WORKUP™

Implémentation d'une plateforme de réseau social d'entreprise

---

Projet réalisé dans le cadre de la présentation au
Titre Professionnel Développeur Web et Web Mobile

SIMPLON.CO

présenté par

Walid BAHIJ

Simplon - Promotion 2024

SOMMAIRE

SOMMAIRE **2 - 3**
INTRODUCTION **4**
LISTE DES COMPÉTENCES COUVERTES PAR LE PROJET **5**

I. Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité

II. Développer la partie back-end d'une application web ou web mobile en intégrant les recommandation de sécurité

CAHIER DES CHARGES **8**

I. Besoins et objectifs de l'application 8
A. *Besoins* *9*
B. Objectifs *10*

MÉTHODOLOGIE DE DÉVELOPPEMENT **11**

I. Choix de l'architecture 12
II. Choix des technologies utilisées 13

MISE EN ŒUVRE DU PROJET **14**

I. Back-end 14

A. Diagramme ERD et relations entre les entités *14*
B. Connexion à la base de données avec Sequelize *15*
C. Serveur Node.js avec Express.js 16
D. Models 21
E. Middlewares 25

F. Controller 29
G. Routes 37
H. Test Unitaire 42
I. Docker 45

DIFFICULTÉS RENCONTRÉES ET LES SOLUTIONS APPORTÉES 47
Difficultés rencontrées et solutions apportées

CONCLUSION 49

INTRODUCTION

Depuis toujours, les technologies et la programmation ont éveillé ma curiosité. Bien que mon parcours initial m'ait éloigné de cet univers, j'ai continué à explorer et à approfondir ces sujets par moi-même. Cette démarche autodidacte m'a permis de développer des compétences techniques et de rester connecté aux évolutions du secteur, tout en cultivant ma passion pour le développement.

Avec le temps, j'ai ressenti le besoin de donner un nouveau souffle à ma carrière et de me tourner vers un domaine qui me passionne réellement. C'est dans cet esprit que j'ai rejoint Simplon, une formation où la pratique occupe une place centrale, permettant de transformer mes compétences en véritables atouts professionnels.

Dans le cadre de cette formation, j'ai réalisé **WorkUup**, un projet personnel qui me tenait à cœur. **WorkUup** est une application de réseau social destinée aux entreprises, conçue pour améliorer la communication interne et encourager les échanges d'idées. Cette plateforme permet à chaque employé de partager des réflexions ou des suggestions pour rendre l'environnement de travail plus agréable et productif.

Ce projet m'a offert l'opportunité d'explorer l'ensemble du processus de développement, de la conception à la réalisation. Il m'a permis d'affiner mes compétences techniques tout en abordant des aspects importants comme l'ergonomie, l'architecture logicielle et la résolution de problèmes. Travailler seul sur **WorkUup** a également renforcé mon autonomie et ma capacité à structurer un projet ambitieux du début à la fin.

En somme, la création de **WorkUup** a été une étape déterminante dans mon parcours. Elle m'a permis de combiner ma passion pour le développement avec une approche concrète, tout en me préparant à relever de nouveaux défis dans le domaine du numérique.

LISTE DES COMPÉTENCES COUVERTES PAR LE PROJET

I. Développer la partie front-end d’une application web ou web mobile en intégrant les recommandations de sécurité

- ***Maquetter une application***

En amont de la réalisation à proprement parler de mon application, j’ai choisi dans un premier temps de travailler sur la réalisation des documents de conception. Dans ce but, j’ai donc réalisé un wireframes du projet, aux formats mobile et desktop pour imaginer la structuration de mes pages, puis enfin j’ai réalisé une charte graphique et des maquettes pour voir concrètement à quoi allait ressembler notre application.

- ***Réaliser une interface utilisateur web statique et adaptable***

Au vu des besoins identifiés en amont de la réalisation du projet, il était impératif que mon application dispose d’une interface dynamique et fluide pour offrir une expérience utilisateur optimale. Cependant, en raison de contraintes de temps, j'ai opté pour une approche plus simple mais efficace, en utilisant principalement du JavaScript pur, accompagné de HTML et CSS. L’utilisation de Bootstrap a également permis d'accélérer le développement en offrant des composants préconçus, tout en garantissant une mise en page responsive et un design moderne. Cette solution m'a permis de répondre aux besoins fonctionnels du projet tout en respectant les délais impartis.

- ***Développer une interface utilisateur web dynamique***

Dans mon projet, j'ai développé une interface utilisateur web moderne et dynamique. En utilisant **HTML** pour la structure, **CSS** (Bootstrap) pour le design responsive, et ***JavaScript*** pour l'interactivité, j'ai créé une application fluide et performante. J'ai manipulé le DOM, gérer des événements avec addEventListener, et conçu une interface intuitive qui s'adapte parfaitement aux différents appareils, offrant une expérience utilisateur optimale.

II. Développer la partie back-end d’une application web ou web mobile en intégrant les recommandations de sécurité

- *Créer un diagramme (ERD/UML)*

Pour structurer les données de mon projet, j’ai d’abord réfléchi aux informations essentielles à intégrer. J’ai ensuite créé un **diagramme ERD** (Entité-Relation) pour visualiser les relations entre les entités et garantir une organisation cohérente (voir page 12) .

- *Créer une base de donnée*

Pour gérer la base de données, j’ai choisi d’utiliser **MySQL,** un outil performant et largement utilisé dans le domaine du développement web. Cette décision m’a permis de transformer facilement les schémas de mon diagramme ERD en une structure relationnelle robuste. En travaillant seul, j’ai pris le temps de tester et de valider chaque étape, depuis la conception jusqu’à l’implémentation, tout en veillant à ce que la base de données soit évolutive et fiable .

- *Développer les composants d’accès aux données*

Dans le cadre de mon projet, les composants d'accès aux données sont les éléments qui permettent de communiquer avec la base de données. Pour ce faire, j’ai utilisé Sequelize comme ORM (Object Relational Mapping), facilitant la gestion des interactions avec la base de données MySQL. Sequelize permet de définir les modèles de données et d'effectuer des opérations CRUD (création, lecture, mise à jour, suppression) sur ces modèles.Ce processus nous permet de gérer de manière fluide les évolutions de la base de données tout au long du développement de l'application .

- *Développer la partie back-end d’une application web ou web mobile*

Pour réaliser le back-end de mon application, j'ai utilisé Node.js avec le framework Express. J'ai structuré l'application avec des routes et des Controllers pour gérer la logique des fonctionnalités. J'ai également mis en place des middleware pour l'authentification des utilisateurs et la validation des données. Cela permet de sécuriser l'accès aux routes et de garantir la cohérence des données envoyées, tout en assurant une architecture claire et modulaire .

RÉSUMÉ DU PROJET

- ***WorkUup*** *est un réseau social d'entreprise que j'ai développé dans le cadre de mon projet de fin de formation.*

- *L'application permet aux employés de partager des idées, des réflexions et des suggestions pour améliorer l'environnement de travail.*

- *Elle offre un espace collaboratif où chaque utilisateur peut publier des posts, commenter ceux des autres et participer à des discussions constructives pour favoriser la collaboration au sein de l'équipe.*

- *Avec des fonctionnalités simples mais efficaces,* ***WorkUup*** *vise à renforcer la communication interne, à valoriser les contributions de chacun et à améliorer la cohésion entre les membres de l'entreprise.*

- *Ce projet s'inscrit dans une démarche visant à transformer les interactions professionnelles et à promouvoir un environnement de travail plus participatif et inspirant.*

CAHIER DES CHARGES

I. Contexte, besoins et objectifs de l’application

Contexte

- Détection d'un ralentissement de la productivité il y a 6 mois
- Baisse constatée de la motivation et de l'implication des employés
- Création d'un comité de pilotage sur le bien-être au travail
- Objectif : Améliorer la communication et l'ambiance entre collègues
- 1 employé sur 3 souhaite améliorer l'ambiance entre collègues
- 1 employé sur 4 veut plus d'échanges entre départements et filiales internationales

A - *Besoins*

Page de Connexion *:*

- Connexion minimale requise
- Authentification par :
    - Email de l'employé
    - Mot de passe
- Fonctionnalités supplémentaires :
    - Possibilité de créer un compte
    - Déconnexion possible
    - Persistance de la session
    - Sécurisation des données de connexion

Page d'Accueil :

- Affichage des posts
- Ordre antéchronologique (du plus récent au plus ancien)

Fonctionnalités des Posts

- **Création de posts** :

Contenu possible :

    - Texte
    - Image (GIF)

Options de gestion :

    - Modification de ses propres posts
    - Suppression de ses propres posts

Système de Commentaire

- Possibilité de commenter les posts des autres utilisateurs

Rôle Administrateur

- Droits étendus de modification/suppression sur tous les posts/users
- Communication des identifiants administrateur

Spécifications Techniques

Back-end

- Base de données : MongoDB ou MySQL
- Schéma UML/ERD requis
- ODM/ORM : Mongoose ou Sequelize
- Technologies :
    - Node.js
    - Express
- Sécurité et gestion des rôles
- Tests unitaires (Jest)

Frontend

- Frameworks possibles :
    - **Vue.js**
    - **React**
    - **Angular**
- Frameworks CSS :
    - **Bootstrap**

Contraintes Supplémentaires

- Projet CRUD (Create, Read, Update, Delete)
- Mobile friendly
- Le frontend sert de support pour le projet CCP2

Recommandations

- Minimiser les informations demandées à la connexion
- Assurer la sécurité des données
- Créer une expérience utilisateur fluide et intuitive
- Permettre une communication informelle entre employés

B - *Objectifs*

Objectif Principal

Améliorer la communication et la motivation des employés en créant un réseau social interne moderne qui facilite les interactions informelles entre collègues.

Objectifs Spécifiques

1. **Amélioration de la Communication**
    - Favoriser les échanges entre différents départements
    - Réduire les barrières hiérarchiques et géographiques
    - Créer un espace de dialogue informel et convivial
2. **Renforcement de la Cohésion**
    - Permettre aux employés de mieux se connaître
    - Développer un sentiment d'appartenance à l'entreprise
    - Stimuler la motivation et l'implication des collaborateurs
3. **Optimisation de la Productivité**
    - Faciliter le partage d'informations
    - Créer des opportunités de collaboration spontanée
    - Réduire l'isolement professionnel

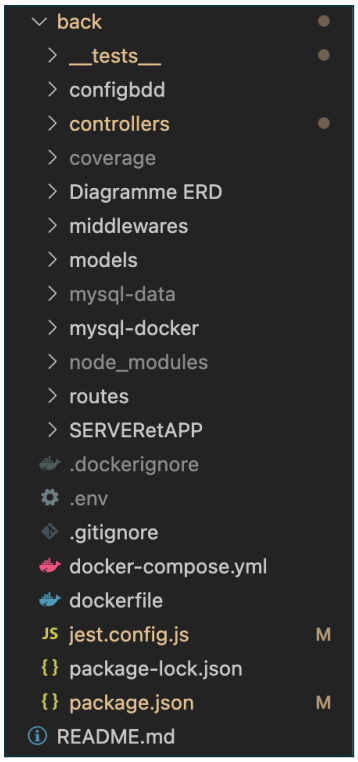
Bénéfices Attendus

1. Augmentation de la satisfaction au travail
2. Amélioration de l'ambiance professionnelle
3. Réduction du turnover
4. Création d'une culture d'entreprise plus collaborative

MÉTHODOLOGIE DE DÉVELOPPEMENT

Dans le cadre de ce projet, une méthodologie de développement basée sur les principes de l'agilité a été adoptée. Cela a permis de structurer le travail en itérations courtes et de garantir une flexibilité et une réactivité accrues face aux évolutions du projet. Le développement a été divisé en plusieurs phases, en commençant par une analyse des besoins métier, suivie par la conception technique et la mise en œuvre.

Le projet a été développé en suivant une architecture MVC (Model-View-Controller), permettant ainsi une séparation claire des responsabilités entre les différentes couches de l'application, ce qui a facilité le développement et la maintenance du code.



I. Choix de l'architecture

Le choix de l'architecture MVC (Model-View-Controller) pour le développement de l'application a été motivé par plusieurs raisons techniques et organisationnelles. Tout d'abord, MVC permet une séparation claire des responsabilités entre les différentes parties de l'application : **Model**, **View** et **Controller**.

Cette séparation facilite la gestion du code, rend les différentes parties de l'application plus modulaires et améliore la maintenabilité du projet à long terme.

Le **Model** gère la logique des données, les **Views** se concentrent sur la présentation des informations à l'utilisateur, et les **Controllers** gèrent l'interaction entre les deux.

Ce découplage améliore également les possibilités de test et d'évolution de chaque composant de manière indépendante.

L'architecture MVC est couramment utilisée dans le développement web en raison de sa modularité et de l'abondance d'outils et de frameworks qui facilitent l'intégration de bonnes pratiques. Elle permet de gérer efficacement les fonctionnalités d'un projet de taille intermédiaire à grande échelle tout en offrant flexibilité et évolutivité. De plus, elle facilite la collaboration entre les développeurs front-end et back-end, chacun étant responsable de différentes parties, ce qui optimise la productivité, la maintenabilité et la robustesse du système

II. Choix des technologies utilisées

Le projet repose sur un ensemble de technologies et de bibliothèques largement utilisées dans le développement web, certaines ayant été abordées au cours de ma formation. **Bcryptjs** et **jsonwebtoken** (JWT) ont été utilisés pour gérer respectivement la sécurisation des mots de passe et l'authentification des utilisateurs via des tokens **JWT,** des notions étudiées en cours pour renforcer la sécurité des applications web.

Multer a été intégré pour la gestion des téléchargements de fichiers, facilitant l'upload d'images et autres fichiers dans l'application, une fonctionnalité souvent nécessaire dans les applications modernes. D'autres bibliothèques comme **Dotenv** ont permis de gérer les variables d'environnement, tandis que **Cors** a été configuré pour gérer les autorisations de partage de ressources entre différents domaines.

Enfin, **Sequelize** a été utilisé comme ORM pour interagir avec la base de données **MySQL**, optimisant les requêtes et les interactions avec les données tout en simplifiant le processus de développement. Pour garantir la qualité du code et la fiabilité des fonctionnalités, j'ai également intégré **Jest** dans le projet pour effectuer des tests unitaires et d'intégration.

npm a été utilisé pour gérer les dépendances du projet, y compris l'installation de Jest pour la mise en place des tests.

Ces technologies ont été choisies pour leur efficacité et leur compatibilité avec l'architecture du projet, et ont été abordées en cours pour leur pertinence dans la création d'applications sécurisées et performantes.

MISE EN ŒUVRE DU PROJET

I. Back-end

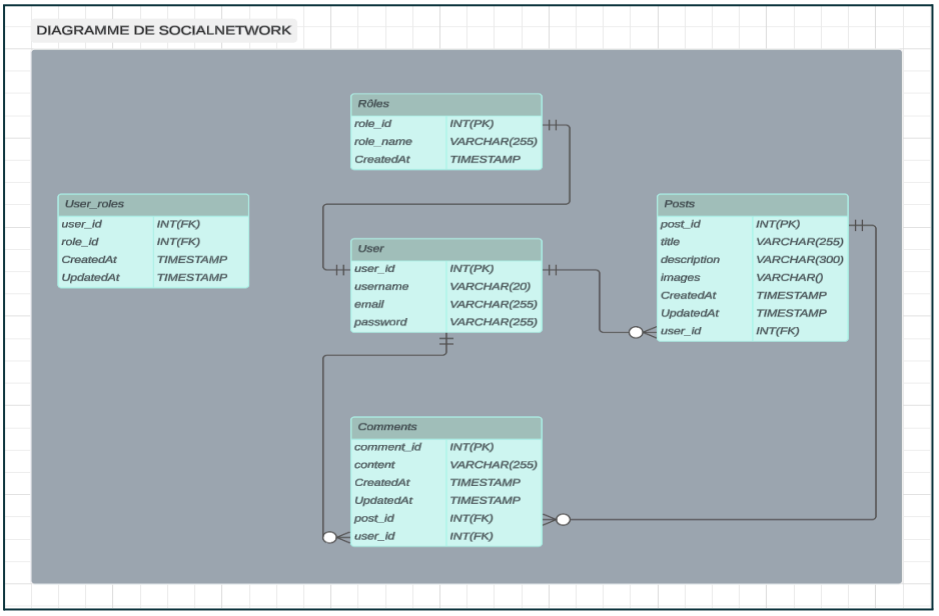
A. Diagramme ERD et relations entre les entités

Le développement du back-end commence par une étape cruciale : la conception de la base de données. Pour ce faire, un diagramme ERD (Entity Relationship Diagram) a été conçu pour visualiser les différentes entités de la base de données ainsi que les relations qui les unissent. Le diagramme ERD permet de mieux comprendre la structure de la base de données et de garantir que toutes les informations nécessaires sont correctement stockées et interconnectées.

Dans le cadre de ce projet, plusieurs entités principales ont été identifiées : User, Posts, Comments, Roles, et user_roles.

- Users contient des informations sur les utilisateurs (nom, email, mot de passe, rôle, etc.).
- Posts représente les publications des utilisateurs, avec des attributs comme le titre, le contenu, et la date de création.
- Comments est lié aux Posts et permet aux utilisateurs d'ajouter des commentaires sur les publications.
- Roles gère les différents types d'utilisateurs (admin, utilisateur standard, etc.).
- user_roles est une table de jointure qui relie les utilisateurs aux rôles, permettant une relation many-to-many.

Les relations définies dans le diagramme sont principalement de type one-to-many (un utilisateur peut avoir plusieurs posts et commentaires) et many-to-many (un utilisateur peut avoir plusieurs rôles). Ce diagramme a servi de base pour la création de la base de données et la configuration des modèles dans Sequelize, un ORM permettant de manipuler la base de données de manière simple et sécurisée.



Après avoir conçu mon diagramme, j'ai initialisé un projet Node.js en utilisant la commande `npm init` dans le répertoire de mon projet. Cette commande m'a permis de créer un fichier `package.json`, qui est essentiel pour gérer les dépendances, les scripts et les configurations de mon projet. Ce fichier permet également de spécifier les informations du projet, comme son nom, sa version, sa description, ainsi que le point d'entrée du programme.

Une fois le projet initialisé, j'ai installé toutes les dépendances nécessaires pour que mon application fonctionne correctement. Pour cela, j'ai utilisé la commande `npm install`, suivie des bibliothèques que j'avais choisies pour mon projet. Parmi ces bibliothèques, on retrouve des outils comme **Sequelize** pour l'ORM, dotenv pour la gestion des variables d'environnement, **CORS** pour la gestion des autorisations de partage de ressources entre domaines, et d'autres dépendances comme **Bcryptjs** et **JSON Web Token** (JWT) pour gérer la sécurité des mots de passe et l'authentification des utilisateurs.

Fichier `package.json`

Ces dépendances ont été ajoutées au fichier `package.json` et téléchargées dans le dossier `node_modules`, qui contient toutes les bibliothèques et modules nécessaires au bon fonctionnement de l'application. Cette étape d'installation des dépendances est cruciale pour s'assurer que l'application dispose de tous les outils nécessaires avant de pouvoir se connecter à la base de données ou de commencer à implémenter les fonctionnalités de l'application.

B. *Connexion à la base de données avec Sequelize*

Une fois que le projet a été initié et après avoir installé les dépendances , la prochaine étape a été de connecter l'application au système de gestion de base de données MySQL. Une base de données a d'abord été créée en amont dans MySQL pour accueillir les différentes tables et les données de l'application.

Sequelize a ensuite été choisi pour son efficacité en tant qu'**ORM** (Object-Relational Mapping) pour gérer les entités de la base de données en JavaScript. Ce choix a permis de simplifier les requêtes SQL tout en offrant une interface plus intuitive pour manipuler les données.

Dans la configuration de Sequelize, des fichiers de connexion ont été créés pour inclure les informations nécessaires à la connexion à la base de données (nom, utilisateur, mot de passe, hôte) via des variables d'environnement stockées dans un fichier .env. Grâce à **dotenv**, ces informations sensibles ont été sécurisées et n'ont pas été codées en dur dans le projet.

```
const { Sequelize } = require('sequelize');
require('dotenv').config({ path: '../.env' });

const sequelize = new Sequelize(process.env.DB_NAME, process.env.DB_USER, process.env.DB_PASSWORD, {
  host: 'localhost', //mettre "mysql" pour tourner docker
  dialect: 'mysql',
  port: 3306,
});



module.exports = sequelize;
```

Cette configuration permet à l'application de se connecter à la base de données MySQL et d'effectuer des opérations de lecture et d'écriture sur celle-ci.

C. Serveur Node.js avec Express.js

Le projet utilise l'environnement **Node.js** pour le serveur back-end, et **Express.js** a été choisi comme framework pour sa simplicité et son efficacité dans la gestion des requêtes HTTP. Le fichier `server.js` a été utilisé pour initialiser et configurer le serveur, tandis que `app.js` a été utilisé pour organiser les différentes parties de l'application (comme la gestion des routes et des middlewares).

- Fichier `server.js`

```
"use strict";
require("dotenv").config();

// Configure et démarre le serveur avec Node.js et Express.…

const http = require("http");
const app = require("./app");
const dbConfig = require("../configbdd/db");

// Je normalise le port
const normalizePort = (val) => {…
};

const port = normalizePort(process.env.PORT || "3000"); // Je définit le port
app.set("port", port); // Je définit le port dans l'application Express

// Je crée le serveur HTTP avec l'application Express
const server = http.createServer(app);

```

```
// Test la connexion grâce a la fonction authenticate
dbConfig
  .authenticate()
  .then(() => {
    console.log("Connexion à la base de données réussie.");
  })
  .catch((err) => {
    console.error("Impossible de se connecter à la base de données :", err);
  });

module.exports = server;
```

Ce code sert à configurer et démarrer un serveur avec Node.js et Express, tout en gérant la connexion à la base de données MySQL via Sequelize. Il inclut plusieurs étapes clés pour assurer la bonne mise en place de l'application.

1. Chargement des variables d'environnement

Le projet utilise un fichier .env pour stocker des informations sensibles telles que les identifiants de la base de données. Cela permet de sécuriser des données comme les mots de passe, les noms de base de données, ou d'autres informations de configuration sans les inclure directement dans le code.

2. Création du serveur

Le serveur HTTP est créé à l'aide de Node.js et de la bibliothèque Express. Express gère les routes et la logique de l'application, tandis que le serveur HTTP permet d'écouter les requêtes entrantes. Le port d'écoute est configuré en fonction d'une variable d'environnement, ou défini à 3000 par défaut si aucune valeur n'est fournie.

3. Gestion des erreurs

Le code inclut une gestion des erreurs lors de l'écoute du serveur. Si une erreur survient, comme un manque de permissions pour utiliser un port ou si le port est déjà utilisé par un autre processus, elle est capturée et un message d'erreur approprié est affiché.

4. Connexion et synchronisation de la base de données

L'application se connecte à la base de données MySQL via Sequelize, un ORM (Object-Relational Mapping) permettant de faciliter les interactions avec la base de données en JavaScript. La méthode sync est utilisée pour synchroniser les modèles de la base de données, créant les tables nécessaires si elles n'existent pas. Cela permet de s'assurer que la structure de la base de données est toujours à jour.

5. Vérification de la connexion à la base de données

Avant de démarrer le serveur, le code teste la connexion à la base de données via la méthode `authenticate` de Sequelize. Si la connexion réussit, un message de confirmation est affiché. En cas d'échec, un message d'erreur est généré, ce qui permet de détecter rapidement tout problème lié à l'accès à la base de données.

- Fichier `app.js`

```
const express = require("express");
const app = express();
const router = require("../routes/routes");
const path = require("path");
const fs = require("fs");
const cors = require("cors");

app.use("/uploads", express.static(path.join(__dirname, "uploads")));
app.use(cors());

//pour éviter le blocage
app.options("*", (req, res) => {
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept, Authorization"
  );
  res.setHeader(
    "Access-Control-Allow-Methods",
    "GET, POST, PUT, DELETE, PATCH, OPTIONS"
  );

  res.sendStatus(200);
});

// Middleware pour afficher les logs des requêtes
app.use((req, res, next) => {
  console.log(
    `Requête reçue avec la méthode: ${req.method}, à l'URL: ${req.url}`
  );
  next();
});

// app.use((req, res, next) => {…

const uploadDir = path.join(__dirname, "..", "serveretapp", "uploads");
if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir, { recursive: true });
}

app.use(express.json());

app.use("/api", router);
```

Ce code configure l'application Express et met en place plusieurs fonctionnalités essentielles pour le bon fonctionnement de l'application web.

1. Configuration des routes et des répertoires statiques

L'application utilise Express pour définir les routes et gérer les requêtes HTTP. La première partie du code permet de servir les fichiers présents dans le répertoire `uploads` de manière statique. Cela permet aux utilisateurs d'accéder directement aux fichiers qui y sont stockés, comme des images ou des documents, via l'URL de l'application.

2. Utilisation de CORS

La bibliothèque CORS (Cross-Origin Resource Sharing) est utilisée pour gérer les autorisations de partage de ressources entre différents domaines. Le middleware `cors()` permet de définir des règles de sécurité qui autorisent ou non l'accès à l'API depuis des origines spécifiques. Cela est particulièrement utile pour éviter que le frontend d'un domaine différent ne soit bloqué lorsqu'il tente d'accéder aux ressources du backend.

3. Configuration des en-têtes CORS pour les requêtes OPTIONS

Une partie du code définit des en-têtes de réponse pour les requêtes `OPTIONS`, une méthode HTTP préliminaire utilisée pour interroger les capacités du serveur avant d'envoyer une requête réelle. Cela permet au navigateur de savoir quelles actions sont autorisées, comme les méthodes HTTP acceptées, les en-têtes autorisés, et les origines autorisées. Cette étape est cruciale pour éviter les erreurs de blocage dues aux politiques CORS.

4. Middleware de journalisation des requêtes

Un middleware est utilisé pour enregistrer chaque requête reçue par le serveur. À chaque fois qu'une requête est effectuée, un message est affiché dans la console, précisant la méthode HTTP utilisée (GET, POST, etc.) ainsi que l'URL de la requête. Cela permet de suivre l'activité du serveur et d'identifier rapidement les requêtes effectuées.

5. Vérification et création du répertoire 'uploads'

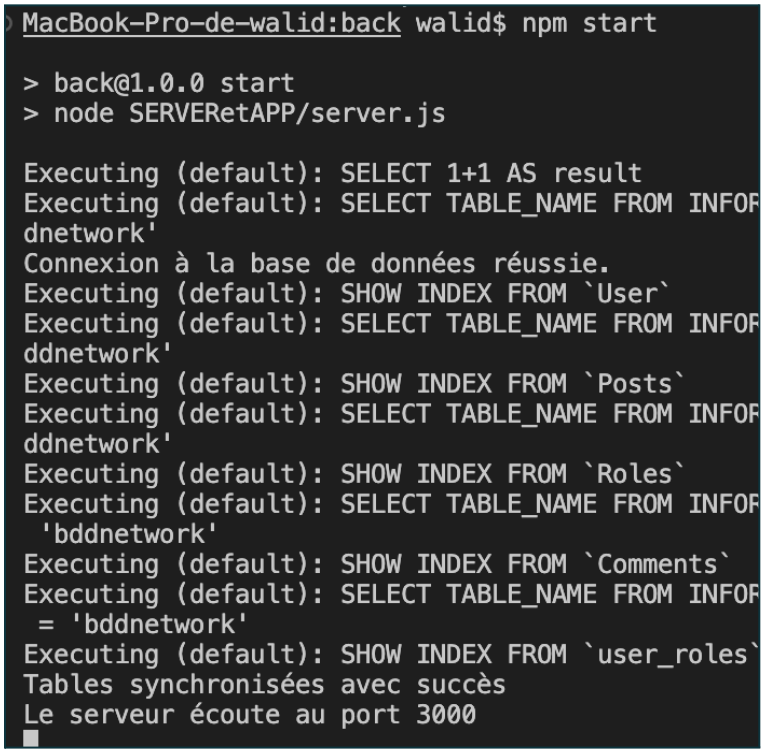
Le code vérifie si le répertoire `uploads` existe sur le serveur. Si ce n'est pas le cas, il crée automatiquement ce répertoire. Cela garantit que l'application peut stocker des fichiers téléchargés sans rencontrer d'erreurs liées à l'absence de ce dossier.

6. Middleware pour gérer les corps de requêtes JSON

L'application est configurée pour traiter les corps des requêtes HTTP envoyées en JSON. Le middleware `express.json()` permet à Express de convertir automatiquement les données JSON envoyées dans les requêtes en objets JavaScript, facilitant ainsi leur utilisation dans l'application.

7. Utilisation du routeur pour gérer les API

Le code configure le routeur d' Express pour gérer les routes de l'API. Toutes les requêtes envoyées à l'URL `/api` sont redirigées vers le fichier de routes, où les logiques de traitement des requêtes sont définies. Ce découpage rend l'application plus modulaire et maintenable.



À cette étape du projet, grâce au fichier de configuration et aux dépendances installées, je peux désormais faire tourner mon serveur et établir la connexion avec la base de données.

```
MYSQL_ROOT_PASSWORD=magenta
MYSQL_DATABASE=bddnetwork
MYSQL_USER=walid
MYSQL_PASSWORD=magenta
DB_HOST=mysql ## ou localhost pour touner en local
DB_PORT=3306
DB_USER=walid
DB_PASSWORD=magenta
DB_NAME=bddnetwork
JWT_SECRET=elbLuckyLuke
```

Voici mon fichier .env, utilisé pour protéger mes données sensibles.

D. Models

Dans le cadre du développement du backend de l'application, la création des modèles est une étape essentielle. Les modèles représentent les tables de la base de données et permettent de structurer et de manipuler les données à l'aide de Sequelize, un ORM (Object-Relational Mapping) qui facilite l'interaction avec la base de données relationnelle MySQL. Dans cette section, nous allons détailler les modèles définis pour le projet, ainsi que leurs relations.

1. Modèle `User`

```
const { DataTypes } = require("sequelize");
const sequelize = require("../configbdd/db")

const User = sequelize.define(
  "User",
  {
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true,
    },

    username: {
      type: DataTypes.STRING(20),
      allowNull: false,
      unique: true,
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: true,
    },
    password: {
      type: DataTypes.STRING,
      allowNull: false,
    },
  },
  {
    tableName: "User",
    modelName: "User",
    timestamps: false,
  }
);

module.exports = User;
```

Le modèle `User` représente les utilisateurs de l'application. Il est défini par plusieurs attributs essentiels :

- **`user_id`** : un identifiant unique pour chaque utilisateur, défini comme clé primaire et auto-incrémentée.
- **`username`** et **`email`** : des champs uniques et non nuls, garantissant que chaque utilisateur dispose d'un nom d'utilisateur et d'une adresse email uniques.
- **`password`** : un champ pour stocker le mot de passe de l'utilisateur, qui sera crypté avant d'être stocké dans la base de données.

L'objectif de ce modèle est de permettre l'enregistrement et la gestion des utilisateurs dans l'application. Il permet également de garantir l'intégrité des données grâce à l'unicité des champs `username` et `email`.

2. **Modèle `Posts`**

```
const { DataTypes } = require("sequelize");
const sequelize = require("../configbdd/db");

// La fonction define permet de définir un modèle représentan
const Posts = sequelize.define(
  "Posts",
  {
    post_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true,
    },
    image: {
      type: DataTypes.STRING,
      allowNull: true,
    },
    title: {
      type: DataTypes.STRING(255),
      allowNull: false,
    },
    description: {
      type: DataTypes.TEXT, // J'utilise DataTypes.TEXT pour
      allowNull: false,
    },
    user_id: {

      type: DataTypes.INTEGER,
      allowNull: false,
      references: {
        model: "Users",
        key: "user_id",
      },
    },
    createdAt: { field: "created_at", type: DataTypes.DATE },
    updatedAt: { field: "updated_at", type: DataTypes.DATE },
  },
  {
    timestamps: true,
  }
);

module.exports = Posts;
```

Le modèle `Posts` permet de représenter les articles ou publications que les utilisateurs partagent dans l'application. Il comprend plusieurs champs importants :

- **`post_id`** : un identifiant unique pour chaque publication, défini comme clé primaire et auto-incrémentée.
- **`title`** : le titre de la publication, qui est une chaîne de caractères obligatoire.
- **`description`** : la description ou le contenu de la publication, qui peut être une chaîne de texte longue.
- **`image`** : un champ optionnel permettant de stocker le chemin d'une image associée à la publication.
- **`user_id`** : une clé étrangère qui permet de lier chaque publication à un utilisateur, représentant ainsi l'auteur de la publication.

Ce modèle est crucial pour gérer les contenus générés par les utilisateurs dans l'application.

3. **Modèle `Comments`**

```
const { DataTypes } = require("sequelize");
const sequelize = require("../configbdd/db");

const Comments = sequelize.define(
  "Comments",
  {
    comment_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    content: {
      type: DataTypes.TEXT,
      allowNull: false,
    },
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
    },
    post_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
    },
  },
  {
    timestamps: true,
    createdAt: "created_at",
    updatedAt: "updated_at",
    tableName: "Comments",
  }
);

module.exports = Comments;
```

Le modèle `Comments` permet de gérer les commentaires laissés par les utilisateurs sur les publications. Il inclut :

- **`comment_id`** : un identifiant unique pour chaque commentaire, qui est une clé primaire et auto-incrémentée.
- **`content`** : le contenu du commentaire, qui est un champ texte non nul.
- **`user_id`** : une clé étrangère pour lier chaque commentaire à un utilisateur spécifique.
- **`post_id`** : une clé étrangère qui associe chaque commentaire à une publication spécifique.

Ce modèle facilite l'interaction entre les utilisateurs de l'application, leur permettant de réagir aux publications des autres.

4. **Modèle `Roles`**

```
const { DataTypes } = require("sequelize");
const sequelize = require("../configbdd/db");

const Comments = sequelize.define(
  "Comments",
  {
    comment_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    content: {
      type: DataTypes.TEXT,
      allowNull: false,
    },
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
    },
    post_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
    },
  },
  {
    timestamps: true,
    createdAt: "created_at",
    updatedAt: "updated_at",
    tableName: "Comments",
  }
);

module.exports = Comments;
```

Le modèle `Roles` permet de définir les rôles des utilisateurs dans l'application, tels que "admin" ou "user". Il comprend :

- **`role_id`** : un identifiant unique pour chaque rôle, défini comme clé primaire et auto-incrémentée.
- **`role_name`** : le nom du rôle (par exemple "admin", "user"), un champ obligatoire.

Ce modèle est essentiel pour la gestion des autorisations et des droits d'accès dans l'application.

5. **Les Relations Entre les Modèles**

```
const { Sequelize } = require("sequelize");
require("dotenv").config({ path: "../.env" });


// Import models
const User = require("./users.model");
const Roles = require("./roles.model");
const Posts = require("./posts.model");
const Comments = require("./comments.model");

// Define associations
Roles.belongsToMany(User, {
  through: "user_roles",
  foreignKey: "role_id",
  otherKey: "user_id",
});

User.belongsToMany(Roles, {
  through: "user_roles",
  foreignKey: "user_id",
  otherKey: "role_id",
  as: "roles",
});

User.hasMany(Posts, { foreignKey: "user_id" });
User.hasMany(Comments, { foreignKey: "user_id" });

Posts.belongsTo(User, { foreignKey: "user_id" });
Posts.hasMany(Comments, { foreignKey: "post_id" });

Comments.belongsTo(User, { foreignKey: "user_id" });
Comments.belongsTo(Posts, { foreignKey: "post_id" });

const ROLES = ["user", "admin"];

module.exports = {
  sequelize,
  User,
  Posts,
  Comments,
  Roles,
  ROLES,
};
```

Les relations entre les différents modèles permettent de structurer les données de manière logique et efficace. Voici les principales relations que j'ai mises en place :

- **User - Posts (One-to-Many)** : Un utilisateur peut publier plusieurs articles, mais chaque article appartient à un seul utilisateur. Cette relation est établie par la clé étrangère `user_id` dans le modèle `Posts`.
- **Posts - Comments (One-to-Many)** : Une publication peut avoir plusieurs commentaires, mais chaque commentaire appartient à une seule publication. La relation est définie par la clé étrangère `post_id` dans le modèle `Comments`.
- **User - Comments (One-to-Many)** : Un utilisateur peut laisser plusieurs commentaires, mais chaque commentaire appartient à un seul utilisateur. Cela est établi par la clé étrangère `user_id` dans le modèle `Comments`.
- **User - Roles (Many-to-Many)** : Un utilisateur peut avoir plusieurs rôles (par exemple, un utilisateur peut être à la fois un "user" et un "admin"). Cette relation est gérée par une table de liaison qui associe les utilisateurs aux rôles via des clés étrangères.

*Le code définit des relations qui permettent de lier les modèles **`User`**, **`Roles`**, **`Posts`**, et **`Comments`** de manière logique. Les utilisateurs peuvent avoir plusieurs rôles, rédiger plusieurs publications, et laisser plusieurs commentaires. De plus, chaque publication peut recevoir plusieurs commentaires. Ces relations sont essentielles pour organiser et manipuler les données dans l'application de manière cohérente.*

6. **Conclusion**

Les modèles sont un élément central dans la structure de la base de données et la gestion des données dans l'application. Grâce à Sequelize, la définition des modèles et la gestion des relations entre eux deviennent des tâches plus simples et plus efficaces. En suivant les bonnes pratiques de modélisation de données, nous garantissons une architecture solide et évolutive pour le projet.

E. Middlewares

Dans ce projet, plusieurs middlewares ont été mis en place pour gérer les interactions entre l'application et les utilisateurs. Ces outils permettent de renforcer la sécurité, de simplifier la gestion des autorisations et d'offrir des fonctionnalités spécifiques, comme l'upload d'images. Voici une présentation des principaux middlewares utilisés.

1. **Middleware de vérification des rôles** : `authorizeRoles`

```
// Middleware pour vérifier les rôles
const authorizeRoles = (...allowedRoles) => {
  return (req, res, next) => {
    if (!req.auth?.roles) {
      return res.status(403).json({
        message: "Roles non définis pour l'utilisateur",
      });
    }

    const hasRole = req.auth.roles.some((role) => allowedRoles.includes(role));

    if (!hasRole) {
      return res.status(403).json({
        message: "Accès refusé, votre rôle ne permet pas cette action.",
      });
    }
    console.log("Rôles requis:", allowedRoles);
    console.log("Rôles utilisateur:", req.auth.roles);

    next();
  };
};
```

Fichier **roleAuth.js**

Le middleware `authorizeRoles` joue un rôle central dans la gestion des autorisations. Il permet de restreindre l'accès à certaines ressources ou fonctionnalités en fonction des rôles attribués à l'utilisateur connecté (comme *admin* ou *user*).

Le fonctionnement est simple :

- Le middleware vérifie si les rôles de l'utilisateur, disponibles dans `req.auth.roles`, correspondent à ceux autorisés pour accéder à une ressource.
- Si l'utilisateur n'a pas le rôle requis, une réponse avec un code HTTP `403` est renvoyée, indiquant un accès refusé.

Cela garantit une gestion fine des droits d'accès, essentielle dans un contexte multi-utilisateur.

2. **Middleware de vérification de la propriété** : `verifyOwnership`

```
const verifyOwnership = (getResource) => {
  return async (req, res, next) => {
    try {
      const resource = await getResource(req);

      if (!resource) {
        return res.status(404).json({ message: "Ressource non trouvée." });
      }

      const isOwner = req.auth.userId === resource.user_id; // Comparaison avec l'utilisateur connecté
      const isAdmin = req.auth.roles.includes("admin");

      if (!isOwner && !isAdmin) {
        return res.status(403).json({ message: "Accès refusé." });
      }

      next(); // Autorisation validée
    } catch (error) {
      console.error("Erreur dans verifyOwnership:", error);
      res.status(500).json({ message: "Erreur serveur" });
    }
  };
};
```

Fichier **roleAuth.js**

Ce middleware est conçu pour valider que l'utilisateur connecté est bien le propriétaire d'une ressource (comme un post ou un commentaire) ou dispose des droits administratifs. Il prend en paramètre une fonction getResource qui retourne la ressource ciblée.

Le processus :

- Récupération de la ressource à partir de la base de données.
- Comparaison de l'user_id de la ressource avec celui de l'utilisateur connecté (req.auth.userId).
- Vérification si l'utilisateur a un rôle administratif (*admin*) pour contourner cette limitation.

En cas de non-conformité, une réponse appropriée est renvoyée (erreur 403 ou 404 si la ressource est introuvable). Ce middleware est indispensable pour protéger les ressources sensibles et prévenir les accès non autorisés.

3. **Middleware de gestion des fichiers : `multer`**

```
const multer = require("multer");
const path = require("path");

// Définir le stockage des fichiers (ici dans le dossier 'uploads')
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, path.join(__dirname, "..", "serveretapp", "uploads")); // Accès au bon dossier
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + path.extname(file.originalname)); // Génère un nom unique pour chaque fichier
  },
});

// Créer une instance de multer avec les options de stockage
const upload = multer({
  storage: storage,
  limits: { fileSize: 5 * 1024 * 1024 }, // Limiter la taille à 5 Mo (par exemple)
  fileFilter: (req, file, cb) => {
    // Filtrer les fichiers pour n'accepter que les images (JPEG, JPG, PNG, GIF)
    const filetypes = /jpeg|jpg|png|gif/;
    const extname = filetypes.test(
      path.extname(file.originalname).toLowerCase()
    );
    const mimetype = filetypes.test(file.mimetype);

    if (extname && mimetype) {
      return cb(null, true);
    } else {
      cb(new Error("Le fichier doit être une image (JPEG, JPG, PNG, GIF)"));
    }
  },
});

module.exports = upload;
```

Fichier **multerUpload.js**

Pour gérer l'upload d'images dans l'application, le middleware `multer` a été configuré.

- **Stockage des fichiers :** Les images sont enregistrées dans un répertoire dédié (`uploads`), avec des noms uniques générés à l'aide de la date actuelle.
- **Filtrage des fichiers :** Seules les images (JPEG, JPG, PNG, GIF) sont acceptées, grâce à une fonction de validation basée sur le type MIME et l'extension.
- **Limitation de taille :** Une taille maximale de 5 Mo par fichier est imposée pour éviter tout abus.

Ce middleware simplifie l'ajout de contenu multimédia tout en garantissant la conformité avec les exigences de l'application.

4. **Middleware d'authentification** : JWT

```
const jwt = require("jsonwebtoken");
// Utilisation de la clé JWT de l'environnement
const safetyKeyJwt = process.env.JWT_SECRET || "safetyKeyJwt";

module.exports = (req, res, next) => {
  try {
    const authHeader = req.headers.authorization;
    if (!authHeader || !authHeader.startsWith("Bearer ")) {
      return res
        .status(401)
        .json({ message: "Token manquant ou mal formaté." });
    }

    const token = authHeader.split(" ")[1];
    const decodedToken = jwt.verify(token, safetyKeyJwt);
    console.log("Token décodé:", decodedToken); // Debug

    if (!Array.isArray(decodedToken.roles)) {
      return res.status(400).json({
        message: "Les rôles dans le token ne sont pas valides.",
      });
    }

    req.auth = {
      userId: decodedToken.userId,
      roles: Array.isArray(decodedToken.roles)
        ? decodedToken.roles
        : [decodedToken.roles],
    };

    console.log("req.auth:", req.auth); // Debug

    next();
  } catch (error) {
    let message = "Token invalide ou manquant.";
    if (error.name === "TokenExpiredError") {
      message = "Le token a expiré.";
    } else if (error.name === "JsonWebTokenError") {
      message = "Le token est invalide.";
    }
    console.error("Erreur JWT:", error); // Debug
    res.status(401).json({ message, error: error.message });
  }
```

Fichier **jwtAuth.js**

Pour sécuriser l'application, un middleware d'authentification utilisant les *JSON Web Tokens* (JWT) a été mis en place.

- **Extraction et validation du token :** Le token est récupéré dans l'en-tête `Authorization` et validé à l'aide de la clé secrète (`JWT_SECRET`).
- **Décodage du token :** Les informations de l'utilisateur, comme son `userId` et ses rôles, sont extraites et ajoutées à l'objet `req.auth`.
- **Gestion des erreurs :** Des messages spécifiques sont renvoyés en cas d'erreur, comme un token expiré ou mal formé.

Ce middleware est essentiel pour protéger les routes sensibles et s'assurer que seules les requêtes authentifiées sont traitées.

Ces middlewares, bien intégrés dans le projet, illustrent l'importance de couches intermédiaires dans un backend bien structuré. Chacun d'eux joue un rôle clé dans la sécurisation et le bon fonctionnement de l'application.

F. Controllers

Après avoir mis en place et expliqué les middlewares essentiels pour sécuriser et structurer mon application, je me suis concentré sur le développement des contrôleurs, et notamment celui dédié à la gestion des utilisateurs, afin de centraliser les opérations telles que l'inscription, la connexion, et la modification des données utilisateur.

I. User

```
const { User, Roles: Role } = require("../models/index");
const { sequelize } = require("../models/index");
const Op = sequelize.Sequelize.Op;
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const safetyKeyJwt = process.env.JWT_SECRET;

exports.deleteUser = async (req, res) => {…
};

exports.signup = async (req, res) => {…
};

exports.login = async (req, res) => {…
};

exports.logout = async (req, res) => {…
};

exports.getAllUsers = async (req, res) => {…
};

exports.updateUser = async (req, res) => {…
};

exports.getUserById = async (req, res) => {…
};
```

Fichier **user.controller.js**

1. **deleteUser**

Description : Supprime un utilisateur par son `id`.
Étapes principales :

1. Récupération de l'`id` de l'utilisateur à partir des paramètres de la requête.
2. Suppression de l'utilisateur dans la base de données avec `User.destroy`.
3. Envoie une réponse en cas de succès ou d'erreur :
    - Si aucun utilisateur n'est trouvé, retourne une erreur 404.
    - Sinon, retourne un message confirmant la suppression.

2. **signup**

Description : Permet à un nouvel utilisateur de s'inscrire.

Étapes principales :

1. Vérifie si l'email et le username existent déjà.
2. Génère un username unique en cas de collision.
3. Hash le mot de passe avec bcrypt pour garantir sa sécurité.
4. Crée un utilisateur dans la base de données.
5. Assigne un rôle :
    - Si des rôles sont spécifiés dans la requête, vérifie et les attribue.
    - Sinon, assigne un rôle par défaut (`user`).
6. Gère les erreurs spécifiques comme les conflits de doublons d'email.

3. **login**

Description : Permet à un utilisateur de se connecter.

Étapes principales :

1. Recherche l'utilisateur par email.
2. Valide le mot de passe avec bcrypt.
3. Génère un JWT contenant l'`id` de l'utilisateur et ses rôles.
4. Retourne le token pour authentifier les futures requêtes.

4. **logout**

Description : Gère la déconnexion.

Étapes principales :

- Retourne simplement une confirmation de déconnexion.

5. **getAllUsers**

Description : Récupérer tous les utilisateurs.
Étapes principales :

1. Utilise `User.findAll` pour récupérer tous les utilisateurs depuis la base de données.
2. Retournez les données ou une erreur en cas de problème.

6. **updateUser**

Description : Met à jour les informations d'un utilisateur existant.

Étapes principales :

1. Récupère l'utilisateur par son `id`.
2. Vérifie l'existence de l'utilisateur.
3. Mettre à jour les champs modifiables comme `username`, `email` ou `password`.
4. Gère la mise à jour des rôles si fourni.
5. Sauvegarde les modifications dans la base de données.

7. **getUserById**

Description : Récupérer un utilisateur spécifique par son `id`.

Étapes principales :

1. Cherche l'utilisateur avec `User.findByPk`.
2. Vérifie s'il existe.
3. Retourne les données de l'utilisateur ou une erreur.

Points forts des controllers

- Sécurité : Utilisation de bcrypt pour hasher les mots de passe et de JWT pour sécuriser les sessions.
- Validation : Vérifie l'unicité des emails et usernames, ainsi que l'existence des rôles.
- Modularité : Les rôles sont gérés dynamiquement, facilitant les évolutions futures.
- Gestion des erreurs : Messages clairs pour les erreurs courantes (conflit d'email, utilisateur introuvable, etc.).

Ces Controllers forment une base solide pour gérer les utilisateurs de manière sécurisée et efficace .

II. Posts

Fichier **post.controller.js**

```
const Post = require("../models/posts.model");
const User = require("../models/users.model");
const path = require("path");

exports.createPost = async (req, res) => {…
};

exports.getAllPosts = async (req, res) => {…
};

exports.getPostById = async (req, res) => {…
};

exports.updatePost = async (req, res) => {…
};

exports.deletePost = async (req, res, next) => {…
};
```

Ces fonctions permettent de créer, lire, modifier et supprimer des posts liés à des User dans une application Node.js, en utilisant Sequelize pour interagir avec la base de données.

1. createPost

Description : Permet à un utilisateur connecté de créer un post.

Étapes principales :

- Extrait les données `title` et `description` du corps de la requête, ainsi que l'image si elle est présente.
- Vérifie si l'utilisateur est authentifié via le `req.auth`.
- Vérifie que les champs obligatoires (titre et description) sont remplis.
- Créé le post en le liant à l'utilisateur via son ID.
- Retournez le post créé ou une erreur en cas de problème.

2. getAllPosts

Description : Récupère tous les posts enregistrés dans la base de données.

Étapes principales :

- Utiliser `Post.findAll` pour récupérer les posts, en incluant les informations sur l'utilisateur (ex. `username`).
- Trie les posts par ordre de création décroissant (`created_at`).
- Formate les résultats pour inclure uniquement le nom d'utilisateur et le nom du fichier image (le cas échéant).
- Retourne les données des posts ou une erreur en cas de problème.

3. getPostById

Description : Récupère un post spécifique par son ID.

Étapes principales :

- Recherche le post via `Post.findByPk`.
- Vérifie si le post existe :
    - Sinon, retourne une erreur 404.
    - Sinon, retourne les détails du post.
- Gère les erreurs de récupération.

4. **updatePost**

Description : Met à jour les informations d'un post spécifique.

Étapes principales :

- Récupérer les nouvelles données (titre, description, image) à partir de la requête.
- Cherche le post par son ID avec `Post.findByPk`.
- Vérifie si le post existe :
    - Si non, retourne une erreur 404.
- Vérifie si l'utilisateur a les droits de modification :
    - Autorise seulement l'auteur du post ou un administrateur.
- Met à jour les champs modifiables tout en conservant les valeurs existantes si aucune nouvelle donnée n'est fournie.
- Sauvegarde les modifications et retourne le post mis à jour.
- Gère les erreurs de mise à jour.

5. deletePost

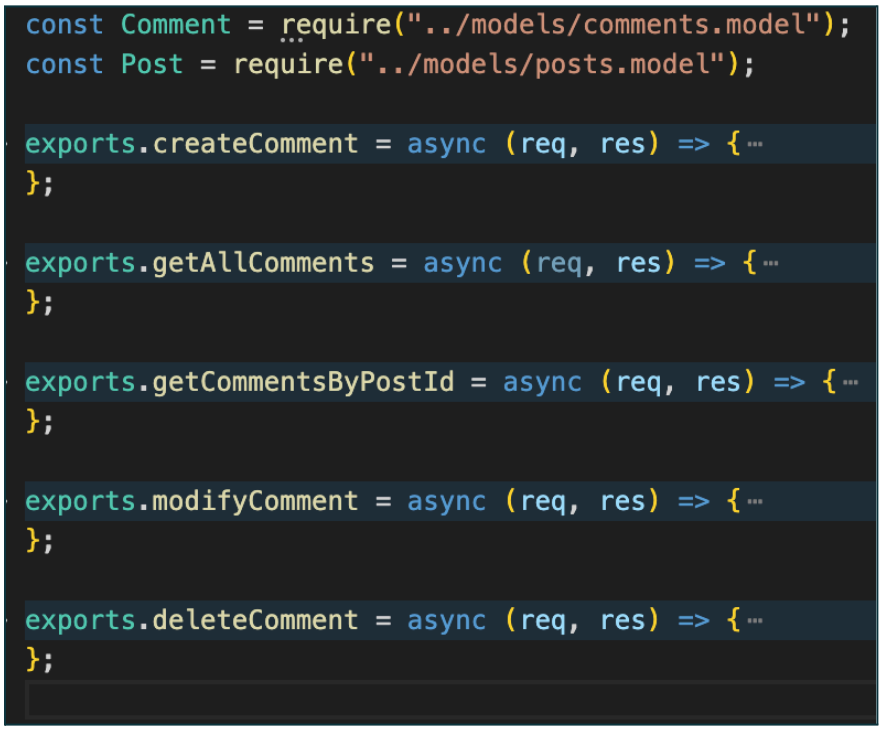
Description : Supprime un post spécifique par son ID.

Étapes principales :

- Cherche le post à supprimer avec `Post.findByPk`.
- Vérifie si le post existe :
    - Si non, retourne une erreur 404.
- Supprime le post avec `post.destroy`.
- Retourne une confirmation de suppression ou une erreur en cas de problème.

Ces Controllers gèrent les interactions principales avec les posts tout en appliquant des vérifications de sécurité comme l'authentification, les permissions, et la gestion des erreurs.

III. Comments



```
const Comment = require("../models/comments.model");
const Post = require("../models/posts.model");

exports.createComment = async (req, res) => {…
};

exports.getAllComments = async (req, res) => {…
};

exports.getCommentsByPostId = async (req, res) => {…
};

exports.modifyComment = async (req, res) => {…
};

exports.deleteComment = async (req, res) => {…
};
```

Fichier **comments.controller.js**

Ces fonctions permettent de créer, lire, modifier et supprimer des commentaires liés à des publications dans une application Node.js, en utilisant Sequelize pour interagir avec la base de données.

1. createComment

But : Créer un commentaire pour un post spécifique.
Entrées :

- Contenu du commentaire (`content`) via `req.body`.
- ID du post (`postId`) via `req.params`.
- ID de l'utilisateur (`userId`) récupéré depuis le middleware JWT.

Étapes :

- Vérifie si le post existe dans la base.
- Crée un commentaire avec le contenu, l'ID du post, et l'ID de l'utilisateur.
- Retourne le commentaire créé ou une erreur.

2. getAllComments

But : Récupérer tous les commentaires en les regroupant par post.

Sortie :

- Chaque objet contient les détails du post (ID, titre, description, image) et un tableau de ses commentaires.

Étapes :

- Récupère tous les commentaires, incluant les informations sur l'utilisateur et le post associé.
- Groupe les commentaires par ID de post.
- Transforme le regroupement en un tableau d'objets pour la réponse.

3. getCommentsByPostId

But : Récupérer tous les commentaires d’un post spécifique.

Entrées :

- ID du post (`postId`) via `req.params`.

Étapes :

- Filtre les commentaires par ID de post.
- Inclut les informations de l'utilisateur ayant écrit chaque commentaire.
- Retourne les commentaires triés par ordre décroissant de création ou une erreur si aucun commentaire n'existe.

4. modifyComment

But : Modifier le contenu d'un commentaire existant.

Entrées :

- Contenu mis à jour (`content`) via `req.body`.
- ID du commentaire (`commentId`) via `req.params`.

Étapes :

- Récupérer le commentaire et son post pour vérifier les droits.
- Autorise la modification si l'utilisateur est l'auteur ou un admin.
- Vérifie que le contenu n'est pas vide.
- Met à jour le commentaire avec un nouveau contenu et une date de modification.
- Retourne le commentaire mis à jour ou une erreur.

5. deleteComment

But : Supprimer un commentaire spécifique.

Entrées :

- ID du commentaire (`commentId`) via `req.params`.

Étapes :

- Récupère le commentaire et vérifie les droits d'accès (admin ou auteur du commentaire).
- Supprimez le commentaire de la base.
- Retourne une confirmation ou une erreur.

Points importants

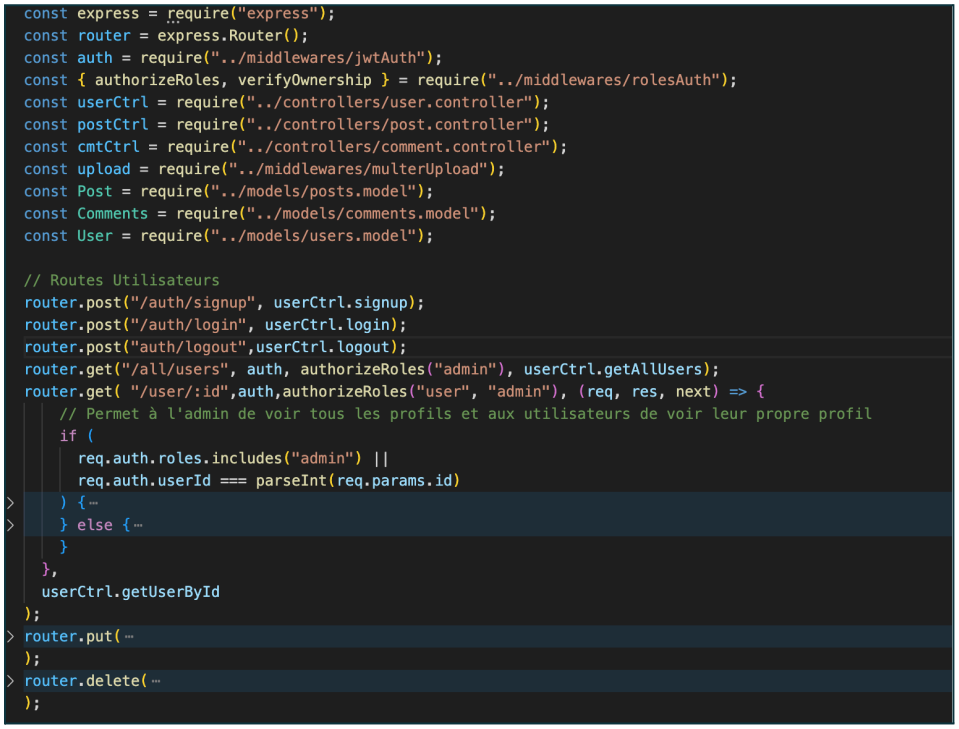
- **Sécurité** :
  - Les opérations sensibles (modification/suppression) sont protégées par une vérification des rôles et des droits des utilisateurs.
- **Gestion des erreurs** :
  - Chaque fonction capture les erreurs et retourne des réponses appropriées avec un statut HTTP (`404`, `403`, `500`).
- **Optimisation** :
  - Utilisation de Sequelize pour inclure les relations (`User`, `Post`) et réduire le nombre de requêtes SQL.
- **Expérience utilisateur** :
  - Messages clairs pour les réussites et les erreurs.

Ces fonctions couvrent les besoins essentiels pour la gestion des commentaires dans une application web moderne, tout en assurant une bonne modularité et une logique métier claire.

G. Routes

1. Routes pour les utilisateurs

Ces routes gèrent toutes les actions relatives aux utilisateurs, telles que l'inscription, la connexion, la gestion des profils, et la suppression des comptes.



```
const express = require("express");
const router = express.Router();
const auth = require("../middlewares/jwtAuth");
const { authorizeRoles, verifyOwnership } = require("../middlewares/rolesAuth");
const userCtrl = require("../controllers/user.controller");
const postCtrl = require("../controllers/post.controller");
const cmtCtrl = require("../controllers/comment.controller");
const upload = require("../middlewares/multerUpload");
const Post = require("../models/posts.model");
const Comments = require("../models/comments.model");
const User = require("../models/users.model");

// Routes Utilisateurs
router.post("/auth/signup", userCtrl.signup);
router.post("/auth/login", userCtrl.login);
router.post("auth/logout",userCtrl.logout);
router.get("/all/users", auth, authorizeRoles("admin"), userCtrl.getAllUsers);
router.get( "/user/:id",auth,authorizeRoles("user", "admin"), (req, res, next) => {
    // Permet à l'admin de voir tous les profils et aux utilisateurs de voir leur propre profil
    if (
      req.auth.roles.includes("admin") ||
      req.auth.userId === parseInt(req.params.id)
    ) {…
    } else {…
    }
  },
  userCtrl.getUserById
);
router.put(…
);
router.delete(…
);
```

- Inscription et Connexion :
  - `POST /auth/signup` : Permet à un utilisateur de s'inscrire en créant un compte.
  - `POST /auth/login` : Permet à un utilisateur de se connecter et de recevoir un token JWT.
  - `POST /auth/logout` : Déconnecte l'utilisateur (cette route semble être en cours d'implémentation).

- Gestion des utilisateurs :
  - `GET /all/users` : Accessible uniquement par un administrateur. Renvoie la liste de tous les utilisateurs
  - `GET /user/:id` : Permet à un utilisateur de voir son propre profil ou à un administrateur de consulter le profil de n'importe quel utilisateur.
  - `PUT /user/:id` : Permet à un utilisateur de mettre à jour son profil ou à un administrateur de modifier celui de n'importe quel utilisateur.
  - `DELETE /user/:id` : Permet à un utilisateur de supprimer son propre compte ou à un administrateur de supprimer le compte de n'importe quel utilisateur.

Mécanismes de sécurité :

- Utilisation de `auth` pour authentifier les utilisateurs via JWT.
- Utilisation de `authorizeRoles` pour restreindre l'accès en fonction des rôles (ex. `admin` ou `user`).
- Vérification des droits de propriété avec `verifyOwnership`

2. Routes pour les posts

```
// Routes pour les Posts
router.post(
  "/post",
  auth,
  authorizeRoles("user", "admin"),
  upload.single("image"),
  (req, res, next) => {
    req.body.userId = req.auth.userId;
    next();
  },
  postCtrl.createPost
);
router.get("/all/posts", postCtrl.getAllPosts);
router.get("/post/:id", auth, postCtrl.getPostById);
router.put(
  "/put/post/:id",
  auth,
  authorizeRoles("user", "admin"),
  verifyOwnership(async (req) => {
    return await Post.findByPk(req.params.id);
  }),
  postCtrl.updatePost
);
router.delete( ⋯
);
```

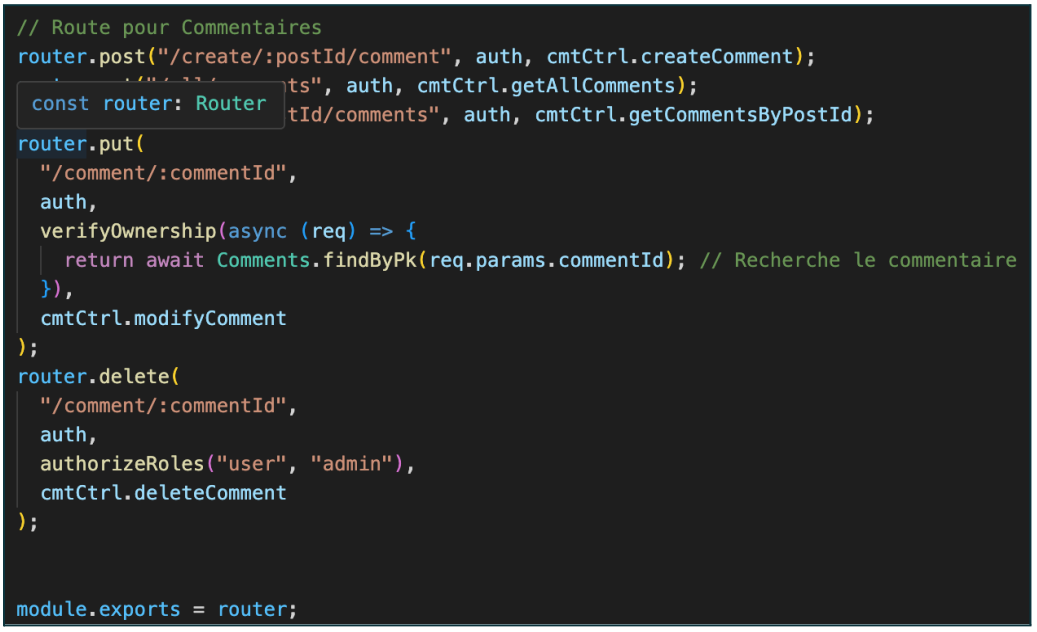
Ces routes gèrent la création, la consultation, la mise à jour et la suppression des posts. Les utilisateurs peuvent publier des contenus avec ou sans images.

- Création :
  - `POST /post` : Crée un nouveau post. L'utilisateur doit être authentifié (`auth`) et disposer des rôles adéquats. Un middleware gère également l'upload d'image avec Multer.
- Consultation :
  - `GET /all/posts` : Récupère tous les posts disponibles pour tous les utilisateurs.
  - `GET /post/:id` : Permet de consulter un post spécifique par son ID (authentification requise).
- Mise à jour :
  - `PUT /put/post/:id` : Permet de mettre à jour un post. Vérifie que l'utilisateur est le propriétaire du post ou qu'il dispose des droits d'administration.
- Suppression :
  - `DELETE /delete/post/:id` : Supprime un post après avoir vérifié si l'utilisateur est propriétaire ou administrateur.

Mécanismes de sécurité :

- Les utilisateurs doivent être authentifiés et leurs droits sont vérifiés avec `authorizeRoles` ou via la comparaison avec l'auteur du post.

3. Routes pour les commentaires



```
// Route pour Commentaires
router.post("/create/:postId/comment", auth, cmtCtrl.createComment);
                                    ts", auth, cmtCtrl.getAllComments);
const router: Router                tId/comments", auth, cmtCtrl.getCommentsByPostId);
router.put(
  "/comment/:commentId",
  auth,
  verifyOwnership(async (req) => {
    return await Comments.findByPk(req.params.commentId); // Recherche le commentaire
  }),
  cmtCtrl.modifyComment
);
router.delete(
  "/comment/:commentId",
  auth,
  authorizeRoles("user", "admin"),
  cmtCtrl.deleteComment
);


module.exports = router;
```

Ces routes permettent aux utilisateurs d'ajouter des commentaires à des posts existants, de consulter des commentaires et de gérer leurs modifications ou suppressions.

- **Création :**
  - `POST /create/:postId/comment` : Permet à un utilisateur authentifié d'ajouter un commentaire à un post existant.
- **Consultation :**
  - `GET /all/comments` : Récupère tous les commentaires avec les informations des utilisateurs et des posts associés.
  - `GET /posts/:postId/comments` : Récupère les commentaires spécifiques à un post, classés par date de création.
- **Mise à jour :**
  - `PUT /comment/:commentId` : Permet à un utilisateur de modifier un commentaire dont il est l'auteur ou à un administrateur d'intervenir.
- **Suppression :**
  - `DELETE /comment/:commentId` : Supprime un commentaire. L'accès est réservé à l'auteur ou à un administrateur.

Mécanismes de sécurité :

- Vérification de la propriété du commentaire avec `verifyOwnership`.
- Restrictions d'accès selon les rôles (utilisateur ou administrateur).

Tableau des routes

	ENDPOINT	OUTPUT	CONTROLLER	AUTH
		Connexion		
POST	/auth/login	Permet de se connecter	user.controller.js	Non-connecté
		users		
GET	all/users	Retourne la liste des utilisateurs	user.controller.js	Administrateur
POST	auth/signup	Permet de s'inscrire	user.controller.js	Non-connecté
GET	/users/:id	Retourne un utilisateur	user.controller.js	Connecté
PUT	/users/:id	Met à jour un utilisateur puis retourne l'utilisateur modifié	user.controller.js	Connecté
DELETE	/users/:id	Retourne un message de suppression (status 204)	user.controller.js	Connecté
		Posts		
GET	all/posts	Retourne une liste de posts	post.controller.js	Non-Connecté
POST	/post	Crée et retourne le post créé	post.controller.js	Connecté
GET	/post/:id	Retourne un post sélectionné	post.controller.js	Connecté
PUT	/put/post/:id	Modifier un post existant	post.controller.js	Connecté
DELETE	delete/post/:id	Ne retourne rien (juste un status 204)	post.controller.js	Connecté
		Comments		
GET	/all/comments	Récupère la liste de toutes les commentaire	comment.controller.js	Connecté
POST	/create/:postID /comment	Créer un commentaire sur un post et retourne un status 201	comment.controller.js	Connecté
GET	/posts/:postID /comments	Récupérer les commentaires d'un post	comment.controller.js	Connecté
PUT	/comment/ :commentID	Met à jour un commentaire	comment.controller.js	Connecté
DELETE	/comment/ :commentID	Supprime un comment et retourne un error message	comment.controller.js	Connecté

H. Test Unitaire

Les tests sont essentiels pour garantir la fiabilité, la stabilité et la performance d'une application. Dans le cadre du développement d'une application Node.js, les tests permettent de vérifier que chaque fonctionnalité fonctionne comme prévu. Il existe plusieurs types de tests, dont les tests unitaires, d'intégration, et fonctionnels, chacun ayant un objectif spécifique.

*Les **tests unitaires** sont utilisés pour tester des fonctions ou des méthodes isolées.*

*Dans cet environnement, on peut utiliser des outils comme **Jest** pour effectuer ces tests. Jest est un framework de test populaire pour Node.js qui offre une API facile à utiliser, des tests asynchrones, et un rapport détaillé des résultats.*

Nous allons maintenant aborder les tests réalisés pour la gestion des utilisateurs dans ton application, en commençant par les tests associés à la route des utilisateurs. Ces tests valident des scénarios comme l'inscription, la connexion, la mise à jour du profil, et la gestion des rôles et autorisations.

Dans le cadre de mon projet, j'ai mis en place des tests unitaires pour assurer la fiabilité des fonctions du contrôleur utilisateur (par exemple, pour l'inscription, la connexion, la mise à jour et la suppression d'un utilisateur) en utilisant le framework Jest. Voici quelques exemples de tests réalisés :

1.
```
// Tests pour la fonction deleteUser
describe("deleteUser", () => {
  it("Doit supprimmer un user", async () => {
    req.params.id = "1";
    User.destroy.mockResolvedValue(1);

    await userController.deleteUser(req, res);

    expect(User.destroy).toHaveBeenCalledWith({ where: { user_id: "1" } });
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.send).toHaveBeenCalledWith({
      message: "Utilisateur supprimé avec succès",
    });
  });
```

   Test de la suppression d'un utilisateur : Pour tester la suppression d'un utilisateur, j'ai mocké la méthode destroy de Sequelize. Le test vérifie que la méthode est bien appelée avec l'ID de l'utilisateur et que la bonne réponse est envoyée :

2.

```
describe("signup", () => {
  it("Doit enregistrer un nouvel user", async () => {
    req.body = {
      email: "test@example.com",
      password: "password",
      username: "testuser",
    };

    User.findOne.mockResolvedValue(null);
    bcrypt.hash.mockResolvedValue("hashedPassword");

    const mockCreatedUser = {
      user_id: 1,
      email: "test@example.com",
      username: "testuser",
      password: "hashedPassword",
      setRoles: jest.fn().mockResolvedValue(true),
    };

    User.create.mockResolvedValue(mockCreatedUser);
    Role.findOne.mockResolvedValue({ role_id: 1, role_name: "user" });

    await userController.signup(req, res);

    expect(User.create).toHaveBeenCalledWith({
      username: "testuser",
      email: "test@example.com",
      password: "hashedPassword",
    });

    expect(res.send).toHaveBeenCalledWith(
      expect.objectContaining({
        message: expect.stringContaining("enregistré avec succès"),
      })
    );
  });
});
```

Test de l'inscription d'un utilisateur (signup) : L'inscription d'un utilisateur passe par plusieurs étapes : la vérification de l'email, le hachage du mot de passe, et l'ajout des rôles. Ce test vérifie que l'utilisateur est bien créé et que le rôle lui est assigné

3.

```
// Tests pour la fonction modifier user
describe("updateUser", () => {
  it("Doit modifier l/user avec succès", async () => {
    const mockUser = {
      id: 1,
      ame",
      email: "old@test.com",
      save: jest.fn(),
      setRoles: jest.fn(),
    };

    req.params = { id: "1" };
    req.body = {
      username: "newname",
      email: "new@test.com",
    };

    User.findByPk.mockResolvedValue(mockUser);

    await userController.updateUser(req, res);

    expect(mockUser.save).toHaveBeenCalled();
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.send).toHaveBeenCalledWith({
      message: "Utilisateur mis à jour avec succès",
      user: mockUser,
    });
  });
```

Cliquez pour réduire la plage.

Test de la mise à jour d'un utilisateur : La mise à jour des informations d'un utilisateur (par exemple, le nom ou l'email) est un autre point crucial. Voici un exemple de test pour cela :

4.

```
// Tests pour la fonction Login
describe("login", () => {
  it("Devrait se connecter avec succès et renvoyer un token", async () => {
    req.body = { email: "test@example.com", password: "password" };
    User.findOne.mockResolvedValue({
      user_id: 1,
      email: "test@example.com",
      password: "hashedPassword",
      roles: [{ role_name: "user" }],
    });
    bcrypt.compare.mockResolvedValue(true);
    jwt.sign.mockReturnValue("token");

    await userController.login(req, res);

    expect(res.json).toHaveBeenCalledWith({
      message: "Connexion réussie",
      token: "token",
    });
  });
```

Test de la connexion d'un utilisateur (login) : Le processus de connexion vérifie si l'email et le mot de passe correspondent à un utilisateur enregistré. Voici le test pour vérifier que la connexion fonctionne et que le token est généré correctement :

Ces tests ont pour objectif de garantir que le code est fiable et robuste face aux erreurs possibles. J'ai effectué des tests pour les posts et les commentaires, mais faute de place dans mon mémoire, je n'ai montré que ceux concernant les utilisateurs. Ces tests ont pour objectif de garantir que le code est fiable et robuste face aux erreurs possibles. J'ai également testé la gestion des erreurs de statut, telles que les utilisateurs introuvables ou les erreurs de base de données, afin de m'assurer que le système réagit correctement dans ces situations.

I. Docker

Docker est un outil essentiel pour la containerisation des applications, permettant ainsi de simplifier le déploiement, la gestion des environnements et l'intégration continue. Grâce à Docker, il est possible de créer des environnements isolés, appelés conteneurs, qui assurent la portabilité des applications, quel que soit l'environnement dans lequel elles sont exécutées. Cette technologie est particulièrement utile pour les projets de développement web, car elle garantit que le code fonctionne de manière identique sur toutes les machines, que ce soit en développement, en test ou en production.

A. Dockerfile

```
# Utiliser Node.js 20
FROM node:20-alpine

WORKDIR /usr/src/app


COPY package*.json ./


RUN npm install


COPY . .


EXPOSE 3000

# Commande de démarrage
CMD ["npm", "start"]
```

Le **Dockerfile** crée une image contenant ton application Node.js, en installant les dépendances nécessaires et en exposant le port 3000 pour accéder à l'application.

Le fichier **Docker Compose** gère deux services : un pour la base de données MySQL et un autre pour ton application Node.js.

Il configure les variables d'environnement nécessaires pour connecter l'application à la base de données et assure que le service Node.js ne démarre qu'une fois MySQL opérationnel.

Enfin, des volumes sont utilisés pour garantir la persistance des données et la synchronisation du code local avec le conteneur.

B. Docker-Compose

```yaml
version: "3.6"

services:
  mysql:
    image: mysql:latest
    container_name: mysql-container
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      MYSQL_DATABASE: ${MYSQL_DATABASE}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
    volumes:
      - ./mysql-data:/var/lib/mysql
    ports:
      - "3306:3306"
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  nodeapp:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      mysql:
        condition: service_healthy
    environment:
      - DB_HOST=${DB_HOST}
      - DB_PORT=${DB_PORT}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}
      - DB_NAME=${DB_NAME}
      - JWT_SECRET=${JWT_SECRET}
    volumes:
      - .:/usr/src/app
      - /usr/src/app/node_modules
```

Le fichier Docker Compose configure deux services principaux :

- un pour MySQL.

Il définit les variables d'environnement nécessaires pour la connexion à la base de données, telles que les informations de connexion et la base de données à utiliser. Le service MySQL est configuré avec une vérification de santé pour garantir qu'il est prêt avant de démarrer l'application.

- Le service Node.js attend que MySQL soit prêt, puis il expose le port 3000 et synchronise le code local avec le conteneur pour faciliter le développement.

DIFFICULTÉS RENCONTRÉES ET LES SOLUTIONS APPORTÉES

Difficultés rencontrées :

L'une des principales difficultés a été de comprendre l'importance de mocker les modèles de la base de données lors de l'écriture des tests unitaires. Initialement, j'avais tenté d'exécuter les tests en interagissant directement avec la base de données réelle, ce qui a causé plusieurs problèmes, notamment des erreurs liées à l'accès aux données ou des performances ralentissant le processus de test. De plus, l'absence de mock des modèles a rendu difficile la simulation des comportements des méthodes comme `findByPk`, `create`, ou `findAll`, ce qui limitait la couverture et l'efficacité des tests.

```
// Configuration des mocks pour les modèles et Sequelize

// Cela permet de simuler le comportement de la base de données
jest.mock("../models/index", () => {…
});

jest.mock("sequelize", () => {
  // Configuration du mock pour Sequelize
  class MockModel {
    static init = jest.fn();
    static findOne = jest.fn();
    static create = jest.fn();
  }

  return {
    Model: MockModel,
    DataTypes: {
      INTEGER: "INTEGER",
      STRING: "STRING",
    },
    Sequelize: jest.fn(() => ({
      define: jest.fn(),
      authenticate: jest.fn(),
      sync: jest.fn(),
    })),
  };
});

// Mock des dépendances
jest.mock("../models/index");
jest.mock("bcryptjs");
jest.mock("jsonwebtoken");
```

Solutions apportées :

Pour surmonter ce problème, j'ai appris à utiliser `jest.mock` pour créer des versions simulées (mocks) des modèles de Sequelize utilisés dans mon projet. J'ai défini des méthodes fictives pour chaque modèle, comme `create`, `findByPk`, et `destroy`, en m'assurant que leurs retours correspondaient aux scénarios de test. Par exemple, j'ai ajouté des implémentations personnalisées pour simuler la création d'un utilisateur ou l'association de rôles. Cette approche m'a permis d'exécuter mes tests de manière isolée, sans dépendre de la base de données réelle, et d'améliorer la vitesse et la fiabilité de mes tests unitaires.

Difficultés rencontrées :

Une des difficultés majeures a été de comprendre et de mettre en place correctement les relations entre les différents modèles Sequelize, comme `User`, `Roles`, `Posts`, et `Comments`. Notre professeur nous avait laissé apprendre par nous-mêmes, ce qui a rendu la tâche encore plus complexe, notamment pour définir les relations `belongsToMany` avec des tables pivot, ainsi que les relations `hasMany` et `belongsTo`. La documentation officielle, bien qu'utile, nécessitait une bonne compréhension des concepts avancés de Sequelize et des bases de données relationnelles, ce qui n'était pas toujours évident au départ. Cela a entraîné des erreurs, comme des clés étrangères incorrectes ou des relations mal configurées, impactant le bon fonctionnement des requêtes.

```
// Import des models
const User = require("./users.model");
const Roles = require("./roles.model");
const Posts = require("./posts.model");
const Comments = require("./comments.model");

// associations
Roles.belongsToMany(User, {
  through: "user_roles",
  foreignKey: "role_id",
  otherKey: "user_id",
});

User.belongsToMany(Roles, {
  through: "user_roles",
  foreignKey: "user_id",
  otherKey: "role_id",
  as: "roles",
});

User.hasMany(Posts, { foreignKey: "user_id" });
User.hasMany(Comments, { foreignKey: "user_id" });

Posts.belongsTo(User, { foreignKey: "user_id" });
Posts.hasMany(Comments, { foreignKey: "post_id" });

Comments.belongsTo(User, { foreignKey: "user_id" });
Comments.belongsTo(Posts, { foreignKey: "post_id" });

const ROLES = ["user", "admin"];
```

Solutions apportées :

Pour surmonter ces obstacles, j'ai pris le temps d'explorer la documentation de Sequelize et de chercher des exemples concrets sur des forums et des tutoriels en ligne. J'ai appris à définir chaque relation étape par étape et à comprendre leur utilité dans le cadre d'un modèle relationnel. Par exemple, j'ai réalisé l'importance des clés étrangères (`foreignKey`) et des alias (`as`) dans les relations `belongsToMany` pour éviter les conflits de noms. En testant progressivement chaque relation et en validant leur bon fonctionnement à l'aide de requêtes simples, j'ai pu corriger les erreurs. Cela m'a permis de construire un schéma relationnel robuste et fonctionnel, facilitant ainsi les interactions entre mes modèles dans l'application.

Conclusion

Ce mémoire marque la fin d'une étape essentielle de mon parcours en développement web et mobile, un chemin riche en apprentissages, en défis et en progrès constants. À travers ce projet, j'ai eu l'opportunité de mettre en pratique les connaissances acquises durant ma formation et de les approfondir en travaillant sur des technologies modernes comme Node.js, Sequelize, Docker, et bien d'autres. Chaque étape, qu'il s'agisse de la conception des modèles de données, de l'implémentation des API RESTful, de la gestion des rôles ou des tests unitaires, a été une occasion précieuse pour perfectionner mes compétences techniques, organisationnelles et collaboratives.

L'un des moments clés de ce projet a été l'utilisation de Docker pour containeriser et orchestrer les différents services nécessaires au bon fonctionnement de l'application. Cette technologie m'a permis de mieux comprendre les enjeux liés au déploiement et à l'intégration continue, tout en assurant une portabilité maximale du projet. La maîtrise de cet outil s'est révélée être un atout indispensable, consolidant ma compréhension des environnements de production modernes. Par ailleurs, l'adoption d'une architecture MVC tout au long de ce projet a favorisé une organisation claire et modulaire du code, rendant l'application plus maintenable et évolutive. Cette approche a également mis en lumière l'importance des bonnes pratiques de programmation et de la structuration rigoureuse des projets.

Sur le plan technique, ce projet m'a confronté à des problématiques variées, allant de la gestion des relations complexes entre les modèles dans Sequelize à la sécurisation des routes et à la validation des données. Ces défis, bien que exigeants, ont été des occasions de me surpasser et d'approfondir des concepts clés comme l'authentification via JWT, la gestion centralisée des erreurs et la mise en œuvre de middlewares personnalisés pour la vérification des permissions. Ces expériences m'ont permis de mieux appréhender les besoins réels des applications en production.

Un autre aspect crucial a été l'intégration des tests unitaires avec Jest, qui m'a sensibilisé à l'importance de la qualité et de la robustesse du code dans un environnement professionnel. Même si je n'ai pas pu inclure tous les tests réalisés dans ce mémoire, leur conception et leur mise en œuvre ont représenté un apprentissage clé. En testant mes composants de manière systématique, j'ai pu anticiper d'éventuelles erreurs, renforcer la fiabilité du projet et garantir une expérience utilisateur fluide.

En ce qui concerne le développement front-end, bien que ce projet ait été axé principalement sur le backend, il a renforcé mon envie de me spécialiser dans cette direction. Mes premiers projets en JavaScript côté client et mon initiation à des frameworks comme React.js m'ont permis de saisir l'importance de concevoir des interfaces fluides et intuitives. Je reste convaincu que l'aspect visuel et interactif des applications joue un rôle crucial dans l'expérience utilisateur, et j'aspire à développer davantage ces compétences.

Enfin, ce mémoire ne témoigne pas seulement d'un projet technique abouti, mais aussi d'un processus d'apprentissage continu. Il a mis en lumière des qualités personnelles telles que la persévérance, la capacité à résoudre des problèmes complexes et à collaborer efficacement avec des pairs et des outils. Ces compétences, alliées à une curiosité constante pour les nouvelles technologies, forment un socle solide pour mes ambitions futures.

En conclusion, ce projet et ce mémoire symbolisent un véritable tournant dans mon parcours professionnel. Ils marquent non pas une fin, mais un nouveau départ vers des réalisations encore plus ambitieuses. À l'avenir, je compte approfondir mes compétences en développement web, en explorant notamment l'univers du front-end et du design d'interfaces. Fort de cette expérience et des compétences acquises, je me sens prêt à relever les défis du monde professionnel et à contribuer activement à des projets innovants et porteurs de sens.

Annexe

Les repos du projet sont accessibles à cette adresse: Le projet est accessible à l'adresse suivante: **https://github.com/Walidd35/SocialNetwork-WorkUup**

RÔLE	EMAIL	MOT DE PASSE
Administrateur	userw@gmail.com	user
Utilisateur	user@gmail.com	user