



**Université Mohammed V – Rabat
École Supérieure de Technologie – Salé
Département Informatique**

Développement et Intégration CI/CD d'une Application de Gestion des Utilisateurs

Réalisé par : Benharrouz Fatima Zahra

Encadré par : M. Mohamed El Farouki

Année universitaire : 2024/2025

Table des matières

I.	Présentation générale du projet	3
1.	Technologies utilisées.....	3
II.	Etapes de mise en place du backend et frontend	3
1.	Structure des dossiers	3
2.	Initialisation du backend :.....	4
3.	Initialisation du frontend :.....	4
4.	Documentation de l'API	4
III.	Explication de la base de données	5
IV.	Dockerisation : étapes et choix faits.....	5
V.	Tests automatisés	6
VI.	Intégration continue avec GitHub Actions – CI/CD	8
VII.	Résultats visuels : Tests, Actions GitHub et Conteneurs Docker	11
VIII.	Difficultés rencontrées et solutions.....	18
IX.	Conclusion et axes d'amélioration	19

Table des figures

Figure 1 Architecure du projet	4
Figure 2 fichier de test.....	7
Figure 3 test.....	8
Figure 4 couverture du code	8
Figure 5 Définition du pipeline CI/CD avec GitHub Actions	9
Figure 6 installation des dependances	10
Figure 7 Build et push Docker.....	10
Figure 8 test 1.....	11
Figure 9 Actions secrets.....	12
Figure 10 Actions secrets 2	12
Figure 11 Ajout du workflow CI/CD avec Git	13
Figure 12 Push du projet sur GitHub	13
Figure 13 Exécution réussie du pipeline CI/CD sur GitHub Actions	14
Figure 14 build and test resussi.....	14
Figure 15 docker build and push	15
Figure 16 docker build and push capture 2	16
Figure 17 Logs du conteneur PostgreSQL dans Docker Desktop	16
Figure 18 Résultats des tests automatisés avec Jest et Supertest	17
Figure 19 Backend connecté à PostgreSQL	17
Figure 20 Interface de gestion des utilisateurs	18

I. Présentation générale du projet

Le projet "User Management" est une application web full-stack permettant la gestion des utilisateurs (ajout, modification, suppression). Elle repose sur une architecture moderne avec un backend en Node.js/Express, un frontend en React, une base de données PostgreSQL et une intégration de tests automatisés avec Jest/Supertest. Le déploiement et l'intégration continue sont assurés via Docker et GitHub Actions.

Le code source complet du projet est disponible sur GitHub :

🔗 <https://github.com/fati199734/user-management>

1. Technologies utilisées

Catégorie	Technologies
Frontend	React.js
Backend	Node.js, Express.js
Base de données	PostgreSQL
Tests	Jest, Supertest
Conteneurisation	Docker, Docker Compose
CI/CD	GitHub Actions
Outils associés	Git, Docker Hub, Visual Studio Code, GitHub

II. Étapes de mise en place du backend et frontend

1. Structure des dossiers

L'application est organisée en trois principaux dossiers :

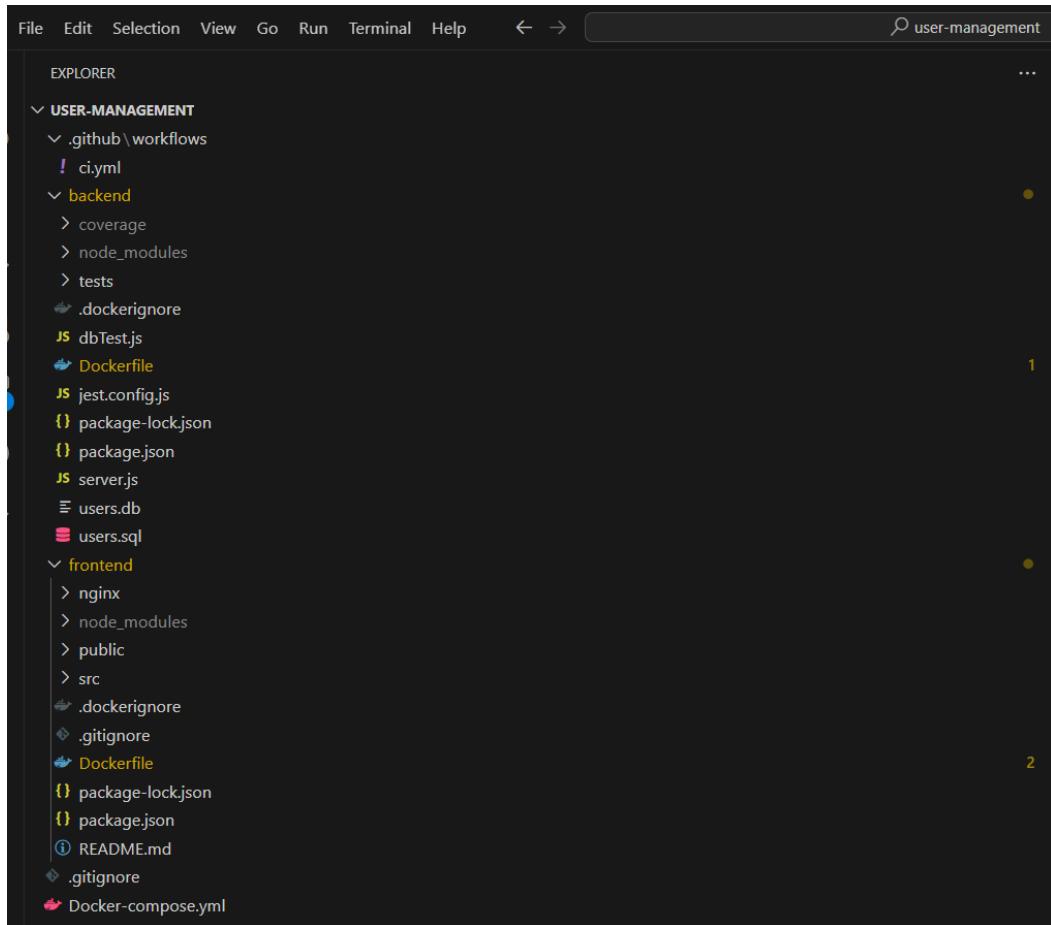


Figure 1 Architecture du projet

2. Initialisation du backend :

- Création du projet Node.js avec Express.
- Définition des routes REST pour les opérations CRUD sur les utilisateurs.
- Connexion à la base de données PostgreSQL via le module pg.
- Mise en place des tests automatisés avec Jest et Supertest.

3. Initialisation du frontend :

- Création du projet React Js.
- Conception de l'interface utilisateur pour la gestion des utilisateurs.
- Intégration des appels API avec Axios.
- Utilisation des hooks useState et useEffect pour la gestion des états.

4. Documentation de l'API

Le backend expose les routes suivantes :

- POST /api/users : créer un nouvel utilisateur

- GET /api/users : récupérer tous les utilisateurs
- PUT /api/users/:id : mettre à jour un utilisateur (optionnel)
- DELETE /api/users/:id : supprimer un utilisateur (optionnel)

Les données sont échangées au format **JSON**.

III. Explication de la base de données

L'application utilise **PostgreSQL** comme système de gestion de base de données. Elle stocke les informations des utilisateurs dans une table users avec les champs suivants :

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

Connexion à PostgreSQL : via la bibliothèque pg dans le backend.

Création automatique : la table users est créée au démarrage si elle n'existe pas.

Pour les tests : une base temporaire SQLite est utilisée avec création et suppression automatique de la table.

IV. Dockerisation : étapes et choix faits

1. Objectif :

- Conteneuriser le backend, le frontend et la base de données pour faciliter le déploiement et la portabilité du projet.
- Automatiser le lancement de l'application avec Docker Compose.

2. Fichiers utilisés :

- Dockerfile pour le backend (Node.js/Express),
- Dockerfile pour le frontend (React avec build Nginx),
- docker-compose.yml pour orchestrer tous les services.

3. Services déclarés :

- backend : expose le port 3001, se connecte à PostgreSQL via les variables d'environnement,
- frontend : build React puis sert avec Nginx sur le port 3000,
- db : PostgreSQL, persisté avec un volume Docker.

4. Choix techniques :

- Utilisation d'images légères (node:alpine, nginx:alpine),
- Utilisation de volumes pour la persistance de la base de données,
- Réseaux Docker internes pour l'interconnexion des conteneurs.

5. Résultat :

L'application complète fonctionne en exécutant une seule commande :
docker-compose up –build

V. Tests automatisés

1. Objectif des tests

Voici les outils utilisés dans le projet : Les tests automatisés permettent de vérifier automatiquement que le code fonctionne comme prévu. Cela permet de :

- Déetecter rapidement les erreurs après chaque modification
- Automatiser les vérifications à chaque changement de code
- Garantir la stabilité de l'application backend (API)

2. Types de tests utilisés

Deux types de tests ont été implémentés :

- Test unitaire : teste une fonctionnalité isolée (ex. : calcul, vérification simple)
- Test d'intégration : teste une ou plusieurs routes de l'API avec une base de données en mémoire

3. Environnement de test

Pour les tests, une base de données SQLite en mémoire est utilisée à la place de PostgreSQL, ce qui permet :

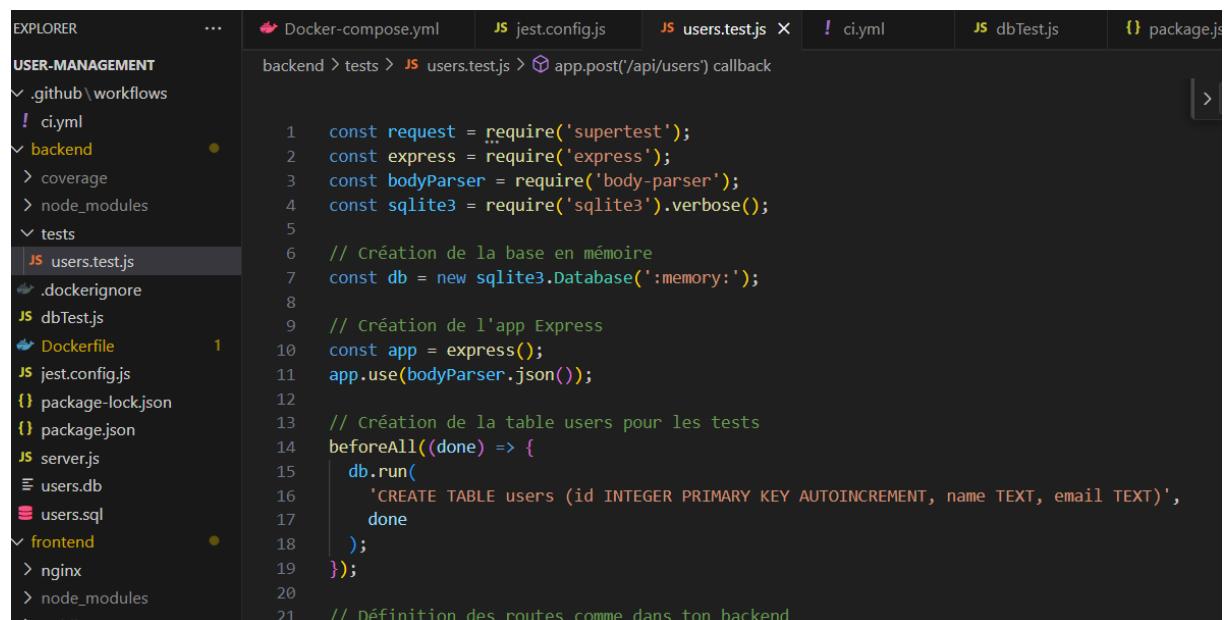
- D'exécuter les tests rapidement sans modifier la vraie base de données
- De garantir un environnement de test propre à chaque exécution

4. Outils et bibliothèques

Outil	Utilité
jest	Framework de tests JavaScript
supertest	Pour tester les routes HTTP de l'API
sqlite3	Base de données temporaire pour les tests
nodemon	Rechargement automatique pendant le dev

5. Exemple de test automatisé

Un fichier de test users.test.js a été créé dans le dossier tests/.



```
const request = require('supertest');
const express = require('express');
const bodyParser = require('body-parser');
const sqlite3 = require('sqlite3').verbose();

// Création de la base en mémoire
const db = new sqlite3.Database(':memory:');

// Création de l'app Express
const app = express();
app.use(bodyParser.json());

// Création de la table users pour les tests
beforeAll((done) => {
  db.run(
    'CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, email TEXT)',
    done
  );
});

// Définition des routes comme dans ton backend
```

Figure 2 fichier de test

6. Commande d'exécution

Les tests sont exécutés avec la commande suivante : **npm test**

```

Ran all test suites.
● PS C:\lp\user-management\backend> npm test
>>

> backend@1.0.0 test
> jest

PASS tests/users.test.js
  ✓ devrait créer un utilisateur (30 ms)
  ✓ devrait récupérer les utilisateurs (4 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        0.79 s, estimated 1 s
Ran all test suites.          npm test

```

Figure 3 test

7. Couverture de code

Pour générer la couverture de code, on peut ajouter ceci dans le fichier package.json : jest – coverage puis « npm test »

```

>> C:\lp\user-management\backend>

> backend@1.0.0 test
> jest --coverage

PASS tests/users.test.js
  ✓ devrait créer un utilisateur (41 ms)
  ✓ devrait récupérer les utilisateurs (6 ms)

-----|-----|-----|-----|-----|-----|
File   | %Stmts | %Branch | %Funcs | %Lines | Uncovered Line #s
server.js |    0 |    0 |    0 |    0 | 1-106
backend/coverage/lcov-report | 0 | 0 | 0 | 0 |
block-navigation.js | 0 | 0 | 0 | 0 | 2-87
prettify.js | 0 | 0 | 0 | 0 | 2
sorter.js | 0 | 0 | 0 | 0 | 2-196
-----|-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:  0 total
Time:        6.164 s
Ran all test suites.
PS C:\lp\user-management\backend>

```

Figure 4 couverture du code

VI. Intégration continue avec GitHub Actions – CI/CD

1. Objectif de l'intégration continue

L'intégration continue (CI) consiste à automatiser l'exécution des tests et des vérifications à chaque modification du code dans le dépôt GitHub. Cela permet :

- De garantir que le projet reste fonctionnel
- D'éviter l'introduction de bugs après une modification
- De tester sur un serveur distant automatiquement après chaque push ou pull request

2. Outil utilisé : GitHub Actions

GitHub Actions est un service intégré dans GitHub permettant de définir des workflows d'automatisation.

3. Fichier de configuration du workflow

Un fichier nommé ci.yml a été créé dans le dossier .github/workflows/.

The screenshot shows a GitHub Actions workflow configuration file named 'ci.yml' in a code editor. The file is located in the '.github/workflows' directory. The code defines a pipeline named 'CI/CD Pipeline' that triggers on pushes and pull requests to the 'main' branch. It contains a single job named 'build-and-test' which runs on an Ubuntu latest runner and includes a step to checkout the repository using the actions/checkout@v3 action.

```
1 name: CI/CD Pipeline
2
3 on:
4   push:
5     branches:
6       - main
7   pull_request:
8     branches:
9       - main
10
11 jobs:
12   build-and-test:
13     runs-on: ubuntu-latest
14
15   steps:
16     - name: Checkout du repo
17       uses: actions/checkout@v3
```

Figure 5 Définition du pipeline CI/CD avec GitHub Actions

```
0
1 jobs:
2   build-and-test:
3     runs-on: ubuntu-latest
4
5     steps:
6       - name: Checkout du repo
7         uses: actions/checkout@v3
8
9       - name: Setup Node.js
10        uses: actions/setup-node@v3
11        with:
12          node-version: '18'
13
14       - name: Installation des dépendances
15         run: |
16           cd backend
17           npm install
18
19       - name: Lancement des tests
20         run: |
21           cd backend
22           npm test
```

Figure 6 installation des dependances

```
1
2 docker-build-and-push:
3   needs: build-and-test
4   runs-on: ubuntu-latest
5
6   steps:
7     - name: Checkout du repo
8       uses: actions/checkout@v3
9
10    - name: Connexion à Docker Hub
11      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin
12
13    - name: Build de l'image Docker
14      run: docker build -t ${{ secrets.DOCKER_USERNAME }}/user-management-backend ./backend
15
16    - name: Push de l'image vers Docker Hub
17      run: docker push ${{ secrets.DOCKER_USERNAME }}/user-management-backend
```

Figure 7 Build et push Docker

1. Fonctionnement du pipeline

Le pipeline CI/CD mis en place avec **GitHub Actions** permet d'automatiser les étapes de test et de déploiement de l'application à chaque mise à jour du code source.

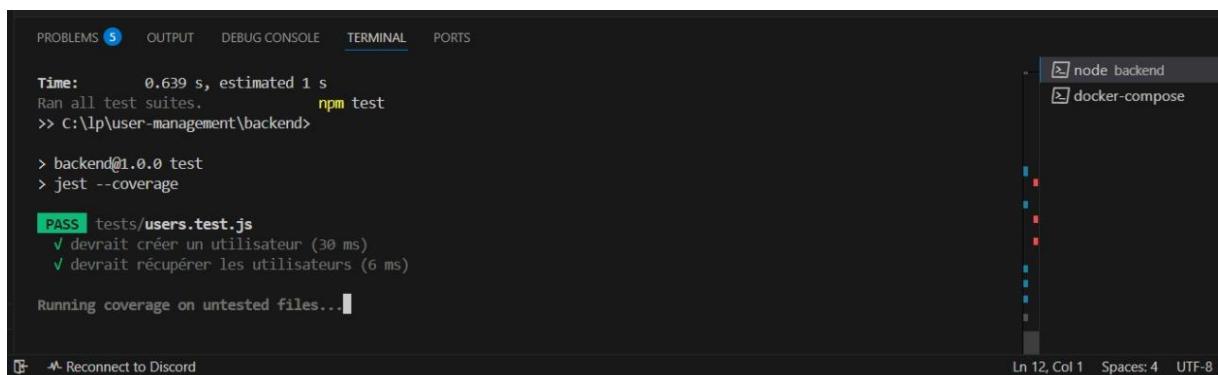
Chaque fois qu'un développeur pousse du code sur la branche main, le workflow se déclenche automatiquement. Il effectue les actions suivantes :

- Récupère le code du dépôt GitHub.
- Installe les dépendances du projet.
- Lance les tests unitaires pour valider le bon fonctionnement.
- Si les tests sont validés, le pipeline se connecte à DockerHub via des identifiants sécurisés (secrets).

- Une nouvelle image Docker du backend est construite et poussée automatiquement vers DockerHub.

Ce processus garantit que le code est toujours testé, validé et prêt à être déployé dans un environnement de production.

VII. Résultats visuels : Tests, Actions GitHub et Conteneurs Docker



The screenshot shows a terminal window with the following output:

```

PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Time: 0.639 s, estimated 1 s
Ran all test suites. npm test
>> C:\lp\user-management\backend>

> backend@1.0.0 test
> jest --coverage

PASS tests/users.test.js
  ✓ devrait créer un utilisateur (30 ms)
  ✓ devrait récupérer les utilisateurs (6 ms)

Running coverage on untested files...

```

To the right of the terminal, a Docker Compose file is visible:

```

version: '3'
services:
  node_backend:
    build: ./node
  docker-compose:
    build: ./compose

```

At the bottom of the terminal window, status information is displayed: Ln 12, Col 1, Spaces: 4, UTF-8.

Figure 8 test 1

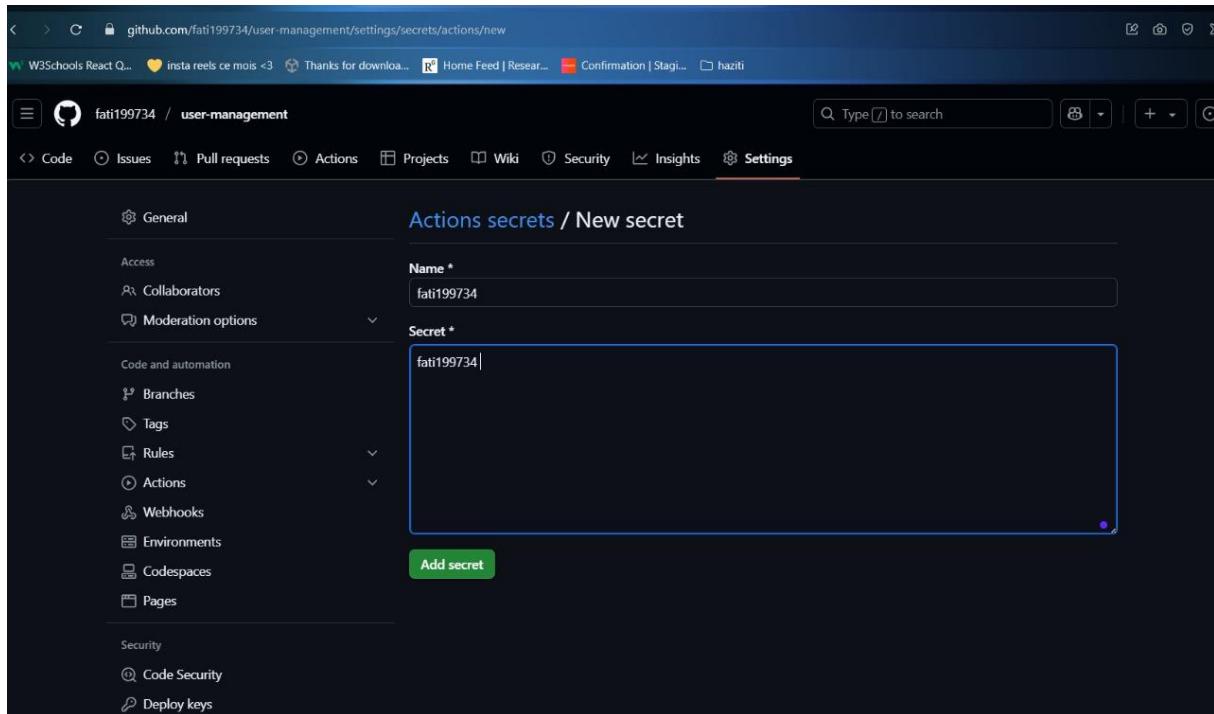


Figure 9 Actions secrets

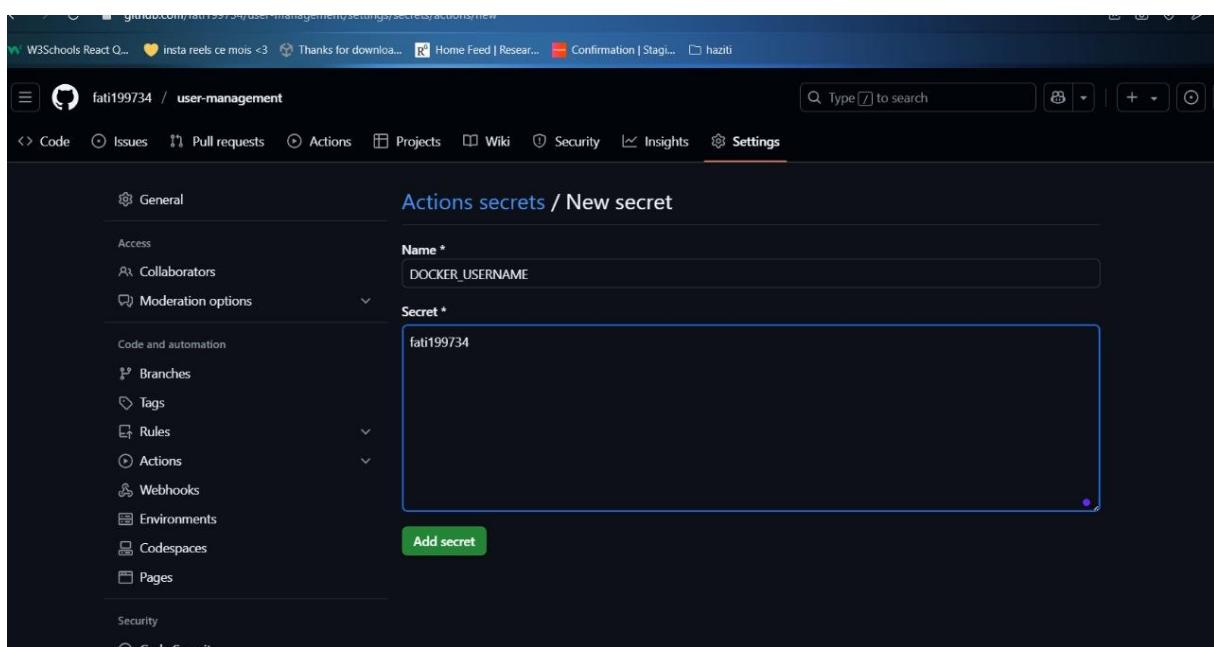


Figure 10 Actions secrets 2

```
● PS C:\lp\user-management> git add .
warning: in the working copy of 'backend/package-lock.json', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'backend/package.json', LF will be replaced by CRLF the next time Git touches it
PS C:\lp\user-management> git commit -m "Ajout de GitHub Actions pour CI/CD"
● >>
[main acc1f2a] Ajout de GitHub Actions pour CI/CD
  15 files changed, 5208 insertions(+), 989 deletions(-)
   create mode 100644 .github/workflows/ci.yml
   create mode 100644 Docker-compose.yml
   create mode 100644 backend/.dockerignore
   create mode 100644 backend/Dockerfile
   create mode 100644 backend/dbTest.js
   create mode 100644 backend/jest.config.js
   create mode 100644 backend/tests/users.test.js
   create mode 100644 backend/users.sql
   create mode 100644 frontend/.dockerignore
   create mode 100644 frontend/Dockerfile
   create mode 100644 frontend/package-default.json
```

Figure 11 Ajout du workflow CI/CD avec Git

```
PS C:\lp\user-management>
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 213 bytes | 213.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/fati199734/user-management.git
  2d2a203..33b94e4 main -> main
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 213 bytes | 213.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 213 bytes | 213.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/fati199734/user-management.git
  2d2a203..33b94e4 main -> main
PS C:\lp\user-management> 
```

Figure 12 Push du projet sur GitHub

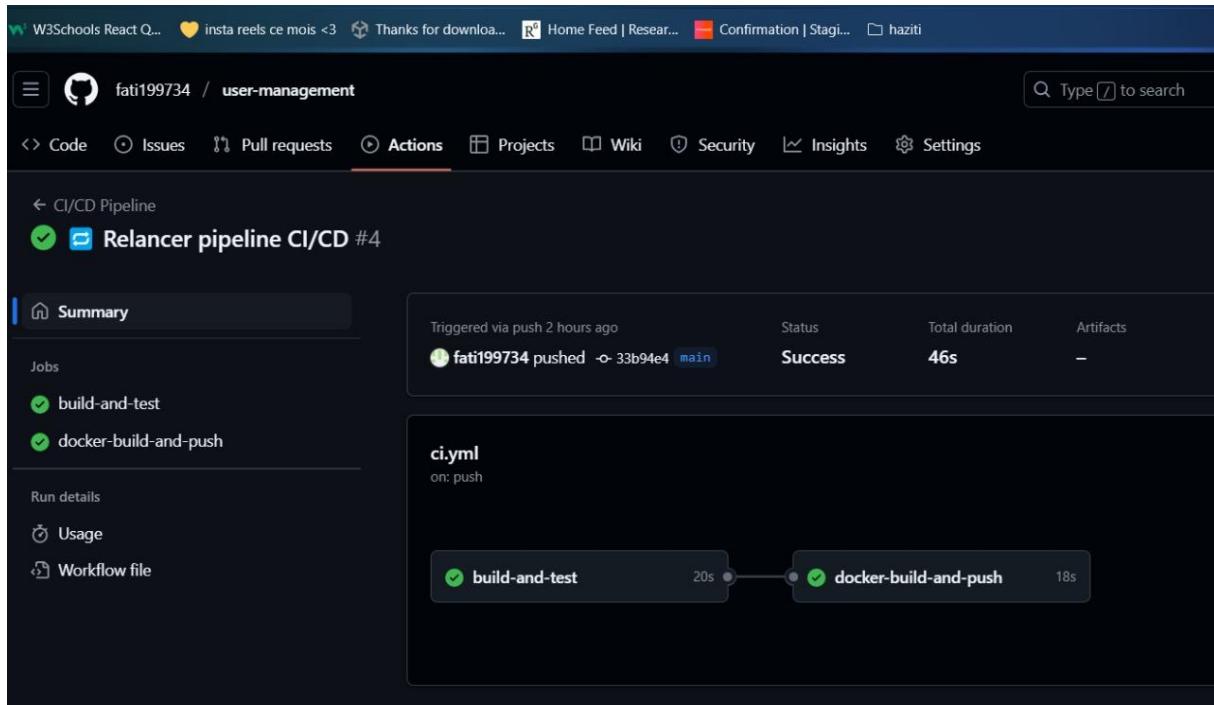


Figure 13 Exécution réussie du pipeline CI/CD sur GitHub Actions

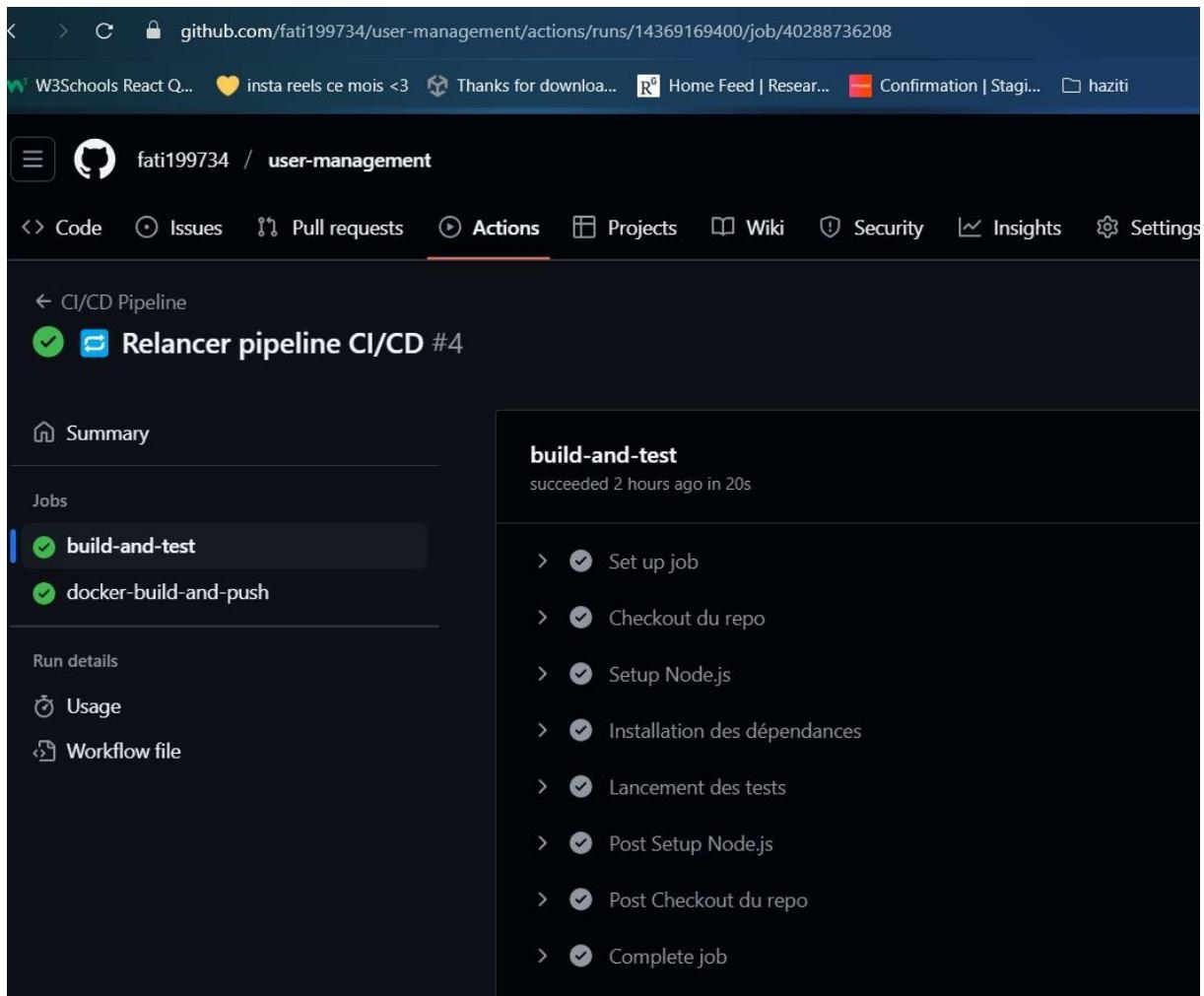


Figure 14 build and test resussi

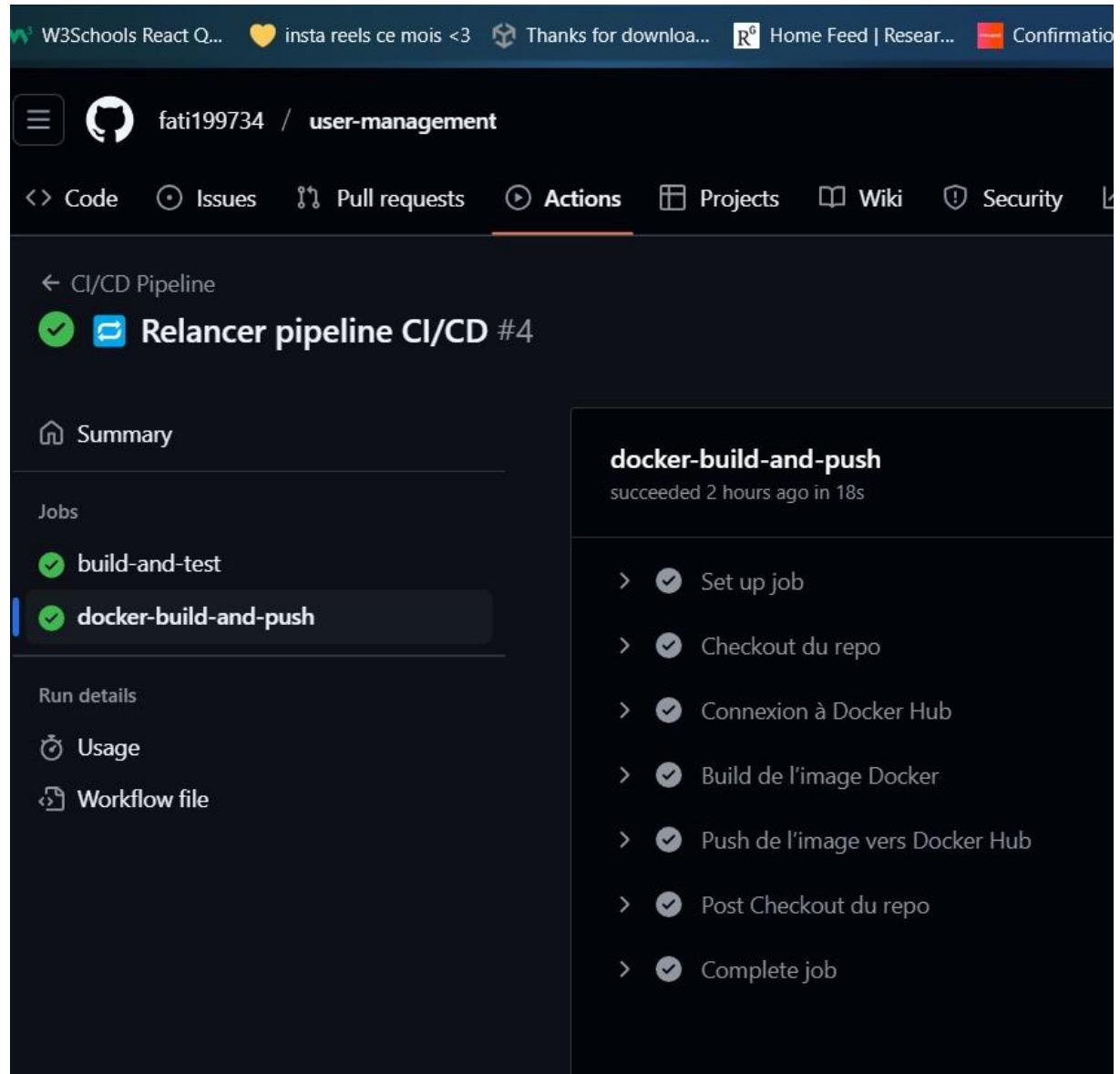


Figure 15 docker build and push

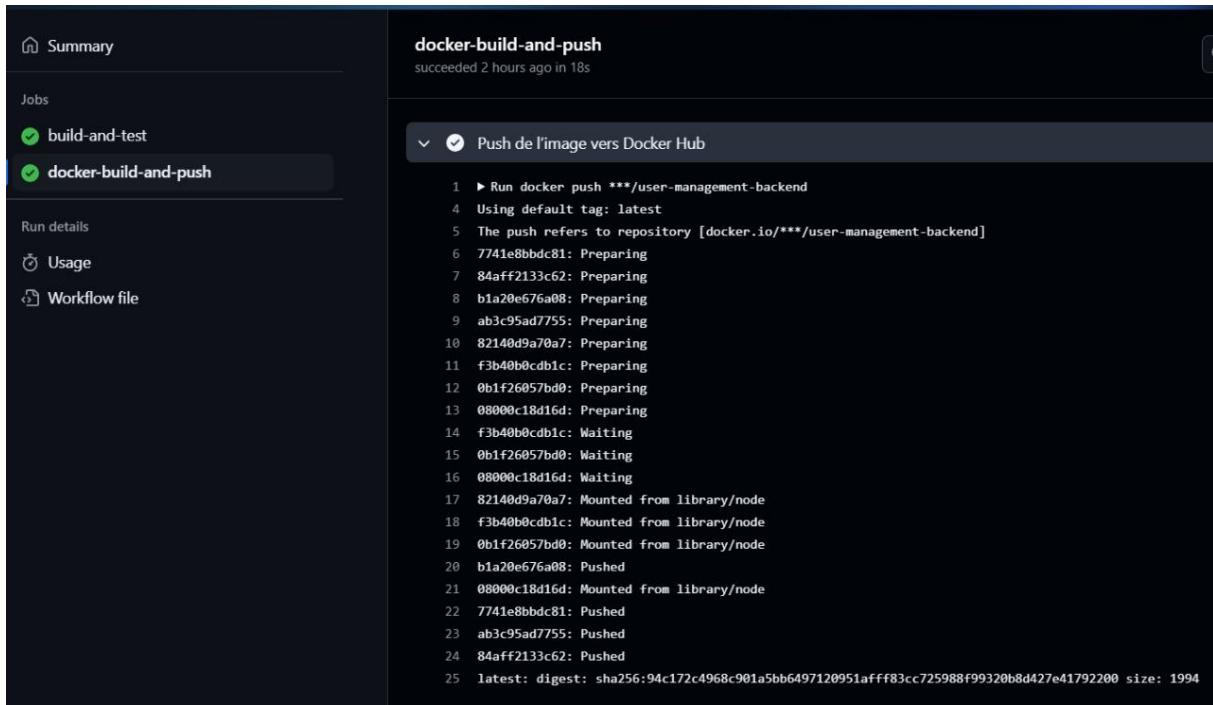


Figure 16 docker build and push capture 2



Figure 17 Logs du conteneur PostgreSQL dans Docker Desktop

The screenshot shows a terminal window with the following output:

```
PROBLEMS 5 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Time: 6.164 s
Ran all test suites.
-----
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
-----
Test Suites: 1 passed, 1 total
-----
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
-----
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
-----
Test Suites: 1 passed, 1 total
-----
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 6.164 s
Ran all test suites.
PS C:\lp\user-management\backend> 
```

Figure 18 Résultats des tests automatisés avec Jest et Supertest

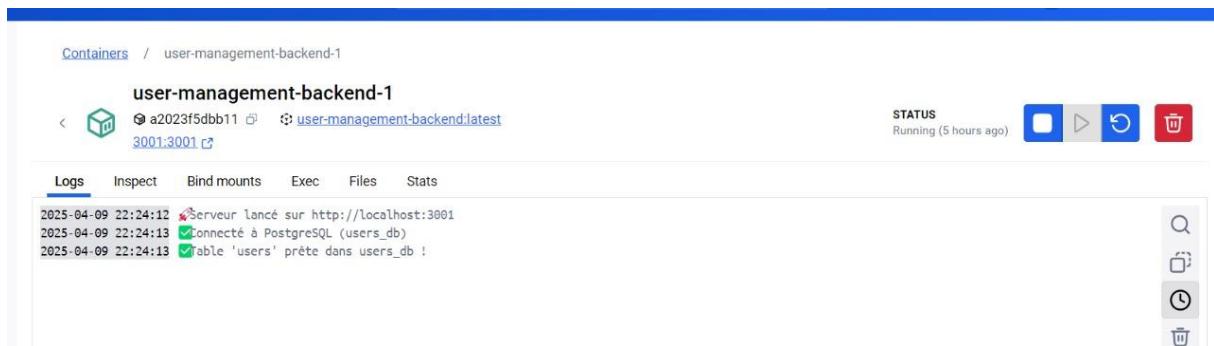


Figure 19 Backend connecté à PostgreSQL

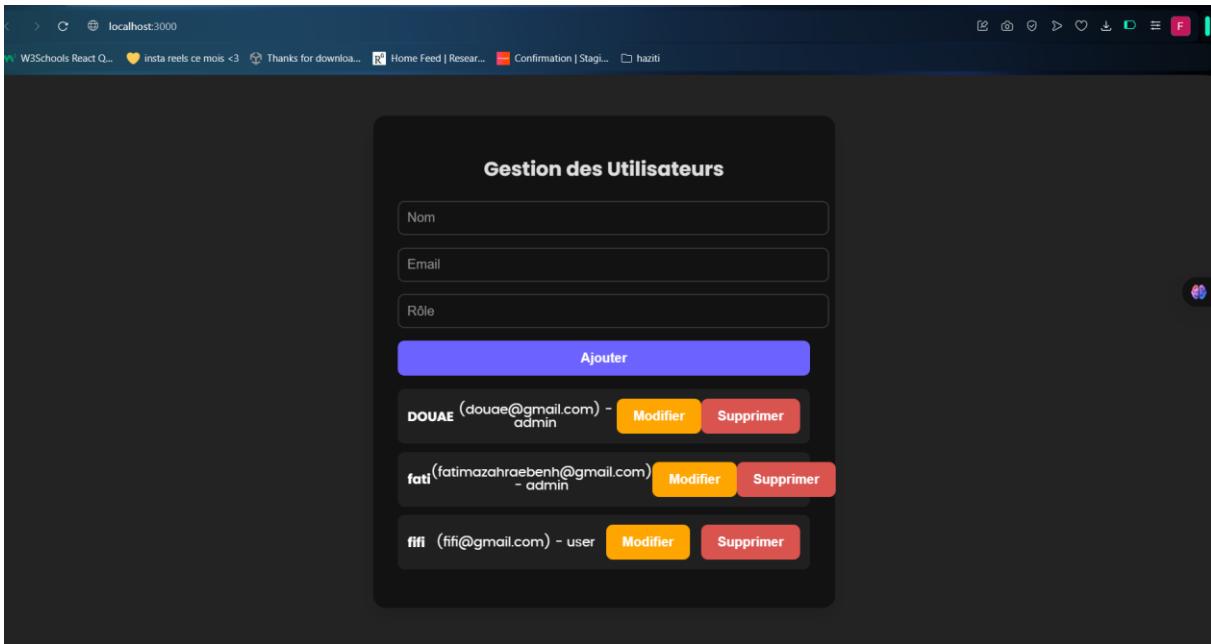


Figure 20 Interface de gestion des utilisateurs

VIII. Difficultés rencontrées et solutions

- **Problème d'authentification DockerHub** : l'accès a échoué lors du push de l'image. La solution a été de générer un nouveau token d'accès et de le configurer dans les secrets GitHub.
- **Erreurs de correspondance des ports** : certains ports définis dans `docker-compose.yml` entraient en conflit. Ils ont été corrigés pour assurer une bonne communication entre services.
- **Tests avec PostgreSQL en local** : les tests unitaires ne devaient pas affecter la base de production. La solution a été d'utiliser SQLite en mémoire uniquement pour les tests.
- **Manque d'expérience avec GitHub Actions** : les premières configurations échouaient. Après lecture de la documentation officielle et tests progressifs, un pipeline fonctionnel a été mis en place.

IX. Conclusion et axes d'amélioration

Le projet "User Management" a permis de mettre en œuvre une application web complète avec un backend Express, une base de données PostgreSQL, un frontend React, et une intégration automatisée avec Docker et GitHub Actions. L'ensemble des fonctionnalités prévues a été réalisé, les tests ont été mis en place et la CI/CD fonctionne correctement.

Axes d'amélioration :

- Ajouter une interface utilisateur plus ergonomique et responsive.
- Implémenter l'authentification (JWT).
- Ajouter des tests de bout en bout (E2E) avec des outils comme Cypress.
- Déployer l'application sur une plateforme cloud (ex : Render, Vercel, ou un VPS personnel).
- Ajouter une gestion avancée des rôles utilisateurs (admin, éditeur...).