📋 Fiche Descriptive : Amélioration Application Node.js Pure

🎯 Contexte

Cette fiche présente les recommandations pour améliorer **CinéReserve**, une application de réservation de places de cinéma développée en **Node.js pur** (sans frameworks ni middlewares externes).

🔐 1. GESTION DE L'AUTHENTIFICATION

🔴 Problèmes Identifiés

- Mots de passe stockés en clair dans la base de données
- Absence de système de sessions ou tokens
- Aucune vérification d'authentification sur les routes protégées
- Vulnérabilité aux attaques par force brute

✅ Solutions Recommandées

1.1 Hashage des Mots de Passe

Module utilisé : `crypto` (natif Node.js)

Implémentation :

- Utiliser `crypto.scrypt()` ou `crypto.pbkdf2()` pour le hashage
- Générer un salt unique par utilisateur avec `crypto.randomBytes(16)`
- Stocker : `hash` + `salt` dans la base de données
- Ne jamais stocker les mots de passe en clair

Exemple de structure :

```json
{
  "username": "admin",
  "passwordHash": "a1b2c3d4...",
  "passwordSalt": "e5f6g7h8..."
}
```

1.2 Système de Tokens JWT Manuel

Module utilisé : `crypto` (natif Node.js)

Implémentation :

- Créer des tokens JWT manuellement : `base64(header).base64(payload).signature`
- Générer la signature avec HMAC-SHA256 : `crypto.createHmac('sha256', secret)`
- Stocker la clé secrète dans les variables d'environnement
- Inclure dans le payload : `userId`, `iat` (issued at), `exp` (expiration)

Structure du token :

```
eyJhbGc...  (header)
.eyJ1c2V... (payload: userId, exp)
.SflKxwR... (signature)
```

1.3 Gestion des Sessions

Stockage : Objet JavaScript en mémoire

Implémentation :

- Créer un objet : `sessions = { 'token123': { userId: 1, expiresAt: timestamp } }`
- Générer des tokens aléatoires : `crypto.randomBytes(32).toString('hex')`
- Définir une durée d'expiration (ex: 24h)
- Nettoyer les sessions expirées avec `setInterval(cleanExpiredSessions, 3600000)`

Header attendu :

```
Authorization: Bearer <token>
```

1.4 Protection des Routes

Fonction : `verifyAuth(req)`

Logique :

1. Extraire le token du header `Authorization`
2. Vérifier l'existence du token dans les sessions
3. Vérifier l'expiration
4. Retourner le `userId` si valide
5. Appeler cette fonction avant chaque handler de route protégée
6. Retourner 401 si échec

📊 2. GESTION DES LOGS

🔴 Problèmes Identifiés

- Logs basiques avec `console.log()`
- Absence de niveaux de log (debug, info, error)
- Pas de persistance des logs
- Impossible de tracer les erreurs ou auditer les actions

✅ Solutions Recommandées

2.1 Système de Logs Structuré

Fonction : `logger(level, message, metadata)`

Niveaux de log :

- `DEBUG` : Informations de débogage (désactivé en production)
- `INFO` : Événements normaux (connexion, création)
- `WARN` : Situations anormales mais gérables
- `ERROR` : Erreurs d'exécution
- `CRITICAL` : Erreurs graves nécessitant une intervention

Format standardisé :

```json
{
  "timestamp": "2025-10-04T14:32:01.123Z",
  "level": "INFO",
  "message": "Utilisateur connecté",
  "requestId": "abc123",
  "userId": 1,
  "ip": "192.168.1.1",
  "duration": "45ms"
}
```

2.2 Logs Persistants

Module utilisé : `fs` (natif Node.js)

Implémentation :

- Utiliser `fs.appendFileSync()` ou `fs.createWriteStream()`
- Organisation par jour : `logs/app-2025-10-04.log`
- Séparer par type :
  - `logs/access.log` : Toutes les requêtes HTTP
  - `logs/error.log` : Uniquement les erreurs
  - `logs/security.log` : Événements de sécurité
  - `logs/app.log` : Logs applicatifs généraux

2.3 Rotation des Logs

Objectif : Éviter les fichiers trop volumineux

Implémentation :

- Vérifier la taille du fichier avant écriture
- Si > 10 MB : archiver (`app-2025-10-04.log` → `app-2025-10-04.log.1`)
- Limiter à 5 archives par jour
- Supprimer les logs > 30 jours (cron simulé avec `setInterval`)

2.4 Logs Contextuels

Éléments à logger :

- `requestId` : UUID unique par requête
- `userId` : Si authentifié
- `ip` : Adresse IP du client
- `userAgent` : Navigateur/client
- `method` + `url` : Route appelée
- `statusCode` : Code de réponse HTTP
- `duration` : Temps de traitement
- `error` : Stack trace si erreur

2.5 Logs de Sécurité

Événements à logger :

- ✅ Connexions réussies et échouées
- ✅ Tentatives de brute force détectées
- ✅ Accès refusés (401, 403)
- ✅ Modifications de données sensibles
- ✅ Rate limiting déclenché
- ✅ Requêtes malformées ou suspectes

⚠️ 3. GESTION DES ERREURS

🔴 Problèmes Identifiés

- Try/catch incomplets
- Messages d'erreur génériques exposés au client
- Pas de distinction entre erreurs techniques et métier
- Risque de crash serveur sur erreurs non gérées

✅ Solutions Recommandées

3.1 Classes d'Erreurs Personnalisées

Implémentation : Étendre la classe `Error` native

Types d'erreurs :

```javascript
ValidationError (400)     // Données invalides
AuthenticationError (401) // Non authentifié
AuthorizationError (403)  // Non autorisé
NotFoundError (404)       // Ressource introuvable
ConflictError (409)       // Conflit (ex: username existe)
ServerError (500)         // Erreur serveur interne
```

Propriétés :

- `name` : Type d'erreur
- `message` : Message descriptif
- `statusCode` : Code HTTP correspondant
- `code` : Code d'erreur applicatif (ex: `AUTH_001`)
- `isOperational` : true si erreur attendue, false si bug

3.2 Handler d'Erreurs Centralisé

Fonction : `handleError(error, req, res)`

Logique :

1. Logger l'erreur avec contexte complet
2. Déterminer si erreur opérationnelle ou bug
3. Retourner une réponse adaptée au client :
   - Erreurs opérationnelles : message clair
   - Bugs : message générique + alerte développeur
4. Ne jamais exposer les stack traces en production

Format de réponse :

```json
{
  "success": false,
  "error": {
    "code": "FILM_002",
    "message": "Places insuffisantes",
    "details": { "disponibles": 5, "demandées": 10 }
  }
}
```

3.3 Validation Robuste des Entrées

Fonction : `validateSchema(data, schema)`

Vérifications :

- **Types** : string, number, boolean, array, object
- **Obligatoire** : champs requis présents
- **Longueurs** : min/max pour strings et arrays
- **Formats** : regex pour email, phone, dates, URLs
- **Plages** : min/max pour numbers
- **Whitelist** : rejeter les champs non définis

Sanitisation :

- Échapper `<`, `>`, `&`, `"`, `'` pour éviter XSS
- Trim les espaces inutiles
- Normaliser les données (lowercase pour emails)

3.4 Gestion des Erreurs Asynchrones

Bonnes pratiques :

- Wrapper toutes les fonctions async dans try/catch
- Utiliser `.catch()` sur toutes les promesses
- Gérer `process.on('uncaughtException')` pour éviter les crashs
- Gérer `process.on('unhandledRejection')` pour les promesses

Exemple de wrapper :

```javascript
function asyncHandler(fn) {
  return async (req, res) => {
    try {
      await fn(req, res);
    } catch (error) {
      handleError(error, req, res);
    }
  };
}
```

3.5 Codes d'Erreur Standardisés

Structure : `CATEGORY_NUMBER`

Exemples :

```
AUTH_001 : Token invalide
AUTH_002 : Token expiré
AUTH_003 : Identifiants incorrects
FILM_001 : Film non trouvé
FILM_002 : Places insuffisantes
USER_001 : Utilisateur existe déjà
VAL_001  : Champ requis manquant
VAL_002  : Format invalide
```

🛡️ 4. SÉCURITÉ

🔴 Problèmes Identifiés

- Vulnérabilité aux attaques par force brute
- Absence de rate limiting
- Pas de validation stricte des entrées
- Headers de sécurité manquants
- Risques d'injections et XSS

✅ Solutions Recommandées

4.1 Rate Limiting

Stockage : Objet JavaScript en mémoire

Implémentation :

```javascript
rateLimit = {
  'IP_ADDRESS': {
    count: 45,
    resetTime: 1696424520000,
    blocked: false
  }
}
```

Règles :

- Limiter à 100 requêtes/minute par IP
- Bloquer 15 minutes si limite dépassée
- Reset automatique avec `setTimeout()`
- Logger les IPs bloquées dans `security.log`
- Retourner 429 (Too Many Requests)

4.2 Protection Brute Force

Mécanisme :

- Compteur de tentatives échouées par username/IP
- Bloquer après 5 tentatives en 15 minutes
- Augmenter le délai progressivement (backoff exponentiel)
- Notification par log des tentatives suspectes

Stockage :

```javascript
loginAttempts = {
  'username_or_ip': {
    count: 3,
    lastAttempt: timestamp,
    blockedUntil: null
  }
}
```

4.3 Validation et Sanitisation

Validation stricte :

- Vérifier les types attendus (typeof, Array.isArray())
- Limiter les longueurs : username (3-30 chars), password (8-64 chars)
- Valider les formats avec regex : email, phone, dates
- Rejeter les valeurs négatives pour IDs et quantités
- Whitelist : accepter uniquement les champs définis

Sanitisation :

```javascript
function sanitizeHtml(str) {
  return str
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#x27;')
    .replace(/&/g, '&amp;');
}
```

4.4 Headers de Sécurité

Headers à ajouter systématiquement :

```javascript
'X-Content-Type-Options': 'nosniff'
'X-Frame-Options': 'DENY'
'X-XSS-Protection': '1; mode=block'
'Strict-Transport-Security': 'max-age=31536000; includeSubDomains'
'Content-Security-Policy': "default-src 'self'"
'Referrer-Policy': 'strict-origin-when-cross-origin'
'Permissions-Policy': 'geolocation=(), microphone=(), camera=()'
```

4.5 Protection Contre les Attaques

NoSQL/JSON Injection :

- Valider que les inputs sont des primitives (string, number, boolean)
- Rejeter les objets et arrays non attendus
- Ne jamais interpoler directement les inputs utilisateur

Path Traversal :

- Vérifier que les chemins de fichiers restent dans `data/`
- Bloquer `..`, `/`, `\` dans les noms de fichiers
- Utiliser `path.resolve()` et vérifier le chemin final

DOS (Denial of Service) :

- Limiter la taille du body à 1 MB max
- Vérifier le header `content-length`
- Timeout sur les requêtes > 30 secondes
- Rate limiting global

XSS (Cross-Site Scripting) :

- Sanitiser toutes les données affichées
- Utiliser `Content-Type: application/json` uniquement
- Headers CSP stricts

4.6 Gestion des Secrets

Variables d'environnement :

```bash
JWT_SECRET=<généré avec crypto.randomBytes(32).toString('hex')>
SESSION_SECRET=<idem>
DB_ENCRYPTION_KEY=<idem>
PORT=3000
NODE_ENV=production
```

Bonnes pratiques :

- Créer un fichier `.env.example` (sans valeurs réelles)
- Ajouter `.env` au `.gitignore`
- Ne JAMAIS commiter les secrets
- Utiliser des secrets de 256 bits minimum
- Rotation régulière des secrets (tous les 90 jours)

4.7 HTTPS (Production)

Implémentation :

- Utiliser `https.createServer()` avec certificats SSL/TLS
- Certificats : Let's Encrypt (gratuit) ou certificat commercial
- Rediriger automatiquement HTTP (port 80) → HTTPS (port 443)
- Forcer `secure: true` et `httpOnly: true` pour les cookies

4.8 Audit et Monitoring

Logs de sécurité :

- Toutes les tentatives d'accès non autorisées
- Modifications de données sensibles
- Changements de configuration
- Patterns d'attaque détectés

Métriques :

- Nombre de requêtes/minute
- Taux d'erreurs 4xx/5xx
- Temps de réponse moyen
- IPs bloquées

🏗️ 5. ARCHITECTURE ET BONNES PRATIQUES

5.1 Séparation des Responsabilités

Structure de fichiers :

```
cinereserve/
├── server.js           # Point d'entrée
├── config.js           # Configuration centralisée
├── modules/
│   ├── auth.js         # Authentification
│   ├── logger.js       # Système de logs
│   ├── validator.js    # Validation
│   ├── errors.js       # Classes d'erreurs
│   ├── security.js     # Rate limiting, sanitization
│   └── database.js     # Opérations DB
├── handlers/
│   ├── films.js        # Routes /films
│   ├── users.js        # Routes /login, /signup
│   └── reservations.js # Routes /reservations
├── data/               # Base de données JSON
└── logs/               # Fichiers de logs
```

5.2 Configuration Centralisée

Fichier : `config.js`

Contenu :

```javascript
module.exports = {
  port: process.env.PORT || 3000,
  jwtSecret: process.env.JWT_SECRET,
  sessionDuration: 24 * 60 * 60 * 1000, // 24h
  rateLimit: {
    maxRequests: 100,
    windowMs: 60 * 1000 // 1 minute
  },
  validation: {
    usernameMinLength: 3,
    passwordMinLength: 8,
    maxBodySize: 1024 * 1024 // 1MB
  },
  logs: {
    directory: './logs',
    maxFileSize: 10 * 1024 * 1024, // 10MB
    retentionDays: 30
  }
};
```

5.3 Documentation

À inclure :

- JSDoc pour chaque fonction
- README.md avec :
  - Installation et démarrage
  - Variables d'environnement requises
  - Exemples de requêtes pour chaque route
  - Codes d'erreur et leur signification
  - Architecture du projet
- CHANGELOG.md : historique des modifications
- SECURITY.md : politique de sécurité

📊 6. PRIORISATION DES AMÉLIORATIONS

🔴 CRITIQUE (À implémenter immédiatement)

Amélioration	Impact	Effort
Hashage des mots de passe	🔥 Très élevé	Faible
Validation des entrées	🔥 Très élevé	Moyen
Rate limiting	🔥 Élevé	Moyen
Gestion des erreurs async	🔥 Élevé	Faible

🔴 IMPORTANT (Court terme - 1 mois)

Amélioration	Impact	Effort
Système de tokens/sessions	Élevé	Moyen
Logs persistants	Élevé	Faible
Headers de sécurité	Élevé	Faible
Protection brute force	Élevé	Moyen

🟡 RECOMMANDÉ (Moyen terme - 3 mois)

Amélioration	Impact	Effort
Rotation des logs	Moyen	Moyen
Classes d'erreurs	Moyen	Moyen
Sanitisation avancée	Moyen	Moyen
Documentation complète	Moyen	Élevé

🟢 BONUS (Long terme - 6 mois)

Amélioration	Impact	Effort
HTTPS	Moyen	Moyen
Métriques et monitoring	Faible	Élevé
Tests automatisés	Faible	Élevé
CI/CD	Faible	Élevé

📚 7. RESSOURCES ET MODULES NATIFS

Modules Node.js Natifs Utilisés

Module	Usage
`http` / `https`	Serveur HTTP/HTTPS
`crypto`	Hashage, signatures, génération aléatoire
`fs`	Lecture/écriture fichiers, logs
`path`	Manipulation de chemins de fichiers
`url`	Parsing d'URLs et query strings
`querystring`	Parsing de query strings
`util`	Fonctions utilitaires (promisify)

Documentation Officielle

- Node.js Crypto : https://nodejs.org/api/crypto.html
- Node.js File System : https://nodejs.org/api/fs.html
- Node.js HTTP : https://nodejs.org/api/http.html
- OWASP Top 10 : https://owasp.org/www-project-top-ten/

✅ 8. CHECKLIST DE MISE EN PRODUCTION

Avant le Déploiement

- [ ] Tous les mots de passe sont hashés
- [ ] Les secrets sont dans des variables d'environnement
- [ ] Rate limiting activé
- [ ] Logs persistants configurés
- [ ] Headers de sécurité en place
- [ ] Validation stricte sur toutes les routes
- [ ] Gestion d'erreurs centralisée
- [ ] Tests manuels effectués
- [ ] Documentation à jour
- [ ] Certificat SSL/TLS obtenu

Monitoring Post-Déploiement

- [ ] Vérifier les logs d'erreur quotidiennement
- [ ] Surveiller les tentatives d'attaque
- [ ] Vérifier l'espace disque (logs)
- [ ] Analyser les performances
- [ ] Tester les sauvegardes
- [ ] Mettre à jour Node.js régulièrement

📞 CONTACT ET SUPPORT

Pour toute question sur cette fiche :

- Consulter la documentation Node.js officielle
- Vérifier les logs de sécurité
- Tester en environnement de développement avant production

Version : 1.0

Date : 4 octobre 2025

Projet : CinéReserve - Application Node.js Pure