

TP - Migration de JSON vers SQLite et Gestion Utilisateurs

Du Stockage Fichier vers Base de Données Relationnelle

IUT Nord Franche-Comté Objectif: Migrer une application Express de stockage JSON vers SQLite, puis ajouter un système de gestion utilisateurs avec relation many-to-many

Contexte

Votre patron vient de vous confier une mission critique ! L' application actuelle stocke les produits dans un fichier JSON (`products.json`), mais cette approche pose des problèmes :

- 🐢 Performances dégradées avec beaucoup de données
- 🔒 Risques de corruption du fichier
- 🚫 Pas de relations entre entités
- 📊 Difficile de faire des requêtes complexes

Mission : Migrer vers SQLite pour améliorer la robustesse et ajouter un système d' utilisateurs qui peuvent acheter des produits.

Objectifs d' Apprentissage

À la fin de ce TP, vous saurez :

- ✅ Utiliser SQLite avec `better-sqlite3` dans Node.js

-  Créer une logique d' initialisation de base de données
 -  Migrer des services de JSON vers SQL
 -  Créer des relations Many-to-Many (utilisateurs ↔ produits)
 -  Organiser le code selon l' architecture MVC
 -  Gérer les transactions et erreurs
-

Prérequis

```
# Vérifiez que vous avez Node.js installé
node --version # v18+ recommandé

# Clonez ou naviguez vers votre projet td3-alt
cd /chemin/vers/td3-alt
```

Structure actuelle du projet :

```
td3-alt/
├─ index.js
├─ package.json
├─ products.json          # ⚠ Sera remplacé par SQLite
├─ controllers/
│   └─ products.controller.js
├─ routes/
│   └─ products.router.js
├─ services/
│   └─ products.service.js  # ⚠ À migrer
├─ middlewares/
│   └─ products.middleware.js
├─ views/
│   └─ products.ejs
└─ public/
```



PARTIE 1 : Migration de Products vers SQLite

Étape 1.1 : Installation de better-sqlite3



Action guidée :

```
npm install better-sqlite3
```



Pourquoi `better-sqlite3` ?

- Synchrone et performant (pas de callbacks asynchrones)
 - Plus simple que `sqlite3` classique
 - Parfait pour des applications moyennes
-

Étape 1.2 : Créer le dossier et le fichier d'initialisation



Action guidée :

```
# Créer le dossier db/  
mkdir db  
touch db/init.js
```



Complétez `db/init.js` :

```

const Database = require('better-sqlite3');
const path = require('path');

// Créer ou ouvrir la base de données
const dbPath = path.join(__dirname, 'store.db');
const db = new Database(dbPath);

// Créer la table products si elle n'existe pas
db.exec(`
    CREATE TABLE IF NOT EXISTS products (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL UNIQUE,
        description TEXT NOT NULL,
        price REAL NOT NULL CHECK(price >= 0)
    )
`);

console.log('✅ Table products créée ou déjà existante');

// Insérer des données initiales si la table est vide
const count = db.prepare('SELECT COUNT(*) as count FROM products').get();

if (count.count === 0) {
    console.log('📦 Insertion des produits initiaux...');

    const insert = db.prepare(`
        INSERT INTO products (name, description, price)
        VALUES (?, ?, ?)
    `);

    insert.run('Laptop Dell XPS', 'Ordinateur portable haute performance',
1299.99);
    insert.run('iPhone 15 Pro', 'Smartphone Apple dernière génération',
1199.00);
    insert.run('Casque Sony WH-1000XM5', 'Casque antibruit premium',
349.99);

    console.log('✅ 3 produits insérés');
}

// Exporter l'instance de la base de données
module.exports = db;

```

Explications :

- `db.exec()` : Exécute du SQL brut (DDL)
 - `db.prepare()` : Prépare une requête pour l' exécuter plusieurs fois (performances optimales)
 - `.run()` : Exécute un INSERT/UPDATE/DELETE
 - `.get()` : Récupère une seule ligne
 - `.all()` : Récupère toutes les lignes
-

Étape 1.3 : Modifier `index.js` pour initialiser la DB

Action guidée :

Dans votre fichier `index.js`, **ajoutez cette ligne en haut, juste après les imports :**

```
const express = require("express");
const axios = require("axios");
const path = require("path");

// ★ AJOUTEZ CETTE LIGNE
require("./db/init"); // Initialise la base de données au démarrage

const app = express();
// ... reste du code
```

 **Pourquoi ?** Cela exécute le fichier `db/init.js` au démarrage de l' application, garantissant que la base est créée et initialisée avant toute requête.

Étape 1.4 : Migrer le Service Products vers SQLite

Action guidée :

Remplacez **TOUT** le contenu de `services/products.service.js` par :

```

const db = require("../db/init");

/**
 * Créer un produit
 * @param {string} name
 * @param {string} description
 * @param {number} price
 * @param {function} callback - Callback error-first (err, data)
 */
const createProduct = (name, description, price, callback) => {
  try {
    // Préparer la requête d'insertion
    const stmt = db.prepare(`
      INSERT INTO products (name, description, price)
      VALUES (?, ?, ?)
    `);

    // Exécuter l'insertion
    const result = stmt.run(name, description, price);

    // Créer l'objet du nouveau produit
    const newProduct = {
      id: result.lastInsertRowid, // ★ ID auto-généré
      name,
      description,
      price
    };

    callback(null, newProduct);
  } catch (error) {
    console.error('✗ Erreur createProduct:', error.message);
    callback(error, null);
  }
};

/**
 * Récupérer tous les produits
 * @param {function} callback
 */
const getProducts = (callback) => {
  try {
    const stmt = db.prepare('SELECT * FROM products');
    const products = stmt.all(); // ★ .all() retourne un tableau

    callback(null, products);
  }
};

```

```

    } catch (error) {
        console.error('✗ Erreur getProducts:', error.message);
        callback(error, null);
    }
};

/**
 * Mettre à jour un produit par son ID
 * @param {number} id
 * @param {string} name
 * @param {string} description
 * @param {number} price
 * @param {function} callback
 */
const updateProduct = (id, name, description, price, callback) => {
    try {
        // Vérifier si le produit existe
        const checkStmt = db.prepare('SELECT * FROM products WHERE id = ?');
        const product = checkStmt.get(id);

        if (!product) {
            return callback(new Error("Product not found"), null);
        }

        // Mettre à jour le produit
        const updateStmt = db.prepare(`
            UPDATE products
            SET name = ?, description = ?, price = ?
            WHERE id = ?
        `);

        updateStmt.run(name, description, price, id);

        callback(null, { id, name, description, price });
    } catch (error) {
        console.error('✗ Erreur updateProduct:', error.message);
        callback(error, null);
    }
};

/**
 * Supprimer un produit par son ID
 * @param {number} id
 * @param {function} callback
 */
const deleteProduct = (id, callback) => {

```

```

try {
  // Vérifier si le produit existe
  const checkStmt = db.prepare('SELECT * FROM products WHERE id = ?');
  const product = checkStmt.get(id);

  if (!product) {
    return callback(new Error("Product not found"), null);
  }

  // Supprimer le produit
  const deleteStmt = db.prepare('DELETE FROM products WHERE id = ?');
  deleteStmt.run(id);

  callback(null, "Produit supprimé avec succès");
} catch (error) {
  console.error('❌ Erreur deleteProduct:', error.message);
  callback(error, null);
}
};

module.exports = {
  createProduct,
  updateProduct,
  deleteProduct,
  getProducts
};

```

🔍 Différences clés JSON → SQLite :

Aspect	JSON (fs)	SQLite (better-sqlite3)
Lecture	<code>fs.readFileSync()</code>	<code>db.prepare().get()</code> ou <code>.all()</code>
Écriture	<code>fs.writeFileSync()</code>	<code>db.prepare().run()</code>
Identifiant	Géré manuellement	<code>AUTOINCREMENT</code> automatique
Concurrence	⚠️ Risque de corruption	✅ Transactions ACID
Performances	🐢 Lent si gros fichier	⚡ Indexé et optimisé

Étape 1.5 : Adapter le Controller (si nécessaire)

Vérification :

Ouvrez `controllers/products.controller.js` et vérifiez que :

- Les callbacks sont bien gérés
- Les réponses JSON sont correctes

⚠ **Modification nécessaire** : Si votre controller utilise `productName` au lieu de `id`, changez :

```
// AVANT (avec name)
exports.updateProduct = async (req, res, next) => {
  const { id } = req.params; // ⚠ Changé de name vers id
  const { name, description, price } = req.body;

  productsService.updateProduct(id, name, description, price, (error,
data) => {
    if (error) {
      const cbError = new Error(error.message);
      cbError.status = error.message === "Product not found" ? 404 :
500;

      next(cbError);
    } else {
      res.status(200).json({
        message: "Produit mis à jour",
        product: data
      });
    }
  });
};
```

Étape 1.6 : Tester la migration

Actions guidées :

```
# Lancer le serveur
npm run dev
```

Tests avec Postman ou curl :

```
# 1. Récupérer tous les produits
curl http://localhost:3000/api/products

# 2. Créer un produit
curl -X POST http://localhost:3000/api/products \
  -H "Content-Type: application/json" \
  -d '{"name":"AirPods Pro","description":"Écouteurs sans fil","price":279.99}'

# 3. Mettre à jour un produit (ID 1)
curl -X PUT http://localhost:3000/api/products/1 \
  -H "Content-Type: application/json" \
  -d '{"name":"Laptop Dell XPS 15","description":"Mis à jour","price":1399.99}'

# 4. Supprimer un produit (ID 2)
curl -X DELETE http://localhost:3000/api/products/2
```

✓ Résultat attendu :

- Les produits sont stockés dans `db/store.db` (fichier SQLite)
- Le fichier `products.json` n'est plus utilisé
- Toutes les opérations CRUD fonctionnent

🎓 DÉFI 1 : Ajouter une recherche par prix (SANS SOLUTION)

🎯 **Objectif :** Créer un endpoint qui retourne les produits dans une fourchette de prix.

📝 Spécifications :

- Route : `GET /api/products/filter?minPrice=100&maxPrice=500`
- Créer une fonction `getProductsByPriceRange(minPrice, maxPrice, callback)` dans le service
- Utiliser une requête SQL avec `WHERE price BETWEEN ? AND ?`

- Gérer les cas où les paramètres sont manquants

Indices :

- Utilisez `req.query.minPrice` et `req.query.maxPrice`
 - Pensez à valider que les prix sont des nombres
 - Testez avec : `curl "http://localhost:3000/api/products/filter?minPrice=200&maxPrice=600"`
-

PARTIE 2 : Ajouter la Gestion des Utilisateurs

Étape 2.1 : Créer la table Users dans `db/init.js`

Action guidée :

Dans `db/init.js`, après la création de la table `products`, ajoutez :

```

// Créer la table users
db.exec(`
  CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    firstname TEXT,
    lastname TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
  )
`);

console.log('✅ Table users créée ou déjà existante');

// Insérer des utilisateurs de test
const userCount = db.prepare('SELECT COUNT(*) as count FROM users').get();

if (userCount.count === 0) {
  console.log('👥 Insertion des utilisateurs initiaux...');

  const insertUser = db.prepare(`
    INSERT INTO users (email, password, firstname, lastname)
    VALUES (?, ?, ?, ?)
  `);

  // ⚠️ Note: En production, TOUJOURS hasher les mots de passe avec
  bcrypt!
  insertUser.run('alice@example.com', 'password123', 'Alice', 'Dupont');
  insertUser.run('bob@example.com', 'password456', 'Bob', 'Martin');
  insertUser.run('charlie@example.com', 'password789', 'Charlie',
'Durand');

  console.log('✅ 3 utilisateurs insérés');
}

```

🔍 Explication :

- `CURRENT_TIMESTAMP` : Date/heure automatique à l'insertion
- `UNIQUE` sur email : Évite les doublons
- ⚠️ **Attention** : Dans un vrai projet, utilisez `bcrypt` pour hasher les mots de passe !

Étape 2.2 : Créer la table de relation Many-to-Many

Action guidée :

Les utilisateurs peuvent acheter plusieurs produits, et un produit peut être acheté par plusieurs utilisateurs → relation **Many-to-Many**.

Dans `db/init.js`, ajoutez :

```

// Créer la table de jonction users_products
db.exec(`
    CREATE TABLE IF NOT EXISTS users_products (
        user_id INTEGER NOT NULL,
        product_id INTEGER NOT NULL,
        quantity INTEGER DEFAULT 1,
        purchased_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        PRIMARY KEY (user_id, product_id),
        FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
        FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE
    )
`);

console.log('✅ Table users_products (jonction) créée ou déjà existante');

// Insérer des achats de test
const purchaseCount = db.prepare('SELECT COUNT(*) as count FROM
users_products').get();

if (purchaseCount.count === 0) {
    console.log('🛒 Insertion des achats initiaux...');

    const insertPurchase = db.prepare(`
        INSERT INTO users_products (user_id, product_id, quantity)
        VALUES (?, ?, ?)
    `);

    // Alice achète 2 Laptops et 1 iPhone
    insertPurchase.run(1, 1, 2); // user_id=1, product_id=1
    insertPurchase.run(1, 2, 1);

    // Bob achète 1 Casque
    insertPurchase.run(2, 3, 1);

    console.log('✅ Achats insérés');
}

```

 **Schéma de la relation :**

users (1) ↔ (N) users_products (N) ↔ (1) products

users:

id	email	firstname
1	alice@example.com	Alice
2	bob@example.com	Bob

products:

id	name	price
1	Laptop Dell XPS	1299
2	iPhone 15 Pro	1199

users_products:

user_id	product_id	quantity
1	1	2
1	2	1
2	3	1

Étape 2.3 : Créer le Service Users

 Action guidée :

Créez le fichier `services/users.service.js` :

```

const db = require("../db/init");

/**
 * Récupérer tous les utilisateurs
 */
const getUsers = (callback) => {
  try {
    const stmt = db.prepare('SELECT id, email, firstname, lastname,
created_at FROM users');
    const users = stmt.all();
    callback(null, users);
  } catch (error) {
    console.error('✗ Erreur getUsers:', error.message);
    callback(error, null);
  }
};

/**
 * Récupérer un utilisateur par ID avec ses produits achetés
 */
const getUserWithProducts = (userId, callback) => {
  try {
    // Récupérer l'utilisateur
    const userStmt = db.prepare(`
      SELECT id, email, firstname, lastname, created_at
      FROM users
      WHERE id = ?
    `);
    const user = userStmt.get(userId);

    if (!user) {
      return callback(new Error("User not found"), null);
    }

    // Récupérer les produits achetés par cet utilisateur
    const productsStmt = db.prepare(`
      SELECT
        p.id,
        p.name,
        p.description,
        p.price,
        up.quantity,
        up.purchased_at
      FROM products p
      INNER JOIN users_products up ON p.id = up.product_id
    `);
  }
};

```



```

        WHERE up.user_id = ?
    `);

    const products = productsStmt.all(userId);

    // Construire la réponse
    const result = {
        ...user,
        products: products
    };

    callback(null, result);
} catch (error) {
    console.error('✗ Erreur getUserWithProducts:', error.message);
    callback(error, null);
}
};

/**
 * Créer un utilisateur
 */
const createUser = (email, password, firstname, lastname, callback) => {
    try {
        const stmt = db.prepare(`
            INSERT INTO users (email, password, firstname, lastname)
            VALUES (?, ?, ?, ?)
        `);

        const result = stmt.run(email, password, firstname, lastname);

        const newUser = {
            id: result.lastInsertRowid,
            email,
            firstname,
            lastname
        };

        callback(null, newUser);
    } catch (error) {
        console.error('✗ Erreur createUser:', error.message);

        // Gérer l'erreur de contrainte UNIQUE
        if (error.message.includes('UNIQUE')) {
            callback(new Error("Cet email existe déjà"), null);
        } else {
            callback(error, null);
        }
    }
};

```

```

    }
  }
};

/**
 * Ajouter un achat (lier user et product)
 */
const addPurchase = (userId, productId, quantity, callback) => {
  try {
    // Vérifier que l'utilisateur existe
    const userCheck = db.prepare('SELECT id FROM users WHERE id = ?').get(userId);
    if (!userCheck) {
      return callback(new Error("User not found"), null);
    }

    // Vérifier que le produit existe
    const productCheck = db.prepare('SELECT id FROM products WHERE id = ?').get(productId);
    if (!productCheck) {
      return callback(new Error("Product not found"), null);
    }

    // Insérer l'achat
    const stmt = db.prepare(`
      INSERT INTO users_products (user_id, product_id, quantity)
      VALUES (?, ?, ?)
      ON CONFLICT(user_id, product_id)
      DO UPDATE SET quantity = quantity + excluded.quantity
    `);

    stmt.run(userId, productId, quantity);

    callback(null, { userId, productId, quantity });
  } catch (error) {
    console.error('✗ Erreur addPurchase:', error.message);
    callback(error, null);
  }
};

module.exports = {
  getUsers,
  getUserWithProducts,
  createUser,
  addPurchase
};

```

Points clés :

- `INNER JOIN` : Récupère les produits liés à un utilisateur
 - `ON CONFLICT ... DO UPDATE` : Incrémente la quantité si l'achat existe déjà
 - Validation de l'existence des entités avant insertion
-

Étape 2.4 : Créer le Controller Users

Action guidée :

Créez `controllers/users.controller.js` :

```

const usersService = require("../services/users.service");

exports.getUsers = async (req, res, next) => {
  try {
    usersService.getUsers((error, users) => {
      if (error) {
        const err = new Error(error.message);
        err.status = 500;
        next(err);
      } else {
        res.status(200).json(users);
      }
    });
  } catch (error) {
    const err = new Error(error.message);
    err.status = 500;
    next(err);
  }
};

exports.getUserWithProducts = async (req, res, next) => {
  try {
    const { id } = req.params;

    usersService.getUserWithProducts(id, (error, user) => {
      if (error) {
        const err = new Error(error.message);
        err.status = error.message === "User not found" ? 404 : 500;
        next(err);
      } else {
        res.status(200).json(user);
      }
    });
  } catch (error) {
    const err = new Error(error.message);
    err.status = 500;
    next(err);
  }
};

exports.createUser = async (req, res, next) => {
  try {
    const { email, password, firstname, lastname } = req.body;

    usersService.createUser(email, password, firstname, lastname,

```

```

(error, user) => {
    if (error) {
        const err = new Error(error.message);
        err.status = error.message.includes("existe déjà") ? 409 :
500;

        next(err);
    } else {
        res.status(201).json({
            message: "Utilisateur créé avec succès",
            user: user
        });
    }
});
} catch (error) {
    const err = new Error(error.message);
    err.status = 500;
    next(err);
}
};

exports.addPurchase = async (req, res, next) => {
    try {
        const { userId, productId, quantity } = req.body;

        usersService.addPurchase(userId, productId, quantity || 1, (error,
data) => {
            if (error) {
                const err = new Error(error.message);
                err.status = 500;
                next(err);
            } else {
                res.status(201).json({
                    message: "Achat enregistré avec succès",
                    purchase: data
                });
            }
        });
    } catch (error) {
        const err = new Error(error.message);
        err.status = 500;
        next(err);
    }
};
};

```

Étape 2.5 : Créer le Router Users

Action guidée :

Créez `routes/users.router.js` :

```
const express = require("express");
const usersController = require("../controllers/users.controller");
const router = express.Router();

// GET /api/users - Récupérer tous les utilisateurs
router.get("/", usersController.getUsers);

// GET /api/users/:id - Récupérer un utilisateur avec ses produits
router.get("/:id", usersController.getUserWithProducts);

// POST /api/users - Créer un utilisateur
router.post("/", usersController.createUser);

// POST /api/users/purchase - Enregistrer un achat
router.post("/purchase", usersController.addPurchase);

module.exports = router;
```

Étape 2.6 : Enregistrer le Router dans `index.js`

Action guidée :

Dans `index.js`, ajoutez :

```
const productsRoutes = require("../routes/products.router");
const usersRoutes = require("../routes/users.router"); // ★ NOUVEAU

// ...

app.use("/api/products", productsRoutes);
app.use("/api/users", usersRoutes); // ★ NOUVEAU
```

Étape 2.7 : Tester les routes Users

Tests guidés :

```
# 1. Récupérer tous les utilisateurs
curl http://localhost:3000/api/users

# 2. Récupérer un utilisateur avec ses produits (ID 1)
curl http://localhost:3000/api/users/1

# 3. Créer un nouvel utilisateur
curl -X POST http://localhost:3000/api/users \
  -H "Content-Type: application/json" \
  -d '{
    "email":"david@example.com",
    "password":"securepass",
    "firstname":"David",
    "lastname":"Leroy"
  }'

# 4. Enregistrer un achat (user 1 achète product 3, quantité 2)
curl -X POST http://localhost:3000/api/users/purchase \
  -H "Content-Type: application/json" \
  -d '{
    "userId":1,
    "productId":3,
    "quantity":2
  }'

# 5. Vérifier que l'achat est bien enregistré
curl http://localhost:3000/api/users/1
```

✓ Résultat attendu pour `GET /api/users/1` :

```
{
  "id": 1,
  "email": "alice@example.com",
  "firstname": "Alice",
  "lastname": "Dupont",
  "created_at": "2025-11-04 10:30:00",
  "products": [
    {
      "id": 1,
      "name": "Laptop Dell XPS",
      "description": "Ordinateur portable haute performance",
      "price": 1299.99,
      "quantity": 2,
      "purchased_at": "2025-11-04 10:30:00"
    },
    {
      "id": 2,
      "name": "iPhone 15 Pro",
      "description": "Smartphone Apple dernière génération",
      "price": 1199.00,
      "quantity": 1,
      "purchased_at": "2025-11-04 10:30:00"
    }
  ]
}
```

🎓 DÉFI 2 : Ajouter la suppression d' un achat (SANS SOLUTION)

🎯 **Objectif** : Créer un endpoint pour supprimer un achat (ligne dans `users_products`).

📝 Spécifications :

- Route : `DELETE /api/users/purchase`
- Body JSON : `{ "userId": 1, "productId": 3 }`
- Créer `removePurchase(userId, productId, callback)` dans le service
- Retourner un message de succès ou une erreur 404 si l' achat n' existe pas

💡 Indices :

- Requête SQL : `DELETE FROM users_products WHERE user_id = ? AND product_id = ?`
 - Utilisez `.run()` et vérifiez `result.changes` pour savoir si une ligne a été supprimée
-

🎓 DÉFI 3 : Calculer le montant total dépensé par un utilisateur (SANS SOLUTION)

🎯 **Objectif** : Ajouter un endpoint qui retourne le montant total des achats d' un utilisateur.

📝 Spécifications :

- Route : `GET /api/users/:id/total`
- Créer `getUserTotalSpent(userId, callback)` dans le service
- Utiliser une requête SQL avec `SUM(p.price * up.quantity)`
- Retourner : `{ "userId": 1, "totalSpent": 3798.97 }`

💡 Exemple de requête SQL :

```
SELECT SUM(p.price * up.quantity) as total
FROM users_products up
INNER JOIN products p ON up.product_id = p.id
WHERE up.user_id = ?
```

🎓 DÉFI 4 : Ajouter une validation avec middleware (SANS SOLUTION)

🎯 **Objectif** : Créer un middleware de validation pour les utilisateurs.

📝 Spécifications :

- Fichier : `middlewares/users.middleware.js`
- Valider :
 - Email au format valide (utilisez le package `validator`)
 - Password minimum 6 caractères
 - Firstname et lastname non vides
- Retourner une erreur 400 avec un message clair si validation échoue

💡 Structure du middleware :

```
const validator = require('validator');

exports.validateUser = (req, res, next) => {
  const { email, password, firstname, lastname } = req.body;

  // TODO: Ajouter les validations

  next(); // Passer au prochain middleware si tout est OK
};
```

Récapitulatif : Avant / Après

Aspect	AVANT (JSON)	APRÈS (SQLite)
Stockage	Fichier JSON	Base de données SQLite
Relations	❌ Impossible	✅ Many-to-Many
Performances	🐢 Lent	⚡ Rapide avec index
Concurrence	⚠️ Risque corruption	✅ ACID sécurisé
Requêtes complexes	❌ Difficile	✅ SQL puissant
Scalabilité	❌ Limitée	✅ Meilleure

Compétences Acquises

✓ Migration de fichier JSON vers SQLite ✓ Utilisation de `better-sqlite3` (préparation, exécution, transactions) ✓ Création de schéma relationnel (1-N, N-N) ✓ Architecture MVC complète (Routes → Controllers → Services → DB) ✓ Gestion d'erreurs et callbacks error-first ✓ CRUD complet sur deux entités liées

Pour Aller Plus Loin

1. **Sécurité** : Hasher les mots de passe avec `bcrypt`
 2. **Authentification** : Ajouter JWT pour protéger les routes
 3. **Validation** : Utiliser `express-validator` ou `joi`
 4. **Pagination** : Implémenter `LIMIT` et `OFFSET` sur les listes
 5. **Transactions** : Utiliser `db.transaction()` pour opérations atomiques
 6. **Tests** : Ajouter des tests avec `jest` et `supertest`
-

Problèmes Courants

1. **Erreur** : `Error: SQLITE_CONSTRAINT: UNIQUE constraint failed` → Vous essayez d'insérer un email ou nom de produit qui existe déjà.
 2. **Erreur** : `Cannot find module './db/init'` → Vérifiez que le dossier `db/` existe et que `init.js` est dedans.
 3. **Les produits ne s'affichent pas** → Vérifiez que `require("./db/init")` est bien appelé en haut de `index.js`.
 4. `result.lastInsertRowid is undefined` → Utilisez `.run()` pour les INSERT, pas `.get()` ou `.all()`.
-



Validation Finale

Checklist de fin de TP :

- ☐ La table `products` existe dans `db/store.db`
 - ☐ La table `users` existe avec des données
 - ☐ La table `users_products` existe (relation N-N)
 - ☐ `GET /api/products` retourne les produits depuis SQLite
 - ☐ `GET /api/users` retourne les utilisateurs
 - ☐ `GET /api/users/:id` retourne un utilisateur avec ses produits
 - ☐ `POST /api/users/purchase` enregistre un achat
 - ☐ Le fichier `products.json` n' est plus utilisé
 - ☐ Les erreurs sont bien gérées (404, 409, 500)
-



Félicitations ! Vous avez réussi la migration et ajouté un système utilisateur complet !

Auteur : IUT Nord Franche-Comté **Dernière mise à jour :** Novembre 2025