

Fiche Descriptive : Gestion de Base de Données (Node.js Pur)

Contexte

Cette fiche présente les recommandations pour améliorer la gestion de la base de données JSON dans CinéReserve, en restant 100% Node.js pur sans ORM ni bibliothèques externes.

1. ÉTAT ACTUEL DE LA BASE DE DONNÉES

Architecture Actuelle

```
data/
├── films.json      # Liste des films
├── users.json      # Comptes utilisateurs
└── reservations.json # Réservations
```

Problèmes Identifiés

Performance

- Lecture/écriture complète du fichier à chaque opération
- Pas de cache en mémoire
- Pas d'indexation
- Lenteur croissante avec le volume de données

Intégrité des Données

- Pas de transactions (ACID)
- Risque de corruption si crash pendant l'écriture
- Pas de verrouillage concurrent (race conditions)
- Pas de contraintes de clés étrangères
- Pas de validation au niveau DB

Sécurité

- Données en clair dans les fichiers JSON
- Pas de chiffrement au repos
- Permissions fichiers non restrictives
- Pas de sauvegarde automatique

Scalabilité

- Limitation à quelques milliers d'enregistrements
 - Pas de pagination efficace
 - Pas de requêtes complexes (jointures, agrégations)
-

2. ARCHITECTURE AMÉLIORÉE

2.1 Structure des Fichiers

```
data/
    ├── films.json      # Table films
    ├── users.json       # Table users
    ├── reservations.json # Table reservations
    ├── indexes/        # Index pour recherches rapides
    |   ├── users_by_username.json
    |   └── reservations_by_user.json
    ├── backups/         # Sauvegardes automatiques
    |   ├── films_2025-10-04.json
    |   └── users_2025-10-04.json
    ├── transactions/    # Journal des transactions
    |   └── transaction_log.json
    └── .locks/          # Fichiers de verrouillage
        ├── films.lock
        └── users.lock
```

2.2 Module de Gestion de Base de Données

Fichier : `modules/database.js`

Fonctions principales :

```
javascript
```

```
// CRUD de base
read(table)          // Lire une table
write(table, data)   // Écrire une table
find(table, query)   // Rechercher
findOne(table, query) // Rechercher un seul élément
insert(table, record) // Insérer
update(table, id, updates) // Mettre à jour
delete(table, id)    // Supprimer

// Avancées
transaction(operations) // Transaction atomique
createIndex(table, field) // Créer un index
query(table, filters, options) // Requête complexe
backup(table)           // Sauvegarder
restore(table, date)    // Restaurer
```

🚀 3. AMÉLIORATIONS DE PERFORMANCE

3.1 Cache en Mémoire

Concept : Garder les données fréquemment accédées en RAM

Implémentation :

```
javascript
```

```
const cache = {
  films: null,
  users: null,
  reservations: null,
  lastLoad: {}
};

// Stratégies de cache
- Cache complet : charger toute la table en mémoire
- Cache TTL : invalider après X minutes (ex: 5 min)
- Cache LRU : garder les N éléments les plus récents
- Write-through : mettre à jour fichier ET cache simultanément
```

Avantages :

- Lectures 100x plus rapides
- Moins de sollicitation du disque
- Meilleure expérience utilisateur

Inconvénients :

- Utilisation de RAM
- Complexité de synchronisation
- Risque de données obsolètes

Recommandation :

- Cache pour `films.json` (données quasi-statiques)
- Pas de cache pour `reservations.json` (données critiques)
- TTL de 5 minutes pour `users.json`

3.2 Indexation

Concept : Créer des structures de recherche rapide

Implémentation :

```
javascript

// Index par username
{
  "admin": 1, // username -> userId
  "user1": 2
}

// Index par filmId pour réservations
{
  "1": [1, 3, 5], // filmId -> [reservationIds]
  "2": [2, 4]
}
```

Quand créer un index :

- Champs recherchés fréquemment (username, email, filmId)
- Champs utilisés dans les filtres
- Clés étrangères

Maintenance :

- Reconstruire l'index après chaque modification
- Stocker les index dans `data/indexes/`
- Invalider le cache d'index si corruption détectée

3.3 Pagination et Limitation

Fonction : `(query(table, filters, { limit, offset, sort }))`

Implémentation :

javascript

```
// Éviter de charger 10 000 réservations
query('reservations', { userId: 1 }, {
  limit: 20,    // 20 résultats max
  offset: 0,    // À partir du début
  sort: { date: -1 } // Plus récent d'abord
});
```

Bénéfices :

- Réduction de la mémoire utilisée
- Réponses plus rapides
- Meilleure expérience mobile

3.4 Chargement Lazy (Paresseux)

Concept : Ne charger que ce qui est nécessaire

Stratégies :

javascript

```
// Au lieu de charger toute la table
const allFilms = readJsonFile('films.json'); // ❌ Lent

// Charger uniquement le film demandé
function getFilmById(id) {
  const films = readJsonFile('films.json');
  return films.find(f => f.id === id);
}

// Encore mieux : utiliser un index
function getFilmByIdFast(id) {
  const index = readJsonFile('indexes/films_by_id.json');
  const position = index[id];
  return readFilmAtPosition(position); // Lecture partielle
}
```

4. INTÉGRITÉ DES DONNÉES

4.1 Transactions Atomiques

Problème : Si le serveur crash pendant une écriture, les données sont corrompues

Solution : Système de transactions

Implémentation :

javascript

```
// 1. Journal des transactions (WAL - Write-Ahead Log)
{
  "transactionId": "tx_abc123",
  "timestamp": "2025-10-04T14:32:01.123Z",
  "operations": [
    {
      "type": "update",
      "table": "films",
      "id": 1,
      "before": { "places_disponibles": 50 },
      "after": { "places_disponibles": 48 }
    },
    {
      "type": "insert",
      "table": "reservations",
      "data": { "filmId": 1, "userId": 1, "nombrePlaces": 2 }
    }
  ],
  "status": "committed"
}
```

Processus :

1. **BEGIN** : Créer une entrée dans le journal
2. **LOG** : Enregistrer toutes les opérations prévues
3. **EXECUTE** : Appliquer les modifications
4. **COMMIT** : Marquer comme réussie
5. **ROLLBACK** : Annuler en cas d'erreur (restaurer "before")

Récupération après crash :

- Au démarrage, vérifier les transactions "pending"
- Rejouer les transactions "committed" non appliquées
- Annuler les transactions "pending"

4.2 Verrouillage de Fichiers

Problème : Deux requêtes simultanées modifient le même fichier = corruption

Solution : Système de locks (verrous)

Implémentation :

```
javascript
```

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

async function acquireLock(table, timeout = 5000) {
  const lockFile = `data/.locks/${table}.lock`;
  const startTime = Date.now();

  while (true) {
    try {
      // Tenter de créer le fichier de verrouillage
      fs.writeFileSync(lockFile, process.pid.toString(), { flag: 'wx' });
      return true; // Lock acquis
    } catch (error) {
      // Lock déjà pris par un autre processus
      if (Date.now() - startTime > timeout) {
        throw new Error('Timeout: impossible d\'acquérir le verrou');
      }
      await sleep(50); // Attendre 50ms et réessayer
    }
  }
}

function releaseLock(table) {
  const lockFile = `data/.locks/${table}.lock`;
  if (fs.existsSync(lockFile)) {
    fs.unlinkSync(lockFile);
  }
}
```

Usage :

```
javascript
```

```
// Avant chaque écriture
await acquireLock('films');
try {
  // Modifications ici
  writeJsonFile('films.json', data);
} finally {
  releaseLock('films'); // TOUJOURS libérer
}
```

Types de locks :

- **Exclusif** : Une seule opération à la fois (écriture)
- **Partagé** : Multiples lectures simultanées OK, mais pas d'écriture

4.3 Validation des Contraintes

Contraintes à implémenter :

Clés Primaires

javascript

```
function validatePrimaryKey(table, id) {  
    const records = readJsonFile(`.${table}.json`);  
    const exists = records.some(r => r.id === id);  
    if(exists) {  
        throw new ValidationError(`ID ${id} existe déjà dans ${table}`);  
    }  
}
```

Clés Étrangères

javascript

```
function validateForeignKey(value, referenceTable, referenceField) {  
    const records = readJsonFile(`.${referenceTable}.json`);  
    const exists = records.some(r => r[referenceField] === value);  
    if(!exists) {  
        throw new ValidationError(`  
            ${value} n'existe pas dans ${referenceTable}.${referenceField}`  
        );  
    }  
}  
  
// Exemple : avant d'insérer une réservation  
validateForeignKey(filmId, 'films', 'id');  
validateForeignKey(userId, 'users', 'id');
```

Contraintes Uniques

javascript

```
function validateUnique(table, field, value, excludeId = null) {
  const records = readJsonFile(`.${table}.json`);
  const duplicate = records.find(r =>
    r[field] === value && r.id !== excludeId
  );
  if (duplicate) {
    throw new ValidationError(`${field} "${value}" existe déjà`);
  }
}

// Exemple : username unique
validateUnique('users', 'username', 'admin');
```

Contraintes de Domaine

javascript

```
function validateDomain(field, value, constraints) {  
    // Type  
    if (constraints.type && typeof value !== constraints.type) {  
        throw new ValidationError(`${field} doit être de type ${constraints.type}`);  
    }  
  
    // Min/Max pour nombres  
    if (typeof value === 'number') {  
        if (constraints.min && value < constraints.min) {  
            throw new ValidationError(`${field} doit être >= ${constraints.min}`);  
        }  
        if (constraints.max && value > constraints.max) {  
            throw new ValidationError(`${field} doit être <= ${constraints.max}`);  
        }  
    }  
  
    // Longueur pour strings  
    if (typeof value === 'string') {  
        if (constraints.minLength && value.length < constraints.minLength) {  
            throw new ValidationError(`${field} trop court (min: ${constraints.minLength})`);  
        }  
        if (constraints.maxLength && value.length > constraints.maxLength) {  
            throw new ValidationError(`${field} trop long (max: ${constraints.maxLength})`);  
        }  
    }  
  
    // Enum  
    if (constraints.enum && !constraints.enum.includes(value)) {  
        throw new ValidationError(`${field} doit être parmi: ${constraints.enum.join(', ')}`);  
    }  
}
```

4.4 Écriture Atomique

Problème : Si crash pendant (`fs.writeFileSync()`), fichier corrompu

Solution : Écriture avec fichier temporaire

Implémentation :

javascript

```
function atomicWrite(filePath, data) {
  const tempPath = `${filePath}.tmp`;
  const backupPath = `${filePath}.backup`;

  try {
    // 1. Sauvegarder l'ancien fichier
    if (fs.existsSync(filePath)) {
      fs.writeFileSync(filePath, JSON.stringify(data, null, 2));
    }

    // 2. Écrire dans un fichier temporaire
    fs.writeFileSync(tempPath, JSON.stringify(data, null, 2));

    // 3. Vérifier l'intégrité (peut-on le relire ?)
    const verification = JSON.parse(fs.readFileSync(tempPath, 'utf8'));

    // 4. Renommer atomiquement (opération système atomique)
    fs.renameSync(tempPath, filePath);

    // 5. Supprimer la backup si succès
    if (fs.existsSync(backupPath)) {
      fs.unlinkSync(backupPath);
    }

    return true;
  } catch (error) {
    // Restaurer depuis la backup si erreur
    if (fs.existsSync(backupPath)) {
      fs.writeFileSync(filePath, JSON.stringify(data, null, 2));
    }
    throw error;
  } finally {
    // Nettoyer les fichiers temporaires
    if (fs.existsSync(tempPath)) fs.unlinkSync(tempPath);
  }
}
```

5. SÉCURITÉ DES DONNÉES

5.1 Chiffrement au Repos

Concept : Chiffrer les fichiers JSON sur le disque

Module : `crypto` (natif Node.js)

Implémentation :

javascript

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

const ENCRYPTION_KEY = process.env.DB_ENCRYPTION_KEY; // 32 bytes
const ALGORITHM = 'aes-256-gcm';

function encrypt(data) {
  const iv = crypto.randomBytes(16); // Vecteur d'initialisation
  const cipher = crypto.createCipheriv(ALGORITHM, Buffer.from(ENCRYPTION_KEY, 'hex'), iv);

  let encrypted = cipher.update(JSON.stringify(data), 'utf8', 'hex');
  encrypted += cipher.final('hex');

  const authTag = cipher.getAuthTag();

  return {
    encrypted,
    iv: iv.toString('hex'),
    authTag: authTag.toString('hex')
  };
}

function decrypt(encryptedData) {
  const decipher = crypto.createDecipheriv(
    ALGORITHM,
    Buffer.from(ENCRYPTION_KEY, 'hex'),
    Buffer.from(encryptedData.iv, 'hex')
  );

  decipher.setAuthTag(Buffer.from(encryptedData.authTag, 'hex'));

  let decrypted = decipher.update(encryptedData.encrypted);
  decrypted += decipher.final('utf8');

  return JSON.parse(decrypted);
}
```