

Ecole supérieure de technologie Salé

# Rapport Technique: Mise en place d'une application de gestion d'utilisateurs

IAWM 2024/2025

EL KHRAIBI Jihane

Encadré par: Mohamed El Farouki

## Table des matières

<b>Rapport Technique : Mise en place d'une application de gestion d'utilisateurs .....</b>	<b>0</b>
<b>1. Présentation générale du projet .....</b>	<b>2</b>
<b>2. Étapes de mise en place .....</b>	<b>2</b>
Backend (Node.js/Express) .....	2
Frontend .....	5
<b>3. Base de données SQLite .....</b>	<b>6</b>
<b>4. Dockerisation.....</b>	<b>8</b>
<b>5. GitHub Actions : Pipeline CI/CD .....</b>	<b>12</b>
<b>6. Tests et validations .....</b>	<b>15</b>
<b>7. Difficultés rencontrées et solutions .....</b>	<b>17</b>
Problème 1 : Configuration des tests avec chai-http .....	17
Problème 2 : Gestion des modules CommonJS vs ESM.....	17
Problème 3 : Persistance des données dans Docker .....	17
<b>8. Conclusion et axes d'amélioration .....</b>	<b>17</b>

# 1. Présentation générale du projet

Ce projet consiste en la création d'une application web de gestion d'utilisateurs avec une architecture moderne et des pratiques DevOps. L'application permet de créer, lire, mettre à jour et supprimer (CRUD) des profils d'utilisateurs, comprenant des informations telles que le nom, prénom, âge, profession et email.

## Objectifs techniques :

- Développer une API RESTful avec Node.js et Express
- Mettre en place une base de données légère et performante
- Implémenter des tests automatisés pour valider le code
- Conteneuriser l'application avec Docker
- Établir un pipeline CI/CD complet avec GitHub Actions

L'application se compose d'un backend RESTful, développé en JavaScript avec Node.js et Express, servant d'interface pour manipuler les données utilisateurs stockées dans une base de données SQLite. Cette architecture permet une séparation claire des responsabilités entre la persistance des données et la logique métier.

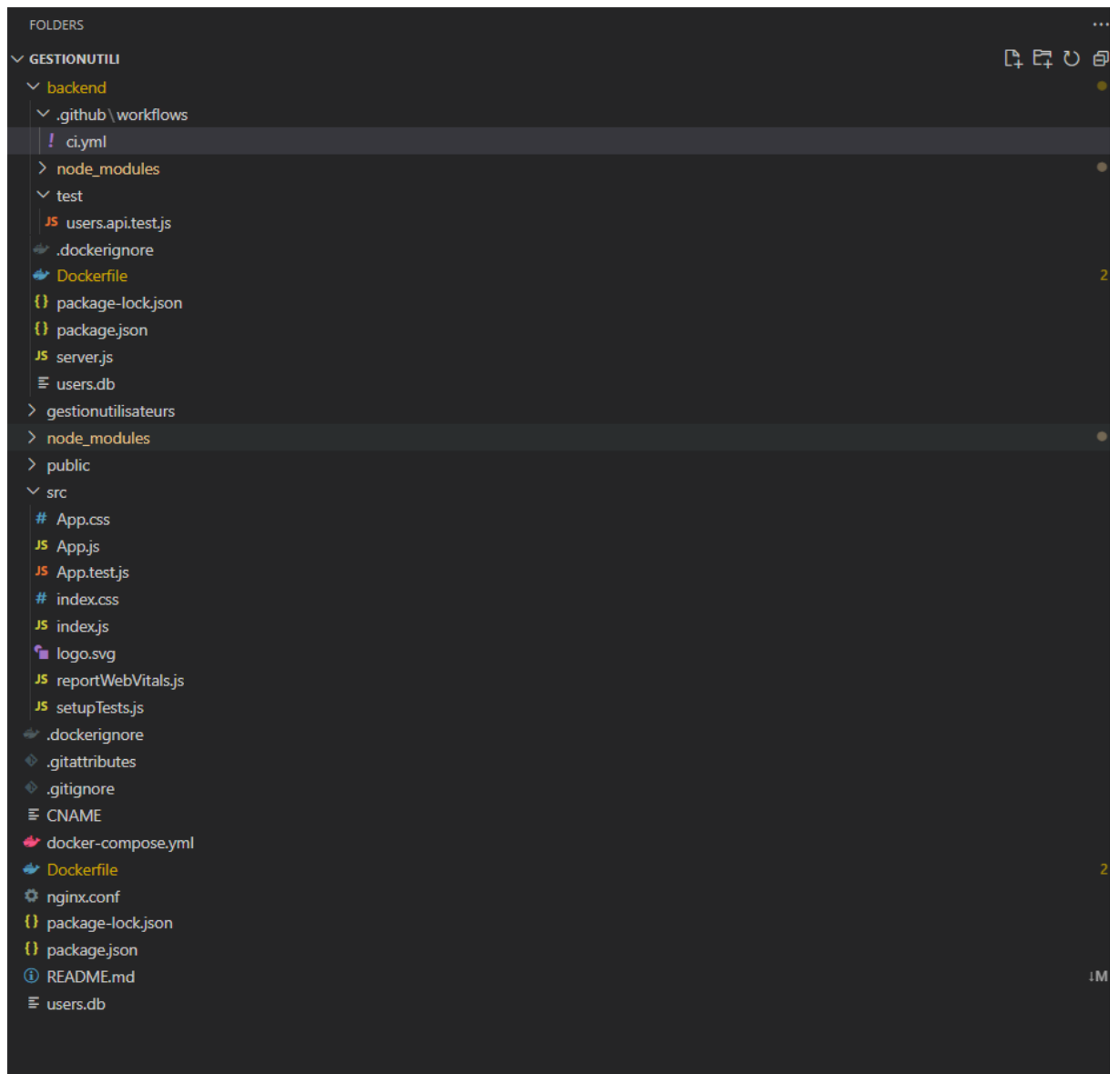
## 2. Étapes de mise en place

### Backend (Node.js/Express)

Le backend est développé avec les technologies suivantes :

- **Node.js** comme runtime JavaScript
- **Express** comme framework web
- **SQLite3** pour la base de données
- **Mocha** pour les tests automatisés

#### 1. Structure initiale du projet



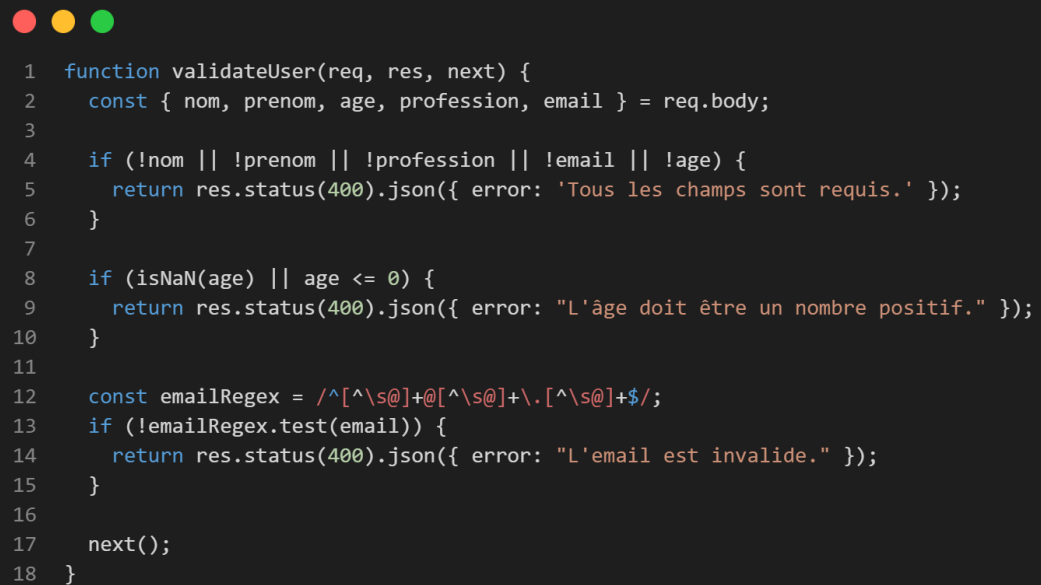
## 2. Implémentation des endpoints REST

```

1 // Routes API
2 app.get('/', (req, res) => res.send('🚀 API Running'));
3
4 // Obtenir tous les utilisateurs
5 app.get('/users', (req, res) => {
6   db.all("SELECT * FROM users", [], (err, rows) => {
7     if (err) return res.status(500).json({ error: err.message });
8     res.json(rows);
9   });
10 });
11
12 // Obtenir un utilisateur par ID
13 app.get('/users/:id', (req, res) => {
14   db.get("SELECT * FROM users WHERE id = ?", [req.params.id], (err, row) => {
15     if (err) return res.status(500).json({ error: err.message });
16     if (!row) return res.status(404).json({ error: "Utilisateur non trouvé." });
17     res.json(row);
18   });
19 });
20
21 // Ajouter un utilisateur
22 app.post('/users', validateUser, (req, res) => {
23   const { nom, prenom, age, profession, email } = req.body;
24   console.log('📧 Données reçues :', req.body);
25
26   db.run(`
27     INSERT INTO users (nom, prenom, age, profession, email)
28     VALUES (?, ?, ?, ?, ?)
29   `, [nom, prenom, age, profession, email], function (err) {
30     if (err) {
31       if (err.message.includes("UNIQUE constraint failed")) {
32         return res.status(400).json({ error: "Cet email existe déjà." });
33       }
34       return res.status(500).json({ error: err.message });
35     }
36     console.log('✅ Utilisateur ajouté avec succès. ID : ${this.lastID}');
37     res.json({ id: this.lastID, nom, prenom, age, profession, email });
38   });
39 });
40
41 // Modifier un utilisateur
42 app.put('/users/:id', validateUser, (req, res) => {
43   const { nom, prenom, age, profession, email } = req.body;
44   const sql = `UPDATE users SET nom = ?, prenom = ?, age = ?, profession = ?, email = ? WHERE id = ?`;
45
46   db.run(sql, [nom, prenom, age, profession, email, req.params.id], function (err) {
47     if (err) {
48       if (err.message.includes("UNIQUE constraint failed")) {
49         return res.status(400).json({ error: "Cet email existe déjà." });
50       }
51       return res.status(500).json({ error: err.message });
52     }
53
54     if (this.changes === 0) {
55       return res.status(404).json({ error: "Utilisateur non trouvé." });
56     }
57
58     res.json({ id: req.params.id, nom, prenom, age, profession, email });
59   });
60 });
61

```

3. **Middleware de validation** Un middleware a été implémenté pour valider les données entrantes :



```

1  function validateUser(req, res, next) {
2      const { nom, prenom, age, profession, email } = req.body;
3
4      if (!nom || !prenom || !profession || !email || !age) {
5          return res.status(400).json({ error: 'Tous les champs sont requis.' });
6      }
7
8      if (isNaN(age) || age <= 0) {
9          return res.status(400).json({ error: "L'âge doit être un nombre positif." });
10     }
11
12     const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
13     if (!emailRegex.test(email)) {
14         return res.status(400).json({ error: "L'email est invalide." });
15     }
16
17     next();
18 }

```

4. **Gestion des erreurs** Des réponses HTTP appropriées sont renvoyées en cas d'erreur :
- 400 : Requête incorrecte (données invalides)
  - 404 : Ressource non trouvée
  - 500 : Erreur serveur

## Frontend

partie frontend a été conçue pour consommer l'API RESTful via des requêtes HTTP et présenter une interface utilisateur intuitive permettant :

- L'affichage de la liste des utilisateurs
- La création de nouveaux utilisateurs via un formulaire
- La modification des informations utilisateur
- La suppression d'utilisateurs

### Liste des Utilisateurs

+ Ajouter Utilisateur

Modifier

Supprimer

jihane jihane - 15 - etudiant - jihane@gmail.com

### 3. Base de données SQLite

SQLite a été choisi pour ce projet pour plusieurs raisons :

- Légèreté et performance
- Absence de configuration serveur nécessaire
- Intégration facile avec Node.js
- Adapté pour des applications de petite à moyenne envergure



```
1 // Connexion à SQLite
2 const db = new sqlite3.Database('./users.db', (err) => {
3   if (err) return console.error(err.message);
4   console.log('📦 Connected to SQLite database.');
```

```
5 });
6
7 // Création de la table si elle n'existe pas
8 db.run(`
9   CREATE TABLE IF NOT EXISTS users (
10     id INTEGER PRIMARY KEY AUTOINCREMENT,
11     nom TEXT NOT NULL,
12     prenom TEXT NOT NULL,
13     age INTEGER CHECK(age > 0),
14     profession TEXT NOT NULL,
15     email TEXT UNIQUE NOT NULL CHECK(email LIKE '%@%')
16   )
17 `);
18
```

### Caractéristiques notables :

- Contrainte d'unicité sur l'email pour éviter les doublons
- Validation de l'âge au niveau de la base de données (doit être positif)
- Vérification basique du format d'email

Cette structure garantit l'intégrité des données et complète les validations effectuées au niveau de l'API.



```

PS D:\gestionutili> npm start

> gestionutili@0.1.0 start
> concurrently "react-scripts start" "npm run start:backend"

[1]
[1] > gestionutili@0.1.0 start:backend
[1] > node backend/server.js
[1]
[1] ✓ Server running at: http://localhost:5000
[1] 📦 Connected to SQLite database.
[0] (node:18388) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWarning:
[0] (Use `node --trace-deprecation ...` to show where the warning was created)
[0] (node:18388) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] DeprecationWarning
[0] Starting the development server...
[0]
[0] Compiled successfully!
[0]
[0] You can now view gestionutili in the browser.
[0]
[0]   Local:            http://localhost:3000
[0]   On Your Network:  http://192.168.56.1:3000
[0]
[0] Note that the development build is not optimized.
[0] To create a production build, use npm run build.
[0]
[0] webpack compiled successfully
[1] 📧 Données reçues : {
[1]   nom: 'jihane',
[1]   prenom: 'jihane',
[1]   age: '15',
[1]   profession: 'etudiant',
[1]   email: 'jihane@gmail.com'
[1] }
[1] ✓ Utilisateur ajouté avec succès. ID : 1

```

## 4. Dockerisation

La conteneurisation de l'application facilite le déploiement et garantit la cohérence entre les environnements de développement, de test et de production.

**Dockerfile (frontend):**



```
1  # Stage 1: Build stage
2  FROM node:20-alpine AS build
3
4  WORKDIR /app
5
6  # Set environment variables for Node.js 20 compatibility
7  ENV NODE_ENV=production
8  ENV NODE_OPTIONS="--openssl-legacy-provider"
9
10 # Copy package files first
11 COPY package*.json ./
12
13 # Install dependencies
14 RUN npm install --no-audit --no-fund --legacy-peer-deps
15
16 # Copy the rest of the application code
17 COPY . .
18
19 # Build the application
20 RUN npm run build
21
22 # Stage 2: Production stage
23 FROM nginx:1.25.2-alpine
24
25 # Copy built files from the build stage
26 COPY --from=build /app/build /usr/share/nginx/html
27 COPY nginx.conf /etc/nginx/conf.d/default.conf
28
29 EXPOSE 80
30
31 CMD ["nginx", "-g", "daemon off;"]
```

**Dockerfile (backend):**



```
1  # Stage 1: Build stage
2  FROM node:20-alpine AS build
3
4  WORKDIR /app
5
6  # Set npm to run in production mode and avoid unnecessary output
7  ENV NODE_ENV=production
8  ENV NPM_CONFIG_LOGLEVEL=error
9
10 # Copy package files first
11 COPY package*.json ./
12
13 # Use npm ci for faster, more reliable installation
14 RUN npm ci --no-audit --no-fund
15
16
17
18 # Copy the rest of the application code
19 COPY . .
20
21 # Stage 2: Production stage
22 FROM node:20-alpine
23
24 WORKDIR /app
25
26 # Set production environment
27 ENV NODE_ENV=production
28
29 # Copy from build stage
30 COPY --from=build /app /app
31
32 # Add health check for the API
33 HEALTHCHECK --interval=30s --timeout=5s --start-period=15s --retries=3 \
34   CMD wget -q --spider http://localhost:5000/health || exit 1
35
36 EXPOSE 5000
37
38 CMD ["node", "server.js"]
```

**Docker Compose :**

```
1  services:
2    backend:
3      build:
4        context: ./backend
5        dockerfile: Dockerfile
6      container_name: api-backend
7      volumes:
8        - ./backend/users.db:/app/users.db
9      ports:
10       - "5000:5000"
11      networks:
12       - app-network
13      restart: unless-stopped
14      healthcheck:
15        test: ["CMD", "wget", "-q", "--spider", "http://localhost:5000/health"]
16        interval: 30s
17        timeout: 100s
18        retries: 3
19        start_period: 30s
20
21    frontend:
22      build:
23        context: ./
24        dockerfile: Dockerfile
25      container_name: react-frontend
26      ports:
27       - "80:80"
28      networks:
29       - app-network
30      depends_on:
31        backend:
32          condition: service_healthy
33      restart: unless-stopped
34      healthcheck:
35        test: ["CMD", "wget", "-q", "--spider", "http://localhost"]
36        interval: 30s
37        timeout: 5s
38        retries: 3
39        start_period: 15s
40
41    networks:
42      app-network:
43        driver: bridge
```

### Choix techniques :

1. **Image de base légère** : Node.js Alpine pour réduire la taille de l'image
2. **Multi-stage build** : Installation des dépendances avant copie du code pour optimiser le cache
3. **Volumes pour la persistance** : Montage de la base de données SQLite pour préserver les données

4. **Variables d'environnement** : Configuration via .env pour faciliter le déploiement dans différents environnements

```
PS D:\gestionutili> docker-compose build --parallel
Compose can now delegate builds to bake for better performance.
To do so, set COMPOSE_BAKE=true.
[+] Building 126.8s (27/27) FINISHED
=> [backend internal] load build definition from Dockerfile
=> => transferring dockerfile: 821B
=> [frontend internal] load metadata for docker.io/library/node:20-alpine
=> [backend auth] library/node:pull token for registry-1.docker.io
=> [backend internal] load .dockerignore
=> => transferring context: 97B
=> [backend internal] load build context
=> => transferring context: 192B
=> [frontend build 1/6] FROM docker.io/library/node:20-alpine@sha256:8bda036ddd59ea51a23bc1a1035d3b5c614e72
=> CACHED [frontend build 2/6] WORKDIR /app
=> CACHED [backend build 3/5] COPY package*.json ./
=> CACHED [backend build 4/5] RUN npm ci --no-audit --no-fund
=> CACHED [backend build 5/5] COPY . .
=> CACHED [backend stage-1 3/3] COPY --from=build /app /app
=> [backend] exporting to image
=> => exporting layers
=> => writing image sha256:cbcd7adef1948c912bce021aa128cc321bed35b28b5b0ef01c181d084624d2ff
=> => naming to docker.io/library/gestionutili-backend
=> [backend] resolving provenance for metadata file
=> [frontend internal] load build definition from Dockerfile
=> => transferring dockerfile: 724B
=> [frontend internal] load metadata for docker.io/library/nginx:1.25.2-alpine
=> [frontend auth] library/nginx:pull token for registry-1.docker.io
=> [frontend internal] load .dockerignore
=> => transferring context: 114B
=> CACHED [frontend stage-1 1/3] FROM docker.io/library/nginx:1.25.2-alpine@sha256:7272a6e0f728e95c8641d219
=> [frontend internal] load build context
=> => transferring context: 1.01MB
=> [frontend build 3/6] COPY package*.json ./
=> [frontend build 4/6] RUN npm install --no-audit --no-fund --legacy-peer-deps
=> [frontend build 5/6] COPY . .
=> [frontend build 6/6] RUN npm run build
=> [frontend stage-1 2/3] COPY --from=build /app/build /usr/share/nginx/html
=> [frontend stage-1 3/3] COPY nginx.conf /etc/nginx/conf.d/default.conf
=> [frontend] exporting to image
=> => exporting layers
=> => writing image sha256:edc2c293d451986b919b679b472addf799c0990571b9b94ab4e3e7796180732e
=> => naming to docker.io/library/gestionutili-frontend
=> [frontend] resolving provenance for metadata file
[+] Building 2/2
✓ backend Built
✓ frontend Built
```

## 5. GitHub Actions : Pipeline CI/CD

Un pipeline d'intégration et déploiement continu a été mis en place avec GitHub Actions pour automatiser les tests, le build et le déploiement de l'application.

**Structure du workflow (ci.yml) :**



```
1  name: CI/CD Pipeline
2
3  on:
4    push:
5      branches:
6        - main
7    pull_request:
8
9  jobs:
10   build-and-deploy:
11     runs-on: ubuntu-latest
12
13     env:
14       DOCKER_IMAGE: ${ secrets.DOCKER_USERNAME }}/gestionutil
15
16     steps:
17       # 1. ✅ Checkout du repo
18       - name: Checkout code
19         uses: actions/checkout@v3
20
21       # 2. ✅ Setup Node.js
22       - name: Setup Node.js
23         uses: actions/setup-node@v3
24         with:
25           node-version: 18
26
27       # 3. ✅ Installation des dépendances
28       - name: Install dependencies
29         run: npm ci
30
31       # 4. ✅ Lancement des tests
32       - name: Run backend tests
33         run: |
34           cd backend
35           npm install
36           npm test
37
38       # 5. ✅ Build Docker image
39       - name: Build Docker image
40         run: docker build -t $DOCKER_IMAGE:latest .
41
42       # 6. ✅ Push image vers Docker Hub
43       - name: Login to Docker Hub
44         uses: docker/login-action@v3
45         with:
46           username: ${ secrets.DOCKER_USERNAME }
47           password: ${ secrets.DOCKER_PASSWORD }
48
49       - name: Push Docker image
50         run: docker push $DOCKER_IMAGE:latest
51
```

## Explications étape par étape :

### 1. Job build-and-test

- **Checkout du code** : Récupère le code source du repository
- **Setup Node.js** : Configure l'environnement Node.js v18
- **Installation des dépendances** : Exécute `npm ci` pour installer les dépendances exactes du `package-lock.json`
- **Lancement des tests** : Exécute la suite de tests avec Mocha

### 2. Job build-and-push (exécuté uniquement après succès des tests)

- **Login Docker Hub** : Authentification avec les credentials stockés dans les secrets GitHub
- **Setup Docker Buildx** : Prépare l'environnement pour la construction d'images multi-architectures
- **Build et push** : Construit l'image Docker et la publie sur Docker Hub avec caching pour optimiser les builds futurs

### 3. Job deploy (exécuté uniquement après le push de l'image)

- **Connexion SSH au serveur** : Utilise les credentials stockés dans les secrets GitHub
- **Création du fichier .env** : Génère un fichier de configuration à partir des secrets
- **Pull et redémarrage des conteneurs** : Met à jour l'image et relance l'application avec `docker-compose`

Cette approche garantit que seul le code qui passe les tests est déployé en production, et que le déploiement est automatisé et reproductible.

## 6. Tests et validations

Les tests automatisés jouent un rôle crucial dans la garantie de la qualité du code et permettent d'identifier rapidement les régressions.

### Tests unitaires et d'intégration :

- Tests des validations de données utilisateur
- Tests des endpoints de l'API
- Vérification des contraintes de la base de données

### L'exécution des tests :



```
📦 Connected to SQLite database.

📦 Connected to SQLite database.

API Users
📧 Données reçues : {
  nom: 'Test',
  prenom: 'User',
  age: 30,
  profession: 'Developer',
  email: 'test@example.com'
}
✅ Utilisateur ajouté avec succès. ID : 9
  ✓ should create a user (164ms)
  ✓ should retrieve a user
  ✓ should update a user
  ✓ should delete a user
```

## Capture d'écran des actions GitHub :

```
report-build-status
succeeded 1 minute ago in 4s

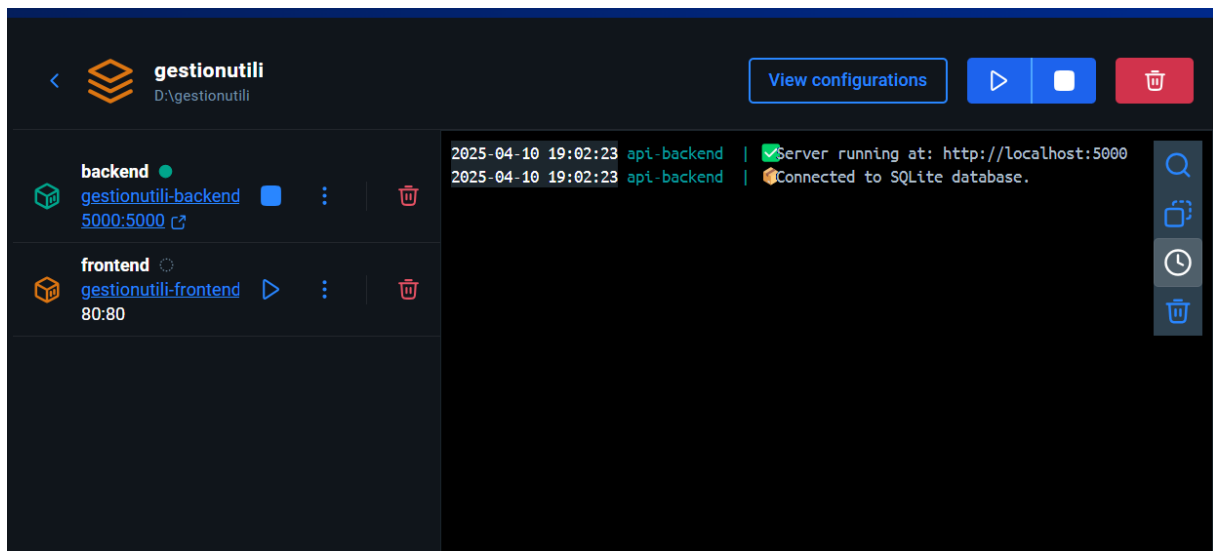
🔍 Search logs

Set up job 0s
1 Current runner version: '2.323.0'
2 ▶ Operating System
6 ▶ Runner Image
11 ▶ Runner Image Provisioner
13 ▶ GITHUB_TOKEN Permissions
17 Secret source: Actions
18 Prepare workflow directory
19 Prepare all required actions
20 Complete job name: report-build-status

Report Build Status 1s
1 ▶ Run gh api -X POST "repos/${GITHUB_REPOSITORY}/pages/telemetry" \
9 {}

Complete job 0s
1 Cleaning up orphan processes
```

## Capture d'écran des conteneurs Docker :



## 7. Difficultés rencontrées et solutions

### Problème 1 : Configuration des tests avec chai-http

**Problème :** Erreur `ERR_PACKAGE_PATH_NOT_EXPORTED` lors de l'exécution des tests. **Solution :** Remplacement de `chai-http` par le module natif `http` de Node.js et création d'un helper personnalisé pour les tests d'API.

### Problème 2 : Gestion des modules CommonJS vs ESM

**Problème :** Incompatibilités entre les imports/exports des différents modules. **Solution :** Standardisation de l'approche en utilisant CommonJS (`module.exports` et `require`) dans l'ensemble du projet et configuration explicite dans `package.json`.

### Problème 3 : Persistance des données dans Docker

**Problème :** Perte des données utilisateur lors du redémarrage des conteneurs. **Solution :** Utilisation d'un volume Docker pour persister la base de données SQLite entre les déploiements.

## 8. Conclusion et axes d'amélioration

Ce projet a permis la mise en place d'une application de gestion d'utilisateurs complète, avec une architecture moderne et des pratiques DevOps robustes. L'utilisation de Node.js, Express et SQLite offre une solution légère et performante, tandis que l'intégration de Docker et GitHub Actions garantit un déploiement fiable et reproductible.

### Axes d'amélioration potentiels :

1. **Migration vers une base de données plus robuste**
  - Envisager PostgreSQL ou MongoDB pour des besoins de scalabilité accrus
  - Implémenter un ORM comme Sequelize ou Prisma pour une gestion plus abstraite des données

2. **Renforcement de la sécurité**

- Ajout d'authentification JWT pour sécuriser l'API
- Mise en place de HTTPS
- Implémentation de rate limiting pour prévenir les abus

3. **Monitoring et logging**

- Intégration d'outils comme Prometheus pour le monitoring
- Centralisation des logs avec ELK Stack ou Graylog

4. **Tests avancés**

- Couverture de code plus complète
- Tests de charge pour valider la performance
- Tests de sécurité automatisés

5. **Documentation API**

- Génération automatique de documentation avec Swagger/OpenAPI

Merci !