

# Dossier de projet professionnel



## Réalisation de l'application web et mobile

--

***TISS-APP***

Présenté par **Samir Mokaddem**

## SOMMAIRE

<b>Introduction</b>	<b>3</b>
Présentation personnelle	3
Présentation du projet en anglais	3
Compétences Couvertes Par Le Projet	3
<b>Organisation Cahier Des Charges</b>	<b>5</b>
Analyse De L'existant	5
Les Utilisateurs Du Projet	5
Les Fonctionnalités Attendues	5
Application Mobile	5
Application Web	7
Contexte Technique	8
<b>Conception Du Projet</b>	<b>9</b>
Choix De Développement	9
Choix Des Langages	9
Choix Des Frameworks	9
Logiciels et autres outils	10
Organisation Du Projet	11
Architecture Logicielle	12
<b>Conception Du Front-end de l'application</b>	<b>13</b>
Arborescence Du Projet	13
Charte Graphique	14
Maquettage	15
<b>Conception Backend De L'application</b>	<b>16</b>
La Base De Données	16
Mise En Place La Base De Données	16
Conception De Base De Données	16
Modèle Conceptuel De Données	17
Modèle Logique De Données	17
Modèle Physique De Données	18
<b>Développement Du Backend De L'application</b>	<b>20</b>
Organisation	20
Arborescence	20
Fonctionnement de l'API	21
Middleware	22
Routage	22
Controller	24
Service	24
Model	24
Sécurité	27
Credentialstuffing : vol login password.	27
Chiffrement Des Données Sensibles	
JWT	28
Gestion des Droits	28
Helmet	30
Exemple de Problématique rencontrée	31
Recherches Anglophones	31

# Introduction

## Présentation personnelle

Je m'appelle Samir Mokaddem, Je suis devenu passionné grâce à ma famille qui travail aussi dans le numérique. J'ai débuté la programmation en regardant des vidéos tutoriel sur YouTube.

J'ai été attiré par l'informatique et les technologies de l'information. J'ai commencé à apprendre à programmer en autodidacte, en explorant des tutoriels en ligne et en créant mes propres projets.

Depuis lors, j'ai perfectionné mes compétences en programmation en apprenant plusieurs langages de programmation

Cela m'a permis de découvrir tous les métiers liés au numérique. J'ai suivi le cursus de la Développeur web et web mobile avec OpenClassrooms en 2022 et j'ai obtenu mon titre (bac+2).

Ce que j'aime en informatique c'est créer et innover. Je suis motivé par la possibilité de résoudre des problèmes et de créer des solutions créatives grâce à la programmation.

Aujourd'hui je suis en Coding School 2 afin de préparer mon titre de Concepteur / développeur d'application web et en alternance au sein de mon centre de formation La Plateforme.

## Présentation du projet en anglais

During my training at La Plateforme, I developed a mobile chat application using React Native. My goal was to create a simple and user-friendly platform for people to connect and communicate with each other. I didn't have a specific target audience in mind, and I wanted the app to be open and accessible to everyone, providing a space for relaxed and honest conversations.

To develop the app, I used the React Native Framework, Additionally, I utilized NodeJs coupled with ExpressJs for API development. Overall, the experience allowed me to gain valuable insights into mobile app development and further develop my skills in React Native.

## **Compétences couvertes par le projet**

**Ce projet couvre les compétences du titre 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
- Concevoir une application
- Développer des composants métier
- 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

# Organisation et cahier des charges

## Analyse de l'existant

Il existe déjà de nombreuses applications Chat permettant à des utilisateurs de communiquer entre eux par le biais de messages comme WhatsApp ou encore google chat.

## Les utilisateurs du projet

Ce projet se décompose en deux parties : une application mobile et une interface web.

La partie application mobile sera utilisée par les utilisateurs. Ils devront s'inscrire puis se connecter pour pouvoir échanger avec d'autres utilisateurs.

La partie site web est utilisée uniquement par les personnes ayant des droits administrateurs. Sur cette espace, les admins pourront gérer le chat, c'est-à-dire la possibilité de supprimer, modifier, ajouter des messages et des utilisateurs.

## Les fonctionnalités attendues

### Application Mobile

#### **Page d'accueil :**

Lorsque l'utilisateur lance l'application mobile, il est redirigé vers une page d'accueil, sur laquelle une navigation permet de se diriger vers la page connexion et inscription.

#### **Page inscription :**

Cette page permet aux utilisateurs de s'inscrire sur l'application. Pour cela, on collecte des informations de base sur l'utilisateur, telles que son nom, son adresse e-mail, son mot de passe pour créer un compte utilisateur.

#### **Page connexion :**

La page de connexion permet aux utilisateurs d'accéder à leur compte et d'utiliser les fonctionnalités réservées aux utilisateurs connectés tels que la personnalisation de leur profil, l'envoi de messages sur le chat etc... Cette page inclut des mesures de sécurité, telles que la vérification du mot de passe et le contrôle des mails déjà existants.

### **Page chat :**

La page de chat permet aux utilisateurs de communiquer en temps réel avec d'autres utilisateurs

### **Page profil :**

La page profil permet aux utilisateurs de gérer leur compte, en modifiant leurs informations personnelles tels que leur nom, prénom et leur photo de profil, etc.

### **Page contact :**

La page contact affiche la liste de tous les utilisateurs de l'application.

## **Application Web (Panel admin)**

### **Page d'accueil :**

Cette page affiche les cinq derniers utilisateurs et messages ainsi qu'un compteur qui permet de voir le nombre total de messages envoyés et le nombre total d'utilisateurs.

### **Page connexion admin :**

La page de connexion permet uniquement aux admins d'accéder au panel admin. Une fois connectés, ils pourront accéder à toutes les fonctionnalités du panel admin.

### **Page admin chat :**

Cette page affiche tous les messages de l'application. C'est ici que l'admin a la possibilité de supprimer des messages.

### **Page admin utilisateurs :**

Cette page affiche tous les utilisateurs de l'application. C'est ici que l'admin a la possibilité de supprimer ou modifier les informations des utilisateurs.

## **Contexte technique**

L'application mobile devra être accessible sur tous les systèmes d'exploitation Android et IOS.

L'application web devra être accessible sur tous les navigateurs.

# Conception du projet

## Choix de développement



**Pour réaliser ce projet, j'ai décidé d'utiliser uniquement du javascript. Avec Node Js comme environnement de développement.**

J'ai choisi d'utiliser JavaScript pour mon projet car ce langage de programmation est riche en fonctionnalités, ce langage a permis la création de nombreuses bibliothèques de code open source facilitant le développement de certaines fonctionnalités.

De plus, j'apprécie la polyvalence de JavaScript, qui peut être utilisé dans de nombreux contextes de développement, tels que les applications web, les applications mobiles, les jeux, les extensions de navigateur, etc. Cela me permet de réutiliser le code et de le rendre plus facilement adaptable à différents environnements.

L'écosystème JavaScript est également très riche, avec une communauté de développeurs très active qui crée constamment de nouvelles bibliothèques et frameworks pour aider les développeurs à résoudre des problèmes courants plus facilement. Les forums et les sites de questions/réponses sont également très fréquentés, ce qui facilite la recherche de solutions à des problèmes spécifiques.

En termes d'intégration, JavaScript peut être facilement intégré avec d'autres technologies, tels que les bases de données, les API, les CMS et les frameworks de backend, pour créer des applications web complètes. De nombreux frameworks de backend populaires tels que Node.js et Express.js sont basés sur JavaScript, ce qui facilite l'intégration avec d'autres technologies de backend courantes telles que MongoDB ou MySQL.

## Choix des frameworks

Pour la création de mon API j'ai choisi d'utiliser Express Js qui est un framework Node Js.

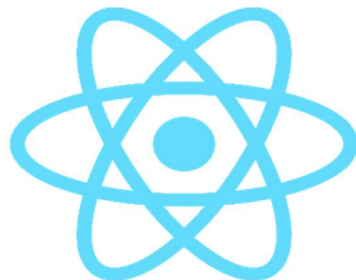
express

**ExpressJs est le framework le plus populaire pour Node Js.**

**C'est un framework minimaliste permettant de garder un certain contrôle dans le développement du projet et apporte peu de surcouche permettant ainsi de garder des performances optimales et une exécution rapide.**

J'ai choisi Express car il a un cadre d'exploitation libre et gratuit. Il comporte un ensemble de paquets, pour des fonctionnalités, des outils... qui aident à rendre le développement plus simple.

Ayant un calendrier à respecter, Express Js permet de développer, de façon rapide et efficace, une API. Ce qui correspond parfaitement à mon besoin.



**Pour Création De Mon Application Mobile, j'ai choisi le framework front-end react native car il est écrit en javascript et qu'il permet le développement d'un seul code pour les plateformes iOS Android.**



## Logiciels Et Autres Outils

Dans le cadre de ce projet, j'ai dû utiliser d'autres outils :

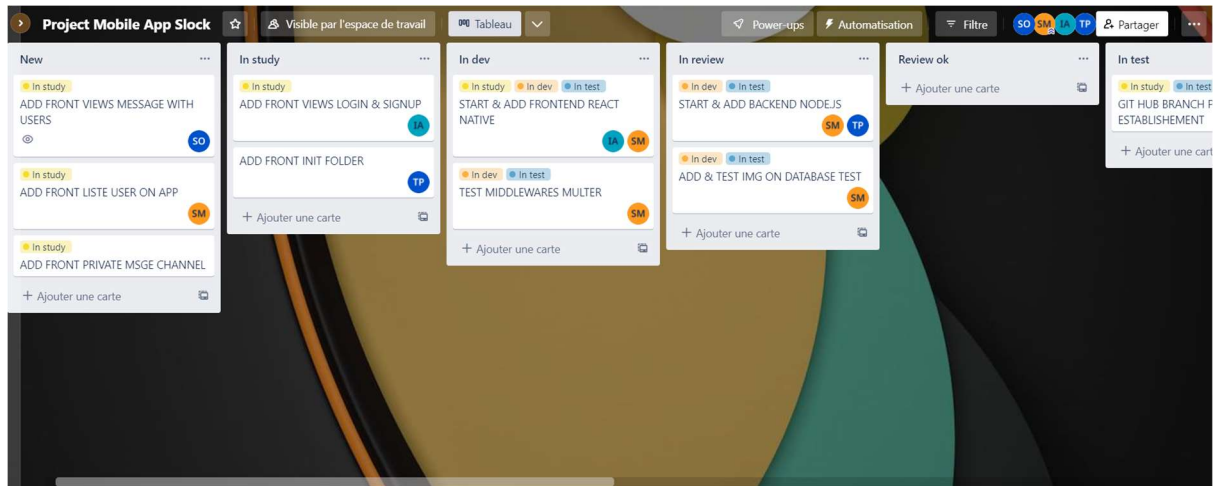
- **Visual Studio Code pour écrire mon code.**
- **Postman pour effectuer les requêtes API.**
- **Git pour le versionning mon code.**
- **Trello pour organiser son travail.**
- **NPM pour installer les paquets.**
- **Figma pour la création de mes maquettes et de la charte graphique.**
- **LucidChart pour le maquettage la base de données.**
- **Expo pour émuler son application mobile sur mon téléphone.**
- **Canvas pour la création du logo.**



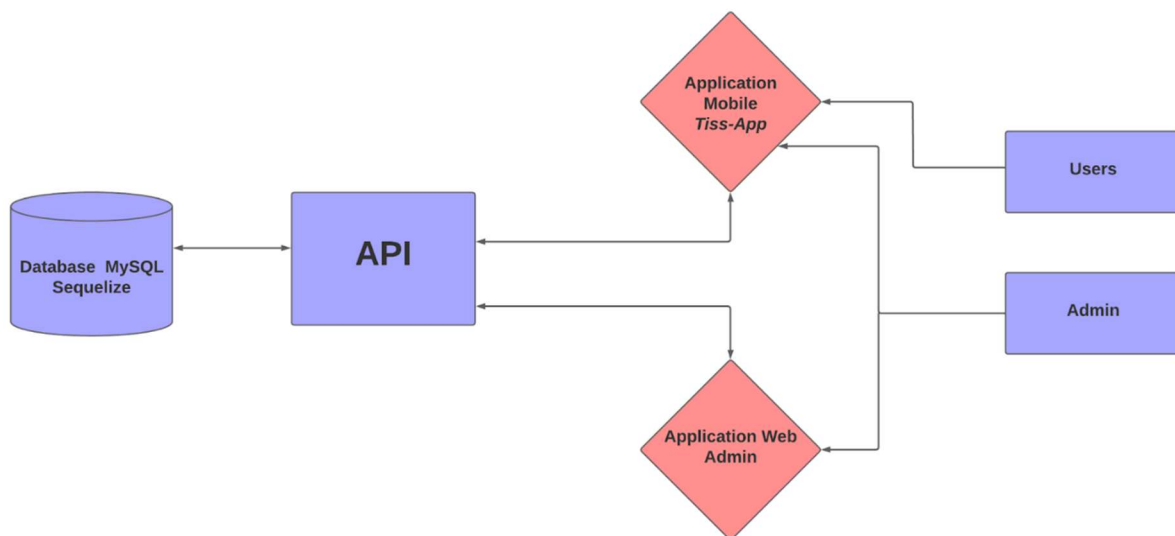
# Organisation Du Projet

Ce projet a été réalisé durant mon année de formation, avec des temps en entreprise et en centre ou j'avais des projets à rendre. Donc l'organisation de mon travail a été essentielle.

J'ai utilisé **Trello** pour lister les tâches à effectuées.



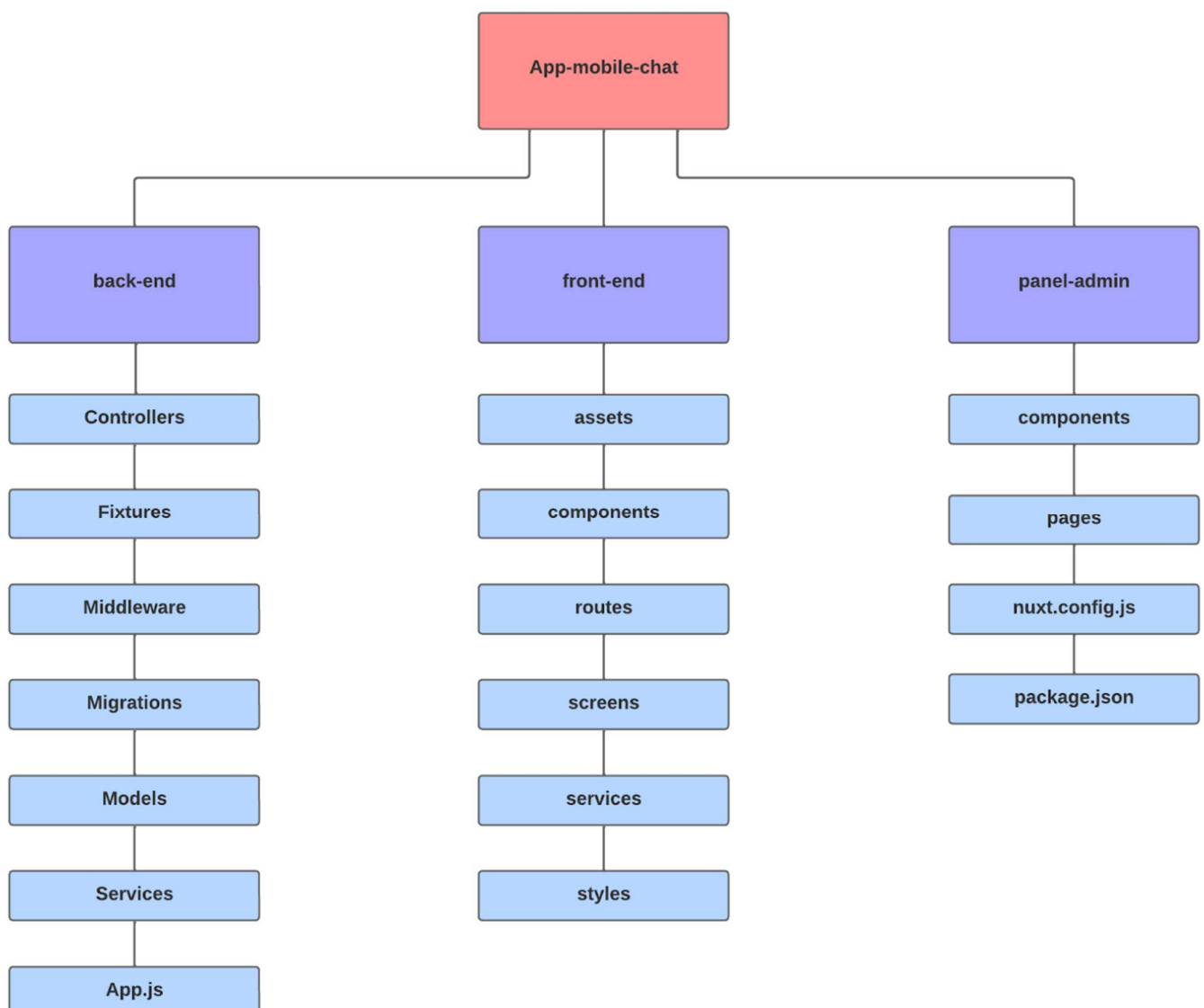
## Architecture logicielle

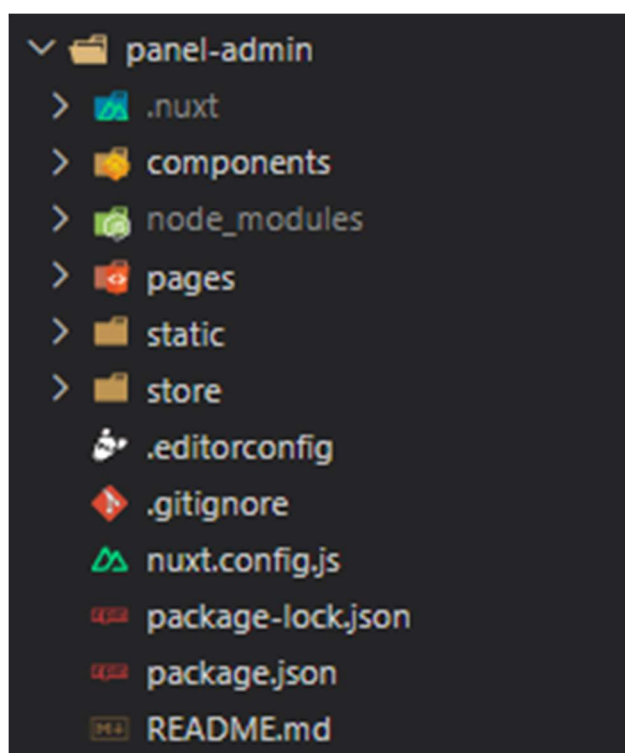
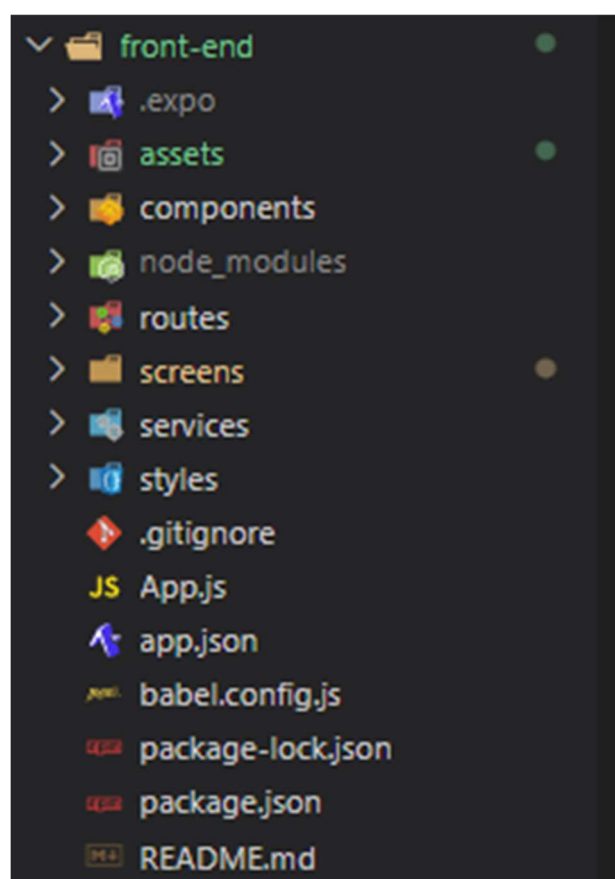
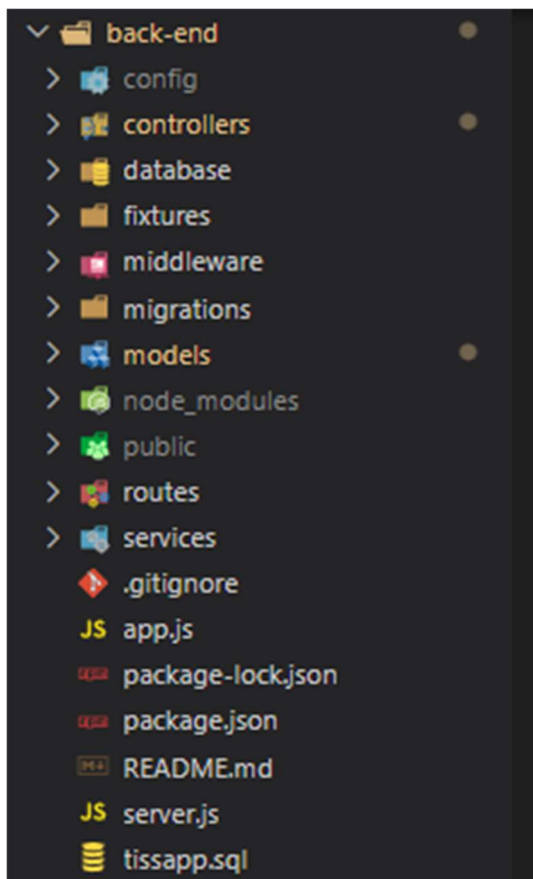


# Conception du Front-end de l'application

Pour créer la maquette et la charte graphique, j'ai choisi d'utiliser **Figma** car il est gratuit, plutôt simple d'utilisation et permet la réalisation de maquettes réalistes.

## Arborescence du projet

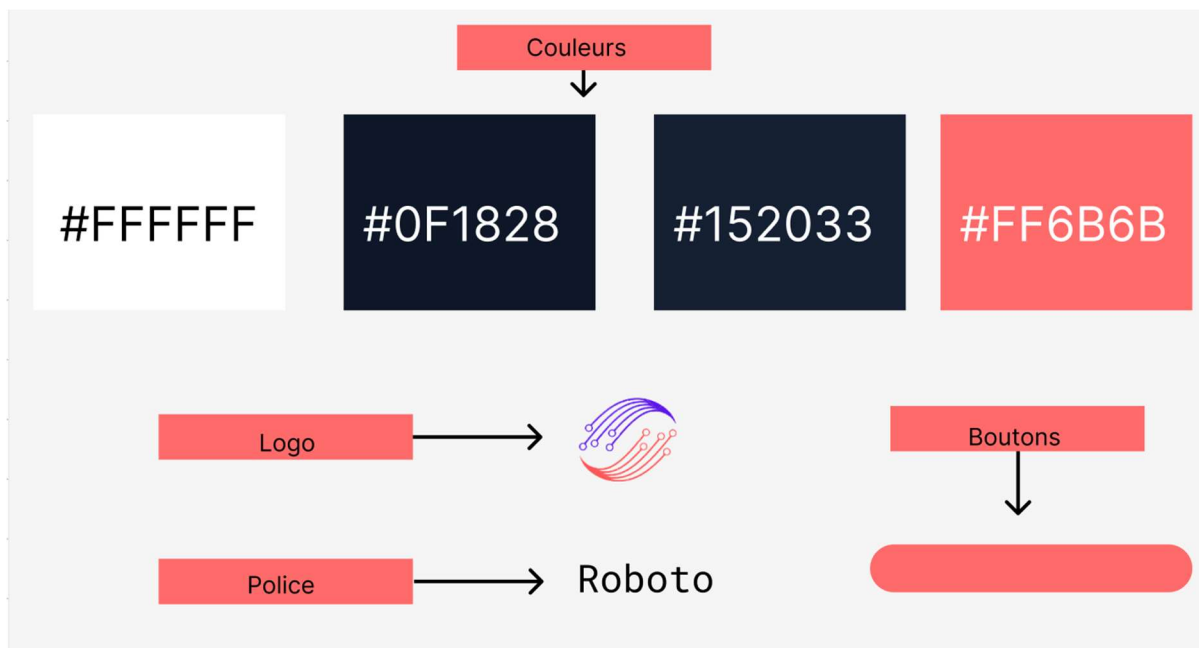




# Charte graphique

Après la création de l'arborescence du projet, j'ai commencé par la création du logo à l'aide **Canvas**. J'ai ensuite récupéré les couleurs de celui-ci pour créer la charte graphique en utilisant **Figma**.

Pour ce qu'il s'agit de la charte graphique, je défini entre autres les couleurs, le logo et la police utilisés pour la typographie afin que le site ait une identité visuelle, qu'il soit attrayant visuellement pour l'utilisateur. L'utilisateur ne voit et ne se fie qu'à l'apparence du site et ne peut pas voir le Back-office.

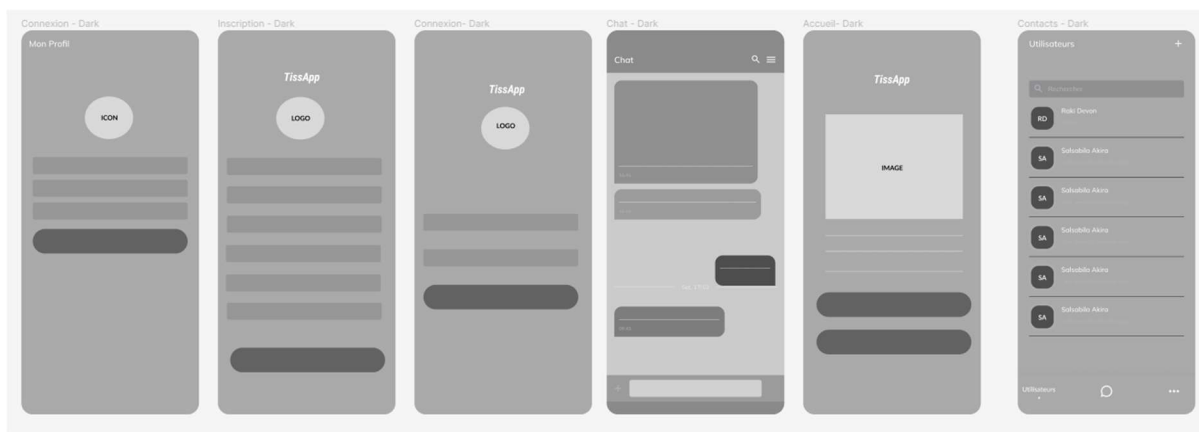


# Maquettage

Les maquettes ont été créées à l'aide de **Figma**.

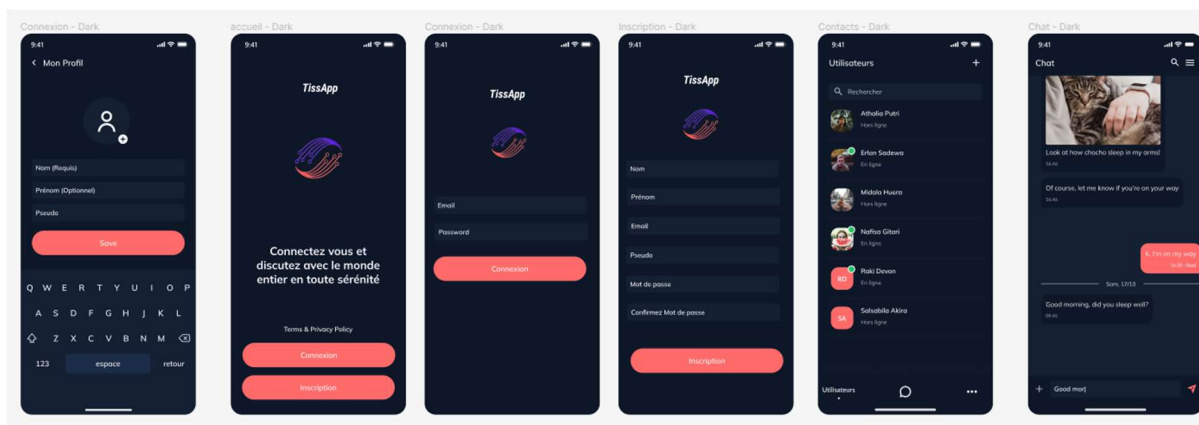
## Wireframe

J'ai commencé par réaliser un wireframe pour que je puisse visualiser et valider comment seront agencés les éléments sur les différentes pages. J'ai choisi un design minimaliste, très en vogue actuellement dans l'univers du web.



## Haute-fidélité

J'ai ensuite réalisé la maquette haute-fidélité permettant au client de voir clairement l'aspect graphique du site avec les couleurs et les images et ainsi validé la charte graphique du site.



# Mise en place d'un component React Native

L'utilité d'un composant React Native est de permettre la création de fonctionnalités et d'interfaces réutilisables dans une application mobile, favorisant ainsi la modularité, la simplicité du code et l'accélération du développement

Le component que j'ai créé permet de gérer l'insertion de messages et leur envoi. Ce composant encapsule la création du message et de l'image ainsi que le traitement de l'envoi du message. Je pourrai maintenant réutiliser ce composant à plusieurs endroits de mon application, ce qui me permettra de gagner du temps et de maintenir une cohérence dans l'expérience utilisateur.

```
import * as ImagePicker from 'expo-image-picker';
import * as Permissions from 'expo-permissions';
import { Ionicons } from '@expo/vector-icons';
import axios from 'axios';
import AsyncStorage from '@react-native-async-storage/async-storage';
import BaseUrl from '../services/BaseUrl';
```

Dans ce component j'import plusieurs package que j'ai installé à l'aide de la commande **npm install**.

```
export default function ImageUploadMessage() {
  const [image, setImage] = useState(null);
  const [newMessage, setNewMessage] = useState('');

  const [modalVisible, setModalVisible] = useState(false);

  // Check textError
  const [postImageError, setPostImageError] = useState('');
  const [postMessageSuccess, setPostMessageSuccess] = useState('');
  const [postMessageError, setPostMessageError] = useState('');

  const [isSending, setIsSending] = useState(false);
  const [messageQueue, setMessageQueue] = useState([]);
```

J'exporte ensuite ma logique et j'utilise les **hooks useState** pour gérer les états locaux des composants. Les **useState** me permettent de déclarer des variables d'état et de les mettre à jour de manière réactive.

L'utilisation de **useState** est bénéfique car cela me permet de suivre et de gérer facilement les changements d'état dans mes composants. Je peux initialiser une variable d'état avec une valeur par défaut et utiliser la fonction de mise à jour associée pour modifier cette valeur ultérieurement.

Cela est particulièrement utile dans le contexte de ce projet car il me permet de maintenir et de synchroniser l'état des différentes parties de l'application. Par exemple, je peux utiliser **useState** pour stocker l'image sélectionnée par l'utilisateur, le nouveau message saisi, ou encore pour contrôler la visibilité d'un élément comme une modal.

Grâce à l'utilisation de **useState**, je peux rendre mon application réactive en mettant à jour dynamiquement les valeurs des variables d'état. Cela facilite également la communication entre les différents composants, car je peux passer ces variables d'état en tant que propriétés et les mettre à jour à mesure que l'utilisateur interagit avec l'application.

Une fois mes variables d'état récupérées je crée ma fonction avec ma requête **Axios**

```
const response = await axios.post(`${API_URL}/api/posts`, postMessage, {
  headers: {
    'Content-Type': 'multipart/form-data',
    'Authorization': `Bearer ${token}`,
  },
});

if (response.status === 201) {
  setNewMessage('');
  setPostMessageSuccess("Message envoyé avec succès");
  removePicture();
}
```

Cette requête permet donc d'envoyer mon messages et d'enregistrer une image s'il y a une image enregistrée dans ma variable d'état **image**.

```
useEffect(() => {
  if (postImageError !== '' || postMessageSuccess !== '' || postMessageError !== '') {
    setTimeout(() => {
      setPostImageError('');
      setPostMessageSuccess('');
      setPostMessageError('');
    }, 2000);
  }
}, [postImageError, postMessageSuccess, postMessageError]);
```

Une fois ma requête envoyée j'utilise aussi un **useEffect** dans ce composant **useEffect** est un **hook** de React qui me permet d'exécuter du code dans mes composants fonctionnels en réponse à des changements ou événements spécifiques. Cela rend mes composants plus réactifs et flexibles.

Lorsque j'ai utilisé **useEffect** dans ce code, c'était pour gérer l'affichage des messages d'erreur et de succès pendant une courte période. J'ai configuré **useEffect** pour surveiller les variables d'état **postImageError**, **postMessageSuccess** et **postMessageError**.



```

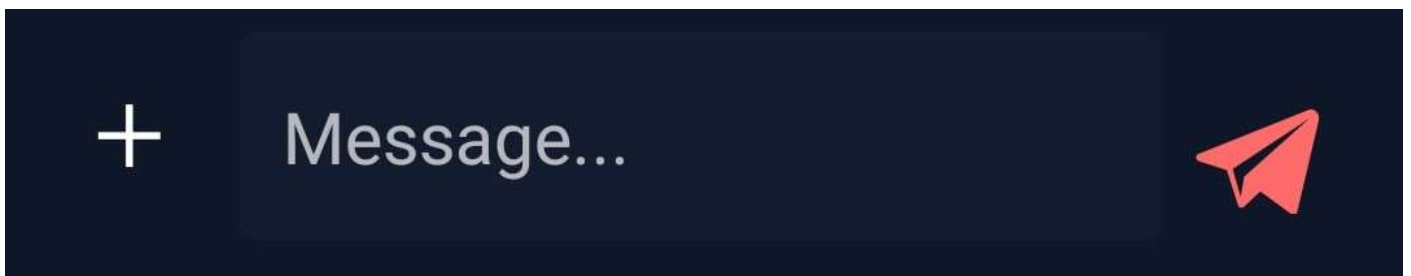
return (
  <View style={PostStyle.postContainer}>
    <View>
      { /* Input & Button views */ }
      {postMessageError !== '' && <Text style={PostStyle.errorText}>{postMessageError}</Text>}
      {postMessageSuccess !== '' && <Text style={PostStyle.SuccessText}>{postMessageSuccess}</Text>}
    </View>
    { /* BTN UPLOAD IMAGE */ }
    <View style={PostStyle.inputContainer}>
      {!image ? (
        <TouchableOpacity onPress={() => setModalVisible(true)} style={PostStyle.selectImageButton}>
          <Ionicons name="add-outline" size={24} color="white" />
        </TouchableOpacity>
      ) : null}
    </View>
  </View>
)

```

Une fois que j'ai terminé la logique de mon component, je peux procéder à la création de l'affichage.

Dans la partie return de mon component, je peux commencer à créer mes éléments en utilisant les balises correspondantes à ceux que je souhaite afficher. Par exemple, je peux utiliser la balise **<Text>** pour afficher du texte, la balise **<Image>** pour afficher une image, la balise **<View>** pour créer des conteneurs, et ainsi de suite.

En résumé, une fois que mon component est fonctionnel et terminé, je peux le réutiliser à plusieurs reprises dans mon application, ce qui améliore la cohérence et l'efficacité de l'expérience utilisateur.



# Conception du Backend de l'application

## Mise en place de la base de données

Pour ce projet, il est indispensable d'avoir une base de données, pour ainsi garder de façon pérenne les messages des utilisateurs.

## Modélisation de la base de données

**MERISE (Méthode d'Etude et de Réalisation Informatique pour les Systèmes d'Entreprise)**, c'est une méthode française qui a émergée dans les années 70 en France et qui permet la modélisation et la conception de S.I. Parmi les ressources informatiques de ces S.I., il y a en particulier les fichiers de données, bases de données et système de gestion de bases de données (S.G.B.D.).

C'est sur ce dernier point que la méthode MERISE nous a été utile, car c'est en se basant sur ses principes de modélisation que nous avons conçu la base de données de Tiss-App.

### Modèle conceptuel de données (MCD)

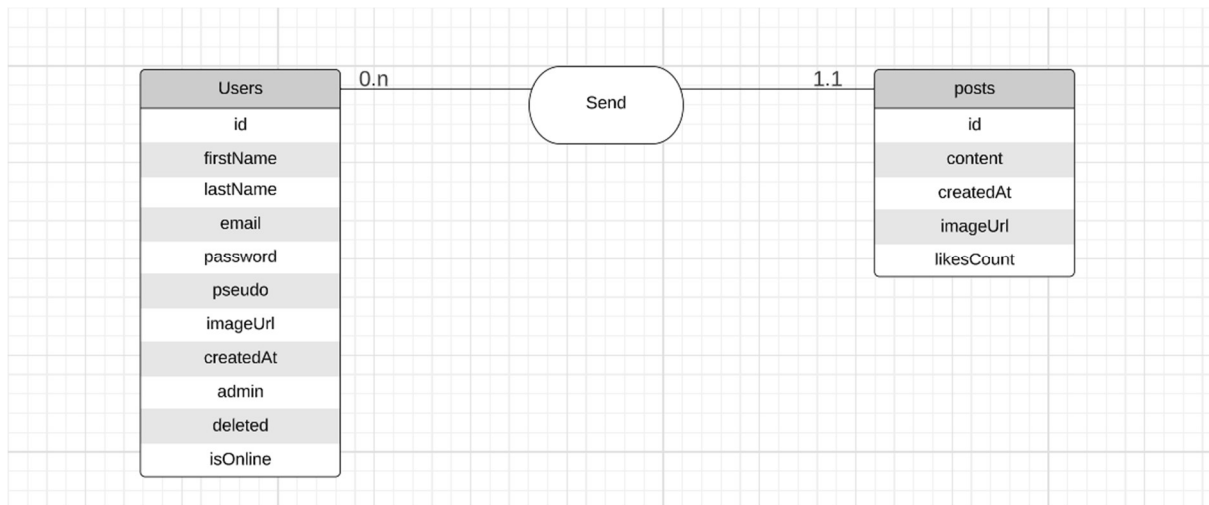
C'est un modèle simplifié de la base de données, où les tables sont seulement au stade d'entité, entre les différentes entités il y a des liaisons que l'on appelle association, qui est le verbe d'action qui décrit l'opération qui se fait entre les deux entités, donc nous distinguons principalement les entités et les associations, d'où son second nom de schéma Entité/Association.

Tout d'abord, il a fallu imaginer tous les besoins du client et notamment la partie admin. A partir de ces besoins, j'ai été en mesure d'établir les règles de gestion des données à conserver.

Ensuite, il faut définir le dictionnaire des données, c'est-à-dire toutes les données élémentaires qui vont être conservées en base de données et définir certaines caractéristiques qui figureront dans le MCD. Parmi ces caractéristiques, on retrouve par exemple la référence d'une donnée et notamment un identifiant unique, sa désignation, son type etc.

Étape finale à sa conception, j'ai pu à partir des informations précédemment recueillies, créer chaque entité, unique et décrite par un ensemble de propriétés ; Et leurs associations permettant de définir les liens et cardinalités entre les entités.

# Le MCD



## Modèle Logique de données (MLD)

Le modèle logique de données (MLD) est une étape intermédiaire entre le modèle conceptuel de données et le modèle physique de données.

Le passage d'un MCD en MLD s'effectue selon quelques règles de conversion précise :

Une entité du MCD devient une table. Dans un SGBD de type relationnel, une table est une structure tabulaire dont chaque ligne correspond aux données d'un objet enregistré et où chaque colonne correspond à une propriété de cet objet.

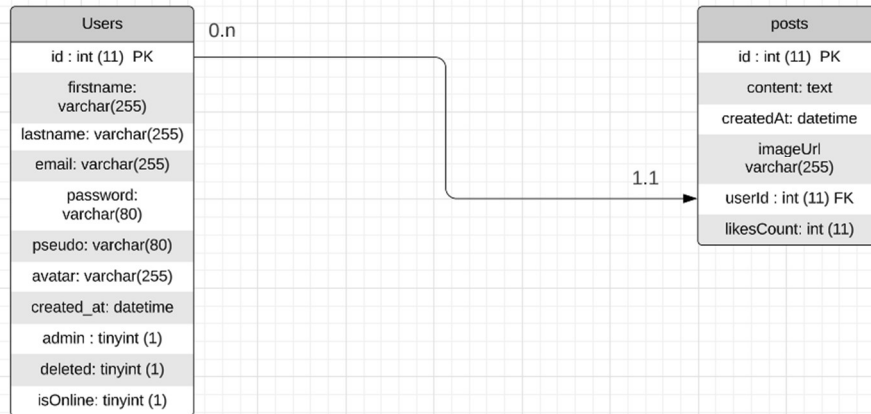
Ces colonnes font notamment référence aux caractéristiques définies dans le dictionnaire de données du MCD.

Les identifiants respectifs de chaque entité deviennent des clefs primaires et toutes les autres propriétés définies dans le MCD deviennent des attributs. Les clefs primaires permettent d'identifier de façon unique chaque enregistrement dans une table et ne peuvent pas avoir de valeur nulle.

Les cardinalités de type "0 : n" / "1 : n" sont représentées à travers le référencement de la clef primaire de la table qui la possède, en clef étrangère au sein de la table auxquelles elle est liée. Si deux tables liées possèdent une cardinalité de type "0 : n / 1 : n", la relation sera traduite par la création d'une table de jonction, dont la clef primaire de chaque table deviendra une clef étrangère au sein de la table de jonction.

# Le MLD

Diagram de MLD Chat



# Développement du backend de l'application

## Organisation

Mon back end a pour but d'être utilisé à la fois pour mon application web et mon application mobile. J'ai décidé de créer une API pour ne pas coder deux fois la même logique métier.

Dans le but de rendre mon backend plus efficace, je me suis concentré sur la logique et l'optimisation de mon code.

J'ai donc effectué des recherches dans ce sens.

Je divise donc mes programmes en différents modules, ainsi cela augmente la lisibilité du code et devient plus facile à maintenir pour les prochaines versions. J'évite les répétitions en créant des fonctions et des services.

La logique de mon code est divisée en services et fichiers.

## Arborescence

J'ai suivi une architecture N-tier. Le principe de cette architecture est la séparation des préoccupations pour éloigner la logique métier des routes de l'API.

Les différentes couches de l'application sont séparées :

- Le routeur,
- La couche applicative qui est divisée en deux couches :
  - Controller,
  - Router,
- La couche data (les modèles).

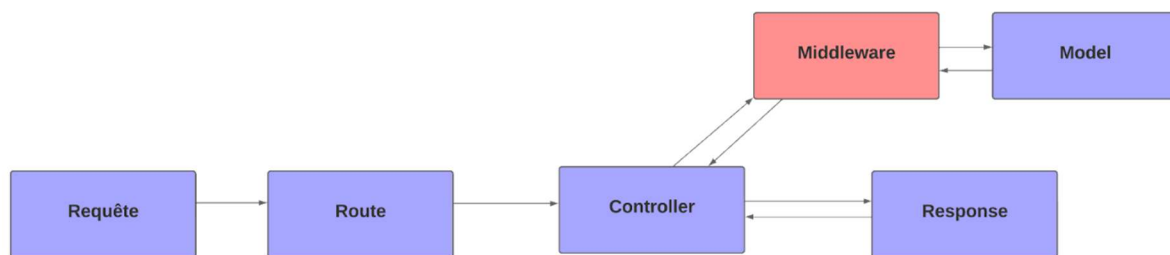
Mon back end est donc composé des dossiers suivants :

- **Routes** : regroupant tous les fichiers de mes routes (un fichier par CRUD d'une table de base de données)
- **Controllers** : Regroupant tous les Controller (un par route)
- **Middleware** : contenant des sous-dossiers (un par route) contenant les fichiers avec les middlewares des routes
- **Models** : contenant les modèles de toutes mes tables (un par table et par fichier)
- **Database** : contient les fichiers nécessaires pour la gestion de la base de données
- **Migrations** : contient les fichiers de migration pour la base de données de l'application.
- **Public** : images uploadées depuis l'application mobile
- **Fixtures** : enregistre des données pré-enregistrées pour la base de données afin de faciliter le développement et les tests.

## Fonctionnement de l'API

Lorsque le client envoie une requête sur mon API, le routeur analyse l'URL, en fonction de la route et de la méthode un Controller est appelé. Ce contrôleur va faire appel au middleware si besoin qui va communiquer avec le modèle afin de récupérer des données. Ensuite, ses données sont analysées par le middleware puis une réponse est envoyée au format JSON avec un code statut.

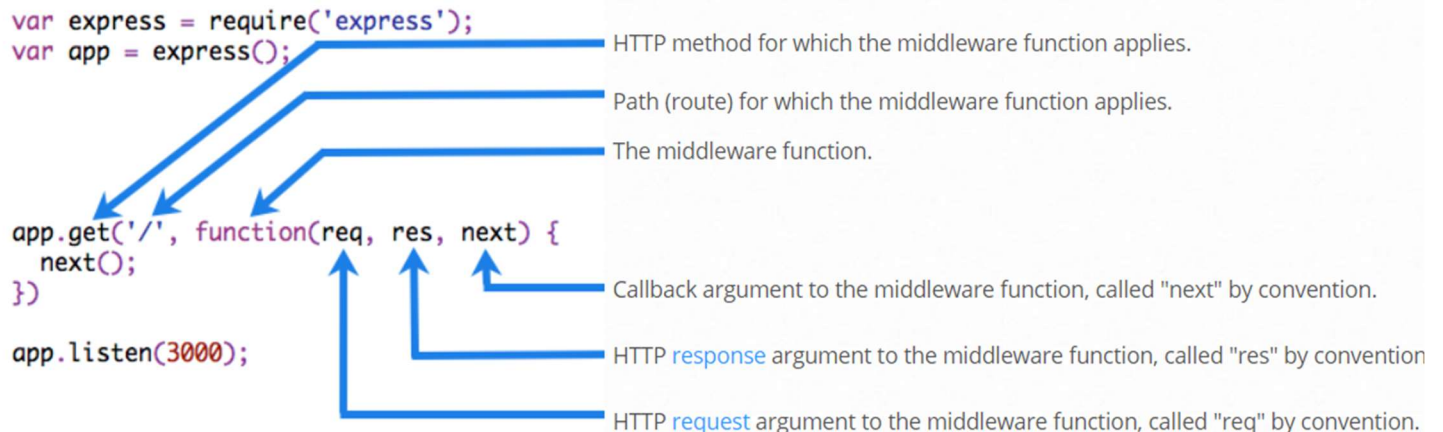
*Architecture de l'API*



Les différents statuts utilisés dans ce projet sont :

- **200 : OK**  
Indique que la requête a réussi
- **201 : CREATED**  
Indique que la requête a réussi et une ressource a été créé
- **400 : BAD REQUEST**  
Indique que le serveur ne peut pas comprendre la requête à cause d'une mauvaise syntaxe
- **401 : UNAUTHORIZED**  
Indique que la requête n'a pas été effectuée car il manque des informations d'authentification
- **403 : FORBIDDEN**  
Indique que le serveur a compris la requête mais ne l'autorise pas
- **404 : NOT FOUND**  
Indique que le serveur n'a pas trouvé la ressource demandée
- **405 : METHOD NOT ALLOWED**  
Indique que la requête est connue du serveur mais n'est pas prise en charge pour la ressource cible
- **500 : INTERNAL SERVER ERROR**  
Indique que le serveur a rencontré un problème.

## Fonctionnement des routes

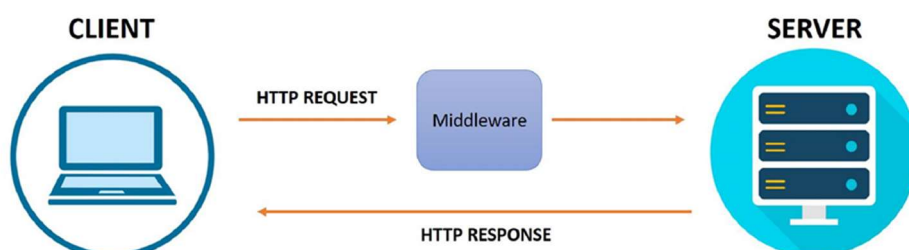


Lorsqu'une application Express reçoit une requête HTTP, elle crée deux objets : "req" contenant les informations de la requête, et "res" contenant les méthodes pour renvoyer une réponse. "req" est utilisé pour accéder aux données de la requête, telles que les paramètres d'URL ou les données du corps. "res" est utilisé pour envoyer une réponse, telle qu'une page HTML ou un objet JSON. Si un middleware est utilisé, il peut être appelé en utilisant la fonction "next", pour passer la requête au middleware suivant ou à la prochaine route correspondante.

## Fonctionnement d'un middleware

Dans une application web construite avec Express, un middleware est un composant logiciel intermédiaire qui est exécuté entre la réception d'une requête et l'envoi d'une réponse. Il peut être utilisé pour effectuer des tâches telles que la validation de données, l'authentification des utilisateurs, la gestion des erreurs, la compression de données, etc.

Le rôle principal d'un middleware dans une application avec Express est d'intercepter les requêtes et les réponses, et d'effectuer des opérations sur celles-ci avant qu'elles ne soient transmises à la route appropriée. Les middlewares sont généralement organisés en chaîne, de sorte que chaque middleware peut effectuer des opérations sur la requête avant de la transmettre au middleware suivant, et sur la réponse avant de la renvoyer à l'utilisateur.



## Fonctionnement des controllers

Les controllers sont créés dans le dossier controllers de mon application, qui contenait toutes les logiques pour ce contrôleur spécifique. J'ai également exporté la fonction du contrôleur en utilisant `module.exports`.

Pour placer le contrôleur dans mon application Node.js, j'ai créé une route correspondante dans mon fichier de routes, qui faisait appel à la fonction du contrôleur.

Exemple : Si une erreur se produit lors de l'appel aux méthodes Sequelize, la fonction catch l'erreur et renvoie une réponse JSON avec un code d'erreur 500 et le message d'erreur dans l'objet JSON.

En gérant les erreurs de cette manière, j'ai pu fournir des réponses claires et informatives aux utilisateurs de mon application, tout en assurant que les erreurs étaient correctement gérées et que mon application restait stable.

```
exports.login = async (req, res, next) => {
  try {
    const response = await User.authenticate(req.body.email, req.body.password);

    if (response.valid) {
      await User.update(
        { isOnline: true },
        { where: { id: response.user.id } }
      );
      res.status(201).json(newToken(response.user));
    } else {
      res.status(401).json({ error: response.message });
    }
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: error.message });
  }
};
```

## Création de la base de données

XAMPP est un logiciel qui regroupe Apache, MySQL, PHP et d'autres outils de développement web en un seul package. En utilisant XAMPP, j'ai pu installer et configurer Apache et MySQL sur ma machine en quelques clics, ce qui m'a permis de me concentrer sur le développement de mon projet plutôt que sur la configuration de l'environnement.

En utilisant XAMPP, j'ai également pu accéder facilement à PhpMyAdmin, une application web qui permet de gérer facilement les bases de données MySQL. PhpMyAdmin m'a permis de créer et de modifier des tables de base de données.

Dans l'ensemble, XAMPP a grandement simplifié le développement de mon projet Node.js avec Sequelize, en fournissant une installation rapide et facile d'Apache et de MySQL, ainsi qu'un accès facile à PhpMyAdmin pour la gestion de la base de données.



## Sequelize

La création de projet est réalisée avec Sequelize pour gérer une base de données MySQL. Après avoir configuré Sequelize dans mon projet et créé mes modèles, j'ai voulu créer ma base de données en utilisant la commande `npx sequelize-cli db:create`. Cette commande a créé une nouvelle base de données avec le nom que j'avais spécifié dans ma configuration Sequelize.

Pour générer une migration depuis un model `npx sequelize-cli migration:generate --name create-table`

Ensuite, j'ai voulu migrer mes modèles vers ma base de données en utilisant la commande `npx sequelize-cli db:migrate`. Cette commande a appliqué toutes les migrations en attente à ma base de données, ce qui a permis de créer toutes les tables et les colonnes définies dans mes modèles. Cela m'a permis de m'assurer que ma base de données était à jour avec les dernières modifications de mon code source.

Dans l'ensemble, l'utilisation de Sequelize a grandement simplifié la gestion de ma base de données dans mon projet Node.js. Les commandes `npx sequelize-cli db:create` et `npx sequelize-cli db:migrate` m'ont permis de créer et de migrer ma base de données en toute simplicité.

```
"use strict";
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable("Posts", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER,
      },
      userId: {
        allowNull: false,
        type: Sequelize.INTEGER,
        references: {
          model: "Users",
          key: "id",
        },
      },
    });
  },
}
```

## Model

La méthode "init" est une méthode statique fournie par la classe "Model" de sequelize. Cette méthode est utilisée pour initialiser la définition de notre modèle de données.

Dans notre cas, j'ai utilisé cette méthode pour définir les différentes propriétés de notre entité "User". J'ai défini le type de données, le statut d'obligation ou non de la propriété, et j'ai également ajouté des fonctions de validation pour certaines propriétés.

J'ai également passé quelques options supplémentaires à cette méthode. Par exemple, j'ai fourni l'instance de "sequelize" que j'utilise pour la gestion de la base de données et j'ai donné un nom à notre modèle pour faciliter sa référence ultérieurement.

Après avoir créé mon model, j'ai dû effectuer une migration pour créer la table correspondante dans ma base de données. Pour cela, j'ai utilisé la commande "npx sequelize-cli migration:generate" pour générer un fichier de migration vide, que j'ai ensuite rempli avec le code nécessaire pour créer ma table.

Dans ce fichier de migration, j'ai spécifié le nom de ma table ainsi que les différentes colonnes que je voulais y ajouter en utilisant la syntaxe fournie par Sequelize.

J'ai également spécifié les types de données pour chaque colonne, ainsi que les contraintes de validation nécessaires.

Une fois le fichier de migration rempli, j'ai utilisé la commande "npx sequelize-cli db:migrate" pour exécuter cette migration et créer ma table dans ma base de données.

## Sécurisation avec TOKEN ADMIN

J'ai créé une fonction middleware qui est chargée de sécuriser certaines routes de l'application en vérifiant la validité du jeton d'authentification.

Tout d'abord, j'ai importé les modules nécessaires pour mettre en œuvre cette fonctionnalité. J'ai utilisé la bibliothèque "jsonwebtoken" pour générer et vérifier les jetons d'authentification, et j'ai importé le modèle "User" de la base de données sequelize.

Ensuite, j'ai créé la fonction middleware elle-même. Cette fonction middleware est appelée chaque fois qu'une requête est envoyée à l'application, et elle vérifie si le jeton d'authentification est valide. Si le jeton est valide, la fonction middleware appelle la fonction "next" pour passer la main au middleware suivant, sinon elle renvoie une erreur d'authentification.

Pour vérifier la validité du jeton d'authentification, j'ai commencé par extraire le jeton de la requête HTTP à partir de l'en-tête "Authorization". J'ai ensuite utilisé la méthode "verify" de la bibliothèque "jsonwebtoken" pour décoder le jeton et obtenir l'ID de l'utilisateur.

Ensuite, j'ai vérifié si l'ID de l'utilisateur obtenu à partir du jeton correspond à l'ID de l'utilisateur qui a envoyé la requête. Si les ID ne correspondent pas, j'ai renvoyé une erreur d'authentification. Sinon, j'ai utilisé le modèle "User" pour récupérer l'utilisateur correspondant à l'ID, et j'ai ajouté cet utilisateur à l'objet de requête ("req.user") pour qu'il soit accessible aux middlewares suivants.

Enfin, si une erreur se produit à tout moment lors de la vérification du jeton ou de la récupération de l'utilisateur, j'ai renvoyé une erreur d'authentification avec un message personnalisé.

En somme, cette fonction middleware est un moyen simple mais efficace de sécuriser les routes de notre application en vérifiant l'authentification de l'utilisateur à l'aide d'un jeton d'authentification.

```
const db = require('../database');
const jwt = require('jsonwebtoken');
const { User } = db.sequelize.models;

module.exports = (req, res, next) => {
  try {
    const token = req.headers.authorization.split(' ')[1]; //récupération du token depuis le header Authorization
    const decodedToken = jwt.verify(token, 'RANDOM_TOKEN_SECRET');
    const userId = decodedToken.userId;
    if (req.body.userId && req.body.userId !== userId) {
      throw 'User ID non valable !';
    } else {
      User.findOne({ where: { id: userId } }).then((user) => {
        req.user = user;
        next();
      });
    }
  } catch (error) {
    res.status(401).json({
      error: new Error('Requête non authentifiée !'),
    });
  }
};
```

## Sécurisation du mot de passe BCrypt

Pour la sécurisation du mot de passe j'utilise deux fonctions :

La fonction **ensurePasswordIsStrongEnough** s'assure que le mot de passe respecte certains critères de complexité : il doit contenir au moins 8 caractères, dont au moins une lettre majuscule, une lettre minuscule, un chiffre et un caractère spécial. Si le mot de passe ne respecte pas ces critères, une erreur est levée.

La deuxième fonction, **addAuthenticationOn**, ajoute des fonctionnalités d'authentification à un modèle d'utilisateur. Cette fonction utilise la bibliothèque **bcrypt** pour chiffrer le mot de passe de l'utilisateur avant de le stocker dans la base de données. Elle définit également une méthode `authenticate` qui permet de vérifier si les informations d'identification fournies sont valides en comparant le mot de passe chiffré stocké en base de données avec le mot de passe fourni en clair.

Dans mon modèle **user**, **ensurePasswordIsStrongEnough**, importée via la validation **ensurePasswordIsStrongEnough**. Cette validation est définie dans le modèle `User` pour s'assurer que le mot de passe de l'utilisateur est suffisamment complexe pour résister aux attaques de force brute et autres formes d'attaques par dictionnaire.

```
const bcrypt = require('bcrypt');

function ensurePasswordIsStrongEnough(value) {
  const regex =
    /^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[@$!%*?&])[A-Za-z0-9\d@$!%*?&]{8,}$/;
  if (!value.match(regex)) {
    throw new Error(
      'Le mot de passe doit contenir au moins 8 caractères (dont au moins une majuscule, une minuscule, un chiffre, un caractère spécial).'
    );
  }
}

function addAuthenticationOn(User) {
  const encryptPassword = (user) => {
    if (user.changed('password')) {
      return bcrypt.hash(user.password, 10).then((hash) => {
        user.password = hash;
      });
    }
  };

  User.authenticate = async (email, password) => {
    const user = await User.findOne({ where: { email, deleted: false } });

    if (!user) {
      return { valid: false, message: 'Utilisateur non trouvé' };
    }

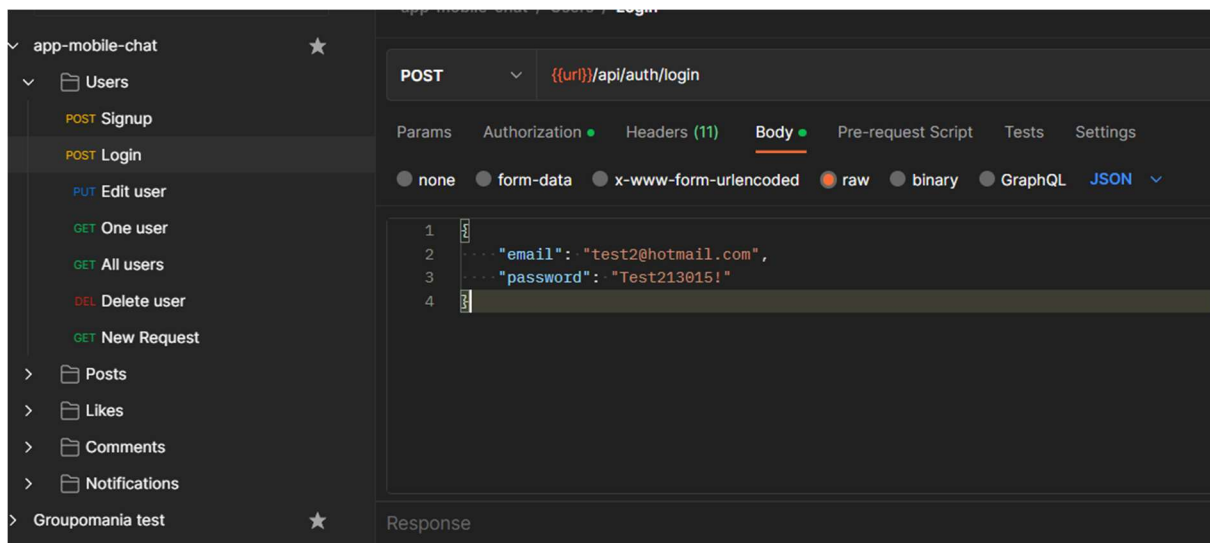
    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (isPasswordValid) return { valid: true, user };
    else return { valid: false, message: 'Mot de passe incorrect' };
  };

  User.beforeCreate(encryptPassword);
  User.beforeUpdate(encryptPassword);
}

module.exports = {
  ensurePasswordIsStrongEnough,
  addAuthenticationOn,
};
```

## Test des routes avec l'outil POSTMAN



Lorsque je développe une API Node.js Express, il est important de tester chaque route et chaque fonctionnalité pour m'assurer qu'elles fonctionnent correctement. Une façon de le faire est de tester l'API en utilisant une application tierce telle que Postman.

En utilisant Postman, je peux envoyer des requêtes HTTP aux différentes routes de mon API et visualiser les réponses renvoyées par l'API. Je peux également inclure des données de requête et de réponse, telles que des paramètres de requête, des corps de requête JSON et des codes d'état HTTP, pour m'assurer que l'API renvoie les résultats attendus.

En outre, Postman peut être utilisé pour tester des routes qui nécessitent une authentification. Je peux inclure les informations d'identification requises dans la requête et m'assurer que l'API les vérifie correctement avant d'autoriser l'accès à la ressource demandée.

## Autorisation des CORS Policy

```
// CORS
app.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content, Accept, Content-Type, Authorization"
  );
  res.setHeader(
    "Access-Control-Allow-Methods",
    "GET, POST, PUT, DELETE, PATCH, OPTIONS"
  );
  next();
});
```

Les navigateurs modernes ont une politique de sécurité stricte appelée "Same-Origin Policy", qui limite les requêtes entre différents domaines. Cela signifie que si j'essaie d'effectuer une requête depuis mon serveur vers un autre domaine, cela pourrait être bloqué par le navigateur.

Pour résoudre ce problème, j'utilise une technique appelée "CORS" (Cross-Origin Resource Sharing), qui me permet de spécifier les domaines qui sont autorisés à accéder à mes ressources. Dans ce code, j'utilise une fonction middleware Express pour définir les en-têtes CORS dans les réponses de mon serveur.

# Développement du Panel d'administration

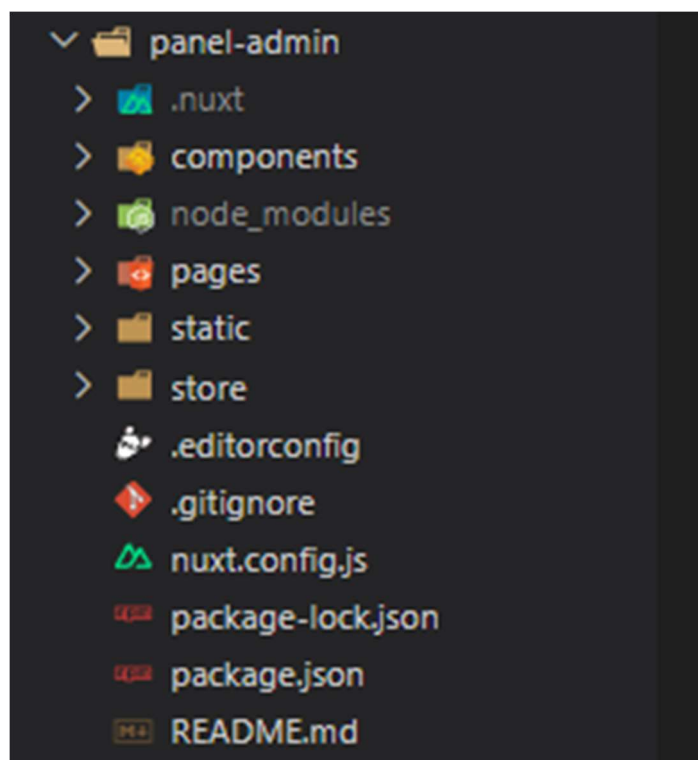
## Création du panel d'administration

J'ai créé un panel d'administration avec Nuxt.js pour gérer l'application de chat TissApp. L'objectif était de permettre aux administrateurs de se connecter et de modifier ou supprimer les profils des utilisateurs ainsi que de supprimer les messages sur le tchat général de l'application.

En utilisant les composants de Nuxt.js, j'ai créé une interface utilisateur simple et efficace qui permet aux administrateurs de naviguer facilement dans le panel d'administration. J'ai également mis en place des fonctionnalités de sécurité pour protéger les données sensibles, telles que l'authentification des administrateurs et la gestion des autorisations.

Pour sécuriser l'application, j'ai utilisé des technologies telles que JSON Web Tokens (JWT) et des middleware Express. Cela m'a permis de garantir que seuls les utilisateurs autorisés ont accès aux fonctionnalités de gestion des utilisateurs et des messages.

Les administrateurs de TissApp peuvent désormais gérer efficacement les utilisateurs et les messages sur leur plateforme de chat.



# Utilisation de Nuxt.js

J'ai choisi d'utiliser Nuxt.js pour plusieurs raisons. Tout d'abord, Nuxt.js est basé sur Vue.js, un framework JavaScript populaire et très performant. Vue.js offre une approche basée sur les composants, ce qui facilite la création d'interfaces utilisateur réactives et modulaires. En utilisant Nuxt.js, j'ai pu bénéficier de la puissance de Vue.js tout en ajoutant des fonctionnalités spécifiques au développement d'applications web.

Ensuite, Nuxt.js fournit une architecture solide pour le développement d'applications universelles. Il prend en charge le rendu côté serveur (SSR) et le pré-rendu statique, ce qui permet d'améliorer considérablement les performances de l'application. Avec Nuxt.js, j'ai pu créer des applications qui se chargent rapidement, offrant une expérience utilisateur fluide, notamment en optimisant le temps de chargement initial et en améliorant le référencement des pages.

De plus, Nuxt.js offre des fonctionnalités avancées telles que le routage automatique. Grâce à cette fonctionnalité, j'ai pu créer des routes dynamiques simplement en organisant mes fichiers dans une structure prédéfinie. Cela a permis de gagner du temps et de maintenir un code clair et bien organisé.

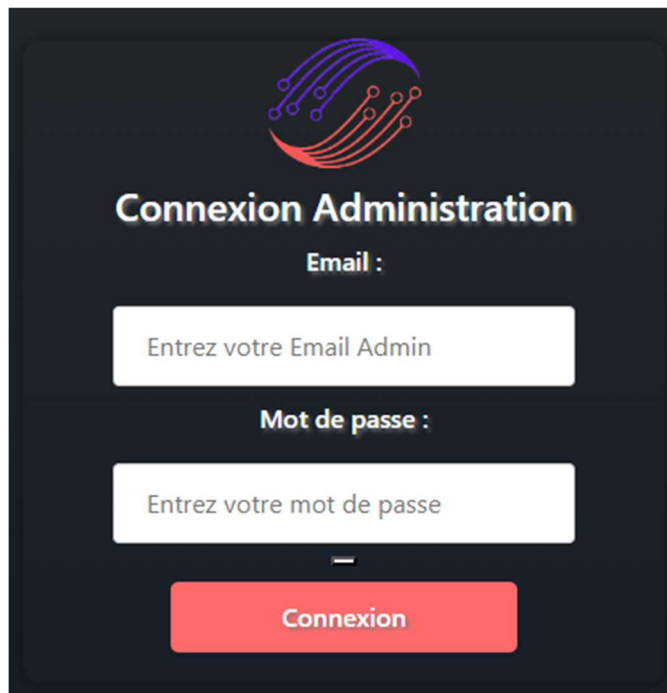
Un autre avantage de Nuxt.js est sa grande flexibilité et sa compatibilité avec un large éventail de bibliothèques et de plugins. J'ai pu intégrer facilement des bibliothèques tierces et des fonctionnalités supplémentaires à mon projet grâce à l'écosystème riche de Nuxt.js.

Enfin, Nuxt.js dispose d'une communauté active et d'une documentation complète. J'ai pu trouver de nombreuses ressources, tutoriels et exemples pour m'aider dans mon apprentissage et résoudre d'éventuels problèmes.





## Page de connexion



Pour permettre aux administrateurs de se connecter, j'ai créé une page de connexion qui prend en compte l'email et le mot de passe des utilisateurs. Les administrateurs sont les seuls utilisateurs ayant un accès complet au panel d'administration. Ainsi, j'ai implémenté une vérification dans la base de données pour s'assurer que seul un utilisateur ayant le champ "admin" à true peut se connecter en tant qu'administrateur.

Une fois qu'un administrateur se connecte avec succès, il peut accéder à toutes les fonctionnalités du panel d'administration, notamment la gestion des utilisateurs et des messages. J'ai utilisé les composants de Nuxt.js pour créer une interface utilisateur claire et facile à utiliser qui permet aux administrateurs de visualiser, modifier ou supprimer les profils des utilisateurs et de supprimer les messages sur le tchat général de l'application TissApp.

En plus de la page de connexion, j'ai mis en place des fonctionnalités de sécurité pour protéger les données sensibles. J'ai utilisé des technologies telles que JSON Web Tokens (JWT) pour générer des jetons d'authentification qui garantissent que les administrateurs connectés ont accès uniquement aux fonctionnalités de gestion de l'application TissApp. De plus, j'ai utilisé des middleware Express pour contrôler l'accès aux routes du panel d'administration et garantir que seuls les administrateurs connectés peuvent accéder aux fonctionnalités de gestion.

## Page de gestion des utilisateurs

Email	Rôle	Date de création	Status	Actions
Mokaddem_s@hotmail.fr	Administrateur	dimanche 7 mai, 23:10:10	En-ligne ●	
mokaddem@go.com	Utilisateur	lundi 8 mai, 02:51:18	Hors-ligne ●	<button>Modifier</button> <button>Supprimer</button>
mokaddem@go.comss	Utilisateur	lundi 8 mai, 00:59:31	Hors-ligne ●	<button>Modifier</button> <button>Supprimer</button>
mokaddem-s@hotmail.com	Utilisateur	lundi 8 mai, 14:52:29	En-ligne ●	<button>Modifier</button> <button>Supprimer</button>

J'ai créé une page d'administration des utilisateurs qui permet aux administrateurs de visualiser et de gérer les profils des utilisateurs de TissApp.

En utilisant les composants de Nuxt.js, j'ai créé une interface utilisateur efficace qui récupère la liste de tous les utilisateurs enregistrés dans la base de données. Les administrateurs peuvent facilement visualiser les informations des utilisateurs, notamment leur email, leur prénom et leur mot de passe.

Les administrateurs peuvent également modifier les informations des utilisateurs. En cliquant sur le bouton de modification, les administrateurs peuvent mettre à jour les champs d'email, de prénom et de mot de passe des utilisateurs.

Pour garantir la sécurité et l'intégrité des données, seuls les administrateurs avec le champ "admin" à true peuvent modifier les informations des utilisateurs. Les administrateurs avec un champ "admin" à false ne peuvent pas effectuer de modifications sur les profils des utilisateurs.

En plus de la modification des profils des utilisateurs, les administrateurs peuvent également supprimer les utilisateurs. Pour supprimer un utilisateur, les administrateurs cliquent simplement sur le bouton de suppression correspondant à l'utilisateur concerné.

Grâce à cette page d'administration des utilisateurs, les administrateurs de TissApp peuvent facilement visualiser et gérer les profils des utilisateurs. Avec des fonctionnalités de sécurité en place, seuls les administrateurs autorisés peuvent effectuer des modifications sur les profils des utilisateurs, garantissant ainsi la sécurité et l'intégrité des données.

## Explication de la logique DELETE Admin = true

```
// REQUEST DELETE USER
async deleteUser(userId) {
  try {
    const data = await fetch(`http://localhost:3100/api/auth/users-delete/${userId}`, {
      method: 'DELETE',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${localStorage.getItem('token')}`
      }
    });
    const response = await data.json();
    console.log("success delete user");
    console.log(response);
    this.getUsers();
  } catch (error) {
    console.log("catch delete user");
    console.log(error);
    this.errorMessage = error.message;
  }
},
```

J'ai créé une fonction appelée **deleteUser(userId)** qui me permet de supprimer un utilisateur en envoyant une requête **DELETE**. Lorsque j'appelle cette fonction, j'envoie une demande à l'URL **http://localhost:3100/api/auth/users-delete/\${userId}** en utilisant la méthode **DELETE**.

Pour cela, j'utilise l'objet **fetch** qui me permet d'effectuer des requêtes HTTP. Cette fonction prend en paramètre l'**ID** de l'utilisateur que je souhaite supprimer.

Ensuite, j'attends la réponse de la requête en utilisant **await data.json()**, ce qui me permet d'obtenir les données renvoyées par le serveur au format **JSON**. Une fois que j'ai les données de la réponse, je les affiche dans la console en utilisant **console.log(response)**.

Si la suppression de l'utilisateur réussit, j'affiche un message de succès dans la console avec **console.log("success delete user")**. Ensuite, j'appelle la fonction **getUsers()** pour mettre à jour la liste des utilisateurs après la suppression.

En cas d'erreur lors de la requête, j'affiche un message d'erreur dans la console avec **console.log(error)**. De plus, j'assigne le message d'erreur à la variable **errorMessage** pour pouvoir l'afficher à l'utilisateur.

```
// CTRL ROUTE ADMIN DELETE USER
exports.deleteUserAccount = async (req, res, next) => {
  try {
    const user = req.user.admin
      ? await User.findOne({ where: { id: req.params.id } })
      : req.user;
    await user.softDestroy();
    res.status(200).json({ message: "Compte supprimé" });
  } catch (error) {}
  res.status(400).json({ error });
};
```

Dans cette route j'utilise le contrôleur **deleteUserAccount** gère la suppression d'un compte utilisateur. Lorsqu'une requête **DELETE** est effectuée, cette fonction est appelée. Si l'utilisateur est un administrateur, elle recherche l'utilisateur cible en utilisant l'ID fourni dans les paramètres de la requête. Ensuite, elle utilise la méthode **softDestroy()** pour marquer le compte comme supprimé dans la base de données. Si la suppression réussit, une réponse avec le **statut 200** et un message indiquant que le compte a été supprimé est renvoyée. En cas d'erreur, une réponse avec le **statut 400** et les détails de l'erreur est renvoyée. En résumé, ce contrôleur permet de supprimer des comptes utilisateurs en fonction des permissions de l'utilisateur qui effectue la demande.

## Page de gestion du chat général

Chat Panel					
ID	Nom d'utilisateur	Image	Message	Date de création	Action
21	Samir Mokaddem Mokaddem_s@hotmail.fr		Test2	jeudi 11 mai, 13:16:05	Supprimer
20	Samir Mokaddem Mokaddem_s@hotmail.fr		Test1	jeudi 11 mai, 13:15:58	Supprimer

En plus de la gestion des utilisateurs, j'ai également créé une page d'administration du chat général de TissApp. Cette page permet aux administrateurs de visualiser et de gérer les messages qui ont été envoyés dans le chat général de l'application.

En utilisant les composants de Nuxt.js, j'ai créé une interface utilisateur claire qui affiche tous les messages qui ont été envoyés dans le chat général. Les messages sont affichés avec des informations sur l'utilisateur qui a posté le message, la date de création du message et le contenu du message.

Les administrateurs peuvent facilement visualiser les messages et, s'ils le souhaitent, peuvent supprimer les messages en cliquant simplement sur le bouton de suppression correspondant au message concerné. Seuls les administrateurs avec le champ "admin" à true peuvent supprimer les messages du chat général.

Pour garantir la sécurité et l'intégrité des données, j'ai également implémenté des fonctionnalités de sécurité pour cette page. Seuls les administrateurs connectés peuvent accéder à la page d'administration du chat général, et seuls les administrateurs avec le champ "admin" à true peuvent supprimer les messages.

## Conclusion

En conclusion, le projet TissApp était une expérience passionnante et enrichissante pour moi. En collaboration avec mes collègues étudiants, nous avons créé une application mobile TissApp pour permettre aux utilisateurs de communiquer facilement via un chat en temps réel.

En utilisant Node.js et Express pour le back-end, ainsi que React Native pour le front-end, nous avons créé une application robuste et conviviale qui a été bien accueillie par les utilisateurs.

Ce que j'ai particulièrement apprécié dans ce projet, c'était le développement du back-end en utilisant Node.js et Express pour gérer les requêtes et les données de l'application. J'ai pu acquérir des compétences précieuses en matière de conception et de développement de l'API, ainsi qu'en matière de gestion de la sécurité et de la performance de l'application.

En utilisant les technologies de pointe pour le développement de l'API, j'ai appris à créer des API robustes et évolutives qui peuvent gérer les demandes d'un grand nombre d'utilisateurs. J'ai également appris à intégrer des fonctionnalités de sécurité, telles que la gestion des autorisations d'accès et la protection contre les attaques malveillantes.

Grâce à ce projet, j'ai également acquis une expérience pratique dans la collaboration avec une équipe de développement, la communication et la gestion de projet. Ces compétences sont essentielles pour réussir dans le domaine du développement de logiciels, et je suis heureux de les avoir acquises grâce à cette expérience.

En somme, le projet TissApp était une expérience incroyablement enrichissante qui m'a permis d'acquérir des compétences précieuses dans le développement de logiciels, la collaboration en équipe et la gestion de projet. Je suis reconnaissant d'avoir eu cette opportunité et j'ai hâte de continuer à développer mes compétences dans ce domaine passionnant.

