

RAPPORT TECHNIQUE

APPLICATION DE GESTION DES UTILISATEURS – FULL STACK AVEC CI/CD

**Réalisé par :
Amraoui Hind**

IAWM

Année universitaire : 2024 – 2025

SOMMAIRE



1. Introduction

1.1 Objectifs du projet

1.2 Technologies utilisées

1.3 Architecture globale

2. Développement de l'application

2.1 Backend avec Express.js

2.2 Frontend avec React.js

2.3 Communication API (Axios)

2.4 Tests des API avec Postman

3. Dockerisation de l'application

4. Mise en œuvre des tests

5. Intégration Continue avec GitHub Actions

6. Déploiement de l'application

7. Conclusion

1.1 OBJECTIFS DU PROJET

Ce projet a pour objectif de concevoir une application web full-stack permettant de gérer des utilisateurs à travers une interface conviviale et dynamique. Il offre les fonctionnalités classiques d'un système CRUD (Create, Read, Update, Delete) à l'aide d'une API REST développée avec Express.js côté backend, et une interface utilisateur développée avec React.js côté frontend.

En plus du développement, le projet met en œuvre des pratiques professionnelles modernes telles que la dockerisation, les tests automatisés avec Mocha/Chai, et l'intégration continue (CI/CD) avec GitHub Actions.

Ce projet est également l'occasion de travailler pour la première fois avec le système de gestion de base de données PostgreSQL, utilisé ici comme base de données relationnelle.

1.2 Technologies utilisées

Le projet repose sur des technologies récentes et performantes du développement web, ainsi que des outils de conteneurisation :

- Backend :
 - Node.js avec Express.js
 - PostgreSQL (base de données relationnelle)
 - pg (client PostgreSQL pour Node.js)
 - Mocha, Chai, Supertest (frameworks de test)
- Frontend :
 - React.js (via Vite)
 - Axios (pour les appels HTTP)
 - CSS (pour le style)
- Infrastructure & DevOps :
 - Docker & Docker Compose (pour l'environnement de développement)
 - Git & GitHub
 - GitHub Actions (pour le pipeline CI/CD)
 - Nginx (serveur web pour servir le frontend en production)

1.3 Architecture globale

Le projet suit une architecture en trois services principaux gérés avec Docker :

- Frontend (React.js) : Application web accessible via le navigateur, qui consomme les données de l'API backend.
- Backend (Express.js) : API REST permettant d'interagir avec la base de données PostgreSQL.
- Base de données PostgreSQL : Conteneur dédié à la gestion des données persistantes.

Le tout est orchestré via Docker Compose, ce qui facilite le développement local, les tests, et le déploiement. L'arborescence du projet est structurée en trois dossiers principaux (frontend, backend, .github) et un fichier docker-compose.yml à la racine.

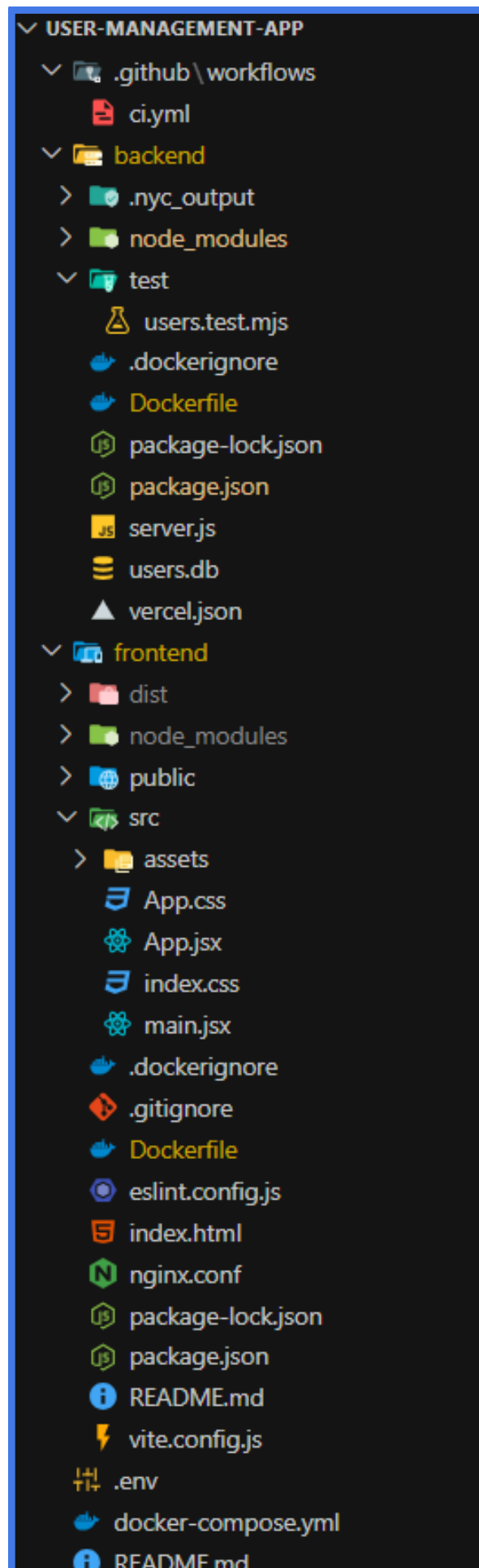


Figure 1 - Arborescence complète du projet user-management-app

2.1 Backend avec Express.js

Le backend de l'application est développé à l'aide de Node.js et Express.js. Il expose une API REST permettant de gérer des utilisateurs via les opérations suivantes :

- **GET /users** : récupérer tous les utilisateurs.
- **GET /users/:id** : récupérer un utilisateur par ID.
- **POST /users** : ajouter un nouvel utilisateur.
- **PUT /users/:id** : mettre à jour les informations d'un utilisateur.
- **DELETE /users/:id** : supprimer un utilisateur.

Pour la gestion des données, nous utilisons PostgreSQL. Cela permet une manipulation simplifiée des modèles et des requêtes SQL.

```
PS C:\Users\dell\user-management-app\backend> node server.js
Serveur backend sur http://localhost:5174
Connecté à PostgreSQL
Table "users" créée ou déjà existante.
```

Figure 2 - Démarrage réussi du serveur backend

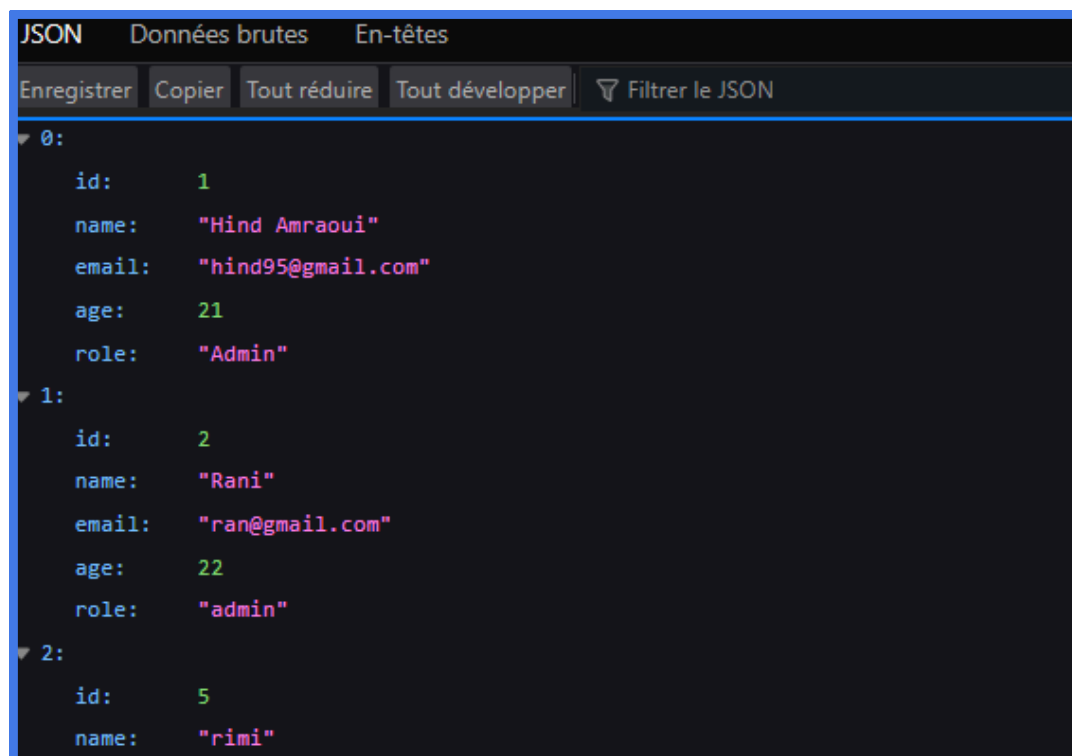


Figure 3 - Extrait des données utilisateurs en base

2.2 Frontend avec React.js

Le frontend est développé avec React.js. Il permet à l'utilisateur d'interagir facilement avec l'API backend à travers une interface claire. Les principales fonctionnalités côté client incluent :

- Affichage de la liste des utilisateurs
- Ajout d'un nouvel utilisateur via formulaire
- Modification des informations d'un utilisateur
- Suppression d'un utilisateur.

```
PS C:\Users\dell\user-management-app\frontend> npm run dev

> frontend@0.0.0 dev
> vite

VITE v6.2.5 ready in 1021 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

Figure 4 - Démarrage du serveur de frontend avec Vite

Gestion des utilisateurs

Ajouter

Liste des utilisateurs

Nom	Email	Âge	Rôle	Actions
Hind	hind@gmail.com	21 ans	User	 
Chifae	chifae@gmail.com	12 ans	User	 
Meryam	mery@gmail.com	23 ans	Admin	 

Figure 5 - l'interface utilisateur

2.3 Communication API (Axios)

La communication entre le frontend et le backend se fait à l'aide de la bibliothèque Axios, qui facilite l'envoi de requêtes HTTP.

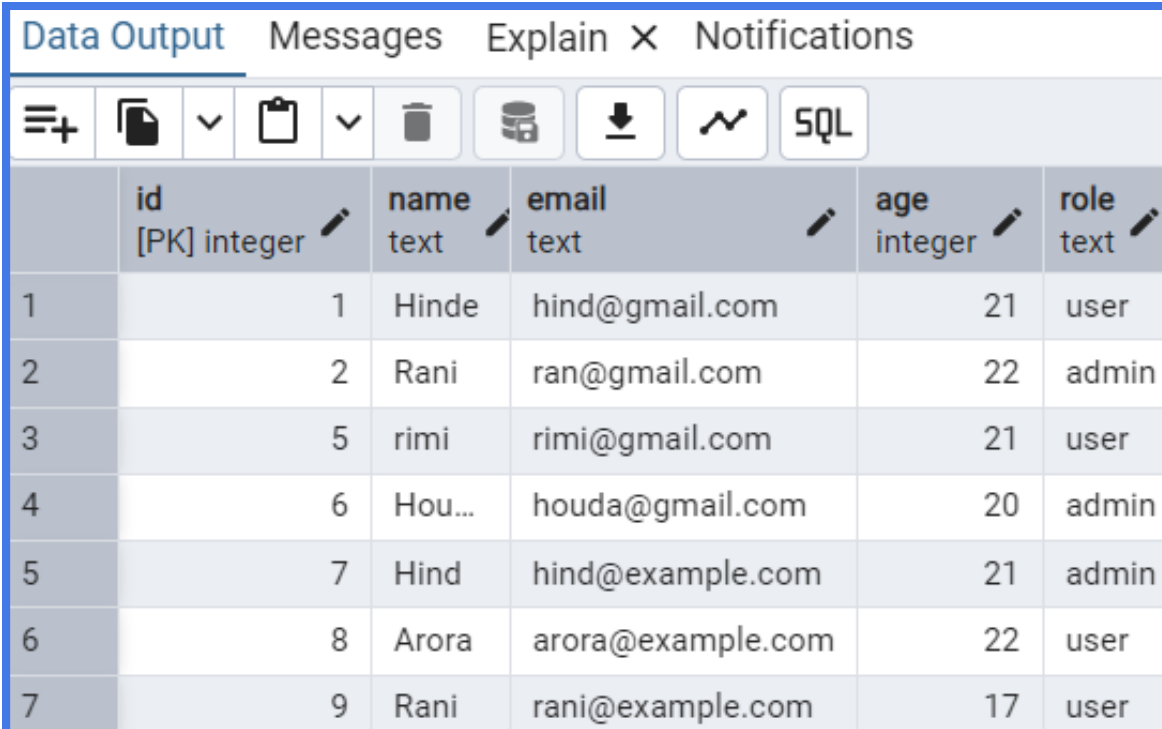
Voici un aperçu des requêtes Axios utilisées :

- `axios.get("/users")` pour récupérer la liste
- `axios.post("/users", data)` pour ajouter un utilisateur
- `axios.put("/users/:id", data)` pour modifier un utilisateur
- `axios.delete("/users/:id")` pour supprimer un utilisateur

2.4 Tests des API avec Postman

Avant d'intégrer le frontend, chaque route de l'API a été testée avec Postman afin de valider le bon fonctionnement du backend. Ces tests sont documentés à l'aide de captures d'écran:

- **GET /users - Liste de tous les utilisateurs**



	id [PK] integer	name text	email text	age integer	role text
1	1	Hinde	hind@gmail.com	21	user
2	2	Rani	ran@gmail.com	22	admin
3	5	rimi	rimi@gmail.com	21	user
4	6	Hou...	houda@gmail.com	20	admin
5	7	Hind	hind@example.com	21	admin
6	8	Arora	arora@example.com	22	user
7	9	Rani	rani@example.com	17	user

Figure 6 - État Initial de la Base de Données

2.4 Tests des API avec Postman

- GET /users – Liste de tous les utilisateurs

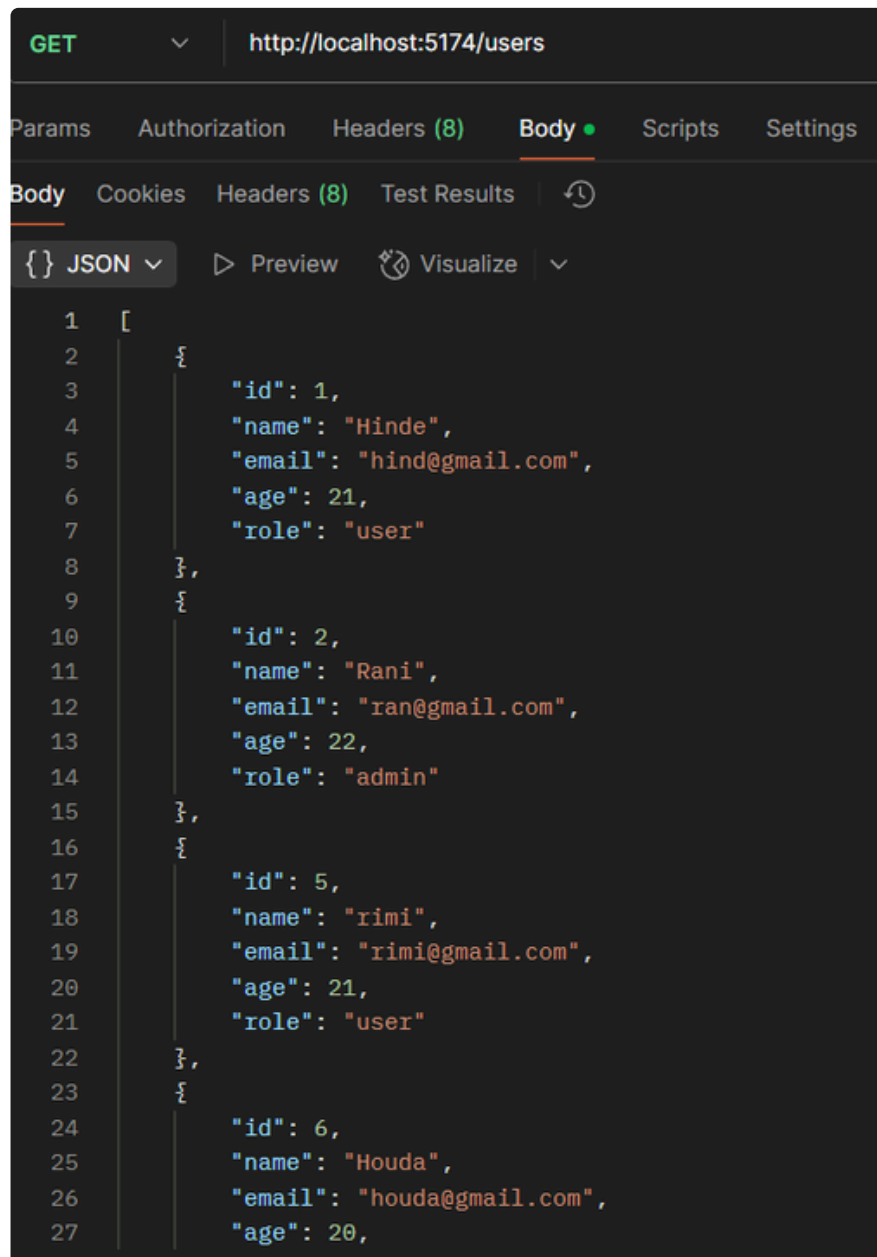


Figure 7 - Requête GET /users dans Postman

2.4 Tests des API avec Postman

- **POST /users – Ajout d'un nouvel utilisateur**

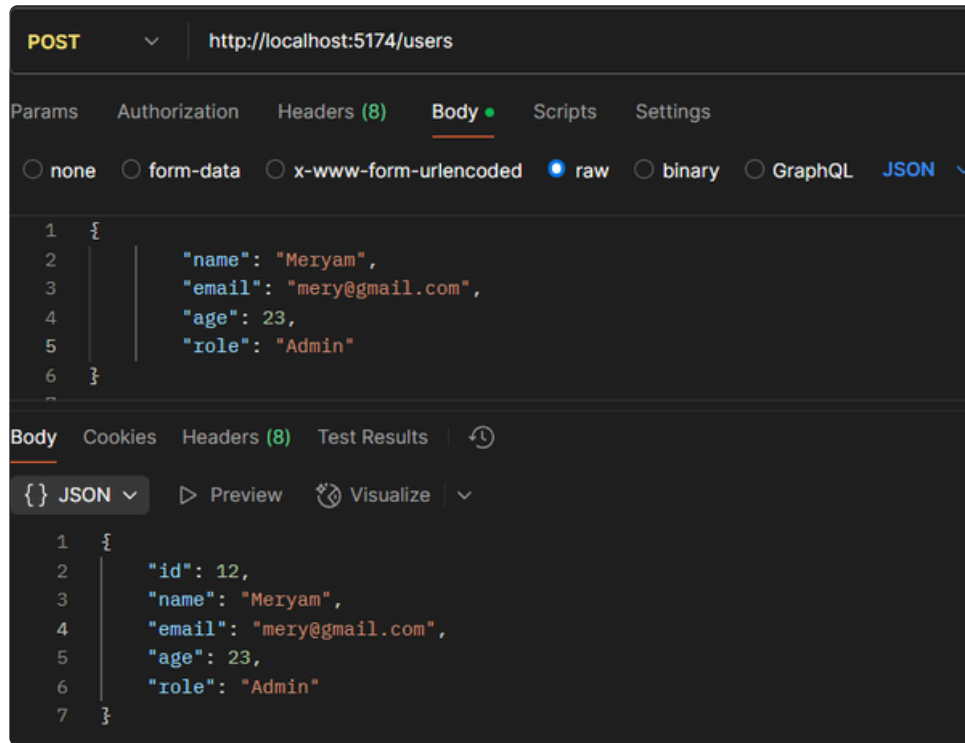


Figure 8 -Requête POST /users ajoutant "Meryam"

	id [PK] integer	name text	email text	age integer	role text
1	1	Hinde	hind@gmail.com	21	user
2	2	Rani	ran@gmail.com	22	admin
3	5	rimi	rimi@gmail.com	21	user
4	6	Houda	houda@gmail.com	20	admin
5	7	Hind	hind@example.com	21	admin
6	8	Arora	arora@example.com	22	user
7	9	Rani	rani@example.com	17	user
8	12	Meryam	mery@gmail.com	23	Admin

Figure 9 -Ajoute de Meryam sur la base de données

2.4 Tests des API avec Postman

- **PUT /users/:id – Mise à jour d'un utilisateur**

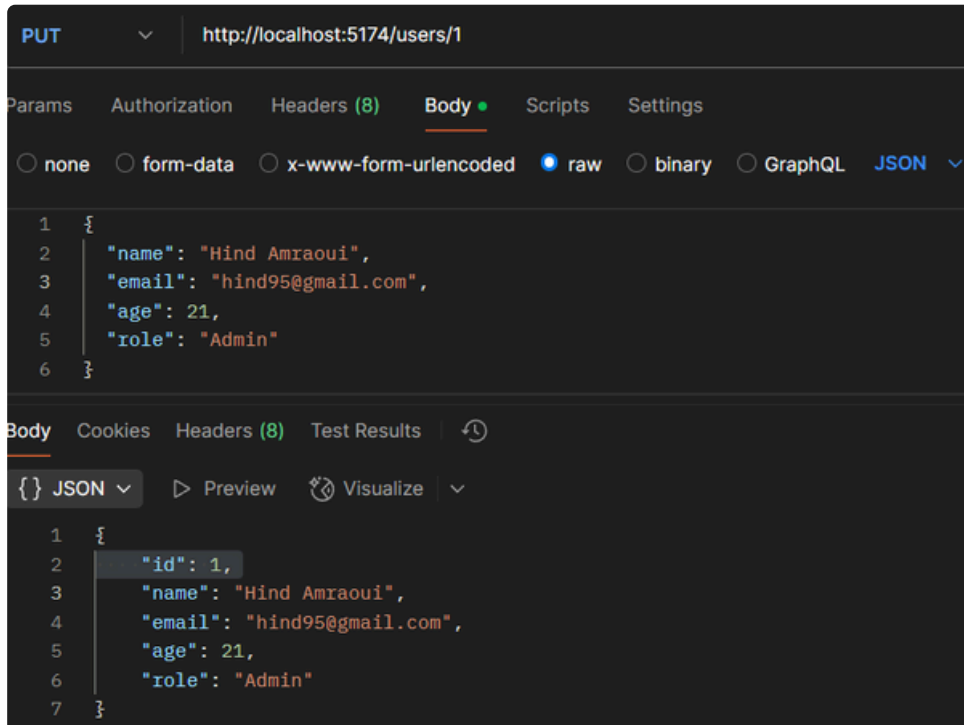


Figure 10 -Requête PUT /users/1 modifiant "Hinde" en "Hind Amraoui"

- **DELETE /users/:id – Suppression d'un utilisateur**

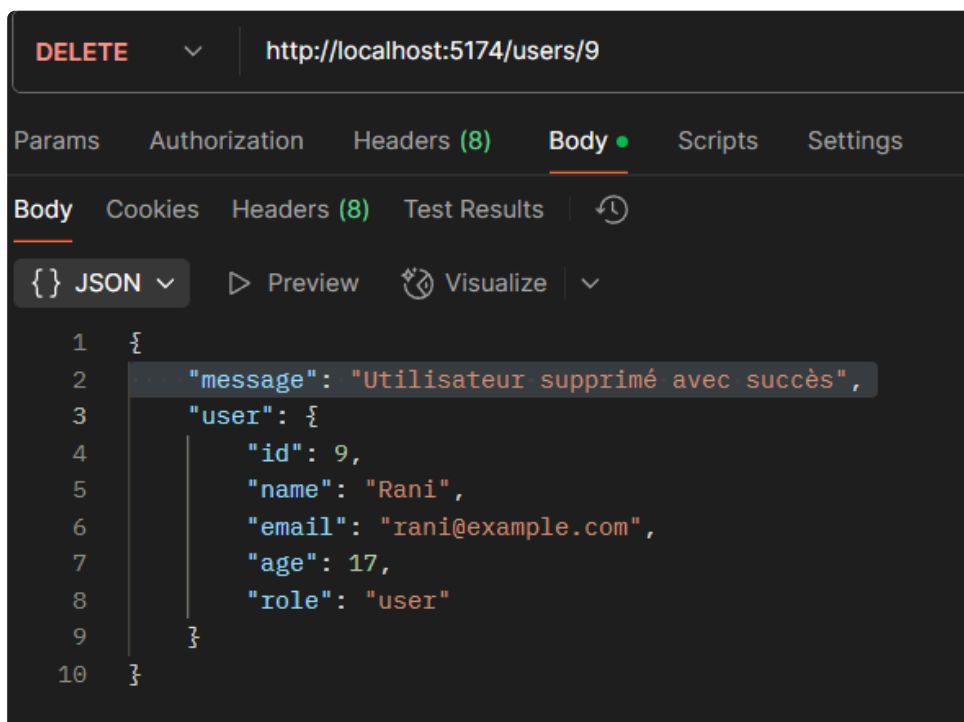


Figure 11 - Requête DELETE /users/9

2.4 Tests des API avec Postman

Vérification via pgAdmin :

Après chaque opération POST, PUT ou DELETE, la base de données a été vérifiée dans pgAdmin afin de s'assurer que les changements étaient bien pris en compte.

	id [PK] integer	name text	email text	age integer	role text
1	2	Rani	ran@gmail.com	22	admin
2	5	rimi	rimi@gmail.com	21	user
3	6	Houda	houda@gmail.com	20	admin
4	8	Arora	arora@example.com	22	user
5	12	Meryam	mery@gmail.com	23	Admin
6	1	Hind Amraoui	hind95@gmail.com	21	Admin

Figure 12 -État final de la base de données après les opérations CRUD

3. Dockerisation de l'application

La dockerisation permet de rendre l'application portable, isolée et facilement déployable dans n'importe quel environnement. Pour cela, trois éléments principaux ont été mis en place : un Dockerfile pour le backend, un Dockerfile pour le frontend, et un fichier docker-compose.yml qui orchestre tous les services.

3.1 Dockerisation du backend

Le backend, développé avec Express.js, a été conteneurisé à l'aide d'un Dockerfile multi-étapes. Ce fichier installe les dépendances, copie les fichiers nécessaires et expose le port utilisé par le serveur :

```
PS C:\Users\dell\user-management-app> docker build -t mon-backend ./backend
[+] Building 10.5s (13/13) FINISHED
-> [internal] load build definition from Dockerfile
-> [internal] load metadata for docker.io/library/node:18-alpine
-> [auth] library/node:pull token for registry-1.docker.io
-> [internal] load .dockerignore
-> [internal] load build context
-> [build 1/6] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e
-> [internal] load build context
-> [build 2/6] WORKDIR /app
-> [build 3/6] COPY package*.json ./
-> [build 4/6] RUN npm install
-> [build 5/6] COPY . .
-> [build 6/6] RUN npm install dotenv
-> [stage-1 3/3] COPY --from=build /app .
-> exporting image
-> exporting layers
-> writing image sha256:f34b3cd334798618f74f4a32c82ddc4a41ad2f4057703fba24c2694aac2c13c4
-> naming to docker.io/library/mon-backend
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/islt8Sezeypasusqelgav211
PS C:\Users\dell\user-management-app>
```

Figure 13 -Construction de l'image Docker pour le backend Node.js/Express

3. Dockerisation de l'application

3.2 Dockerisation du frontend

Le frontend, développé avec React.js et Vite, est construit puis servi via un serveur Nginx dans une image multi-étapes :

```
PS C:\Users\dell\user-management-app> docker build -t mon-frontend ./frontend
[+] Building 1.2s (15/15) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 678B
=> [internal] load metadata for docker.io/library/node:18-alpine
=> [internal] load metadata for docker.io/library/nginx:stable-alpine
=> [internal] load .dockerignore
=> => transferring context: 95B
=> [build 1/6] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e
=> [production 1/3] FROM docker.io/library/nginx:stable-alpine@sha256:d2c11a1e63f280585d8225996fd666436277a54e8c0ba728fa9afff28f075bd7
=> [internal] load build context
=> => transferring context: 542B
=> CACHED [build 2/6] WORKDIR /app
=> CACHED [build 3/6] COPY package*.json ./
=> CACHED [build 4/6] RUN npm install
=> CACHED [build 5/6] COPY . .
=> CACHED [build 6/6] RUN npm run build
=> CACHED [production 2/3] COPY --from=build /app/dist /usr/share/nginx/html
=> [production 3/3] COPY nginx.conf /etc/nginx/conf.d/default.conf
=> exporting to image
=> => exporting layers
=> => writing image sha256:9ca2365b69617599ae51829cbd706e2614a5c1ff2c7658d7806af43f94d1d889
=> => naming to docker.io/library/mon-frontend
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/z6npt6owtrb4viltods4ml149
PS C:\Users\dell\user-management-app>
```

Figure 14 - Construction de l'image Docker pour le frontend React

3.3 Orchestration avec Docker Compose

Le fichier docker-compose.yml permet de démarrer l'ensemble de l'application (frontend, backend, base de données PostgreSQL) avec une seule commande : **docker-compose up --build**

```
=> CACHED [frontend build 4/6] RUN npm install
=> CACHED [frontend build 5/6] COPY . .
=> CACHED [frontend build 6/6] RUN npm run build
=> CACHED [frontend production 2/3] COPY --from=build /app/dist /usr/share/nginx/html
=> CACHED [frontend production 3/3] COPY nginx.conf /etc/nginx/conf.d/default.conf
=> [frontend] exporting to image
=> => exporting layers
=> => writing image sha256:0a5258877138dcebc7fccb0bd0245d532434b5732a839e3a0708f99c15ee5093
=> => naming to docker.io/library/user-management-app-frontend
=> [frontend] resolving provenance for metadata file
[+] Running 7/7
  ✔ backend                               Built
  ✔ frontend                               Built
  ✔ Network user-management-app_app-network Created
  ✔ Volume "user-management-app_user-management-app_pgdata" Created
  ✔ Container postgres_container           Created
  ✔ Container user-management-app-backend-1 Created
  ✔ Container user-management-app-frontend-1 Created
Attaching to postgres_container, backend-1, frontend-1
postgres_container | The files belonging to this database system will be owned by user "postgres".
postgres_container | This user must also own the server process.
postgres_container |
postgres_container | The database cluster will be initialized with locale "en_US.utf8".
postgres_container | The default database encoding has accordingly been set to "UTF8".
postgres_container | The default text search configuration will be set to "english".
postgres_container |
postgres_container | Data page checksums are disabled.
postgres_container |
postgres_container | fixing permissions on existing directory /var/lib/postgresql/data ... ok
postgres_container | creating subdirectories ... ok
postgres_container | selecting dynamic shared memory implementation ... posix
postgres_container | selecting default "max_connections" ... 100
postgres_container | selecting default "shared_buffers" ... 128MB
postgres_container | selecting default time zone ... Etc/UTC
postgres_container | creating configuration files ... ok
```

Figure 15 - Déploiement complet de l'application avec Docker Compose

4. Mise en œuvre des tests

4.1 Implémentation des tests:

Dans cette partie, des tests unitaires et d'intégration ont été implémentés pour vérifier le bon fonctionnement des différentes parties de l'application backend.

Tests unitaires avec Mocha/Chai

Les tests unitaires sont utilisés pour vérifier qu'une fonction ou un composant individuel fonctionne correctement de manière isolée. Un test unitaire a été écrit pour tester la route GET /users du backend, qui récupère la liste des utilisateurs depuis la base de données PostgreSQL. Le test vérifie que la réponse est une liste d'utilisateurs et que le statut de la réponse est 200 (OK).

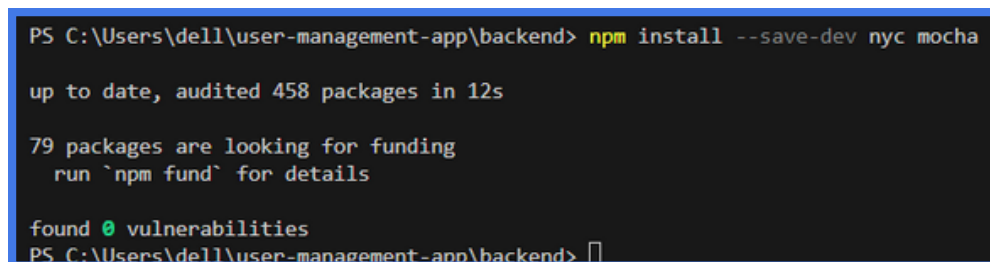
Tests d'intégration

Les tests d'intégration sont utilisés pour tester l'interaction entre plusieurs composants de l'application. Un test d'intégration a été ajouté pour tester la route POST /users, qui permet d'ajouter un nouvel utilisateur à la base de données. Le test vérifie que la création d'un nouvel utilisateur répond avec le bon statut (201) et que l'utilisateur est effectivement ajouté.

4.2 Configuration de la couverture de code:

Pour s'assurer que les tests couvrent suffisamment de code, la couverture de code a été configurée à l'aide de l'outil NYC. NYC génère un rapport détaillé indiquant les parties du code qui sont couvertes par les tests et celles qui ne le sont pas. Cela permet de garantir que les fonctionnalités clés du backend sont bien testées.

Installation de NYC



```
PS C:\Users\dell\user-management-app\backend> npm install --save-dev nyc mocha
up to date, audited 458 packages in 12s

79 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\dell\user-management-app\backend>
```

Figure 16 - Installation des dépendances de test avec npm

4. Mise en œuvre des tests

Configuration du script de test

Dans le fichier package.json, j'ai configuré un script de test pour lancer Mocha avec NYC afin de générer un rapport de couverture :

```
    > Debug
    "scripts": {
      "dev": "nodemon server.js",
      "test": "nyc mocha"
    },
```

Figure 17 - Configuration des scripts npm pour le débogage et les tests.

4.3 Lancer les tests localement:

Une fois les tests écrits et la couverture de code configurée, les tests peuvent être lancés localement avec la commande suivante :

```
Serveur backend sur http://localhost:5174
Connecté à PostgreSQL
Table "users" créée ou déjà existante.
PS C:\Users\dell\user-management-app\backend> npm test

> backend@1.0.0 test
> mocha

Serveur backend sur http://localhost:5174

Test de l'API Users
  1) GET /users devrait retourner un tableau

0 passing (16ms)
1 failing

1) Test de l'API Users
   GET /users devrait retourner un tableau:
    TypeError: chai.request is not a function
    at Context.<anonymous> (file:///C:/Users/dell/user-management-app/backend/test/users.test.js:10:10)
    at processImmediate (node:internal/timers:466:21)

Connecté à PostgreSQL
Table "users" créée ou déjà existante.
```

Figure 18 -Erreur de test API Users – Échec de la requête GET /users.

4. Mise en œuvre des tests

- Le test de l'API Users échoue lors de la requête GET /users.
- L'erreur indique : `TypeError: chal.request is not a function`.
- Le test vérifiait si la route retourne un tableau, mais l'appel API n'a pas pu s'exécuter correctement.

"J'ai essayé à plusieurs reprises de résoudre ce problème en testant différentes solutions, mais sans succès. Parmi les améliorations que je prévois de mettre en œuvre :

- Corriger l'erreur liée à `chal.request` en m'assurant que les dépendances et les imports sont corrects.
- Approfondir ma compréhension de Chai et Mocha pour éviter des erreurs similaires à l'avenir.
- Ajouter des tests plus robustes pour couvrir d'autres endpoints de l'API."

4.4 Documentation de la procédure de test:

Pour exécuter les tests localement et assurer que l'application est bien testée : Nous devons installer les dépendances : Exécutez une commande `npm install --save-dev mocha chai supertest nyc` pour installer les dépendances nécessaires aux tests :

```
PS C:\Users\dell\user-management-app\backend> npm install --save-dev mocha chai supertest nyc

up to date, audited 458 packages in 4s

79 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 19 -Installation des dépendances de test avec npm (Mocha, Chai, Supertest, NYC).

5. Intégration Continue avec GitHub Actions

Cette section présente la mise en place d'un pipeline CI (Intégration Continue) pour automatiser les tests du backend lors de chaque push ou pull request. L'objectif est de garantir que les modifications du code ne cassent pas les fonctionnalités existantes.

5.1 Objectif du pipeline CI/CD

L'objectif principal est de :

- Vérifier automatiquement le bon fonctionnement du backend à chaque mise à jour du code.
- Exécuter les tests unitaires et d'intégration définis avec Mocha/Chai.
- Assurer une meilleure qualité de code et un retour rapide en cas d'erreur.

5.2 Fichier de configuration .yml

Le pipeline est défini dans un fichier YAML situé à l'emplacement suivant :

```
|—— .github/  
|   |—— workflows/  
|       |—— ci.yml
```

Avec le fichier **ci.yml** déclenche l'exécution du job backend-tests à chaque push ou pull request.

5.3 Étapes automatisées (build, test, vérifications)

Le pipeline se compose des étapes suivantes :

1. Checkout du dépôt : Récupère le code source depuis GitHub.
2. Configuration de l'environnement Node.js : Installe la version Node.js requise.
3. Installation des dépendances du backend : Lance npm install dans le dossier backend.
4. Exécution des tests : Lance les tests avec la commande npm test.
5. Build Docker image.
6. Push de l'image vers Docker Hub .
7. Déploiement sur un serveur via SSH.

Chaque étape est automatisée et exécutée dans un environnement propre (ubuntu-latest). Le retour d'exécution est visible directement dans l'interface GitHub

5. Intégration Continue avec GitHub Actions

Lors de l'automatisation du processus CI/CD via GitHub Actions, plusieurs étapes ont été mises en place, notamment : la récupération du dépôt, l'installation des dépendances backend, l'exécution des tests, la création de l'image Docker, et sa publication sur Docker Hub.

Cependant, l'étape d'exécution des tests backend a échoué.



Figure 20 - Erreur d'exécution des mocha (Permission denied).

Cela signifie que la commande mocha n'a pas pu être exécutée en raison d'un problème de permission. Malgré l'installation des dépendances via npm install, l'exécutable mocha ne semble pas être accessible ou exécutable dans le contexte du workflow GitHub Actions.

Cette erreur vient renforcer la nécessité d'une amélioration mentionnée dans la partie précédente (4. Mise en œuvre des tests). Il est donc prévu de corriger ce problème de permissions et de mieux configurer l'environnement de test dans le pipeline CI/CD.

6. Déploiement de l'application

Déploiement du frontend avec Vercel

Pour le déploiement du frontend, j'ai utilisé la plateforme Vercel, qui permet de déployer facilement les applications React.js à partir d'un dépôt GitHub.

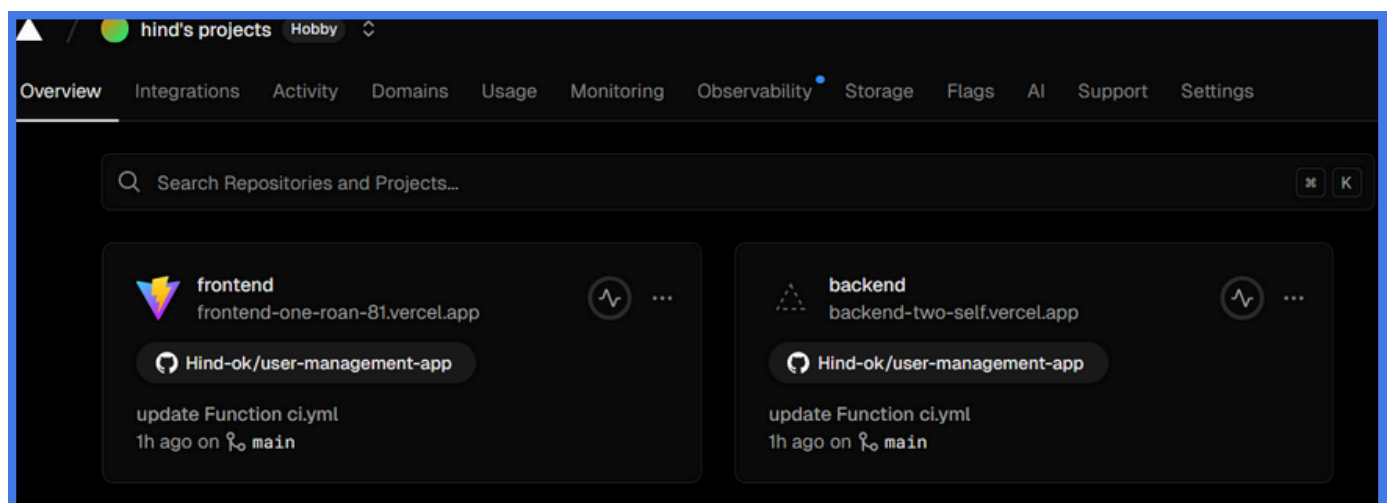
- Le déploiement se fait automatiquement à chaque push sur la branche principale.
- L'interface est intuitive et propose un hébergement gratuit pour les projets frontend.
- L'URL publique est générée automatiquement par Vercel, ce qui facilite le partage de l'application.
- <https://frontend-one-roan-81.vercel.app/>

Cependant, il est important de noter que :

Vercel ne prend pas en charge le déploiement des applications backend Express.js.

- <https://backend-two-self.vercel.app/>

Seul le frontend React.js peut y être hébergé gratuitement. Le backend nécessite une autre solution d'hébergement comme un serveur VPS.



- **Figure 21 - Vue d'ensemble des dépôts frontend et backend (user-management-app)**

7. CONCLUSION

7.1 Bilan du projet

Ce projet full-stack a été une expérience riche et formatrice, combinant des technologies modernes telles qu'Express.js, React.js, PostgreSQL, Docker, et GitHub Actions. De la création d'un backend robuste à l'élaboration d'un frontend interactif, chaque étape a permis de mettre en œuvre les meilleures pratiques de développement. La dockerisation a assuré la portabilité de l'application, tandis que les tests avec Mocha/Chai ont consolidé la fiabilité du système. Enfin, l'intégration continue via GitHub Actions a introduit une logique d'automatisation précieuse dans le cycle de vie du développement.

Ce projet m'a permis de développer non seulement des compétences techniques, mais aussi une capacité à analyser, corriger et anticiper les problèmes d'intégration et de déploiement.

7. CONCLUSION

7.2 Améliorations futures possibles

Pour rendre cette application encore plus fiable, complète et professionnelle, plusieurs pistes d'amélioration peuvent être envisagées :

- *Correction des problèmes liés aux tests : Résoudre les erreurs existantes dans les tests Mocha/Chai afin de garantir un environnement de test stable, fiable, et automatisé en continu avec GitHub Actions.*
- *Ajout d'une authentification sécurisée : Mettre en place un système d'authentification avec JWT pour sécuriser l'accès aux fonctionnalités de gestion.*
- *Dashboard d'administration : Développer une interface d'administration avec des graphiques et statistiques pour un meilleur suivi des utilisateurs et des actions.*

Fin du Rapport

