TP – API REST Express.js avec base SQLite : gestion des super-héros

Table des matières

Objectif

Développer une API REST en Node.js avec Express.js pour manipuler des données de super-héros, d'abord via un fichier JSON puis via une base SQLite.

I. Version 1 – API avec fichier JSON

1.1. Objectif

Développer une première version rapide de l'API en lisant les données du fichier `SuperHerosComplet.json`.

1.2. Étape 1 – Initialisation

- Créer un dossier `api-superheros`
- `npm init -y`
- `npm install express`

1.3. Étape 2 – Lecture JSON

- Lire le fichier JSON au démarrage avec `fs.promises`
- Stocker les héros en mémoire

1.4. Étape 3 – Routes à implémenter

- `GET /heroes` – tous les héros
- `GET /heroes/:id` – un héros par ID
- `GET /heroes?publisher=Marvel` – filtrer par éditeur
- `GET /heroes/search?q=man` – recherche sur le nom
- `POST /heroes` – ajout en mémoire
- `DELETE /heroes/:id` – suppression en mémoire

1.5. Étape 4 – Tests et compte rendu

- Tests avec Postman / curl
- Présentation des routes
- Justification des choix

II. Version 2 – API complète avec SQLite

2.1. Objectif

Remplacer la version JSON par une persistance réelle avec SQLite et étendre les fonctionnalités.

2.2. Partie 1 – Import initial depuis JSON

1. Placer le fichier `SuperHerosComplet.json` à la racine
2. Adapter le script d'import si nécessaire :

```
const fs = require('fs');
const data = JSON.parse(fs.readFileSync('./SuperHerosComplet.json', 'utf-8'));
const insert = db.prepare(`INSERT INTO heroes (name, publisher, gender, race, power,
alignment, height_cm, weight_kg, createdAt)
VALUES (@name, @publisher, @gender, @race, @power, @alignment, @height_cm, @weight_kg,
@createdAt)`);

const count = db.prepare('SELECT COUNT(*) as total FROM heroes').get();
if (count.total === 0) {
  const now = new Date().toISOString();
  for (const hero of data) {
    insert.run({
      name: hero.name,
      publisher: hero.publisher,
      gender: hero.gender,
      race: hero.race,
      power: hero.power,
      alignment: hero.alignment,
      height_cm: parseInt(hero.height_cm),
      weight_kg: parseInt(hero.weight_kg),
      createdAt: now
    });
  }
  console.log('Données initiales importées.');
} }
```

2.3. Étape 1 – Initialisation du projet

```
npm init -y
npm install express better-sqlite3
```

Créer les fichiers :

- `index.js`
- `database.js`
- `superheros.db` (généré automatiquement)

2.4. Étape 2 – Base SQLite (`database.js`)

```
const Database = require('better-sqlite3');
const db = new Database('superheros.db');

db.exec(`CREATE TABLE IF NOT EXISTS heroes (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  publisher TEXT,
  gender TEXT,
  race TEXT,
  power TEXT,
  alignment TEXT,
  height_cm INTEGER,
  weight_kg INTEGER,
  createdAt TEXT
);`);

module.exports = db;
```

2.5. Étape 3 – Serveur Express (`index.js`)

```
const express = require('express');
const db = require('./database');
const app = express();
app.use(express.json());
```

2.6. Étape 4 – Routes de l'API

GET /heroes

Renvoie tous les héros

```
app.get('/heroes', (req, res) => {
  const heroes = db.prepare('SELECT * FROM heroes').all();
  res.json(heroes);
});
```

a) GET /heroes/:id

Un héros par ID

b) GET /heroes/search?q=bat

Recherche par nom avec LIKE

c) GET /heroes?publisher=DC

Filtrage

d) GET /heroes/sorted?by=height

Tri dynamique (avec précaution sur injection SQL)

e) POST /heroes

Ajout d'un héros (générer createdAt automatiquement)

f) DELETE /heroes/:id

Suppression par ID

2.7. Étape 5 – Gestion des erreurs

- Héros non trouvé : 404
- Données manquantes : 400
- ID invalide ou SQL dangereux : 422

2.8. Étape 6 – Bonus possibles

- Export CSV : `/heroes/export?publisher=DC`
- Tri multi-champs : `/heroes/sorted?by=height,weight`
- Statistiques : `/heroes/stats` (par éditeur, moyennes...)

III. CR. Et notation

3.1. Consignes de rendu – TP API Super-Héros

Chaque étudiant (ou binôme) remettra un **compte rendu structuré** avec les éléments suivants :

1. Introduction

- Objectif du TP
- Présentation des deux versions : version fichier JSON puis version SQLite

2. Version JSON

- Méthode de lecture des données
- Routes implémentées : brève description + extrait de code
- Tests réalisés (captures Postman ou `curl`)
- Limites constatées (ex. : absence de persistance)

3. Version SQLite

- Présentation de la base (`CREATE TABLE`, structure)
- Méthode d'import initial
- Routes implémentées (GET, POST, DELETE, recherche, tri) avec extraits de code
- Gestion des erreurs (404, 400, 422)
- Résultats des tests (captures d'écran ou commandes)

4. Bonus réalisés (si applicable)

- Export CSV
- Tri multichamps
- Statistiques (ex : moyenne de taille, nombre par éditeur)

5. Analyse critique

- Avantages/inconvénients des deux approches
- Difficultés rencontrées et solutions apportées
- Améliorations possibles

6. Conclusion

- Bilan du travail accompli
- Ce que vous avez appris / retenu

3.2. Documents attendus

- Le **lien vers le github :** (`index.js`, `database.js`, `import.js`, etc.)

- Le **fichier de compte rendu** (PDF)
- Le fichier `README.md`

3.3. Évaluation (sur 20 points)

Critère	Points
Version JSON fonctionnelle	3
Connexion SQLite et création de table	3
Routes CRUD opérationnelles	6
Recherches, tris et filtres	3
Gestion des erreurs	2
Propreté du code et documentation	1
Bonus (CSV, stats, tri avancé)	2
Total	**/20**