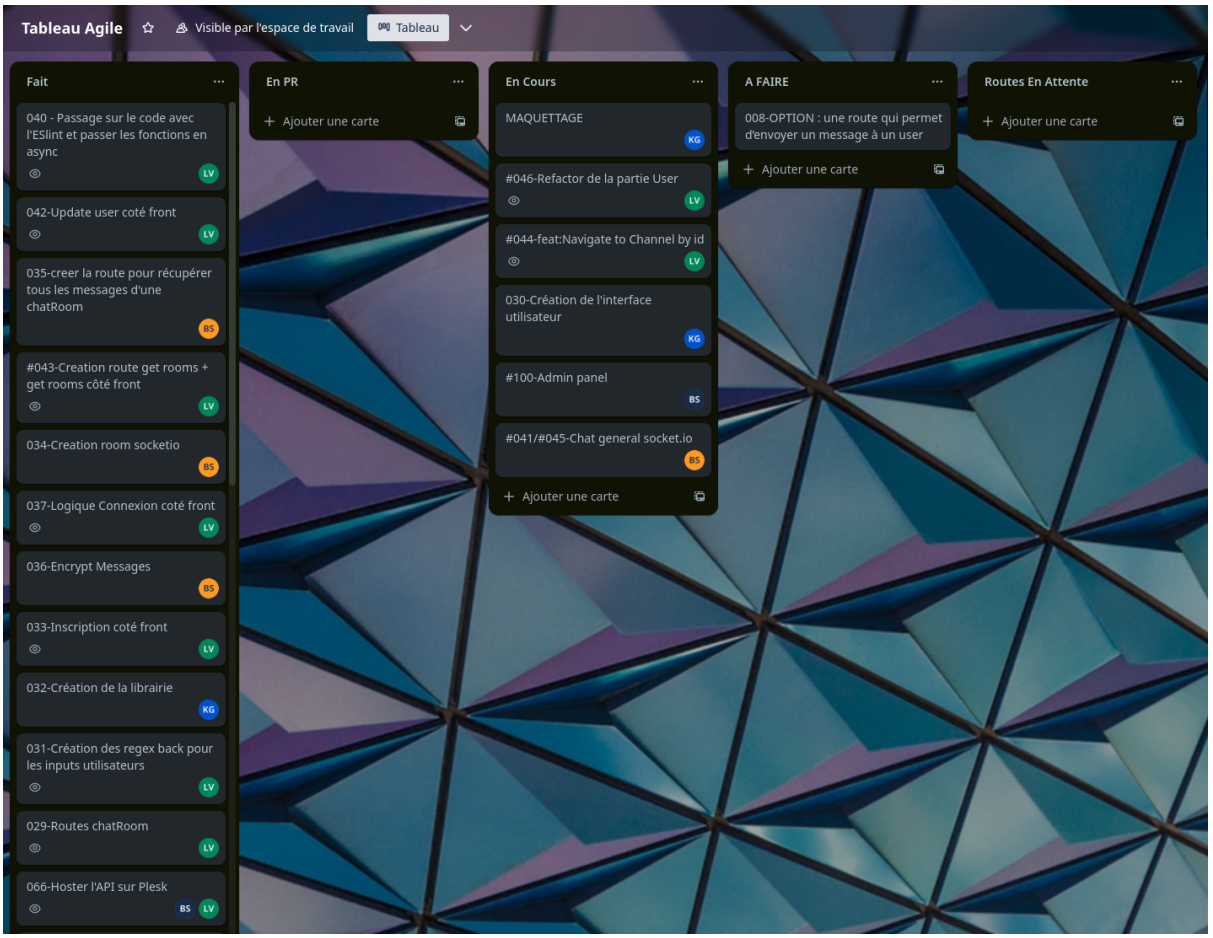
Ce projet à été réalisé dans le cadre de notre formation sur notre temps de présence en centre (du 02/01/23 au 27/01/23).
Ayant un timing serré il a fallu nous organiser et bien répartir les tâches. Pour ce faire nous avons premièrement listé toutes les features par ordre de priorités sur un fichier excel en classant chaque feature selon un MUST, SHOULD, COULD, WISH. Cela nous a permis de déterminer les versions de notre application.

<u>Notre MUST, SHOULD, COULD, WISH pour ARCO :</u>

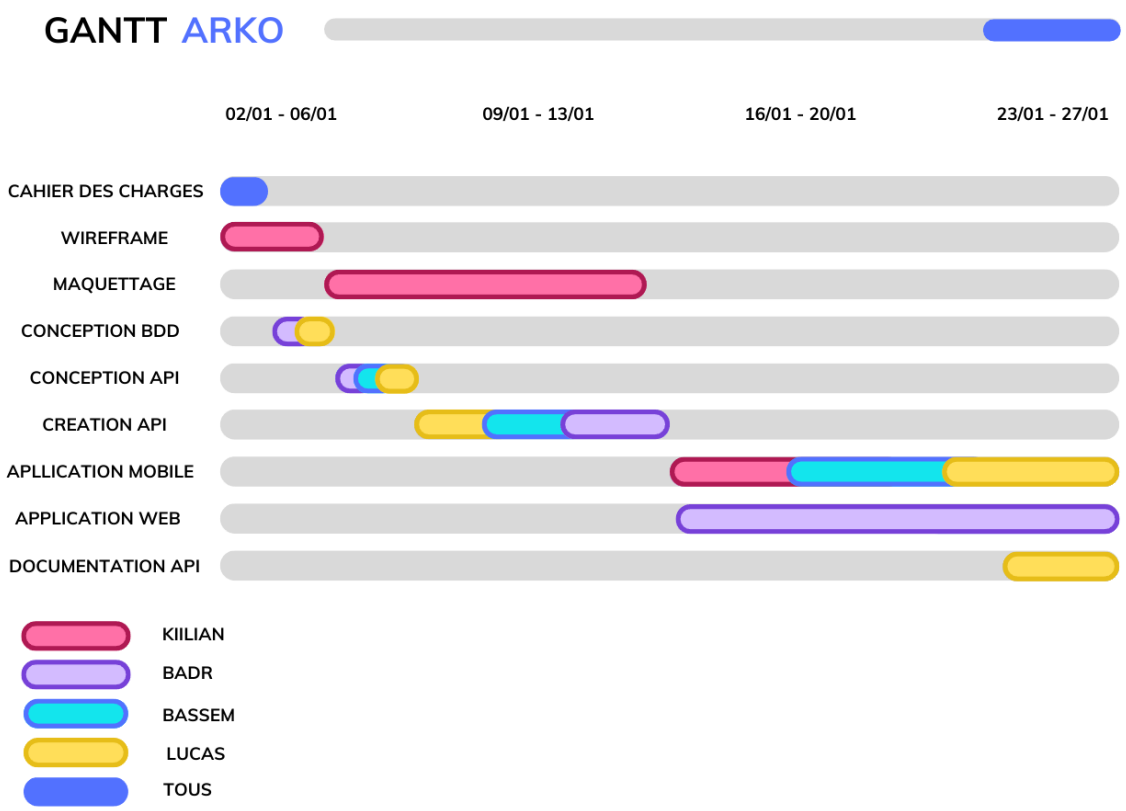
MUST (v1)
1. Une inscription
2. Une connexion
3. Un salon de chat général
4. Une page profile
5. Un annuaire de tous les utilisateurs
6. Une page qui détaille les informations d'un user
7. Panel Admin (CRUD)
8. Un utilisateur peut supprimer son message
SHOULD (v2)
1. Chat privé avec un utilisateur
2. Barre de recherche pour utilisateurs
3. Envoyer des photos/vidéos
4. Authentification double facteur
5. Un utilisateur peut modifier son message
6. Si première ouverture de l'application, tutoriel de comment s'en servir sous forme de slide en se servant des frames Figma
7. Dans l'update modifier son avatar
COULD (v3)
1. Créer des chatrooms avec plusieurs utilisateurs
2. Ajouter un "Répondre" quand on reste appuyer sur un message
WISH (v4)
1. Connexion via Google, facebook...
2. Envoyer un message vocal

Ensuite nous avons établi une timeline à travers un “Diagramme de Gantt”. Cet outil nous a permis de planifier notre projet afin de respecter la première deadline du 27/01/23.

Nous avons ensuite rédigé et assigné les tickets grâce à Trello.



Nous avons également établi un diagramme de Gantt afin d'avoir une timeline à suivre :



Concernant GitHub, afin de collaborer au mieux nous avons définis les règles suivantes :

- Il est impossible de merger directement sa branche avec la branche "Master". Il faut obligatoirement créer une Pull Request, cette dernière doit être review et approuvée par minimum deux membres du groupe. La Pull Request met la demande de merge en attente, les utilisateurs peuvent parcourir le code ajouté par le développeur et soit approuver soit ajouter un commentaire sur un bout de code afin de donner des conseils ou demander des explications.
- Nous avons définis une norme de nommage pour nos branches et commits:

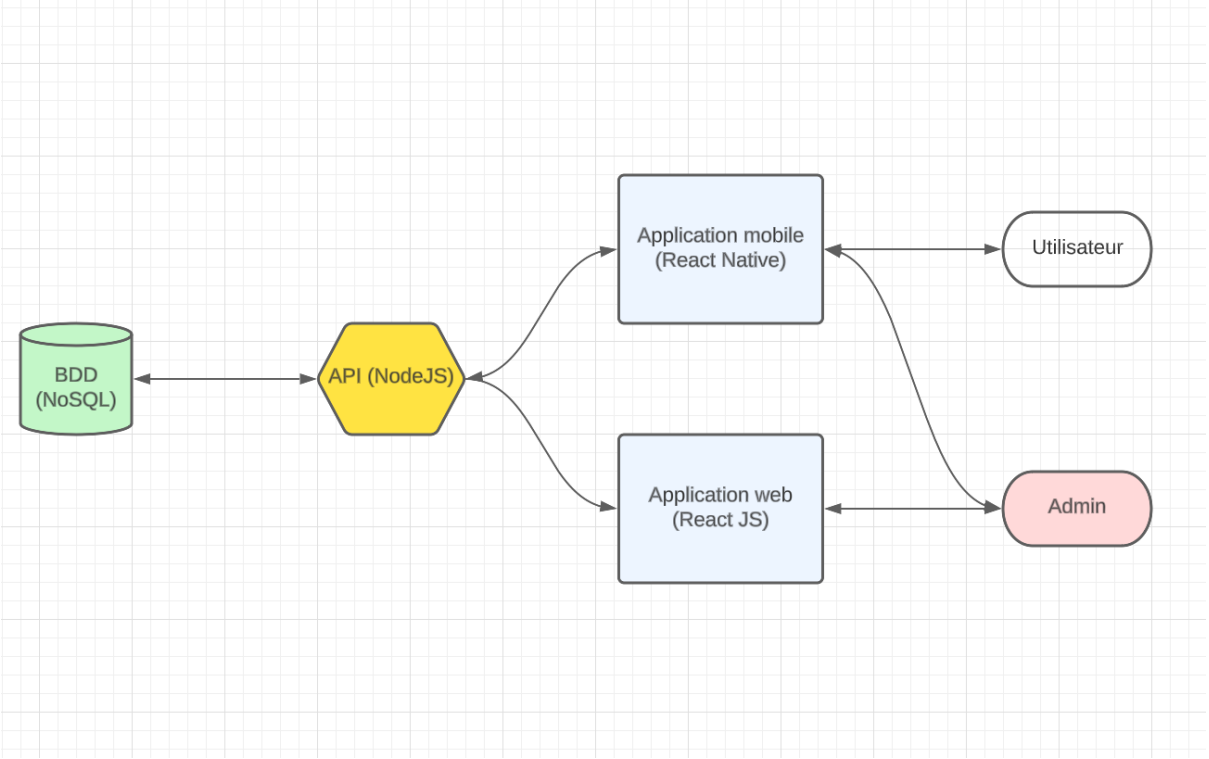
  -> Branches : elles commencent par # suivi du numéro de la tâche (exemple : #042).

-> Commits : <numéro de tâche>-<nature de la tâche>:<descriptif du commit> (exemple: 042-feat:add-get-user-route)

Celà nous a permis d'avoir une unité dans notre façon de travailler.

Architecture des applications :

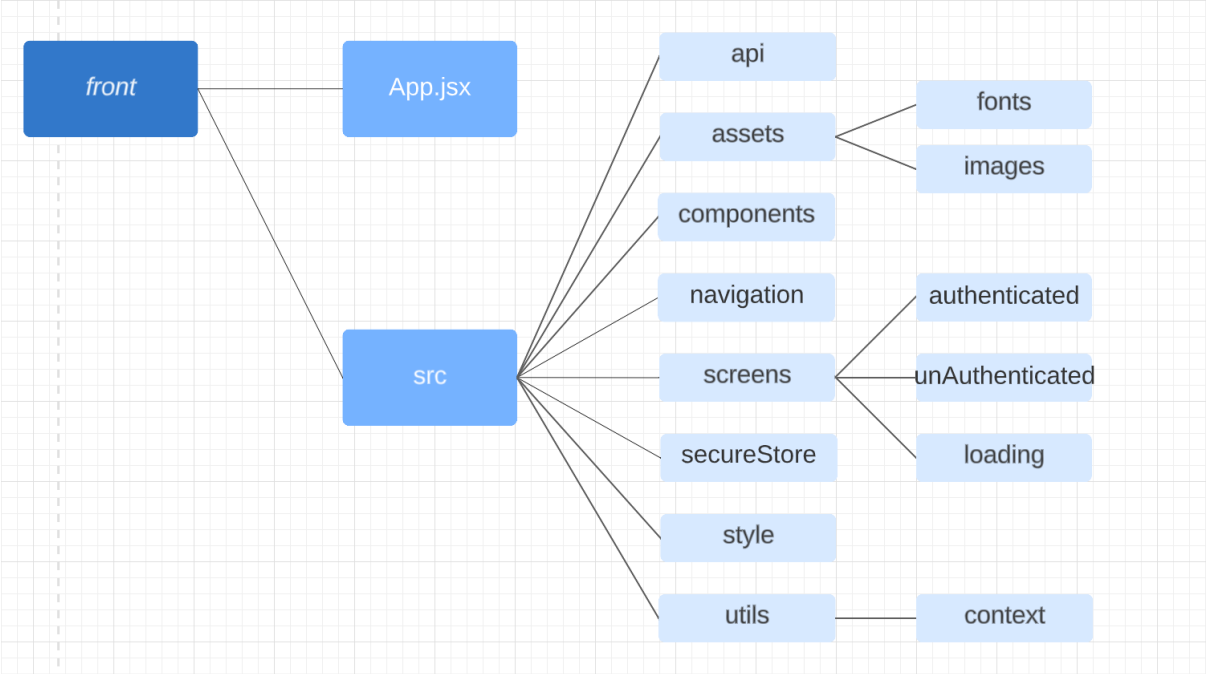
Les deux applications (web et mobile) s'appuient toutes deux sur la même API reliée à la base de données.
Tous les utilisateurs auront accès à l'application mobile, mais seuls les utilisateurs avec le rôle admin auront accès à l'application web.



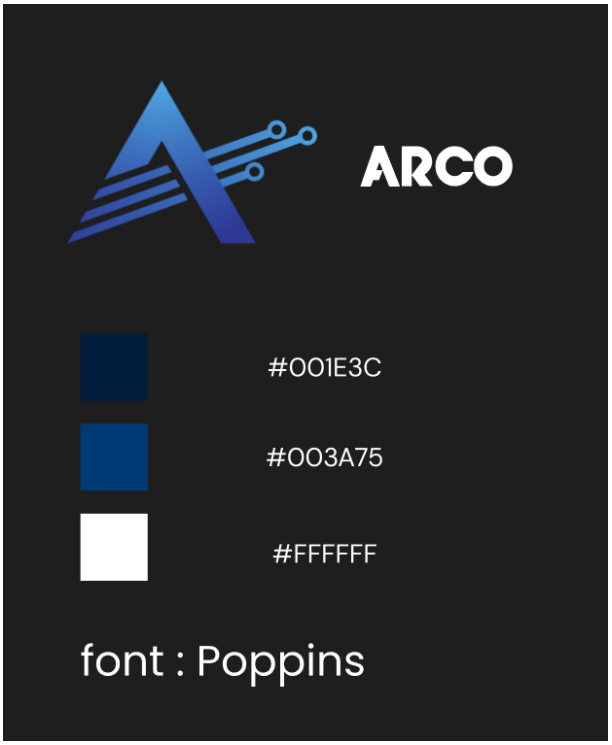
CONCEPTION DU FRONT-END DE L'APPLICATION

Pour créer les maquettes nous avons utilisé Figma, de par sa gratuité, sa simplicité d'utilisation et son aspect collaboratif. Il y est effectivement facile d'y travailler en équipe, car il est possible de travailler à plusieurs en même temps sur le même projet.

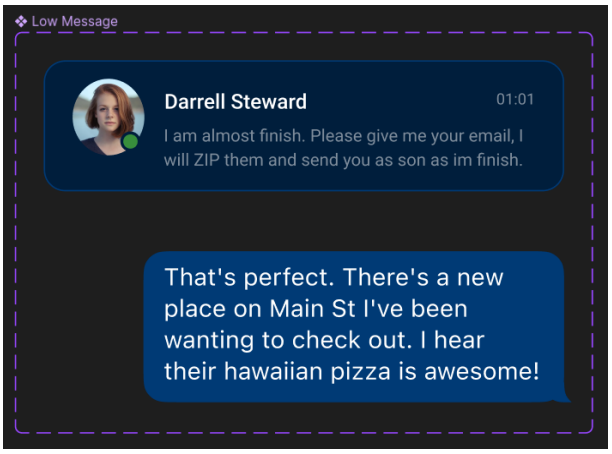
Arborescence du projet :



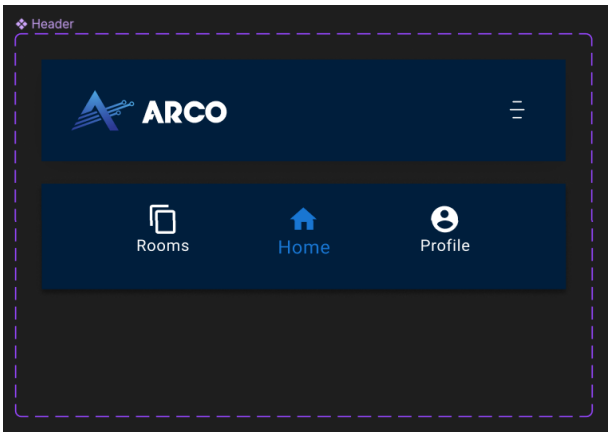
Charte graphique :



Nous avons commencé par établir une charte graphique avec logo, un code couleur ainsi qu'une police d'écriture pour notre application.



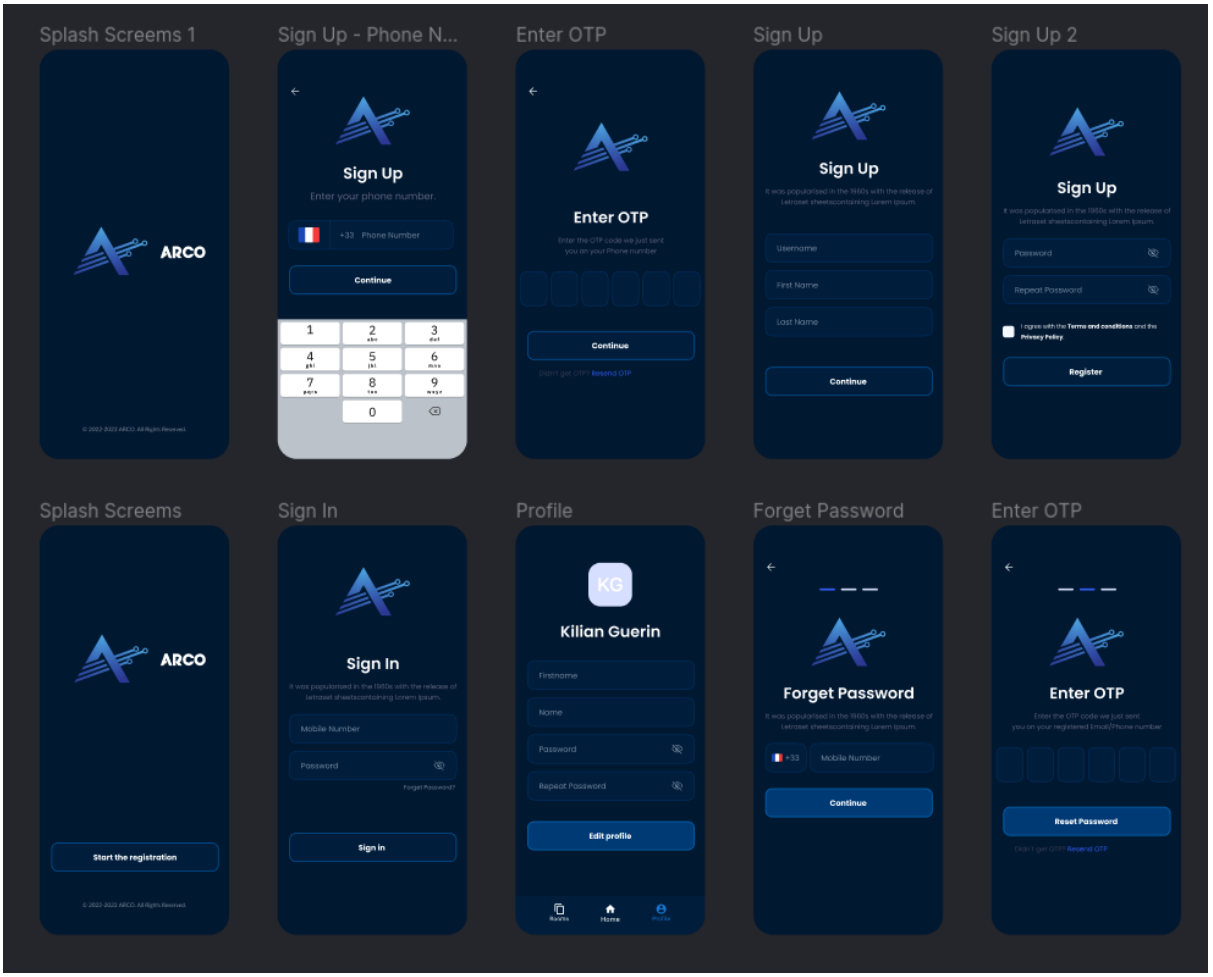
Une fois la charte graphique établie nous avons commencé à créer des composants Figma réutilisables afin de faciliter la création des futures maquettes.



Ici par exemple la bottom-navigation que l'on retrouve dans quasi l'intégralité des écrans.

Maquettage de l'application mobile :

Pour maquette l’application nous avons utilisé Figma, et nous sommes servis des composants précédemment créés.



CONCEPTION DU BACK-END DE L'APPLICATION

Modélisation de la base de données

Nous avons fait le choix de travailler en s'appuyant sur une base de données NoSQL (Not only SQL).

Avantages d'une base de données NoSQL pour une application mobile de chat:

- Modèle de données flexibles: les données peuvent être facilement ajoutées et/ou supprimées sans impacter la structure de la base de données.
- Facilement scalable: permet de gérer un très grand flux de données.
- Disponibilité de la données. Le NoSQL permet d'avoir de très bonnes performances sur la vitesse d'accès aux données.
- Il est aussi possible de travailler avec des données structurées (Not ONLY sql).

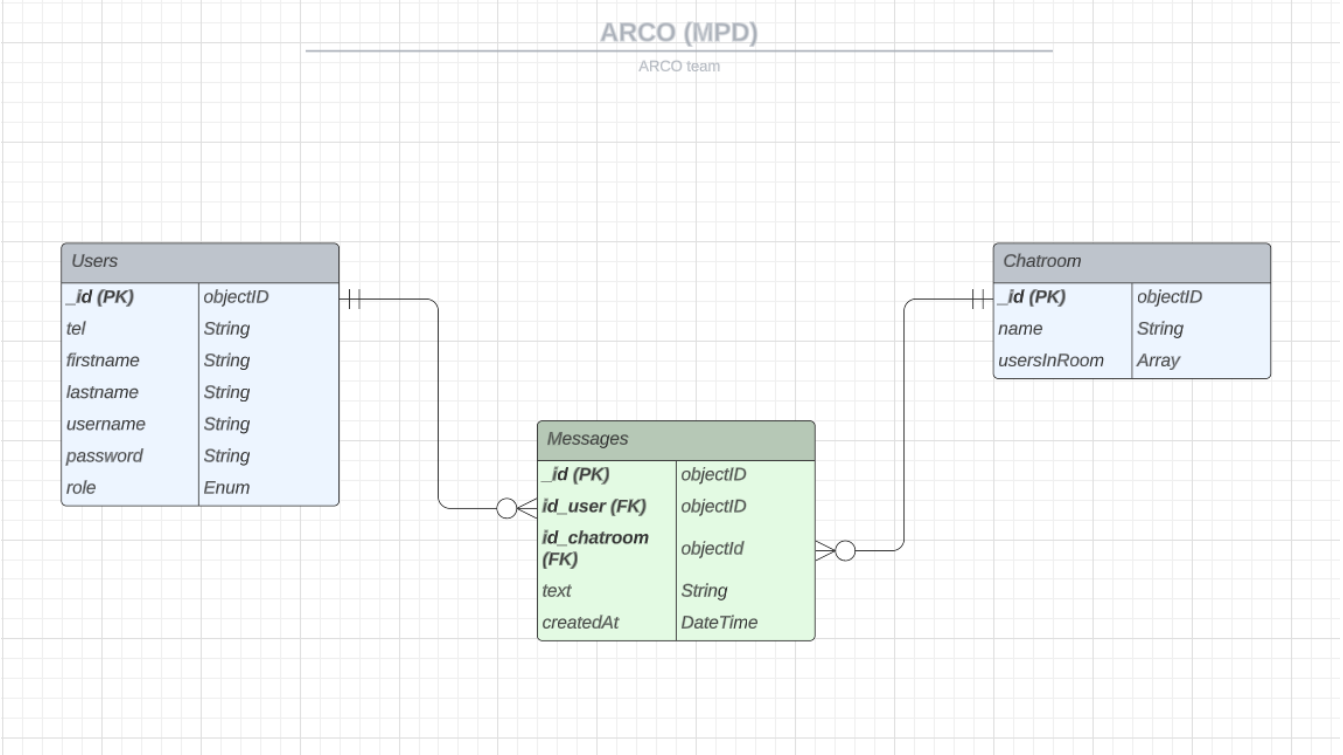
Pour ce faire nous avons choisi MONGO DB comme **système de gestion de base de données.**

Un autre avantage de MongoDB est le fait que la base de données est directement hébergée sur le Cloud de MongoDB. Ce qui facilite le travail d'équipe pendant le développement du projet.

Lors de la conception de la base de données nous avons créé 3 collections :

- User
- Message
- Chatroom

Représentation des collections et relations entre elles :



La collection “Messages” joue le rôle de table intermédiaire. A savoir qu'elle contient obligatoirement un ID USER et un ID CHATROOM (clefs étrangères). Ce qui permet sur mes routes API de pouvoir chercher un message par son auteur mais aussi savoir dans quel chatroom il se trouve.

Les clés primaires sont annotées PK (primary key) et les clés étrangères FK (foreign key).

Représentation d’un objet de la table “message” :
Création d’une API

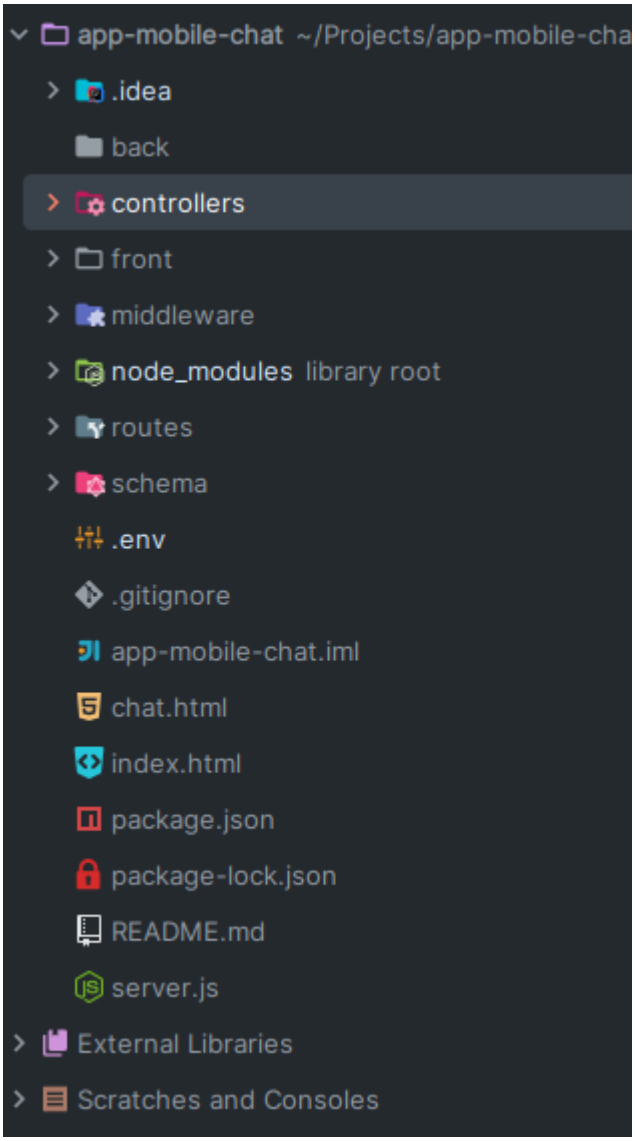
Conception et développement back-end de l’application (API)

Pour faire le lien entre notre front et notre base de données nous avons décidé de concevoir puis développer une API (Application Programming Interface).

Développer une API a part permet d'extraire et séparer la logique du front. De ce fait le front, au moyen de requêtes API, peut envoyer et recevoir des données déjà formatées.

Le back-end étant séparé du front celà améliore la clarté et l'accessibilité aux données.

Arborescence :



Le back-end est composé des dossier suivants :

- Les controllers : ils gèrent la logique et ainsi ce que doivent retourner les routes.

- Les routes : toutes les routes qui pourront être interrogées par le

front (CRUD).

- Middleware : c'est un composant intermédiaire dans notre requête, il permet d'effectuer des vérifications liés à une requête (vérification de l'intégrité des Tokens par exemple).

- Schemas : contient les fichiers qui permettent de générer et configurer les collections en base de données.

Fonctionnement de l'API :

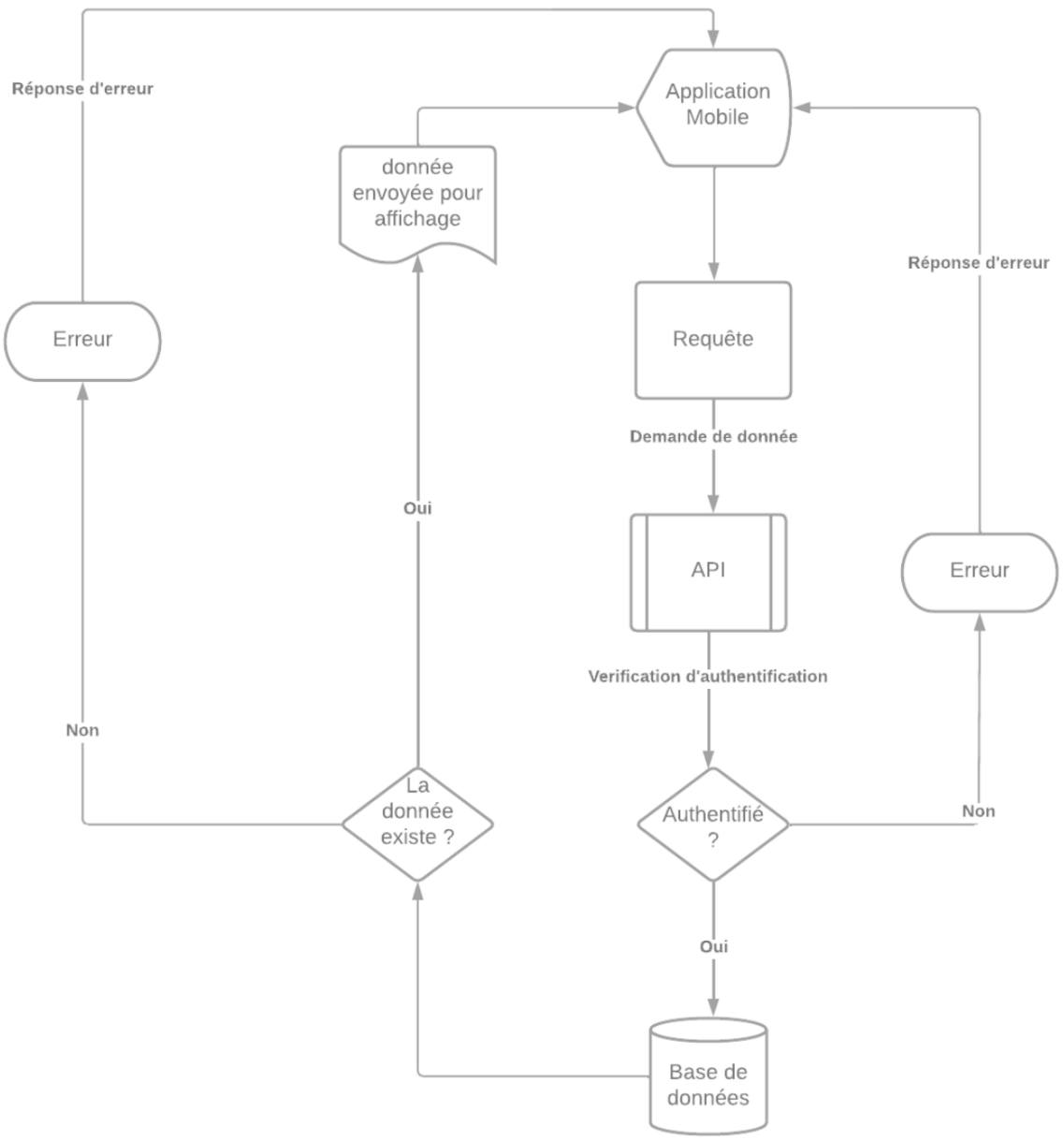
Le côté client interroge une route API au moyen d'une requête API qui peut peut avoir différents types selon ses besoins :

- GET : permet de recevoir de la donnée.
- POST: permet d'envoyer des données.
- DELETE : permet de supprimer des données.
- PUT : update total d'une entité.
- PATCH : update partiel d'une entité.

Chaque route est définie par une URI (extension de l'URL) qui permet d'interroger le controller qui y est associé. Cette fonction callback controller interagit avec la base de données et retourne un objet JSON et le code statut adéquat.

Avant d'interroger une fonction controller un middleware peut être interrogé comme composant intermédiaire, c'est souvent le cas pour vérifier si un utilisateur est bien authentifié.

Schéma de l'API :



Notre API est une **API REST**, c'est-à-dire qu'elle se plie à des standards. Le format en entrée et sortie de endpoint est uniformisé ainsi qu'un format d'erreur standardisé avec un code de retour http correspondant.

La documentation de l'API est donc facilement accessible et lisible pour les utilisateurs car ce sont toujours les mêmes standards.

Code	Définition
200 : OK	*Indique que la requête a réussi. Cela signifie que la*

Code	Définition
	demande du client a été traitée avec succès par le serveur, et la réponse renvoyée contient les informations demandées.
201 : CREATED	*Indique que la requête a réussi et qu'une ressource a été créée. Ce code est généralement renvoyé après une opération de création réussie, indiquant que la ressource demandée a été créée avec succès sur le serveur.*
204 : NO CONTENT	*Indique que la requête a réussi, mais qu'il n'y a pas de contenu à renvoyer dans la réponse. Contrairement aux autres codes de statut, le code 204 est généralement utilisé lorsque le serveur a traité avec succès la demande du client, mais ne renvoie aucune donnée supplémentaire dans la réponse (ex : pour le DELETE).*
400 : BAD REQUEST	*Indique que le serveur ne peut pas comprendre la requête en raison d'une mauvaise syntaxe ou d'une structure incorrecte. Cela peut se produire si la requête est mal formée, si des paramètres sont manquants ou si les données envoyées ne sont pas valides selon les attentes du serveur.*
401 : UNAUTHORIZED	*Indique que la requête n'a pas été effectuée car des informations d'authentification sont manquantes ou invalides. Ce code est généralement renvoyé lorsque l'accès à la ressource demandée nécessite une authentification, mais les informations fournies ne sont pas valides ou sont absentes.*
403 : FORBIDDEN	*Indique que le serveur a compris la requête, mais ne l'autorise pas. Cela peut être dû à des raisons telles que des permissions insuffisantes pour accéder à la ressource demandée, une tentative d'accès à une ressource protégée sans les autorisations*

Code	Définition
	appropriées, ou une politique de sécurité en place qui interdit l'accès.
404 : NOT FOUND	*Indique que le serveur n'a pas trouvé la ressource demandée. Cela peut se produire si l'URL ou le chemin d'accès spécifié dans la requête ne correspond à aucune ressource existante sur le serveur.*
409 : CONFLICT	*Indique un conflit dans la requête. Ce code de statut est généralement utilisé lorsqu'il y a un conflit avec l'état actuel du serveur, c'est-à-dire lorsque la requête ne peut pas être traitée en raison d'un conflit avec une ressource existante. Dans le cas spécifique de l'adresse e-mail déjà utilisée, le code 409 peut être approprié pour signaler ce conflit.*
500 : INTERNAL SERVER ERROR	*Indique que le serveur a rencontré un problème interne lors du traitement de la requête. Ce code est utilisé lorsque le serveur rencontre une situation inattendue qui l'empêche de répondre correctement à la demande du client. Il peut s'agir d'erreurs de configuration, de bogues logiciels ou d'autres problèmes internes du serveur.*

Fonctionnement des Controllers :

Nous avons donc un fichier JS controller par collection en base de données (et un controller qui gère les erreurs) :

- chatRoomController.js
- messageController.js
- userController.js

Dans ces fichiers composés de fonctions, nous appelons et envoyons nos données à la base de données grâce aux méthodes fournies par Mongo.

Exemple de fonction au sein du controller “chatRoomController” :

```
1 usage  lucas-verdier
const createChatRoom = async (req, res) : Promise<void>  => {
    const { name, usersInRoom } = req.body;
    try {
        if (checkRole(req.auth.role)) {
            const chatRoom : HydratedDocument  = new ChatRoom( doc: { name, usersInRoom });
            await chatRoom.save();
            res.status(201).json({ chatRoom });
        }
    } catch (err) {
        res.status(401).json({ err });
    }
};
```

```
1 usage  lucas-verdier
const checkRole = (role) : boolean  => {
    if (role !== 'admin') {
        throw new Error('Vous ne possédez pas les droits pour effectuer cette action.');
    }
    return true;
}
```

On voit donc ici une fonction asynchrone qui permet à un admin de créer une nouvelle chatRoom.
La donnée est envoyée dans le body (corps de la requête) sous format JSON.

Quand la route est interrogée, la fonction “checkRole” si l’utilisateur à le rôle “admin”. Si c’est le cas les informations du body sont “save” (méthode de mongo) sont sauvegardées en base de données. Si l’utilisateur n’est pas admin une erreur est renvoyée au front afin qu’il

puisse gérer l'erreur.

L'objet req est généré par la requête HTTP et représente la requête alors que l'objet res, également généré par la requête, représente la réponse à retourner.

Fonctionnement des routes :

Dans ce dossier sont présents les fichiers qui gèrent les routes API (URI). Pour chaque fichier controller, il y a un fichier route. Dans ce fichier on on déclare donc le type de requête HTTP (PUT, GET, POST, DELETE), l'URI qu'il faut interroger depuis la front, la fonction que l'uri prend en callback et le middleware qui doit être interrogé.

Exemple du fichier chatRoom.js issu du dossier route :

```
router.post( path: '/create', auth, createChatRoom);
router.put( path: '/insert-user', auth, addUserToChatRoom);
router.put( path: '/remove-user', auth, removeUserFromChatRoom);
router.get( path: '/rooms', getAllRooms);

module.exports = router;
```

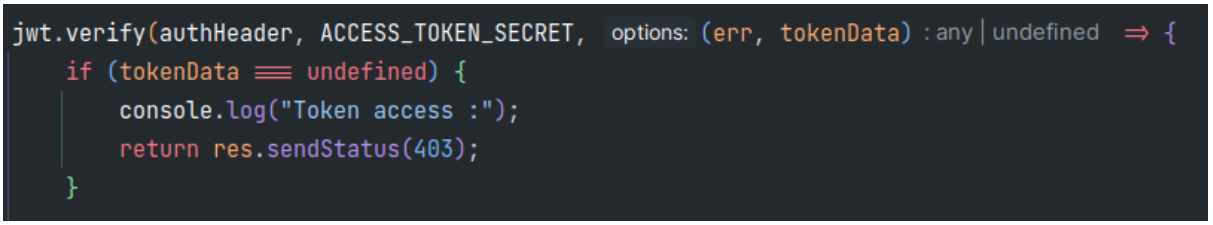
Pour reprendre l'exemple de la création d'une chatRoom (première route), on voit ici que c'est une route de type POST, qui peut être interrogée au moyen de l'URI "/create", qui passe par le middleware "auth" qui check la validité des tokens de l'utilisateur, et a pour retour la fonction createChatRoom du controller chatRoom.

Fonctionnement des middleware :

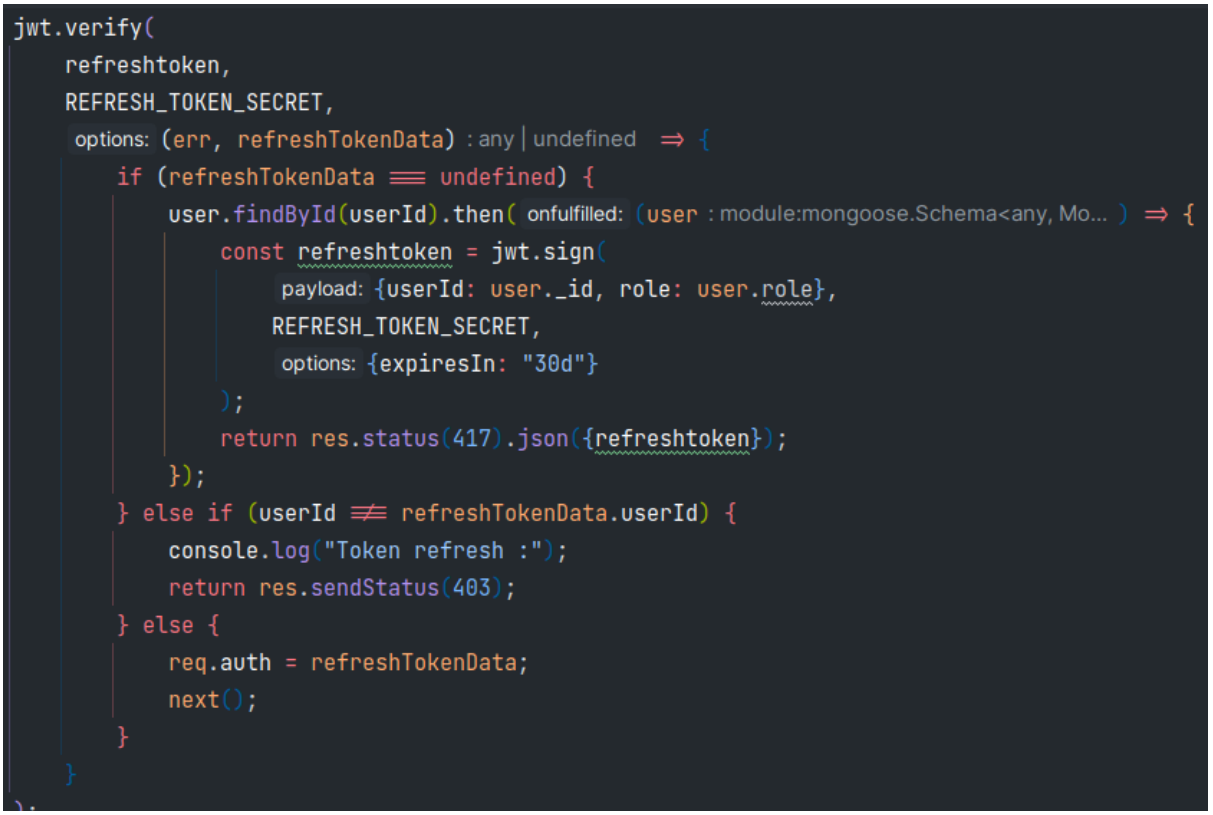
Le middleware sous ExpressJS est une fonction créée qui agit comme composant intermédiaire entre la requête et le callback. Le middleware permet d'effectuer des tâches avant que la requête passe à la fonction callback.
Par exemple pour vérifier si un utilisateur est authentifié et si ses tokens sont valides.

Ci-dessous, toujours pour l'exemple de la création d'une chatRoom, nous avons besoin de savoir si l'utilisateur est connecté (en mode admin qui plus est). Quand l'utilisteur interroge la route, la requête passe d'abord par le middleware 'auth', ce dernier vérifie l'intégrité du token d'authentification et du refresh token.

Vérification du token d'authentification :



Vérification du refresh token :



Le token JWT (JSON WEB TOKEN) d'authentification contient des

informations permettant d’identifier un utilisateur (son id, son rôle), c’est en quelque sorte la carte de visite de l’utilisateur sur l’application.

Le refresh token, lui, est régénéré tous les N temps. Il permet tout ces temps de vérifier l’intégrité de l’utilisateur et ses informations. Celà permet d’ajouter une couche de sécurité, car si jamais un utilisateur est corrompu et modifie son rôle par exemple, à l’expiration du refresh token, les informations seront vérifiées. Si elles ne correspondent pas, l’utilisateur ne sera plus authentifié.