

Dossier Projet DWWM

MyBook

Yanis Bennadji

2025



MyBook

Sommaire

1. Présentation	4
2. Compétences mises en œuvre	6
3. Cahier des charges	8
• Objectifs fonctionnels	
• Charte graphique	
4. Environnement technique	10
• Langages et frameworks utilisés	
• Outils de développement	11
• Librairies, API et services externes	12
5. Réalisation Frontend	13
• Maquettes de l'application	
• Interfaces statiques	15
• Interfaces dynamiques	16
6. Réalisation Backend	20
• Mise en place de la base de données	
• Développer des composants d'accès aux données SQL	24
• Ouverture NoSQL	28
• Développer des composants métier côté serveur	29
• Sécurité et bonnes pratiques	32
• Tests unitaires	34
7. Déploiement	36
8. Veille technologique	37
9. Améliorations possibles	39
10. Conclusion	40

Présentation

Présentation personnelle :

Je m'appelle Yanis Bennadji j'ai vingt-trois ans, j'habite et je suis originaire de Marseille. Depuis très jeune étant passionné par le domaine de l'informatique, le numérique et tout ce qui peut toucher à l'électronique, j'ai rejoint l'école LaPlateforme_ pour en apprendre plus sur le monde du développement web. Au cours d'une formation de 16 mois j'ai pu renforcer les acquis que j'avais déjà en autodidacte sur le développement web et continuer d'apprendre dans un bon cadre.

Présentation de la formation :

La formation CDPI pour le titre de DWWM organisé par LaPlateforme est donc une formation de 16 mois dont 17 semaines de stages au sein d'une entreprise, dans mon cas L'Atelier_ qui m'a permis une mise en application direct sur des cas concrets en entreprise. Ayant eu la chance d'avoir une longue période de formation j'ai pu acquérir différentes compétences que ce soit sur la conception de projet, le travail en équipe ou bien le Front-end et Back-end.

Présentation du projet :

Ce dossier est la représentation des compétences que j'ai pu acquérir durant ma formation CDPI avec l'école LaPlateforme_ pour le passage du titre professionnelle de Développeur Web et Web Mobile. Il abordera la conception,

développement, et production de mon projet perso MyBook qui valide l'acquisition des huit compétences demandés.

J'ai réalisé ce projet en autonomie complète grâce à l'expérience que j'ai pu obtenir au sein de la formation.

Le projet **MyBook** viens d'une idée de ma part, étant fan de cinéma, j'utilise souvent Letterboxd pour répertorier les films que j'ai pu voir dans le temps, c'est un excellent outils pour noter, donner une critique et partager avec les autres utilisateurs autour de différents films. Etant également fan de littérature j'ai chercher une alternative à Letterboxd pour la lecture, il en existe plusieurs la plus populaire étant Goodreads sauf que j'ai rencontré un soucis avec son usage, j'ai trouvé l'interface tout sauf agréable et intuitive pour l'utilisateur.

A partir de ce constat là je me suis dit, pourquoi pas faire mon propre carnet de lecture en ligne inspiré de Letterboxd avec une interface plus à mon image.

Compétence mises en oeuvres

1. Frontend

Maquetter des interfaces utilisateurs web ou web mobiles :

En préparation de la réalisation de mon application MyBook, j'ai commencé par travailler sur les documents de conception. J'ai rapidement réfléchi à l'expérience utilisateur ainsi qu'aux fonctionnalités possibles de l'application. L'étape suivante a donc été de préparer une charte graphique et au moins une maquette, afin d'avoir un premier aperçu de l'apparence de l'application.

Réaliser une interface statique web et adaptable :

Dans le but de rendre l'application accessible à tous et simple d'utilisation, quel que soit l'âge des utilisateurs, il était essentiel de concevoir une interface répondant à ces critères. Il fallait également prendre en compte l'adaptabilité à tous types de supports, y compris les appareils mobiles, afin de garantir un accès en toutes circonstances.

Developper une interface utilisateur web et dynamique :

Comme pour la partie frontend, la partie backend a été réfléchie en amont. J'ai donc commencé par définir les données nécessaires à mon application ainsi que leur structure. J'ai mis en place un MCD et un MLD, puis j'ai choisi PostgreSQL comme base de données pour sa fiabilité et sa compatibilité avec Prisma.

2. Backend

Mettre en place une base de donnée relationnelle :

Comme pour la partie Frontend, la partie Backend était réfléchis en amont, j'ai donc commencé à réfléchir à ce que je voulais comme données dans mon application et leur structures également. J'ai donc mis en place un MCD et MLD, pour la base de donnée j'ai plus tard choisis PostgreSQL.

Développer des composants d'accès aux données SQL et NoSQL :

Dans mon projet MyBook, j'ai utilisé Prisma pour gérer les interactions avec ma base de données relationnelle. J'ai également eu l'occasion de me former aux bases NoSQL, comme MongoDB, ce qui m'a permis de mieux comprendre les différences et les cas d'usage entre SQL et NoSQL.

Développer des composants métier coté serveur

Après avoir mis en place la base de données et les routes, j'ai développé la logique métier de l'application, comme l'authentification (connexion, inscription), la gestion des avis, l'ajout de livres à la collection et certaines règles spécifiques (par exemple : empêcher un utilisateur de noter un livre plusieurs fois).

Cette logique a été structurée dans des contrôleurs Express, avec des middlewares pour sécuriser les routes via les tokens JWT.

Cahier des charges

Objectif fonctionnels :

Une fois l'idée en tête il était temps de commencer à préparer tout cela en me donnant des objectifs fonctionnels pour l'application web.

L'objectif principal de **MyBook** est de proposer une plateforme (dans le futur social) dédiée au lecteurs, leur permettant de suivre, noter et de laisser un avis sur leurs lectures au fil du temps avec une interface intuitive et moderne.

Les fonctionnalités attendus en priorité sont :

- Rechercher un livre via API à partir d'un titre ou d'un auteur ;
- Créer un compte utilisateur sécurisé avec gestion par mail et mot de passe ;
- Se connecter / déconnecter et gérer sa session utilisateur ;
- Ajouter un livre à sa collection personnelle avec une date de fin de lecture ;
- Attribuer une note et rédiger un avis sur un livre lu ;
- Consulter sa collection personnelle et suivre ses différentes lectures au fil du temps ;
- Personnaliser son profil utilisateur avec une bio, un avatar et ses livres préférés mis en avant ;
- Ajouter des livres à ses favoris et définir un top 4 personnel ;
- Accéder au profil publics d'autre utilisateurs pour consulter leur activités récentes ;

- Une page administration pour l'admin du site permettant de gérer les différents utilisateurs ainsi que les reviews laissé par ces derniers ;
- Naviguer facilement sur le site grâce à une interface responsive claire et ergonomique.

Charte graphique :

Pour la charte graphique et la direction artistique du site, je voulais comme couleur principale du vert sapin, quelque chose d'assez sobre qui peut rappeler le confort et la tranquillité de la lecture.

J'ai donc choisis comme couleur principale :



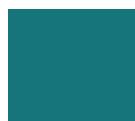
- Un vert assez sombre



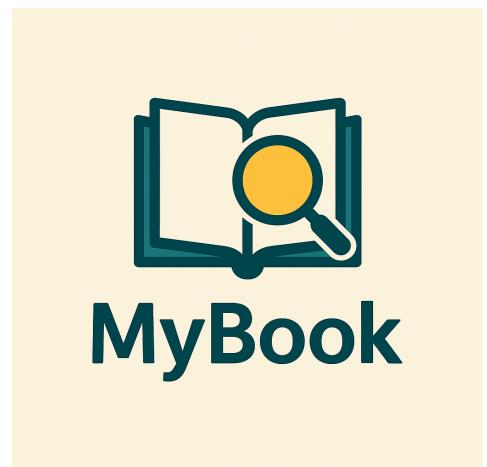
- Un jaune



- Un Beige



- Un bleu canard



C'est les couleurs que l'on retrouve sur le logo de MyBook, j'ai bien évidemment garder du blanc, du noir histoire de garder un thème assez sobre pour le site.

Environnement technique

Langages et Frameworks utilisés :

Pour le projet j'ai décidé de faire un environnement en full JavaScript, que ce soit côté client ou serveur, c'est le langage que je préfère utiliser actuellement en m'appuyant sur différents Frameworks et outils.

Frontend :

- Javascript
- React.js pour les interfaces
- Vite pour l'initialisation du projet
- TailwindCSS pour le style
- React Router pour la navigation côté client
- Context API pour la gestion de l'état global

Backend :

- Node.js comme environnement d'exécution
- Express.js en serveur léger pour la création d'API REST
- Prisma en ORM pour la gestion de la base de donnée
- PostgreSQL pour la base de donnée relationnelle

Architecture du projet :

Le projet est structuré selon l'architecture MVC (Model, View, Controller) côté backend pour séparer de manière claire les fichiers.

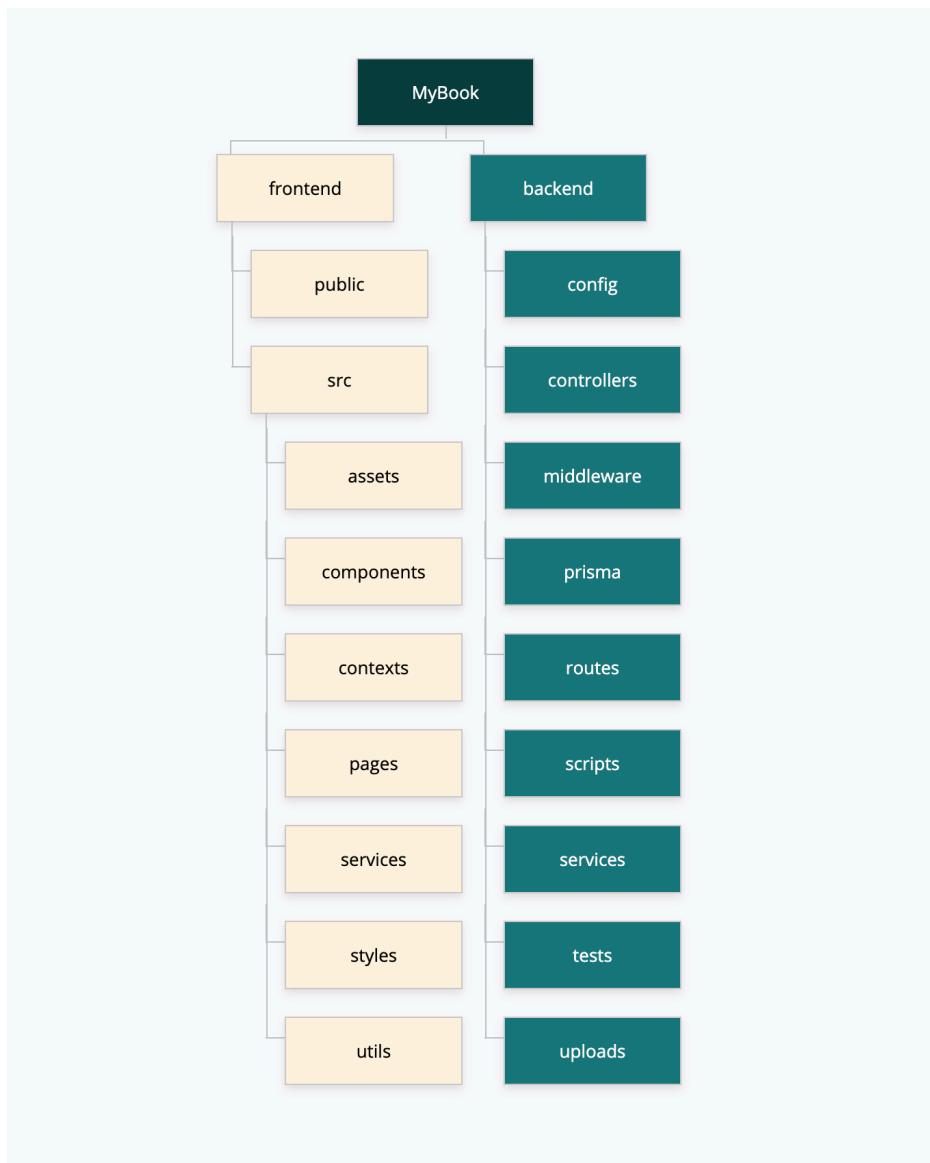
Notre modèle déclaré en utilisant Prisma via le fichier schema.prisma

Nos Controllers responsable de la logique métier, des validations et des interactions entre les données et l'utilisateur.

Les routes connectés au Controllers pour traiter les requêtes REST.

Côté frontend le projet est basé sur React.js donc une architecture de composants réutilisables regroupés sous différents dossier, pages, services, components.

Voilà à quoi ressemble l'arborescence du projet :



Outils de développement :

- Visual Studio Code (VSCode) comme éditeur de code
- Prettier, ESLint, TailwindCSS IntelliSense comme extension de VSCode

- Git et Github pour la gestion des versions ainsi que le déploiement futur
- PostgreSQL et Prisma Studio pour la visualisation de la base de donnée

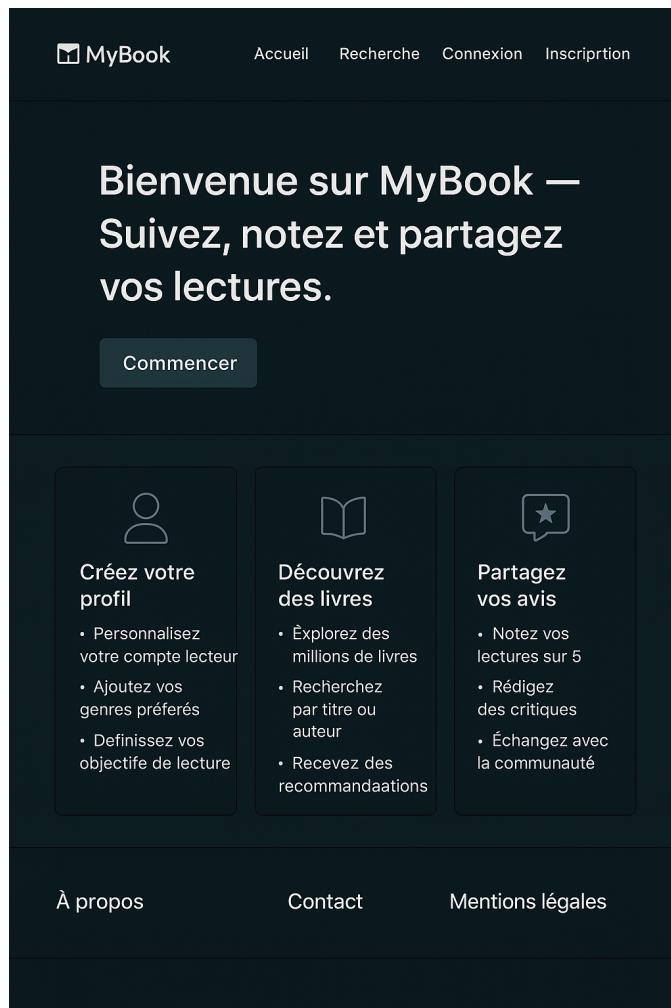
Librairies, API et services externes :

- Authentification (JWT, bcryptjs)
- Uploads (multer)
- Emails (nodemailer)
- UI (Material UI, Tailwind, Emotion)
- Parsing / SEO (html-react-parser, react-helmet-async)
- Tests (jest, supertest)
- Dev Tools (ESLint, nodemon, Vite...)

Réalisation Frontend

Maquette de l'application :

Avant de commencer les maquettes de mon projet MyBook, j'avais déjà une idée précise du rendu que je souhaitais pour le site. Le projet étant initialement inspiré de Letterboxd, je voulais vraiment m'appuyer sur leur interface, que je trouve très intuitive. En utilisant Figma et les différents outils qu'il propose, notamment pour la génération et la création de maquettes, j'ai pu concrétiser mon idée pour la page d'accueil, qui serait composée de trois cartes destinées à présenter le site web.



Cette maquette est la première version avant même d'avoir voir les couleurs du logo dessus, elle a servis à poser les bases par la suite du projet.

L'autre page qui à mon sens était importante à travailler était la page « Diary » ou aussi appeler journal de lecture, je voulais un carnet vraiment dans la même inspiration que Letterboxd car je trouve la mise en forme parfaite pour trier les lectures. Voici le rendus chez Letterboxd :

29		Thelma & Louise	1991	★★★★★	♥	≡	✎	...
06		Zodiac	2007	★★★★★	♥	≡	✎	...
19		Didi (弟弟)	2024	★★★★★	♥	≡	✎	...
17		Better Days	2019	★★★★★	♥	≡	✎	...
05		Memories of Murder	2003	★★★★★	♥	≡	✎	...
25		Conclave	2024	★★★★★	♥	≡	✎	...
24		V for Vendetta	2005	★★★★★	×	♥	≡	...
20		Kingsman: The Secret Service	2014	★★★★★	♥	≡	✎	...
23		The Gentlemen	2019	★★★★★	♥	≡	✎	...
21		In the Mood for Love	2000	★★★★★	♥	≡	✎	...
18		Anora	2024	★★★★★	♥	≡	✎	...
18		Gladiator	2000	★★★★★	♥	≡	✎	...
16		My Book	2024	★★★★★	♥	≡	✎	...

On retrouve les informations les plus importantes comme le nom du film, l'affiche, la date de visionnage ou encore la note donné par l'utilisateur. Donc en partant de cela j'ai donc fais ma page pour mon projet MyBook à mon tour.

The screenshot shows a web-based application for managing a reading list. At the top, there's a header with the logo 'MyBook', a search bar containing 'Rechercher un livre...', and a user profile 'Sinayz'. Below the header, a section titled 'Mes Livres Lus' (My Read Books) displays two entries:

- Mai 2025**: Shows a book entry for 'Blackwater 1 - La crue' by Michael McDowell, rated 5 stars, with the note 'Superbe roman d'horreur !'. The date is listed as MAI 6 2025.
- Avril 2025**: Shows a book entry for 'L'Assassin royal (Tome 1) - L'Apprenti assassin' by Robin Hobb, rated 5 stars, with the note 'Incroyable coup de coeur.'. The date is listed as AVR 23 2025.

A large magnifying glass icon is located on the right side of the main content area.

Réaliser des interfaces utilisateur statiques web ou web mobile :

En créant ce projet j'avais l'intention qu'il soit accessible sur différents formats d'écran possible donc que ce soit du Desktop, du mobile, tablette ou autre, j'ai donc utiliser la breakpoints responsive design de TailwindCSS. En effet avec TailwindCSS il est possible d'utiliser leur breakpoints déjà définis pour les 5 formats d'écran les plus courants, les suivants :

Breakpoint prefix	Minimum width	CSS
'sm'	40rem (640px)	'@media (width >= 40rem) { ... }'
'md'	48rem (768px)	'@media (width >= 48rem) { ... }'
'lg'	64rem (1024px)	'@media (width >= 64rem) { ... }'
'xl'	80rem (1280px)	'@media (width >= 80rem) { ... }'
'2xl'	96rem (1536px)	'@media (width >= 96rem) { ... }'

Grâce à cela j'ai pu construire une interface responsive pour la totalité de mon site :

Ajouter un livre version Desktop

Ajouter un livre version mobile

Développer une interface utilisateur web dynamique :

Mon projet MyBook est donc basé entièrement autour de React.js ce qui va permettre à l'utilisateur d'avoir une expérience fluide et d'interagir en temps réel avec le contenu de la page sans recharge. L'ensemble des interactions repose sur l'utilisation des state management (useState, useEffect) permettant à l'interface de réagir en fonction des différentes données reçues de l'API de Google ou de mon backend.

L'une des premières fonctionnalités sur lesquelles j'ai travaillé était la connexion à l'API de Google pour la recherche des livres de manière dynamique. Pour cela j'ai donc commencé par récupérer ma clé API auprès de Google et ensuite la stocker de manière sécurisée dans un fichier .env dans mon dossier frontend sous cette forme.



1 VITE_GOOGLE_BOOKS_API_KEY=your_google_books_api_key_here

Une fois celle-ci stockée de manière sécurisée il me fallait maintenant créer un service pour l'API de Google que j'ai nommé googleBooksApi.js avec différentes optimisations pour gérer les requêtes de l'API. En effet l'API de Google permet 1000 requêtes quotidiennes et lors de la phase de développement celles-ci peuvent monter très vite, j'ai donc instauré un cache pour éviter de faire des appels API identiques dans un délai de 5 minutes. J'ai également mis en place un throttling histoire d'optimiser le tout et de limiter à une requête toutes les deux secondes pour éviter la limite.

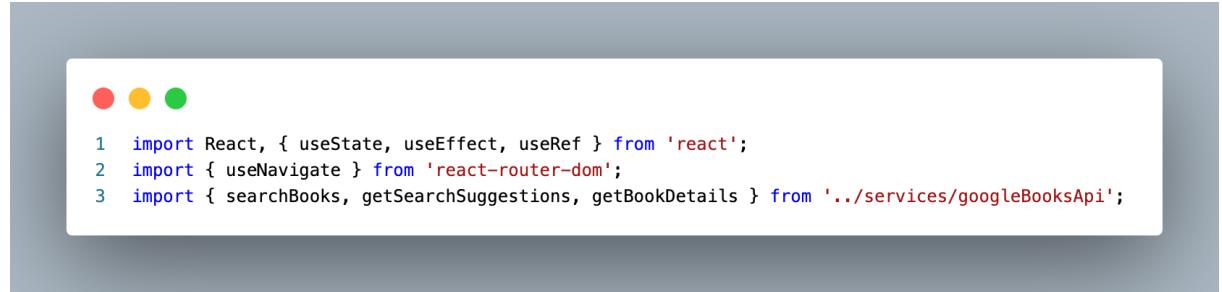
Dans ce service qui gère la recherche de livre via l'API j'ai trois fonctions principales :

1. searchBooks : Cette fonction va permettre de rechercher les livres par mot-clé (titre, auteur), garder en cache les résultats, transformer la réponse de l'API pour avoir des informations plus claires dans un format simplifié et en cas d'erreur renvoie un message d'erreur clair.
2. getBookDetails : Pour celle-ci on récupère les informations d'un livre grâce à son ID, toujours une mise en cache pour éviter de répéter un appel API.
3. getSearchSuggestions : Celle-ci est conçue pour alimenter la fonction d'autocomplétion implémentée dans ma barre de recherche, elle permet de garder en cache les touches tapées par l'utilisateur pour lui proposer une suggestion de livre avec ce qu'il a écrit. J'ai volontairement laissé un délai de 300ms pour que si l'utilisateur tape rapidement « H A R », le service envoie une seule requête plutôt qu'une différente pour chaque lettre.

Voici un exemple d'une des fonctions, la principale, searchBooks :

```
1  export const searchBooks = async (query, options = {}) => {
2    try {
3      const cacheKey = `search-${query}-${JSON.stringify(options)}`;
4      const cached = cache.get(cacheKey);
5
6      // Return cached results if valid
7      if (cached && Date.now() - cached.timestamp < CACHE_DURATION) {
8        return { results: cached.data, error: null };
9      }
10
11    const searchFn = async () => {
12      const params = {
13        q: query,
14        key: API_KEY,
15        maxResults: options.maxResults || 12,
16        orderBy: options.orderBy || 'relevance',
17        printType: options.printType || 'books',
18        filter: options.filter || 'partial',
19        langRestrict: 'fr'
20      };
21
22      const response = await axios.get(BASE_URL, { params });
23
24      if (!response.data.items) {
25        return [];
26      }
27
28      // Transform API response to our application model
29      return response.data.items.map(item => ({
30        id: item.id,
31        title: item.volumeInfo.title,
32        authors: item.volumeInfo.authors || [],
33        description: item.volumeInfo.description,
34        thumbnail: item.volumeInfo.imageLinks?.thumbnail,
35        averageRating: item.volumeInfo.averageRating,
36        publisher: item.volumeInfo.publisher,
37        publishedDate: item.volumeInfo.publishedDate,
38        pageCount: item.volumeInfo.pageCount,
39        language: item.volumeInfo.language,
40        categories: item.volumeInfo.categories,
41        previewLink: item.volumeInfo.previewLink
42      }));
43    };
44
45    // Execute throttled search request
46    const results = await throttleRequest(searchFn);
47
48    // Cache the results
49    cache.set(cacheKey, {
50      data: results,
51      timestamp: Date.now()
52    });
53
54    return { results, error: null };
55  } catch (error) {
56    console.error('Erreur lors de la recherche de livres:', error);
57    let errorMessage = 'Une erreur est survenue lors de la recherche.';
58
59    // Handle rate limiting errors
60    if (error.response?.status === 429) {
61      errorMessage = 'Limite de requêtes atteinte. Veuillez patienter quelques secondes et réessayer.';
62      // Wait longer before next request
63      lastRequestTime = Date.now() + 5000;
64    }
65
66    return { results: [], error: errorMessage };
67  }
68};
```

Maintenant que le service est prêt la prochaine étape est d'établir une barre de recherche et une page de résultat de recherche. Pour rendre le tout dynamique j'ai créer un composant BookSearch.jsx qui est ma barre de recherche, j'en ai fais un composant histoire d'assurer un entretien plus facile du projet et d'également pouvoir la re-utiliser plus tard dans d'autre cas. La première étape est d'importer les fonctions utilisés :



```
1 import React, { useState, useEffect, useRef } from 'react';
2 import { useNavigate } from 'react-router-dom';
3 import { searchBooks, getSearchSuggestions, getBookDetails } from '../services/googleBooksApi';
```

Grâce a l'import, les fonctions sont disponibles a l'utilisation sur ce composant, maintenant il nous faut établir une barre de recherche qui vas permettre deux choses :

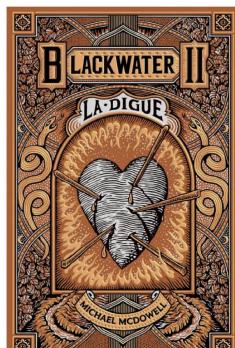
- Si l'utilisateur clique sur un livre proposé avec l'autocomplétion, l'emmener directement sur la page BookDetailsPage.jsx, qui vas afficher donc le livre sélectionné avec toutes les informations
- Si l'utilisateur valide la recherche sans choisir de livre le rediriger vers la page SearchResultsPage.jsx qui vas afficher les résultats correspondant avec la recherche de l'utilisateur.

Voila comment est gérer la récupération des informations plus en détails ainsi qu'un exemple du rendus de la page BookDetailsPage.jsx

```

1 const handleBookSelect = async (book) => {
2   if (!book) return;
3
4   // Quand un livre est sélectionné depuis l'autocomplétion, récupérons ses détails complets
5   try {
6     setLoading(true);
7     const bookDetails = await getBookDetails(book.id);
8     const completeBook = {
9       id: bookDetails.id,
10      title: bookDetails.volumeInfo.title,
11      authors: bookDetails.volumeInfo.authors || [],
12      description: bookDetails.volumeInfo.description,
13      thumbnail: bookDetails.volumeInfo.imageLinks?.thumbnail,
14      averageRating: bookDetails.volumeInfo.averageRating,
15      publisher: bookDetails.volumeInfo.publisher,
16      publishedDate: bookDetails.volumeInfo.publishedDate,
17    };
18    setSelectedBook(completeBook);
19  } catch (err) {
20    console.error("Erreur lors de la récupération des détails:", err);
21    setSelectedBook(book); // Fallback sur les données partielles
22  } finally {
23    setLoading(false);
24  }
25
26 setQuery(book.title);
27  setShowResults(false);
28};
29

```



Blackwater 2 – La Digue

Michael McDOWELL

Tandis que la ville se remet à peine d'une crue dévastatrice, le chantier d'une digue censée la protéger charrie son lot de conséquences : main d'œuvre incontrôlable, courants capricieux, disparitions inquiétantes. Pendant ce temps, dans le clan Caskey, Mary-Love, la matriarche, voit ses machinations se heurter à celles d'Elinor, son étrange belle-fille, mais la lutte ne fait que commencer. Manigances, alliances contre-nature, sacrifices, tout est permis. À Perdido, les mutations seront profondes, et les conséquences, irréversibles. Au-delà des manipulations et des rebondissements, de l'amour et de la haine, Michael McDowell (1950-1999), -co-créateur des mythiques Beetlejuice et L'Étrange Noël de Monsieur Jack, et auteur d'une trentaine de livres, réussit avec Blackwater à bâtir une saga en six romans aussi -addictive qu'une série Netflix, baignée d'une atmosphère unique et fascinante digne de Stephen King. Découvrez le deuxième épisode de Blackwater, une saga matricale avec une touche de surnaturel et un soupçon d'horreur. .

Éditeur: Monsieur Toussaint Louverture

Date de publication: 22 avril 2022

Pages: 244

Langue: Français

[Voir sur Google Books →](#)

Critiques

Connectez-vous pour ajouter ce livre à votre collection

Aucune critique pour le moment. Soyez le premier à donner votre avis !

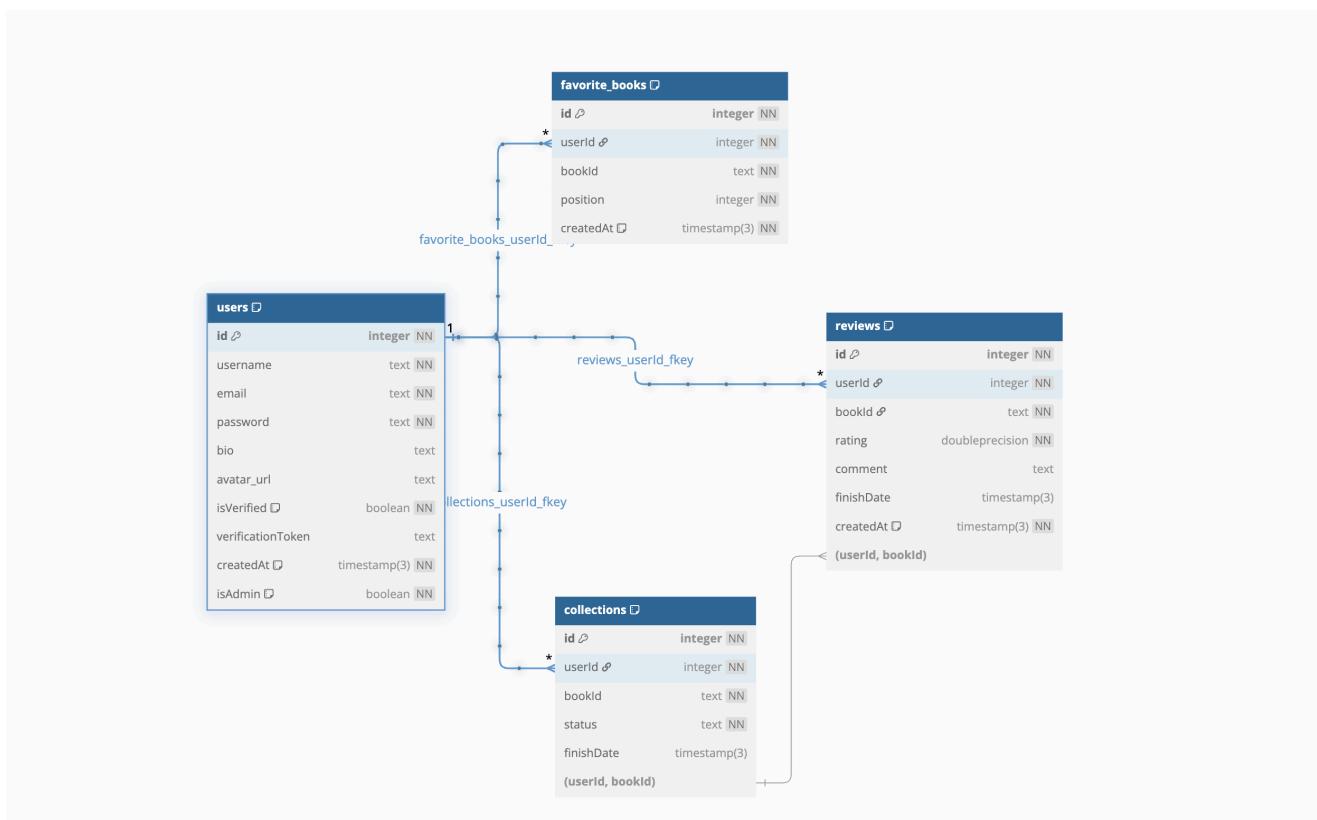


Réalisation Backend

Mise en place de la base de donnée relationnelle :

Pour mon projet, j'ai conçu et mis en place une base de donnée relationnelle à l'aide de PostgreSQL, tout en utilisant l'ORM Prisma pour faciliter la gestion des modèles et des relations. J'ai d'abord réfléchis à la conception de cette base de donnée composée de quatre tables, le but étant de faire une base assez simple tout en subvenant au besoin du projet.

Voici le schéma MLD représentant ma base de donnée :



Concernant la modélisation des données j'ai identifié les entités principales dont j'allais avoir besoin :

- User : représente les utilisateurs de l'application
- Review : enregistre les avis d'utilisateurs sur les livres
- Collection : permet à chaque User de garder et suivre les livres qu'il a lu
- FavoriteBook : liste quatre livre favoris de l'utilisateur avec un classement

Une fois les entités définies, j'ai identifié les relations :

- Un utilisateur peut écrire plusieurs avis (1-N).
- Un utilisateur peut ajouter plusieurs livres à sa collection (1-N).
- Un utilisateur peut avoir jusqu'à 4 livres favoris, chacun ayant une position unique (1-N, avec contrainte sur le nombre et l'unicité).

Chaque Review est obligatoirement lié à un livre présent dans la Collection de l'utilisateur, ce qui m'a amené à utiliser une relation composite (userId, bookId).

Pour la conception et l'implémentation j'ai utiliser Prisma comme ORM, c'est très pratique pour générer automatiquement les migrations SQL ou avoir une synchronisation facile entre les modèles de l'application et la base réelle. Grâce à Prisma on génère les entités de manière déclarative via un fichier schema.prisma puis on génère le tout automatiquement avec la base PostgreSQL.

Chaque entité est définie sous forme de model contenant tout les champs principaux ainsi que les relations avec les autres attributs, voila un exemple :

```

1  * * User Model
2  * Represents application users with authentication and profile data
3  */
4  model User {
5      id          Int      @id @default(autoincrement())
6      username   String
7      email      String    @unique
8      password   String
9      bio        String?
10     avatar_url String?
11     isVerified Boolean  @default(false)
12     isAdmin    Boolean  @default(false)
13     verificationToken String? @unique
14     createdAt   DateTime @default(now())
15     // ? Relationships
16     reviews     Review[]
17     collections Collection[]
18     favoriteBooks FavoriteBook[]
19
20     @@map("users")
21 }
22
23 /**
24  * * Review Model
25  * Stores user reviews and ratings for books
26 */
27 model Review {
28     id          Int      @id @default(autoincrement())
29     userId      Int
30     bookId     String
31     rating      Float
32     comment     String?
33     finishDate  DateTime?
34     createdAt   DateTime @default(now())
35     // ? Relationships
36     user        User    @relation(fields: [userId], references: [id], onDelete: Cascade)
37     collection  Collection @relation(fields: [userId, bookId], references: [userId, bookId])
38
39     @@unique([userId, bookId])
40     @@map("reviews")
41 }

```

Les relations sont définies directement avec les attributs `@relation` tout en précisant les clés étrangères. Des clés composites ont été créées également dans un but d'éviter à l'utilisateur de pouvoir ajouter deux fois le même livre à la collection, d'écrire deux fois un avis pour un même livre ou de ne pas ajouter plusieurs fois un même livre en favoris.

Prisma fournit un avantage conséquent pour interagir avec la base de données de manière sécurisée comme par exemple pour la récupération des livres favoris d'un utilisateur :



```
1 const favorites = await prisma.favoriteBook.findMany({  
2   where: { userId },  
3   orderBy: { position: 'asc' }  
4});
```

Cela évite les requêtes SQL longues et complexes tout en gardant un contrôle total sur les entités. Toute cette structure permet d'éviter les incohérences et de gérer facilement les relations complexes entre les utilisateurs et les livres de l'API.

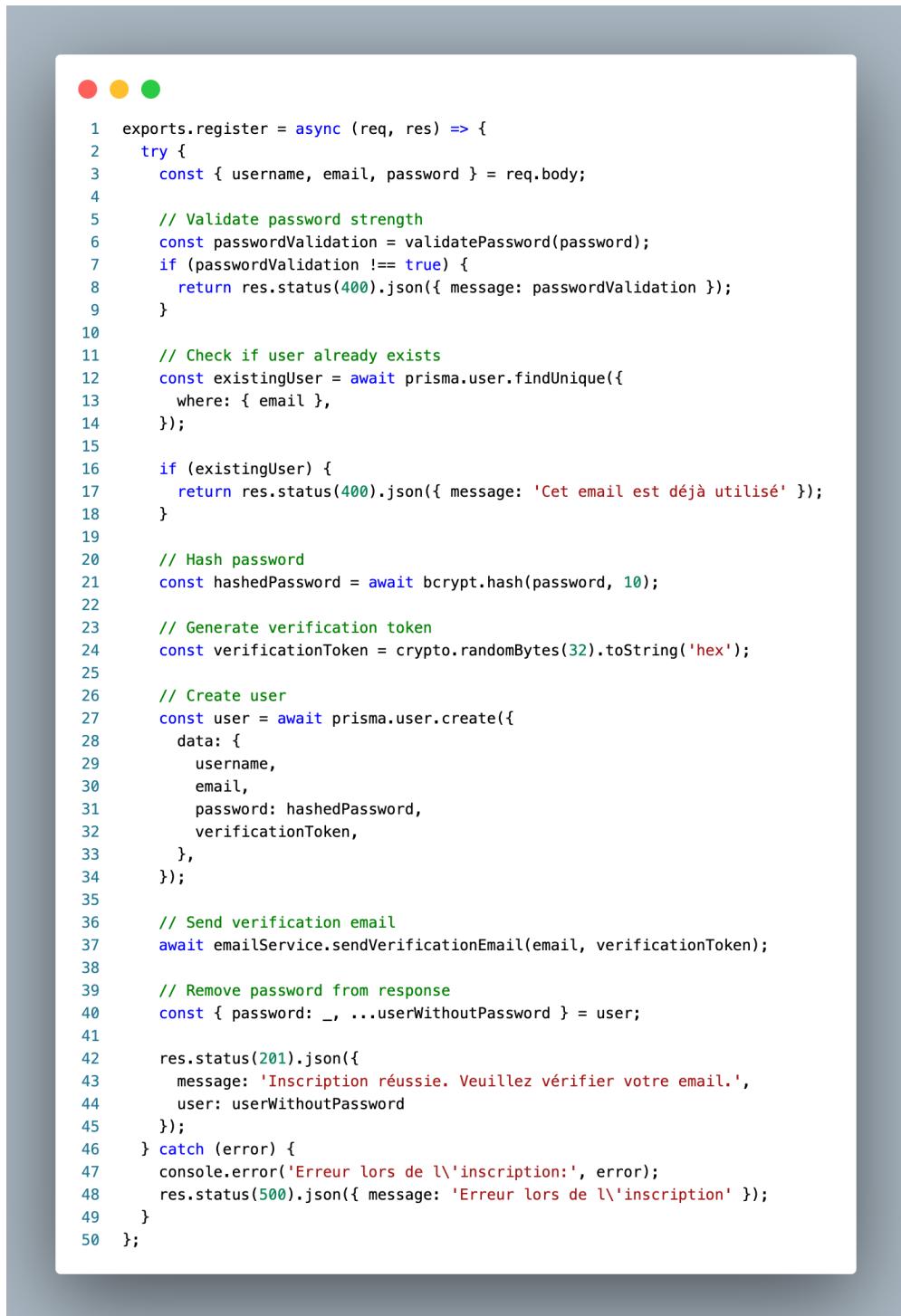
Développer des composants d'accès aux données SQL :

Dans mon projet, mes données sont stockées dans une base PostgreSQL et j'ai utilisé Prisma pour interagir avec. J'ai donc créé un système d'inscription d'inscription sécurisé avec un envoi d'email pour valider le compte sinon l'utilisateur ne peut pas se connecter. Pour le système de connexion je l'ai fait en utilisant un token JWT.

Pour gérer tout le système d'authentification j'ai créé un controller adapté nommé authController.js, ce fichier va permettre de gérer tout le processus de connexion et d'inscription, ce composant interagit directement avec la base de données via Prisma. Lorsqu'un utilisateur souhaite s'inscrire le controller :

- Vérifie la complexité du mot de passe (8 caractères, une majuscule, un chiffre et un symbole)

- Vérifie l'unicité de l'email dans la base de donnée pour ne pas avoir de doublon
- Hash le mot de passe avec bcrypt avant de l'enregistrer dans la base de donnée
- Génère un token de vérification et l'envoie par email à l'aide de nodemailer
- Crée enfin l'utilisateur dans la base de donnée à l'aide de Prisma



```

1 exports.register = async (req, res) => {
2   try {
3     const { username, email, password } = req.body;
4
5     // Validate password strength
6     const passwordValidation = validatePassword(password);
7     if (passwordValidation !== true) {
8       return res.status(400).json({ message: passwordValidation });
9     }
10
11    // Check if user already exists
12    const existingUser = await prisma.user.findUnique({
13      where: { email },
14    });
15
16    if (existingUser) {
17      return res.status(400).json({ message: 'Cet email est déjà utilisé' });
18    }
19
20    // Hash password
21    const hashedPassword = await bcrypt.hash(password, 10);
22
23    // Generate verification token
24    const verificationToken = crypto.randomBytes(32).toString('hex');
25
26    // Create user
27    const user = await prisma.user.create({
28      data: {
29        username,
30        email,
31        password: hashedPassword,
32        verificationToken,
33      },
34    });
35
36    // Send verification email
37    await emailService.sendVerificationEmail(email, verificationToken);
38
39    // Remove password from response
40    const { password: _, ...userWithoutPassword } = user;
41
42    res.status(201).json({
43      message: 'Inscription réussie. Veuillez vérifier votre email.',
44      user: userWithoutPassword
45    });
46  } catch (error) {
47    console.error('Erreur lors de l\'inscription:', error);
48    res.status(500).json({ message: 'Erreur lors de l\'inscription' });
49  }
50};

```

Dans le même fichier j'ai définis les fonctions pour les différentes vérifications sauf pour celle d'envoie d'email que je détaillerais plus tard. Pour la vérification du mot de passe c'est assez simple :

```
1 function validatePassword(password) {
2     // Password validation (min 8 chars, 1 uppercase, 1 number, 1 symbol)
3     const minLength = 8;
4     const hasUpperCase = /[A-Z]/.test(password);
5     const hasNumber = /[0-9]/.test(password);
6     const hasSymbol = /[!@#$%^&*()_+=\[\]\{\};:'"\\"|,.<>\?]/.test(password);
7
8     if (password.length < minLength) {
9         return 'Le mot de passe doit contenir au moins 8 caractères';
10    }
11    if (!hasUpperCase) {
12        return 'Le mot de passe doit contenir au moins une majuscule';
13    }
14    if (!hasNumber) {
15        return 'Le mot de passe doit contenir au moins un chiffre';
16    }
17    if (!hasSymbol) {
18        return 'Le mot de passe doit contenir au moins un symbole';
19    }
20
21    return true;
22 }
```

Grâce à une simple vérification de chaque cas, cela assure que le mot de passe est sécurisé pour l'utilisateur.

Maintenant pour gérer la connexion au site web notre authController.js vas nous permettre d'autoriser la connexion seulement après avoir passer plusieurs vérifications qui sont :

- Une recherche de l'utilisateur dans la base de donnée en utilisant prisma
- Une gestion en cas d'erreur retournant un message pour les principales erreurs rencontrés (401)
- Une vérification du mot de passe haché

- Une vérification de l'email, voir si il est bien confirmé via notre service SMTP (isVerified)
- Une génération d'un token JWT valable pendant 24h signé de la clé de l'application
- Un renvoie au client des données de l'utilisateurs sans le mot de passe

Donc pour résumé en utilisant Prisma j'ai pus interagir avec la base de donnée avec la mise en place de composants d'accès au données permettant une gestion efficaces des données des utilisateurs.

Voici plus en détails le code pour la partie login du controller :

```

● ● ●
1 exports.login = async (req, res) => {
2   try {
3     const { email, password } = req.body;
4
5     // Find user with all necessary data
6     const user = await prisma.user.findUnique({
7       where: { email },
8       select: {
9         id: true,
10        email: true,
11        username: true,
12        password: true,
13        isVerified: true,
14        isAdmin: true,
15        bio: true,
16        avatar_url: true,
17      }
18    });
19
20    if (!user) {
21      return res.status(401).json({ message: 'Email ou mot de passe incorrect' });
22    }
23
24    // Check if email is verified
25    if (!user.isVerified) {
26      return res.status(401).json({ message: 'Veuillez vérifier votre email avant de vous connecter' });
27    }
28
29    // Verify password
30    const validPassword = await bcrypt.compare(password, user.password);
31    if (!validPassword) {
32      return res.status(401).json({ message: 'Email ou mot de passe incorrect' });
33    }
34
35    /**
36     * ? JWT Generation
37     * Create authentication token with user ID
38     */
39    const token = jwt.sign(
40      {
41        userId: user.id,
42        isAdmin: user.isAdmin
43      },
44      process.env.JWT_SECRET,
45      { expiresIn: '24h' }
46    );
47
48    // Remove password from response
49    const { password: _, ...userWithoutPassword } = user;
50
51    // Return user and token
52    res.json({
53      user: userWithoutPassword,
54      token
55    });
56  } catch (error) {
57    console.error('Erreur lors de la connexion:', error);
58    res.status(500).json({ message: 'Erreur lors de la connexion' });
59  }
60};

```

Ouverture sur le NoSQL :

Bien que mon projet MyBook repose entièrement sur une base de données relationnelle structurée (PostgreSQL), je connais également les alternatives NoSQL, très utilisées dans le développement web moderne.

Contrairement aux bases relationnelles qui reposent sur un schéma fixe (tables, relations, types), les bases NoSQL adoptent une approche beaucoup plus flexible, ce qui permet de stocker des données sous forme de documents, de paires clé/valeur, de graphes ou de colonnes, selon le système choisi.

Par exemple, avec une base MongoDB, les données sont organisées en documents JSON stockés dans des collections, sans obligation de structure commune. Cela permet d'adapter dynamiquement le format des données selon les besoins métier, sans passer par des migrations complexes. C'est idéal pour des projets en évolution rapide, avec des modèles de données changeants.

Les solutions comme Firebase (Realtime Database ou Firestore) proposent également une architecture NoSQL, particulièrement adaptée aux applications mobiles ou temps réel, grâce à une synchronisation directe entre le client et le serveur.

Dans le cadre d'une évolution future de MyBook, l'intégration d'une solution NoSQL pourrait être pertinente pour certains modules spécifiques comme :

- Le stockage de logs applicatifs,
- La gestion de messages ou notifications en temps réel,
- L'implémentation de fonctionnalités sociales, comme les réactions ou commentaires rapides, qui ne nécessitent pas toujours de rigidité relationnelle.

Cette ouverture technologique me permet d'adapter mes choix à la nature des données à traiter. Je suis donc capable d'utiliser des bases SQL ou NoSQL en fonction du contexte et des objectifs du projet.

Même si MyBook n'intègre pas directement de composant NoSQL, j'ai manipulé ce type de base (MongoDB, Firebase) dans d'autres projets ou exercices de formation, ce qui me permet de comprendre les enjeux et avantages de ces technologies dans un environnement professionnel.

Développer des composants métier côté serveur :

Pour les composants métiers de mon application MyBook j'ai pu créer des méthodes spécifiques pour chacune des fonctionnalités avec une mise en place de Service, middlewares et controllers.

Pour gérer l'envoie d'email j'ai utilisé nodemailer avec une configuration SMTP Gmail et j'ai créé un service dédié directement à cette fonctionnalité, il se nomme emailService.js.

Pour pouvoir rendre le tout fonctionnel il m'a fallu d'abord configurer le « transporter », c'est à dire qui va envoyer l'email, dans mon cas j'ai choisi mon adresse mail lié à la formation pour mes premiers test. Dedans on configurer notre service utiliser et on donne l'email ainsi que le mot de passe d'application que Google fournit pour utiliser ce genre de service, bien évidemment les variables sont dans le fichier .env pour les sécuriser.

Voilà à quoi ressemble le transporter :



```
1 const transporter = nodemailer.createTransport({
2   service: 'gmail',
3   auth: {
4     user: process.env.EMAIL_USER,
5     pass: process.env.EMAIL_PASSWORD
6   },
7   tls: {
8     rejectUnauthorized: false
9   }
10});
```

La prochaine étape était de configurer l'envoi de l'email :



```
1 exports.sendVerificationEmail = async (email, token) => {
2   const verificationUrl = `${process.env.FRONTEND_URL}/verify-email/${token}`;
3
4   const mailOptions = {
5     from: process.env.EMAIL_USER,
6     to: email,
7     subject: 'Vérification de votre compte MyBook',
8     html: `
9       <h1>Bienvenue sur MyBook !</h1>
10      <p>Merci de vous être inscrit. Veuillez cliquer sur le lien ci-dessous pour vérifier votre compte :</p>
11      <a href="${verificationUrl}">Vérifier mon compte</a>
12      <p>Si vous n'avez pas créé de compte, vous pouvez ignorer cet email.</p>
13    `;
14};
```

On crée notre verificationUrl qui est à l'adresse de notre frontend avec un token unique généré lors de l'inscription de l'utilisateur (rappel à notre authController.js). On configure également l'envoie de l'email avec l'objet du mail ainsi que le contenu et une fois que l'utilisateur clique sur le lien, la colonne isVerified de la base de données passe en TRUE.

Voici le mail :



yanis.bennadji@laplateforme.io
À moi ▾

mar. 6 mai 15:01 ⭐ 🌟 ↵ ⏺

Bienvenue sur MyBook !

Merci de vous être inscrit. Veuillez cliquer sur le lien ci-dessous pour vérifier votre compte :

[Vérifier mon compte](#)

Si vous n'avez pas créé de compte, vous pouvez ignorer cet email.

Répondre Transférer

Différents services et controller sont mis en place sur le site pour la gestion des fonctionnalités comme :

- Gestion des collections et des reviews (avec date, note et commentaire)
- Vérification du rôle de l'utilisateur (si admin ou user)
- Gestion des favoris
- Validation des données à chaque requête

Tout cela est réalisé avec un respect des bonnes pratiques tels que :

- Sécurisation des données via hachage de mot de passe, vérification des requêtes, vérification des droits et de l'utilisateur avant modification de certaines données
- L'utilisation d'une programmation orientée objet moderne
- L'utilisation de test unitaire ainsi que plusieurs tests manuels de chaque fonctionnalité

Tout le projet est fait avec une bonne pratique et voici un exemple des différentes routes pour la partie des livres favoris de l'utilisateur :

Méthode	Endpoint	Description	Authentification
GET	/api/favorite-books/	Récupérer les livres favoris de l'utilisateur	Oui
GET	/api/favorite-books/users/:userId	Favoris d'un utilisateur spécifique	Oui
POST	/api/favorite-books/	Ajouter un livre au favoris	Oui

Méthode	Endpoint	Description	Authentification
PUT	/api/favorite-books/:bookId/position	Modifier la position d'un livre favoris	Oui
DELETE	/api/favorite-books/:bookId	Supprimer un livre des favoris	Oui

Sécurité et mise en place :

La sécurité a été un aspect essentiel dès le début du développement de mon application MyBook.

J'ai pris en compte les principales failles de sécurité web (identifiées notamment via OWASP) et mis en place des mesures concrètes pour protéger les données utilisateurs, sécuriser les échanges avec le serveur, et limiter les abus.

Voici les protections mises en œuvre :

- Authentification sécurisée avec JWT + middleware Express

Chaque utilisateur reçoit un token JWT signé à la connexion. Ce token est vérifié à chaque requête protégée grâce à un middleware personnalisé. Cela évite toute action non autorisée sur les routes sensibles (modification de profil, ajout de livres, etc.).

- Hachage des mots de passe

Les mots de passe sont hachés avec bcrypt avant d'être stockés dans la base.

En cas de fuite de la base de données, aucune donnée sensible ne peut être exploitée en clair.

- Protection contre l'injection SQL

J'utilise Prisma ORM, qui permet d'écrire des requêtes sécurisées et paramétrées, rendant impossible l'injection SQL manuelle dans les champs de formulaire.

- Gestion des variables sensibles avec dotenv

Toutes les données critiques (clé d'API, URL de base, secrets JWT) sont stockées dans des fichiers .env et ne sont jamais versionnées grâce au .gitignore.

Ces variables sont ensuite chargées via le package dotenv de manière centralisée et sécurisée.

- Sécurisation des en-têtes HTTP avec Helmet.js

J'ai intégré le middleware Helmet côté serveur pour ajouter automatiquement des en-têtes de sécurité : protection contre le clickjacking, détection de contenu mixte, désactivation de certaines fonctionnalités à risque.

- Limitation des requêtes (rate limiting)

Pour empêcher les attaques par force brute ou spam, j'ai mis en place un rate limiter sur les routes critiques (comme la connexion). Cela limite le nombre de tentatives possibles par adresse IP dans un laps de temps donné.

- Protection contre les attaques XSS (Cross-Site Scripting)

J'utilise un filtre XSS (paquet xss-clean) pour nettoyer les entrées utilisateur, notamment dans les avis/commentaires.

Cela empêche qu'un script malveillant soit injecté et exécuté dans l'interface d'un autre utilisateur.

En complément, l'application respecte aussi des pratiques de base comme :

- Le CORS configuré strictement,
- La validation côté serveur de tous les formulaires,
- Le logging des erreurs sans exposition d'informations sensibles.

Ces différentes mesures rendent l'application plus robuste face aux attaques les plus courantes. Je suis également conscient qu'en environnement professionnel, il serait nécessaire d'aller encore plus loin : audit sécurité, tests de pénétration, chiffrement de données sensibles, authentification à double facteur, etc.

Tests unitaires

Test unitaire :

Pour tester les fonctionnalités de mon application j'ai mis en place quelque tests unitaires pour faciliter le développement. J'ai donc fais deux fichiers de test :

- auth.test.js qui test tout le système de connexion ainsi que l'inscription ce qui comprends les mots de passe invalide les erreurs de base de données, l'envoie d'email ou les champs manquants
- collection.test.js qui test l'ajout de livres à la collection, la récupération des livres lus et la validation et gestion des erreurs

Voici un exemple d'un des test présent dans collection.test.js :

```
● ● ●
1 * Test that database errors are handled properly
2 */
3 test('handles database errors when retrieving books', async () => {
4     // Mock request and response
5     const req = {
6         user: { id: 1 }
7     };
8
9     const res = {
10         json: jest.fn(),
11         status: jest.fn().mockReturnThis()
12     };
13
14     // Mock a database error
15     prisma.collection.findMany.mockRejectedValue(new Error('Database connection failed'));
16
17     // Call the function
18     await collectionController.getReadBooks(req, res);
19
20     // Assertions
21     expect(res.status).toHaveBeenCalledWith(500);
22     expect(res.json).toHaveBeenCalledWith({
23         message: 'Erreur lors de la récupération des livres'
24     });

```

Et voilà le résultat quand on lance le test :

```
PASS  tests/collection.test.js
Add Book to Collection
  ✓ successfully adds a new book to collection (2 ms)
  ✓ returns error when book already exists in collection
  ✓ handles database errors properly (13 ms)
  ✓ handles missing required fields (1 ms)
Get Read Books
  ✓ successfully retrieves user read books (1 ms)
  ✓ returns empty array when user has no read books
  ✓ handles database errors when retrieving books (1 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        0.105 s, estimated 1 s
```

Des erreurs sont présentes dans la console du test mais c'est normal, elles sont là pour s'assurer que tout les cas d'erreurs possibles sont pris en compte par mon code.

Déploiement

Dans la dernière phase de mon projet, j'ai entamé le déploiement de l'application sur Render, une plateforme cloud moderne qui permet d'héberger à la fois le frontend, le backend, et la base de données PostgreSQL.

J'ai d'abord déployé le backend Express en tant que Web Service, avec configuration des scripts (npm install, node app.js) et des variables d'environnement (DATABASE_URL, JWT_SECRET, etc.).

J'ai également commencé à déployer le frontend (React + Vite) en tant que Static Site, avec la commande de build (npm run build) et la liaison au backend via VITE_API_URL.

La base de données PostgreSQL a été créée directement sur Render, avec gestion des accès sécurisés et intégration à Prisma.

Même si le déploiement complet n'est pas encore finalisé, j'ai compris les étapes clés pour mettre une application fullstack en ligne, et j'ai appris à gérer :

- La séparation des environnements (développement / production),
- La configuration des variables sensibles avec .env,
- Le système de déploiement continu via GitHub,
- La sécurisation des échanges frontend/backend (CORS, tokens).

Je poursuis encore l'apprentissage de certains aspects du déploiement en production (gestion des erreurs, surveillance, performance, sécurité), mais cette expérience m'a déjà permis de mieux comprendre les enjeux réels d'un projet web en ligne.

Veille technologique

Tout au long de ma formation et particulièrement lors du développement de mon projet MyBook, j'ai mis en place une veille technologique régulière. Elle m'a permis de suivre les bonnes pratiques, de mieux comprendre les outils que j'utilisais et de rester à jour sur les évolutions du web.

Cette veille s'est faite de façon quotidienne, en consultant :

- La documentation officielle des outils utilisés (React, Prisma, Express, Tailwind, etc.),
 - Des blogs techniques (dev.to, Medium, Hashnode),
 - Des forums (Stack Overflow, GitHub issues),
 - Et en suivant l'actualité sur des chaînes YouTube et newsletters spécialisées (Daily Dev, Frontend Focus...).

Elle m'a permis de :

- Approfondir mes connaissances sur React, notamment sur l'usage avancé des hooks (useEffect, useContext, useReducer) et la structuration en composants réutilisables.
- Comprendre comment Vite optimise le chargement des ressources par rapport à Webpack, et pourquoi il est mieux adapté pour un projet moderne.
- Maîtriser les outils de style comme Tailwind CSS, en m'appuyant sur les retours de la communauté pour structurer mes classes de manière claire et lisible.
- Suivre les meilleures pratiques de sécurisation d'API en Express : usage de JWT, gestion des headers avec Helmet, et configuration de cors.

- Approfondir l'écosystème Prisma : migrations, seed, types, génération automatique de client, bonnes pratiques pour éviter les erreurs de typage ou de logique SQL.
- Me sensibiliser aux problématiques de performances et de SEO dans une application React avec des routes dynamiques et un frontend basé sur une SPA.

En plus du SQL, j'ai également pris le temps d'explorer les bases NoSQL comme MongoDB ou Firebase, afin de mieux comprendre dans quels contextes ces technologies sont plus adaptées.

Enfin, cette veille m'a aussi poussé à explorer des sujets connexes à mon projet :

- Le déploiement moderne via Render (CI/CD simplifié, gestion des environnements),
- L'importance des .env bien organisés,
- Et la gestion des erreurs utilisateurs pour améliorer l'expérience (UX).

Cette habitude de veille m'a permis non seulement de gagner en autonomie, mais aussi de faire des choix technologiques éclairés tout au long du développement. Elle reste pour moi une pratique essentielle pour progresser et rester compétent dans un domaine en constante évolution.

Amélioration possibles

Parmi les pistes d'amélioration, je prévois notamment :

- Ajout d'un système de messagerie privée entre utilisateurs.

Cela permettrait aux membres de la plateforme d'échanger autour de leurs lectures, de se recommander des livres, ou de créer des discussions autour d'un thème ou d'un auteur.

- Possibilité de répondre aux avis (reviews) postés sur les livres.

Cette fonctionnalité apporterait une dimension communautaire plus forte, où les utilisateurs peuvent interagir, réagir ou débattre autour d'un même livre. Elle serait modérée et limitée à un système de commentaires avec notifications.

- Ajout de filtres et de tris avancés dans la collection :

par auteur, date de lecture, note, statut (lu/en cours/à lire), ou genre, pour offrir une meilleure navigation et une personnalisation plus poussée.

- Création de listes ou de challenges de lecture, que les utilisateurs pourraient suivre, partager ou créer eux-mêmes (ex : "Top 10 livres de fantasy", "Mes lectures 2025", etc.).

- Notifications (mails ou intégrées) pour informer l'utilisateur des réponses à ses avis, des nouveaux livres dans ses genres favoris ou des activités de ses amis.

- Amélioration de la partie admin avec plus de statistiques, de visibilité sur les utilisateurs actifs, et des outils de modération.

Ces évolutions nécessitent d'enrichir à la fois la base de données, les contrôleurs backend, l'interface utilisateur, et d'assurer une bonne sécurisation des nouvelles fonctionnalités.

Elles visent à transformer MyBook en une plateforme communautaire complète et vivante autour de la lecture.

Conclusion

La réalisation de mon projet MyBook a été une expérience formatrice et complète, qui m'a permis de mettre en pratique l'ensemble des compétences acquises durant ma formation DWWM. En partant d'une idée simple – créer une plateforme web de gestion de lectures inspirée de Letterboxd – j'ai pu structurer un vrai projet de bout en bout : conception, maquettage, développement frontend et backend, intégration d'une API externe, gestion des données, sécurisation, et déploiement en ligne.

Ce projet m'a permis de consolider mes bases en React, Express, Prisma et PostgreSQL, mais aussi de prendre conscience de l'importance des bonnes pratiques de développement : organisation du code, sécurité, gestion des états, structuration en composants, documentation, et réflexion sur l'expérience utilisateur.

Au-delà de l'aspect technique, ce projet m'a appris à travailler de manière autonome, à résoudre mes blocages par la veille et la recherche, et à toujours m'adapter. J'ai aussi mieux compris l'importance de l'architecture dans un projet fullstack, ainsi que la nécessité d'avoir une approche progressive et rigoureuse.

Je ressors de cette réalisation avec la satisfaction d'avoir mené un projet personnel complet, concret, fonctionnel, et aligné avec les attentes d'un développeur web moderne. Cela me donne confiance pour la suite de mon parcours professionnel, avec l'envie de continuer à apprendre, à évoluer, et à travailler sur des projets toujours plus ambitieux.