ChatGPT

Revue de Sécurité Globale pour la Plateforme Carnet de Santé (PERN)

Contexte : Cette application web de dossier médical numérique manipule des données sensibles (PII/PHI). Elle utilise un backend Node/Express avec PostgreSQL, un frontend React (Vite) + Tailwind, et un système d'authentification JWT (tokens d'accès courts en mémoire et tokens de rafraîchissement longs en HttpOnly cookie). Plusieurs rôles utilisateur existent (Patient, Médecin, Admin). L'enjeu est de garantir une **sécurité maximale** en production.

Nous présentons d'abord un **aperçu global** des risques et contre-mesures, puis des **recommandations détaillées** avec bonnes pratiques (DO) et pièges à éviter (DON'T), accompagnés d'exemples de code et d'analogies pédagogiques.

Aperçu Global des Risques & Mesures

1. Broken Access Control (IDOR) – *Contrôles d'accès insuffisants* : Risque que des utilisateurs accèdent à des données qui ne leur appartiennent pas (ex: un patient accédant aux documents d'un autre via une URL modifiée). **Mesure :** Implémenter systématiquement des vérifications côté serveur de l'autorisation, en fonction de l'identité et du rôle de l'utilisateur connecté. Par exemple, s'assurer qu'un patient ne puisse demander que **ses** propres ressources, et qu'un médecin ne voie que les données de **ses** patients autorisés. Chaque requête doit être filtrée selon l'ID utilisateur et le rôle associé, afin d'éviter les failles de type IDOR (Insecure Direct Object Reference) [1].

2. SQL Injection (SQLi) – *Injection SQL* : Risque d'injection de code SQL via des entrées utilisateur mal gérées, pouvant compromettre la base de données. **Mesure :** Utiliser exclusivement des **requêtes paramétrées** avec le module pg (e.g. `pool.query("SELECT ... WHERE id=$1", [valeur])`), ce qui prévient l'exécution de code malveillant dans la requête [2]. Éviter de construire des morceaux de requête en concaténant des chaînes fournies par l'utilisateur. Si une requête nécessite des éléments dynamiques (nom de table, colonne tri, etc.), restreindre ces valeurs à une liste blanche côté code ou utiliser des utilitaires d'échappement comme **pg-format** [3]. En un mot : **TOUJOURS séparer le code SQL des données utilisateur**.

3. Cross-Site Scripting (XSS) – *Scripts intersites malveillants* : Risque que des scripts injectés par un acteur malveillant soient stockés ou reflétés dans l'application (via un champ profil, nom de fichier, compte-rendu médical, etc.), et exécutés dans le navigateur d'un autre utilisateur. **Mesure : Nettoyer et valider** toutes les entrées contenant du texte libre avant stockage (suppression/échappement des balises HTML dangereuses côté backend), et **sécuriser le rendu** côté frontend. React échappe par défaut le contenu inséré via JSX, empêchant l'exécution de `<script>` injecté [4]. Il faut donc s'y tenir : ne jamais insérer du HTML brut non maîtrisé dans le DOM (éviter `dangerouslySetInnerHTML` à moins d'une nécessité absolue, et seulement après avoir désinfecté le contenu) [5]. Pour tout contenu riche (par ex. texte formaté) provenant de l'utilisateur, utiliser une bibliothèque de **sanitization** (ex: DOMPurify) afin de filtrer les éléments potentiellement dangereux avant de les afficher [6] [7].

4. Insecure File Uploads – *Téléversement de fichiers non sécurisé* : Risque lié aux fichiers que les utilisateurs peuvent uploader (PDF, images médicales…). Stocker les fichiers en **BLOB dans la base** offre

un contrôle d'accès unifié (passe par la DB) mais peut alourdir celle-ci et compliquer la mise à l'échelle. Un stockage **externe (S3 ou autre)** décharge la DB, mais requiert de bien contrôler les URL d'accès (liens signés, droits d'accès) pour éviter des fuites. **Mesures :** *Validation stricte* des fichiers uploadés : n'autoriser que certaines extensions **et** vérifier le type réel du fichier. Ne pas se fier uniquement à l'extension ou au MIME déclaré par le client (facilement falsifiables) [8]. Par exemple, un `.html` déguisé en `.pdf` ne doit pas passer. Utiliser `file.mimetype` **ET** inspecter le contenu binaire (via une lib comme `file-type`) pour s'assurer que le fichier est bien du type attendu [9]. Renommer les fichiers stockés avec un identifiant interne (éviter d'utiliser directement le nom original de l'utilisateur sans nettoyage, pour prévenir l'exécution s'il contient des caractères spéciaux). **Servir les fichiers de manière sûre** : toujours définir `Content-Type` sur le type attendu (ex. `application/pdf`) et ajouter `Content-Disposition: attachment` pour forcer le téléchargement plutôt que l'exécution inline du contenu [10]. Envoyer aussi `X-Content-Type-Options: nosniff` pour empêcher le navigateur de deviner un type différent [10]. Ces en-têtes garantissent qu'un fichier potentiellement dangereux (ex: un PDF piégé avec du script) ne pourra pas compromettre l'application en se comportant comme du HTML actif. Enfin, rejeter ou scanner les fichiers trop volumineux ou suspects, et envisager un scan antivirus pour les documents uploadés dans un contexte santé.

5. Authentication & Session Management – *Authentification et gestion de session* : Le système JWT mis en place (access token court 15min en mémoire React, refresh token 7j en HttpOnly) est un bon schéma de base, mais comporte des points d'attention : **Tokens volés ou réutilisés**. Un token JWT est un **jeton au porteur** : quiconque le possède peut l'utiliser jusqu'à expiration [11]. Il faut donc minimiser les chances qu'il soit volé et limiter les dégâts s'il l'est. **Mesures :** Utiliser toujours **HTTPS** (chiffrement en transit) pour empêcher l'interception. La durée de vie courte (15 min) de l'access token est judicieuse [12] – cela réduit la fenêtre d'exploitation en cas de vol. Stocker l'access token en mémoire (contexte React) évite de l'exposer dans des stockages persistants du navigateur (localStorage) où un script malveillant pourrait le lire ; toutefois en cas de XSS, un attaquant pourrait potentiellement appeler des actions à votre insu. **Prévenir le vol de refresh token :** le cookie HttpOnly + SameSite=Strict est bon, car inaccessible en JS et non envoyé sur des requêtes cross-site. Cependant, un XSS pourrait quand même déclencher une requête AJAX vers `/api/auth/refresh` (même origine) et obtenir un nouveau token – il faut donc **éviter les failles XSS en premier lieu** (voir point 3). Pour renforcer la sécurité des refresh tokens, appliquez le principe de **rotation** : **à chaque utilisation d'un refresh token pour obtenir un nouveau token, émettre un nouveau refresh token et invalider l'ancien** [13]. Ainsi, un token volé ne sera utilisable qu'une fois, empêchant un attaquant de le réutiliser (stratégie anti-replay) [14]. Mettez en œuvre un suivi des refresh tokens en base (par ex., stocker un identifiant de « famille » de tokens et détecter si un ancien token est présenté après qu'il a été échangé – dans ce cas invalider toute la famille) [15] [16]. Si la rotation est complexe à implémenter manuellement, au minimum **versionnez** les tokens : associez à chaque utilisateur un champ `tokenVersion` qui est inclus dans le payload JWT. En cas de logout ou de compromission suspectée, incrémentez `tokenVersion` en base – ainsi, tous les anciens tokens (contenant l'ancienne version) seront refusés lors de la vérification. **À faire également :** invalider le refresh token côté serveur lors du logout (suppression en BD) pour qu'il ne puisse plus servir. Veiller à ce que l'API de logout soit appelée sur **tous** les appareils/tabs si l'utilisateur se déconnecte (ex: en front, effacer le contexte et éventuellement informer les autres onglets via le stockage local ou un mécanisme de broadcast). **Protection contre les replays :** inclure un identifiant unique (JWT ID, `jti`) dans chaque token et garder une liste de jti déjà utilisés pour détecter des réutilisations anormales peut être un plus. Enfin, **ne jamais stocker** les tokens JWT (surtout refresh) en clair en BD – stockez une empreinte hash si possible, de sorte que si la base est compromise, les tokens ne le soient pas immédiatement.

⚠ *Note analogique :* Considérez les tokens comme des **clés d'accès**. L'access token est une clé temporaire (badgé visiteur à durée limitée) – courte validité pour limiter l'usage abusif. Le refresh token est une clé maître plus puissante ; on la garde sous haute protection (HttpOnly cookie). Si l'une fuit, il

faut immédiatement **changer les serrures** (rotation/invalidations) pour que l'intrus ne puisse pas ouvrir la porte plus d'une fois.

6. Security Misconfiguration – *Mauvaise configuration de sécurité* : Risque que des paramètres par défaut ou des configurations insuffisantes laissent des portes ouvertes (ex: en-têtes HTTP manquants, CORS mal réglé, messages d'erreur trop verbeux). **Mesures :** Utiliser **Helmet** sur le serveur Express pour ajouter les en-têtes de sécurité essentiels. Par défaut, Helmet active une douzaine d'en-têtes utiles : Content-Security-Policy (CSP) pour limiter les sources de scripts, **Strict-Transport-Security** pour forcer l'utilisation de HTTPS, **X-Frame-Options** pour empêcher le clickjacking, **X-Content-Type-Options** pour éviter le *MIME sniffing*, **Referrer-Policy** pour réduire les fuites de référent, etc. [17] [18]. Il supprime aussi `X-Powered-By` (qui divulgue Express) et désactive le vieillissant `X-XSS-Protection`. Il convient d'examiner ces paramètres et éventuellement de personnaliser CSP (en autorisant explicitement les domaines nécessaires pour scripts, fonts, etc.). **CORS :** Configurer le middleware CORS de façon restrictive : spécifier précisément l'origine autorisée (l'URL du frontend en production) au lieu d'autoriser tout (`*`), surtout que l'appli utilise des cookies de rafraîchissement. En effet, avec `credentials: true`, on **ne peut pas** utiliser `Access-Control-Allow-Origin: *` [19] – il faut lister le domaine exact du frontend. Assurez-vous également que les méthodes HTTP et en-têtes autorisés sont correctement configurés (GET, POST, etc., et e.g. Authorization, Content-Type en-têtes). **Rate Limiting :** Mettre en place une limitation de débit sur les endpoints sensibles, en particulier `/api/auth/login` (pour éviter les brute force sur les mots de passe) et `/api/auth/refresh` (pour éviter des attaques par déni de service ou des essais de tokens). Par exemple, limiter à ~5 tentatives de login par IP par 15 minutes, avec un délai progressif en cas d'échecs répétés. Des modules comme `express-rate-limit` peuvent faciliter cela. **Messages d'erreur :** Veiller à ne pas divulguer d'informations techniques sensibles dans les réponses d'erreur. En production, désactiver le stack trace Express par défaut. Fournir des messages génériques aux clients (ex: "Erreur serveur, veuillez réessayer" pour un code 500, ou "Identifiants invalides" pour un login raté, sans préciser si l'utilisateur ou le mot de passe est incorrect). Logguez en interne les détails, mais ne les renvoyez pas au client. De même, un endpoint de recherche d'utilisateur ne devrait pas révéler "email non trouvé" vs "mot de passe incorrect" de façon distincte, car cela aide un attaquant à vérifier l'existence d'un compte. **Administration des configurations :** s'assurer que les fichiers sensibles (.env, clés) ne sont pas exposés (ex: ne pas les servir statiquement). Vérifier que l'application est bien en mode `NODE_ENV=production` (certains frameworks changent leur comportement de sécurité en prod). Enfin, pensez à configurer correctement les **politiques de cookies** : HttpOnly et SameSite=Strict sont déjà mis, ajoutez `Secure` (ce qui est fait) pour forcer HTTPS, et envisagez `SameSite=None` + `Secure` si le front et le back sont sur des domaines différents et qu'on **doit** autoriser le cookie cross-site (dans votre cas, SameSite=Strict sur le refresh token empêche son envoi via un domaine tiers, ce qui est approprié puisque seule votre SPA sur le même domaine devrait y accéder).

7. Vulnerable Dependencies – *Dépendances vulnérables* : Risque qu'une librairie tierce utilisée (backend ou frontend) présente une faille de sécurité connue. Étant donné la stack Node/React, il y a de nombreuses dépendances (Express, pg, React, etc.) qu'il faut surveiller. **Mesures :** Mettre en place un processus de **veille et de mise à jour régulière** des packages. Utilisez `npm audit` pour scanner les vulnérabilités connues dans vos dépendances (côté backend **et** frontend) – et appliquez les correctifs (`npm audit fix`) lorsque c'est possible. Activez **GitHub Dependabot** sur le repo : il surveillera vos `package.json` et vous alertera (voire proposera des PRs automatiques) en cas de découverte d'une faille sur l'une des versions utilisées [20] [21]. Des outils comme **Snyk** peuvent également être intégrés pour une analyse de sécurité continue plus poussée (y compris pendant le développement CI/CD). L'objectif est de réduire le délai entre la publication d'un correctif de sécurité pour une lib et son déploiement dans votre application. **Bonnes pratiques :** Éviter les packages non maintenus ou douteux, privilégier les librairies réputées. Surveillez le changelog de dépendances critiques (ex: Express, pg) pour appliquer les mises à jour mineures de sécurité. En front, faites attention aux

packages npm que vous ajoutez – tout code tiers est un vecteur potentiel (ex: une lib compromise par une dépendance malveillante). Un audit manuel ponctuel des dépendances (par exemple, repérer les packages qui n'ont pas été mis à jour depuis des années ou avec des warnings connus) peut aider. Automatiser ces scans (via un pipeline CI qui fail en cas de vulnérabilités critiques non résolues, par ex.) ajoute une couche de protection.

8. Monitoring & Anomaly Detection – *Journalisation et détection d'anomalies* : Enfin, une sécurité solide s'accompagne de **surveillance active** : détecter rapidement les comportements suspects peut prévenir ou limiter les dommages d'une attaque en cours. **Mesures :** Implémenter des **logs d'audit** pour les actions sensibles : connexions (réussies et échouées), créations/suppressions de comptes, accès aux dossiers médicaux, téléchargements de documents, modifications de données critiques, etc. Ces logs doivent enregistrer l'horodatage, l'utilisateur concerné, l'action, et l'adresse IP ou tout identifiant d'origine. Ensuite, mettez en place des **alertes sur patterns anormaux** : par exemple, si un même IP ou compte engendre 10 échecs de login en 5 minutes, cela peut déclencher une alerte (et temporairement bloquer l'IP ou le compte). De même, si un utilisateur téléverse une quantité inhabituelle de fichiers en peu de temps, ou si un médecin consulte un volume anormalement élevé de dossiers de patients sans raison, ce sont des indicateurs d'un éventuel abus ou compte compromis. Vous pouvez utiliser des solutions de monitoring applicatif : intégrer vos logs à un système de type **SIEM** (Security Information and Event Management) tel que Datadog, Splunk, Elastic Stack (ELK) ou autre, qui permet de définir des règles de corrélation et de recevoir des alertes en cas d'incident. Pour une approche plus légère, même de simples scripts ou jobs cron analysant les fichiers de log et envoyant un email d'alerte en cas d'événement suspect peuvent aider. **Auditabilité :** en milieu santé (conformité type HIPAA), il est souvent requis de garder une trace de qui a accédé à quelle donnée et quand. Assurez-vous donc que chaque consultation de données patient par un médecin est tracée (sans exposer le contenu, mais "Dr X a consulté le dossier du patient Y le 05/07/2025 à 10:30"). Cela permet non seulement de dissuader les usages inappropriés (sachant que c'est tracé), mais aussi de mener des investigations post-incident le cas échéant.

⚠ *Analogiquement,* si l'application est un hôpital numérique, il faut non seulement des **gardiens aux portes** (contrôles d'accès, authentification forte), des **caméras de sécurité** (logs d'audit), mais aussi un **agent de sécurité qui patrouille** (analyses automatiques des logs à la recherche d'anomalies) et un **plan d'urgence** (procédure de réponse en cas de détection d'un incident, ex: invalider toutes les sessions si une compromission majeure est détectée).

Recommandations Détaillées par Thème (avec DO/DON'T & Exemples)

Nous approfondissons ci-dessous chaque point avec des exemples de code Node/Express ou React correspondants, afin d'illustrer concrètement les bonnes pratiques à adopter et les écueils à éviter.

1. Broken Access Control (IDOR) – Contrôle d'accès rigoureux

Les **failles de contrôle d'accès** se manifestent souvent par des IDOR : un utilisateur devine ou modifie un identifiant dans une URL ou un formulaire et accède à une ressource qui ne lui appartient pas. Par exemple, sans protection, un patient authentifié ID=10 pourrait appeler l'API `/api/patients/11/documents` et voir les documents du patient 11. Ce type de vulnérabilité est très courant si le backend ne vérifie pas l'autorisation au-delà du simple fait d'être connecté [1].

DO:

- **Vérifier l'identité et le rôle** pour chaque requête accédant à une ressource **multi-utilisateurs**. Par exemple, pour une route `/api/patients/:id`, s'assurer que :
- Si l'utilisateur est de rôle `patient`, alors `:id` correspond à son propre ID, sinon retour 403.
- Si c'est un `médecin`, vérifier qu'il a le droit de voir le patient `:id` (par exemple, qu'il est son médecin traitant ou qu'il a un rendez-vous avec lui). Cela peut nécessiter une requête supplémentaire en base (ex: vérifier dans la table des rendez-vous ou d'affectation) avant de retourner les données.
- Si c'est un `admin`, selon la politique définie, peut-être n'autoriser que certaines actions (dans votre cas, l'admin ne devrait pas accéder aux données médicales des patients – donc même si techniquement possible, appliquer le principe de moindre privilège et refuser l'accès aux documents patients pour le rôle admin).
- **Utiliser des middlewares d'autorisation centralisés** : cela semble déjà amorcé dans votre code. Par exemple, vous avez un middleware `authenticate` (qui extrait `req.userId` et `req.userRole` du JWT), puis `authorize("patient")` ou `authorize(["medecin","admin"])` pour filtrer par rôle, et enfin des middlewares comme `restrictPatientToSelf("id")` ou `checkDocumentOwnership`. C'est une bonne approche modulaire. Assurez-vous simplement que ces middlewares couvrent bien tous les cas :
- `restrictPatientToSelf` doit être appliqué partout où un patient ID apparaît dans l'URL et que seul ce patient devrait y accéder. Votre implémentation actuelle ne restreint pas les médecins/admin (elle fait `if role===patient then compare ids` [22]). C'est correct pour empêcher un patient X d'accéder à Y. **Mais pour les médecins**, vous aurez besoin d'un contrôle supplémentaire ailleurs : par exemple, un médecin ne devrait accéder aux documents d'un patient que s'il est le médecin de ce patient. Il serait pertinent d'ajouter un middleware du genre `restrictDoctorToPatient` qui vérifie via une jointure (table de rendez-vous ou de suivi) que `req.userId` (médecin) est bien associé au patient `req.params.patient_id`. Actuellement, `restrictPatientToSelf` n'empêche pas un médecin non autorisé d'accéder à n'importe quel patient, car il fait `if medecin then next()` [22]. Donc implémentez un check supplémentaire, par ex:

```
// Middleware d'autorisation pour médecin sur une ressource patient
const ensureDoctorHasPatient = async (req, res, next) => {
  if (req.userRole !== 'medecin') return next(); // ne concerne que les
médecins
  const patientId = Number(req.params.patient_id || req.params.id);
  if (!patientId) return res.status(400).send("Patient ID manquant");
  // Requête en base : ce médecin a-t-il accès à ce patient ?
  const result = await pool.query(
    "SELECT 1 FROM rendez_vous WHERE patient_id=$1 AND medecin_id=$2
LIMIT 1",
    [patientId, req.userId]
  );
  if (result.rowCount === 0) {
    return res.status(403).json({ message: "Accès non autorisé pour ce
patient" });
  }
  next();
};
```

Vous pourriez adapter la logique selon votre modèle (ex: table `PatientsMedecins` ou consultation). L'idée est de restreindre l'horizontal **aussi pour les médecins**, pas uniquement entre patients.

- **Filtrer côté base de données si possible** : Une autre couche de protection consiste à inclure la condition d'appartenance directement dans la requête SQL, en fonction du rôle. Par ex., pour renvoyer les documents d'un patient, au lieu de `SELECT * FROM documents WHERE patient_id = $1`, on peut faire :

```
SELECT * FROM documents
WHERE id = $1
  AND (
        (patient_id = $2)              -- si le token correspond au
patient lui-même
     OR (medecin_id_createur = $3)     -- ou s'il y a un champ medecin
propriétaire du doc qui correspond
     OR ($4 = true)                   -- ou une condition booléenne
autorisant l'admin, passée en param.
  );
```

... en passant en paramètres $2 l'ID de l'utilisateur connecté (si patient), $3 l'ID s'il est médecin, $4 un booléen `isAdmin`. Ce genre de requête un peu complexe garantit que même si la logique applicative avait un défaut, la DB n'enverrait rien si les critères ne sont pas remplis. **Néanmoins**, cela complexifie le code SQL et peut être moins lisible – à peser selon les cas. Un contrôle applicatif clair via middleware est souvent plus simple à raisonner.

- **Penser aux deux dimensions d'accès : horizontale et verticale** :
- *Horizontale* : empêcher un utilisateur d'un rôle donné d'accéder aux données d'un autre utilisateur de même niveau (ex: patient vs patient, médecin vs médecin). C'est le cas typique d'IDOR.
- *Verticale* : empêcher l'escalade de privilège – ex: qu'un patient appelle une route d'admin ou qu'un médecin crée un compte admin. Vous avez déjà `authorize(role)` qui couvre cela. Vérifiez toutes les routes admin/médecin pour s'assurer qu'un utilisateur malintentionné ne peut pas accomplir une action hors de son rôle via un simple changement de paramètre.
- **Protéger même les routes "cachées" du frontend** : Ne jamais présumer que le client n'affichera pas tel bouton ou tel lien. Un attaquant pourra toujours fabriquer manuellement des requêtes HTTP. Donc chaque contrôle d'accès doit être **côté serveur**. (Exemple : même si l'UI n'affiche pas la liste de tous les patients à un médecin lambda, celui-ci pourrait appeler `/api/patients?search=` via Postman – il faut que l'API filtre dans la réponse ou refuse).

DON'T:

- **Ne pas faire confiance aux données du client pour la sécurité.** Évitez par exemple d'accepter du client un champ dans le body censé correspondre à un userId et de l'utiliser sans vérification. Imaginons une route `POST /api/reports` où un médecin soumet un compte-rendu avec `patient_id` dans le JSON. Il pourrait tricher et mettre l'ID d'un patient qui n'est pas le sien. Pour contrer cela, ignorez ce champ et sur le serveur, réattribuez-le en fonction du contexte (ex: si le médecin ne peut écrire que pour le patient actuellement consulté, assurez-vous qu'il a bien ce patient).
- **Ne comptez pas sur l'obfuscation d'identifiants.** Parfois, des développeurs "cachent" les vrais IDs (ex: en les hachant ou utilisant des UUID complexes) pensant que ça suffit à éviter IDOR. Ce

n'est pas une vraie sécurité : même un ID UUID peut être deviné ou leaké. Seule la vérification d'autorisation compte [23] [24].

- **Ne laissez pas d'endpoint non protégé listant toutes les données.** Par exemple, un `GET /api/patients` devrait **toujours** être restreint (chez vous, seuls médecin/admin peuvent l'appeler). Assurez-vous qu'aucune route ne retourne plus de données que nécessaire selon le rôle (principe du moindre privilège).
- **Évitez d'outrepasser les contrôles en debug.** (Ex: un middleware temporairement commenté, ou un bypass "si user.isAdmin" pour aller plus vite en test). Ce genre de porte dérobée finit souvent par rester par erreur en prod – à proscrire absolument.

Exemple de code correct (DO) : Restriction d'accès sur l'ID du patient dans Express, combinant vérification de rôle et propriété. Ici on s'assure que seul le patient propriétaire ou un médecin autorisé peut accéder aux documents du patient :

```
// Route protégée : GET /api/patients/:patientId/documents
router.get(
  '/patients/:patientId/documents',
  authenticate,  // Extrait req.userId et req.userRole depuis le token
  authorize(['patient', 'medecin']),  // Doit être connecté en tant que
patient ou médecin (admin exclu ici volontairement)
  async (req, res) => {
    const patientId = Number(req.params.patientId);
    const currentUserId = req.userId;
    const currentRole = req.userRole;

    if (currentRole === 'patient') {
      // Un patient ne peut voir que ses propres documents
      if (patientId !== currentUserId) {
        return res.status(403).json({ error: 'Accès interdit à un autre
dossier patient.' });
      }
    } else if (currentRole === 'medecin') {
      // Un médecin ne peut voir que les documents de ses patients suivis
      const result = await pool.query(
        'SELECT 1 FROM suivi_patients WHERE patient_id=$1 AND medecin_id=$2',
        [patientId, currentUserId]
      );
      if (result.rowCount === 0) {
        return res.status(403).json({ error: 'Accès interdit : ce patient ne
fait pas partie de vos patients.' });
      }
    }
    // Si on est ici, l'accès est autorisé selon le rôle.
    const docs = await pool.query('SELECT * FROM documents WHERE
patient_id=$1', [patientId]);
    res.json(docs.rows);
  }
);
```

Dans cet exemple, on voit clairement la logique : patient -> compare les IDs, médecin -> vérifie en base (table `suivi_patients` fictive) l'autorisation. Un admin n'est même pas listé dans `authorize` pour cette route (il n'y a pas de raison qu'il consulte les documents patients dans notre politique).

Exemple de code vulnérable (DON'T) : Pas de vérification d'IDOR du tout – le piège à éviter :

```
// Mauvais exemple: ne fait aucune vérification d'accès propriétaire
app.get('/api/patients/:id/documents', authenticate, async (req, res) => {
  const patientId = req.params.id;
  // Failles:
  // 1) Aucun contrôle que l'utilisateur connecté correspond à cet id ou y
est autorisé
  // 2) Un patient lambda pourrait accéder aux documents d'un autre patient
en modifiant l'URL
  // 3) Un médecin pourrait accéder à n'importe quel patientId sans
restriction
  const docs = await pool.query(`SELECT * FROM documents WHERE patient_id = $
{patientId}`);
  res.json(docs.rows);
});
```

Ici, même si le user est authentifié, on ne restreint pas **qui** peut demander **quel** identifiant. C'est exactement la faille IDOR. De plus, cet exemple concatène directement l'ID dans la requête SQL, ce qui introduit en plus une injection SQL possible (voir point 2). Ce double défaut pourrait permettre à un attaquant d'exfiltrer des documents de tous les patients, voire d'exécuter du SQL arbitraire – à proscrire.

2. SQL Injection – Paramétrer les requêtes SQL correctement

SQLi survient dès qu'une entrée utilisateur est insérée directement dans une requête SQL sans précaution, permettant à un attaquant de modifier la requête. Par exemple, un champ de recherche ou un ID de paramètre d'URL mal gérés peuvent conduire à une injection. Avec Node-postgres (`pg`), vous utilisez *normalement* des requêtes paramétrées préparées, ce qui est très bien : le fait d'utiliser `$1`, `$2` ... dans vos requêtes et de fournir un tableau de valeurs à part garantit que le moteur PostgreSQL traitera ces valeurs comme des données, pas comme du SQL [2] . Cependant, il y a encore des erreurs à éviter :

DO:

- **Toujours utiliser les placeholders `$1, $2, ...` fournis par `pg` pour insérer les variables.** Par exemple :

  ```
  const userId = req.params.id;
  const result = await pool.query('SELECT * FROM users WHERE id = $1',
  [userId]);
  ```

  Ici, même si `userId` contient quelque chose de malveillant (ex: `1; DROP TABLE users;--`), il ne sera pas exécuté : PostgreSQL va le prendre comme une valeur de paramètre littérale, et la requête sera sûre (probablement retournant 0 résultat au pire, mais pas d'effet de bord).

- **Valider/assainir les entrées** même avant la requête. Par exemple, si on attend un entier pour un ID, on peut convertir avec `Number()` et s'assurer que c'est un nombre valide, sinon renvoyer une erreur 400. Ça ajoute une couche de sûreté (même si paramétré, c'est bien d'éviter des choses qui n'ont pas de sens comme `abc` en ID). De même, pour des filtres ou champs de recherche, on peut limiter la longueur maximale des entrées, etc.
- **Utiliser des listes blanches pour les parties dynamiques de requête.** C'est un point souvent oublié : les placeholders protègent les **valeurs**, mais pas les identifiants SQL (noms de table/colonne, mots-clés) [3] . Si, par exemple, vous offrez un tri dynamique via un paramètre `sortBy` que le client peut choisir (disons `"name"` ou `"date"`), on ne peut pas faire `... ORDER BY $1` en espérant que ce $1 protège le nom de colonne – PostgreSQL n'autorise pas de paramétrer un identifiant de colonne de cette manière. Certains développeurs tentent des concaténations du style:

```
const sort = req.query.sortBy; // e.g. "name; DROP TABLE ..." injecté
const query = `SELECT * FROM patients ORDER BY ${sort}`;
pool.query(query);
```

Ceci est dangereux. **Solution :** définir un tableau de colonnes autorisées :

```
const allowedSorts = ['name', 'date', 'age'];
const sort = allowedSorts.includes(req.query.sortBy) ?
req.query.sortBy : 'name';
const query = `SELECT * FROM patients ORDER BY ${sort}`;
```

Ici on n'insère dans la requête que des valeurs sûres prédéfinies. Pour plus de sûreté, vous pouvez encore échapper le nom de colonne via une fonction ou utiliser une lib comme `pg-format` qui échappe les identifiants correctement [3] .
- **Préférer un ORM ou Query Builder si possible,** qui font ce travail pour vous. Vous n'en utilisez pas (juste `pg`), ce qui est acceptable si vous êtes rigoureux. Continuez à être vigilant sur chaque concaténation de string liée au SQL.
- **Tester** vos défenses d'injection. Par exemple, essayer d'entrer `' OR 1=1--` dans les champs de formulaire en dev et voir si ça jette une erreur (c'est bon signe) ou si ça renvoie des données inattendues. Utilisez des outils automatiques ou des bibliothèques de fuzzing sur vos endpoints pour voir s'ils résistent aux patterns connus.

DON'T:

- **NE JAMAIS** construire une requête SQL en concaténant directement des variables utilisateur sans protection. Par exemple :

```
const user = req.body.username;
const pwd = req.body.password;
const result = await pool.query(
  "SELECT * FROM users WHERE username = '" + user + "' AND password =
'" + pwd + "';"
);
```

C'est un exemple extrême de vulnérabilité : un attaquant pourrait mettre `user = 'alice'` et `pwd = "' OR '1'='1"` et la requête deviendrait

`SELECT * FROM users WHERE username = 'alice' AND password = '' OR '1'='1';` – la clause OR true permettrait de bypass le mot de passe [2] . Même si vos cas sont moins évidents, toute concaténation introduit un risque.

- Ne pas utiliser `pg.query` avec du texte formatté type template strings contenant des valeurs non échappées. Ex: `pool.query(`UPDATE table SET col=${val} WHERE id=${id}`);` est équivalent à concaténer – donc vulnérable.
- **Évitez d'utiliser des requêtes dynamiques complexes** type `pg.query("SELECT * FROM " + tableName)` où `tableName` vient d'un utilisateur. Si vraiment vous avez une fonctionnalité d'accès à différentes tables en fonction d'un paramètre (rare en appli web), validez ce paramètre strictement (liste blanche). Dans l'idéal, ne laissez pas l'utilisateur spécifier une table ou une colonne du tout.
- **Ne pas ignorer les erreurs de la DB.** Si une injection est tentée, souvent PostgreSQL renverra une erreur (synthaxe SQL invalide). Logguez ces erreurs et inspectez-les, ça peut être un signe qu'une attaque a été tentée. Ne renvoyez pas l'erreur brute au client (car cela donne des infos), mais log interne oui.
- **Ne stockez pas de données user directement dans des requêtes SQL pré-écrites.** Par exemple, éviter d'avoir des migrations ou procédures stockées qui intègrent des valeurs non contrôlées. C'est plus rare, mais c'est arrivé que des dumps SQL ou CSV d'import soient piégés avec des commandes SQL.

Exemple de code correct (DO) : Paramétrage et utilisation appropriée de requêtes :

```
// Bonne pratique: utiliser des requêtes paramétrées
app.get('/api/appointments', authenticate, async (req, res) => {
  try {
    // On filtre par médecin connecté éventuellement
    const medecinId = req.userRole === 'medecin' ? req.userId : null;
    let queryText = 'SELECT * FROM rendez_vous';
    let params = [];
    if (medecinId) {
      queryText += ' WHERE medecin_id = $1';
      params.push(medecinId);
    }
    queryText += ' ORDER BY date DESC';  // tri fixe, ou prédéfini
    const result = await pool.query(queryText, params);
    res.json(result.rows);
  } catch(err) {
    console.error('DB error', err);
    res.status(500).send('Erreur interne');
  }
});
```

Ici, on utilise `params` pour insérer `medecinId` s'il s'applique. On n'a **aucune concaténation directe de variable dans le SQL** (sauf pour coller des morceaux fixes comme `ORDER BY date DESC`, ce qui est sans danger). Ce code est insensible à l'injection même si `req.userId` était compromis (mais comme il vient du token, c'est sous contrôle serveur).

Exemple de code vulnérable (DON'T) :

```
// Mauvaise pratique: injection SQL possible
app.get('/search', authenticate, async (req, res) => {
  const term = req.query.q; // terme de recherche fourni par l'utilisateur
  try {
    const query = `SELECT * FROM patients WHERE name LIKE '%${term}%'`;
    // ☠ Si term contient un apostrophe ou des guillemets, cela brise la
requête.
    // Un attaquant pourrait mettre term = %' OR '1'='1 --
    // La requête deviendrait : SELECT * FROM patients WHERE name LIKE '%%'
OR '1'='1--%'
    // Ce qui retourne tous les patients (faille d'IDS) voire provoque une
erreur SQL selon le cas.
    const result = await pool.query(query);
    res.json(result.rows);
  } catch(err) {
    res.status(500).send(err.message); // ☠ de plus, renvoie l'erreur brute
potentiellement
  }
});
```

Plusieurs problèmes ici : la requête est construite via template string avec `term` inséré tel quel – injection possible. On renvoie même `err.message` directement, ce qui pourrait révéler la requête mal formée au client (information leak). Pour corriger : paramétrer la requête (`... WHERE name LIKE $1`, avec `%${term}%` comme valeur paramétrée), et ne pas divulguer l'erreur SQL.

3. Cross-Site Scripting (XSS) – Protection des entrées et sorties

XSS stocké ou réfléchi est un danger majeur pour une appli gérant des données utilisateur, car un script injecté peut voler des informations sensibles (tokens, données perso) ou usurper des actions au nom de la victime. Dans votre appli, on peut identifier plusieurs surfaces de risque XSS :

- **Champs de profil utilisateur (patients ou médecins)** : noms, prénoms, adresse, ou champs de texte libre comme "biographie" du médecin, "antécédents médicaux" du patient. Si un attaquant y insère du HTML/JS, et qu'il est rendu tel quel plus tard dans le profil (par exemple, un médecin qui consulterait la bio du patient), ça exécuterait le code.
- **Noms de documents uploadés** : si vous affichez le nom original du fichier (ex: "ordonnance_<script>alert(1)</script>.pdf"), cela pourrait injecter du HTML dans la page de liste de documents.
- **Comptes-rendus médicaux, messages, notes** : tout champ où du texte saisi par l'un est visible par un autre (par ex, un médecin écrit un compte-rendu que le patient lit, ou vice versa) – il faut s'assurer qu'aucun n'a pu y glisser du script.
- **Paramètres dans l'URL éventuellement** : si vous passez des valeurs de requête dans le frontend pour, disons, un champ de recherche, attention aux méthodes de rendu (souvent moins problématique en React car pas de templating string côté serveur).
- **Contenu des fichiers** : ce n'est pas exactement XSS au sens strict, mais attention à un fichier potentiellement interprété comme du HTML par le navigateur (d'où l'importance du header `nosniff` et content-type correct, vu plus haut).

React par défaut, comme mentionné, est assez **XSS-friendly** car il encode les entités HTML. Si vous faites `<p>{user.bio}</p>`, et que `user.bio = "<script>alert(1)</script>"`, React ne va pas exécuter le script : il va l'afficher littéralement comme texte (en remplaçant `<` par `<` etc.) [4]. C'est une bonne protection de base, donc **respectez ce flux de données unidirectionnel sans insertion dangereuse**.

DO:

- **Nettoyer côté serveur** les entrées susceptibles de contenir du code. Par exemple, vous pouvez utiliser des fonctions de strip HTML. Une approche simple : si un champ n'est censé contenir que du texte brut (nom, adresse), on peut carrément enlever les caractères `<` ou `>` ou encoder toute la chaîne à l'enregistrement. Ex:

```
// Exemple: Nettoyer une bio de patient avant stockage
const rawBio = req.body.bio;
// Supprime toute balise HTML en conservant le texte entre elles:
const bio = rawBio.replace(/<[^>]+>/g, '');
```

  Ici on vire toutes balises. Ou on peut utiliser un utilitaire plus complet (DOMPurify peut s'utiliser sur Node via JSDOM, ou des lib comme `sanitize-html` côté Node). L'idée est de **ne jamais stocker** de script injecté si on peut l'éviter. Ainsi, même en cas de bug de rendu, le script n'est pas là.
- **Échapper les sorties côté serveur si vous faites du rendu serveur** (dans une API REST pure, on ne renvoie que du JSON – a priori pas de rendu HTML côté serveur, donc ce point est plus pour les applis SSR ou les templates).
- **Côté frontend React** : continuer d'utiliser des balises normales avec du contenu entre `{}`. Ne pas utiliser `dangerouslySetInnerHTML` sauf nécessité absolue. Si vous devez afficher du HTML riche venant de la base (par ex, vous stockez du contenu HTML pour un compte-rendu formaté), *nettoyez-le avant affichage*. Par exemple, avec DOMPurify:

```
import DOMPurify from 'dompurify';
...
const safeHTML = DOMPurify.sanitize(report.content);
return <div dangerouslySetInnerHTML={{ __html: safeHTML }}></div>;
```

  Ainsi, même si `report.content` contenait `<script>...</script>`, DOMPurify l'enlèvera [6] [7]. *Pédagogie :* on l'appelle "dangerously" pour rappeler que c'est risqué – toujours purifier avant usage [5].
- **Sanitizer sur les noms de fichier** quand vous les affichez : ne jamais insérer le nom du fichier tel quel dans du HTML sans échappement. En React, si vous faites `{file.name}`, c'est bon. Si vous devez par exemple mettre le nom dans un attribut HTML, faites attention à bien le échaper. Ex: `<img alt={file.name} ...>` est correct, React échappe. Mais si c'était du pur string concatat (rare en React), ce serait dangereux.
- **Content Security Policy (CSP)** : Envisagez de définir une CSP restrictive via Helmet (Content-Security-Policy header). Par exemple, interdire l'exécution de tout script inline ou tiers non approuvé. Une CSP typique : `default-src 'self'; script-src 'self' 'nonce-xyz'; object-src 'none'; style-src 'self' 'unsafe-inline'; frame-ancestors 'none';` etc. Une CSP ne remplace pas l'assainissement XSS, mais ajoute une couche : si malgré tout un `<script>` malicieux arrivait dans la page, le navigateur le bloquerait si CSP interdit

inline scripts ou sources non autorisées. **Attention**: CSP nécessite de gérer les nonce si vous avez du script injection légitime (Google Analytics, par ex.), et en React SPA c'est à configurer sur le serveur statique. C'est un bonus, pas obligatoire, mais souhaitable pour une appli médicale où on veut limiter tout vecteur XSS.

- **Tester régulièrement** : utilisez par ex. un outil comme OWASP ZAP ou un scanner XSS sur vos formulaires (ou essayez manuellement en entrant des `<script>alert(1)</script>` partout) pour voir si ça ressort exécuté. Si vous trouvez un endroit où c'est le cas, corrigez immédiatement en ajoutant sanitization.

DON'T:

- **Ne pas insérer de HTML non maîtrisé dans le DOM.** Évitez les situations comme : vous recevez du backend une chaîne contenant du HTML (par ex en Markdown converti) et vous faites `dangerouslySetInnerHTML={{__html: data}}` sans nettoyage. C'est le schéma classique d'une faille XSS dans une appli React. Si le `data` n'est pas 100% de confiance, c'est une porte ouverte.
- **Ne pas désactiver l'échappement de template.** Dans certains frameworks, on peut faire des choses du genre `<%= unsafeVar %>` vs `<%- safeVar %>` (en EJS par ex). En React, la plupart du temps c'est échappé, donc c'est bien.
- **Évitez d'autoriser du contenu HTML riche soumis par l'utilisateur** à moins que ce ne soit une fonctionnalité voulue (ex: permettre au patient de formater son texte). Si oui, alors c'est plus délicat : il faut limiter les balises autorisées (par ex, autoriser `<b>`, `<i>` mais pas `<script>` ni `<iframe>` etc.). Des libs de sanitization peuvent accepter une config (listes blanches de tags).
- **Ne pas oublier le** DOM XSS** : même si vous sécurisez le backend, un développeur frontend pourrait introduire une vulnérabilité purement côté client. Par exemple, utiliser `innerHTML` pour injecter un bout de HTML construit à partir de variables user (ex: un composant qui fait `element.innerHTML = "<p>"+userInput+"</p>"`). Cela contourne la protection JSX. Donc, rappelez à toute l'équipe frontend de ne pas utiliser `innerHTML` direct, ou d'utiliser les méthodes React/Angular/Vue de binding sécurisées.

Exemple de code correct (DO) :

Supposons qu'on a un champ "allergies" que le patient peut remplir librement (texte). On va le nettoyer côté serveur et l'afficher côté client de façon sûre.

Backend (Express) :

```
// Middleware de nettoyage d'entrée (exemple simple)
function sanitizeInput(str) {
  return str.replace(/</g, "&lt;").replace(/>/g, "&gt;");
}

app.post('/api/patients/profile', authenticate, async (req, res) => {
  const patientId = req.userId;
  const rawAllergies = req.body.allergies;
  const allergies = sanitizeInput(rawAllergies);
  await pool.query('UPDATE profile SET allergies=$1 WHERE patient_id=$2',
[allergies, patientId]);
```

```
  res.sendStatus(200);
});
```

Ici, on transforme `<` en `<` etc., de sorte que si quelqu'un avait mis `<script>alert(1)</script>`, la base contiendra `<script>alert(1)</script>`. Ça se stocke "échappé". Certains préfèrent stocker tel quel et n'échapper qu'au rendu, c'est une stratégie possible aussi (mais assurez-vous de le faire *chaque* fois au rendu). Ici on fait le choix de stocker déjà neutre.

Frontend (React) :

```
// Affichage du champ allergies dans le profil, en React:
<div className="allergies">
  <strong>Allergies : </strong>
  {userProfile.allergies}
</div>
```

On se contente de mettre la valeur dans une expression JSX, React s'occupe d'afficher les entités HTML sans danger. Même si `userProfile.allergies` contenait des séquences `<script>`, elles apparaîtront comme `<script>` à l'écran (ce qui est un peu moche mais inoffensif). On pourrait améliorer en remplaçant ces séquences par rien à l'enregistrement pour ne pas polluer l'affichage. Le principal est : pas de `dangerouslySetInnerHTML` ici.

Exemple de code vulnérable (DON'T) :

```
// Mauvais : injection directe de HTML potentiellement dangereux
<div className="bio" dangerouslySetInnerHTML={{ __html: userProfile.bio }}></
div>
```

Si `userProfile.bio` contient `Bonjour <script>stealCookies()</script>`, cette approche va exécuter le script lors du rendu du composant. C'est **dangereux**. Sans nettoyage préalable, à proscrire. Si vraiment vous aviez un HTML légitime (par ex, l'utilisateur a mis des <b> et <i> pour formatter son texte), utilisez un sanitizer *avant*. Mais le mieux reste de stocker et rendre du texte brut, ou d'utiliser un formatage balisé safe (ex: Markdown->HTML avec une lib de confiance configurée en mode sécurisé).

Autre mauvais exemple, purement côté client JS :

```
// Mauvais : manipulation du DOM directe sans échappement
const msg = getUserInput(); // l'attaquant saisit: </
div><script>alert('XSS')</script>
document.getElementById('output').innerHTML = "<p>" + msg + "</p>";
```

Ici, le `msg` va briser la structure et injecter un script. Si on tient à faire du DOM manual, il faudrait utiliser `textContent` à la place :

```
document.getElementById('output').textContent = msg;
```

qui n'interprétera pas le HTML. Mais dans React, on évite ce genre de code de toute façon.

En bref : traquez toute possibilité où du contenu fourni par l'utilisateur peut se transformer en du code interprété dans le navigateur d'un autre utilisateur, et placez-y des barrières (sanitization, échappement, CSP en dernier recours, etc.).

4. Insecure File Uploads – Sécuriser le téléversement et la distribution des fichiers

Le téléversement de fichiers utilisateur combine plusieurs risques : injection de malwares, attaques par type de fichier trompeur, surcharge du stockage, exécution non voulue, etc. Détaillons les points soulevés :

Stockage BLOB dans la base vs Stockage fichier externe :

- **En base de données (BLOB)** :
- *Avantages:* accès contrôlé via SQL (on peut appliquer les mêmes requêtes d'autorisation, joindre avec l'ID du propriétaire, etc.), backup unifié avec la DB, pas de soucis de synchronisation de fichiers.
- *Inconvénients:* la base gonfle vite (les BLOBs impactent les performances des backups, index, etc.), chaque accès fichier passe par l'app web (pas de CDN cache facile), potentiellement moins efficace pour servir de gros fichiers binaires.
- *Sécurité:* Si la DB est compromise, l'attaquant a les fichiers (mais s'il a la DB, il a de toute façon déjà les données texte...). Par contre, pas de risque de laisser traîner un fichier accessible publiquement par URL hasardeuse – tout transit doit repasser par l'appli (ce qui peut être plus sûr).
- **En stockage de fichiers (ex: AWS S3, système de fichiers serveur)** :
- *Avantages:* peut soulager la DB, permettre l'utilisation de CDN ou du stockage objet optimisé pour fichiers, scalable séparément.
- *Inconvénients:* il faut bien gérer l'ACL (permissions). Sur S3, par exemple, il ne faut pas que le bucket soit public en entier. On peut générer des URL signées à durée limitée pour que seuls les utilisateurs autorisés téléchargent le fichier. Sur un serveur de fichiers local, il faut s'assurer que l'appli sert le fichier via Express et que le dossier n'est pas exposé directement.
- *Sécurité:* Attention aux erreurs de config (ex: bucket S3 public exposant des données sensibles). Il faut idéalement chiffrer les fichiers au repos si c'est du PHI sensible. Aussi, si on stocke sur le même serveur que l'application, **ne pas** mettre ces fichiers dans le dossier public de l'appli (sinon une requête directe URL pourrait bypasser l'auth). Mieux vaut les mettre hors du dossier web et avoir un endpoint pour les servir après contrôle.

Validation des fichiers uploadés :

- **Limiter par type MIME et extension :** Décidez quels types de fichiers sont autorisés (PDF, JPEG, PNG... peut-être DOCX si besoin). Dès la réception via Multer, utilisez `fileFilter` pour refuser les fichiers dont l'extension ou le type ne correspond pas. *Exemple:*

```
const storage = multer.memoryStorage();
const upload = multer({
  storage,
  fileFilter: (req, file, cb) => {
    const allowedExt = ['.pdf', '.jpg', '.jpeg', '.png'];
    const ext = path.extname(file.originalname).toLowerCase();
    const allowedMime = ['application/pdf', 'image/jpeg', 'image/png'];
```

```
    if (!allowedExt.includes(ext) || !
allowedMime.includes(file.mimetype)) {
      return cb(new Error('Type de fichier non autorisé.'), false);
    }
    cb(null, true);
  },
  limits: { fileSize: 5 * 1024 * 1024 } // par ex limite 5 MB
});
```

Ceci vérifie extension ET mimetype. **Mais** comme mentionné, `file.mimetype` est fourni par le client/navigateur et se base sur l'extension souvent [8] . Donc ça ne suffit pas contre un `.exe` renommé en `.pdf`.

- **Inspecter la signature du fichier** : Utilisez une librairie comme `file-type` (qui lit les premiers octets du fichier pour déterminer le vrai type). Par exemple :

```
const FileType = await import('file-type');
...
const buffer = file.buffer; // Buffer du fichier (si memoryStorage)
const type = await FileType.fileTypeFromBuffer(buffer);
if (!type || !['pdf', 'jpg', 'png'].includes(type.ext)) {
  return cb(new Error('Le contenu du fichier ne correspond pas au type
attendu'), false);
}
```

Ainsi, même si quelqu'un renomme un `.exe` en `.png`, FileType détectera par son header binaire que c'est un PE exe, pas une image.

- **Renommer le fichier stocké** : Ne pas conserver `file.originalname` tel quel comme nom final sans nettoyage. Il peut contenir des caractères spéciaux, espaces, etc., qui posent problème en URL. Ou pire, des séquences path traversal (`../`). Utilisez par ex. un UUID ou composez un nom sûr :

```
const safeName = 'doc_' + Date.now() + path.extname(file.originalname);
```

Au minimum, utilisez `path.basename` pour enlever tout chemin.

- **Scanner antivirus** (optionnel, mais dans santé c'est pas une mauvaise idée) : Intégrer un scanner type ClamAV pour les fichiers uploadés pourrait détecter des malware connus. C'est un plus, surtout si un jour vous autorisez des formats comme .docx qui peuvent contenir des macros.
- **Limiter la taille** : vos `limits.fileSize` dans Multer empêchent des abus (ne pas saturer disque ou mémoire).
- **Stocker de manière sécurisée** : Si BD, mettre dans une table séparée ou avec un champ BYTEA. Si filesystem, stocker dans un dossier non accessible directement par le serveur web. Si S3, configurer le bucket en privé, et ne donner accès qu'au travers de clés temporaires.

Service des fichiers aux clients :

- **Toujours authentifier/autoriser** avant de fournir le fichier. Si vous avez un endpoint `/api/patients/:patientId/documents/:docId/download`, il doit vérifier que l'utilisateur a droit

à ce document (comme vous faites avec `checkDocumentOwnership` – n'oubliez pas d'y inclure un check pour le médecin vs patient comme discuté en point 1) avant d'aller chercher le fichier.
- **Headers de sécurité lors de la réponse :** Comme dit, ajouter `Content-Disposition: attachment; filename="<nom original.safe>"`. Ainsi, le navigateur téléchargera le fichier au lieu de tenter de l'ouvrir directement dans la page. Pour une image, vous pourriez permettre l'affichage inline, mais pour tout le reste (PDF, DOCX…), c'est souvent plus sûr de forcer le download, surtout si l'appli n'a pas besoin de l'intégrer en iframe.
- **MIME correct :** Envoyez `Content-Type: application/pdf` pour un PDF, etc., afin que le navigateur sache quoi en faire. N'essayez pas de servir un PDF avec `text/html` (sinon le navigateur pourrait potentiellement l'interpréter différemment).
- **X-Content-Type-Options: nosniff** : Helmet le met par défaut globalement [10], sinon ajoutez-le sur les réponses de fichiers. Cela évite que certains navigateurs tentent de deviner le type du fichier et d'exécuter du contenu actif dedans.
- **Pas d'exécution côté serveur :** Par précaution, ne faites jamais "ouvrir" un fichier uploadé sur le serveur lui-même (ex: ne passez pas un PDF uploadé à une commande système sans contrôle, etc.) – cela pourrait déclencher des failles sur le serveur (ex: image malformée exploitant une librairie d'image).
- **Optionnel – Domaines séparés :** Certaines applis servent les fichiers statiques depuis un autre sous-domaine (ex: `media.carnetsante.com`) sans cookie et avec CSP restreinte, pour limiter l'impact si un contenu malicieux était servi. Ça vaut surtout pour du contenu public. Dans votre cas, comme tout est privé, ce n'est pas nécessairement utile. Mais c'est bon de connaître la pratique.

DON'T:

- **Ne stockez pas en clair des fichiers potentiellement exécutables accessibles publiquement.** Par exemple, ne permettez pas l'upload de `.html` ou `.js`. Si vraiment un jour vous devez accepter du .html (rare ici), stockez-le de sorte qu'il ne puisse pas être servi depuis le même domaine web ou alors en l'envoyant avec `Content-Disposition: attachment` pour éviter qu'il s'exécute dans le contexte de votre application.
- **Ne faites pas confiance à** `req.file.mimetype` **uniquement.** Comme vu, c'est contournable [9]. Toujours double-check.
- **Ne laissez pas l'utilisateur retrouver un fichier via un ID facile** sans vérifier l'autorisation. Exemple de schéma à éviter : vous stockez les fichiers sur disque sous leur `document_id` (mettons `123.pdf`), et un utilisateur astucieux pourrait essayer d'accéder à `https://server/uploads/124.pdf`. Si votre serveur sert statiquement le dossier, il accédera au fichier du voisin ! D'où l'importance de ne pas servir statique ou alors d'utiliser des noms imprévisibles (hash longs) + ACL forte.
- **Ne pas oublier d'effacer les fichiers supprimés.** Si un patient supprime un document, supprimez le fichier de stockage aussi (sinon accumulation de données non référencées, et risque qu'elles soient encore accessibles si quelqu'un connaissait l'URL).
- **Ne loggez pas le contenu** ou le chemin complet des fichiers uploadés dans vos logs applicatifs (ou au moins, nettoyez-les), au cas où un nom de fichier contiendrait des caractères bizarres, pour éviter du XSS via les consoles d'admin (cas rare, mais par exemple si vous aviez une interface web d'admin affichant les noms de fichiers des logs).
- **Ne négligez pas les .zip ou autres archives** : Si vous autorisez les archives, soyez conscient qu'elles peuvent contenir des fichiers piégés, ou consommer beaucoup de CPU lors de la décompression (zip bomb). Peut-être refusez .zip .rar à moins d'un besoin précis. Ou imposez des contrôles (taille après décompression, etc.).

Exemple de configuration correcte (DO) : Multer avec filtrage et utilisation de file-type (TypeScript style pseudo-code pour concision) :

```
import multer from 'multer';
import { fileTypeFromBuffer } from 'file-type';
import path from 'path';

const storage = multer.memoryStorage(); // on stocke en mémoire pour pouvoir
analyser avant d'enregistrer
const upload = multer({
  storage,
  limits: { fileSize: 10 * 1024 * 1024 }, // 10 MB max
  fileFilter: async (req, file, cb) => {
    try {
      const ext = path.extname(file.originalname).toLowerCase();
      const allowedExt = ['.pdf', '.png', '.jpg', '.jpeg'];
      const allowedMime = ['application/pdf', 'image/png', 'image/jpeg'];
      if (!allowedExt.includes(ext) || !allowedMime.includes(file.mimetype))
{
        return cb(new Error('Type de fichier interdit'), false);
      }
      // Vérification du contenu réel du fichier
      const buffer = file.buffer;
      const type = await fileTypeFromBuffer(buffer);
      if (!type) {
        return cb(new Error('Impossible de déterminer le type du fichier'),
false);
      }
      if (type.mime === 'application/pdf') {
        if (ext !== '.pdf') {
          return cb(new Error('Extension ne correspond pas au contenu PDF'),
false);
        }
      } else if (type.mime === 'image/png' || type.mime === 'image/jpeg') {
        if (!['.png', '.jpg', '.jpeg'].includes(ext)) {
          return cb(new Error('Extension ne correspond pas à l\'image'),
false);
        }
      } else {
        return cb(new Error('Type de fichier non autorisé (contenu)'),
false);
      }
      cb(null, true);
    } catch (err) {
      cb(err, false);
    }
  }
});
```

Ensuite usage :

```
router.post('/api/patients/documents', authenticate,
upload.single('document'), async (req, res) => {
  // req.file a été validé à ce stade
  const safeFilename = `doc_${Date.now()}_${req.userId}$
{path.extname(req.file.originalname).toLowerCase()}`;
  // Si stockage local :
  fs.writeFileSync(path.join(UPLOAD_DIR, safeFilename), req.file.buffer);
  // En base, enregistrer metadata:
  await pool.query('INSERT INTO document(name, patient_id, mime) VALUES($1,
$2, $3)',
                   [safeFilename, req.userId, req.file.mimetype]);
  res.status(201).send('Fichier uploadé');
});
```

Cet exemple refuse tout ce qui n'est pas PDF ou image, vérifie extension/mime cohérence, limite la taille, et génère un nom de fichier sûr unique.

Exemple de pratique vulnérable (DON'T) :

- Accepter tout fichier sans filtrage :

```
app.post('/upload', upload.any(), (req, res) => {
  // .any() accepte tout, pas de fileFilter
  // Puis stocke directement:
  fs.writeFileSync('/var/www/uploads/' + req.files[0].originalname,
req.files[0].buffer);
  res.send("OK");
});
```

  Ici, un attaquant peut uploader `evil.html` contenant un script, et ensuite y accéder via `http://site/uploads/evil.html` (si le serveur sert statiquement /uploads). Il pourra exécuter des actions sous le domaine du site -> XSS stocké via fichier. De plus, pas de limite de taille, pas de vérif type (il pourrait uploader un `.exe`).
- Ne pas mettre `Content-Disposition` : Si vous servez un PDF directement et qu'il a une charge active ou qu'il est interprété dans le navigateur, ça peut poser problème (certains PDF embarquent du JavaScript interne ou des exploits de lecteur PDF). En forçant le download, vous sortez du contexte navigateur.
- Oublier nosniff : Internet Explorer par ex, sans `nosniff`, pourrait décider d'exécuter un `<script>` présent dans un fichier mal nommé.

En somme, traitez les fichiers uploadés comme **potentiellement dangereux** par défaut. Ne faites confiance ni au nom, ni au type déclaré, ni au contenu, sans vérifier. Et contrôlez strictement qui y accède et comment ils sont exécutés/téléchargés.

5. Authentication & Session Management – Renforcement du flux JWT

Le système JWT mis en place est déjà structuré, revenons sur chaque point pour s'assurer qu'il n'y a pas d'angles morts :

- **Access Token en mémoire React** : C'est une méthode qui évite de le stocker dans un endroit persistant (localStorage ou cookie). Ainsi, si un attaquant n'a pas déjà du code dans la page (XSS), il ne peut pas récupérer le token. En revanche, si un XSS est présent, l'attaquant peut directement effectuer des actions via l'application (en appelant les fonctions qui utilisent le token en mémoire) voire voler le token s'il trouve un moyen d'accès à cette variable. On considère souvent la mémoire React comme relativement sûre car non accessible globalement, mais n'oublions pas qu'un XSS peut parcourir le DOM ou les variables globales. Cela dit, **garder le token hors du localStorage est fortement recommandé** (localStorage est lisible par n'importe quel script tiers compromis). Donc continuez avec le stockage mémoire, et rafraîchissez-le via le refresh token au besoin.
- **Refresh Token en HttpOnly Cookie** : Très bien, HttpOnly + Secure + SameSite=Strict protège contre la plupart des vols de cookie (XSS ne peut le lire, et CSRF ne peut pas le soumettre car Strict empêche l'envoi cross-site). Veillez à bien définir `Secure` (seulement HTTPS) en prod.
- **Courte durée de l'access token (15 min)** : Aligné avec les bonnes pratiques (quelques minutes à heures max) [12] . Ça limite l'impact d'un vol. Vous pourriez même réduire à 5-10 min si l'UX le permet (mais 15 min c'est souvent un bon compromis).
- **Longue durée du refresh (7 jours)** : Acceptable, bien qu'avec rotation on peut se permettre plus car vol moins utile. Sans rotation, 7 jours c'est potentiellement critique si volé (l'attaquant a une semaine pour en abuser). Mais on va justement recommander la rotation pour atténuer ce risque [13] .
- **Invalider le refresh token à la déconnexion** : C'est essentiel. Stockant les refresh tokens en BD, un simple `DELETE FROM refresh_tokens WHERE token = ...` suffit. Si vous stockez un hash du token, même principe avec la valeur hash. Ainsi, même si l'attaquant volait le cookie de refresh *juste avant* que l'utilisateur se déconnecte, ce token ne fonctionnerait plus après le logout.
- **Menaces spécifiques :**
- *Token reuse (réutilisation illicite)* : Concernant un access token, s'il est volé, l'attaquant peut l'utiliser jusqu'à son expiration. Pas moyen de l'empêcher à part le blacklister (ce qui nécessite un storage côté serveur des tokens invalidés, ce qui casse un peu le principe stateless du JWT). Pour vos tokens d'accès, probablement pas de liste noire (pas mentionnée). Une approche pour invalidation côté serveur serait d'inclure, comme dit, un champ `tokenVersion` ou `lastLogoutTime` dans le JWT. Sans aller trop loin : considérez qu'un vol d'access token ne donne que 15 min d'accès, ce qui est déjà pas mal mais c'est la fenêtre de risque. **Mitigation** : surveiller (point 8 monitoring) si un token volé est utilisé (par ex, usage depuis une IP ou User-Agent très différent).
- *Refresh token theft (vol du cookie de refresh)* : Scénarios possibles : XSS (mais HttpOnly le protège contre lecture directe, l'attaquant devra utiliser le cookie de manière indirecte via une requête qu'il déclenche, comme expliqué), ou vol du cookie via un autre vecteur (ex: un malware sur la machine de l'utilisateur, ou un ex-admin qui accède à la DB des tokens s'ils n'étaient pas hashés). **Mesures** : la rotation est le meilleur rempart. Avec refresh token rotation activée, un token volé ne pourra être utilisé qu'une fois et sera détecté comme reuse ensuite [25] [26] . Concrètement, si l'attaquant utilise l'ancien token après que l'utilisateur légitime l'ait utilisé, le système doit voir que "token déjà utilisé -> invalider toute la chaîne" [16] . C'est ce qu'Auth0 fait avec un concept de "token family" et reuse detection. Vous pouvez implémenter plus simplement : à chaque refresh, générer un nouveau token et invalider l'ancien (le supprimer de la DB). Si un ancien token arrive,

c'est qu'il y a un problème -> on peut alors *invalider tous les tokens de cet utilisateur* et exiger reconnexion (par sécurité). C'est radical, mais c'est ce que recommande Auth0 [15] [27].

- *Replay attacks (rejeu de token)* : L'exemple typique est pour le refresh (comme ci-dessus). Pour l'access token, un replay signifie un attaquant interceptant la requête API (mais en TLS c'est peu probable) ou vol de token par XSS. On peut mitiguer en liant le token à un contexte (par ex, inclure l'IP ou un device ID dans le token – mais ça complexifie car l'IP peut changer, etc.). Une solution plus élégante pour l'access token est d'utiliser des **Proof-of-Possession** tokens au lieu de bearer (hors scope ici, c'est plus complexe, on reste sur du bearer JWT). Donc accepter qu'un bearer volé = utilisable, d'où l'importance de le protéger (XSS).
- *Token leakage on logout or multi-tab* : Imaginez un utilisateur ouvre deux onglets. Il se déconnecte sur l'un (le refresh token cookie est supprimé via `res.clearCookie`). Sur l'autre onglet, s'il tente une action, le access token en mémoire peut toujours être présent jusqu'à expiration. Sans le refresh token, il ne pourra pas en obtenir un nouveau quand il expirera, mais pendant quelques minutes il pourrait encore fonctionner. C'est un cas connu (logout non global tant que le token n'est pas expiré). **Solution :** on peut stocker côté serveur un timestamp `lastLogout` pour l'utilisateur, et dans chaque requête protéger une route, comparer `token.iat` (issue time) avec `lastLogout`. Si `iat < lastLogout`, alors le token a été émis avant le dernier logout -> le considérer invalide. Il faut alors stocker ce lastLogout en mémoire serveur ou BD. C'est faisable et pas trop coûteux. Sinon, on tolère que les autres onglets deviendront invalides d'ici 15 min max. Selon la sensibilité, on peut implémenter le check pour être strict.
- *Rotation vs versioning :* Ce n'est pas exclusif. Vous pourriez stocker une colonne `refreshVersion` pour chaque user. A chaque login initial, set refreshVersion = 1. Incluez `refreshVersion` dans le payload du refresh JWT (signé). Lors du refresh, vérifiez que le `refreshVersion` du token correspond à celui en DB. Si un user se logout de partout volontairement (ou en cas de vol détecté), incrementer la version -> tous anciens refresh JWT ne matchent plus. C'est plus simple que de stocker tous les token ids. C'est valable s'il n'y a qu'un refresh token actif par utilisateur. Si vous autorisez plusieurs (ex: connexion multi-device), vous aurez plusieurs entrées en DB et la version globale les révoquerait tous en cas d'incident (ce qui peut être acceptable).
- *Multi-device considerations:* Si patients et médecins peuvent se connecter depuis plusieurs appareils en parallèle, vous devrez décider : un logout invalide-t-il tous les refresh tokens de l'utilisateur ou juste celui de l'appareil courant ? Souvent, on invalide juste le courant (ex: "Logout" ne déconnecte pas vos autres sessions ailleurs). Donc votre DB peut avoir plusieurs refresh tokens par user (avec un device or session ID). Rotation s'applique par token. Si vol d'un token d'un appareil, les autres restent valides. Versioning global invaliderait tous d'un coup (plus brutal).
- **Protection contre** brute force de tokens **: Étant des JWT signés, c'est quasi impossible de forcer trouver un valid token sans la clé secrète. Donc pas un souci si secrets solides. Par contre, votre endpoint `/api/auth/refresh` doit vérifier qu'il reçoit un refresh token valide. Un attaquant pourrait spammer ce endpoint avec des tokens aléatoires (ou expirés) pour surcharger le système.** Mesure** : Rate-limit ce endpoint aussi (voir point 6). Et éventuellement, si vous trouvez dans vos logs des tentatives avec des tokens invalides trop fréquentes, alerter.
- **Algo JWT & Secrets :** Utilisez un algo solide (HS256 ou mieux HS512, ou RS256 si clés asymétriques). Ne jamais utiliser `alg: none`. Assurez-vous que la secret key JWT est longue et stockée dans `.env` non exposée. Helmet désactive par défaut les choses type X-Powered-By, mais assurez-vous aussi de ne pas divulguer ce secret par erreur (ex: pas de commit du .env sur GitHub etc.).

- **Expiration & clock drift:** 15m expiration, prévoyez peut-être une petite marge de quelques secondes sur le serveur pour accepter un token fraîchement expiré (quelques secondes) si besoin, ou gérez le code d'erreur côté client pour rafraîchir si token expiré.
- **Refresh via httpOnly cookie – CSRF:** SameSite=Strict effectivement empêche une page externe de déclencher le refresh. Donc un attaquant ne peut pas, via une image ou iframe, forcer l'utilisateur à rafraîchir le token et voler la réponse, car la réponse ira en XHR (et SameSite empêcherait la requête externe, et de plus il ne pourrait pas lire la réponse cross-domain). Donc tant qu'il n'y a pas XSS, le refresh endpoint ne peut être touché que par votre SPA légitime.

DO:

- **Implémenter la rotation de refresh token** : comme recommandé, c'est une amélioration notable de sécurité [13] [14]. Cela signifie : quand le client appelle `/api/auth/refresh` avec un token valide, le serveur :
- Génère un nouvel access token (AT) **et** un nouvel refresh token (RT2).
- Stocke le nouvel RT2 en base, et supprime/blacklist l'ancien RT1.
- Renvoie le AT + (éventuellement le RT2 en cookie ou dans réponse).
- Le client doit remplacer le cookie ancien par le nouveau (en httpOnly cookie, ça se fait via `Set-Cookie`).
- Si ensuite un attaquant tente de réutiliser RT1, la base ne le trouvera pas -> refus (on peut retourner 401 et invalider la session).
- Optionnel: si une reuse est détectée, c'est signe de vol. À ce moment, invalider toutes sessions de l'utilisateur, forcer un re-login, et prévenir l'utilisateur éventuellement.
- **Stocker un identifiant de token en base** : Plutôt que stocker le token entier, stocker son hash (pour ne pas divulguer le secret JWT si c'est JWT, ou si c'est un GUID random c'est bon). Cela permet aussi de distinguer deux refresh simultanés. Si vous reçevez deux fois le même token avant qu'il soit invalidé, c'est potentiellement une attaque (ou un bug de double requête).
- **Utiliser une secret différente pour les refresh tokens** (vous avez JWT_REFRESH_SECRET distinct du JWT_SECRET – c'est bien). Comme ça, même si un access token est compromis, on ne peut pas fabriquer un refresh.
- **Force logout / revoke** : Prévoir une route de logout global (admin peut-être) pour invalider toutes sessions d'un user en cas de suspicion (ex: patient appelle support disant "je vois des activités bizarres", on invalide tout, en supprimant tous ses refresh tokens en DB, et en augmentant son tokenVersion).
- **Notifier les connexions sensibles** : Par exemple, envoyer un email quand un nouveau device s'est connecté (optionnel, UX vs secu). Ou lister dans le profil de l'utilisateur ses sessions actives (comme Google le fait) avec option de "Se déconnecter de tous les appareils".
- **Stocker les tokens en mémoire serveur pour contrôle ?** – Vous pouvez garder une liste en mémoire cache ou DB des JWT valides ou invalidés. Cependant, cela va à l'encontre du stateless du JWT. Mieux vaut s'appuyer sur la DB des refresh pour le contrôle d'état, et sur la signature + version pour l'access token.
- **Limiter la portée du token** : Vous avez déjà par rôle (le JWT contient probablement le rôle). On pourrait aussi inclure plus de claims: par ex, un claim "scp" (scope) qui liste ce que le token peut faire. Pour l'instant, peut-être surdimensionné. Mais par exemple, un médecin pourrait avoir un scope différent d'un patient, bien que le rôle suffise.
- **Bien paramétrer les cookies** : HttpOnly, Secure, SameSite on l'a dit. Sur le access token, ne le mettez surtout pas en cookie (vous ne l'avez pas fait, c'est bien).
- **Penser aux refresh token** idle timeout** : Certains systèmes expirent un refresh token si non utilisé depuis X temps. Par ex, 7j valides mais si l'utilisateur ne se reconnecte pas du tout pendant 7j, token expiré => re-login. C'est correct. Vous pourriez diminuer la durée si souhaité (ex: 7j peut-être 1j en très sensible, mais 7j c'est souvent ok pour UX).

- **Secourir l'utilisateur en cas de compromission** : Si vous suspectez qu'un token a fui (ex: reuse détectée), forcez le changement de mot de passe de l'utilisateur car ça pourrait venir d'un malware local. Ce n'est pas technique JWT, mais procédure.

DON'T:

- **Ne stockez jamais un JWT dans localStorage** (vous ne le faites pas, continuez ainsi, beaucoup d'applis font l'erreur).
- **Ne mettez pas l'access token en cookie** (sinon il serait envoyé partout, risque de CSRF, etc.). Votre approche actuelle évite cela.
- **Ne réutilisez pas un refresh token plusieurs fois** sans rotation. Sinon, s'il est volé, l'attaquant peut continuellement obtenir des nouveaux AT tous les 15 min pendant toute la durée (7j voire plus si pas expiré) sans se faire détecter. Avec rotation, la première utilisation du voleur le trahit.
- **Ne pas ignorer les erreurs de validation de token.** Si `authenticate` middleware trouve un token expiré ou invalide, renvoyez un 401 avec un message approprié. Le front doit gérer en redirigeant vers login ou en tentant un refresh s'il n'a pas déjà fait.
- **Ne mettez pas d'info sensible dans le payload JWT.** JWT n'est pas chiffré, juste signé. Donc n'y mettez pas de PII genre numéro de sécu, etc. Limitez aux identifiants techniques et rôle.
- **Ne pas oublier de vérifier la signature** : Utilisez bien la lib JWT (`jsonwebtoken` par ex) pour vérifier le token avec la clé. Un attaquant ne doit pas pouvoir bricoler le payload (ex: se donner rôle admin) sans que la signature faille. Aussi, spécifiez l'algorithme attendu au vérificateur (pour éviter une attaque où on passerait un JWT avec alg=none ou un alg différent, qui ne devrait plus être possible avec libs modernes mais par précaution).
- **Ne loggez pas les tokens en entier** : parce que si vos logs sont compromis, ce serait dommage d'offrir les tokens (même si expirables). Logguez leur identifiants, ou les 8 premiers caractères, etc., à la rigueur, mais pas tout.

Exemple de rotation (DO) : pseudo-code du refresh endpoint sécurisé :

```
app.post('/api/auth/refresh', async (req, res) => {
  const token = req.cookies['refreshToken'];
  if (!token) return res.sendStatus(401);
  try {
    const payload = jwt.verify(token, JWT_REFRESH_SECRET);
    const tokenId = payload.jti; // unique ID du token
    const userId = payload.sub;
    // Vérifier en DB
    const result = await
pool.query('SELECT * FROM refresh_tokens WHERE id=$1 AND user_id=$2',
[tokenId, userId]);
    if (result.rowCount === 0) {
      return res.sendStatus(403); // refresh token invalide (peut-être déjà
utilisé ou révoqué)
    }
    // Token valide, on peut l'invalider et en émettre un nouveau
    await pool.query('DELETE FROM refresh_tokens WHERE id=$1', [tokenId]);
    const newTokenId = generateRandomId();
    const newRefreshToken = jwt.sign({ sub: userId, jti: newTokenId },
JWT_REFRESH_SECRET, { expiresIn: '7d' });
    // stocker le nouveau:
    await pool.query('INSERT INTO refresh_tokens(id, user_id, expires_at)
```

```
VALUES($1,$2,$3)', [newTokenId, userId, Date.now()+7*24*3600*1000]);
    // émettre le nouvel access token
    const accessToken = jwt.sign({ sub: userId, role: payload.role /* etc
*/ }, JWT_SECRET, { expiresIn: '15m' });
    // Set-Cookie du nouveau refresh:
    res.cookie('refreshToken', newRefreshToken, { httpOnly: true, secure:
true, sameSite: 'Strict', maxAge: 7*24*3600*1000 });
    res.json({ accessToken });
  } catch (err) {
    console.error("Refresh error:", err);
    return res.sendStatus(403);
  }
});
```

Ici on voit : on supprime l'ancien token dès qu'on s'en sert. Si quelqu'un essaye de le réutiliser après, la DB n'a plus d'entrée -> on retournera 403. On pourrait à ce moment logguer "tentative de reuse tokenId X pour user Y" pour enquêter. On génère un nouveau refresh et on le stocke. L'AT est renvoyé en JSON, le front le stocke en mémoire. Notez qu'on a un jti sur le refresh token, c'est important pour distinguer différents tokens d'un même user. On pourrait aussi stocker un champ `used` booléen au lieu de delete, pour garder une trace des anciens tokens utilisés (dépend de combien de temps on veut garder l'historique).

Exemple de mauvaise gestion (DON'T) :

- **Refresh token statique** : ne jamais changer le refresh token une fois créé, sauf au login. Ça veut dire qu'il peut être volé et utilisé silencieusement.
- **Pas de vérif en BD** : si vous aviez choisi de ne pas stocker du tout les refresh tokens (certains stateless ne stockent pas, et la révocation est alors très compliquée), vous ne pourriez pas invalider facilement. Ce modèle sans persistance marche mal en pratique car on ne peut pas logout proprement sans garder une blacklist quelque part.
- **Access token trop long** : si vous aviez mis le access token 7j et pas de refresh, ce serait catastrophique (vol = accès une semaine entière). Donc garder short.
- **Exposer le refresh token au JS** : si vous aviez stocké refresh en localStorage, un XSS l'aurait pris. HttpOnly cookie évite ça.

6. Security Misconfiguration – Configuration HTTP, CORS, limitations, erreurs

Beaucoup a été couvert globalement, mais allons plus en détail technique :

- **Helmet & HTTP Headers** : On a cité la liste par défaut [17] [18]. Assurez-vous d'appeler `app.use(helmet())` **avant** vos routes dans Express. Par défaut tout est activé, mais vérifiez les console warnings (par ex, Helmet pourrait se plaindre si aucune CSP n'est définie, selon la version). Vous pouvez ajouter des directives CSP via `helmet.contentSecurityPolicy`. Un exemple minimal CSP pour une SPA React:

```
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "'unsafe-inline'", "'unsafe-eval'"], //
```

```
Note: create-react-app en dev nécessite 'unsafe-eval'. En prod on peut
l'enlever.
      connectSrc: ["'self'", "api.trusteddomain.com"], // si appels API
externes
      imgSrc: ["'self'", "data:"],
      styleSrc: ["'self'", "'unsafe-inline'"], // Tailwind injecte du
style inline
      objectSrc: ["'none'"],
      upgradeInsecureRequests: [], // ajoute la directive pour auto
passer en https si http (utile avec HSTS)
    }
  }
}));
```

La CSP doit être adaptée à vos besoins (par ex si vous chargez Google Fonts ou autres ressources externes, il faut les autoriser). CSP est puissante mais complexe – si pas parfaitement configurée, il vaut mieux en mettre une simple "par défaut" que rien. Outre CSP, assurez-vous que **Strict-Transport-Security** est envoyé (Helmet le fait) – en production sur votre domaine public, cela indique aux navigateurs de toujours utiliser HTTPS (et par ex, de ne pas autoriser de connexion HTTP même si l'utilisateur tape http://, après la première visite). Un exemple: `Strict-Transport-Security: max-age=63072000; includeSubDomains; preload`. Ce header doit être envoyé *uniquement sur HTTPS*, et idéalement dès le lancement.

- **Cookie Security** : Comme mentionné, SameSite=Strict sur refresh token, c'est bien. Si un jour vous hébergez le front sur un domaine différent du back (SPA sur autre domaine), vous devrez passer SameSite=None + Secure pour permettre l'envoi du cookie cross-site via fetch (avec `credentials: include`), ce qui ouvre une surface CSRF. Mais si c'est domaine distinct plus https, et que le back accepte seulement ce domaine (via CORS origin), ça limite. A étudier selon déploiement. Pour l'instant, j'imagine front et back seront sur le même domaine (ou du moins même site) en prod – souvent on fait api.mondomaine.com et app.mondomaine.com (sous-domaines => ça compte comme *même site* si eTLD+1 identique ? Non, SafeSite Lax/Strict considère sous-domaines comme même site je crois car c'est eTLD+1 identique). À vérifier, mais SameSite=Strict sur domain=api.monsite.com cookie et front venant de monsite.com peut ne pas envoyer le cookie car différent sous-domaine. Dans ce cas, configurer `cookie.domain = ".mondomaine.com"` ou utiliser SameSite=None. Bref, la config CORS+cookie doit être testée en environnement réaliste.
- **CORS configuration** : Utilisez le package `cors` d'Express. Par exemple:

```
const corsOptions = {
  origin: 'http://localhost:
5173', // en dev, et en prod mettre l'URL du front
  credentials: true, // pour autoriser envoi de cookie
  methods: ['GET','POST','PUT','DELETE'], // méthodes permises
  allowedHeaders: ['Content-Type', 'Authorization'] // les en-têtes que
le front peut envoyer
};
app.use(cors(corsOptions));
```

Comme vu, ne jamais mettre origin: '' *en même temps que credentials: true* [19] *. Soit* '' sans credentials (API publique sans cookies), soit spécifique avec. Egalement, vous pouvez restreindre

les headers exposés, etc. Mais le minimun ci-dessus est bon. Testez bien que les cookies de refresh passent (il faut fetch avec `credentials: 'include'` côté front).

- **Rate Limiting / Bruteforce** : Au-delà de login/refresh, envisagez aussi un rate limit global pour éviter un flood sur l'API qui saturerait le serveur. Par exemple, limiter chaque IP à X requêtes par minute globalement (sauf peut-être endpoints statiques). Librairie `express-rate-limit` usage:

```
const generalLimiter = rateLimit({ windowMs: 60*1000, max: 100 }); //
100 req/min par IP
app.use('/api/', generalLimiter);
const authLimiter = rateLimit({ windowMs: 15*60*1000, max: 10 }); // 10
tentatives/15min
app.use('/api/auth/login', authLimiter);
```

  Ajustez les chiffres. Attention à ne pas bloquer des appels légitimes (100 par min peut être trop bas si l'appli fait plein de petites requêtes). Vous pouvez affiner par route critique (login, refresh, password reset). Pensez aussi au *captcha* si vous subissez des attaques de mot de passe par force brute. Au bout de X échecs, vous pourriez demander un recaptcha ou autre (pour un service santé c'est envisageable pour protéger les comptes).
- **Error Handling & Info Leak** : On en a parlé. Concrètement, en Express, enlevez `app.get('env')` = 'development' en prod (en dev c'est OK d'avoir stacktrace). On peut utiliser `app.use(helmet.hidePoweredBy())` pour enlever X-Powered-By: Express (Helmet le fait). Ne renvoyez jamais d'erreur SQL ou stacktrace. A la place, logguez `console.error(err)` et renvoyez `res.status(500).send("Internal Server Error")`. Pour les erreurs d'auth, utilisez 401 sans préciser si l'email existe ou pas. Idem pour reset password: vous pouvez toujours retourner un 200 "Email envoyé si le compte existe" même si rien n'a été envoyé, pour ne pas révéler que l'email n'est pas dans la base. Les messages d'erreur front peuvent être plus détaillés mais jamais inclure d'infos secrètes. En cas d'erreur de validation (400), vous pouvez quand même dire quel champ est invalide, c'est user-friendly et pas problématique. Juste pas d'infos système.
- **HTTPS partout** : Assurez-vous que l'appli sera servie en HTTPS en prod. Si vous avez des environnements mobiles ou autres, pareil. Le header HSTS mentionné aidera après coup.
- **Configuration du serveur** : Au-delà de l'app elle-même, vérifier les config du serveur web/proxy (NGINX par ex) – activer TLS 1.2+, désactiver anciens chiffrement, etc. Ce n'est pas côté code, mais fait partie de "Security Misconfiguration". Utilisez Qualys SSL Labs pour tester votre déploiement TLS quand ce sera prêt.
- **Database config** : Veillez à ce que la connexion PostgreSQL ne soit pas exposée au public (firewall), utiliser des comptes DB avec les droits minimum (ex: l'application DB user ne devrait pas être superuser, juste les privileges nécéssaires sur les tables).
- **Paramètres divers** : Désactiver listing de répertoires sur le serveur de fichiers, s'assurer que les secrets (.env) ne fuitent pas (ne pas servir des fichiers de config).
- **Logs** : Configurer un format de log qui n'expose pas d'info sensible. Par ex, ne pas logger les mots de passe en clair (même en debug). Ni les tokens. Filtrez les logs HTTP pour enlever Authorization header ou cookies.
- **Third-party** : Faire attention aux outils tiers que vous intégrez (ex: si vous utilisez Google Analytics ou autre script, ça peut potentiellement injecter du script, respectez votre CSP ou ajoutez-le en autorisé).
- **Admin protection** : Si vous avez un admin panel, pensez à le mettre sur une URL distincte et ajouter une auth multi-facteur ou un filtrage IP si possible. Juste par excès de zèle, car admin a

beaucoup de pouvoir (approuver médecins, etc., donc il a un compte admin dans app, qui est JWT protégé de base, mais on peut rajouter des couches comme 2FA).

Exemple config Helmet/CORS (DO) :

```
const helmet = require('helmet');
const cors = require('cors');
app.use(helmet());
app.use(cors({
  origin: 'https://mon-app-frontend.com',
  credentials: true
}));
```

Ceci en place, plus, plus bas, un handler d'erreurs :

```
// Global error handler
app.use((err, req, res, next) => {
  console.error(err.stack || err); // log interne
  if (!res.headersSent) {
    res.status(500).json({ message: 'Une erreur est survenue. Nous
investiguons.' });
  }
});
```

Le message est générique. On peut personnaliser en 404 aussi : `app.use((req,res) => res.status(404).send("Not Found"));`.

Exemple de mauvaise config (DON'T) :

- Omettre Helmet : pas de headers de protection – augmente risques de XSS (pas de CSP), de clickjacking (pas de X-Frame-Options), etc.
- CORS mal réglé :

  ```
  app.use(cors({ origin: '*', credentials: true }));
  ```

  Ceci est invalide de toute façon (navigateur bloque), mais si c'était possible, ce serait une faute, car n'importe quel site pourrait faire des requêtes avec les cookies de vos utilisateurs, menant à CSRF.
- Pas de rate limit : site susceptible aux brute force.
- Laisser `NODE_ENV=development` en prod : Express enverra les stacktraces dans les réponses d'erreur.
- Avoir un `.env` accessible via une route statique (ex: si on sert tout le dossier par mégarde).
- Utiliser des identifiants par défaut : ex, ne pas changer le secret JWT fourni par un boilerplate (ça s'est vu).
- Oublier de mettre à jour les config quand vous déployez sur un domaine réel (ex: SameSite mis à Strict, mais front et back sont sur domain différents -> le refresh ne marche plus car cookie non envoyé – l'appli pourrait dysfonctionner).

- Laisser les ports ouverts : assurez-vous que le serveur Express est derrière un proxy, ou du moins, fermé sur les ports inutiles (ça dépasse peut-être votre rôle de dev, mais mentionnons-le).

7. Vulnerable Dependencies – Gestion proactive des dépendances

Le code que vous écrivez peut être parfait, une faille peut venir d'une simple librairie dans votre `node_modules`. Quelques conseils concrets :

- **npm audit** : Comme mentionné, utilisez-le régulièrement. En dev, `npm install` l'exécute automatiquement et signale les vulnérabilités. Prenez ces rapports au sérieux : s'il y a des **vulnérabilités élevées/critiques** dans vos dépendances, ne mettez pas en prod sans régler le problème. Soit en mettant à jour la lib (`npm audit fix` fait parfois la mise à jour mineure automatiquement), soit en appliquant un patch si disponible, ou en cherchant une alternative.
- **Dependabot** : Activez-le via GitHub Settings du repo ou le fichier `dependabot.yml`. Il fera des MR automatiques quand une nouvelle version d'une dépendance corrige une faille connue. Ça couvre front et back tant que c'est un repo JS. D'après la communauté, Dependabot utilise la même base de vulnérabilité que npm audit [20], donc l'un ou l'autre retrouvera les mêmes soucis. L'avantage de Dependabot est la **mise à jour automatique** avec test CI.
- **Snyk/OSSAR** : Vous pouvez aussi utiliser Snyk (cli ou GitHub Action) qui scanne le projet et peut trouver des vulnérabilités ou mauvaises config. Snyk offre une base parfois plus large, notamment sur les vulnérabilités de logique, ou il peut monitorer en continu.
- **Mise à jour régulière** : Programmez peut-être un créneau mensuel ou bimensuel pour "passer les updates". Les dépendances JS évoluent vite. Ne pas mettre à jour pendant 1 an, c'est s'exposer à plein de failles. Par exemple, Express v4 a pu avoir des patches de sécurité, etc.
- **Surveiller les annonces** : abonnez-vous aux flux de sécurité des principaux composants (Node.js lui-même – tenez-le à jour LTS, PostgreSQL – assurez-vous de patcher la version serveur si admin devops s'en occupe, React – suit la version, etc.).
- **Frontend dependencies** : Même logique, d'autant plus qu'un vulnérabilité frontend peut mener à XSS (ex: une lib de parsing markdown qui permet XSS, etc.).
- **Transitives** : Parfois la faille est dans une sous-dépendance. npm audit le détecte aussi. Il arrive qu'on ne puisse pas facilement mettre à jour car c'est bloqué par une version imposée par la lib parent. Dans ce cas, voyez s'il existe un `resolutions` possible ou un fork patché. Ou faites pression (ouvrir une issue) sur le mainteneur de la lib pour qu'il mette à jour sa dépendance.
- **Build du frontend** : Si vous utilisez Vite, c'est bien plus simple et sécurisé que des bundlers plus anciens. Assurez-vous quand même de configurer la production build pour qu'elle supprime les messages de debug, etc.
- **Dev Dependencies** : Celles-ci ne vont pas en prod (sauf s'il y a une erreur de packaging). Mais parfois, elles peuvent être utilisées par des outils (ex: webpack dev server faillible => attention en dev local sur réseau).
- **Audit du code** : Au-delà des packages, pensez à un code scan (par ex, GitHub CodeQL, ou SonarQube) pour repérer des patterns dangereux dans votre code (le genre de choses qu'on discute ici).
- **Permissions minimales** : Si votre appli utilise Docker images, prenez les images officielles à jour (Node:18 par ex), et ne pas installer des packages système inutiles. Chaque composant externe = potentielle faille.
- **Monitoring des dépôts** : Activez GitHub Advanced Security (si dispo gratuitement maintenant pour les publics) qui fait du secret scanning (vérifie que vous n'avez pas leaké des clés par mégarde).
- **Exemple concret** : Admettons qu'une version de `jsonwebtoken` ait une faille (ça s'est vu). Dependabot vous fera une PR "update jsonwebtoken from x.x to y.y to fix CVE-XXX". Vous la testez (s'assurer qu'il n'y a pas de breaking change) puis mergez. Autre exemple front : une

librairie de masque de champ de saisie avait du code malveillant inséré (cas d'event-stream en 2018). Ce genre de supply chain attack est plus dur à détecter automatiquement. Mais en gardant vos libs à jour, vous réduisez la fenêtre durant laquelle une version compromise pourrait vous affecter (les attaques sont souvent détectées et la lib retirée/nette version publiée rapidement).

- **Ne pas ignorer les vulnérabilités "modérées"** : Même si npm audit indique "low" ou "moderate", prenez le temps de comprendre si ça peut vous impacter. Par ex, une vuln dans `minimatch` moderate, peut-être sans gravité directe. Mais une vuln "prototype pollution" dans Lodash moderate peut quand même mener à XSS dans certaines conditions. Donc ne pas balayer d'un revers de main, essayez de corriger un max.
- **Backport** : Si vous ne pouvez pas upgrader car changement majeur, voyez s'il y a un patch. Exemple, Express 4.x a une faille, Express 5 n'est pas encore stable ou incompatible – dans ce cas, voyez les conseils de la communauté (souvent, ils publient un 4.x patch).
- **Tests** : Ayez une bonne suite de tests (unit et integration) pour pouvoir mettre à jour les dépendances en confiance (les tests vous diront si quelque chose casse après upgrade).
- **Dependencies status** : Utilisez `npm outdated` pour voir les versions. Mettez à jour dans la mesure du possible.

Exemple (DO) : Intégration de Dependabot – vous ajoutez un fichier `/.github/dependabot.yml` :

```
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/backend"
    schedule:
      interval: "daily"
  - package-ecosystem: "npm"
    directory: "/frontend"
    schedule:
      interval: "daily"
```

Ainsi, chaque jour il check. Vous pouvez mettre weekly pour moins de bruit.

Exemple (DON'T) :

- Ignorer un rapport npm audit critical en se disant "ça va". Par ex, un report sur `pg` (imaginons) qui dit que la version utilisée a une faille de authent. Si on ne met pas à jour, un attaquant pourrait l'exploiter.
- Utiliser des forks non maintenus ou des libs douteuses de github sans audit.
- Laisser des dépendances non utilisées (faites le ménage de temps en temps avec `depcheck` par ex). Moins de dépendances = moins de surface.
- Ne pas mettre à jour Node : Node lui-même publie des mises à jour de sécurité (ex: Node 18.7 vs 18.12 LTS etc.). Surveillez ces annonces (newsletter Node).

8. Monitoring & Anomaly Detection – Supervision continue et journaux d'audit

Une fois l'application sécurisée, il faut assumer qu'une attaque peut quand même survenir (zero-day, credentiels volés, etc.). D'où l'importance de **détecter et réagir**.

Que monitorer ?

- **Authentification :**
- Nombre de tentatives échouées par compte et par IP. Un pic d'échecs peut signaler un brute force en cours. Vous devriez déjà limiter, mais monitorer en plus (ex: 50 tentatives sur 10 comptes différents depuis la même IP, on pourrait bloquer l'IP au niveau firewall ou WAF).
- Succès de login géographiquement éloignés dans un court laps de temps pour le même compte (indicatif de partage de credentials ou vol). Par ex, un patient se log depuis la France puis 1h après depuis la Russie – c'est louche. Vous pourriez alerter l'utilisateur ou invalider tous tokens et demander reset password. Ceci nécessite de logguer l'IP/lieu de chaque login.
- Utilisation des refresh tokens : idéalement logguer "refresh utilisé pour user X, device Y, heure". Si un refresh token volé est utilisé (rejeté par rotation), c'est critique : générez une alerte haute priorité (puisque ça signifie probablement qu'un attaquant a un cookie de l'utilisateur).
- **Accès aux données sensibles :**
- Par ex, un médecin qui consulte 100 dossiers patients en 5 minutes – peu probable dans un usage normal, peut-être un script malveillant ou un compte compromis siphonnant la base. Idem pour un patient qui télécharge tous ses documents en rafale – c'est peut-être lui, ou quelqu'un d'autre.
- Tracez les lectures de données sensibles (sans forcément tout logguer – les logs peuvent devenir énormes). Au moins logguer "Médecin X a affiché la liste des documents du patient Y le [date]". Si c'est fait modérément, pas de souci. Si vous voyez dans les logs qu'en une journée un médecin a accédé à *tous* les patients de la base (pas seulement les siens), alors il y a eu contournement de sécurité ou malveillance interne.
- **Téléversements de fichiers :**
- Un grand nombre de fichiers uploadés en peu de temps, ou des fichiers d'un type non attendu (si par malheur un .html passait).
- La taille totale stockée qui augmente drastiquement (attaque DoS par remplissage de disque ?). Vous pouvez mettre en place des quotas par utilisateur (ex: un patient peut avoir max 100 Mo de documents). Si atteint, logguer ou prévenir.
- **Modifications et actions critiques :**
- Changement de mot de passe, email modifié – logguez ces événements (qui, quand, old->new?), et possiblement notifier l'utilisateur par email ("si ce n'est pas vous, contactez support").
- Suppression de compte ou de document – logguez pour audit (sait-on jamais, pour restaurer ou enquêter).
- Changement de rôle (si admin promeut un médecin, etc.) – tracer.
- **Intégrité des logs** :
- Stockez les logs de manière sécurisée, pour qu'un attaquant ne puisse pas simplement effacer ses traces si jamais il réussit à escalader sur le serveur. Par exemple, envoyez les logs vers un serveur distant ou un service managé (Datadog, CloudWatch, ELK) où l'attaquant n'a pas facilement accès.
- Mettez en place une rotation des logs pour ne pas saturer le disque, mais sauvegardez-les suffisamment longtemps (en santé, conserver logs d'accès 6 mois à 1 an est prudent, checker régulation locale).
- **SIEM** :
- Si l'application prend de l'ampleur avec de nombreuses données sensibles, intégrer à un SIEM permet de corréler avec d'autres événements (pare-feu, système d'exploitation, etc.). Par exemple, si un même IP tente du brute force sur l'appli web *et* scanne les ports du serveur, un SIEM peut faire le lien. Pour démarrer, ce n'est peut-être pas nécessaire, mais c'est bon de l'avoir en tête.
- Des solutions open-source existent (ELK + Elastalert, Wazuh / OSSEC pour host IDS, etc.).
- **Alerting** :

- Configurez des alertes emails/Slack pour certains triggers : X logins échoués -> alerte admin sécurité. Reuse de refresh détecté -> alerte critique devops. Taux d'erreurs 500 soudain > 5% -> alerte (peut être une attaque en cours provoquant crash).
- Un monitoring de disponibilité type uptime robot est utile aussi (pas une menace, mais pour savoir si une attaque DoS a fait tomber l'appli).
- **Analyse régulière** :
- Passez en revue les logs d'accès administrateur. Par exemple, l'admin ne doit en théorie pas accéder aux données patients, donc s'il essaye d'appeler un endpoint patient, soit c'est un bug UI, soit il tente de faire quelque chose de potentiellement non autorisé – à vérifier.
- De même, auditez les comptes médecins : Vérifiez qu'un médecin n'a pas accédé aux données en dehors de ses patients. Si oui, c'est soit un bug de vos contrôles (il ne devrait pas y arriver) soit un acte malveillant interne.
- Gardez une trace des changements de configuration système (infrastructure as code ou autre) pour investiguer en cas de problème.
- **Plan de réponse** : Ce n'est pas monitoring en soi, mais ayez un plan : si une alerte critique se déclenche (ex: suspicion de compte compromis), quelle est la procédure ? (Ex: suspendre le compte ou réinitialiser mot de passe, contacter le propriétaire, analyse forensics des actions faites avec ce compte, etc.). Étant dans la santé, notifier possiblement les autorités/data protection officer si une violation de données est confirmée (légalité RGPD/HIPAA).

Outils open source suggérés :

- **Winston** (pour logger en Node avec différents transports).
- **Morgan** (middleware express pour logs HTTP basiques).
- **OSSEC/Wazuh** (IDS/IPS host-based).
- **Fail2ban** (peut lire les logs auth et bannir IP au niveau firewall).
- **Elastic Stack (ELK)** : très courant pour centraliser logs et créer des tableaux de bord + alertes (via Kibana watcher ou Elastalert).
- **Prometheus + Grafana** : pour monitoring métriques système et custom (ex: nb de requêtes, latence). Grafana peut alerter.
- **Sentry** : pour collecter erreurs applicatives front et back (non-sécurité, mais pour améliorer robustesse).
- **Auditd** : sur le serveur Linux, pour suivre par ex modifications non souhaitées de fichiers système.

Exemple (DO) : Log d'audit pour accès dossier patient :

```
// Dans le contrôleur qui retourne les infos patient au médecin
if (req.userRole === 'medecin') {
  console.log(`AUDIT: Le Dr.${req.userId} a consulté le profil du patient $
{patientId} le ${new Date().toISOString()}`);
}
```

Ça écrit dans la console (redirigez la console vers un fichier de log via Winston avec rotation journalière).

Un script externe ou un administrateur peut plus tard analyser ces lignes pour vérifier les abus.

Exemple d'alerte (pseudo) :

```
// Exemple simplifié d'alerte sur échecs de login
let failedLogins = {}; // objet en mémoire: { ip: count }
app.post('/api/auth/login', async (req, res) => {
  const ip = req.ip;
  const ok = await checkCredentials(req.body.user, req.body.pass);
  if (!ok) {
    failedLogins[ip] = (failedLogins[ip] || 0) + 1;
    if (failedLogins[ip] > 10) {
      alertSecurityTeam(`IP ${ip} has ${failedLogins[ip]} failed logins in
short time`);
      failedLogins[ip] = 0; // reset or implement time window
    }
    return res.status(401).send('Invalid creds');
  }
  // ... login success ...
  failedLogins[ip] = 0;
});
```

En prod, on ferait ça plus proprement (en base Redis par ex, pour persistance, et un système d'alerte plus robuste), mais c'est l'idée.

DON'T:

- Ne pas monitorer du tout : on serait aveugle à une intrusion jusqu'à ce que quelqu'un se plaigne.
- Collecter des tonnes de données sans personne pour les regarder : trouver le bon niveau d'agrégation. Il vaut mieux 5 alertes utiles par semaine que 5000 lignes de log par jour que personne ne lit. Configurez des filtres/alertes pour extraire le signal de la masse.
- Laisser les logs sur le serveur de prod sans backup : si le serveur crashe ou est compromis, vous perdez les traces de ce qui s'est passé. Externalisez-les.
- Ignorer des alertes : si un système d'alarme spam beaucoup, on tend à l'ignorer. Ajustez pour qu'une alerte = action.
- Enregistrer des données sensibles en clair dans les logs : par ex, pas de numéro de CB en clair, pas de mot de passe (même hashé ce n'est pas utile de logguer).
- Oublier la dimension légale : en Europe, les journaux d'accès aux données de santé doivent être conservés et consultables sur demande, et bien sécurisés. Assurez-vous de respecter les exigences (par ex, la CNIL en France exige qu'on puisse fournir l'historique des accès au dossier médical d'un patient si demandé – d'où l'importance d'avoir ces logs d'audit).

Conclusion : En appliquant rigoureusement ces recommandations, vous allez considérablement renforcer la sécurité de votre application de santé. En résumé, **vérifiez les autorisations** à chaque requête, **validez toutes les entrées** (texte, fichiers, requêtes), **protégez vos tokens** par des bonnes pratiques JWT, **durcissez la configuration** du serveur (headers, CORS, limitations), **maintenez votre environnement** à jour (dépendances, patches), et **surveillez activement** l'activité pour réagir vite en cas d'incident. Cette approche défensive en profondeur (« defense-in-depth ») est essentielle pour une plateforme manipulant des données médicales, où la confidentialité et l'intégrité des informations sont critiques.

En suivant ces DOs et évitant les DON'Ts, vous créez des couches successives de sécurité : même si une couche faillit, la suivante prend le relais. Continuez à vous tenir informé des nouvelles menaces et bonnes pratiques (la sécurité est un domaine évolutif), et faites régulièrement des tests de pénétration

de votre application (idéalement par des tiers) pour valider l'efficacité de ces mesures dans le monde réel. Bon développement sécurisé ! [2] [10]

---

[1] [23] [24] Preventing broken access control in express Node.js applications | Snyk
https://snyk.io/blog/preventing-broken-access-control-express-node-js/

[2] [3] Queries – node-postgres
https://node-postgres.com/features/queries

[4] [5] [6] [7] React XSS Guide: Understanding and Prevention
https://www.stackhawk.com/blog/react-xss-guide-examples-and-prevention/

[8] [9] node.js - How to validate file extension with Multer middleware - Stack Overflow
https://stackoverflow.com/questions/60408575/how-to-validate-file-extension-with-multer-middleware

[10] Unrestricted File Upload | OWASP Foundation
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

[11] [12] [13] [14] [15] [16] [25] [26] [27] What Are Refresh Tokens and How to Use Them Securely | Auth0
https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/

[17] [18] How to use Helmet.js to secure your Node.js Express app - DEV Community
https://dev.to/codexam/how-to-use-helmetjs-to-secure-your-nodejs-express-app-4b1l

[19] CORS: Cannot use wildcard in Access-Control-Allow-Origin when ...
https://stackoverflow.com/questions/19743396/cors-cannot-use-wildcard-in-access-control-allow-origin-when-credentials-flag-i

[20] [21] security - Does `npm audit` add any value when using `dependabot` - DevOps Stack Exchange
https://devops.stackexchange.com/questions/12017/does-npm-audit-add-any-value-when-using-dependabot

[22] raw.githubusercontent.com
https://raw.githubusercontent.com/AssilM/CarnetDeSante/main/backend/src/middlewares/ownership.middleware.js