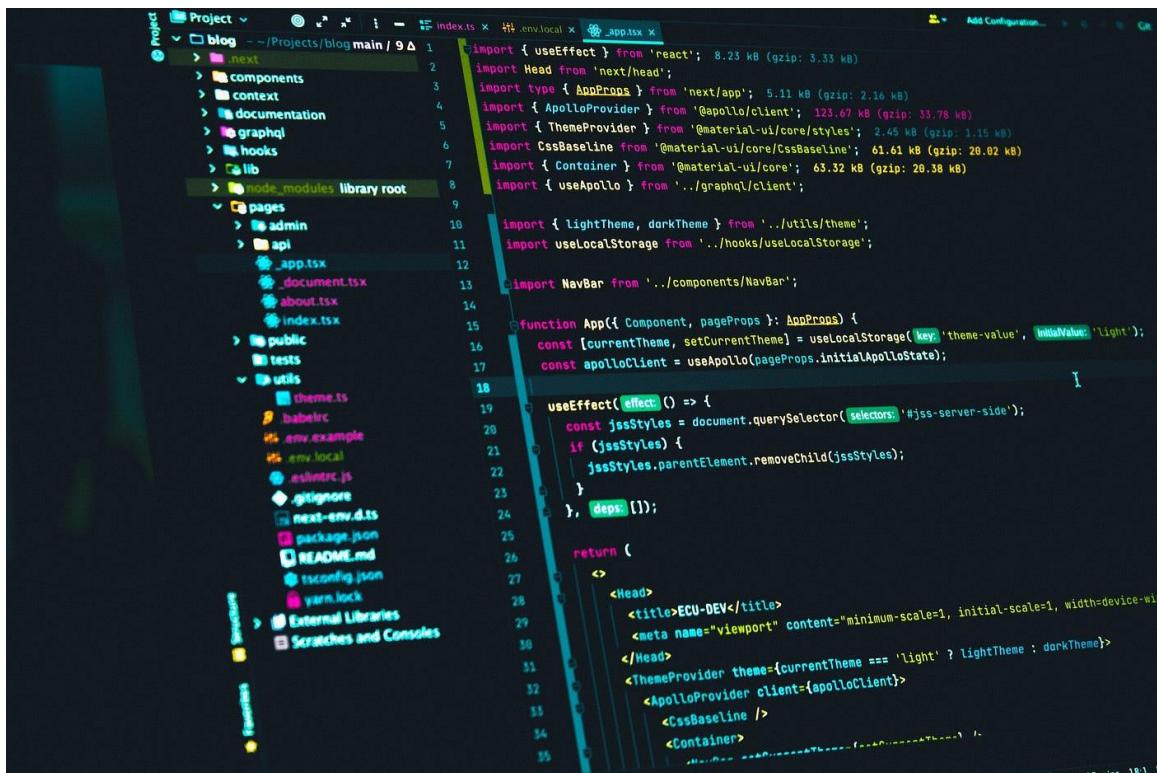


P_295 Back-end



The screenshot shows a code editor with a dark theme. On the left is a file tree for a Next.js project named 'blog'. The tree includes 'next', 'components', 'context', 'documentation', 'graphql', 'hooks', 'lib', 'node_modules' (marked as 'library root'), 'pages' (containing 'admin', 'api', '_app.tsx', '_document.tsx', 'about.tsx', 'index.tsx'), 'public', 'tests', and 'utils' (containing 'theme.ts'). Other files visible include '.env.example', '.env.local', 'eslint.json', '.gitignore', 'next-env.dts', 'package.json', 'README.md', 'tsconfig.json', and 'yarn.lock'. In the center is a code editor window showing a component named 'App'. The code uses TypeScript and React. It imports various packages like 'useEffect' from 'react', 'Head' from 'next/head', and 'ApolloProvider' and 'ThemeProvider' from '@apollo/client'. It also imports 'CssBaseline' and 'Container' from '@material-ui/core'. The component logic includes setting a local storage theme preference and rendering a header with a title and meta tags.

```
import { useEffect } from 'react';
import Head from 'next/head';
import type { AppProps } from 'next/app';
import { ApolloProvider } from '@apollo/client';
import { ThemeProvider } from '@material-ui/core/styles';
import CssBaseline from '@material-ui/core/CssBaseline';
import Container from '@material-ui/core/Container';
import { useApollo } from './graphql/client';

import { lightTheme, darkTheme } from '../utils/theme';
import useLocalStorage from '../hooks/useLocalStorage';

import NavBar from '../components/NavBar';

function App({ Component, pageProps }: AppProps) {
  const [currentTheme, setCurrentTheme] = useLocalStorage('key', 'theme-value', { initialValue: 'light' });
  const apolloClient = useApollo(pageProps.initialApolloState);

  useEffect(() => {
    const jssStyles = document.querySelector(`selector: #jss-server-side`);
    if (jssStyles) {
      jssStyles.parentElement.removeChild(jssStyles);
    }
  }, [deps]);
}

return (
  <>
    <Head>
      <title>ECU-DEV</title>
      <meta name="viewport" content="minimum-scale=1, initial-scale=1, width=device-width" />
    </Head>
    <ThemeProvider theme={currentTheme === 'light' ? lightTheme : darkTheme}>
      <ApolloProvider client={apolloClient}>
        <CssBaseline />
        <Container>
          <Component {...pageProps} />
        </Container>
      </ApolloProvider>
    </ThemeProvider>
  </>
)
```

Samuel Sallaku, Romain Schertenleib, Bastien Segalen – CID2B
Section Informatique - Vennes

32p
G. Charmier

Table des matières

1	INTRODUCTION	3
1.1	TITRE.....	3
1.2	DESCRIPTION.....	3
1.3	MATÉRIEL ET LOGICIELS À DISPOSITION	3
1.4	PRÉREQUIS	3
1.5	LES POINTS SUIVANTS SERONT ÉVALUÉS	3
2	ANALYSE.....	3
2.1	PLANIFICATION DES TÂCHES	3
2.2	ROUTES	4
2.3	MCD/MLD	5
2.3.1	<i>Explication de la structure</i>	6
2.3.2	<i>Expliquer les cardinalités</i>	6
2.4	STRUCTURE DU CODE.....	7
2.5	SCHÉMA.....	8
3	RÉALISATION	8
3.1	EXPLICATION D'AUTHENTIFICATION ET GESTION DE RÔLES	8
3.2	SÉCURITÉ.....	8
3.3	EXPLICATION DES FONCTIONNALITÉS TECHNIQUES	9
3.4	ECO-CONCEPTION	10
1.	Optimisation des ressources serveur.....	10
2.	Optimisation des requêtes et de la base de données	10
4	TESTS	10
4.1	TESTS RÉALISÉS	10
5	CONCLUSION.....	11
5.1	ORGANISATION DU CODE (GitHub)	11
5.2	CONCLUSION GÉNÉRALE	11
5.3	CONCLUSION PERSONNELLE.....	11
5.4	CRITIQUE CONSTRUCTIVE.....	12
6	WEBOGRAPHIE	12
6.1	WEBOGRAPHIE.....	12

1 INTRODUCTION

1.1 Titre

Réaliser le back-end d'un site de livres

1.2 Description

Projet en lien avec le module C295, donc réaliser le back-end d'une application web permettant de faire le CRUD des livres, en montrant seulement du JSON et non du HTML ou bien CSS.

1.3 Matériel et logiciels à disposition

- 1x PC ETNL
- GitHub
- VS Code
- MS Teams
- Insomnia / Postman
- Internet

1.4 Prérequis

- Connaissances de base en C295
- Connaissances de base en I293

1.5 Les points suivants seront évalués

- Le rapport
- Les planifications (initiale et détaillée)
- Le journal de travail
- Le code et les commentaires
- Les documentations de mise en œuvre et d'utilisation

2 ANALYSE

2.1 Planification des tâches

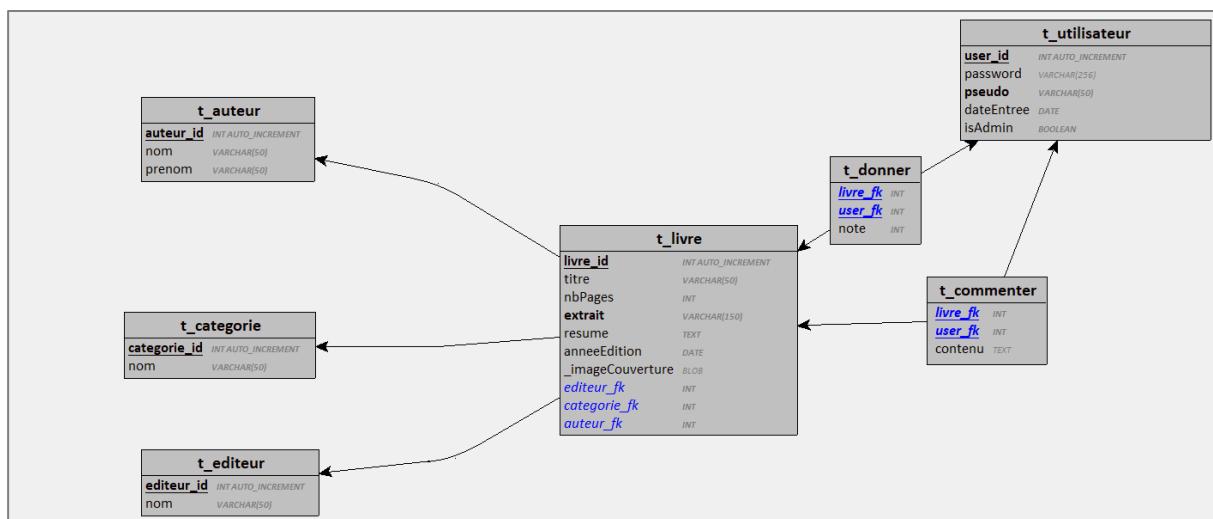
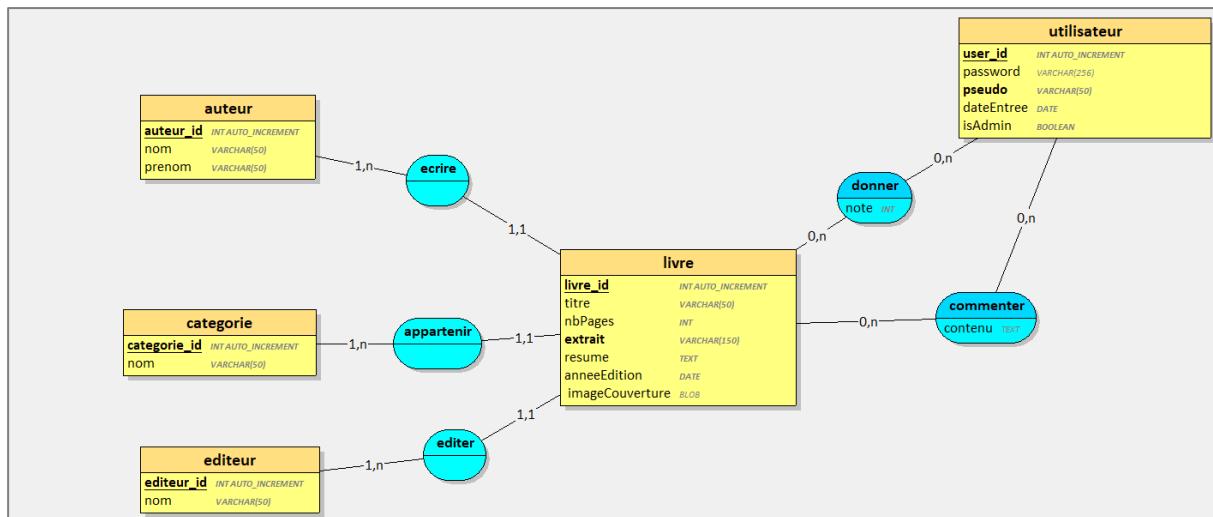
Consultez notre GitHub Projects

Lien : <https://github.com/users/po77qxd/projects/1/views/1>

2.2 Routes

Verbe HTTP	URI	Json	Description
Get	/	Non	Page d'accueil
Get	/api/books	Non	Obtenir la liste des livres
Post	/api/books	{JSON livre}	Ajout d'un livre
Put	/api/books/2	{JSON livre}	Mettre à jour un livre
Delete	/api/books/4	Non	Supprimer 1 livre
Get	/api/books/3	Non	Voir les détails
Get	/api/books/1/comments	Non	Obtenir les commentaires d'un livre
Post	/api/books/3/comments	{JSON comment}	Ajouter un commentaire
Get	/api/books/1/notes	Non	Récupérer les notes d'un livre
Post	/api/books/1/notes	{JSON note}	Ajouter une note a un livre
Post	/api/login	[username, password]	Obtenir un jwt
Get	/api/authors/	Non	Obtenir la liste des auteurs
Get	/api/authors/1	Non	Obtenir les détails d'un auteur
Get	/api/authors/1/books	Non	Obtenir les livre qu'un auteur a écrit
Post	/api/register/	{pseudo,password}	Ajouter un user
Get	/api/users/1/books	Non	Obtenir les livres d'un user
Get	/api/categories/1/books	Non	Obtenir la catégorie d'un livre
Get	/api/categories/1	Non	Obtenir une certaine catégorie
Get	/api/categories	Non	Obtenir les catégories
Get	/api-docs	Non	Documentation Swagger

2.3 MCD/MLD



T_auteur :

Auteur_id : ID de l'auteur

Nom : nom de l'auteur

Prénom : prénom de l'auteur

T_categorie :

Categorie_id : ID de la catégorie

Nom : nom de la catégorie

T_editeur :

Editeur_id : ID de l'éditeur

Nom : nom de l'éditeur

T_livre :

Livre_id : ID du livre

Auteur : Samuel Sallaku

Modifié par : Samuel Sallaku

Version: 529 du 12.03.2025 10:15

Titre : titre du livre

NbPages : nombre de pages du livre

Extrait : un extrait du livre

Résumé : un petit résumé du livre

AnnéeEdition : l'année d'édition

_imageCouverture : Une image de couverture du livre

Editeur_fk : clé étrangère de l'éditeur

Catégorie_fk : clé étrangère de la catégorie

Auteur_fk : clé étrangère de l'auteur

T_utilisateur :

User_id : Id de l'utilisateur

Password : mot de passe de l'utilisateur

Pseudo : Nom d'utilisateur

DateEntree : date de création de l'utilisateur

IsAdmin : rôle pour les utilisateurs pour savoir s'ils sont admin ou pas

T_donneur :

Livre_fk : ID et clé étrangère venant de la table t_livre

User_fk : ID et clé étrangère venant de la table t_user

Note : note qui sera donnée au livre (appréciation)

T_commenter :

Livre_fk : ID et clé étrangère venant de la table t_livre

User_fk : ID et clé étrangère venant de la table t_user

Contenu : contenu du commentaire qui sera fait par rapport à un livre

2.3.1 Explication de la structure

La table t_livre se trouve au centre de la page car il s'agit de la table principale. Le centre de l'application

Pourquoi avoir sorti les tables :

- T_auteur
- T_categorie
- T_editeur

Ces tables sont trop importantes pour juste être dans la table T_livre c'est pourquoi elles sont dehors.

2.3.2 Expliquer les cardinalités

Ecrire :

1, N / un auteur écrit minimum 1 livre et au maximum un nombre de livre infini

1,1 / un livre est écrit par 1 et unique auteur

Appartenir :

1, N / une catégorie a minimum 1 livre et au maximum un nombre de livre infini

1,1 / un livre a une seule catégorie

Éditer :

1, N / un éditeur édite minimum 1 livre et au maximum un nombre de livre infini

1,1 / un livre est édité par 1 et unique éditeur

Donner :

0, n / un utilisateur donne entre 0 et un nombre infini de note

0, n / un livre a entre 0 et un nombre infini de note

Commenter :

0, n / un utilisateur donne entre 0 et un nombre infini de commentaire

0, n / un livre a entre 0 et un nombre indéfini de commentaire

2.4 Structure du code

```

    < app
      > node_modules
    < src
      < auth
        JS auth.mjs
        JS privKey.mjs
      > db
        > image
          JS mock-author.mjs
          JS mock-book.mjs
          JS mock-category.mjs
          JS mock-comment.mjs
          JS mock-editor.mjs
          JS mock-note.mjs
          JS mock-user.mjs
          JS sequelize.mjs
        > middleware
        > models
      < public\css
        # home.css
      < routes
        JS authors.mjs
        JS books.mjs
        JS categorie.mjs
        JS helper.mjs
        JS login.mjs
        JS register.mjs
        JS users.mjs
      > tests
      < Views
        < partial
          < footer.ejs
          < home.ejs
        JS app.mjs
        JS swagger.mjs
      .env
      docker-compose.yml
    {} package-lock.json
    {} package.json
  
```

Le code est structuré de manière claire et propre. Les méthodes sont toujours au-dessus des importations, à part par exemple dans le fichier app.mjs où ils sont importés juste en dessous du middleware. Dans la structure des dossiers nous avons suivi la structure de l'exercice API REST SELF MACHINE.

```

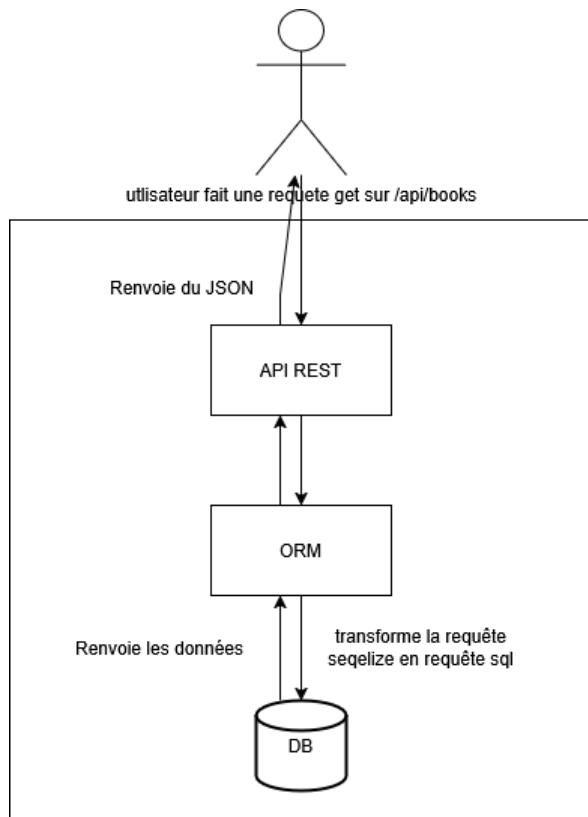
import express from "express";
import auth from "../auth/auth.mjs";
import { success } from "../routes/helper.mjs";
import { Book, Category, Comment, Note } from "../db/sequelize.mjs";
import { ValidationError, Op } from "sequelize";
import { upload } from "../middleware/multer.mjs";
import multer from "multer";
  
```

Les modèles sont faits dans de fichiers apparts, chacun ayant sa propre structure avec (ou pas) de validation de données. Pour la DB nous avions également des mock, donc des données déjà insérées pour les tables ainsi que le fichier sequelize.mjs qui fait la connexion de la DB et va aussi forcer la synchro pour pouvoir insérer ses données. Le dossier image va contenir des images qui seront stockées et utilisées pour le champ « couverture » d'un livre.

Dans node_modules, nous avons toutes les dépendances pour ce projet, comme MySQL, l'ORM Sequelize et d'autres. Dans le dossier auth nous avons aussi le middleware qui servira de vérifier les jetons JWT lors d'une requête à une certaine page web protégée.

Le système de routes est fait dans un dossier routes où ils ont tous leur propre méthode ainsi que les différentes requêtes http. Elles vont ensuite être appelés dans le fichier app.mjs lors d'une requête.

2.5 Schéma



3 RÉALISATION

3.1 Explication d'authentification et gestion de rôles

Pour l'authentification, voir le chapitre ci-dessous.

Pour la gestion de rôles, nous avons simplement implémenté dans la base de données un champ dans la table des utilisateurs étant "isAdmin", qui sera un booléen pour avoir si un utilisateur est Admin ou pas.

3.2 Sécurité

Nous n'avions pas de sécurité implémentée dans ce projet, car celui-là est en lien avec un autre module, étant le l183.

Mais nous avions quand même des routes protégées, qui ont donc besoin d'un jeton JWT pour pouvoir y accéder. Ceci est grâce à la méthode auth que nous avions implémenté.

Celui-là va donc vérifier le jeton quand il sera entré lors d'une requête à un page sécurisée. La création se fait lors du login de l'utilisateur.

```
const validToken = jwt.verify(jwtToken, privKey, (error, decoded) => {
  if (error) {
    const message = "Accès refusé, jeton expiré ou invalide";
    return res.status(401).json({ message });
  }
  const userId = decoded.userId;

  if (req.body.userId && req.body.userId !== userId) {
    const message = "Utilisateur inexistant";
    return res.status(401).json({ message });
  } else {
    next();
  }
});
```

Donc cela vérifie le jeton reçu et la clé privée et s'il y a une erreur et que le jeton est faux ou qu'il n'est plus valide il va renvoyer un code d'erreur 401 avec le message : "Accès refusé, jeton expiré ou invalide"

3.3 Explication des fonctionnalités techniques

7.4 Fonctionnalités requises (techniques)

L'application backend consiste à la mise en place d'une API REST permettant à un consommateur de développer le frontend de l'application.

Cette API REST doit être la plus complète possible avec toutes les notions vues en cours, à savoir :

- Un ensemble de routes permettant la gestion des livres, des catégories, des utilisateurs, etc
- Une validation de toutes les données fournies par le consommateur de l'API
- Une gestion des statuts http (200, 3xx, 4xx, 5xx) et des erreurs
- Une recherche sur les livres, catégories, etc
- Un système d'authentification basé sur les jetons JWT
- Une documentation Swagger la plus complète possible
- Des tests de votre API avec Insomnia ou Postman
- Des tests automatisés avec vitest
- Une intégration continue dans github actions
- Une « dockerisation » du backend
 - Il est demandé de créer et de codes des routes dans notre API REST. Nous avons un ensemble de routes pour gérer les, voir l'ensemble des routes plus haut.
 - Ici il est demandé de faire une validation de données donc ne pas mettre tout ce qu'on veut dans les champs. La validation des données a été faite pour certains champs comme les noms des utilisateurs, donc une limite de caractères et puis un message d'erreur en cas de donnée null.
 - Il nous est demandé de gérer les statuts lors d'erreur, etc. La gestion des statuts HTTP a été faite dans chaque route où il y aurait une éventuelle erreur

- Un système d'authentification pour protéger des routes. Système d'authentification basé sur les jetons JWT est fait, voir plus haut comment.
- Une documentation simple d'une route avec Swagger. Une documentation Swagger a été faite pour une route étant le GET pour les livres.
- Il est demandé de faire tests de requêtes en utilisant Insomnia. Des tests de l'API ont été faits avec Insomnia, voir l'exportation JSON dans le dossier rendu.
- Un test automatisé a été fait avec vitest.
- Une intégration continue dans GitHub actions a été faite.
- La dockerisation n'a pas été faite puisque celui-ci n'était plus demandé.

3.4 Eco-conception

1. Optimisation des ressources serveur

- Utilisation d'Express.js, un framework minimaliste et léger, ce qui réduit la consommation de ressources par rapport à des solutions plus lourdes.
- Réduction du nombre de dépendances inutiles pour alléger le projet et limiter les calculs inutiles.
- Gestion efficace des requêtes avec des **middleware** optimisés pour éviter les traitements inutiles.

2. Optimisation des requêtes et de la base de données

- **Utilisation d'un ORM efficace** (comme Sequelize) pour limiter les requêtes inutiles.

4 TESTS

4.1 Tests réalisés

Pour tester les routes de l'application, on a utilisé insomnia. Les tests automatisés sont faits avec vitest. Exemple de test :

```
const BEFORE_ALL_TIMEOUT = 30000; // 30 sec

describe("test que le code renvoie 404 quand on accède à une route qui existe pas", () => {
  let response;
  let body;

  beforeAll(async () => {
    response = await fetch("http://localhost:3000/api/1234");
    body = await response.json();
  }, BEFORE_ALL_TIMEOUT);

  test("Aurait du renvoyer 404", () => {
    expect(response.status).toBe(404);
  });
});
```

Describe() est le nom du test. Ensuite on utilise la méthode beforeAll() pour que le code s'exécute avant le reste, et dans la méthode on fetch() la route qui nous intéresse.

Ensuite on utilise la méthode test() pour vérifier que le résultat est bien celui que l'on attendait.

Les tests sont aussi exécutés lors de pull request ou de commit sur le main. (GitHub Actions)

5 CONCLUSION

5.1 Organisation du code (GitHub)

Le lien du repo GitHub : <https://github.com/po77qxd/SSS-passionLecture>

Nous avons utilisé GitHub pour collaborer. Quand on veut faire une nouvelle fonctionnalité, on crée une Branch nommé « DEV-nomFonctionnalité », par exemple DEV-docSwagger.

5.2 Conclusion générale

Ce projet nous a permis d'approfondir nos compétences en développement back-end. Travailler en équipe nous a également appris à organiser notre code de manière claire et à suivre des bonnes pratiques de développement comme faire des Branches au lieu de tous travailler sur main puis bien éviter les conflits.

Nous nous sommes bien entendus et compris entre nous car, nous sommes un groupe qui ont déjà travaillé ensemble avant et ce projet a pu nous faire travailler plus efficacement et approfondir nos connaissances dans le backend.

En conclusion, cette expérience nous a permis d'acquérir une vision claire du développement d'une API REST et nous sommes convaincus que ces compétences nous seront précieuses pour nos futurs projets.

5.3 Conclusion personnelle

Romain :

Ce projet d'API REST a été une super expérience pour moi. J'ai vraiment pu approfondir ma compréhension du backend, notamment sur la gestion des requêtes HTTP, l'authentification et la base de données. J'ai rencontré quelques défis, surtout sur la gestion des erreurs et l'optimisation des requêtes, mais j'ai réussi à construire une API fonctionnelle et bien structurée.

Bastien : Ce projet était intéressant car j'ai appris comment réaliser le backend d'une application. J'ai pu apprendre comment faire des routes GET, DELETE, etc. et j'ai appris comment utiliser un ORM tel que Sequelize. J'ai aussi appris comment faire l'authentification avec des jwt. J'ai donc beaucoup amélioré mes compétences en backend et ce projet me sera très utile pour mon futur.

Samuel : Dans ce projet je sens que j'ai progressé, appris et approfondi mes connaissances en backend. Avant ce projet je ne savais pas comment faire du backend ou même faire des routes qui ont des actions spécifiées comme GET, POST, etc. Je trouve que ce projet est très utile et me servira durant ma formation et mon futur. Malgré des problèmes de compréhension au début, j'ai pu me mettre à niveau et bien comprendre le module ainsi que ce projet avec mes camarades qui m'ont aussi aidé.

5.4 Critique Constructive

Ce qu'on aurait pu améliorer serait la gestion des erreurs. Nous pouvons faire un middleware d'erreur personnalisé au lieu de répéter des blocs de « catch »

Il nous manquait plus de tests automatisés, une meilleure validation de données car nous n'avons pas validé toutes les données lors d'une création de livre, par exemple.

Mais dans la globalité le projet a bien été réalisé. Nous avons pu bien nous partager les tâches et travailler en équipe

Points positifs à garder :

- Architecture du code que nous avons utilisé
- Efficacité et qualité du travail
- Communication
- Esprit d'équipe

6 WEBOGRAPHIE

6.1 Webographie

Repo GitHub

<https://github.com/po77qxd/SSS-passionLecture>

Projet GitHub

<https://github.com/users/po77qxd/projects/1>