PROJET FINAL — Sécurité des applications web

Mathis BOHEC & Marie COURTOIS

Repo GitHub : https://github.com/MathisBoMe/projet-final

SOMMAIRE

Résumé exécutif — Mesures de sécurité implémentées

Notre API Cinema met en place une stratégie de défense en profondeur : authentification robuste, contrôle d'accès, validation systématique des entrées, protections réseau (CORS/headers/TLS), limitation des abus (rate limiting / taille payload), et traçabilité (logging sécurité). L'objectif est de réduire la surface d'attaque, limiter l'impact d'un incident et améliorer la capacité d'investigation.

1) Authentification & autorisation

- **JWT + Refresh tokens** : access token court (2h) et refresh token (7 jours) avec rotation automatique.
- **Limitation de session** : maximum 5 refresh tokens actifs par utilisateur afin de réduire l'exposition (vol d'un token = fenêtre plus petite).
- **Hash mots de passe** : bcrypt (10 rounds) + politique de longueur 6–128 caractères.
- **Anti-énumération** : messages d'erreurs génériques sur login/inscription.
- **RBAC** : séparation des rôles (user / admin) via middleware, avec journalisation des accès refusés.

2) Protection des données

- **Validation & sanitisation** : types, formats, longueurs, trimming, bornage des champs ; protection contre injections et mass assignment par whitelisting des champs.
- **Gestion d'erreurs** : middleware centralisé masquant les détails techniques en production, logs serveur uniquement, traitement spécifique selon le type d'erreur (JWT, Prisma, MongoDB).

3) Communications

- **Headers de sécurité (Helmet)** : CSP, anti-clickjacking, nosniff, protections XSS ; HSTS en production.
- **CORS** : allowlist dynamique via variables d'environnement.

4) Protection contre attaques

- **Rate limiting multi-niveaux** : global (100 req/min) + authentification (10 tentatives / 15 min), journalisation des dépassements.
- **Limitation de taille** : payload JSON / urlencoded limité à 10 MB (DoS).
- **Validation environnement** : vérification au démarrage des variables critiques + force minimale du JWT_SECRET (≥ 32 caractères).

5) Monitoring & logging

- **Logs de sécurité** : tentatives de login (succès/échec), accès à ressources sensibles, violations de rate limit, erreurs sécurité ; corrélation possible par userId/IP.

6) Bonnes pratiques

- Architecture modulaire (middlewares dédiés, validations réutilisables) ; principes : moindre privilège, fail securely, ne jamais faire confiance aux entrées.
- Alignement OWASP Top 10 : couverture des risques critiques (injection, access control, misconfig, crypto, composants vulnérables, DoS, etc.).

1. Présentation du projet

a. Contexte

Dans le cadre du module Sécurité des applications web, nous avons repris et finalisé une application web autour d'un cinéma en mettant l'accent sur une mission "réelle" : une application existe (ou démarre) et doit être auditée puis sécurisée sérieusement avant mise en production.

L'objectif était de construire une API back-end permettant de gérer un catalogue de films, tout en appliquant de manière concrète les notions vues en cours : sécurité des communications, gestion d'authentification, durcissement OWASP, gestion des secrets et utilisation d'outils de sécurité.

b. Objectifs (fonctionnels / non fonctionnels)

Objectifs fonctionnels

- CRUD des films (administrateur)
- Inscription, authentification, gestion de profil utilisateur
- Notation de films par les utilisateurs authentifiés
- Recherche/filtrage des films (titre, genre, réalisateur)
- Administration : gestion du catalogue + gestion des utilisateurs

Objectifs non fonctionnels

- Sécurité : mise en place de protections concrètes (JWT, CORS, validation, rate limiting, etc.)
- Fiabilité : gestion d'erreurs propre, logs, comportements cohérents
- Maintenabilité : architecture MVC + séparation des responsabilités
- Testabilité : tests des endpoints critiques + documentation Swagger

c. Acteurs / rôles

- Visiteur : consulte et recherche des films, sans interaction
- Utilisateur authentifié : note des films, consulte son profil
- Administrateur : gère le catalogue de films et les utilisateurs
- Client externe : application web/mobile consommant l'API

2. Objectifs techniques

a. Architecture

L'API respecte une architecture 3-tiers :
Client → API Express → Bases de données

Ce découpage permet :

- une séparation claire des responsabilités,
- une meilleure maintenabilité,
- une surface d'attaque mieux contrôlable (politiques CORS, auth centralisée, etc.).

b. Technologies & justification

- Node.js / Express : framework léger et adapté aux APIs REST
- PostgreSQL + Prisma : données structurées et relationnelles (films, acteurs, réalisateurs, jointures)
- MongoDB + Mongoose : stockage de logs / historiques d'événements (et potentiellement notes selon vos choix)
- bcrypt : hash robuste des mots de passe (obligatoire)
- JWT (jsonwebtoken) : access token + refresh token (auth stateless)
- cors : contrôle strict des origines autorisées
- helmet : headers HTTP de sécurité
- dotenv : gestion des variables d'environnement
- morgan : logs HTTP
- Swagger : documentation **interactive (/api-docs)**

3. Conception fonctionnelle

a. Scénarios utilisateur

1. Consulter les films : liste + détail (visiteur)
2. Rechercher un film : filtre par titre/genre/réalisateur
3. S'inscrire / se connecter : création compte + token JWT
4. Noter un film : utilisateur connecté attribue une note

5. Administrer : admin crée/modifie/supprime films (+ gestion users)

b. Cas d'usage clés

- UC01 : Lister / consulter les films
- UC02 : Rechercher / filtrer
- UC03 : Inscription / login
- UC04 : Notes sur les films
- UC05 : Administration du catalogue (et des utilisateurs)

c. Modèle de données (MCD/MLD)

Entités principales

- Film, Réalisateur, Acteur, Utilisateur
- Relations :
  - *Réaliser* : 1 réalisateur → N films ; 1 film → 1 réalisateur
  - *Jouer_dans* : relation N–N (table de jointure Film_Acteur)
  - *Noter* : un utilisateur peut noter plusieurs films, un film peut recevoir plusieurs notes

Implémentation relationnelle (exemple MLD)

- REALISATEUR(id, name, age, nationality)
- ACTEUR(id, name, age, nationality)
- FILM(id, name, release_date, id_realisateur FK)
- FILM_ACTEUR(id, id_film FK, id_acteur FK)

d. Répartition SQL vs NoSQL (justification)

- PostgreSQL : entités fortement structurées et relationnelles (films, acteurs, réalisateurs, participations…) pour garantir cohérence + contraintes + intégrité.
- MongoDB : données flexibles et orientées documents (logs, historiques d'événements, et potentiellement notes si vous les stockez côté

NoSQL) afin d'enregistrer facilement des événements/objets sans migrations fréquentes.

4. Développement de l'application

a. Structure Express REST MVC

- routes/ : endpoints + branchement middlewares
- middlewares/ : CORS, auth JWT, RBAC, validation, rate limiting, gestion erreurs
- controllers/ : traitement HTTP (req/res), appels services
- services/ : logique métier (règles, validations métiers)
- prisma/ : schéma + migrations PostgreSQL
- models/ (Mongo) : schémas Mongoose (logs/historiques/notes si retenu)
- docs/ : Swagger/OpenAPI

b. Documentation & tests

- Documentation Swagger exposée sur /api-docs
- Tests des endpoints critiques : auth, routes admin, notation, recherche

[À COMPLETER : outils de test + exemples de cas testés]

5. Sécurisation de l'API

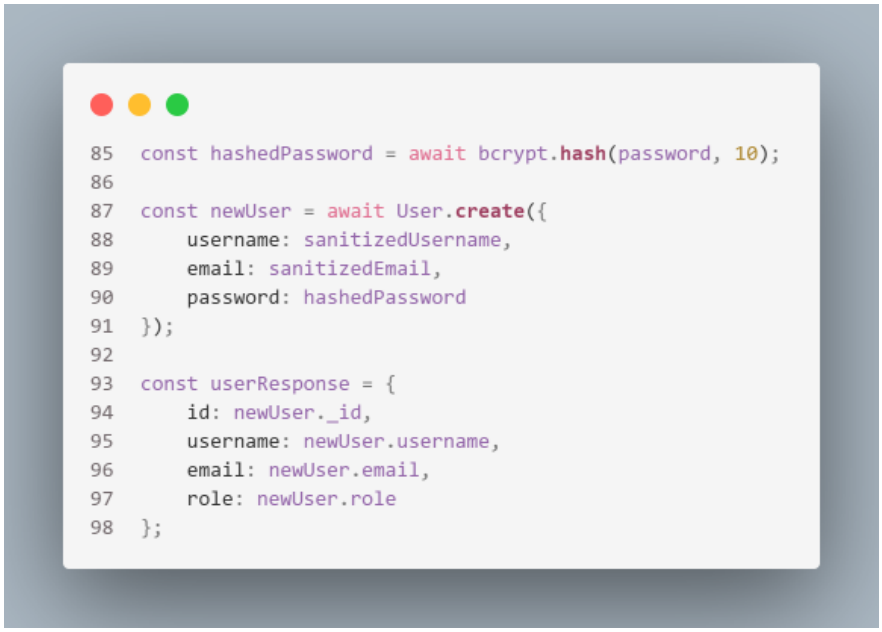
a. Authentification & gestion de session (JWT + bcrypt)

Hashage des mots de passe (bcrypt)

Objectif : éviter tout stockage de mot de passe en clair et limiter l'impact d'une fuite de base.

Preuve (implémentation)
Lors de l'inscription, le mot de passe est hashé avec bcrypt (coût = 10)

puis stocké en base. La réponse API renvoie un objet utilisateur sans le champ password.



```
85  const hashedPassword = await bcrypt.hash(password, 10);
86
87  const newUser = await User.create({
88      username: sanitizedUsername,
89      email: sanitizedEmail,
90      password: hashedPassword
91  });
92
93  const userResponse = {
94      id: newUser._id,
95      username: newUser.username,
96      email: newUser.email,
97      role: newUser.role
98  };
```

Comportement attendu

- En base : seul le hash bcrypt est stocké.
- En réponse : le hash n'est jamais renvoyé au client.

Risque couvert

- Réduction drastique des attaques offline en cas de compromission de la base (OWASP : cryptographic failures / bonnes pratiques auth).

Authentification par JWT (access token)

Objectif : authentification stateless pour protéger les routes privées et supporter le contrôle d'accès par rôle.

Preuve (implémentation)
Au login, l'API génère un token JWT signé avec un secret d'environnement, contenant userId et role, avec une expiration de 2 heures.

```
147         const token = jwt.sign(
148             { userId: user._id, role: user.role },
149             process.env.JWT_SECRET,
150             { expiresIn: "2h" }
151          );
152
153          res.json({
154             token,
155             user: {
156                 id: user._id,
157                 username: user.username,
158                 email: user.email,
159             },
160          });
161
```

Comportement attendu

- Le client stocke le token et l'envoie sur les routes protégées via Authorization: Bearer <token>.
- Au-delà de 2h : token expiré → refus d'accès (en pratique : 401 si middleware de vérif JWT).

Risque couvert

- Empêche l'accès aux routes protégées sans token valide.
- Supporte l'autorisation "par rôle" puisque role est porté par le token (OWASP : Identification & Authentication Failures + Broken Access Control).

Point de vigilance

- La sécurité dépend de la robustesse de JWT_SECRET (long, aléatoire, jamais commit).
- 2h = confortable, mais en production on peut réduire l'expiration pour diminuer la fenêtre en cas de vol (option d'amélioration).

Refresh token (génération + stockage + limitation à 5 tokens actifs)

Objectif : permettre de renouveler une session sans réauthentification complète tout en limitant l'exposition en cas de fuite de refresh token (défense en profondeur).

Preuve (implémentation)

- Génération d'un refresh token fort :
    - crypto.randomBytes(64).toString('hex') → token aléatoire (64 bytes encodés hex)
- Stockage côté utilisateur dans un tableau refreshTokens
- Limitation à 5 refresh tokens actifs :
    - si refreshTokens.length >= 5, suppression du token le plus ancien via refreshTokens.shift()
    - ajout du nouveau token via refreshTokens.push(refreshToken)
- Mise à jour en base :
    - User.findByIdAndUpdate(user._id, { refreshTokens })
- Réponse renvoyée au client :
    - res.json({ accessToken, refreshToken, user: {...} })

```
142  const refreshToken = crypto.randomBytes(64).toString('hex');
143
144  // Limiter à 5 refresh tokens actifs par utilisateur (rotation)
145  const userWithTokens = await User.findById(user._id).select('+refreshTokens');
146  const refreshTokens = userWithTokens.refreshTokens || [];
147  if (refreshTokens.length >= 5) {
148      refreshTokens.shift();
149  }
150  refreshTokens.push(refreshToken);
151
152  await User.findByIdAndUpdate(user._id, { refreshTokens });
153
154  securityLogger.logLoginAttempt(req, true);
155
156  res.json({
157      accessToken,
158      refreshToken,
159      user: {
160          id: user._id,
161          username: user.username,
162          email: user.email,
163      },
164  });
```

Comportement attendu

- À chaque authentification, un refresh token est émis et stocké côté serveur.
- L'utilisateur ne peut pas cumuler plus de 5 refresh tokens : l'ancien est éjecté en premier (FIFO).
- En cas de compromission d'un vieux refresh token, sa validité est limitée par cette politique de rotation/éviction.

Point de vigilance (audit)

- Le refresh token est stocké en base en clair (tel que visible) : amélioration possible = stocker un hash du refresh token (comme un mot de passe) et comparer côté serveur.
- En production multi-instances, aucun souci particulier ici (car stockage en DB), contrairement au CSRF en mémoire.

b. Protection contre les attaques automatisées (Brute force / DoS basique)

Rate limiting global + renforcé sur le login

Objectif : réduire les attaques par brute force et limiter les abus (scan/spam).

Preuve (implémentation)
Deux limites sont appliquées :

- **global** : 100 requêtes / 1 minute
- **login** : 10 tentatives / 15 minutes sur /auth/login

Comportement attendu

- Dépassement → réponse standard du middleware : HTTP 429 Too Many Requests.

- Le login est significativement ralenti en cas de brute force.

Risque couvert

- Brute force password stuffing sur /auth/login.
- Dégradation de service “low effort”.

Limites / points production

- En multi-instances, il faut un store partagé (ex : Redis) sinon les limites sont “par instance”.
- Derrière un proxy, config trust proxy pour une détection IP correcte.

c. CORS (contrôle des origines et durcissement des appels navigateur)

Allowlist d’origines + blocage des origines non autorisées

Objectif : empêcher des sites tiers non autorisés d’appeler l’API depuis un navigateur.

Preuve (implémentation)

- Origines autorisées : allowedOrigins
- Origine non autorisée → Error("Non autorisé par CORS")
- Requêtes sans Origin : possibles (Postman / mobile / server-to-server)
- credentials: true
- Headers autorisés : Authorization, Content-Type, X-Requested-With
- Méthodes autorisées : GET, POST, PUT, DELETE, PATCH, OPTIONS

```
app.use(cors({
    origin: function (origin, callback) {
        // Autoriser les requêtes sans origine (mobile apps, Postman, etc.) en développement
        if (!origin && process.env.NODE_ENV !== 'production') {
            return callback(null, true);
        }
        if (allowedOrigins.indexOf(origin) !== -1 || !origin) {
            callback(null, true);
        } else {
            callback(new Error('Non autorisé par CORS'));
        }
    },
    credentials: true,
    allowedHeaders: ["Authorization", "Content-Type", "X-Requested-With"],
    methods: ["GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"]
}));
```

Comportement attendu

- Origine autorisée (dans allowedOrigins) → requête acceptée (callback(null, true)).
- Origine non autorisée → requête refusée (Error: "Non autorisé par CORS").
- Requête sans header Origin :
    - fréquent pour Postman, mobile apps, ou appels serveur-à-serveur ;
    - la config peut l'accepter (selon NODE_ENV + logique actuelle).

Justification (pourquoi accepter parfois !origin)

- CORS protège surtout contre les attaques depuis un navigateur (où Origin est présent).
- Un client non navigateur peut ne pas envoyer Origin : l'objectif est de ne pas bloquer ces usages légitimes.

Point de vigilance

- En production, si on veut une politique plus stricte "browser only", on peut choisir de refuser les requêtes sans Origin, mais cela peut bloquer Postman / certains clients.
- de sécurité majeur (les non-browser clients n'ont pas les mêmes contraintes), mais c'est à justifier : "les clients serveur-à-serveur peuvent être autorisés même sans Origin, le risque principal CORS concerne le navigateur".

d. Validation des données & réponses d'erreur cohérentes

Middleware de validation centralisé (express-validator)

Objectif : refuser toute requête dont les données ne respectent pas le contrat attendu (types, formats, contraintes), avant d'entrer dans la logique métier.

Preuve (implémentation)
Le middleware validate rejette la requête si validationResult(req) contient des erreurs et renvoie HTTP 422 avec un tableau d'erreurs.

```
1  const { validationResult } = require("express-validator");
2
3  function validate(req, res, next) {
4    const errors = validationResult(req);
5    if (!errors.isEmpty()) {
6      return res.status(422).json({ errors: errors.array() });
7    }
8    next();
9  }
10
```

Les règles sont définies au niveau des routes (body/query/params). Le middleware validate ne fait que centraliser la réponse 422.

Comportement attendu

- Entrée invalide → 422 Unprocessable Entity + { errors: [...] }
- Entrée valide → next() → contrôleur exécuté

Risque couvert

- Réduit les injections "structurelles" (objets inattendus).
- Réduit les crashes / incohérences.
- Couplé à des bornes de longueur, aide à limiter DoS/ReDoS (comme votre cas Sonar regex).

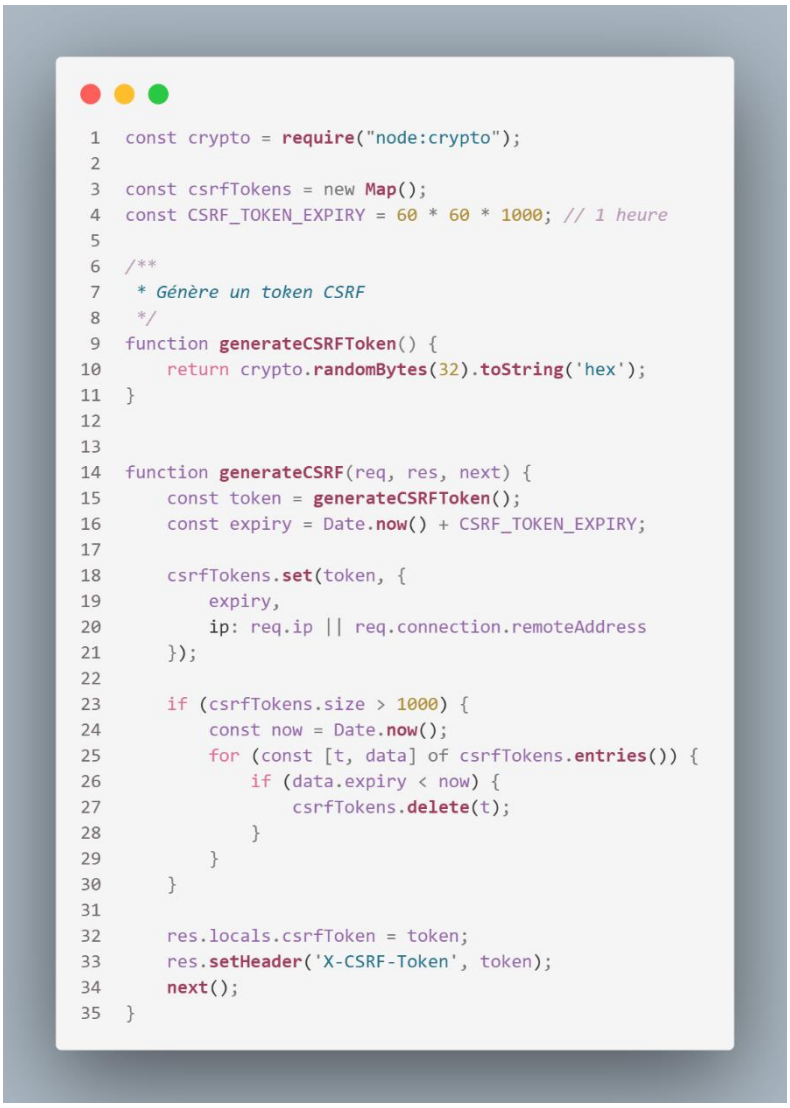
e. Protection CSRF (implémentation custom : token + expiration + usage unique)

Génération de token CSRF (1h) et stockage temporaire

Objectif : empêcher qu'un site tiers déclenche des actions à la place d'un utilisateur (CSRF), surtout pertinent lorsque l'auth utilise des cookies / credentials.

Preuve (implémentation)

- génération d'un token via crypto.randomBytes(32) (token fort)
- expiration configurée à 1 heure
- stockage dans une Map (csrfTokens) avec expiry + IP
- nettoyage opportuniste si la map dépasse 1000 entrées
- token renvoyé au client via header X-CSRF-Token



```
const crypto = require("node:crypto");

const csrfTokens = new Map();
const CSRF_TOKEN_EXPIRY = 60 * 60 * 1000; // 1 heure

/**
 * Génère un token CSRF
 */
function generateCSRFToken() {
    return crypto.randomBytes(32).toString('hex');
}


function generateCSRF(req, res, next) {
    const token = generateCSRFToken();
    const expiry = Date.now() + CSRF_TOKEN_EXPIRY;

    csrfTokens.set(token, {
        expiry,
        ip: req.ip || req.connection.remoteAddress
    });

    if (csrfTokens.size > 1000) {
        const now = Date.now();
        for (const [t, data] of csrfTokens.entries()) {
            if (data.expiry < now) {
                csrfTokens.delete(t);
            }
        }
    }

    res.locals.csrfToken = token;
    res.setHeader('X-CSRF-Token', token);
    next();
}
```

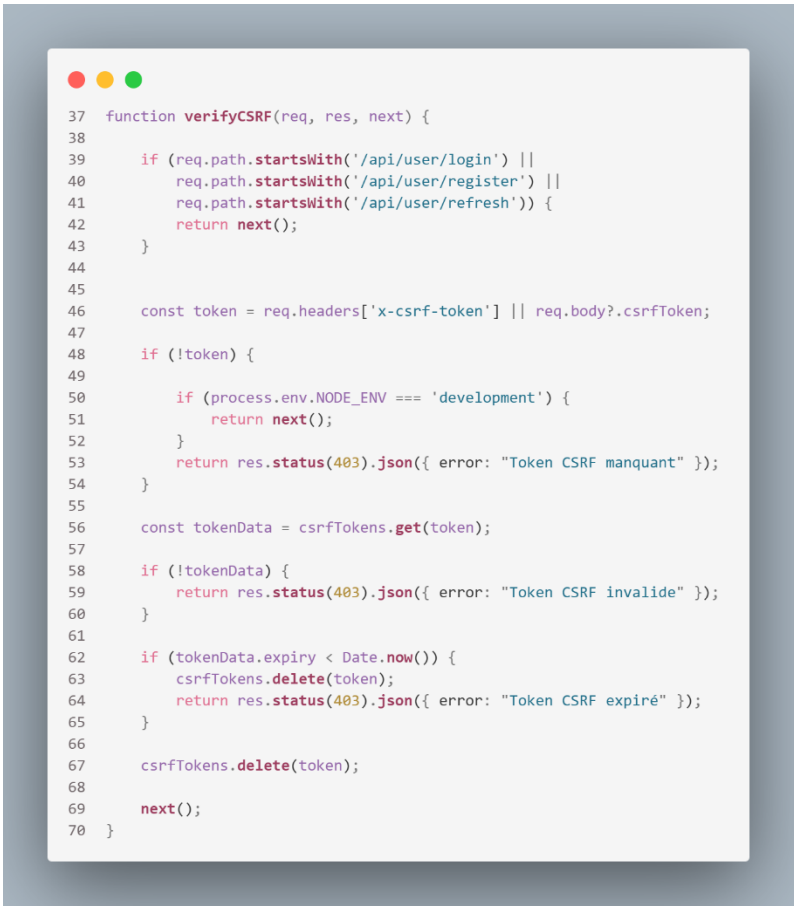
Points précis à mentionner

- Token fort (32 bytes → hex)
- Expiration 1h
- Stockage temporaire en mémoire + cleanup (limite de croissance)

Vérification CSRF (403) + usage unique

Preuve (implémentation)

- certaines routes sont exemptées : /api/user/login, /api/user/register, /api/user/refresh
- token récupéré via header x-csrf-token ou body csrfToken
- si absent/invalide/expiré → 403 avec message explicite
- si OK → suppression du token (csrfTokens.delete(token)) → usage unique



```
function verifyCSRF(req, res, next) {

    if (req.path.startsWith('/api/user/login') ||
        req.path.startsWith('/api/user/register') ||
        req.path.startsWith('/api/user/refresh')) {
        return next();
    }


    const token = req.headers['x-csrf-token'] || req.body?.csrfToken;

    if (!token) {

        if (process.env.NODE_ENV === 'development') {
            return next();
        }
        return res.status(403).json({ error: "Token CSRF manquant" });
    }

    const tokenData = csrfTokens.get(token);

    if (!tokenData) {
        return res.status(403).json({ error: "Token CSRF invalide" });
    }

    if (tokenData.expiry < Date.now()) {
        csrfTokens.delete(token);
        return res.status(403).json({ error: "Token CSRF expiré" });
    }

    csrfTokens.delete(token);

    next();
}
```

Comportement attendu

- en prod : absence token → 403
- token invalide/expiré → 403

- token valide → passe, puis token supprimé (usage unique)

Point de vigilance (à écrire, très “audit”)

- Le bypass en développement (if NODE_ENV===development return next()) est normal pour faciliter les tests, mais doit être assumé : “CSRF strict en production”.
- Stockage en mémoire (Map) : en prod multi-instances, il faudra centraliser (Redis) sinon un token généré sur une instance peut ne pas être reconnu par une autre.
- L’IP est stockée mais pas vérifiée dans l’extrait : si vous ne l’utilisez pas, vous pouvez l’indiquer comme piste d’amélioration (“lier token à IP/session”).

Limites et implications production

- “Limite : stockage en mémoire → en prod multi-instances, nécessiterait Redis / store partagé.”
- “L’IP est enregistrée mais non exploitée dans la version actuelle ; amélioration possible : lier le token à l’IP ou à un identifiant de session.”

f. Monitoring / logging sécurité (accès sensibles)

Logging des accès sensibles (preuve)

Objectif : conserver des traces exploitables en cas d’incident (accès admin, actions sensibles, tentatives suspectes), sans exposer d’informations côté client.

Preuve (implémentation)

Vous loggez un événement SENSITIVE_ACCESS avec :

- timestamp ISO
- userId et role (si présent)
- ressource + action
- IP + user-agent
- méthode HTTP + path

→ Appelée sur les routes admin (CRUD films, suppression utilisateurs) et sur les endpoints de refresh token.
→ But : tracer qui fait quoi (userId/role/IP/path).

```
logSensitiveAccess(req, resource, action) {
    const logData = {
        timestamp: new Date().toISOString(),
        event: 'SENSITIVE_ACCESS',
        userId: req.user?.userId || 'N/A',
        role: req.user?.role || 'N/A',
        resource,
        action,
        ip: req.ip || req.connection.remoteAddress,
        userAgent: req.get('user-agent'),
        method: req.method,
        path: req.path
    };

    console.log(`[SECURITY] 🔒 Accès sensible: ${action} sur ${resource} par utilisateur ${logData.userId} (${logData.role})`);
},
```

Ce que ça apporte

- Traçabilité (qui a fait quoi, quand, depuis où)
- Aide au debug sécurité (corrélation brute force / accès admin)

Point d'amélioration "pro"

- En prod : logger structuré (JSON) + stockage centralisé (Elastic/Grafana Loki) + niveaux (info/warn/error).

g. Protection contre les vulnérabilités OWASP (mapping mesures → risques)

Cette section relie les protections mises en place aux risques OWASP rappelés dans le cours. Pour chaque risque, nous indiquons le scénario d'attaque, la mesure appliquée, et la preuve (extrait de code / capture).

A01 — Broken Access Control (contrôle d'accès insuffisant)

Risque / scénario : un utilisateur non autorisé tente d'accéder à une route sensible (ex : actions admin) ou d'effectuer une action réservée. Mesures mises en place :

- JWT contenant role et userId, utilisé comme base du contrôle d'accès côté API.
- Logging des actions sensibles via logSensitiveAccess(…) (remember: trace "qui fait quoi").

Preuves :

- JWT inclut role dans le payload ({ userId, role }) et est signé.
- Logging "SENSITIVE_ACCESS" contenant userId, role, path, method.

Résultat attendu :

- Sans token valide : accès refusé (401 via middleware d'auth).

- Avec token valide mais rôle insuffisant : accès refusé (403 via RBAC).
- Les tentatives/actions critiques sont traçables via logs.

```
logSensitiveAccess(req, resource, action) {
    const logData = {
        timestamp: new Date().toISOString(),
        event: 'SENSITIVE_ACCESS',
        userId: req.user?.userId || 'N/A',
        role: req.user?.role || 'N/A',
        resource,
        action,
        ip: req.ip || req.connection.remoteAddress,
        userAgent: req.get('user-agent'),
        method: req.method,
        path: req.path
    };

    console.log(`[SECURITY] 🔒 Accès sensible: ${action} sur ${resource} par utilisateur ${logData.userId} (${logData.role})`);
},
```

A02 — Cryptographic Failures (failles cryptographiques / secrets mal gérés)

Risque / scénario :

- mots de passe stockés en clair,
- interception réseau (HTTP),
- secrets faibles ou exposés.

Mesures mises en place :

- Hash des mots de passe avec bcrypt (coût 10).
- Activation HTTPS en local (certificat TLS auto-signé) pour tester un environnement proche prod.

Preuves :

- bcrypt.hash(password, 10) avant stockage + réponse sans champ password.
- Configuration HTTPS via certificat/clé (décrite dans la partie TLS).

Résultat attendu :

- Une fuite de base ne donne pas accès aux mots de passe en clair.
- Les échanges HTTP→HTTPS sont chiffrés (en prod, certificat CA requis).

A05 — Security Misconfiguration (mauvaise configuration)

Risque / scénario :

- CORS trop permissif (*), headers absents, méthodes non limitées.

Mesures mises en place :

- CORS en allowlist (allowedOrigins) avec rejet explicite si origine non autorisée.
- credentials: true + headers/méthodes explicitement listés.
  Preuves :
- callback CORS Non autorisé par CORS
- allowedHeaders et methods définis
  Résultat attendu :
- Un site tiers (origin non listée) ne peut pas appeler l'API depuis un navigateur.
- Les méthodes et headers sont contrôlés, évitant une ouverture inutile.

A07 — Identification & Authentication Failures (authentification/identité)

Risque / scénario :

- brute force sur login,
- sessions trop longues / pas d'expiration,
- auth contournable.
  Mesures mises en place :
- JWT signé + expiration 2h.
- Rate limiting multi-niveaux : global + login (10 tentatives / 15 min).
  Preuves :
- expiresIn: "2h" dans le JWT.
- rateLimit global (100/min) + /auth/login (10/15min).
  Résultat attendu :
- Brute force fortement ralenti et déclenche 429.
- Token expiré → l'accès doit être refusé (middleware JWT).

A08 — Software & Data Integrity Failures / Injections (selon formulation du cours)

Risque / scénario :

- entrées non validées entraînant injections (NoSQL/SQL) ou comportements inattendus.
- exemple concret chez vous : risque autour de findOne + critères issus du client (signalé par Sonar).
  Mesures mises en place :
- Validation centralisée des entrées avec express-validator + rejet 422.

- Bornage des entrées (ex. limiter longueur avant regex) pour éviter entrées “pathologiques”.
  Preuves :
- middleware validate renvoyant 422.
- correction hotspot Sonar “regex DoS” par limitation de caractères (comme tu l’as décrit).
  Résultat attendu :
- les requêtes invalides sont bloquées avant logique métier.
- réduction des risques d’injection “structurelle” et de DoS via entrées démesurées.

CSRF — Cross-Site Request Forgery (souvent évalué même si plus “Top 10 classique”)

Risque / scénario :

- un site externe déclenche une requête à la place de l’utilisateur (si cookies/credentials sont utilisés).
  Mesures mises en place :
- Token CSRF fort (32 bytes), expirant à 1h, à usage unique.
- Vérification sur requêtes (403 si manquant/invalide/expiré).
  Preuves :
- crypto.randomBytes(32) + CSRF_TOKEN_EXPIRY = 1h
- csrfTokens.delete(token) après validation (usage unique)
- erreurs 403 “manquant / invalide / expiré”
  Résultat attendu :
- une requête mutative sans token CSRF est rejetée en production.

h. Récapitulatif “preuves → protection”

- **bcrypt (10 rounds) :** mots de passe non récupérables en clair.
- **JWT signé (2h) :** auth stateless, support du rôle.
- **rate limit :** global 100/min + login 10/15min → brute force ralenti (429).
- **CORS allowlist + credentials :** uniquement origines autorisées, headers/méthodes limités.
- **validate (422) :** validation centralisée, format d’erreurs standard.
- **CSRF (1h, usage unique, 403) :** prévention d’actions cross-site en prod.
- **logging accès sensible :** audit trail minimal pour actions critiques.

6. Outils de sécurité

a. SonarQube (analyse statique)

Objectif

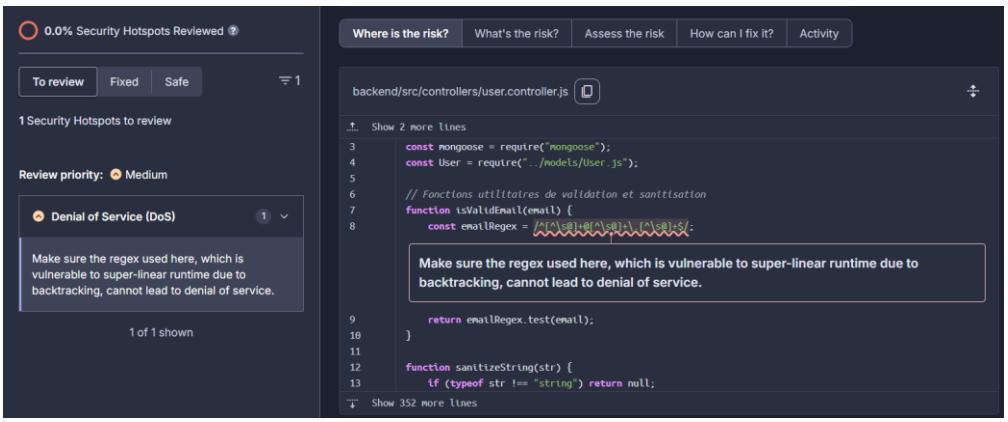
SonarQube permet d'identifier des vulnérabilités, "security hotspots" et mauvaises pratiques directement dans le code (analyse statique), afin de corriger en amont des failles exploitables (injections, DoS, erreurs de validation, etc.).

Méthode

- Analyse du projet via SonarQube pendant le développement et lors d'une Pull Request.
- Revue des Security Hotspots et des alertes liées aux contrôleurs (entrée utilisateur).

Résultats principaux observés

1. Security Hotspot – Regex vulnérable à une attaque DoS (ReDoS)
    - SonarQube a signalé qu'une regex d'email pouvait provoquer un temps d'exécution super-linéaire à cause du backtracking, et donc mener à une Denial of Service si un attaquant envoie une chaîne volontairement "pathologique".



    - Correction appliquée : ajout d'une limite de longueur sur l'entrée avant exécution de la regex (borne de caractères).
    - Bénéfice sécurité : réduction du risque de blocage CPU sur une route exposée.
2. Contrôleur User – risque lié à l'utilisation de findOne/findById avec entrées utilisateur

- SonarQube a attiré l'attention sur des appels de recherche utilisateur (ex. findOne) qui peuvent devenir dangereux si on injecte des objets inattendus ou des opérateurs (ex : $ne, $gt) dans les champs attendus.
- Correction appliquée :
  - validation stricte des types (string attendu, etc.)
  - sanitisation (trim, limitation de longueur, format email…)
  - logique "whitelist" : on ne consomme que les champs attendus, jamais un objet complet non maîtrisé.
- Bénéfice sécurité : empêche des filtres "déguisés" (ex $ne:null) ou des structures non prévues.

3. Bug de validation de types
   - SonarQube a détecté que certaines validations de types n'étaient pas correctement appliquées (types pas vérifiés comme prévu dans certains chemins).
   - Correction : durcissement des contrôles de type + centralisation via le middleware de validation (422).

Conclusion SonarQube
Les retours SonarQube ont été utiles pour corriger des risques concrets (DoS par regex, injection via entrées non strictes, validation type), et pour justifier des mesures "défense en profondeur" (bornage/validation avant logique métier).

b. OWASP ZAP (analyse dynamique)

Objectif

OWASP ZAP permet d'analyser l'application "comme un attaquant" (analyse dynamique) et de remonter des alertes basées sur l'observation des réponses HTTP et de la configuration côté client/serveur (headers, cache, cookies, exposition de données, etc.).

Méthode

- Scan passif (baseline) sur l'URL locale de l'application (ex : https://localhost:3000/).
- Analyse des alertes remontées (niveau de risque, niveau de confiance, endpoints concernés).

- Vérification de l'impact réel sur l'application (présence de données sensibles, logique d'authentification et stockage des tokens côté client).

Alerte observée : "Re-examine Cache-control Directives" (Informational / Low confidence)

- Description (résumé) : certaines réponses peuvent être mises en cache par le navigateur ou des proxies si les directives Cache-Control sont absentes ou insuffisantes.
- Impact potentiel : faible dans l'absolu, mais peut devenir important si des données sensibles sont cachées (profil, réponses d'auth, données admin), notamment sur un poste partagé ou via un proxy.

Correctifs / mesures appliquées (réduction du risque)

Même si l'alerte est "informatif", nous avons appliqué des mesures côté client et côté session afin de réduire fortement le risque lié à une éventuelle conservation locale de données d'authentification :

1) Stockage des tokens côté client (réduction exposition)

- Access Token : stocké uniquement en mémoire (via configuration Axios / runtime), jamais en localStorage ni sessionStorage.
    - Objectif : éviter un token persistant facilement exfiltrable en cas de XSS et limiter la persistance des identifiants côté navigateur.
- Refresh Token : stocké en sessionStorage, donc supprimé à la fermeture de l'onglet.
    - Objectif : limiter la durée de vie côté client et réduire l'exposition sur poste partagé.

2) Renouvellement automatique contrôlé (meilleure gestion des expirations)

- Mise en place d'un intercepteur Axios : en cas de réponse 401 (access token expiré), le front déclenche automatiquement la route de refresh, récupère un nouvel access token puis rejoue la requête initiale.
- Au démarrage de l'application : récupération du refresh token depuis sessionStorage pour réinitialiser proprement la session (si présent).

3) Déconnexion renforcée / révocation serveur

- "Logout amélioré" : le refresh token est révoqué côté serveur (suppression/invalidation) et supprimé côté client.

- Objectif : empêcher la réutilisation d'un refresh token après déconnexion.

4) Rotation et limitation côté serveur (contrôle des sessions)

- Rotation/éviction des refresh tokens avec une limite de 5 tokens actifs par utilisateur, afin de réduire la fenêtre d'exposition en cas de fuite d'un ancien refresh token.

Conclusion OWASP ZAP

L'alerte ZAP sur Cache-Control est informative et à faible confiance, mais elle met en évidence un point classique : il faut éviter de conserver des informations sensibles en cache. Dans notre cas, nous avons réduit le risque principalement via une gestion stricte des tokens côté client (pas de localStorage, access token en mémoire, refresh token en sessionStorage), un refresh automatique contrôlé, et une révocation/rotation des refresh tokens côté serveur, ce qui limite fortement l'impact d'une conservation locale involontaire.

Amélioration recommandée (production) : ajouter explicitement des headers anti-cache (Cache-Control: no-store, Pragma: no-cache) sur les routes sensibles (profil, admin, réponses d'auth), pour durcir le comportement navigateur/proxy même si l'alerte est "informative".

c. npm audit (dépendances)

Objectif
npm audit vérifie si des dépendances contiennent des vulnérabilités connues (CVE/advisories) et propose des mises à jour/corrections.

Résultat observé

- 4 vulnérabilités détectées : 2 moderate + 2 high.
- Vulnérabilités notables (selon le rapport) :
    - body-parser (moderate) : DoS possible dans certains cas d'URL encoding.
    - glob (high) : risque d'injection de commande dans certains usages CLI (souvent via outils de test/dev).
    - js-yaml (moderate) : prototype pollution dans certaines fonctions de merge.

- jws (high) : vérification HMAC incorrecte (important car lié à l'écosystème JWT).

Actions appliquées / recommandées

- Exécuter npm audit fix puis re-tester.
- Mettre à jour les dépendances qui tirent jws (souvent via jsonwebtoken) afin d'obtenir une version corrigée.
- Vérifier si certaines vulnérabilités sont dans des dépendances de dev (ex : jest) : si oui, l'impact prod est moindre, mais on corrige quand même pour garder une chaîne de build propre.

Conclusion npm audit
La correction/maîtrise des dépendances fait partie de la sécurité applicative : même avec un code sûr, une dépendance vulnérable peut réintroduire un risque (notamment autour des libs JWT).

7. Gestion des secrets & configuration

a. Conformité attendue

- aucune donnée sensible en clair
- séparation données / secrets / configuration
- variables d'environnement + .env + .env.example
- aucun secret dans le front

b. Mesures proposées/attendues

- .env ignoré par Git, .env.example fourni (sans valeurs)
- secrets : JWT_SECRET, REFRESH_SECRET, URLs DB, etc.
- logs : ne jamais logger tokens / mots de passe
- configuration par environnement (dev/test/prod)

c. Guide de lancement (environnement de développement)

Prérequis

- Node.js + npm installés

- Bases de données accessibles (si nécessaires au run)
- Projet cloné en local

1) Installation des dépendances

- À la racine du projet :
    - npm install
- Dans backend/ :
    - npm install
- Dans frontend/ :
    - npm install

2) Configuration des variables d'environnement (backend)
Créer un fichier backend/.env (non versionné) avec les variables suivantes :

- JWT_SECRET=<secret_long_aleatoire_32+_caracteres>
- KEYPATH=../security/server.key
- CERTPATH=../security/server.cert
- ALLOWED_ORIGINS=https://localhost:5173,https://localhost:3000
- NODE_ENV=development

Remarque : le contenu exact des secrets (JWT_SECRET, clés TLS...) n'est pas présent dans le rapport et ne doit pas être commit (bonne pratique de sécurité).

3) Mise en place du certificat TLS local
À la racine du projet, créer un dossier security/ contenant :

- server.key (clé privée)
- server.cert (certificat)

Ces fichiers servent uniquement au HTTPS en local (certificat auto-signé). Ils ne sont pas approuvés par une CA (normal en dev), mais permettent de tester l'application en HTTPS.

Option recommandée (dev) : génération via OpenSSL

- Générer une clé privée + un certificat auto-signé (TLS) avec OpenSSL
- Placer server.key et server.cert dans security/

4) Lancement

- Depuis la racine :

- npm run dev

Si le projet ne démarre pas via la racine, lancer séparément :

- Backend : npm run dev (dans backend/)
- Frontend : npm run dev (dans frontend/)

- Les fichiers security/server.key, .env et tout secret ne sont pas versionnés (gitignore)
- Le certificat est auto-signé : OK en dev, mais en prod il faut un certificat CA + config TLS stricte

8. Utilisation de l'IA

Le projet impose une transparence complète : l'IA est un outil, pas un auteur.

Outils utilisés

- ChatGPT
- Cursor

Pour quoi

- aide à la rédaction du compte rendu
- aide au développement du front afin de respecter la deadline

Gains

- gain de temps (structuration, reformulation, aide ponctuelle)

Pertes / limites

- rendu parfois "moins humain" si on ne retravaille pas le texte
- risque de suggestions génériques nécessitant validation technique
- nécessité de garder un regard critique (tests + revue de code)

9. Limites, difficultés, axes d'amélioration

Limites actuelles :

- refresh token / révocation perfectible
- durcissement headers/CSP selon le front
- couverture de tests à augmenter
- monitoring/alerting à renforcer

Axes d'amélioration :

- tests d'intégration sécurité (RBAC, injections, bruteforce)
- rotation refresh tokens + blacklist/hash en DB
- politique de logs structurés (corrélation, niveau, rétention)
- durcissement TLS + configuration prod plus stricte
- revue de code pour plus de maintenabilité

10. Conclusion

L'API Cinema remplit les objectifs fonctionnels (catalogue, recherche, auth, notes, administration) tout en intégrant les exigences de sécurisation : communications sécurisées (TLS/headers), authentification robuste (bcrypt + JWT + expiration + protections), traitement des vulnérabilités OWASP, gestion saine des secrets/configuration, et utilisation d'outils d'audit (SonarQube, OWASP ZAP, npm audit) avec analyse et corrections/justifications.