



# Dossier de Projet :

Simple Chat, application mobile de messagerie en temps réel

---

Projet réalisé dans le cadre de la présentation au Titre de Concepteur Développeur d'Application

---

Présenté par

Rémi LOPEZ

Entreprise

Digimood

Tuteur

Anthony Gross

Organisme de formation

La Plateforme\_

# SOMMAIRE

---

## I. Compétences du référentiel couvertes par le projet

Activités types .....	p. 4
-----------------------	------

---

## II. Résumé du projet

Présentation du projet en anglais .....	p. 5
---	------

---

## III. Cahier des Charges

Analyse de l'existant .....	p. 6
Besoins .....	p. 6
Spécifications fonctionnelles	
Application mobile .....	p. 7
Application desktop .....	p. 8
Contexte technique .....	p. 8

---

## IV. Conception du projet

Gestion de projet .....	p. 9
Spécifications techniques .....	p. 10
Architecture logicielle .....	p. 11

---

## V. Conception de l'API

Conception de la base de données .....	p. 12
MCD et MLD .....	p. 14
Mise en place de la base de données .....	p. 15

---

## VI. Développement de l'API

Arborescence et fonctionnement de l'API .....	p. 18
Routeur .....	p. 19
Middleware .....	p. 20

Service .....	p. 21
Modeles .....	p. 24
Socket .....	p. 24
Sécurité .....	p. 25
Test unitaires .....	p. 26
Recherche anglophone .....	p. 28

---

## VII. Conception de l'application mobile

Charte graphique .....	p. 29
Maquette .....	p. 29
User story .....	p. 29

---

## VIII. Développement de l'application mobile

Arborescence .....	p. 30
Fonctionnement et sécurité .....	p. 31
Requêtes vers l'API .....	p. 33
Navigators .....	p. 34
Screens .....	p. 35
Composants .....	p. 37
Sockets .....	p. 38
Recherche anglophone .....	p. 38

---

## IX. Développement de l'espace administrateur

Fonctionnalités .....	p. 40
Arborescence .....	p. 40
Fonctionnement et sécurité .....	p. 41

---

## X. Annexe

Maquettes .....	p. 42
-----------------	-------

# I. COMPÉTENCES DU RÉFÉRENTIEL

Au cours de la formation de Concepteur développeur d'Application, nous avons abordé trois activités types qui sont représentées par des compétences :

---

## ACTIVITÉ TYPE N°1

Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité :

- ▶ Maquetter une application
- ▶ Développer une interface utilisateur de type desktop
- ▶ Développer des composants d'accès aux données
- ▶ Développer la partie front-end d'une interface utilisateur
- ▶ Développer la partie back-end d'une interface utilisateur

---

## ACTIVITÉ TYPE N°2

Concevoir et développer la persistance des données en intégrant les recommandations de sécurité :

- ▶ Concevoir une base de données
- ▶ Mettre en place une base de données
- ▶ Développer les composants dans le langage d'une base de données

---

## ACTIVITÉ TYPE N°3

Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité :

- ▶ Collaborer à la gestion des 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

## II. RÉSUMÉ DU PROJET

Dans le cadre de la formation de Concepteur Développeur d'Application, nous devons réaliser une application mobile connecté à une API, permettant d'effectuer des opérations de type CRUD en base de données.

En équipe, nous avons choisit de travailler sur une application de messagerie en temps réel, permettant à des utilisateurs, une fois inscrits et connectés, de dialoguer dans des groupes publiques et/ou privés ainsi que dans des conversations privés avec un seul utilisateur.

L'utilisateur peut ainsi créer un groupe ou une conversation privé, modifier les informations du groupe ou le supprimer (uniquement s'il en est l'auteur) puis, il peut envoyer un message, le modifier et le supprimer (uniquement s'il en est l'auteur).

Si un ou des utilisateur(s) sont présents dans la fenêtre de conversation de groupe ou privée, le contenu se met à jour automatiquement à l'envoi d'un message par l'expéditeur.

L'enjeu de ce projet n'est pas de révolutionner les applications déjà existantes comme WhatsApp ou Discord, mais de proposer une solution réalisée en peu de temps, fonctionnelle et qui remplit notre cahier des charges, à savoir :

- Messagerie en temps réel via l'utilisation de sockets
- Aucune données n'est conservée lors d'une suppression

Pour des raisons évidentes de sécurité et de modération, nous avons également développé une solution web permettant à ou aux administrateur(s) de gérer rapidement la liste des utilisateurs, des messages échangés au sein d'un groupe ainsi que le groupe lui-même.

Dans une version future de l'application, on pourrait mettre en place un système de notification lorsqu'un utilisateur reçoit un message, ou lorsqu'un utilisateur se connecte pour ajouter du dynamisme et se rapprocher un maximum des meilleures applications disponibles actuellement sur le marché.

# III. CAHIER DES CHARGES

---

## ANALYSE DE L'EXISTANT

Il existe un grand nombre d'applications mobiles sous forme de messagerie en temps réel telles que Google Chat, WhatsApp ou encore Discord. Elles disposent toutes de fonctionnalités plus ou moins similaires comme la création de groupes de discussion ou encore de conversations privées.

Nous avons axé notre projet sur une application qui reprend les fonctionnalités décrites précédemment, mais dans un contexte où les données d'un utilisateur persistent facilement sur internet, nous avons décidé que lorsque l'utilisateur supprime un message, une conversation, un groupe, aucune donnée n'est conservée en base de données.

Ce projet ne se veut pas révolutionnaire, mais on souhaite s'inscrire dans une démarche de transparence et de confiance avec l'utilisateur.

---

## EXPRESSION DES BESOINS

Comme nous souhaitons développer une application mobile de messagerie en temps réel, le besoin principal est la mise à jour en temps réel de l'affichage pour chaque utilisateur lorsqu'il interagit dans un groupe ou une conversation privée.

Le besoin secondaire de notre application est de ne conserver aucune donnée de l'utilisateur lorsque celui-ci effectue une suppression de groupe, conversation privée et / ou messages.

Pour répondre à ces besoins, on dispose de deux interfaces, une première mobile qui concentre l'application de messagerie en temps réel, et la seconde côté web pour l'administrateur du site.

L'utilisateur moyen n'est pas spécifié, il peut communiquer avec les personnes inscrites sur Simple Chat et rejoindre des groupes de discussion.

L'administrateur du site quant à lui, dispose d'une interface personnelle lui permettant de consulter l'ensemble des utilisateurs, groupes de discussion et messages au sein de ces derniers. Il peut également effectuer une mission de modération en supprimant du contenu jugé inapproprié par exemple.

---

## SPÉCIFICATION FONCTIONNELLES - APPLICATION MOBILE

### UTILISATEURS NON AUTHENTIFIÉ

**Page d'accueil :** l'utilisateur dispose de deux boutons lui permettant d'accéder à la page de connexion ou la page d'inscription.

**Page d'inscription :** l'utilisateur saisie les informations demandées dans le formulaire puis valide ce dernier. Si tout est correct, l'utilisateur est redirigé vers la page de connexion.

**Page de connexion :** l'utilisateur saisie son adresse email et son mot de passe. Si tout est correct, il est alors redirigé vers son espace personnel regroupant ses discussions.

### UTILISATEURS AUTHENTIFIÉ

**Page de groupes :** contient la liste des groupes publique et la liste des groupes privés et publique auquel appartient l'utilisateur. Il dispose d'un bouton lui permettant de créer un nouveau groupe

**Page de création de groupe :** contient un formulaire où l'utilisateur est invité à saisir le nom et la confidentialité (privé, publique) du groupe. Une fois validé, il est redirigé vers une page de gestion de membres au sein du groupe

**Page de gestion des membres au sein d'un groupe :** contient une liste des utilisateurs ajoutés au groupe par son créateur ainsi qu'une liste des utilisateurs non présents dans le groupe. Le créateur peut ainsi aisément gérer les membres et cette page reste accessible depuis les paramètres du groupe.

**Page de paramètre d'un groupe :** contient un formulaire pour éditer le nom et la confidentialité du groupe, un lien vers la page de gestion des membres et un bouton pour supprimer définitivement le groupe en question

**Page de discussion d'un groupe :** contient les messages échangés par les utilisateurs, la possibilité de rédiger, éditer et supprimer un message si l'utilisateur en est l'auteur.

**Page de conversations privées :** contient la liste des utilisateurs ainsi que la liste des conversations privées créées par l'utilisateur lui-même ou la personne souhaitant communiquer exclusivement avec lui.

**Page de discussion d'une conversation privée :** contient les messages échangés par les utilisateurs, la possibilité de rédiger, éditer et supprimer un message si l'utilisateur en est l'auteur.

**Page de paramètre d'une conversation privée :** contient un formulaire permettant de bloquer ou débloquer l'autre utilisateur ainsi qu'un bouton pour supprimer définitivement la conversation

**Page de profil utilisateur :** contient certaines informations de l'utilisateur, un formulaire pour éditer ses informations personnelles ainsi qu'un bouton de déconnexion

---

## SPÉCIFICATION FONCTIONNELLES - APPLICATION WEB

### ADMINISTRATEUR NON AUTHENTIFIÉ

**Page de connexion :** l'administrateur saisie son adresse email et son mot de passe. Si tout est correct, il est alors redirigé vers son espace personnel.

### ADMINISTRATEUR AUTHENTIFIÉ

**Page de groupes :** contient la liste des groupes publique et privé, il peut consulter ou supprimer les groupes de façon individuelle.

**Page de discussion d'un groupe :** contient la liste des messages échangés au sein d'un groupe, il peut consulter et supprimer individuellement ces derniers.

**Page d'utilisateurs :** contient une liste d'utilisateurs, il peut consulter les informations de ces derniers et les supprimer de façon individuelle.

---

## CONTEXTE TECHNIQUE

Concernant l'application mobile, elle devra être accessible sur tous les systèmes d'exploitation Android et IOS. Elle devra également être connecté à un serveur en temps réel pour répondre à ce besoin essentiel de notre application.

Pour l'application web, elle devra être accessible sur tous type de navigateur. Elle devra également être responsive pour être facilement consultée depuis un smartphone ou une tablette.



# IV. CONCEPTION DU PROJET

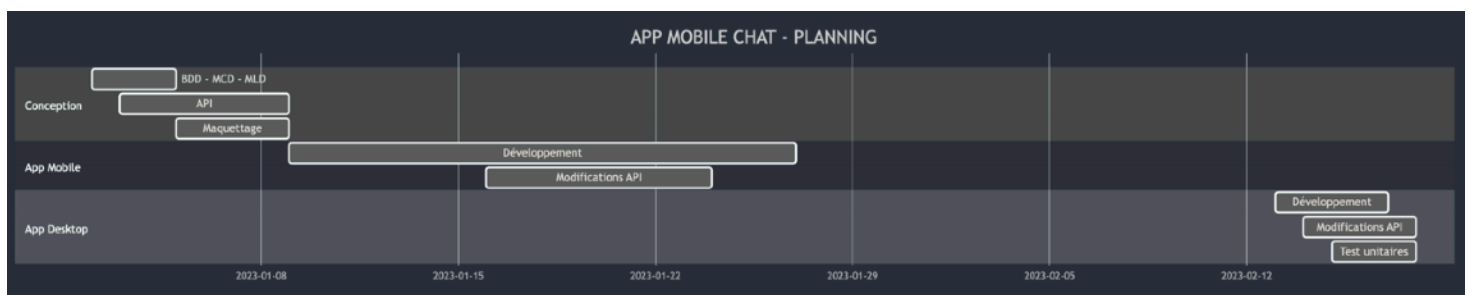
## GESTION DE PROJET

Afin de répondre au mieux aux attentes que nous avons autour de notre projet, nous avons décidé de gérer la réalisation de l'application de chat en temps réel de façon **agile**, et en plusieurs phases.

Tout d'abord, nous avons réfléchi aux besoins et la viabilité de notre application. Une fois nos discours accordés, nous sommes passé à la phase suivante et nous avons commencé la planification de nos objectifs.

Pour cela, nous avons décidé de répartir nos tâches en **sprint** d'une semaine, permettant de rapidement s'adapter en cas d'objectifs trop complexes, peu réalisables. Pour cela, nous faisons un point en début de semaine suivante pour observer notre avancement dans ce projet.

Afin de visualiser plus facilement nos tâches, nous avons modélisé un **diagramme de Gantt** pour représenter nos tâches principales qui ont par la suite, été divisées en petites tâches.



Nous sommes ensuite passé à la phase d'exécution durant laquelle nous avons utilisé divers **logiciels de collaboration** d'équipe tels que **Trello** pour que chacun s'entende sur les tâches à réaliser et leurs échéances, ou encore **Google Chat** pour communiquer efficacement et rapidement. De plus, ce logiciel de messagerie permet de s'échanger tout type de documents et de texte. Une véritable aubaine pour communiquer rapidement des bouts de code ou autre.

Lorsque toutes les tâches quotidiennes étaient rédigées et attribuées, nous passons en phase de suivi et de contrôle pour surveiller en temps réel la faisabilité du projet selon la durée allouée.

Pour cela, nous disposons de divers colonnes sur Trello :

- **A faire** : tâches non attribuées
- **En cours** : tâches attribuées et en cours de réalisation
- **En test** : tâches attribuées, réalisées et en cours de tests
- **Bloqué** : tâches attribuées mais existence d'un point bloquant
- **Terminé** : tâches attribuées, réalisées et testées

Nous nous assurons qu'aucun effet de bord ne viendrait entraver notre travail en effectuant des **Pull Request** sur le Repository Github, en demandant à un membre de l'équipe d'effectuer une revue de code.

---

## SPÉCIFICATIONS TECHNIQUES

Concernant les caractéristiques technique de notre projet, nous choisis d'utiliser pleinement la langage javascript. En effet, il convient parfaitement pour développer une API mais également une interface client.

Pour l'API :

- **Node.js** : permet la création d'une application javascript côté serveur.
- **Express** : framework rapide et léger qui permet la création d'une API.
- **Sequelize** : ORM destiné à l'utilisation via Node.js pour la configuration, la connexion et les requêtes en base de données.
- **JSON Web Token** : package permettant de générer et vérifier des tokens.
- **socket.io** : framework rapide et léger permettant la mise en place du temps réel (côté serveur)

Pour l'application mobile :

- **React Native** : framework permettant le développement d'application mobile hybride (IOS, Android).
- **React Navigation** : compatible avec React Native et Expo, permet de configurer le routage et la navigation au sein d'une application mobile.
- **Expo** : permet l'émulation de l'application mobile, très utilisé lors de la phase de développement.
- **socket.io client** : framework rapide et léger permettant la mise en place du temps réel (côté client)

Pour l'application desktop :

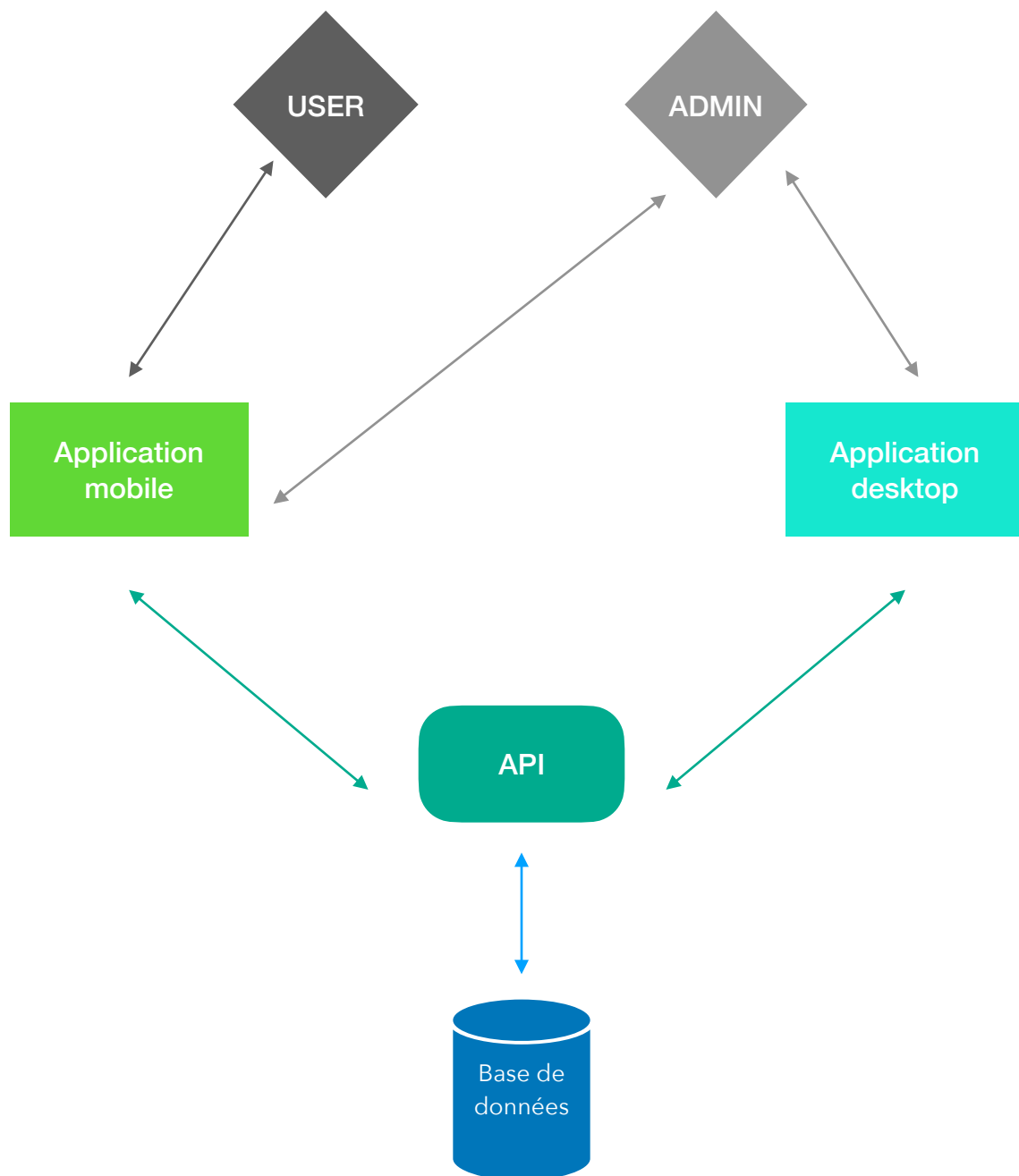
- **React.js** : bibliothèque javascript très similaire à React Native mais plus adapté pour le développement d'application web.

---

## ARCHITECTURE LOGICIELLE

L'API étant commune à l'application mobile et à l'application web, seul l'utilisateur possédant le droit ADMIN en base de données pourra accéder à l'application web.

L'application mobile est quant à elle accessible aux utilisateurs possédant le droit USER ainsi qu'aux utilisateurs possédant le droit ADMIN. Cependant, un administrateur ne peut effectuer les opérations possibles depuis l'interface web via l'application mobile. Il dispose ainsi des mêmes droits que les utilisateurs de type USER.



# V. CONCEPTION DE L'API

---

## CONCEPTION DE LA BASE DE DONNÉES

Avant de développer une API et/ou le back-end d'une application, il est important de choisir un type de base de données. Il existe des base de données NoSQL ou relationnelles.

Pour les besoins de ce projet, nous avons décidé d'utiliser une base de données relationnelle afin d'effectuer des opérations en base de données de façon efficace. Ensuite, nous avons définies les entités que nous souhaitons stocker en base de données ainsi que les champs qui les composent.

Pour qu'un utilisateur puisse s'inscrire et se connecter, nous avons défini l'entité **User** qui contient les champs suivants :

- Username
- Firstname
- Lastname
- Email (unique)
- Password
- Roles (User ou Admin)
- Created\_at
- Updated\_at

Une fois l'utilisateur inscrit et connecté, nous avons défini l'entité **Channel**, pour symboliser les groupes de discussion, qui contient les champs suivants :

- Name
- Creator (ID de l'utilisateur)
- Private
- Created\_at
- Updated\_at

L'entité Channel est reliée à l'entité User par une table de liaison nommée **UserChannel**, qui contient les champs suivants :

- User\_id
- Channel\_id
- Created\_at
- Updated\_at

Afin qu'un message envoyé au sein d'un groupe de discussion soit relié à un groupe, nous avons défini l'entité **ChannelMessage**, qui contient les champs suivants :

- Message
- Created\_at
- Updated\_at
- User\_id (expéditeur du message)
- Channel\_id

L'utilisateur disposant également de la possibilité de créer des conversations privées, nous avons défini l'entité **ConversationMessage**, qui comme l'entité précédente, contient les champs suivants :

- Message
- Created\_at
- Updated\_at
- User\_id (expéditeur du message)
- User\_id (destinataire du message)
- Conversation\_id

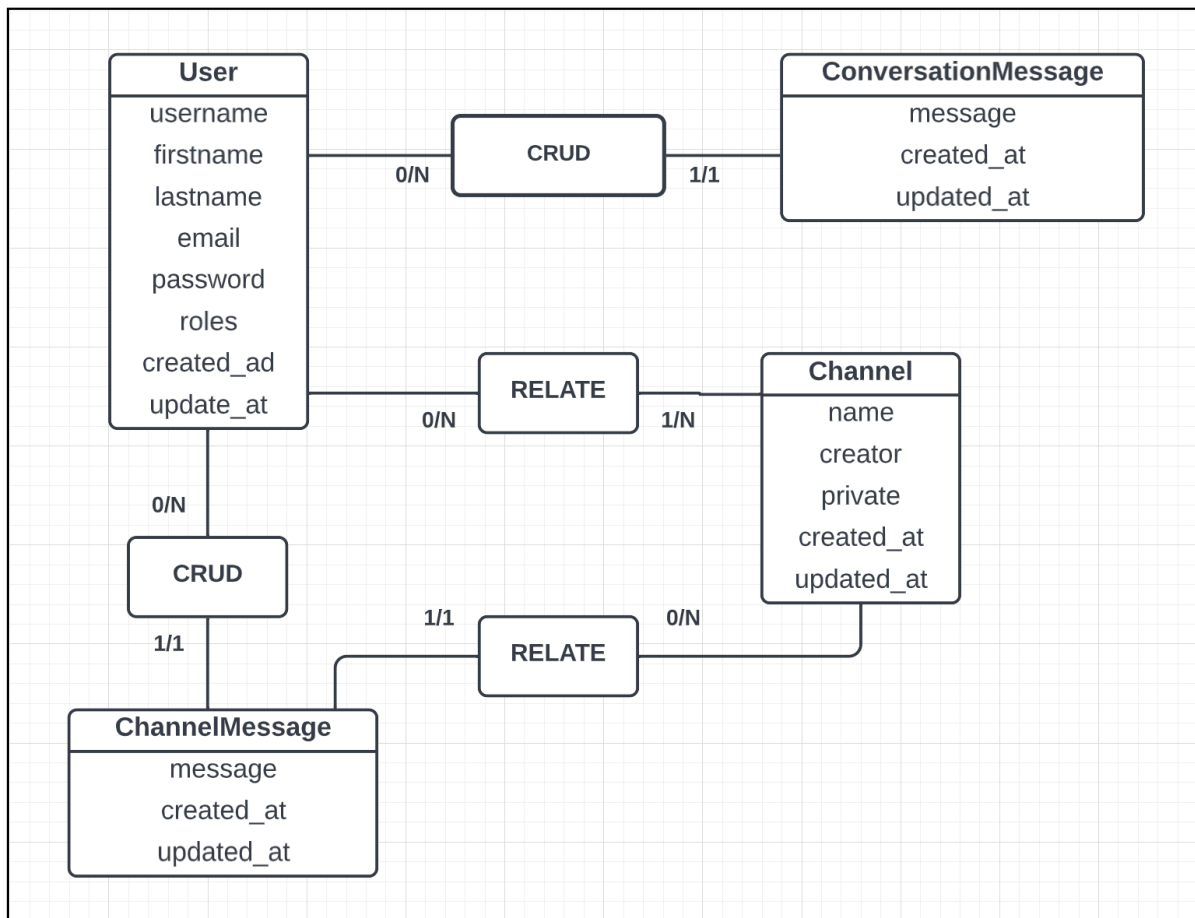
Enfin, pour retrouver aisément une conversation et les messages au sein de celle-ci, nous avons défini une table de liaison entre l'utilisateur et les messages, nommée **UserConversation**. Cette entité contient les champs suivants :

- Blocked
- Created\_at
- Updated\_at
- User\_id (expéditeur du 1er message)
- User\_id (destinataire du 1er message)

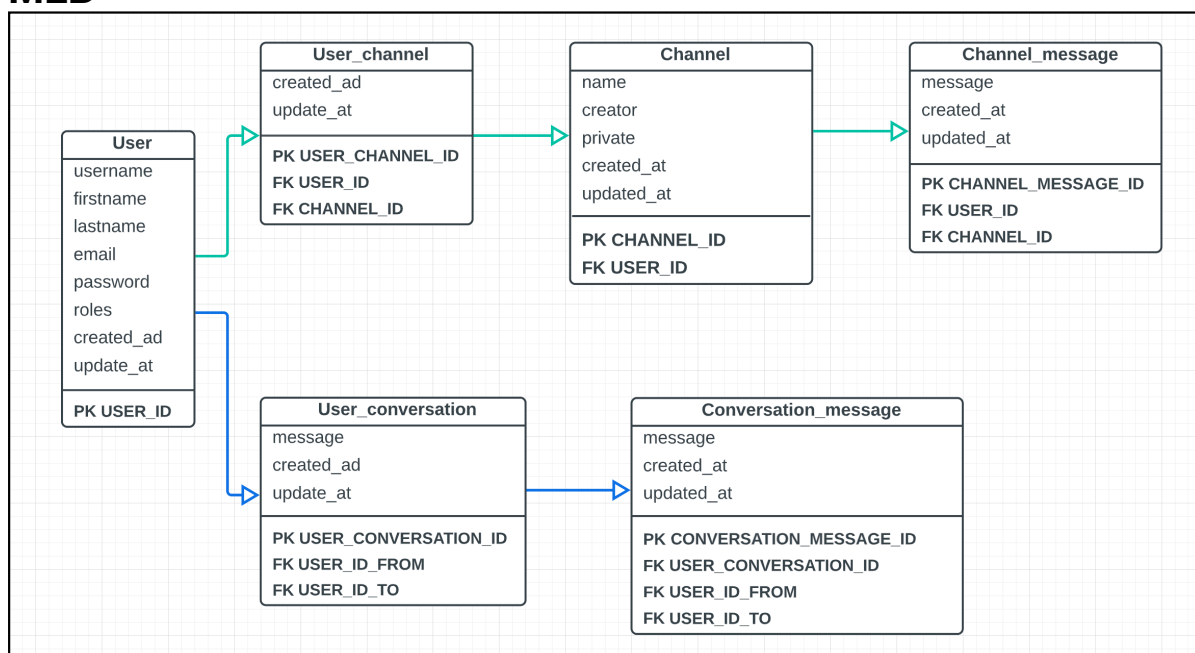
Chaque entité contient une clé primaire symbolisée par un ID auto-incrémenté qui permet de lier certaines tables entre elles.

Une fois les entités et les champs associés définis, nous avons réalisé un Modèle Conceptuel de Données ainsi qu'un Modèle Logique de Données pour symboliser les relations entre les différentes tables.

## MCD



## MLD



---

## MISE EN PLACE DE LA BASE DE DONNÉES

J'ai choisit d'utiliser Sequelize, un puissant ORM pour Node.js, compatible avec différent moteurs de base de données comme MySQL, ce qui correspond parfaitement à nos attentes définies lors de la phase de conception.

La première étape de la mise en place consistait à créer un fichier config.js contenant un module exportable, qui contient les informations nécessaires pour effectuer la connexion :

```
module.exports = {
  HOST: "localhost",
  USER: "root",
  PASSWORD: "root",
  DB: "chat_api",
  dialect: "mysql",
  pool: {
    max: 5,
    min: 0,
    acquire: 30000,
    idle: 10000,
  }
};
```

On retrouve ci-dessus les paramètres incontournables tels que le nom de la base de données, l'utilisateur et son mot de passe ou encore l'hôte. De plus, on retrouve la spécification 'dialect' qui correspond au langage utilisé pour notre application, le langage SQL.

Pour des raisons de sécurité, ce module est déclaré indépendamment et utilisé lors de la configuration via une instance de Sequelize dans un fichier index.js présent dans le dossier models. On note également que le port est définit dans un fichier .env.

```
const sequelize = new Sequelize(
  dbConfig.DB,
  dbConfig.USER,
  dbConfig.PASSWORD,
  {
    host: dbConfig.HOST,
    dialect: dbConfig.dialect,
    operatorsAliases: 0,
    port: process.env.MYSQL_PORT,

    pool: {
      max: dbConfig.pool.max,
      min: dbConfig.pool.min,
      acquire: dbConfig.pool.acquire,
      idle: dbConfig.pool.idle
    }
  }
);
```

Une fois la configuration de la base de données effective, nous avons pu nous atteler à la déclaration des entités et des champs qui les composent.

Dans chaque fichier modèle, on crée un module que l'on exporte et qui prend Sequelize en paramètre. On peut ensuite définir le type, le nom du champ, une valeur par défaut, etc.

Une fois toutes les entités créées, on peut implémenter les relations / associations en utilisant par exemple, pour des Foreign Keys, la méthode `belongsTo`.

Voici un exemple d'entité, avec `UserChannel`, qui possède deux clés étrangères déclarées via la méthode `belongsTo` :

```
module.exports = (sequelize, Sequelize) => {
  User = require('./user.model')(sequelize, Sequelize);
  Channel = require('./channel.model')(sequelize, Sequelize);

  const UserChannel = sequelize.define('UserChannel', {
    id: {
      type: Sequelize.INTEGER,
      autoIncrement: true,
      primaryKey: true
    },
    createdAt: {
      field: 'created_at',
      type: "TIMESTAMP",
      defaultValue: sequelize.literal("CURRENT_TIMESTAMP"),
      allowNull: false,
    },
    updatedAt: {
      field: 'updated_at',
      type: "TIMESTAMP",
      defaultValue: sequelize.literal("CURRENT_TIMESTAMP"),
      allowNull: true,
    },
  }, {
    tableName: 'user_channel'
  });

  UserChannel.belongsTo(User, {
    foreignKey: 'user_id',
  });
  UserChannel.belongsTo(Channel, {
    foreignKey: 'channel_id',
  });

  return UserChannel;
};
```

Chacune des entités est ensuite appelée à la suite de l'instance de `sequelize` dans le fichier `index.js`, présent dans le dossier `models`.

Ces entités pourront ensuite être appelées dans les services pour effectuer diverses opérations en base de données, selon la requête.



```

const db = {};

db.Sequelize = Sequelize;
db.sequelize = sequelize;

db.user = require("./user.model.js")(sequelize, Sequelize);
db.channel = require("./channel.model.js")(sequelize, Sequelize);
db.userChannel = require("./userChannel.model.js")(sequelize, Sequelize);
db.channelMessage = require("./channelMessage.model.js")(sequelize, Sequelize);
db.userConversation = require("./userConversation.model.js")(sequelize, Sequelize);
db.conversationMessage = require("./conversationMessage.model.js")(sequelize,
Sequelize);

module.exports = db;

```

Enfin, la connexion à la base de données s'effectue depuis le fichier app.js situé à la racine, dans lequel on appelle la configuration de la base de données. J'ai pour cela utilisé la méthode `sync()` qui est une promesse. Elle peut retourner une erreur dans le cas où la promesse initiale est rejetée.

```

const db = require("./model")

db.sequelize.sync()
  .then(() => {
    console.log('Synced db.')
  })
  .catch((err) => {
    console.log('Error : ' + err.message)
  });

```

# VI. DÉVELOPPEMENT DE L'API

## ARBORESCENCE ET FONCTIONNEMENT

Le back-end étant consommé par l'application mobile et l'application web, nous avons décidé de développer une API commune aux deux applications. L'avantage de cette méthodologie est principalement que la base de données et les services sont communs. En effet, cela permet par exemple de réutiliser des services pour un utilisateur sur l'application mobile et pour l'administrateur, en contrôlant leur rôles respectifs, leurs ID, etc

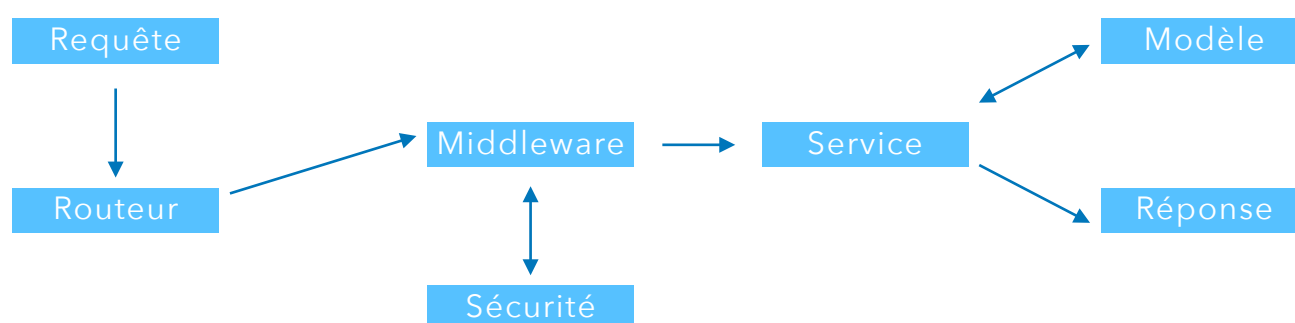
Pour répondre à la problématique de sécurité, nous avons utilisé une architecture N-tier qui permet de séparer la logique du code, les composants métiers ou encore l'accès aux données.

L'API est divisée en différentes couches qui sont les suivantes :

- Les modèles : un par entité
- Les services : un par entité (accès aux données)
- Les routeurs : un par fichier service (endpoints)
- Les middlewares : assure la sécurité (JWT, contrôle du rôle, ...)
- Les sockets : un fichier regroupant les requêtes en temps réel
- La sécurité : génération et vérification des JWT (access et refresh)
- Les tests : un par entités pour contrôler chacun des endpoints.

Au niveau du fonctionnement de l'API, lorsque le client envoie une requête côté serveur, selon l'entité appelée / nécessaire, des contrôles sont effectués au niveau des services, pour s'assurer que l'utilisateur possède le rôle demandé et que son ID correspond à l'ID envoyé en paramètre dans la requête.

Une réponse est alors renvoyée à l'utilisateur au format JSON, contenant un code statut et des données, pour être ensuite exploitée côté client.



Les différentes méthodes HTTP utilisés au sein de ce projet sont :

- GET : pour la récupération des données sous format LIST ou RETRIEVE
- POST : pour la création de données
- PUT : pour la modification de données existantes
- DELETE : pour la suppression de données existantes

Les différents statuts utilisés dans ce projet sont :

- 200 : OK  
Indique que la requête s'est bien déroulée et retourne les données demandées.
- 201 : CREATED  
Indique que la requête s'est bien déroulée et que la ressource a été créée.
- 401 : UNAUTHORIZED  
Indique que la requête n'a pas été effectuée car le token n'est pas fourni ou est incorrect.
- 404 : NOT FOUND  
Indique que le serveur n'a pas trouvé la ressource demandée
- 500 : INTERNAL SERVER ERROR  
Indique que le serveur a rencontré un problème.

---

## ROUTEUR

Le dossier router comporte un fichier par entité. Ainsi, chaque fichier répertorie une liste d'endpoints qui font le liens entre les services et les requêtes effectuées côté client.

Un endpoint peut-être accessible publiquement ou nécessiter un contrôle via un middleware. Par exemple, pour l'entité Channel :

```
/* PUBLIC : CHANNELS LIST */
router.get(
  '/api/channels',
  async (req, res) => {
    channel.getAllChannel(req, res)
  }
);

/* PRIVATE : CREATE A CHANNEL */
router.post(
  '/api/user/:id/channel',
  authMiddleware,
  async (req, res) => {
    channel.createChannel(req, res)
  }
);
```

---

## MIDDLEWARE

Pour reprendre l'exemple précédent, lorsque l'utilisateur souhaite accéder à un endpoint privé, le middleware `authMiddleware` s'assure que l'utilisateur envoie un token dans les headers puis effectue une vérification selon une clé spécifique.

Si le token est présent dans les headers et correspond à l'utilisateur après vérification, il est alors accessible depuis les services via la variable `tokenData`, stockée dans le `req.body`. Autrement, une erreur de type 401 ou 404 est retournée côté client.

```
const authMiddleware = (req, res, next) => {
  const auth = req.headers.authorization;

  if (auth && auth.startsWith('Bearer ')) {
    const token = auth.slice(7);

    try {
      const tokenData = verifyToken(token);
      req.body.tokenData = tokenData;
      next();
    } catch (error) {
      res.status(401).send({
        status: 'Error',
        message: 'You need to check your credentials'
      });
    }
  } else {
    res.status(404).send({
      status: 'Error',
      message: 'No credentials'
    });
  }
};
```

Ensuite, dans un service, on appelle le middleware `securityMiddleware` qui va contrôler que l'utilisateur faisant l'appel auprès de l'API possède bien le rôle demandé, et que son ID soit identique à celui présent en paramètre de l'URL.

```
async function securityMiddleware(token, id) {
  const userToken = token;

  const user = await User.findOne({
    attributes: ['id', 'username', 'roles'],
    where: {
      id: userToken.id
    }
  })
  const userRole = user.roles

  if ((userRole === 'USER') && userToken.id === id) {
    return 1;
  } else if (userRole === 'ADMIN') {
    return 2;
  } else {
    return 0;
  }
}
```

Par exemple, dans le service permettant la création d'un groupe de discussion, le `securityMiddleware` retourne une valeur selon le rôle et l'ID de l'utilisateur. Cela permet de contrôler l'accès aux données pour un utilisateur uniquement, un admin uniquement ou les deux.

```
async function createChannel(req, res){
  const id = req.params.id;

  const userToken = req.body.tokenData;
  const userControl = await securityMiddleware(userToken, id)

  if((userControl === 1) || (userControl === 2)) {
    // ...
  } else {
    res.status(500).send({
      status: 'Error',
      message: "You're not authorized to create a Channel",
    });
  }
}
```

---

## SERVICE

Les services proposés pour l'entité **USER**, à l'image d'un CRUD, permettent de :

- Récupérer une liste des utilisateurs et de certaines informations
- Créer un utilisateur (inscription)
- Obtenir un JWT (lors de la connexion)
- Récupérer les informations d'un utilisateur
- Rejoindre un groupe publique
- Modifier ses informations personnelles
- Supprimer son compte
- Obtenir un nouveau JWT

Les services proposés pour l'entité **CONVERSATION** permettent de :

- Récupérer toutes les conversations privées pour un utilisateur
- Récupérer une conversation privée
- Récupérer la valeur 'blocked' pour savoir si un utilisateur en a bloqué un autre
- Créer une conversation privée
- Modifier la valeur 'blocked'
- Supprimer une conversation privée

Les services proposés pour l'entité **CONVERSATION MESSAGE** permettent de :

- Récupérer tous les messages d'une conversation privée
- Créer un message
- Modifier un message
- Supprimer un message

Les services proposés pour l'entité **CHANNEL** permettent de :

- Récupérer tout type de groupe uniquement pour l'administrateur
- Récupérer tous les groupes publique
- Récupérer la liste des utilisateurs membre d'un groupe
- Récupérer la liste des utilisateurs non membre d'un groupe
- Récupérer la liste des groupes d'un utilisateur
- Créer un groupe
- Modifier un groupe
- Ajouter un utilisateur à son groupe
- Supprimer un utilisateur de son groupe
- Supprimer un groupe

Les services proposés pour l'entité **CHANNEL MESSAGE** permettent de :

- Récupérer tous les messages d'un groupe
- Créer un message
- Modifier un message
- Supprimer un message

Exemple de service **sans middleware** pour l'entité **Channel** :

```
async function getAllChannel(req, res){
  await Channel.findAll({
    order: [
      ['created_at', 'DESC'],
    ],
    attributes: [
      'id',
      'name',
      'creator',
      'private',
      'created_at',
    ],
    where: {
      private: 0,
    },
  })
  .then(channels => {
    res.status(200).send({
      status: 'Success',
      data: channels,
    });
  })
  .catch(err => {
    res.status(500).send({
      status: 'Error',
      message: err.message
    });
  });
}
```

Exemple de service **avec middleware** pour l'entité **Channel** :

```
async function createChannel(req, res){
  const id = req.params.id;

  const userToken = req.body.tokenData;
  const userControl = await securityMiddleware(userToken, id)

  const channel = {
    name: req.body.name,
    private: req.body.private,
    creator: userToken.id,
  };

  if((userControl === 1) || (userControl === 2)) {
    Channel.create(channel)
      .then(channel => {
        const joinChannel = {
          user_id: channel.creator,
          channel_id: channel.id,
        };

        UserChannel.create(joinChannel)

        res.status(201).send({
          status: 'Success',
          data: channel,
        });
      })
      .catch(err => {
        res.status(500).send({
          status: 'Error',
          message: err.message
        });
      });
  } else {
    res.status(500).send({
      status: 'Error',
      message: "You're not autorized to create a Channel",
    });
  }
}
```

Comme on peut le voir ci-dessus, il y a une uniformisation au niveau du paramètre status qui sera soit 'Success', soit 'Error' ainsi que du paramètre data pour une simplicité d'exploitation des données côté client.

Seul les requête retournant une erreur 500 renvoient un paramètre message, qui permet également côté client, d'effectuer un contrôle assez rapide et efficace sur le bon déroulement d'une requête.

---

## MODÈLE

Au niveau des modèles, nous disposons d'un fichier par entité nécessaire au bon fonctionnement de l'API. On retrouve ainsi :

- User : contient l'ensemble des champs liés à cette entité
- Channel : contient l'ensemble des champs liés à cette entité
- ChannelUser : table de liaison entre les entités User et Channel
- ChannelMessage : contient l'ensemble des champs liés à cette entité
- ConversationMessage : contient l'ensemble des champs liés à cette entité
- ConversationUser : table de liaison entre les entités User et ConversationMessage

\* cf. Conception de l'API

---

## SOCKET

Les sockets permettent l'utilisation du temps réel dans notre application de messagerie. Par exemple, lorsqu'un utilisateur A envoie un message à un utilisateur B dans une conversation privée, la liste des messages des deux utilisateurs est rafraîchie pour afficher le message envoyé.

Au sein du fichier principal, dans lequel je synchronise la connexion à la base de données, le CORS ou encore les routeurs, je déclare :

```
const { Server } = require("socket.io");
const { Socket } = require('./socket');

. . .

const server = http.createServer(app);
const io = new Server(server);

Socket(io);
```

La méthode Socket( ) est appelée et prend en paramètre io, qui correspond à une instantiation du serveur. Cette méthode contient la connexion à la base de données ainsi que divers méthodes permettant l'échange en temps réel.

```
const db = require("../models");

const Socket = (io) => {
  io.on('connection', (socket) => {
    console.log('a user connected : ' + socket.id);

    . . .

    socket.on('disconnect', () => {
      console.log('user disconnected : ' + socket.id);
      socket.removeAllListeners();
    });
  });
}
```



---

## SÉCURITÉ

Il est important de bien sécuriser son API puisqu'un utilisateur malveillant peut tenter de récupérer des informations en base de données de diverses façons :

- Injection SQL : l'utilisateur malveillant tente d'injecter du code dans le langage SQL pour accéder à des informations d'un ou plusieurs utilisateurs.
- Credential Stuffing : l'utilisateur malveillant tente de voler les identifiants de connexion d'un utilisateur
- Man-in-the-middle : l'utilisateur malveillant tente de pousser un utilisateur à se connecter à un service comprimas dans le but de récupérer ses identifiants ou token.

Afin de sécuriser l'API, j'ai mit en place divers middlewares et contrôles pour m'assurer qu'un utilisateur ne tente pas de compromettre l'intégrité de l'API.

Premièrement, j'utilise bcrypt pour hacher le mot de passe d'un utilisateur lorsque celui-ci s'inscrit à l'application de messagerie en temps réel.

Ensuite, lorsque l'utilisateur se connecte via ses identifiants, j'utilise le JSON Web Token qui est généré en deux exemplaires, avec deux clés secrètes différentes. Le premier est l'access token (durée de vie d'environ 5 min) et le deuxième est le refresh token (durée de vie d'environ 10 min) qui permet le rafraichissement des deux tokens lorsque le premier arrive à expiration.

Ce token est ensuite exploité dans un middleware, puisqu'une fois décodé, on accède à l'ID, le rôle et le pseudo de l'utilisateur. Il est obligatoirement envoyé en paramètre dans le headers.

---

## TEST UNITAIRE

La quasi-totalité des endpoints nécessitant un token, il est primordial de tester la sécurité de ces derniers dans le cas où un individu tente d'accéder à des requêtes sans envoyer de token valide.

Dans un dossier nommé 'test', j'ai créé un fichier de test par entité. Ensuite, j'ai installé les dépendances Jest et Supertest qui permettent la création de tests unitaires pour des API développées en Node.js.

Exemple d'écriture de tests unitaires pour l'entité USER et les services qui lui sont associés :

```
const request = require("supertest");
const {server, app} = require("../index");
const db = require('../models/index');

require("dotenv").config();

/* Connecting to the database before each test. */
beforeEach(async () => {
  await db.sequelize.sync()
    .then(() => {
      console.log('Synced db.')
    })
    .catch((err) => {
      console.log('Error : ' + err.message)
    });
});

/* Closing database connection after each test. */
afterEach(async () => {
  await server.close();
});

/**
 * Public endpoint : GET all users
 */
describe('GET /api/users', () => {
  it('Should return all users', async () => {
    const res = await request(app)
      .get('/api/users');

    expect(res.statusCode).toBe(200);
    expect(res.body.status).toBe('Success');
    expect(res.body.data.length).toBeGreaterThan(0);
  });
});

/**
 * Private endpoints : user CRUD
 */
describe('User logged in with JWT', () => {
  let token = '';
  let user_id = 0;
```

```

beforeAll(async () => {
  await request(app)
    .post('/api/register')
    .send({
      username: 'username__test',
      email: 'username@test.com',
      password: 'test',
      firstname: 'first__test',
      lastname: 'last__test',
    });
});

beforeAll(async () => {
  const res = await request(app)
    .post('/api/login')
    .send({
      email: 'username@test.com',
      password: 'test'
    });

  token = res.body.data.access_token;
  user_id = res.body.data.user_id
});

it('Should update user informations', async () => {
  const res = await request(app)
    .put(`/api/user/${user_id}`)
    .set('Authorization', `Bearer ${token}`)
    .send({
      firstname: 'update__first__test',
    });

  expect(res.statusCode).toBe(200);
  expect(res.body.status).toBe('Success');
  expect(res.body.data.updated.firstname).toBe('update__first__test');
});

it('Should delete user', async () => {
  const res = await request(app)
    .delete(`/api/user/${user_id}`)
    .set('Authorization', `Bearer ${token}`)

  expect(res.statusCode).toBe(200);
  expect(res.body.status).toBe('Success');
});
});

```

Pour expliquer brièvement l'écriture des tests unitaires, qui sont répliquées pour chaque entités selon leurs endpoints, j'effectue une connexion et déconnexion à la base de données entre chaque test.

Ensuite, s'il existe des endpoints public j'effectue une requête asynchrone et je m'assure que le résultat renvoyé correspond à celui retourné lorsque l'opération s'est bien déroulé (code statut égal à 200)

En revanche, pour les endpoint privés nécessitant un token, je suis obligé de créer un utilisateur et de le connecter pour obtenir son ID et son token. Cela permet également de tester si ce endpoint fonctionne correctement.

Enfin, j'écris les tests unitaires correspondants aux endpoints à tester. Généralement, il s'agit d'une modification, d'une lecture de données ou de la suppression.

En lançant la commande `'npm run test'` dans le terminal, on observe très rapidement si chaque endpoint, et donc service, retourne la valeur attendue.

---

## RECHERCHE EN ANGLAIS

Durant le développement de l'API, j'ai rencontré diverses problématique comme par exemple pour le développement de l'API, l'implémentation du JWT ou encore le développement de tests unitaires.

Pour répondre efficacement à ces problématiques, j'ai consulté la documentation de Sequelize, d'Express, de JWT, de JEST ou encore de Socket.io.

Par exemple, pour effectuer une connexion aux sockets côté API, j'ai utilisé la recherche « [socket.io documentation](#) » et obtenu le résultat suivant :

```
npm install socket.io
```

That will install the module and add the dependency to `package.json`. Now let's edit `index.js` to add it:

```
const express = require('express');
const app = express();
const http = require('http');
const server = http.createServer(app);
const { Server } = require("socket.io");
const io = new Server(server);

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', (socket) => {
  console.log('a user connected');
});

server.listen(3000, () => {
  console.log('listening on *:3000');
});
```

Notice that I initialize a new instance of `socket.io` by passing the `server` (the HTTP server) object. Then I listen on the `connection` event for incoming sockets and log it to the console.

- Introduction
- The web framework
- Serving HTML
- Integrating Socket.IO
- Emitting events
- Broadcasting
- Homework
- Getting this example

# VII. CONCEPTION DE L'APP MOBILE

## CHARTES GRAPHIQUE

Pour la création du logo, nous avons utilisé Canva, un puissant logiciel web accessible gratuitement. Ensuite, pour la maquette et la charte graphique utilisée dans l'application mobile, nous avons utilisé les codes couleurs suivants :

#F1F1F1

#000000

#FF6347

#1C81C5

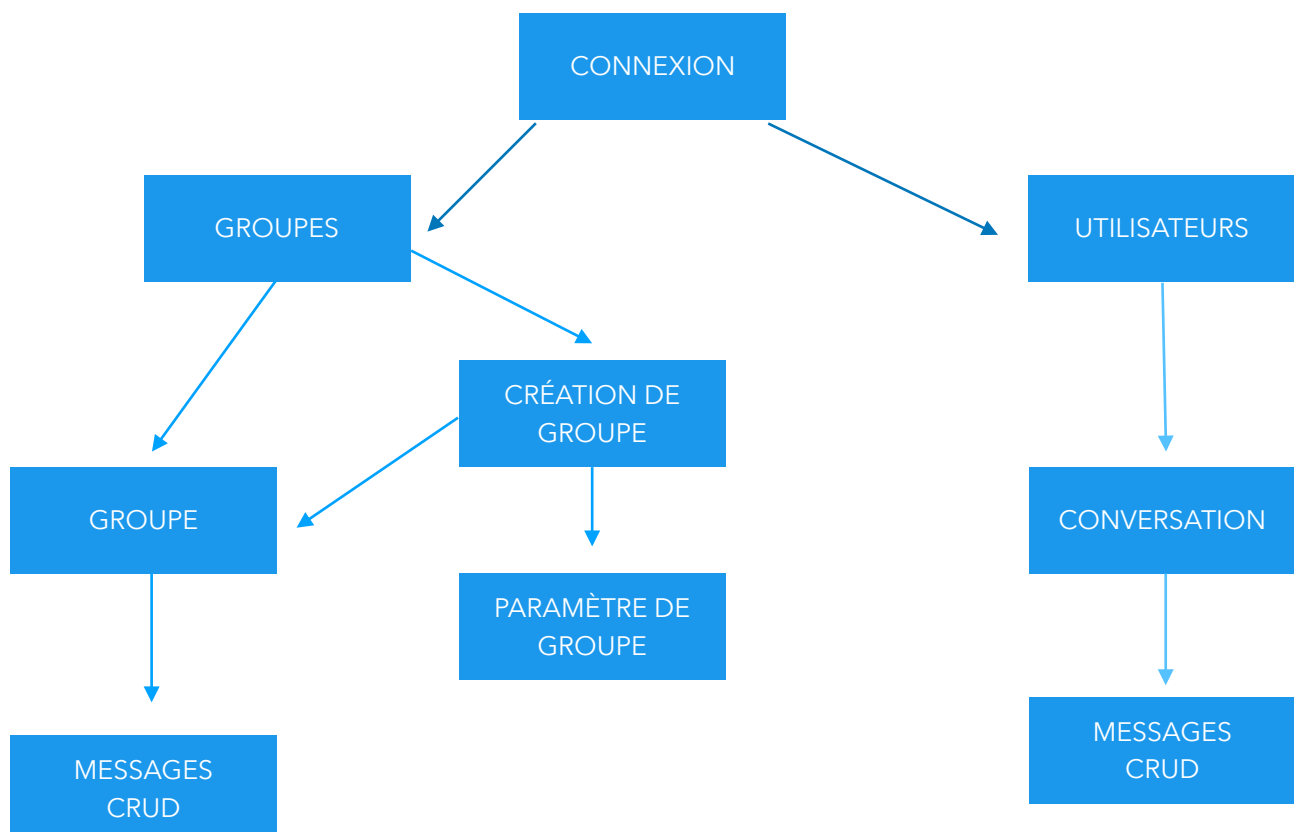
#6EC24A

## MAQUETTE

Pour la réalisation de la maquette, nous avons utilisé FIGMA, un logiciel d'UI / UX performant qui permet également de réaliser un prototype dans lequel les différentes pages créées sont connectées et simulent une application web ou mobile.

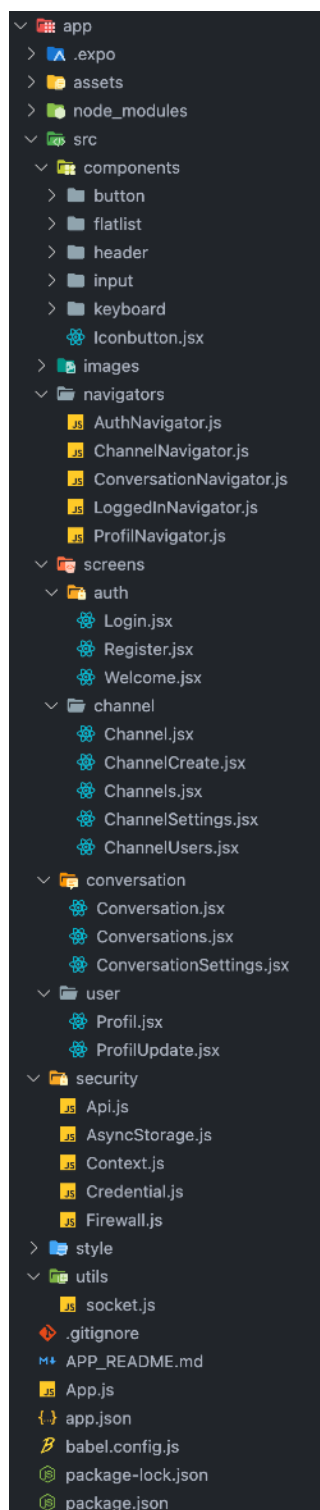
\* Toutes les maquettes de l'application mobile se trouvent en annexe.

## USER STORY



# VIII. DÉVELOPPEMENT DE L'APP MOBILE

## ARBORESCENCE



L'**architecture de l'application mobile** est présentée dans l'image ci-contre. Le coeur de l'application se situe dans le dossier **src** qui contient plusieurs couches de dossiers pour répartir la logique et faciliter la maintenance.

Le dossier **navigators** contient divers routeurs React Navigation, pour la navigation inter-entités, par exemple, AuthNavigator.js contient le screen Welcome.jsx, le screen Login.jsx et le screen Register.jsx.

Le dossier **screens** contient les écrans nécessaires à l'application, pour répondre aux fonctionnalités exprimées dans le cahier des charges. On retrouve un dossier par entité, ainsi que des fichiers contenant diverses opérations de type CRUD.

Le dossier **components** contient principalement des composants réutilisables et utilisés pour l'accessibilité, l'affichage de données ou la navigation.

Le dossier **security** contient diverses méthodes utilisées pour l'appel à l'API, le stockage des tokens, l'utilisation d'un contexte pour la transmission de données entre les composants parents et enfants, la régénération du token une fois l'utilisateur connecté.

Le dossier **utils** contient la configuration et la connexion des Socket côté client.

On retrouve une documentation de l'application mobile dans le fichier **APP\_README.md**, dans lequel est mentionné les langages utilisés, comment lancer l'application depuis le terminal ou encore l'architecture du projet.

---

## FONCTIONNEMENT ET SÉCURITÉ

Le coeur de l'application se situe dans le fichier App.js et contient diverses couches de sécurité. On utilise un provider pour toute l'application ainsi qu'un firewall pour s'assurer que l'utilisateur est bien connecté. Ces points là sont décrit un peut plus bas.

```
const Stack = createStackNavigator();

export default function App() {
  const [user, setUser] = React.useState(0);

  const userCredential = async () => {
    await getUserId()
      .then((res) => setUser(res))
  }

  React.useEffect(() => {
    userCredential();
  }, [])

  return (
    <AuthState.Provider value={{ user, setUser }}>
      <NavigationContainer>
        <Firewall>
          <Stack.Navigator>
            {user == 0 ? (
              <Stack.Group screenOptions={{ headerShown: false }}>
                <Stack.Screen
                  name="Auth"
                  component={AuthNavigator}
                />
              </Stack.Group>
            ) : (
              <Stack.Group>
                <Stack.Screen
                  name="App"
                  component={LoggedInNavigator}
                  options={{ headerShown: false }}
                />
              </Stack.Group>
            )}
          </Stack.Navigator>
        </Firewall>
      </NavigationContainer>
    </AuthState.Provider>
  );
}
```

On utilise la méthode getUserId( ) qui retourne l'ID de l'utilisateur lorsque celui-ci est connecté. Le résultat est ensuite stocké dans la variable user via le hook useState.

```
const getUserId = async () => {
  try {
    const jsonValue = await AsyncStorage.getItem('user_id')
    return jsonValue != null ? JSON.parse(jsonValue) : null;
  } catch(e) {
    console.log(e);
  }
}
```

Cela permet de contrôler rapidement si l'utilisateur est bien connecté, à saisir ses identifiants et qu'ils correspondent à ceux stocké en base de données. Si tel est le cas, il peut alors profiter pleinement de l'application et utiliser les navigators présent dans le groupe nommé App.

L'application utilise un provider dans lequel on passe la valeur de l'ID de l'utilisateur. Ce dernier sera ensuite accessible et modifiable depuis n'importe quel composant enfants, sans forcément devoir le passer en props.

```
const AuthState = React.createContext({
  user: 0,
  setUser: (u) => {}
});
```

Enfin, les navigators sont englobés par un firewall, qui check si l'ID de l'utilisateur est différent de 0 (valeur initiale) et active la connexion au socket. Il effectue également une régénération des tokens de l'utilisateurs pour qu'il reste connecté une fois ses identifiants saisis et contrôlés.

```
function Firewall({ children }) {
  const { user } = React.useContext(AuthState);

  React.useEffect(() => {
    if(user !== 0) {
      socket.auth = { user };
      socket.connect();
    }

    let interval = setInterval(() => {
      check()
    }, 1000 * 60 * 5); // 1000ms * 60s * 5 = 5min

    return function cleanup() {
      clearInterval(interval);
    }
  }, [user])

  const check = async () => {
    if(user !== 0) regenerateToken();
  }

  return <>{children}</>
}
```



---

## REQUÊTES VERS L'API

On retrouve des méthodes réutilisables et génériques, qui prennent selon les besoins de l'API différents paramètres à définir lors de l'utilisation de celles-ci.

Chaque méthode est auto-documentée pour aisément retrouver les paramètres nécessaires lors de son utilisation.

Pour les méthodes nécessitant le token de l'utilisateur, on utilise l'asyncStorage pour récupérer celui-ci et l'envoyer dans le headers. Il sera ensuite accessible et exploitable au sein de l'API pour s'assurer que l'utilisateur possède les droits demandés pour effectuer des opérations en base de données depuis un endpoint.

```
/**
 * Request without access token
 * @param {*} path endpoint URL
 * @param {*} method GET, POST, PUT or DELETE
 */
const simpleRequest = async (path, method) => {
  let result = null;

  await fetch("http://127.0.0.1:3001/api/" + path, {
    method: method,
    headers: {
      'Content-Type': 'application/json',
    }
  })
  .then((response) => response.json())
  .then((data) => { result = data });

  return result;
};

/**
 * Request without access token
 * @param {*} path endpoint URL
 * @param {*} method GET, POST, PUT or DELETE
 * @param {*} content object value
 */
const secureRequestContent = async (path, method, content) => {
  const access = await getAccessToken()
  let result = null;

  await fetch("http://127.0.0.1:3001/api/" + path, {
    method: method,
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${access}`,
    },
    body: JSON.stringify(content),
  })
  .then((response) => response.json())
  .then((data) => { result = data });

  return result;
};
```

---

## NAVIGATORS

En React Native, les navigators permettent de déclarer le routage de l'application. Un navigator peut contenir une imbrication de screens ou de groupes de screen.

Les navigators sont très pratique pour s'assurer qu'un utilisateur ne puisse accéder aux pages dont il a accès. Par exemple, une fois que l'utilisateur est connecté grâce à ses identifiants, s'il souhaite afficher la liste des conversations privées, les messages au sein de l'une d'entre elles ou les paramètres, la fonction ConversationNavigator est appelée.

Elle contient différents screens (LIST/CREATE, GET et UPDATE) qui s'adapte à l'opération demandée par l'utilisateur. Chaque screen peut prendre divers paramètres comme le titre de l'écran, l'affichage du header, le composant à afficher, ...

```
const ConversationStack = createNativeStackNavigator();

export default function ConversationNavigator() {
  return (
    <ConversationStack.Navigator screenOptions={{ headerShown: false }}>
      <ConversationStack.Screen
        name="Conversations"
        component={Conversations}
      />
      <ConversationStack.Screen
        name="Conversation"
        component={Conversation}
        options={({ route }) => ({ title: route.params.name })}
      />
      <ConversationStack.Screen
        name="ConversationSettings"
        component={ConversationSettings}
        options={({ route }) => ({ title: route.params.name })}
      />
    </ConversationStack.Navigator>
  );
}
```

Comme vous avez pu le constater dans la partie fonctionnement et sécurité, le fichier App.js contient un ternaire retournant le navigator d'authentification si l'ID de l'utilisateur est égal à 0, soit le navigator authentifié si l'ID est différent de 0.

Cette imbrication de navigator permet de s'assurer que l'utilisateur peut accéder à une conversation privée uniquement depuis la liste de celle-ci ou encore qu'il peut modifier / supprimer une conversation uniquement depuis le screen d'une conversation privée.

De plus, cela permet de modifier l'affichage entre l'écran de saisie d'identifiant et la page d'accueil lorsque l'utilisateur s'est connecté avec succès.

```

export default function LoggedInNavigator() {
  return (
    <UserTab.Navigator screenOptions={({ route }) => ({
      tabBarIcon: ({ focused, color, size }) => {
        let iconName;

        if (route.name === 'Accueil') {
          iconName = focused ? 'ios-home' : 'ios-home-outline';
        } else if (route.name === 'Messages') {
          iconName = focused ? 'ios-chatbubbles' : 'ios-chatbubbles-
            outline';
        } else if (route.name === 'Groupes') {
          iconName = focused ? 'ios-people' : 'ios-people-outline';
        } else if (route.name === 'Compte') {
          iconName = focused ? 'ios-man' : 'ios-man-outline';
        }

        return <Ionicons name={iconName} size={size} color={color} />;
      },
      tabBarActiveTintColor: 'tomato',
      tabBarInactiveTintColor: 'gray',
    })>
    <UserTab.Screen
      name="Groupes"
      component={ChannelNavigator}
      options={{ headerShown: false }}
    />
    <UserTab.Screen
      name="Messages"
      component={ConversationNavigator}
      options={{ headerShown: false }}
    />
    <UserTab.Screen
      name="Compte"
      component={ProfilNavigator}
      options={{ headerShown: false }}
    />
    </UserTab.Navigator>
  )
}

```

---

## SCREENS

Les screens sont les composants utilisés dans les navigators. Ils peuvent contenir ou non des requêtes, et celles-ci peuvent être de différents types pour répondre au mieux aux opérations demandées par l'utilisateur.

On retrouve ainsi selon la configuration du screen, une requête sécurisée ou une requête ne nécessitant pas de token dans les headers.

Par exemple, pour le screen de connexion, on retrouve une requête simplifiée (puisque l'utilisateur n'est pas encore authentifié, l'utilisation du contexte pour mettre à jour l'ID de l'utilisateur si la connexion s'est déroulée avec succès ainsi que l'utilisation du stockage asynchrone pour stocker le résultat de la requête (tokens, ID).

Extrait de code ci-dessous :

```
export default function Login({ navigation }) {
  const [email, onChangeEmail] = React.useState('');
  const [password, onChangePassword] = React.useState('');

  const { setUser } = React.useContext(AuthState);

  const userLogin = async () => {
    if(email !== '' && password !== ''){
      let user = {
        'email': email,
        'password': password,
      }

      await simpleRequestContent('login', 'POST', user)
        .then((res) => {
          setCredentials(
            res.data.access_token,
            res.data.refresh_token,
            res.data.user_id
          )
          setUser(res.data.user_id)
        })
    }
  }

  return (
    <SafeAreaView style={styles.screen}>
      <FixedHeaderGoBack
        goBack={() => navigation.goBack()}
      />

      <KeyboardView>
        <View style={styles.viewAuth}>
          <View style={styles.whiteCard}>
            <FormInput
              onChangeText={onChangeEmail}
              value={email}
              placeholder="Email"
              keyboardType="email-address"
            />

            <SecureInput
              onChangeText={onChangePassword}
              value={password}
            />

            <Pressable
              title={'Connexion'}
              onPress={() => userLogin()}
              style={styles.buttonMv2}
            >
              <Text style={styles.blackBtnText}>Connexion</Text>
            </Pressable>
          </View>
        </View>
      </KeyboardView>
    </SafeAreaView>
  )
}
```

L'affichage présente divers composants personnalisables, notamment un input pour l'adresse email, un input sécurisé pour le mot de passe (affichage ou non de la saisie) et un bouton.

---

## COMPOSANTS

Un composant est un élément réutilisable et personnalisable. En effet, on peut définir ou non une liste précise de paramètres (props) à implémenter lorsque l'on appelle un composant.

Grâce à ses composants, le code gagne en maintenabilité et clarté. Chaque composant est auto-documenté et donne des informations sur les props attendus, s'ils sont optionnels ou non.

La maintenabilité est renforcée par le fait que lorsque l'on souhaite modifier un composant, les modifications sera appliquées à chaque appel du composant. Il faut cependant être vigilant aux effets de bords si les modifications apportées sont vraiment significatives.

Par exemple, pour le composant `FormInput` qui est appelé dans le formulaire de connexion, d'inscription ou encore de modification du nom d'un groupe, d'un utilisateur ... si je modifie ce composant, il sera mis à jour à tous les endroits où il est appelé.

```
import * as React from 'react';
import { TextInput } from 'react-native';

import styles from '../..//style/style';

/**
 * @param {*} style CSS (default = styles.input)
 * @param {*} onChangeText TextInput tag onChangeText method
 * @param {*} value TextInput tag value
 * @param {*} placeholder TextInput tag placeholder
 * @param {*} keyboardType TextInput tag keyboardType
 * @param {*} lines TextInput tag numberOfLines (default = 1)
 */
export default function FormInput(props) {
  return (
    <TextInput
      style={props.style ? props.style : styles.input}
      onChangeText={props.onChangeText}
      value={props.value}
      placeholder={props.placeholder}
      keyboardType={props.keyboardType}
      autoCapitalize="none"
      placeholderTextColor="#aaa"
      multiline={props.lines ? true : false}
      numberOfLines={props.lines ? props.lines : 1}
    />
  )
}
```

---

## SOCKETS

L'utilisation des sockets côté client est relativement simple puisque la connexion au serveur ne nécessite que ces quelques lignes :

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

const URL = "http://localhost:3001";
const socket = io(URL, { autoConnect: false });

export default socket;
```

Ensuite, lorsque l'on souhaite utiliser le temps réel, on appelle la constante socket dans une vue. Par exemple pour la création d'un message, une fois l'enregistrement en base de données effectué, on ajoute :

```
socket.emit("get-conversation-msg", id);
```

On utilise également un useEffect pour recevoir les données retournées par les sockets, toujours dans l'exemple ci-dessus :

```
React.useEffect(() => {
  socket.on('conversationMsg', (res) => {
    setConversation(res);
  })
}, [socket]);
```

---

## RECHERCHE EN ANGLAIS

Durant le développement de l'application mobile, j'ai consulté en permanence la documentation de React Native puisqu'il s'agissait d'une totale découverte et d'une première utilisation.

J'ai rencontré une problématique en lien avec la navigation entre les différentes pages, en utilisant uniquement le menu au bas de l'écran, et en délimitant des groupes de screens.

Pour résoudre la navigation entre les différents screens sous forme de groupes, j'ai utilisé la librairie React Navigation et sa documentation. Par exemple, un groupe de screens pour les conversations pourra comprendre une page listant toutes les conversations de l'utilisateur, une page pour envoyer des messages au sein de la conversation ou encore une page pour éditer ou supprimer la conversation.

Lien : <https://reactnavigation.org/docs/nesting-navigators>

Nesting navigators means rendering a navigator inside a screen of another navigator, for example:

```

function Home() {
  return (
    <Tab.Navigator>
      <Tab.Screen name="Feed" component={Feed} />
      <Tab.Screen name="Messages" component={Messages} />
    </Tab.Navigator>
  );
}

function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Home"
          component={Home}
          options={{ headerShown: false }}
        />
        <Stack.Screen name="Profile" component={Profile} />
        <Stack.Screen name="Settings" component={Settings} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}

```

In the above example, the Home component contains a tab navigator. The Home component is also used for the Home screen in your stack navigator inside the App component. So here, a tab navigator is nested inside a stack navigator:

- Stack.Navigator
  - Home (Tab.Navigator)
    - Feed (Screen)
    - Messages (Screen)
  - Profile (Screen)
  - Settings (Screen)

Nesting navigators work very much like nesting regular components. To achieve the behavior you want, it's often necessary to nest multiple navigators.

# IX. DÉVELOPPEMENT DE L'ESPACE ADMINISTRATEUR

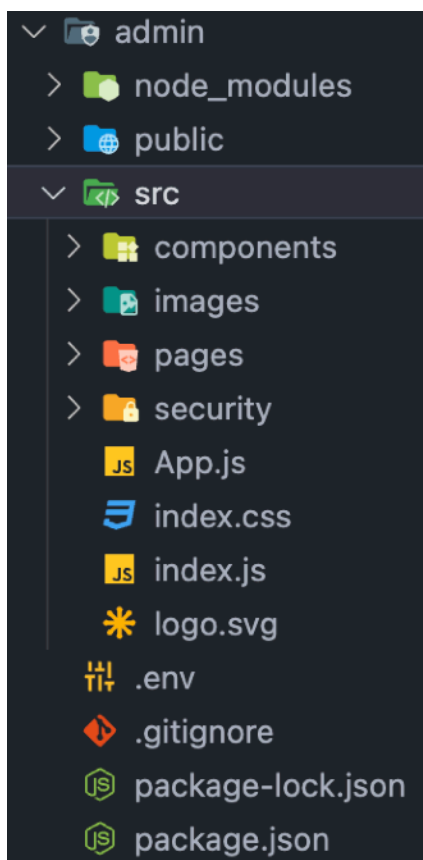
## FONCTIONNALITÉS

Une fois l'administrateur connecté à son espace via l'application web, il dispose d'un écran lui permettant de consulter rapidement la liste des conversations de groupes et des utilisateurs de l'application.

Pour les conversations de groupes, il peut supprimer directement chacun des groupes et / ou supprimer un ou plusieurs messages au sein de celui-ci.

Pour les utilisateurs, il peut directement supprimer chacun des utilisateurs depuis la page listant ces derniers.

## ARBORESCENCE



L'**architecture de l'application web** est présentée dans l'image ci-contre. Le coeur de l'application se situe dans le dossier **src** qui contient plusieurs couches de dossiers pour répartir la logique et faciliter la maintenance.

Le dossier **components** contient principalement des composants réutilisables et utilisés pour l'accessibilité, l'affichage de données ou la navigation.

Le dossier **pages** contient les différentes vues, ainsi on retrouve cinq fichiers : connexion, dashboard, liste des groupes de conversation, template pour un groupe de conversation et liste des utilisateurs.

Le dossier **security** contient diverses méthodes utilisées pour l'appel à l'API, le stockage des tokens, l'utilisation d'un contexte pour la transmission de données entre les composants parents et enfants, la régénération du token une fois l'utilisateur connecté.



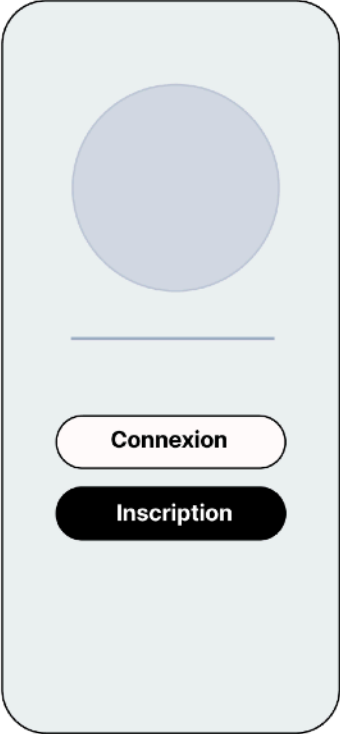
---

## FONCTIONNEMENT ET SÉCURITÉ

De nouveaux endpoints ont été créés pour répondre aux besoins d'un administrateur en plus des endpoints existants et contrôlant le rôle de l'utilisateur souhaitant effectuer des opérations en base de données.

# X. ANNEXE

Ecran ACCUEIL



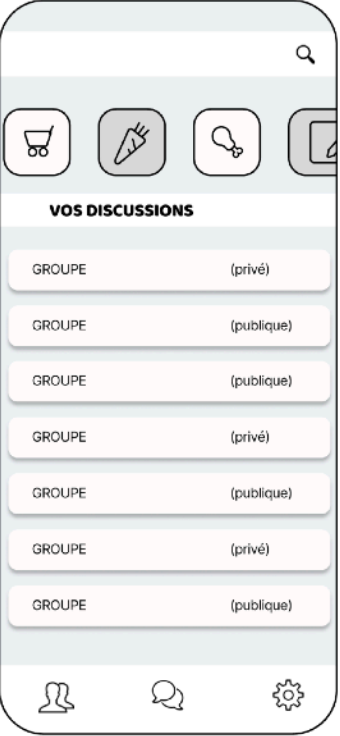
Ecran INSCRIPTION



Ecran CONNEXION



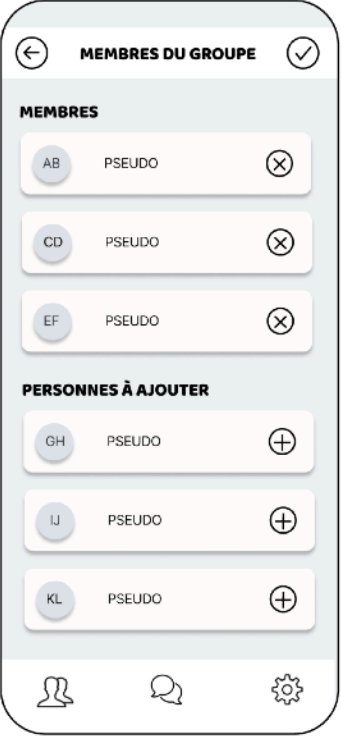
Ecran GROUPES



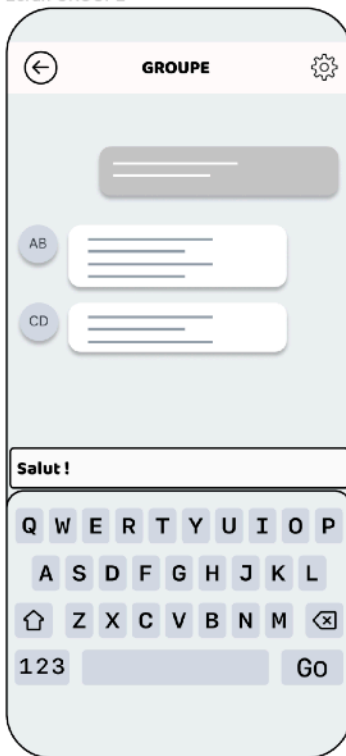
Ecran CREATION GROUPE



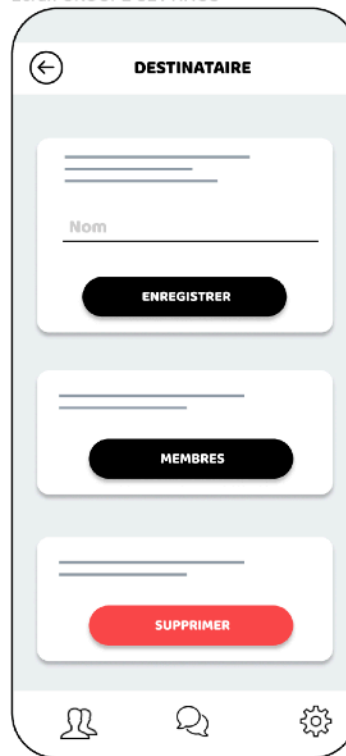
Ecran MEMBRES GROUPE



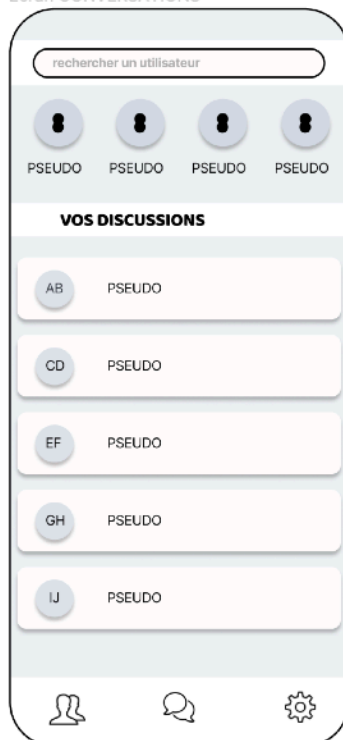
Écran GROUPE



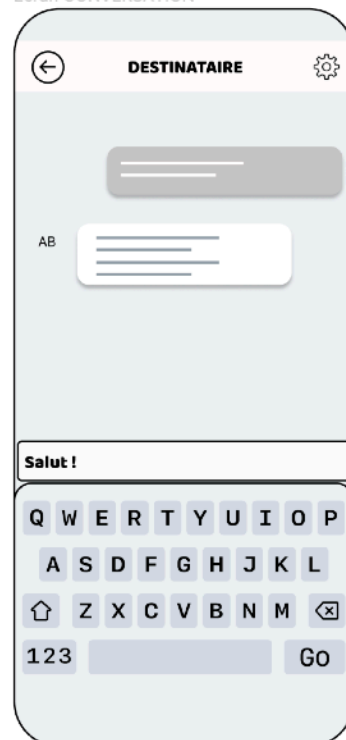
Ecran GROUPE SETTINGS



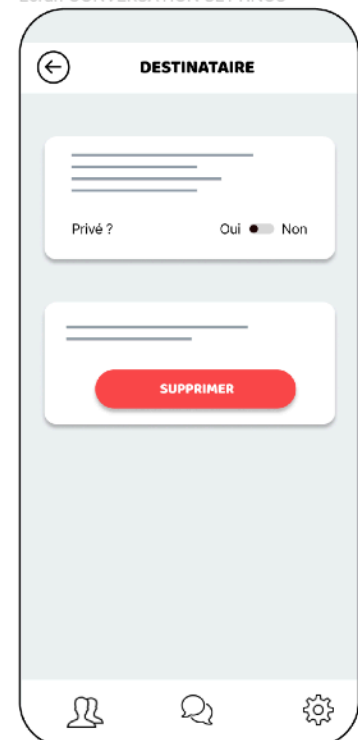
Ecran CONVERSATIONS



Écran CONVERSATION



Ecran CONVERSATION SETTINGS



Ecran PROFIL



Ecran MODIFICATION PROFIL

