🔒 Fiche Descriptive : Sécurisation d'une API Node.js Pure

Projet : CinéReserve

Contexte : Application de réservation de places de cinéma

Stack : Node.js pur (sans frameworks externes)

Date : Octobre 2025

📋 TABLE DES MATIÈRES

---

1. GESTION DE L'AUTHENTIFICATION

🎯 Objectif

Sécuriser l'accès à l'application et protéger les données utilisateurs.

⚠️ Problèmes actuels

- ❌ Mots de passe stockés en clair
- ❌ Absence de gestion de sessions
- ❌ Aucune protection des routes sensibles
- ❌ Pas de limitation des tentatives de connexion

✅ Solutions recommandées

A. Hashage des mots de passe avec crypto

Module : `crypto` (natif Node.js)
Méthode : `scrypt` ou `pbkdf2`

Principe :

- Générer un salt unique par utilisateur (16 bytes aléatoires)
- Hasher le mot de passe avec le salt
- Stocker : `salt + hash` dans la base de données
- À la connexion : recalculer le hash et comparer

Avantages :

- Impossible de retrouver le mot de passe original
- Chaque hash est unique même pour des mots de passe identiques
- Résistant aux attaques par rainbow tables

B. Système de tokens JWT "maison"

Module : `crypto` (natif Node.js)

Structure du token :

```
{
  userId: 123,
  username: "admin",
  iat: 1696435200,  // Date de création
  exp: 1696438800   // Date d'expiration
}
```

Processus :

1. Encoder les données en base64
2. Générer une signature HMAC avec une clé secrète
3. Format final : `header.payload.signature`
4. Envoyer le token au client après login
5. Client renvoie le token dans l'header `Authorization: Bearer <token>`

Durée de vie recommandée : 15-30 minutes

C. Middleware d'authentification

Fonction : `verifyToken(req, res, next)`

Responsabilités :

- Extraire le token du header Authorization
- Vérifier la signature du token
- Vérifier que le token n'est pas expiré
- Attacher les données utilisateur à `req.user`
- Retourner 401 si invalide

Utilisation : Appeler avant chaque route protégée

D. Refresh tokens

Principe : Double système de tokens

- **Access Token :** Court (15 min), pour les requêtes courantes
- **Refresh Token :** Long (7 jours), pour renouveler l'access token

Workflow :

1. Login → recevoir access + refresh tokens
2. Requêtes avec access token
3. Access token expiré → utiliser refresh token pour en obtenir un nouveau
4. Refresh token expiré → re-login obligatoire

Stockage refresh tokens : En base de données avec date d'expiration

---

E. Rate limiting sur /login

Objectif : Empêcher les attaques par brute force

Implémentation :

- Stocker en mémoire : `{ ip: { attempts: 3, lastAttempt: timestamp } }`
- Incrémenter à chaque échec
- Bloquer pendant 15 minutes après 5 échecs
- Réinitialiser après succès ou expiration

---

2. GESTION DES LOGS

🎯 Objectif

Tracer toutes les opérations pour faciliter le debug, l'audit et la détection d'incidents.

⚠️ Problèmes actuels

- ❌ Logs uniquement dans la console
- ❌ Pas de persistance
- ❌ Pas de niveaux de gravité
- ❌ Impossible de retrouver l'historique

✅ Solutions recommandées

A. Système de logs structuré

Module : `logger.js` personnalisé

Niveaux de logs :

Niveau	Usage	Exemple
DEBUG	Détails techniques	"Variable X = 42"
INFO	Événements normaux	"Utilisateur connecté"
WARN	Situations anormales non bloquantes	"Tentative de connexion échouée"
ERROR	Erreurs nécessitant attention	"Échec lecture fichier"
CRITICAL	Erreurs critiques	"Serveur inaccessible"

Format standardisé :

```
[2025-10-04T14:32:15.234Z] [INFO] [auth] Connexion réussie pour user:admin (IP: 192.168.1.10)
```

B. Rotation des logs

Stratégie : Un fichier par jour + limite de taille

Règles :

- Nouveau fichier chaque jour : `app-2025-10-04.log`
- Si fichier > 10 MB → créer `app-2025-10-04-part2.log`
- Conserver 30 jours d'historique
- Archiver en `.gz` les logs > 7 jours
- Supprimer automatiquement les logs > 30 jours

Implémentation : Vérification à chaque écriture ou via un cron job

---

C. Catégorisation des logs

Structure de dossiers :

```
logs/
├── access.log      # Toutes requêtes HTTP
├── error.log       # Erreurs uniquement
├── security.log    # Auth, tentatives suspectes
├── database.log    # Opérations sur les données
└── app.log         # Logs généraux
```

Redirection automatique : Chaque type de log va dans son fichier approprié

D. Informations à logger

Pour chaque requête HTTP :

- Timestamp
- Méthode + URL + Paramètres
- IP du client
- User-Agent
- Utilisateur authentifié (si applicable)
- Status code de réponse
- Temps de traitement (ms)

Pour les authentifications :

- Tentatives de login (succès/échec)
- Tokens générés/révoqués
- Changements de mot de passe
- Sessions expirées

Pour les erreurs :

- Message d'erreur complet
- Stack trace
- Contexte (URL, user, params)
- Données de la requête (sanitisées)

Pour les actions sensibles :

- Création/modification/suppression de réservations
- Changements sur les films (places disponibles)
- Actions administratives

E. Configuration par environnement

Variable : `LOG_LEVEL` dans `.env`

Environnement	LOG_LEVEL	Comportement
Development	DEBUG	Tous les logs, console + fichier
Staging	INFO	Logs importants, fichier uniquement
Production	WARN	Warnings et erreurs, fichier uniquement

---

3. GESTION DES ERREURS

🎯 Objectif

Capturer, traiter et communiquer les erreurs de manière cohérente et sécurisée.

⚠️ Problèmes actuels

- ❌ Gestion basique try/catch
- ❌ Messages techniques exposés au client
- ❌ Pas de centralisation
- ❌ Difficile de débugger

✅ Solutions recommandées

A. Classe d'erreur personnalisée

Fichier : `utils/AppError.js`

Propriétés :

```javascript
class AppError {
  constructor(message, statusCode, code, isOperational) {
    this.message = message;
    this.statusCode = statusCode;
    this.code = code;  // Ex: "FILM_NOT_FOUND"
    this.isOperational = isOperational;  // true/false
    this.timestamp = new Date().toISOString();
  }
}
```

Types d'erreurs :

- **Opérationnelles :** Prévisibles (404, validation, etc.)
- **Système :** Bugs, erreurs inattendues

B. Gestionnaire centralisé

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

Responsabilités :

1. Logger l'erreur complète
2. Déterminer si erreur opérationnelle ou système
3. Formater la réponse selon l'environnement
4. Envoyer la réponse au client

Réponse en production :

```json
{
  "success": false,
  "code": "FILM_NOT_FOUND",
  "message": "Le film demandé n'existe pas"
}
```

Réponse en développement :

```json
{
  "success": false,
  "code": "FILM_NOT_FOUND",
  "message": "Le film demandé n'existe pas",
  "stack": "Error: Film not found\n    at handleGetFilm..."
}
```

C. Wrapper pour fonctions async

Objectif : Capturer automatiquement les erreurs async

Implémentation :

```javascript
const asyncHandler = (fn) => {
  return (req, res, next) => {
    Promise.resolve(fn(req, res, next)).catch(next);
  };
};
```

Usage : Envelopper toutes les fonctions async de routes

D. Validation des entrées

Module : `utils/validator.js`

Fonctions de validation :

- `isEmail(email)` → Valide format email
- `isStrongPassword(password)` → Min 8 caractères, majuscules, chiffres
- `isPositiveInteger(value)` → Nombre entier positif
- `isInRange(value, min, max)` → Valeur dans intervalle
- `sanitizeString(str)` → Nettoie caractères dangereux

Principe : Valider AVANT traitement, rejeter avec erreur 400

E. Codes d'erreur standardisés

Format : Code en SNAKE_CASE + message clair

Exemples :

Code	Status	Message
FILM_NOT_FOUND	404	Film non trouvé
INVALID_CREDENTIALS	401	Identifiants incorrects
INSUFFICIENT_SEATS	400	Pas assez de places disponibles
TOKEN_EXPIRED	401	Votre session a expiré
RATE_LIMIT_EXCEEDED	429	Trop de requêtes, réessayez plus tard

---

F. Gestion des rejets non capturés

Écouter les événements process :

```javascript
process.on('unhandledRejection', (error) => {
  // Logger l'erreur
  // Fermer proprement le serveur
  // Redémarrer l'application
});

process.on('uncaughtException', (error) => {
  // Logger l'erreur
  // Arrêter immédiatement l'application
});
```

G. Timeouts

Objectif : Éviter les blocages infinis

Implémentation :

- Timeout global par requête : 30 secondes
- Timeout lecture fichier : 5 secondes
- Timeout requêtes externes : 10 secondes

Méthode : `setTimeout()` avec `clearTimeout()` après succès

---

4. SÉCURITÉ

🎯 Objectif

Protéger l'application contre les attaques courantes et les vulnérabilités.

⚠️ Problèmes actuels

- ❌ Pas de validation stricte des inputs
- ❌ Headers de sécurité absents
- ❌ Vulnérable aux attaques par injection
- ❌ Pas de rate limiting
- ❌ Secrets dans le code

✅ Solutions recommandées

A. Validation et sanitization des inputs

Règles d'or :

- ✅ Ne JAMAIS faire confiance aux données utilisateur
- ✅ Valider type, format, longueur, caractères autorisés
- ✅ Rejeter tout ce qui ne correspond pas aux règles
- ✅ Logger les tentatives suspectes

Techniques :

- Whitelisting (autoriser uniquement certains caractères)
- Blacklisting (interdire certains patterns)
- Échappement de caractères spéciaux
- Limitation de la taille des inputs

B. Headers de sécurité HTTP

À ajouter dans chaque réponse :

Header	Valeur	Protection
X-Content-Type-Options	nosniff	Empêche MIME sniffing
X-Frame-Options	DENY	Empêche clickjacking
X-XSS-Protection	1; mode=block	Protection XSS basique
Strict-Transport-Security	max-age=31536000	Force HTTPS
Content-Security-Policy	default-src 'self'	Limite sources de contenu
Referrer-Policy	no-referrer	Cache l'origine des requêtes

C. Rate limiting global

Objectif : Prévenir les attaques DoS et brute force

Implémentation :

- Tracker requêtes par IP en mémoire (Map)
- Limite : 100 requêtes par 15 minutes par IP
- Retourner 429 si dépassement
- Nettoyer les entrées expirées toutes les minutes

Endpoints sensibles : Limites plus strictes sur /login, /signup

---

D. Protection CSRF

Principe : Empêcher les requêtes cross-site non autorisées

Implémentation :

1. Générer un token CSRF unique à chaque session
2. L'envoyer au client (cookie ou réponse)
3. Client doit renvoyer le token dans les requêtes POST/PUT/DELETE
4. Serveur vérifie que le token correspond

Note : Moins critique pour une API REST pure (pas de cookies de session)

E. HTTPS obligatoire

Module : `https` (natif Node.js)

Étapes :

1. Obtenir un certificat SSL/TLS (Let's Encrypt gratuit)
2. Remplacer `http.createServer()` par `https.createServer()`
3. Passer les options `key` et `cert`
4. Rediriger automatiquement HTTP → HTTPS

En production : HTTPS OBLIGATOIRE, jamais de HTTP

F. Gestion des secrets

Règles :

- ❌ JAMAIS de secrets hardcodés dans le code
- ✅ Utiliser un fichier `.env` (non versionné)
- ✅ Lire avec `process.env.SECRET_KEY`
- ✅ Ajouter `.env` au `.gitignore`

Secrets à externaliser :

- Clés de signature JWT
- Clés API externes
- Identifiants de base de données
- Configuration serveur (ports, domaines)

G. Sécurité des fichiers

Si upload de fichiers :

Validations :

- Vérifier l'extension (whitelist)
- Vérifier le MIME type (ne pas se fier uniquement à l'extension)
- Limiter la taille (ex: max 5 MB)
- Scanner avec antivirus si possible

Stockage :

- Hors du dossier web accessible
- Renommer avec UUID aléatoire
- Permissions restrictives (lecture seule)

H. Prévention des injections

Types d'injections à prévenir :

NoSQL Injection :

- Valider que les IDs sont bien des nombres/strings simples
- Éviter l'évaluation dynamique de requêtes

Code Injection :

- Ne JAMAIS utiliser `eval()`, `Function()`, `exec()` sur des inputs
- Éviter `require()` dynamique basé sur input utilisateur

Path Traversal :

- Valider les chemins de fichiers
- Interdire `..`, `/`, `\` dans les noms de fichiers

I. Audit de sécurité

Actions régulières :

- Analyser les logs pour détecter patterns suspects
- Monitorer les tentatives de login échouées
- Suivre les requêtes 404 (scan de vulnérabilités ?)
- Alertes automatiques sur comportements anormaux

Métriques à surveiller :

- Taux d'erreurs 401/403
- Pics de trafic inhabituels
- Origines géographiques des attaques
- Endpoints les plus ciblés

J. Principe du moindre privilège

Règles d'accès :

- Chaque utilisateur ne doit accéder qu'à SES données
- Vérifier `req.user.id === reservation.userId` avant toute opération
- Ne pas se fier uniquement aux paramètres de requête
- Implémenter des rôles (user, admin) si nécessaire

---

5. ARCHITECTURE RECOMMANDÉE

📁 Structure du projet

```
cinereserve/
│
├── server.js                 # Point d'entrée principal
│
├── config/
│   ├── config.js             # Configuration centralisée
│   └── .env.example          # Template variables d'environnement
│
├── utils/
│   ├── logger.js             # Système de logs
│   ├── errorHandler.js       # Gestion centralisée des erreurs
│   ├── validator.js          # Fonctions de validation
│   ├── security.js           # Helpers de sécurité
│   ├── auth.js               # Gestion tokens/sessions
│   └── crypto.js             # Hashage, encryption
│
├── middleware/
│   ├── authenticate.js       # Vérification tokens
│   ├── rateLimit.js          # Rate limiting
│   ├── securityHeaders.js    # Headers de sécurité
│   ├── validator.js          # Validation des requêtes
│   └── errorMiddleware.js    # Interception erreurs
│
├── routes/
│   ├── films.js              # Routes films
│   ├── auth.js               # Routes authentification
│   └── reservations.js       # Routes réservations
│
├── controllers/
│   ├── filmsController.js    # Logique métier films
│   ├── authController.js     # Logique métier auth
│   └── reservationsController.js  # Logique métier réservations
│
├── data/                     # Base de données JSON
│   ├── films.json
│   ├── users.json
│   ├── reservations.json
│   └── tokens.json           # Refresh tokens actifs
│
├── logs/                     # Fichiers de logs
│   ├── access.log
│   ├── error.log
│   ├── security.log
│   └── app.log
│
├── .env                      # Variables d'environnement (non versionné)
├── .gitignore
├── package.json
└── README.md
```

🔄 Flux de traitement d'une requête

```
Requête HTTP
    ↓
1. securityHeaders middleware (ajoute headers de sécurité)
    ↓
2. rateLimit middleware (vérifie limits)
    ↓
3. logger middleware (log la requête)
    ↓
4. validator middleware (valide les données)
    ↓
5. authenticate middleware (vérifie le token si route protégée)
    ↓
6. Router (dirige vers le bon contrôleur)
    ↓
7. Controller (logique métier)
    ↓
8. Réponse JSON
    ↓
9. errorMiddleware (si erreur survenue)
    ↓
Réponse HTTP
```

---

6. PLAN D'IMPLÉMENTATION

🎯 Phase 1 : Sécurité Critique (Priorité HAUTE)

Durée estimée : 2-3 jours

Tâche	Effort	Impact
1. Hashage des mots de passe avec crypto	2h	⭐⭐⭐⭐⭐
2. Système de tokens JWT maison	4h	⭐⭐⭐⭐⭐
3. Middleware d'authentification	2h	⭐⭐⭐⭐⭐
4. Validation stricte des inputs	3h	⭐⭐⭐⭐⭐
5. Headers de sécurité HTTP	1h	⭐⭐⭐⭐

Résultat : Application sécurisée contre les attaques basiques

---

🎯 Phase 2 : Robustesse (Priorité MOYENNE)

Durée estimée : 2-3 jours

Tâche	Effort	Impact
6. Système de logs structuré	4h	⭐⭐⭐⭐
7. Gestionnaire d'erreurs centralisé	3h	⭐⭐⭐⭐
8. Rate limiting global	3h	⭐⭐⭐⭐
9. Rate limiting sur /login	2h	⭐⭐⭐⭐
10. Classe AppError personnalisée	2h	⭐⭐⭐

Résultat : Application stable et tracée

🎯 Phase 3 : Production-Ready (Priorité MOYENNE)

Durée estimée : 2 jours

Tâche	Effort	Impact
11. Configuration HTTPS	2h	⭐⭐⭐⭐⭐
12. Gestion des variables d'environnement	1h	⭐⭐⭐⭐
13. Rotation des logs	3h	⭐⭐⭐
14. Protection CSRF	2h	⭐⭐⭐
15. Timeouts sur requêtes	2h	⭐⭐⭐

Résultat : Application déployable en production

🎯 Phase 4 : Fonctionnalités Avancées (Priorité BASSE)

Durée estimée : 2-3 jours

Tâche	Effort	Impact
16. Refresh tokens	4h	⭐⭐⭐⭐
17. Audit et monitoring	3h	⭐⭐⭐
18. Logs par catégorie	2h	⭐⭐⭐
19. Codes d'erreur standardisés	2h	⭐⭐
20. Sanitization avancée	3h	⭐⭐⭐

Résultat : Application professionnelle complète

📊 Métriques de succès

Sécurité :

- ✅ 0 mot de passe en clair
- ✅ 100% des routes sensibles protégées
- ✅ Tous les inputs validés
- ✅ HTTPS activé en production

Performance :

- ✅ Temps de réponse < 200ms (95e percentile)
- ✅ Rate limiting fonctionnel
- ✅ Pas de memory leaks

Observabilité :

- ✅ 100% des erreurs loggées
- ✅ Traçabilité complète des actions
- ✅ Alertes sur comportements suspects

Qualité du code :

- ✅ Gestion d'erreurs cohérente
- ✅ Code commenté et documenté
- ✅ Architecture modulaire

📚 RESSOURCES ET DOCUMENTATION

Modules Node.js natifs utilisés

- `crypto` - Cryptographie et hashage
- `http` / `https` - Création du serveur
- `fs` - Lecture/écriture de fichiers
- `path` - Manipulation de chemins
- `url` - Parsing d'URLs
- `querystring` - Parsing de query strings

Documentation officielle

- Node.js Crypto
- Node.js HTTPS
- OWASP Top 10
- JWT RFC 7519

Bonnes pratiques

- Node.js Security Best Practices
- OWASP Cheat Sheet Series

✅ CHECKLIST DE VALIDATION

Avant déploiement en production :

Sécurité :

- [ ] Mots de passe hashés avec salt
- [ ] Tokens JWT implémentés et testés
- [ ] HTTPS activé
- [ ] Headers de sécurité présents
- [ ] Rate limiting actif
- [ ] Validation de tous les inputs
- [ ] Pas de secrets dans le code
- [ ] Fichier .env configuré

Logs :

- [ ] Système de logs fonctionnel
- [ ] Rotation des logs configurée
- [ ] Logs par catégorie
- [ ] Pas de données sensibles dans les logs

Erreurs :

- [ ] Gestionnaire centralisé en place
- [ ] Messages d'erreur sécurisés (pas de détails techniques)
- [ ] Codes d'erreur standardisés
- [ ] Gestion des rejets non capturés

Tests :

- [ ] Tests de connexion (succès/échec)
- [ ] Tests de rate limiting
- [ ] Tests de validation d'inputs
- [ ] Tests de gestion d'erreurs
- [ ] Tests de sécurité (injections, XSS, etc.)

🎓 CONCLUSION

Cette fiche représente un guide complet pour transformer une application Node.js prototype en une solution **production-ready** et sécurisée, tout en restant en **Node.js pur** sans dépendances externes.

Temps total estimé : 8-11 jours de développement

Niveau de complexité : Intermédiaire à Avancé

ROI : Application professionnelle, sécurisée et maintenable

L'implémentation de ces recommandations garantit :

- 🔒 **Sécurité** contre les attaques courantes
- 📊 **Observabilité** complète de l'application
- 🛡️ **Robustesse** face aux erreurs
- 🚀 **Scalabilité** pour la croissance future

Document créé par : Assistant IA Claude

Version : 1.0

Dernière mise à jour : Octobre 2025