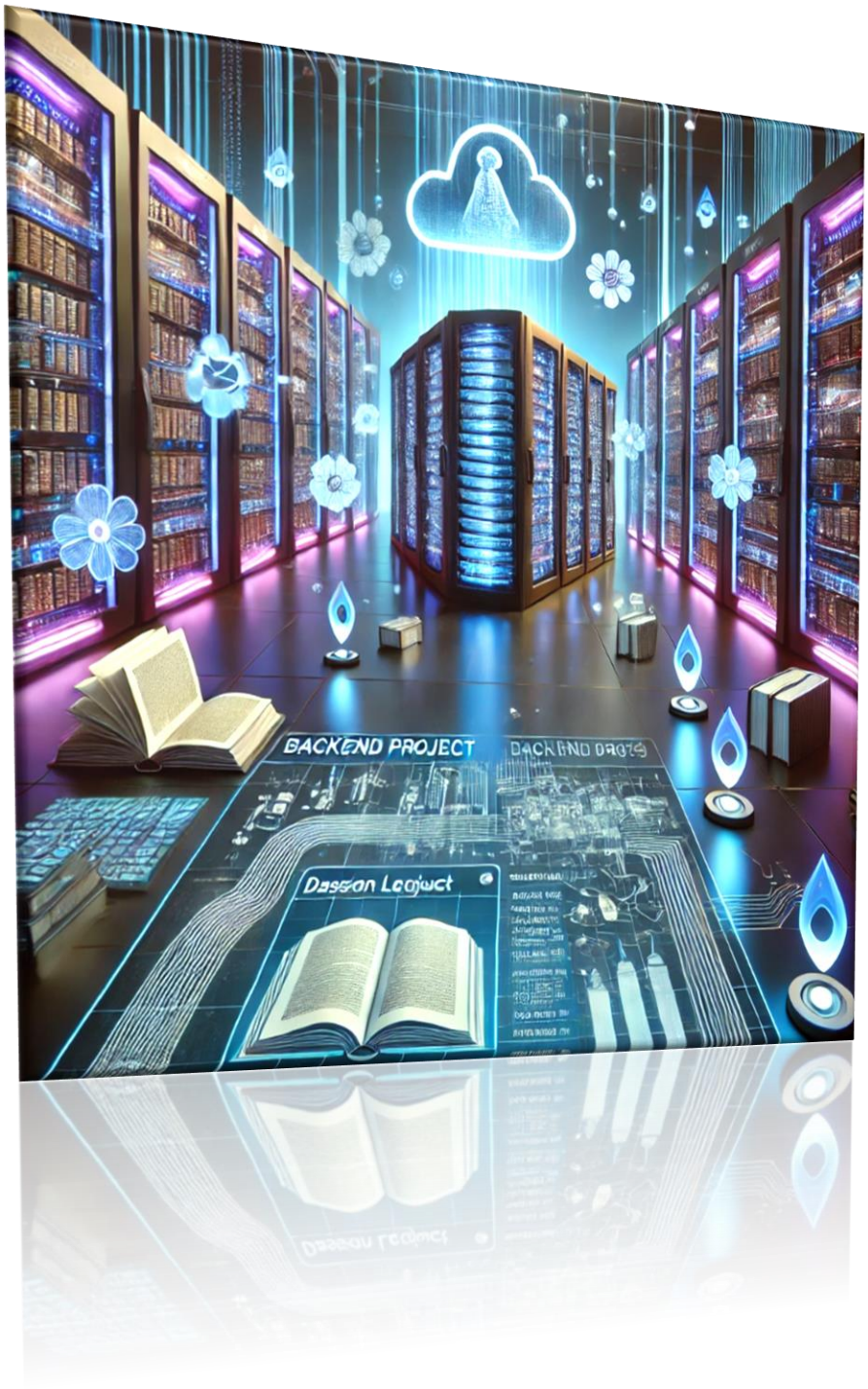


[1]

Site de Vennes

2nd semestre

Denis Romain, Herrera Gonzalo, Moreira Thomas

Chef de projet Charmier G.

[1] Image réalisé avec ChatGPT

Introduction

Le projet Passion Lecture consiste à développer le backend d'une application dédiée aux passionnés de lecture. Ce projet, intégré au module 295, vise à permettre aux utilisateurs de partager, consulter et noter des livres. L'application s'appuiera sur une API REST sécurisée, permettant la gestion des livres, des catégories, des utilisateurs et des interactions entre eux.

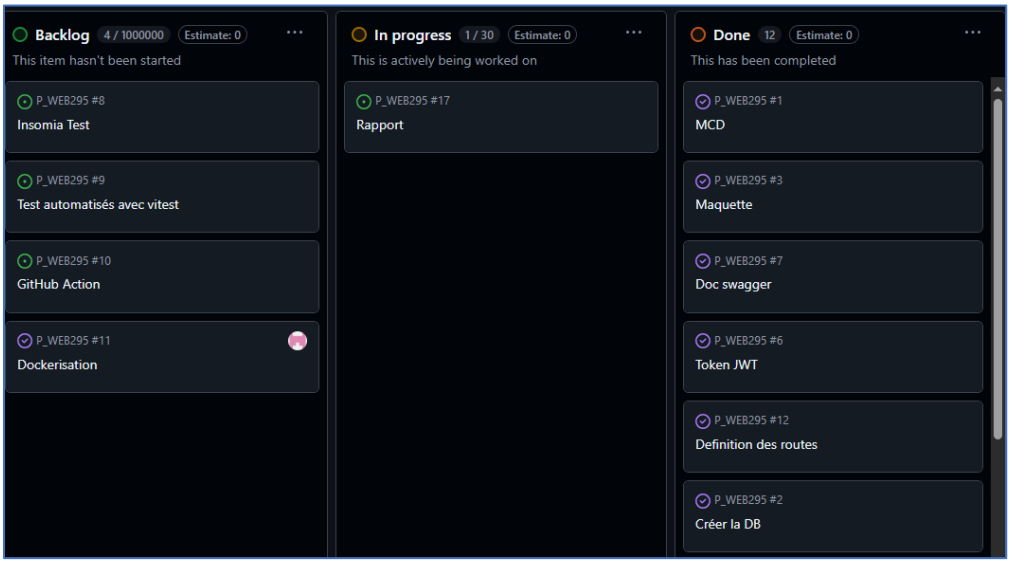
L'API doit intégrer une authentification JWT sécurisée, une validation sécure des données et la gestion des erreurs avec des statuts HTTP appropriés. Pour garantir une expérience utilisateur fluide, une documentation détaillée sera générée via Swagger.

Le backend sera conçu avec Node.js, Express.js et Sequelize pour interagir avec une base de données MySQL. Le projet inclura également une intégration continue via GitHub Actions et une conteneurisation avec Docker pour garantir une meilleure portabilité. Enfin, une attention particulière sera portée à la conception web éco-responsable afin de proposer une application optimisée et conforme aux bonnes pratiques de développement durable.

Analyse

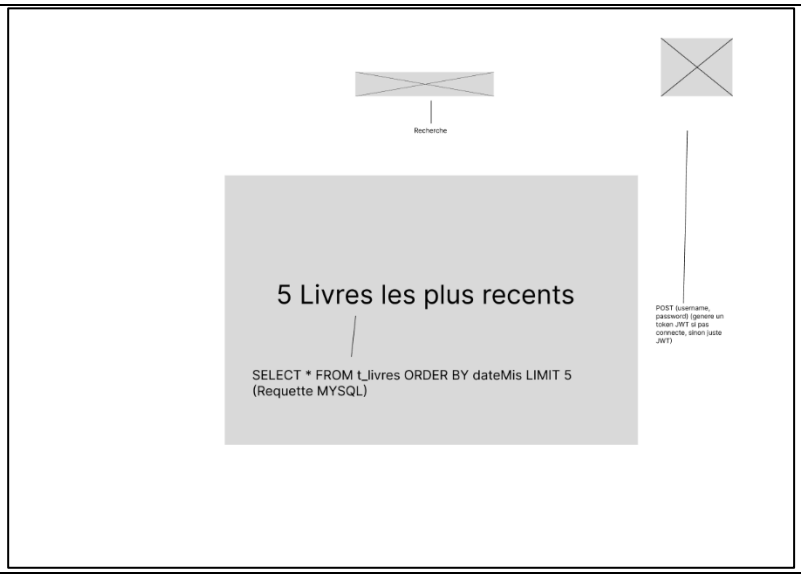
Planification - Gestion de groupe

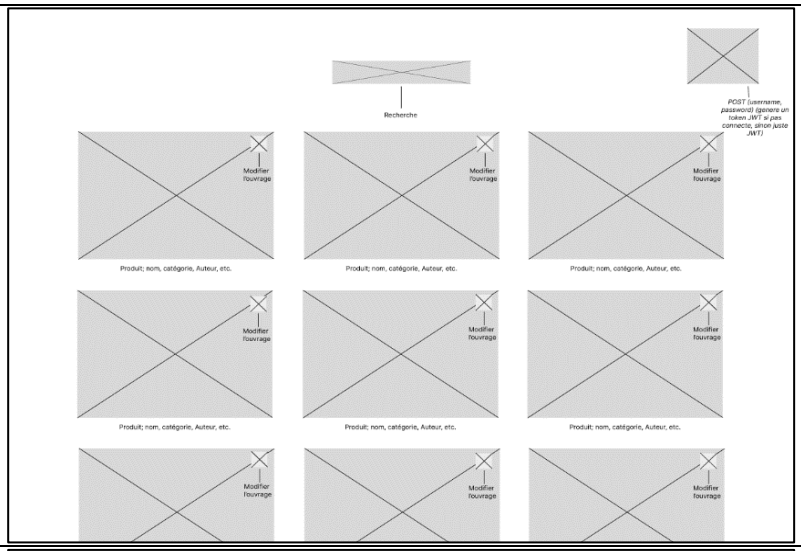
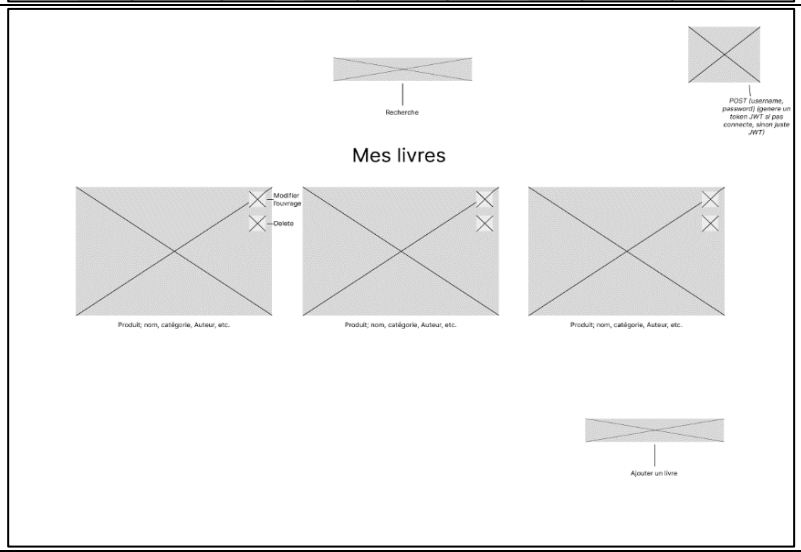
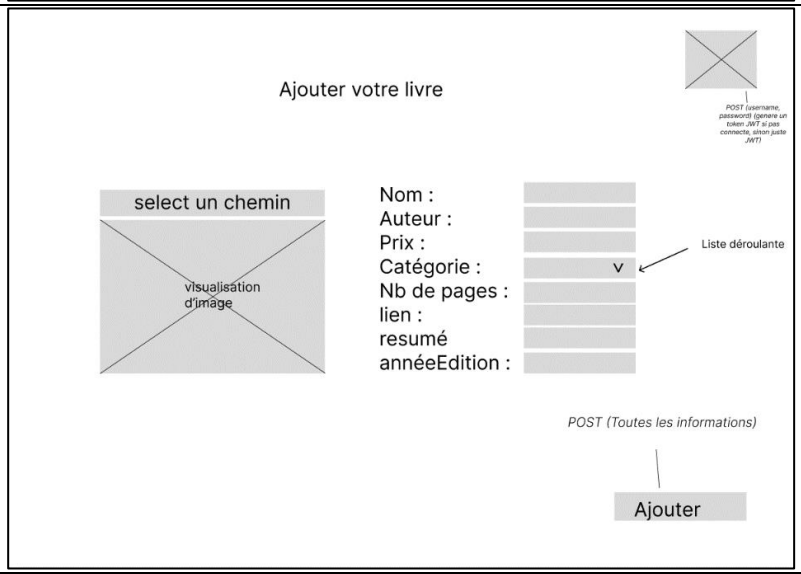
Tout au long de ce projet. Nous avons décidé d'utiliser le projet GitHub afin d'organiser nos tâches et d'être plus efficaces dans notre flux de travail. Nous avons utilisé le modèle de tableau Kanban avec trois colonnes : le backlog, où nous avons mis toutes nos tâches, la colonne de progression, où nous avons mis toutes les tâches sur lesquelles nous travaillions actuellement, et la colonne terminée, pour signaler au reste du groupe qu'une fonctionnalité était terminée. Ci-dessous, une image du tableau Kanban

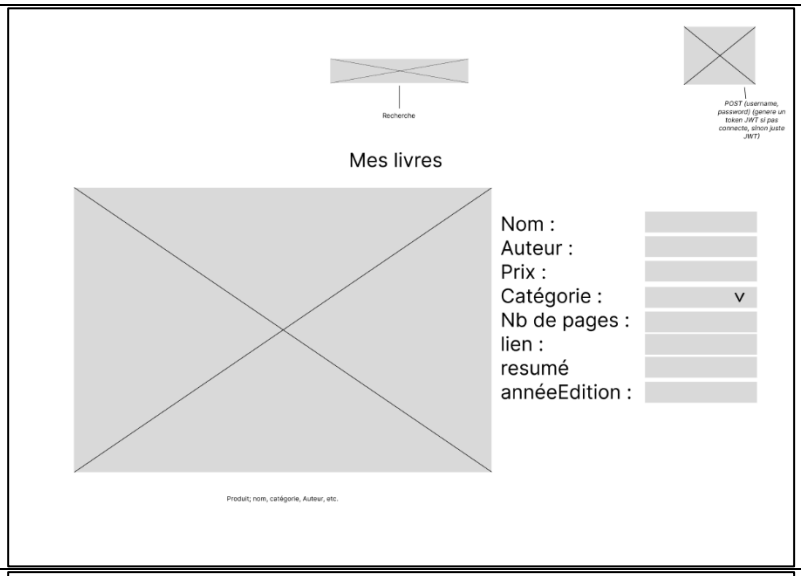
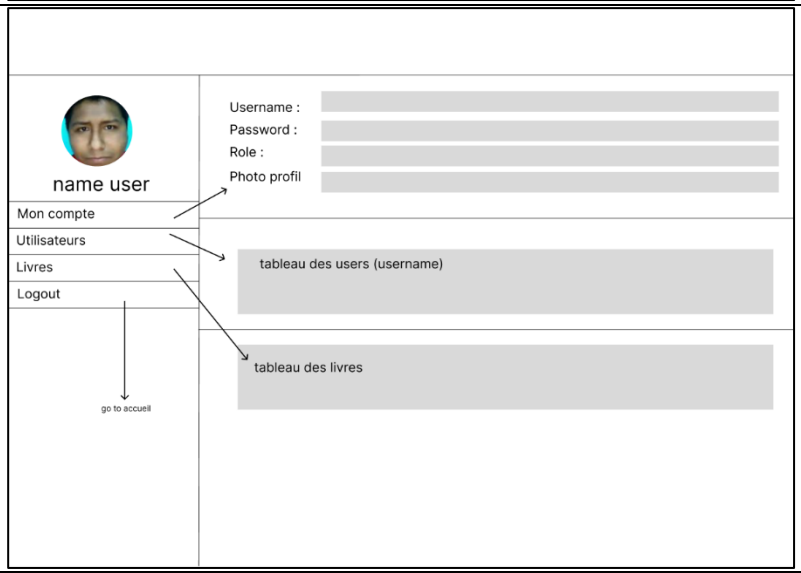


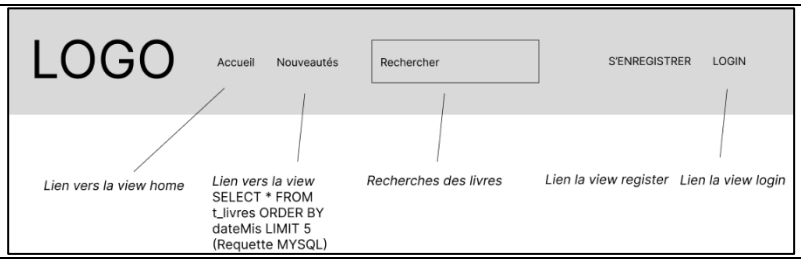
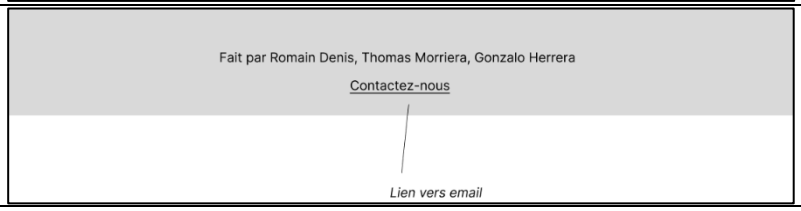
Maquettes

Voilà nos maquettes, vous pouvez suivre le lien du Figma [ici.]

Accueil	

Liste des ouvrages par catégorie	
User	
Ajout d'un ouvrage	

User – modification	
Vue détail d'un livre	
Utilisateur admin	

Header	
Footer	

Routes - API REST

Nous avons divisé l'API en deux grandes catégories :

- **Routes générales** : Récupération des informations de livres, catégories, utilisateurs, authentification, modifications, mis à jour et suppression.
- **Autres routes :** Bien que pas demandés, nous avons décidé de faires les routes pour user.mjs pour la gestion d'utilisateur.

Routes générales

Action	Chemin	Json ?	Description
POST	/livre/	Toute la t_livre + image	Ajoute un livre.
DELETE	/livre/ID/	Toute la t_livre	Supprimer un livre.
PUT	/livre/ID/	Toute la t_livre	Mettre à jour un livre.
GET	/	Non	Prend les 5 livres les plus récents
POST	/livres/:id/comments/	Le contenu + UserID	Poste un commentaire
GET	/livres/:id/comments/	Non	Regarde tous les commentaires d'un livre
POST	/livres/:id/notes/	La note + UserID	Poste une note
GET	/livres/:id/notes/	Non	Regarde toutes les notes d'un livre
GET	/accueil/	Non	Récupère les 5 derniers livres.
GET	/livres/	Paramètres voulus	Cherche un livre avec les paramètres demandés.
GET	/auteur/	Non	Récupère tous les auteurs
GET	/auteur/:id/	Non	Récupère les auteurs correspondant à un ID
GET	/auteur/:id/livres/	Non	Récupère tous les livres d'un auteur
GET	/categories/	Non	Récupère toutes les catégories
GET	/categories/:id/livres/	Non	Récupère tout livres qui correspond à une catégorie
GET	/categories/:id/	Non	Récupère les catégories qui correspondent à l'id

Routes user.mjs :

Action	Chemin	Json ?	Description
GET	/users/	Non	Montre la liste des utilisateurs.
GET	/users/:id	Non	Cherche utilisateur par ID.
GET	/users/:username	Non	Cherche utilisateur par username.
DELETE	/users/:id	Non	Supprimer un utilisateur avec l'ID.
DELETE	/users/:username	Non	Supprimer un utilisateur avec l'username.
PUT	/users/:id	Username & MDP	Mettre à jour un ID.
PUT	/users/:id	Username & MDP	Mettre à jour un username.
POST	/signup/	Username & MDP	Enregistre un nouveau user.
POST	/login/	Username et MDP	Connecte un user.

Analyse de la DB, MCD, MLD, MPD

Voici notre MCD (une copie du fichier looping est dans le répertoire du projet). Il a les associations suivantes :

1. Un livre ne peut avoir qu'une seule catégorie
2. Une catégorie peut avoir plusieurs livres
3. Un livre ne peut avoir qu'un seul éditeur
4. Un éditeur peut avoir plusieurs livres
5. Un livre ne peut avoir qu'un seul auteur
6. Un auteur peut avoir plusieurs livres
7. Un livre appartient et est créé par un utilisateur
8. Un livre peut avoir plusieurs commentaires de plusieurs utilisateurs différents
9. Un livre peut avoir plusieurs notes de plusieurs utilisateurs différents

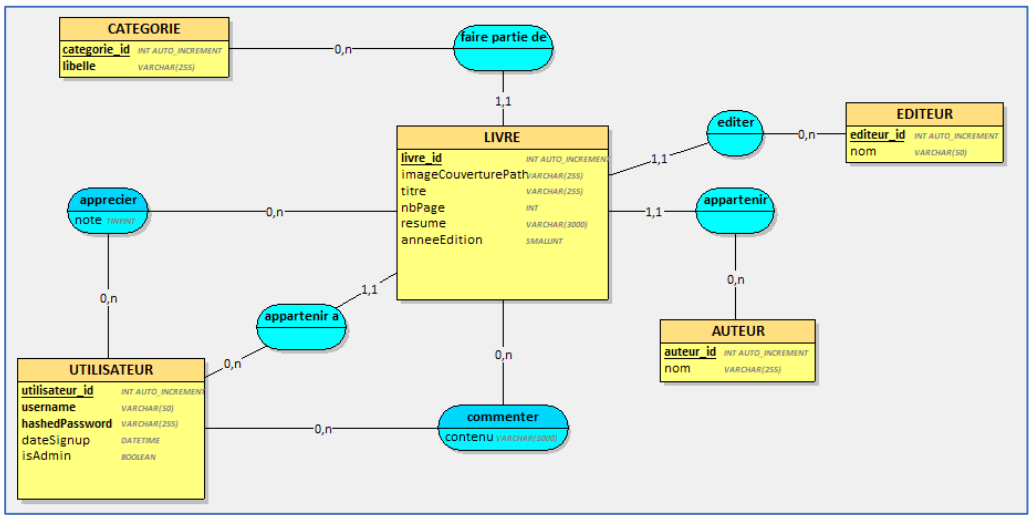
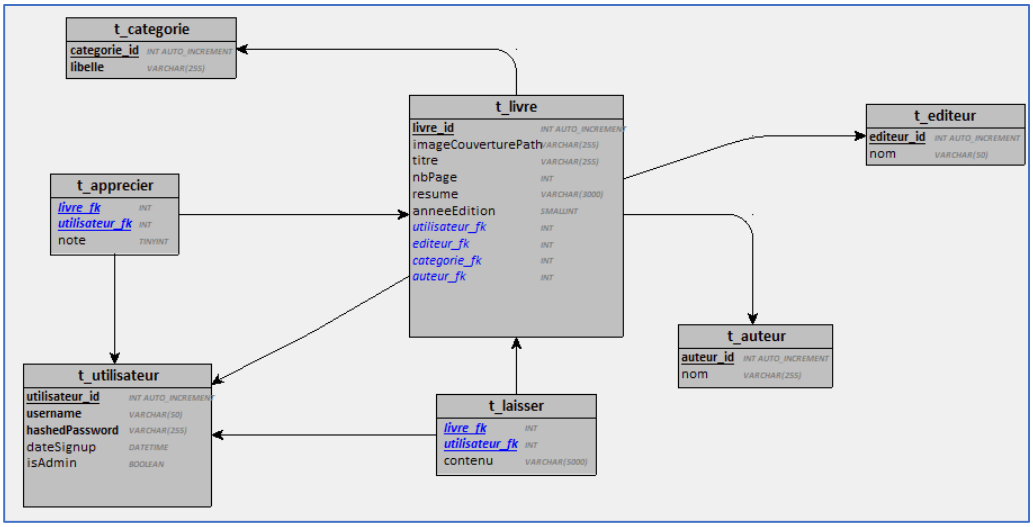


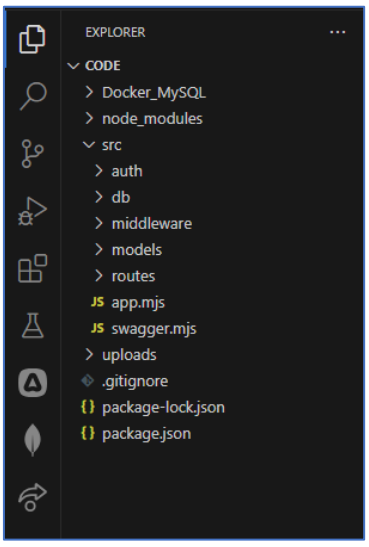
Table	Champ	Type	Description
t_livre	livre_id	Clef primaire	L'id d'un livre
t_livre	imageCouverturePath	Chaine de caractère	Le chemin de l'image de couverture
t_livre	titre	Chaine de caractère	Le titre d'un livre
t_livre	nbPage	Entier	Le nombre de page dans un livre
t_livre	resume	Chaine de caractère	Le résume du livre
t_livre	utilisateur_fk	Clef étrangère	Clef étrangère pour l'id de l'utilisateur du livre
t_livre	editeur_fk	Clef étrangère	Clef étrangère pour l'id de l'éditeur du livre
t_livre	categorie_fk	Clef étrangère	Clef étrangère pour l'id de la catégorie du livre
t_livre	auteur_fk	Clef étrangère	Clef étrangère pour l'id de l'auteur du livre
t_livre	anneeEdition	Entier	L'année d'édition
t_categorie	categorie_id	Clef primaire	L'id d'une catégorie
t_categorie	libelle	Chaine de caractère	Le nom de la catégorie
t_editeur	editeur_id	Clef primaire	L'id d'un éditeur
t_editeur	nom	Chaine de caractère	Le nom de l'éditeur
t_utilisateur	Utilisateur_id	Clef primaire	L'id d'un utilisateur
t_utilisateur	username	Chaine de caractère	Le nom entré par l'utilisateur
t_utilisateur	hashedPassword	Chaine de caractère	Le password entré par l'utilisateur, après avoir été haché
t_utilisateur	dateSignup	Date	La date l'utilisateur a créé son compte
t_utilisateur	isAdmin	Booléen	Si l'utilisateur est admin ou pas
T_apprecier	note	Entier	La note donnée par l'utilisateur a un livre
T_apprecier	Livre_fk	Clef étrangère	Cette table est avec un ID composé : ça

T_apprecier	Utilisateur_fk	Clef étrangère	veut dire que l'id est la réunion des fks de t_livre et t_utilisateur
T_laisser	contenu	Chaine de caractère	Le commentaire donné par l'utilisateur a un livre
T_ laisser	Livre_fk	Clef étrangère	Cette table est avec un ID composé : ça veut dire que l'id est la réunion des fks de t_livre et t_utilisateur
T_ laisser	Utilisateur_fk	Clef étrangère	

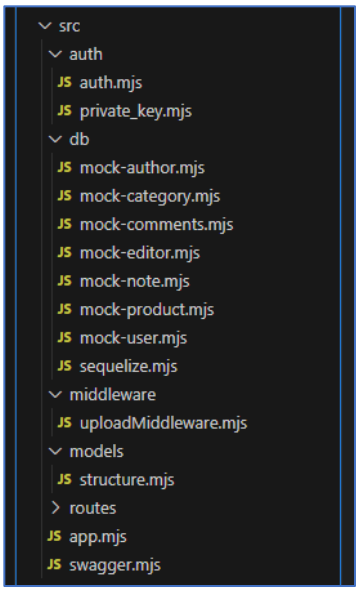


Analyse de la structure du code

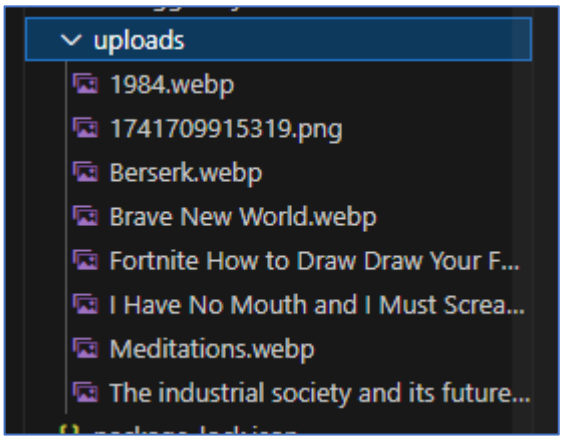
Notre code a été développé en plusieurs fichiers .mjs pour une meilleure compréhension du projet et n'est pas se perdre en travaillant. Voici une image du répertoire de notre code général.



- Dans le dossier **“Docker_MySQL”** se trouve la configuration pour créer notre container Docker et travailler avec.
- Dans le dossier **“node_modules”** se trouvent tous les paquets/librairies qu’on a besoin pour lancer et faire tourne notre projet.
- Dans le dossier “src” se trouve tous les dossiers, dont ils se trouvent des fichiers qu’on a fait pendant le développement de l’application. Aussi il y a deux fichiers, un pour la documentation swagger et autre pour gérer les routes initiales et lancer le serveur avec une connexion à la base de données. Voici une photo avec une vue générale.

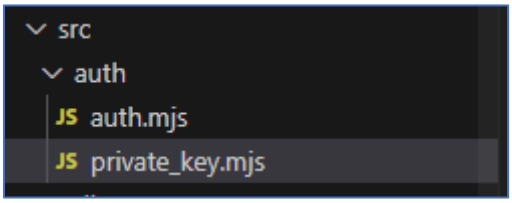


- Dans le dossier “uploads” se trouvent toutes les photos des couvertures des livres. Voici une photo la liste d’images.

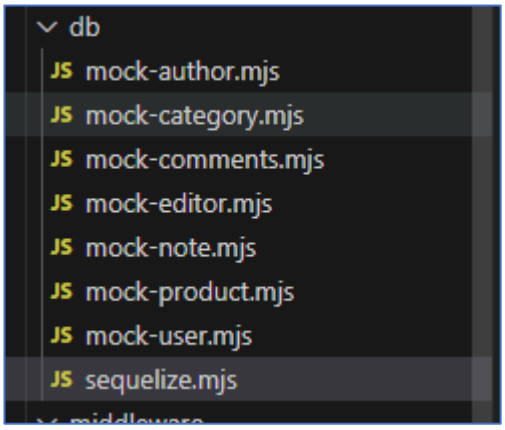


- Dans les fichiers “package-lock.json” et “package-lock.json” se trouve la configuration des dépendances à installer pour le développement du projet. Voici une photo dépendances dont on a eu besoin.

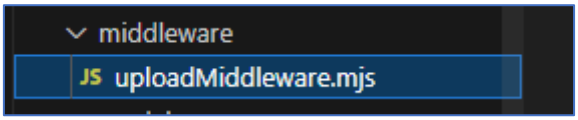
- Dans le dossier **"auth"** se trouvent les fichiers qui gèrent et valident la génération du token JWT avec une clé privée.



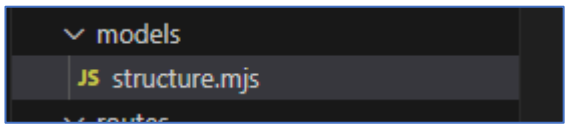
- Dans le dossier **"db"** se trouvent des fichiers mocks pour ajouter dans notre base de données des données fausses afin de tester notre API REST, pour cela il faut que la base de données et la table soient créée avant d'importer ces données. Pour les importations des mocks, on a un fichier **"sequelize.mjs"** qui appel chaque fichier et les importations se font avec un ordre donné pour une bonne cohérence dans la base de données.



- Un dossier **"middleware"** est mis en place avec le propos de pouvoir gérer les chemins des images pour la couverture des livres.



- Un dossier **“models”** est créé avec un fichier **“structure.mjs”** est un des plus importants des dossiers, car crée les tables dans la base de données avec les définitions de relations.



- Un dossier **“routes”** se trouve dans notre projet, dans ce dossier se trouve plusieurs fichiers .mjs et dans chaque fichier se trouvent toutes les routes de chaque type de donnée à vouloir manipuler. Donc pour une meilleure organisation et compréhension du code, on a décidé de faire un fichier pour route et pas un seul fichier.

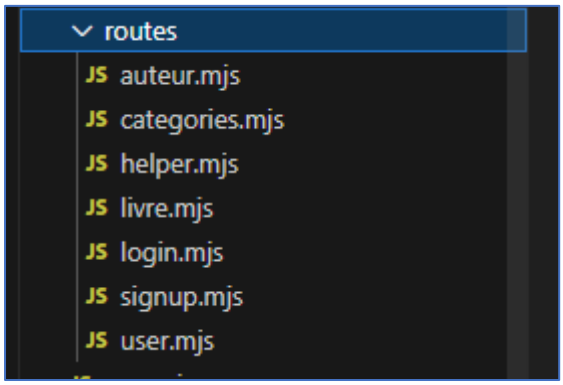
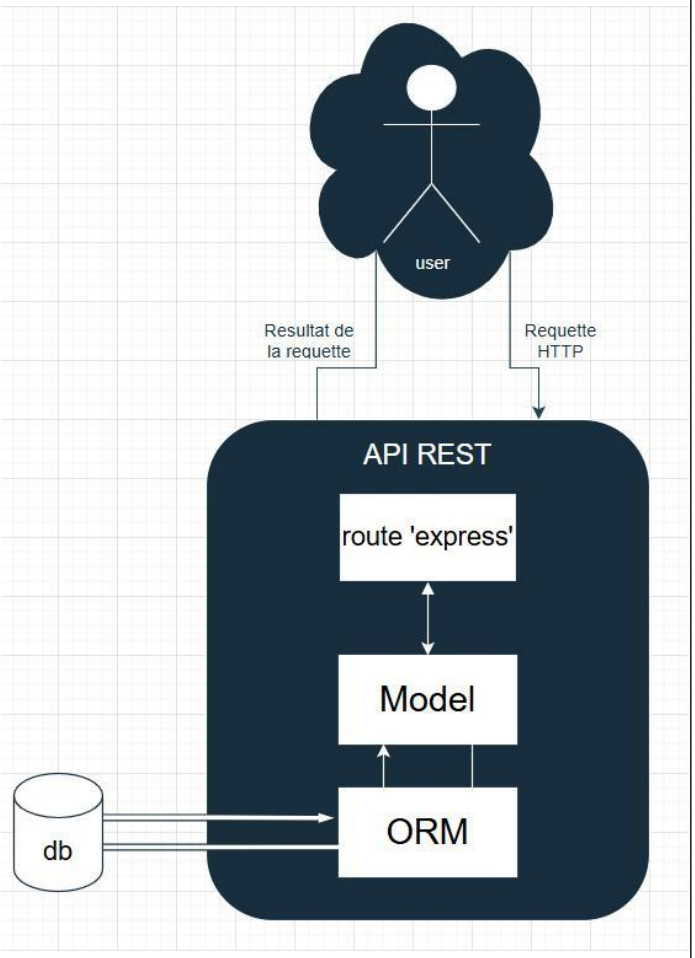


Schéma de l'architecture



1. **User (Frontend)** :
    - L'utilisateur envoie une requête HTTP à l'API REST (via un navigateur ou une application frontend).
    - Il reçoit le résultat de la requête en réponse.
2. **API REST (Backend)** :
    - Gère les requêtes HTTP envoyées par l'utilisateur via des **routes Express** (framework pour Node.js).
    - Ces routes transmettent la requête aux **models**, qui contiennent la logique métier et gèrent les données.
3. **Model** :
    - Représente la structure des données et contient la logique pour interagir avec la base de données.
    - Il utilise un **ORM** (Object-Relational Mapping) pour simplifier les interactions avec la base de données.
4. **ORM (Object-Relational Mapping)** :
    - Sert d'interface entre le **Model** et la **base de données**.
    - Permet d'exécuter des requêtes SQL sans écrire directement du SQL, en manipulant les données sous forme d'objets.
5. **Base de données (db)** :
    - Stocke les informations de l'application.
    - L'ORM envoie les requêtes à la base de données et récupère les résultats pour les transmettre au **Model**, qui les renvoie ensuite aux routes de l'API REST.

Réalisation

L'algorithme utilisé pour gérer l'authentification

D'abord pour tester l'authentification il faut avoir un utilisateur, lequel on peut utiliser celui qui est déjà dans la base de données ou créer un nouveau.

Pour le **registre** d'un nouvel utilisateur, nous avons fait de cette manière :

```
// Endpoint POST pour créer un nouvel utilisateur (inscription)
signupRouter.post("/", async (req, res) => {
  try {
    const { username, password } = req.body; // Extraire username et password du corps de la requête

    // Vérifier si l'utilisateur existe déjà
    const existingUser = await User.findOne({ where: { username } });
    if (existingUser) {
      // Si trouvé, renvoyer un statut 409 (conflit)
      return res.status(409).json({ message: "Ce username est déjà utilisé." });
    }
```

- On extrait l'username et password de la requête
- On compare dans la base de données si cet utilisateur existe déjà ou pas, si c'est le cas donc on retourne un message à l'utilisateur en lui disant que ça existe.

```
// Hacher le mot de passe avec bcrypt (salt de 10)
const hashedPassword = await bcrypt.hash(password, 10);

// Créer un nouvel utilisateur avec la date actuelle et isAdmin à false
const newUser = await User.create({
  username,
  hashedPassword,
  dateSignup: new Date(),
  isAdmin: false,
});
```

- Si l'username n'est pas trouvé dans la base de données, alors on continue et le mot de passe est haché avec un sel aléatoire de 10 caractères.
- La création d'un nouvel utilisateur est fait et enregistré dans la base de données (le rôle par défaut c'est "user" car il existe un seul admin.

```
9   // Route POST pour connecter un utilisateur
10  loginRouter.post("/", (req, res) => {
11    // Recherche de l'utilisateur par username dans la DB
12    User.findOne({ where: { username: req.body.username } })
13      .then((user) => {
14        // Si aucun utilisateur n'est trouvé, renvoyer une 404
15        if (!user) {
16          return res
17            .status(404)
18            .json({ message: "L'utilisateur demandé n'existe pas" });
19        }
20
21        // Comparer le mot de passe fourni avec le mot de passe haché stocké
22        bcrypt
23          .compare(req.body.password, user.hashedPassword)
24          .then((isPasswordValid) => {
25            // Si le mot de passe est incorrect, renvoyer une 401
26            if (!isPasswordValid) {
27              return res
28                .status(401)
29                .json({ message: "Le mot de passe est incorrect." });
30            }
```

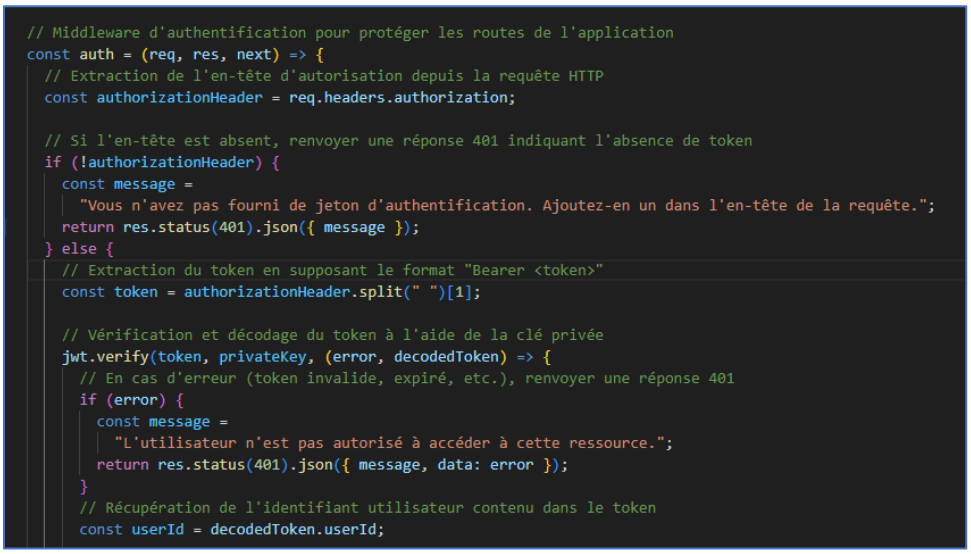
- D'abord on valide avec la base de données si un utilisateur existe ou pas, si ce n'est pas le cas, alors un message est renvoyé à l'utilisateur.
- Si l'utilisateur existe dans la base de données, alors on utilise bcrypt et on hashe le mot de passe fourni par l'utilisateur avec le sel enregistré dans la base de données et on valide si c'est correct ou pas, dans le cas qu'il ne soit pas correct alors un message est renvoyé à l'utilisateur.

```
            // Si le mot de passe est valide, générer un token JWT avec l'ID de l'utilisateur
            const token = jwt.sign({ userId: user.utilisateur_id }, privateKey, {
              expiresIn: "1y", // Le token expire dans 1 an
            });

            // Renvoyer la réponse avec les données de l'utilisateur et le token
            return res.json({
              message: "L'utilisateur a été connecté avec succès",
              data: user,
              token,
            });
          });
        })
        .catch((error) => {
          // En cas d'erreur, renvoyer une réponse 500 et log l'erreur
          console.error("Erreur lors de la connexion :", error);
          return res
            .status(500)
            .json({ message: "Une erreur est survenue", error });
        });
    });
```

- Si l'username et mot de passe sont valides, alors un token JWT est généré pour la session de cet utilisateur avec une expiration d'une année et un message de connexion réussit et renvoyé.
- S'il y a une erreur lors de la connexion, alors un message et renvoyé à l'utilisateur.

Pour la génération d'un **token JWT,** nous avons fait de cette manière :

- Dans un fichier, nous avons créé la constante avec notre clé privée et on l'exporte afin de l'utiliser ailleurs. Ce fichier ou clé ne doit pas être dans un commit.



```
// Middleware d'authentification pour protéger les routes de l'application
const auth = (req, res, next) => {
  // Extraction de l'en-tête d'autorisation depuis la requête HTTP
  const authorizationHeader = req.headers.authorization;

  // Si l'en-tête est absent, renvoyer une réponse 401 indiquant l'absence de token
  if (!authorizationHeader) {
    const message =
      "Vous n'avez pas fourni de jeton d'authentification. Ajoutez-en un dans l'en-tête de la requête.";
    return res.status(401).json({ message });
  } else {
    // Extraction du token en supposant le format "Bearer <token>"
    const token = authorizationHeader.split(" ")[1];

    // Vérification et décodage du token à l'aide de la clé privée
    jwt.verify(token, privateKey, (error, decodedToken) => {
      // En cas d'erreur (token invalide, expiré, etc.), renvoyer une réponse 401
      if (error) {
        const message =
          "L'utilisateur n'est pas autorisé à accéder à cette ressource.";
        return res.status(401).json({ message, data: error });
      }
      // Récupération de l'identifiant utilisateur contenu dans le token
      const userId = decodedToken.userId;
```

- Ce middleware protège les routes en s'assurant que la requête contient un jeton d'authentification. Il vérifie la présence du token dans l'en-tête, l'extrait, puis le décode et le valide à l'aide d'une clé privée. Si le jeton est invalide ou si l'identifiant utilisateur du corps de la requête ne correspond pas à celui du token, l'accès est refusé. Sinon, la requête est autorisée à continuer.

Comprend une explication des mesures prises pour les aspects de sécurité.

L'algorithme utilisé pour gérer la gestion des rôles

La gestion de nos rôles est très simple. Lorsqu'un utilisateur est enregistré, il est défini comme non-administrateur. Pour obtenir le rôle d'administrateur, il faut le modifier dans la base de données, ce qui ne peut être fait que par un administrateur DB. Cela permet d'empêcher un utilisateur d'envoyer une requête se définissant lui-même comme administrateur, et que tous les administrateurs deviennent administrateurs de manière contrôlée.

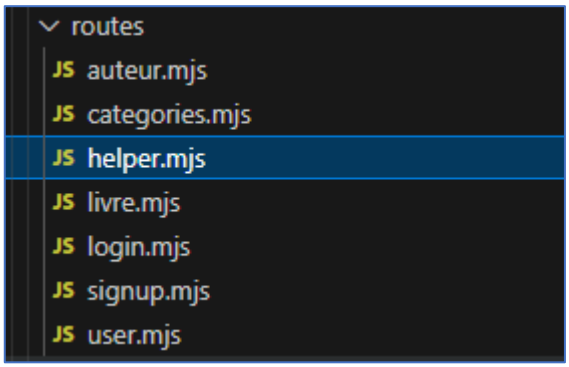
```
// Créer un nouvel utilisateur avec la date actuelle et isAdmin à false
const newUser = await User.create({
  username,
  hashedPassword,
  dateSignup: new Date(),
  isAdmin: false,
});
```

Ecoconception Web

- Nous avons rendu possible l'utilisation d'images. Webp pour les couvertures des livres. Ainsi, le logiciel est plus léger et plus facile à utiliser.
- Nous avons également utilisé l'ORM Sequelize pour optimiser le processus de base de données
- Nous avons également utilisé Express, un Framework très léger.

Un ensemble de routes permettant la gestion des livres, des catégories, des utilisateurs, etc.

Notre backend répond à ce besoin. Il gère également les auteurs et les éditeurs.



Une validation de toutes les données fournies par le consommateur de l'API
Nous validons nos données via tous nos modèles. Par exemple, pour notre table t_apprecier, voici notre validateur :

Il valide les trois types de données de la table, avec leur type et leur longueur, lorsque c'est possible.

```
note: {
  type: DataTypes.TINYINT,
  allowNull: false,
  validate: {
    isInt: { msg: "La note doit être un entier." },
    min: { args: [0], msg: "La note minimale est 0." },
    max: { args: [10], msg: "La note maximale est 10." },
  },
},
livre_fk: {
  type: DataTypes.INTEGER,
  allowNull: false,
  validate: {
    isInt: { msg: "L'identifiant du livre doit être un entier." },
  },
},
utilisateur_fk: {
  type: DataTypes.INTEGER,
  allowNull: false,
  validate: {
    isInt: { msg: "L'identifiant de l'utilisateur doit être un entier." },
  },
},
```

Une gestion des statuts http (200, 3xx, 4xx, 5xx) et des erreurs

Toutes nos routes ont été créées avec des try catch afin de couvrir tous les problèmes possibles pouvant survenir lors d'une requête.

Prenons l'exemple de cette requête :

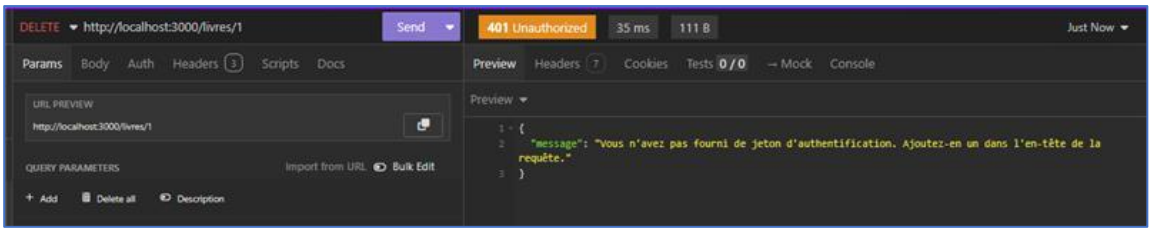
Le code gère les codes d'erreur 400, 404, 201 et 500.

```
livreRouter.post("/:id/notes", auth, async (req, res) => {
  const { note, utilisateur_id, livre_id } = req.body;

  if (
    note === undefined ||
    utilisateur_id === undefined ||
    livre_id === undefined
  ) {
    return res.status(400).json({
      message: "La note, utilisateur_id et livre_id sont nécessaires.",
    });
  }

  try {
    const book = await Livre.findByPk(livre_id);
    if (!book) {
      return res
        .status(404)
        .json({ message: "Le livre demandé n'existe pas." });
    }
```

```
      return res.status(201).json({
        message: "Note ajoutée avec succès.",
        rating: newRating,
      });
    } catch (error) {
      return res.status(500).json({
        message: "La note n'a pas pu être ajoutée.",
        error: error.message,
      });
```

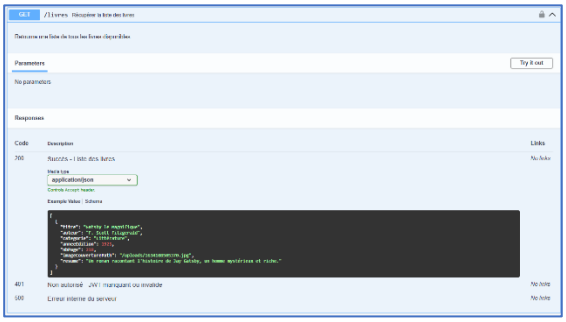
Un système d'authentification basé sur les jetons JWT

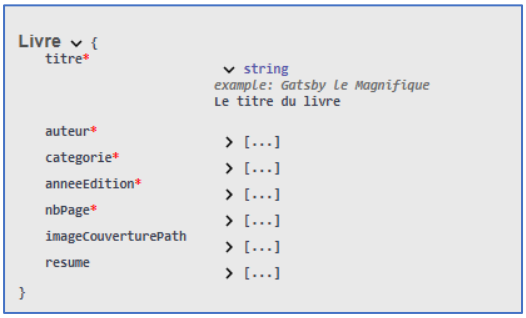
Nous disposons d'un système d'authentification JWT fonctionnel. Si l'utilisateur tente d'effectuer une requête sans jeton JWT valide, la requête sera bloquée. Par exemple, dans cette requête, l'utilisateur tente de supprimer un livre. Sans authentification, l'API bloque la requête et affiche un message indiquant qu'un jeton JWT est requis.



Une documentation Swagger la plus complète possible

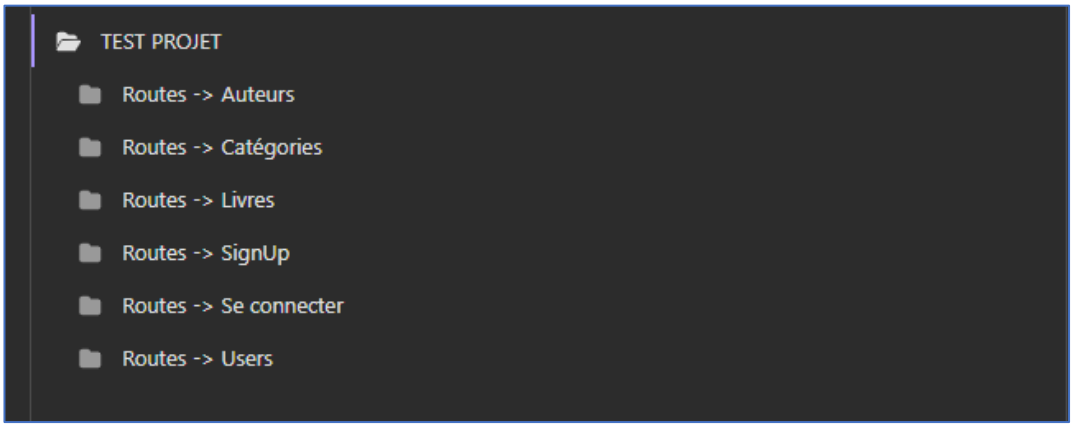
Voici notre documentation Swagger. La première image présente un exemple de commande GET / LIVRES. Elle contient des exemples de valeurs et explique pourquoi chaque code d'erreur peut se produire. La seconde image présente un exemple de table livre, avec le type de chaque champ, un exemple et une description.





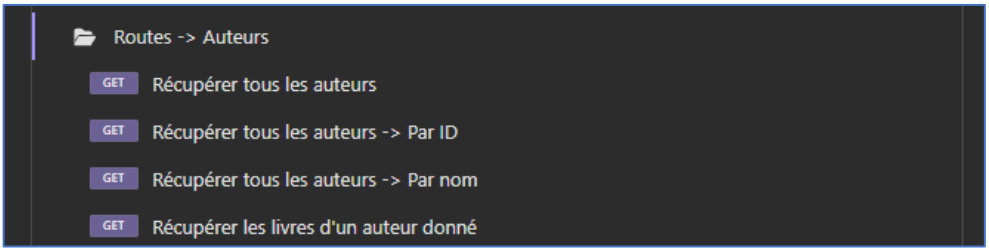
Des tests de votre API avec Insomnia ou Postman

Pour faire de test de notre API Rest nous avons travaillé avec Insomnia. Afin de bien s'organiser et ne pas mélanger les requetés HTTP à faire, nous avons créée des dossiers pour chaque type de route à faire. Voici une image de nos dossiers.

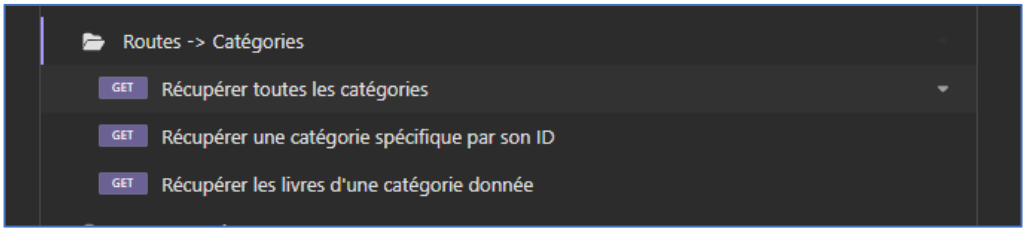


Dans chaque dossier se trouvent toute une liste des requêtes HTTP qui aident à tester notre application :

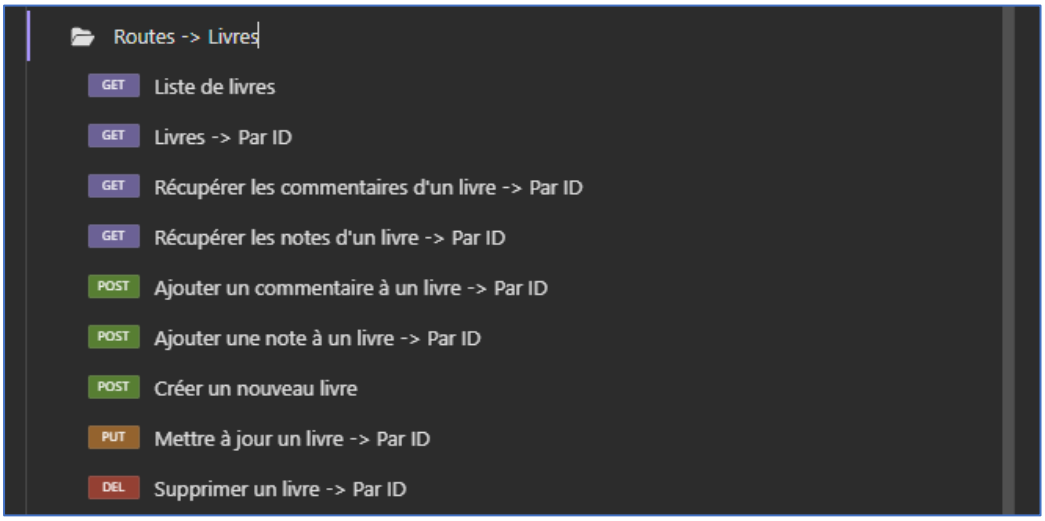
Requêtes pour la route **Auteur** :



Requêtes pour la route **Catégories** :



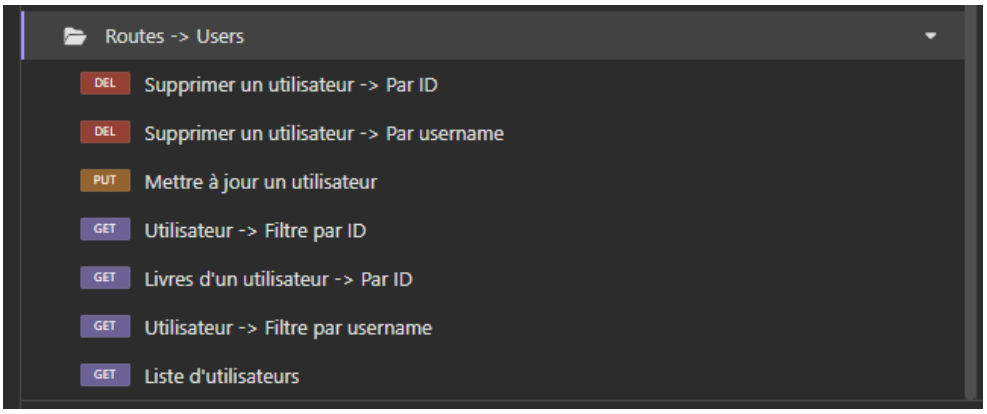
Requêtes pour la route **Livres**:



Requêtes pour la route **SignUp** :

Requêtes pour la route **Se connecter** :

Requêtes pour la route **Users** :



Conclusion

Conclusion générale

En résumé, tout au long de ce projet, nous avons eu l'opportunité d'approfondir nos connaissances et de consolider nos compétences sur différentes notions abordées au cours du module mais aussi d'en acquérir de nouvelles telles Express et Sequelize. La méthodologie que nous avons adoptée pour découper notre projet en 2 parties distinctes (front et back) nous a permis de rester organisé et de travailler les points de chaque pile technologique de manière approfondie.

Néanmoins, il demeure des axes d'amélioration, nous aurions par exemple souhaité une utilisation plus optimisée de GitHub, qui aurait pu améliorer la gestion du projet et la collaboration. Afin de mieux partager des réflexions au sein de l'équipe.

De plus, bien que le projet ait été intéressant, nous aurions apprécié un peu plus de complexité afin de relever de nouveaux défis. Concernant la documentation, certains ont ressenti une certaine redondance vers la fin.

En termes de collaboration, le groupe a fonctionné de manière fluide, avec une bonne répartition des tâches et une implication équilibrée de chacun. Cette bonne dynamique nous motive pour la suite du projet, et nous avons hâte de travailler sur le développement du frontend ensemble.

Conclusion Romain Denis

En conclusion, j'ai vraiment apprécié ce projet. Je pense qu'avoir un module séparé pour le backend et le frontend serait un choix judicieux pour le module frontend. Je trouve le projet assez simple et je pense qu'il aurait été plus captivant s'il avait été plus complexe, car il s'agissait simplement d'une version plus grande de ce que nous avions

fait dans le module, sans trop de réflexion. Je pense que le projet GitHub a été un peu négligé et que le groupe dans son ensemble aurait pu l'utiliser davantage. Concernant la collaboration, je trouve que notre groupe a été très fluide et que chacun a très bien joué son rôle. J'ai hâte de retravailler sur le projet frontend avec eux.

Conclusion Gonzalo Herrera

En conclusion, je trouve ce projet intéressant. Il m'a permis d'approfondir les concepts du module que je ne maîtrisais pas encore parfaitement. Il m'a également permis de comprendre la structure d'un projet comme celui-ci, que nous utiliserons dans nos travails après notre formation. Je pense aussi que j'aurais été très confus si nous avions fait le frontend et le backend dans le même module. La seule chose que je changerais, c'est la quantité de documentation, car c'est devenu assez répétitif vers la fin.

Conclusion Thomas Moreira

Ce projet a été une très bonne expérience. J'ai pu me familiarise avec Node js, Express & Sequelize, tout en travaillant sur une API REST bien structurée, avec une authentification JWT.

Le fait que le backend & le frontend soit deux modules séparer est vraiment bien. Le coté GitHub n'a pas été beaucoup explorer, cela ne me dérangeant pas. Tout s'est bien passé dans l'équipe, impatient pour le frontend.

Critique sur la planification

La gestion des tâches était clairement à revoir. Bien que nous ayons utilisé GitHub Project pour catégoriser nos tâches, la mise à jour était partielle et, parfois, pas organisée. Au lieu de systématiquement tenir à jour les tâches du GitHub Project, nous discutions justement des tâches entre nous, créant alors un suivi oral sans beaucoup de forme.

Un de nos plus gros obstacles était le méga planning fait au lancement. En effet, celles-ci étaient très grandes et larges pour vraiment viser un objectif précis. Nous aurions dû les découper en petites tâches précises afin de les suivre au fur et à mesure et de facilité le travail de chacun.

De plus, étant seulement trois dans l'équipe, l'utilisation de GitHub Project semblait un peu bizarre, car cet outil donne l'impression d'être fait pour des plus grandes équipes.

Pour le prochain projet, il faudrait donc nous discipliner à catégoriser les tâches de manière plus précise et lisible, où chacun se doit de compléter une tâche, afin de bien suivre le projet tout au long de sa création. En effet, cela facilite, indéniablement, la répartition du travail et le suivi des tâches.

Utilisation de l'IA

Nous avons utilisé l'IA uniquement pour s'informer, comprendre nos erreurs & bugs, en aucun cas pour effectuer le travail à notre place.

Webographie

- romaindenis1/P_WEB295
- Backlog · P_WEB295
- Figma
- ChatGPT