Rapport Tâche 4 : Développement Web Full-Stack avec Node.js, Express, React et MySQL

Firas Ben Hmida
Encadré par NextGen Coding

22 juillet 2025

Table des matières

Chapitre 1

Introduction

Dans cette tâche, j'ai conçu une application web complète en utilisant les technologies modernes du développement full-stack. Le backend est développé en Node.js avec le framework Express.js, connecté à une base de données MySQL. Le frontend est développé en React avec TailwindCSS, et l'authentification est assurée par JWT.

Chapitre 2

Node.js et Express.js

2.1 Node.js

Node.js est un environnement d'exécution JavaScript côté serveur. Il permet de créer des serveurs web performants, basés sur un modèle non-bloquant (asynchrone), idéal pour les applications temps réel et les APIs.

2.2 Express.js

Express.js est un framework léger pour Node.js qui simplifie la création de serveurs et d'APIs REST. Il permet de définir facilement des routes HTTP, de gérer les requêtes et réponses, et d'organiser le code de manière modulaire.

2.3 Les Routes et les APIs

Les routes représentent les chemins d'accès à des ressources. Avec Express, on peut créer des APIs qui réagissent à différentes méthodes HTTP :

- **GET** : récupérer des données.
- **POST** : ajouter des données.
- **PUT** : mettre à jour des données existantes.
- **DELETE** : supprimer des données.

Les données sont généralement échangées en format JSON (JavaScript Object Notation), un format léger, lisible et largement utilisé dans les communications entre frontend et backend.

2.4 Captures d'écran - Routes avec Express

FIGURE 2.1 – Exemple de routes API REST créées avec Express.js

Lien GitHub : github.com/firasbenhmida/Routes-avec-ExpressJS

Chapitre 3

Test des Routes avec Postman

Pour tester mon API, j'ai utilisé Postman, un outil indispensable pour simuler des requêtes HTTP.

3.1 Fonctionnement

Postman permet de :
— Envoyer des requêtes GET, POST, PUT, DELETE.
— Visualiser les réponses du serveur.
— Ajouter des headers, body, tokens, etc.

3.2 Captures d'écran - Tests API

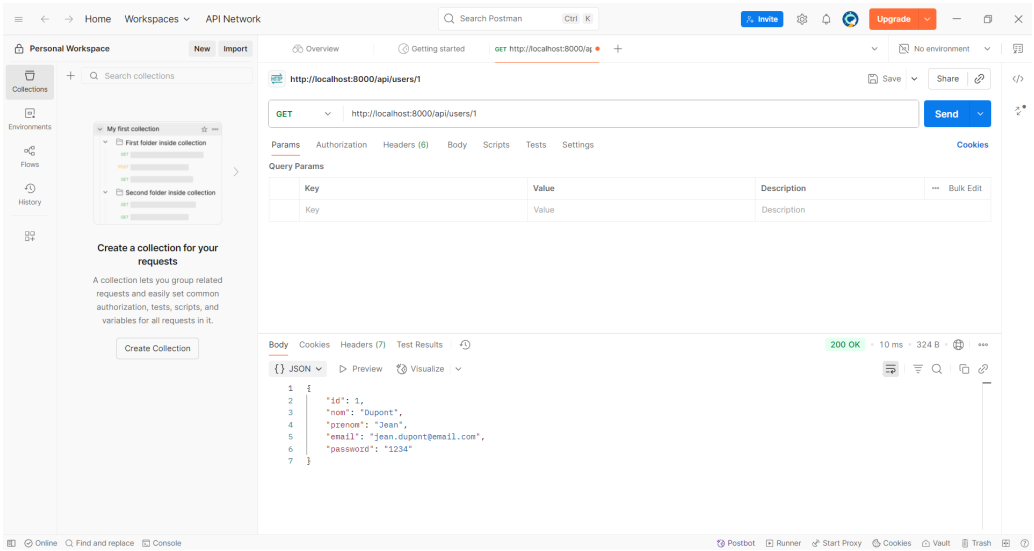

FIGURE 3.1 – Test d'une requête GET

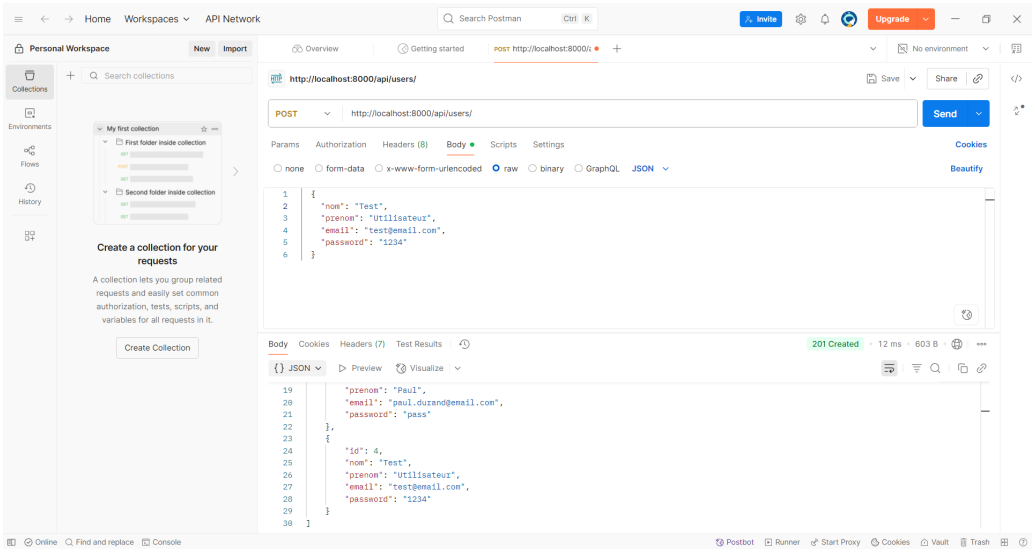

FIGURE 3.2 – Test d'une requête POST

Chapitre 4

Connexion avec MySQL

4.1 Pourquoi MySQL ?

MySQL est une base de données relationnelle robuste, idéale pour les applications web. Elle permet de stocker les utilisateurs, produits, commandes, etc., de façon structurée.

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
1	id	int			Non	*Aucun(e)*		AUTO_INCREMENT	Modifier Supprimer Plus
2	nom	varchar(50)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
3	prenom	varchar(50)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
4	email	varchar(100)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
5	password	varchar(255)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
6	role	varchar(50)	utf8mb4_0900_ai_ci		Non	*Aucun(e)*			Modifier Supprimer Plus

FIGURE 4.1 – Schéma de la table des utilisateurs

4.2 Connexion

J'ai configuré une connexion entre le backend Express et MySQL. Les identifiants sont sécurisés via un fichier `.env`. Les requêtes SQL sont ensuite exécutées pour manipuler les données (insertion, lecture, mise à jour, suppression).

backend > .env

```
1  DB_HOST=localhost
2  DB_USER=root
3  DB_PASSWORD=
4  DB_NAME=app
5  PORT=3000
6  ACCESS_TOKEN_SECRET=CLE_JSON_WEB_TOKEN
```

FIGURE 4.2 – Connexion à la base de données

Chapitre 5

Développement d'une Application Full-Stack

5.1 Architecture

L'application est divisée en deux parties :
— **Backend** : Node.js + Express + MySQL
— **Frontend** : React.js + TailwindCSS

5.2 Captures d'écran

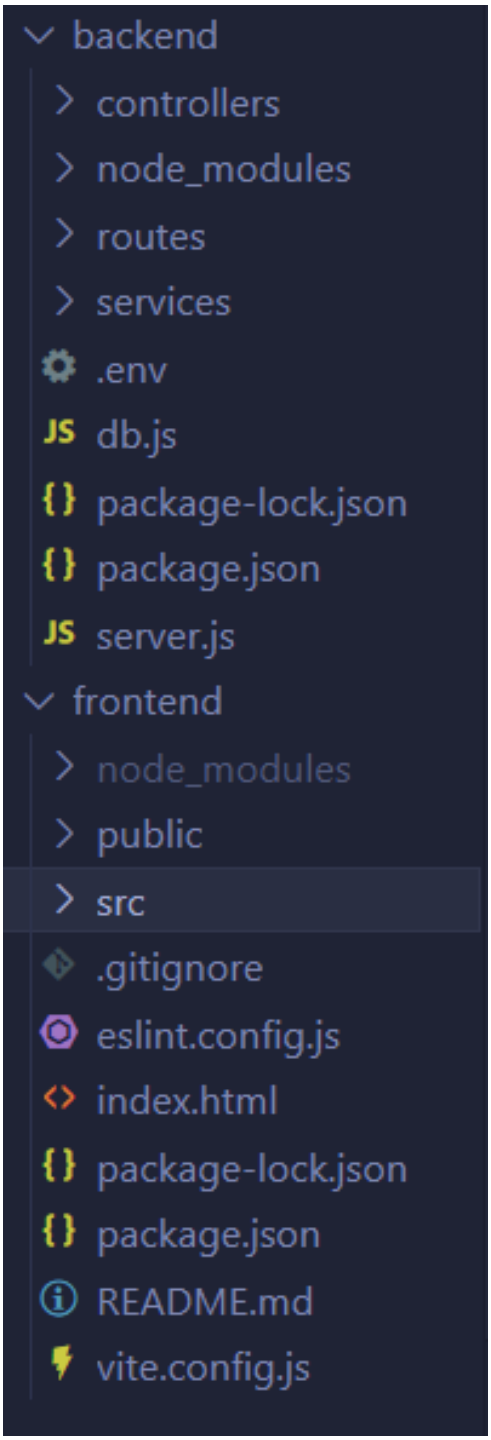

FIGURE 5.1 – Architecture de l'application full-stack (Backend + Frontend + Base de données)

```
export async function registerUser(userData) {
  const { nom, prenom, email, password } = userData;
  const hashedPassword = await bcrypt.hash(password, 10);
  try {
    const result = await db.query(
      "INSERT INTO users (nom, prenom, email, password, role) VALUES (?, ?, ?, ?, 'Utilisateur')",
      [nom, prenom, email, hashedPassword]
    );
    return result;
  } catch (error) {
    console.log(error);
    return null;
  }
}
```

FIGURE 5.2 – Fonction de création d'utilisateur avec hachage du mot de passe et insertion dans MySQL

```
export default function Register() {
  const handleRegister = async (e) => {
    try {
      const response = await fetch("http://localhost:3000/api/register", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          nom: form.nom,
          prenom: form.prenom,
          email: form.email,
          password: form.password
        })
      });
      const data = await response.json();
      if (response.ok) {
        localStorage.setItem("accessToken", data.accessToken);
        localStorage.setItem("user", JSON.stringify(data.user));
        navigate("/profile");
      } else {
        if (data.message && data.message.includes("existe déjà")) {
          setRegisterError("Cet email existe déjà.");
        } else {
          setRegisterError(data.message || "Erreur lors de l'inscription.");
```

FIGURE 5.3 – Exemple de fonction d'inscription d'un utilisateur côté client (React)

```
export const register = async (req, res) => {
  try {
    const userData = {
      nom: req.body.nom,
      prenom: req.body.prenom,
      email: req.body.email,
      password: req.body.password
    };
    const existingUser = await userService.findUserByEmail(userData.email);
    if (existingUser) {
      return res.status(400).json({ message: "Cet email existe déjà." });
    }
    await userService.registerUser(userData);
    const user = await userService.loginUser(userData.email, userData.password);
    const userPayload = { id: user.id, name: user.nom, email: user.email, role: user.role };
    const accessToken = generateAccessToken(userPayload);
    res.status(201).json({ message: "Inscription réussie !", user, accessToken });
  } catch (err) {
    console.error("Erreur lors de l'inscription:", err);
    res.status(500).json({ message: "Erreur serveur lors de l'inscription." });
  }
```

FIGURE 5.4 – Exemple de route d'inscription utilisateur côté serveur (Express)

5.3 Fonctionnalités développées

— Authentification sécurisée par JWT.
— Gestion des utilisateurs (CRUD).
— Séparation des rôles (routes protégées).

5.4 Captures d'écran

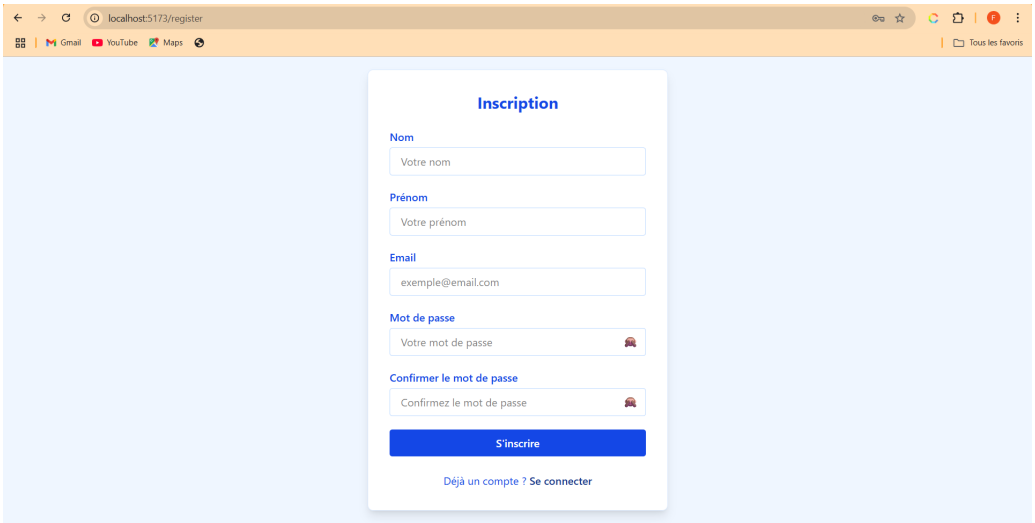

FIGURE 5.5 – Interface Register

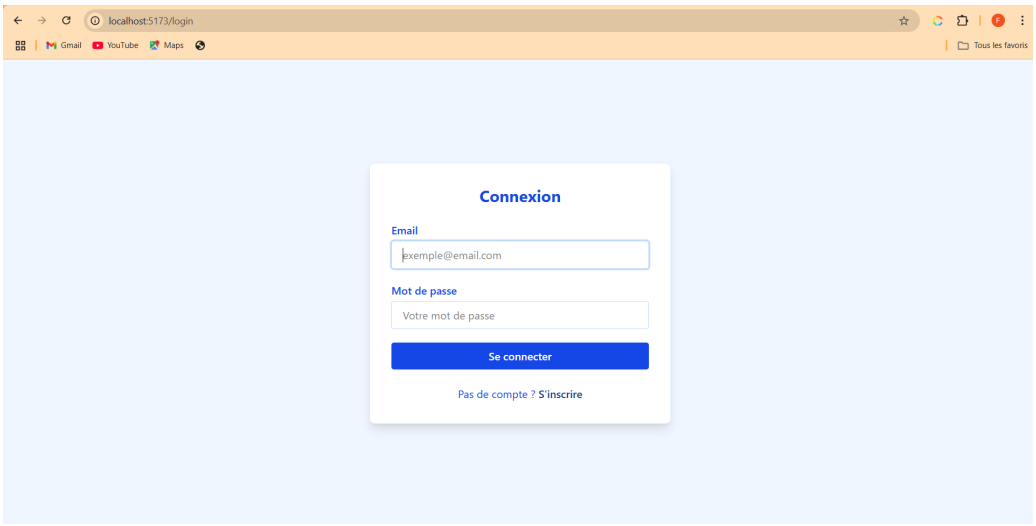

FIGURE 5.6 – Interface Login

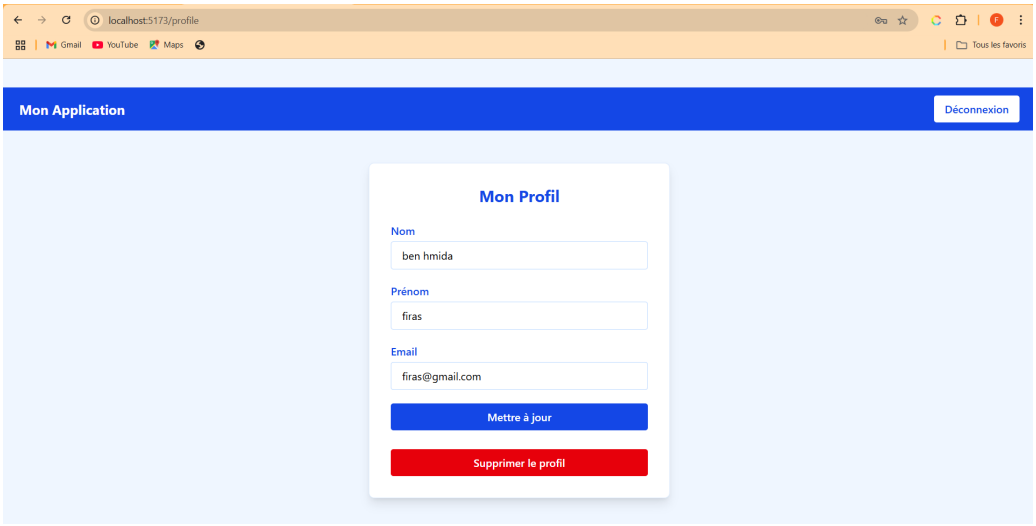

FIGURE 5.7 – Interface Profile

Lien GitHub complet (frontend + backend) : github.com/firasbenhmida/fullstack-app

Chapitre 6

Authentification avec JWT

6.1 Pourquoi JWT ?

JWT (JSON Web Token) permet d'assurer une authentification sécurisée. Après connexion, un token est généré et stocké côté client. Ce token est ensuite envoyé dans les headers pour authentifier chaque requête.

6.2 Avantages

— Sécurité renforcée.
— Gestion des accès privés/publics.
— Scalabilité (sans session serveur).

Chapitre 7

Ressources utilisées

- Node.js Crash Course
- Express Crash Course
- MySQL Node.js Express
- Build a Full Stack CRUD App using React Tailwind Node PostgreSQL|Best practice Industry standard
- JWT Authentication Tutorial - Node.js
- Documentation :
  - Github
  - Google
  - Youtube

Chapitre 8

Conclusion

Ce projet m'a permis de maîtriser les bases du développement backend avec Node.js et Express, la gestion d'une base de données MySQL, et l'implémentation d'une authentification sécurisée avec JWT. Grâce à React et TailwindCSS, j'ai pu offrir une interface moderne et responsive. Cette expérience m'a permis de comprendre l'architecture complète d'une application web et de devenir plus autonome dans le développement full-stack.