Université Mohammed V- Rabat
جامعة محمد الخامس- الرباط

Ecole Supérieure de Technologie - Salé
المدرسة العليا للتكنولوجيا - سلا

Département Réseaux Informatique

Filière Licence : ingénierie des applications web et mobiles

Développement Fullstack + CI/CD avec GitHub Actions

Préparation de l'environnement de travail.

Réalisée Par :

Houda Harmane

Année universitaire : 2025/2026

I. Présentation générale du projet :

Ce projet a pour objectif la conception et la mise en œuvre d'une application web fullstack permettant la gestion des utilisateurs à travers des fonctionnalités classiques telles que l'ajout, la modification, la suppression et l'affichage des utilisateurs.

Backend

La partie serveur (backend) est développée à l'aide du framework **Express.js**, qui permet de construire des API REST robustes et performantes en JavaScript. Le backend gère l'ensemble des opérations CRUD (Create, Read, Update, Delete) relatives aux utilisateurs.
Les données des utilisateurs sont stockées dans une **base de données SQLite**, légère et facile à configurer, ce qui en fait un excellent choix pour une application de démonstration ou de petite à moyenne échelle. Une attention particulière est portée à la structure de la base de données, à la validation des entrées, ainsi qu'à la gestion des erreurs.

Frontend

Le frontend est développé avec **React.js**, une bibliothèque JavaScript populaire pour la création d'interfaces utilisateurs dynamiques et réactives.
L'interface permet aux utilisateurs d'interagir facilement avec l'application : formulaire d'ajout, tableau de visualisation des utilisateurs, options de modification et de suppression. L'accent est mis sur une navigation fluide et une bonne ergonomie, avec l'utilisation de composants réutilisables et une gestion efficace des états via le hook `useState` et potentiellement `useEffect`.

Conteneurisation

L'ensemble de l'application (frontend et backend) est **conteneurisé à l'aide de Docker**. Chaque partie de l'application fonctionne dans un conteneur indépendant, ce qui facilite le déploiement, la portabilité et la gestion des dépendances. Un fichier `docker-compose.yml` permet de lancer et d'orchestrer facilement les différents services (API, base de données, frontend).

CI/CD

Enfin, le projet intègre une chaîne d'intégration et de déploiement continu (**CI/CD**) via **GitHub Actions**. Des workflows sont définis pour automatiser les étapes de test, de construction de l'image Docker, et de déploiement. Cela permet d'assurer la qualité du code et la mise en production rapide à chaque mise à jour ou fusion de branches dans le dépôt GitHub

II. Mise en place du Backend :

```
Backend-express-sqlite/
│
├── controllers/
│   └── usersController.js    # Logique CRUD des utilisateurs
│
├── database/
│   ├── db.js             # Configuration SQLite
│   └── users.db          # Fichier de base de données
│
├── middlewares/
│   ├── errorHandler.js     # Gestion des erreurs
│   └── logger.js         # Middleware de logging
│
├── routes/
│   └── users.js          # Routes API
│
├── tests/
│   └── api.test.js       # Tests API
│
├── .env                # Variables d'environnement
├── .env.example        # Template .env
├── app.js              # Configuration Express
├── package.json        # Dépendances
└── server.js           # Point d'entrée
```

1. Création du Serveur (server.js)

- Initialisation d'une application Express.js comme serveur web.
- Configuration du port via les variables d'environnement (ex: `3001`).
- Lancement du serveur avec un message de confirmation.

2. Configuration de l'Environnement (dotenv)

- Utilisation du package **dotenv** pour charger les variables sensibles depuis un fichier `.env`.
- Exemple de variables :
  - `PORT=3001`
  - `DB_PATH=./database/users.db` (pour SQLite).
- Le fichier `.env.example` sert de template pour la configuration.

3. Connexion à SQLite (sqlite3)

- Utilisation du module **sqlite3** pour une base de données légère et sans serveur.
- La connexion s'établit automatiquement au démarrage.
- Création d'une table `users` (si inexistante) avec les champs :

- id (clé primaire auto-incrémentée).
- name (texte).
- email (texte unique).

4. Routes RESTful

- **GET** /api/users → Liste tous les utilisateurs.
- **POST** /api/users → Ajoute un utilisateur (requiert name et email).
- **PUT** /api/users/:id → Met à jour un utilisateur par son ID.
- **DELETE** /api/users/:id → Supprime un utilisateur par son ID.
- Les routes sont définies dans un fichier séparé (routes/users.js) pour une meilleure organisation.

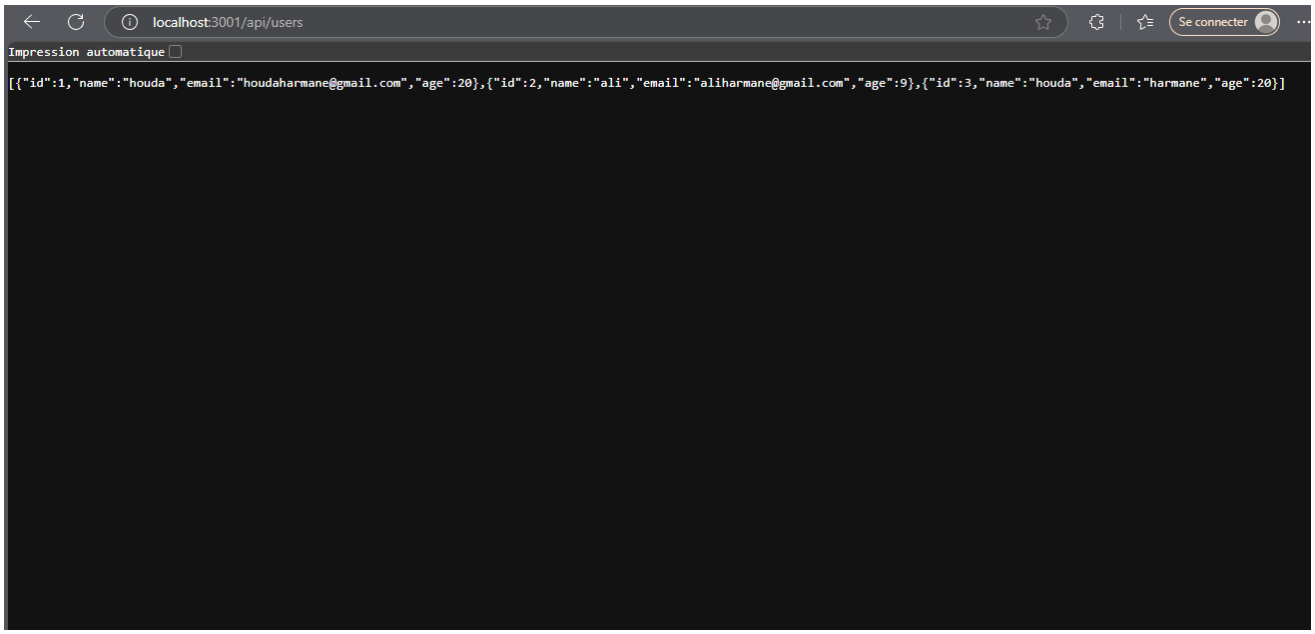


Figure 1 : L'image bakcend.png **affiche une** réponse JSON typique **renvoyée par l'API backend (Express.js) lors d'une requête** GET /api/users**.**

III. Mise en place du Frontend :

```
frontend/
├── public/                 # Dossier des fichiers statiques
│   ├── index.html          # Template HTML principal (point d'entrée)
│   ├── favicon.ico         # Icône de l'application
│   └── assets/             # Dossier pour les images/fonts (optionnel)
│
├── src/                    # Dossier principal du code source
│   ├── components/         # Composants React réutilisables
│   ├── pages/              # Pages de l'application
│   ├── services/           # Logique de communication API (Axios)
│   ├── styles/             # Fichiers CSS/Sass
│   ├── App.jsx             # Composant racine de l'app
│   └── main.jsx            # Point d'entrée JavaScript
│
├── nginx/                  # Configuration NGINX pour Docker
│   └── nginx.conf          # Configuration du serveur web
│
├── .dockerignore           # Fichiers ignorés par Docker
├── .env                    # Variables d'environnement (local)
├── .env.example            # Template des variables d'environnement
├── .gitignore              # Fichiers ignorés par Git
├── Dockerfile              # Configuration Docker pour le frontend
├── package.json            # Dépendances et scripts npm
├── package-lock.json       # Verrouillage des versions
└── README.md               # Documentation du projet
```

1. Composants Principaux

- **UserList**
  - Affiche la liste des utilisateurs dans un tableau responsive.
  - Intègre des boutons d'action (éditer/supprimer).
  - Met à jour dynamiquement les données après chaque opération.
- **UserForm**
  - Formulaire contrôlé avec validation des champs (nom, email).
  - Gère la création et la modification d’utilisateurs.
  - Réinitialisation automatique après soumission.

2. Communication Backend

- **Axios** pour les requêtes HTTP :
  - GET pour récupérer la liste des utilisateurs.
  - POST/PUT pour ajouter/modifier.
  - DELETE pour supprimer.
- Gestion centralisée des erreurs (ex : timeout, serveur indisponible).

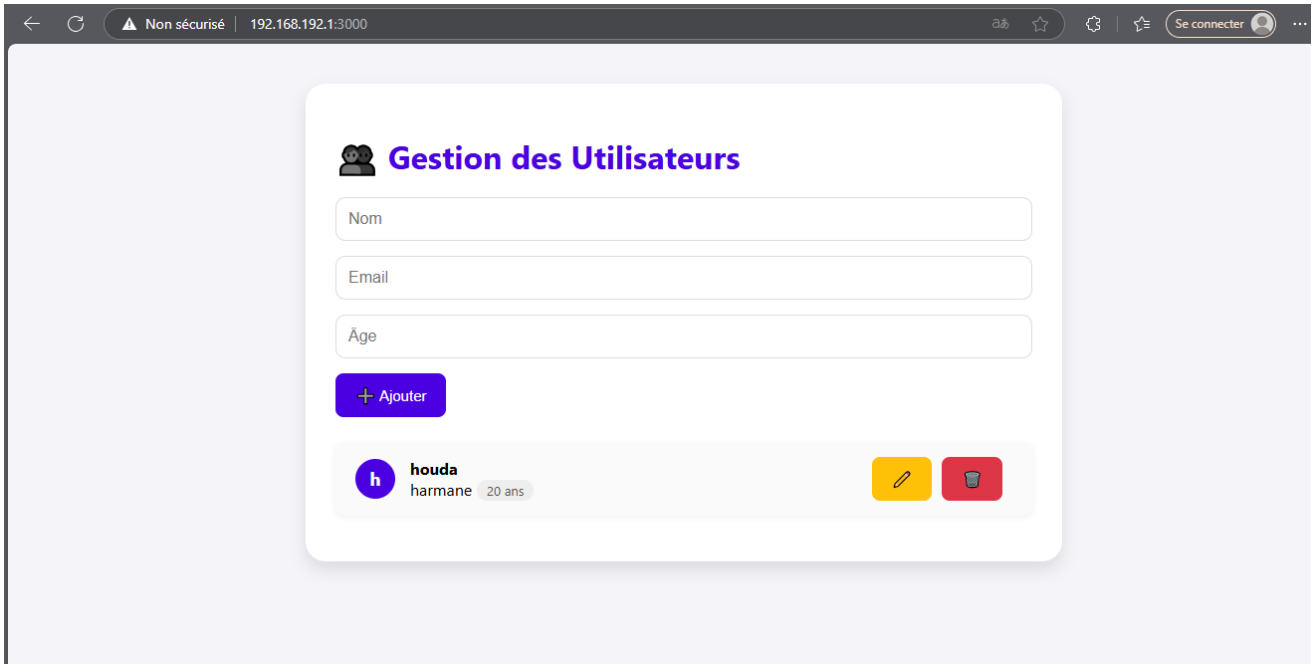


Figure 2 : L'image illustre une interface utilisateur simplifiée pour la gestion des utilisateurs

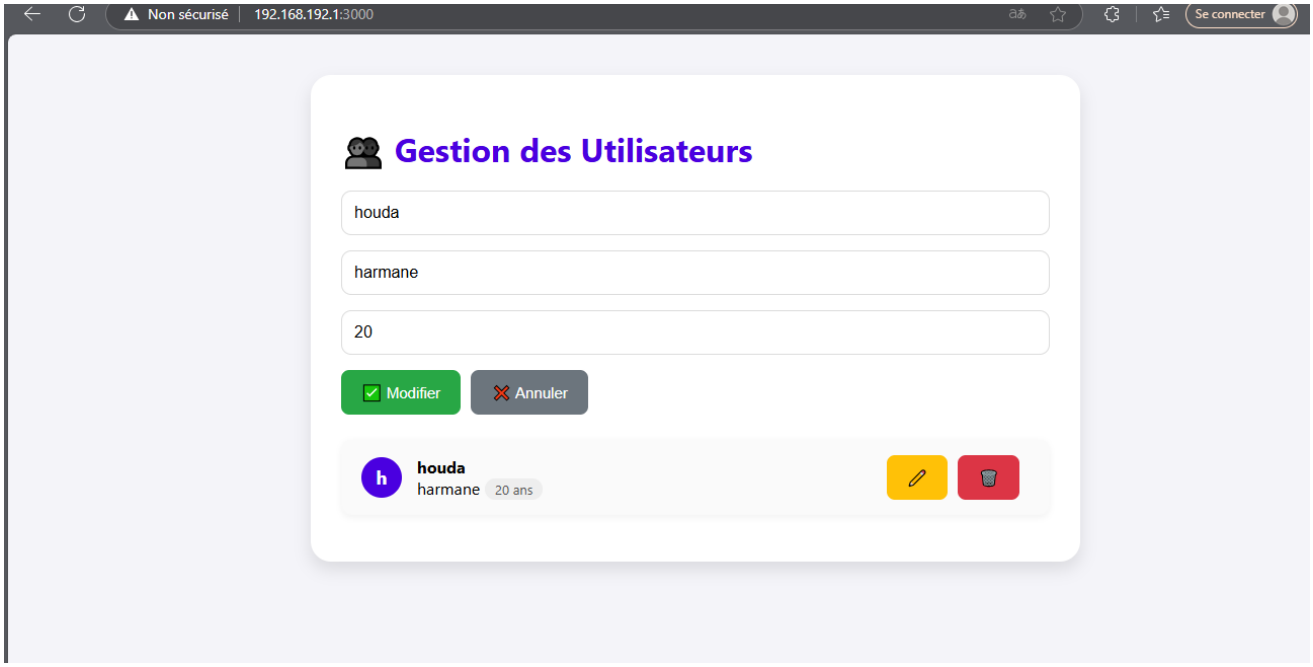


Figure 3 : Cette figure démontre l'interaction entre l'affichage des données et leur modification

IV. Explication de la Base de Données :

Explication de la base de données

Pour ce projet, la base de données choisie est **SQLite**, une solution légère, rapide et parfaitement adaptée à des applications web simples ou en développement local.

Structure de la base

La base de données est constituée d'une seule table nommée `users`, dont la structure est la suivante :

Champ	Type	Contraintes
id	INTEGER	PRIMARY KEY AUTOINCREMENT
name	TEXT	NOT NULL
email	TEXT	NOT NULL, UNIQUE
age	INTEGER	-

Interactions

Les interactions avec la base sont réalisées à travers les routes Express :

- **`GET /api/users`** → récupère tous les utilisateurs
- **`POST /api/users`** → ajoute un nouvel utilisateur
- **`PUT /api/users/:id`** → met à jour un utilisateur
- **`DELETE /api/users/:id`** → supprime un utilisateur

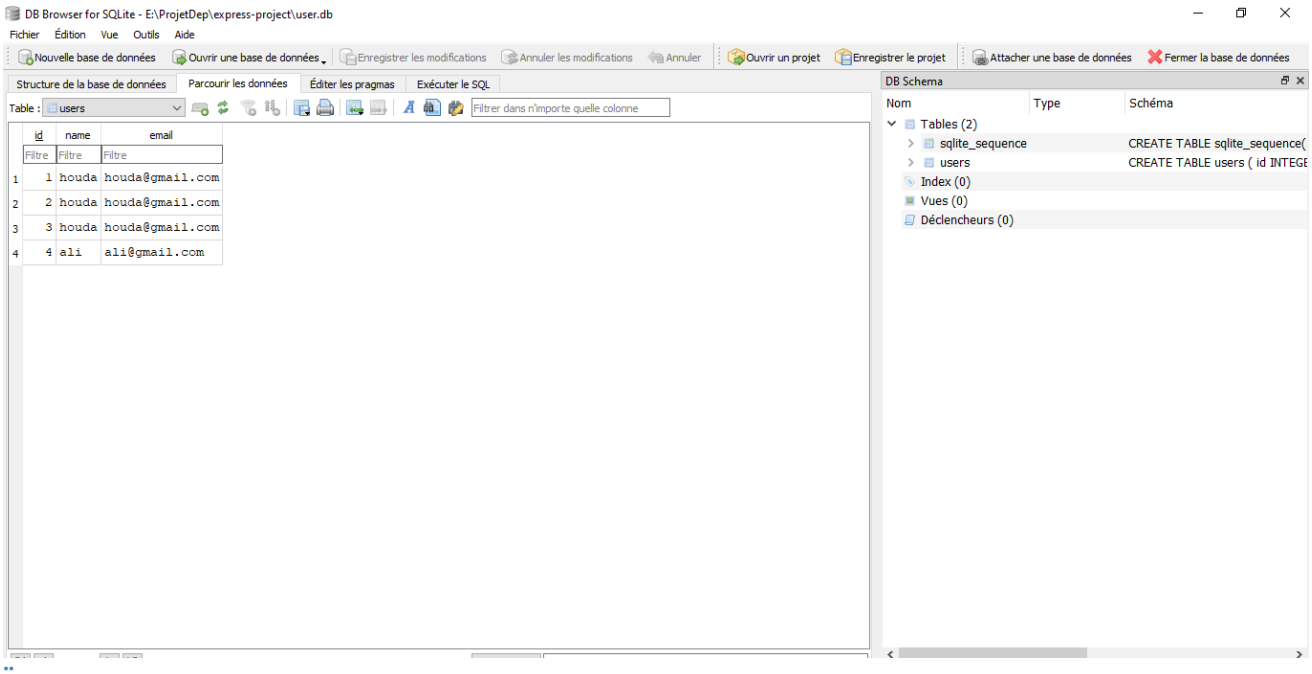


Figure 4 : Capture d'écran de DB Browser for SQLite montrant la structure et le contenu de la table users

V. Dockerisation :

Dans ce projet, Docker a été utilisé pour conteneuriser les deux parties de l'application : le **backend Node.js** et le **frontend React**. L'objectif principal de la Dockerisation était d'assurer une portabilité, une cohérence et une facilité de déploiement entre les différents environnements de développement, test et production.

Étapes de dockerisation

1. Dockerisation du Backend (Express.js + SQLite)

- Création d'un fichier `Dockerfile` dans le dossier `backend`

2. Dockerisation du Frontend (React)

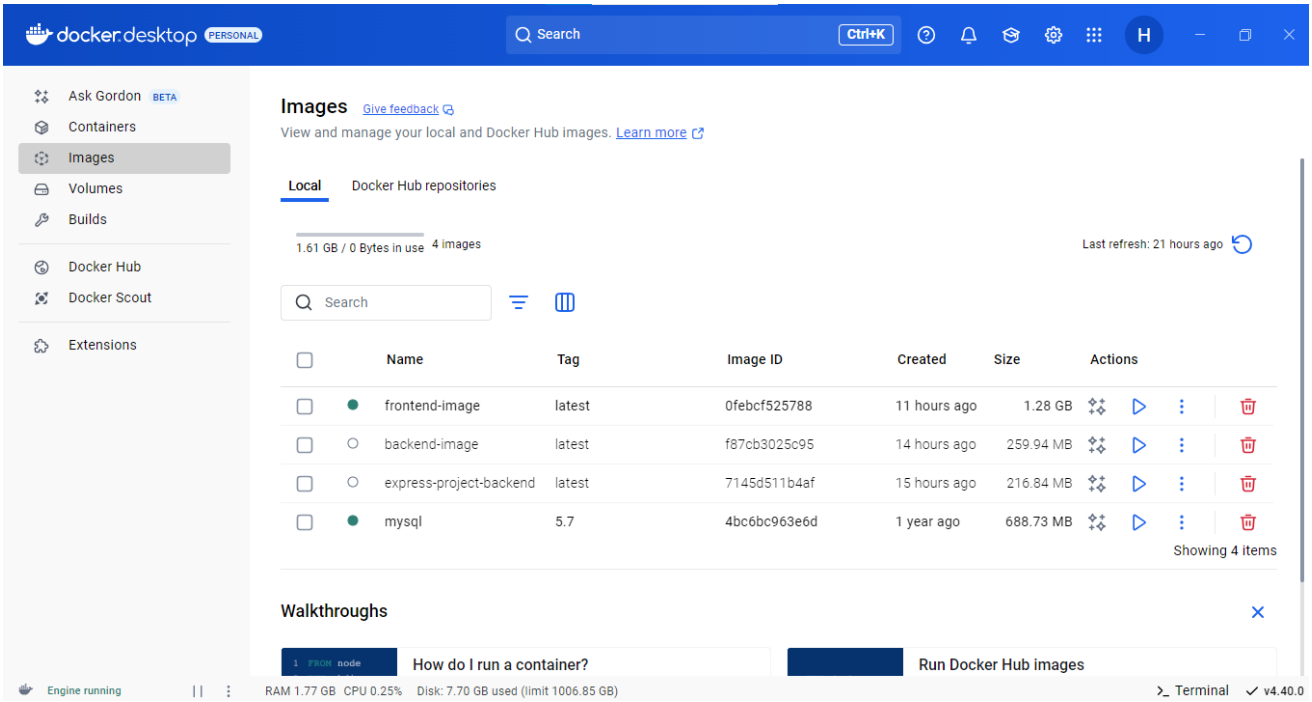
- Création d'un `Dockerfile` dans le dossier `frontend`

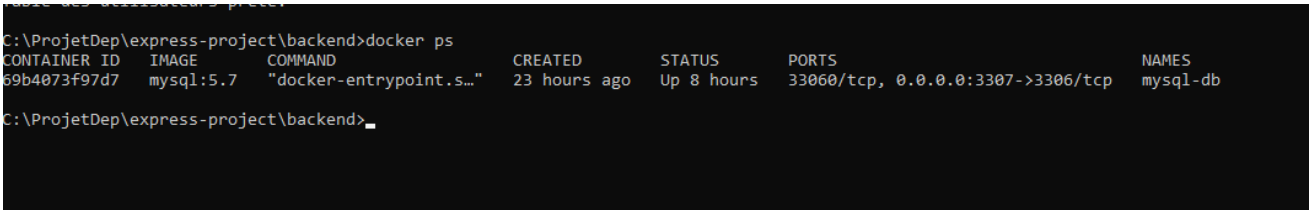
3. Utilisation de `docker-compose`

- Un fichier `docker-compose.yml` a été utilisé pour gérer les deux conteneurs facilement

Choix techniques

- **Node.js comme base image** : légère, officielle et adaptée au projet.
- **Nginx pour le frontend** : rapide, efficace et standard pour servir une app React en production.
- **Docker Compose** : permet de lancer plusieurs conteneurs avec une seule commande (`docker-compose up`).
- **SQLite** : ne nécessite pas de conteneur séparé, simplifiant la configuration du backend.

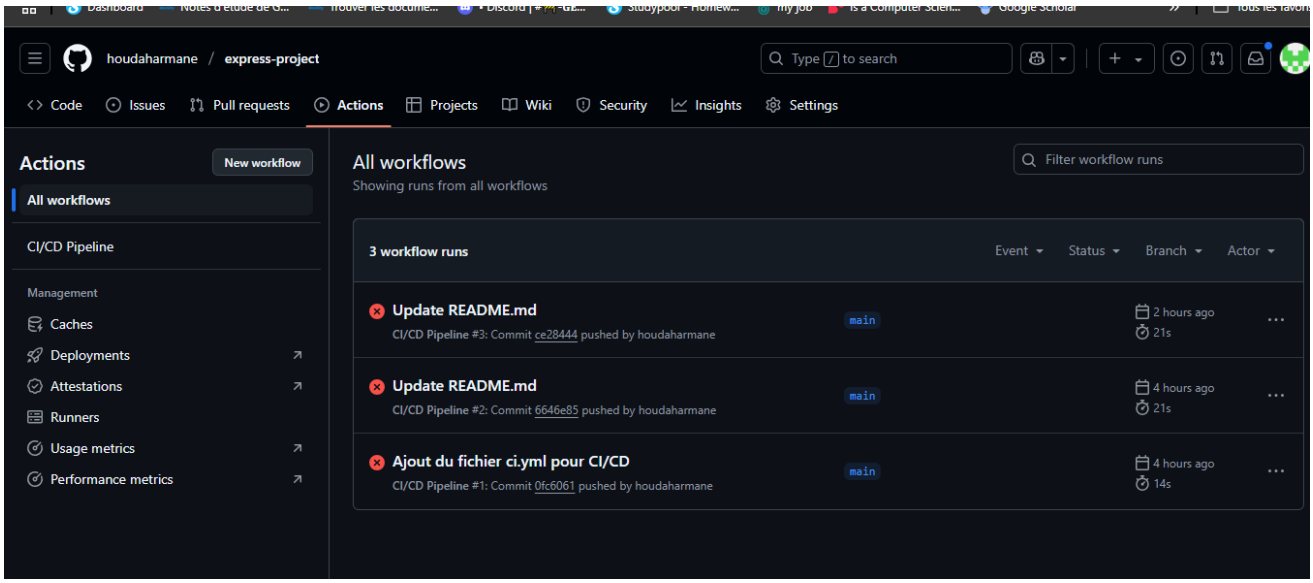




VI. GitHub Actions :

Le pipeline CI/CD a été mis en place à l'aide de **GitHub Actions** à travers un fichier `ci.yml` situé dans `.github/workflows/`.

Ce pipeline permet d'automatiser les étapes suivantes à chaque push ou pull request sur la branche `main`.



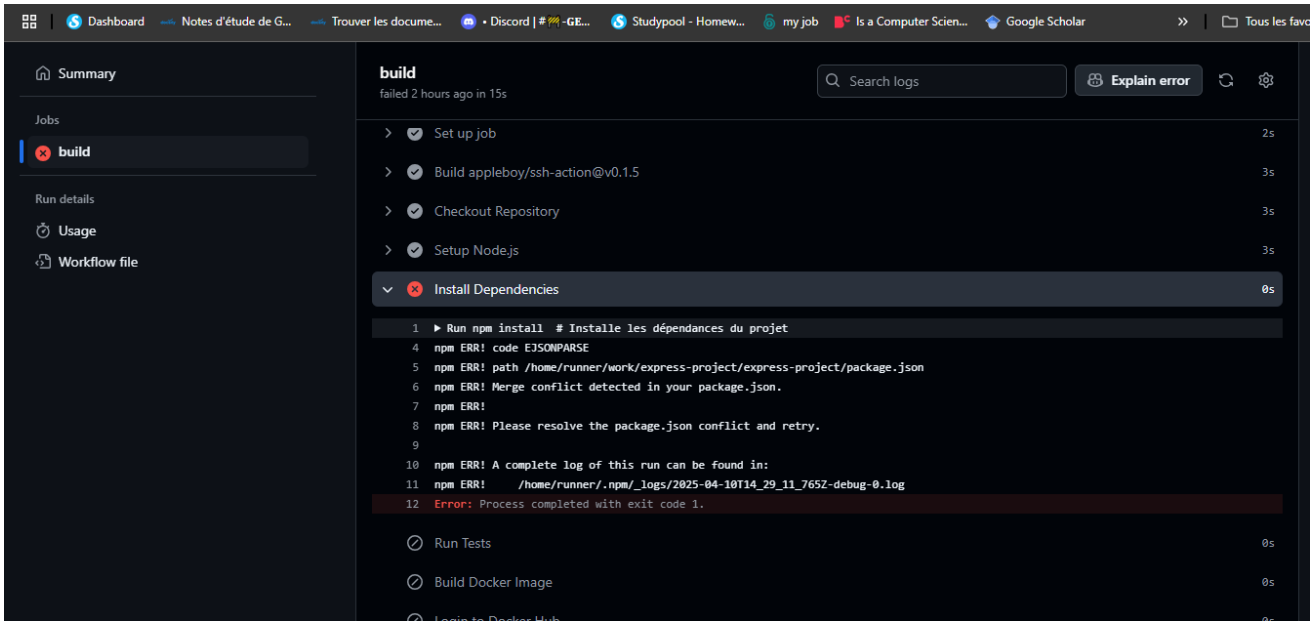


Figure 5 : *Échec du workflow CI/CD lors de l'étape de build Docker*

Difficulté rencontrée : Conflit de fusion dans package.json lors du CI/CD

Description de l'erreur :

Lors de l'exécution du pipeline CI/CD sur GitHub Actions, une erreur critique s'est produite à l'étape d'installation des dépendances avec la commande `npm install`. Le message d'erreur retourné était :

```
npm ERR! code EJSONPARSE

npm ERR! Merge conflict detected in your package.json.
```

Analyse technique :

L'erreur est due à un **conflit de fusion Git non résolu** dans le fichier `package.json`. Ce type de conflit apparaît lorsque deux branches ont modifié les mêmes lignes dans un fichier, et qu'aucune résolution manuelle n'a été effectuée avant le commit. Les marqueurs typiques de conflit `<<<<<<<`, `=======`, `>>>>>>>` étaient encore présents dans le fichier au moment de l'exécution du pipeline.

Impact immédiat :

- Blocage total de l'installation des dépendances (étape essentielle avant l'exécution des tests).
- Interruption du pipeline CI/CD avec un **code d'erreur 1**, ce qui empêche la suite du processus (tests, build Docker, déploiement...).
- Échec visible dans la section *Actions* de GitHub.

Tentatives de résolution :

Plusieurs démarches ont été entreprises :

- Suppression manuelle des marqueurs de conflit dans `package.json`.
- Rebase ou pull avec `--rebase` pour tenter de fusionner proprement.
- Réinitialisation des fichiers conflictuels.
- Tentatives de `git rebase --continue` ou `--abort`.

Résultat :

Malgré tous les essais, **le conflit n'a pas pu être résolu correctement**, ce qui a continué de bloquer le pipeline CI/CD. Une nouvelle version propre du fichier a été envisagée pour repartir sur une base saine.

VII. Problèmes Rencontrés :

1. Conflits de Versionnement

- Conflits fréquents dans les fichiers de configuration lors des fusions entre branches
- Incohérences entre les environnements de développement dus aux versions différentes des dépendances

2. Performances API

- Temps de réponse variables selon la charge
- Blocages de l'interface pendant les appels API longs
- Gestion insuffisante des erreurs réseau

3. Déploiement et CI/CD

- Échecs intermittents des pipelines sans raison apparente
- Variables d'environnement manquantes ou incorrectes en production
- Builds Docker trop longs et gourmands en ressources

4. Tests et Qualité

- Couverture de tests insuffisante sur les composants critiques
- Tests flaky produisant des résultats incohérents
- Absence de tests d'intégration entre frontend et backend

VIII. Conclusion:

Ce projet de développement fullstack avec intégration CI/CD a été riche en apprentissages et en défis techniques. Bien que nous ayons atteint l'objectif principal de mettre en place une application fonctionnelle avec son pipeline de déploiement, plusieurs difficultés sont venues mettre en lumière des axes d'amélioration essentiels pour des projets futurs.