1. Introduction

- Contexte du projet : Présentation de Z-Event et des objectifs de l'application.
- Objectifs de l'application : Résumer brièvement les principales fonctionnalités de gestion du matériel, du monitoring et de la gestion des streamers.

2. Architecture de l'application

- Diagramme de l'architecture (si possible).
- Technologies utilisées (Base de données, back-end, front-end, etc.).
- API (s'il y en a) : quelles fonctionnalités sont exposées via des endpoints ?

3. Modèle de données

- Description des différentes tables de la base de données.
- Schéma de la base de données avec les relations entre les tables.

4. Fonctionnalités

Pour chaque fonctionnalité :

- Description des objectifs de la fonctionnalité.
- Composants impliqués (base de données, back-end, front-end).
- Exemple d'utilisation (exemple de requête API, ou description du processus utilisateur).
- **Gérer le stock de matériel**
- **Monitorer l'événement**
- **Gestion des streamers et des thématiques**

5. Diagrammes UML

- Diagramme de classes.
- Diagramme de séquence ou d'activité pour les interactions principales.

6. Installation et configuration

- Instructions pour installer l'application en local.
- Dépendances (frameworks, bibliothèques, etc.).
- Instructions pour déployer l'application sur un serveur (si nécessaire).

7. Sécurité

- Mesures de sécurité prises (authentification, gestion des permissions).
- Sécurisation des flux de données (si nécessaire, HTTPS, JWT, etc.).

8. Tests et validation

- Stratégie de test (tests unitaires, tests d'intégration).
- Environnement de test.

9. Maintenance et amélioration future

- Informations sur la maintenance de l'application.
- Idées pour des futures fonctionnalités ou améliorations

Documentation Technique de l'Application de Gestion pour Z-Event

1. Réflexions Initiales Technologiques sur le Sujet

1.1. Objectifs du Projet

L'application a pour but d'améliorer la gestion interne de Z-Event en automatisant plusieurs processus clés :

- Gestion du **stock de matériel**.
- **Monitoring en temps réel** des événements (état des streams).
- Gestion de la **disponibilité des streamers** et de la thématique de leurs lives.

1.2. Choix Technologiques

Les choix technologiques sont motivés par la nécessité d'une application performante, intuitive et capable de traiter un volume important de données en temps réel :

- **Back-end** : Utilisation de **Node.js** pour sa capacité à gérer des événements en temps réel de manière efficace. Un framework comme **Express.js** permettra de créer facilement une API RESTful pour gérer les données de l'application.
- **Front-end** : Utilisation de **React.js** pour une interface utilisateur réactive et dynamique, qui offre une bonne gestion des états de l'application.
- **Base de données** : **MySQL** ou **PostgreSQL** pour stocker les données relatives aux matériels, streamers et monitoring. Ces bases relationnelles sont idéales pour gérer les relations entre les entités comme le stock et les streamers.
- **API** : Le back-end expose une API RESTful qui permet aux différentes parties de l'application (front-end, monitoring) d'interagir avec la base de données.
- **Authentification et sécurité** : Pour sécuriser l'application, un système d'authentification basé sur **JWT** (JSON Web Token) sera implémenté. Cela permettra de gérer les permissions et garantir que seules les personnes autorisées ont accès aux données.

2. Configuration de l'Environnement de Travail

2.1. Prérequis

Avant de commencer, assurez-vous d'avoir installé les outils suivants :

- **Node.js** (pour le serveur back-end) : v14+.
- **NPM** ou **Yarn** (gestionnaire de packages).
- **React.js** pour le front-end.
- **MySQL/PostgreSQL** pour la base de données.

2.2. Configuration Initiale

1. **Cloner le dépôt du projet :**

```
bash
Copier le code
git clone https://github.com/nom-utilisateur/z-event-management.git
cd z-event-management
```

2. **Installation des dépendances :** Pour le back-end :

```
bash
Copier le code
cd backend
npm install
```

Pour le front-end :

```
bash
Copier le code
cd frontend
npm install
```

3. **Configuration de la base de données :**
    a. Créez une base de données MySQL/PostgreSQL appelée `zevent_db`.
    b. Configurez le fichier `.env` pour inclure les informations de connexion à la base de données :

```
bash
Copier le code
DB_HOST=localhost
DB_USER=root
DB_PASS=password
DB_NAME=zevent_db
```

4. **Démarrage du projet :** Lancer le serveur back-end :

```
bash
Copier le code
npm start
```

Lancer le front-end :

```
bash
Copier le code
npm start
```

3. Modèle Conceptuel de Données (ou Diagramme de Classe)

Voici le modèle conceptuel de données qui structure les différentes entités de l'application.

3.1. Schéma des tables principales :

a. Matériel

Matériel_ID	Nom	Quantité	État	Date_d_achat
1	Caméra	10	Bon	2023-01-15
2	Microphone	20	Moyen	2022-11-10

b. Streamer

Streamer_ID	Nom	Disponibilité	Thématique
1	Streamer 1	2024-10-21 15h-18h	Jeu vidéo
2	Streamer 2	2024-10-22 18h-22h	Atelier cuisine

c. Monitoring

Monitoring_ID	Streamer_ID	Début	Durée	Qualité
1	1	2024-10-21 15h00	3h	Bonne
2	2	2024-10-22 18h00	4h	Moyenne

4. Diagramme d'Utilisation et Diagramme de Séquence

4.1. Diagramme d'Utilisation (Use Case)

Le diagramme d'utilisation permet de visualiser les interactions entre les utilisateurs (administrateurs et streamers) et les fonctionnalités du système.

Acteurs :

- **Administrateur** : Gère le stock, surveille les streamers, ajuste les plannings.
- **Streamer** : Consulte son planning et ajuste la thématique de son live.

```
plaintext
Copier le code
+---------------------+       +--------------------+
| Administrateur      |       | Streamer           |
+---------------------+       +--------------------+
       |                             |
       |                             |
       v                             v
+-----------------------+     +----------------------+
| Gérer stock matériel  |     | Consulter planning   |
+-----------------------+     +----------------------+
       |
       v
+-----------------------+
| Monitorer événement   |
+-----------------------+
```

4.2. Diagramme de Séquence

Le diagramme de séquence montre les interactions entre les composants lors de la mise à jour du stock de matériel.

1. **Administrateur** : Ajoute ou modifie un article dans le stock.
2. **Front-end** : Envoie une requête API POST/PUT au serveur.
3. **Back-end** : Le serveur traite la requête et met à jour la base de données.
4. **Base de données** : Mise à jour des informations du stock.

5. Documentation de Déploiement

5.1. Étapes du Déploiement

Le déploiement de l'application se déroule en plusieurs phases :

1. **Configuration du serveur**
    a. Utilisation d'un VPS ou d'un service cloud (comme AWS, Azure, ou DigitalOcean).
    b. Installer **Node.js** et le gestionnaire de base de données choisi (MySQL ou PostgreSQL).
2. **Installation des dépendances**
    a. Cloner le dépôt sur le serveur.
    b. Installer les dépendances avec `npm install` ou `yarn install`.
3. **Configuration de l'environnement**
    a. Configurer les variables d'environnement (comme dans le fichier `.env`) pour connecter la base de données en production.
4. **Lancer l'application**
    a. Utilisation de **PM2** pour gérer les processus Node.js en production :

```
bash
Copier le code
pm2 start server.js
```

5. **Sécuriser l'application**
    a. Configurer un certificat SSL pour sécuriser les communications (via **Let's Encrypt** par exemple).
    b. Utiliser un pare-feu pour restreindre les accès au serveur.

5.2. Surveillance et Maintenance

- **Logs** : Configurer un système de logs pour surveiller les erreurs et les performances.
- **Sauvegarde des données** : Mettre en place une stratégie de sauvegarde régulière de la base de données.

Cahier des Charges de l'Application de Gestion pour le Z-Event

1. Contexte du Projet

Le **Z-Event** est un événement caritatif majeur où des streamers se mobilisent pour collecter des dons en faveur d'associations. Avec une organisation aussi complexe, la gestion des aspects techniques et humains est primordiale pour assurer le succès de l'événement. Le Z-Event souhaite donc moderniser et améliorer la gestion de son matériel, le suivi en temps réel des streams, ainsi que la gestion des plannings des streamers et des thématiques de leurs lives.

2. Expression du Besoin

Problématique :

Le Z-Event rencontre des difficultés à gérer manuellement les multiples aspects logistiques et techniques, ce qui peut entraîner des pertes d'efficacité et des erreurs dans l'organisation. Afin de résoudre ce problème, l'organisation souhaite mettre en place une application capable de gérer ces trois domaines essentiels :

- Le **stock de matériel** utilisé pour l'événement.
- Le **monitoring en temps réel** des streams pour suivre les performances techniques et anticiper les problèmes.
- La **gestion des disponibilités des streamers** et des thématiques de leurs lives.

Objectifs :

- Améliorer la traçabilité et la gestion du matériel.

- Avoir une vue en temps réel des performances des streams et détecter rapidement tout dysfonctionnement.
- Permettre une gestion centralisée et automatisée des plannings et thématiques des streamers pour une meilleure coordination.

3. Spécifications Fonctionnelles

L'application doit répondre aux fonctionnalités suivantes :

3.1. Gestion du Stock de Matériel

- **Fiche produit** : Pour chaque matériel, l'application doit enregistrer et afficher les informations suivantes : nom, type de matériel, quantité, état (neuf, usé, etc.), et date d'achat.
- **Modification du stock** : Possibilité d'ajouter, modifier ou supprimer du matériel, ainsi que de mettre à jour les quantités disponibles.
- **Alerte de stock bas** : Envoi d'une notification ou affichage d'une alerte lorsque le niveau d'un matériel tombe en dessous d'un seuil défini.

3.2. Monitoring en Temps Réel

- **Suivi des streams** : Afficher en temps réel l'état des streams (actif, déconnecté, etc.), la qualité du stream (bonne, moyenne, mauvaise), la durée du stream en cours.
- **Alertes techniques** : Si un stream rencontre des difficultés techniques (faible bande passante, interruption), envoyer une alerte pour que les équipes puissent réagir rapidement.
- **Historique des streams** : Conserver un historique des performances des streams pour consultation ultérieure (nom du streamer, date, qualité, durée).

3.3. Gestion des Streamers et de leurs Thématiques

- **Planning des streamers** : Un agenda où sont listés les créneaux horaires des streamers, avec la possibilité de voir les périodes de disponibilité et de programmer des sessions.
- **Thématique des lives** : Pour chaque session de streaming, indiquer la thématique prévue (jeu vidéo, cuisine, etc.) et permettre aux administrateurs et aux streamers de modifier cette thématique si nécessaire.

- **Notifications de disponibilité** : Rappels et notifications pour les streamers lorsqu'une session approche ou qu'une modification est effectuée sur leur planning.

4. Spécifications Techniques

4.1. Front-end

- **Technologie** : L'interface utilisateur sera développée avec **React.js** pour sa réactivité et son efficacité dans la gestion de grandes quantités de données en temps réel.
- **Responsiveness** : L'application doit être utilisable sur plusieurs types de dispositifs (ordinateurs, tablettes, mobiles).

4.2. Back-end

- **Technologie** : Le serveur sera développé en **Node.js** avec le framework **Express.js** pour créer une API RESTful, permettant la gestion des données de stock, des streamers, et du monitoring.
- **API** : Le back-end exposera une API qui permettra au front-end de récupérer et de modifier les données (CRUD sur le matériel, les plannings des streamers, etc.).

4.3. Base de Données

- **Système** : La base de données sera de type relationnelle (MySQL ou PostgreSQL).
- **Schéma** :
    - **Table matériel** : Gère les informations sur le stock de matériel.
    - **Table streamers** : Gère les informations sur les streamers (nom, disponibilité, thématique).
    - **Table monitoring** : Gère l'état et les performances des streams en temps réel.

4.4. Sécurité

- **Authentification** : Le système d'authentification sera géré par **JSON Web Tokens (JWT)** pour protéger les routes critiques de l'API.
- **Permissions** : Différents niveaux d'accès seront définis pour les administrateurs et les streamers (gestion complète ou consultation uniquement).

5. Contraintes et Exigences

- **Performance** : L'application doit pouvoir gérer un grand nombre de streamers et de matériel en simultané sans ralentissement.
- **Fiabilité** : Le monitoring des streams doit être en temps réel avec des alertes immédiates en cas de problème.
- **Scalabilité** : L'application doit être conçue pour évoluer facilement si le nombre de streamers ou la quantité de matériel à gérer augmente.
- **Sécurité** : Les données sensibles doivent être protégées, et l'accès à certaines fonctionnalités limité aux utilisateurs autorisés.

6. Livrables

- Une **application web** fonctionnelle avec une interface utilisateur pour gérer le stock de matériel, le monitoring et les plannings de streamers.
- **Documentation technique** expliquant l'architecture du projet, les choix technologiques, et les processus d'installation et de maintenance.
- **Base de données** opérationnelle avec le schéma relationnel pour stocker les informations nécessaires.
- **Tests unitaires** et tests d'intégration pour valider le bon fonctionnement des différentes fonctionnalités.

7. Délais et Planification

- **Phase de conception** : 2 semaines.
- **Développement du back-end** : 4 semaines.
- **Développement du front-end** : 4 semaines.
- **Phase de tests et ajustements** : 2 semaines.
- **Déploiement final** : 1 semaine.

Spécifications des Technologies Utilisées et Justification des Choix

1. Technologies Utilisées

a. Back-end : Node.js et Express.js

- **Justification du choix** :

- **Asynchronisme** : **Node.js** permet une gestion efficace des opérations asynchrones, ce qui est crucial pour une application de monitoring en temps réel. Cela permet de traiter plusieurs requêtes en parallèle, réduisant ainsi les temps de latence.
- **Communauté active** : Node.js bénéficie d'une vaste communauté qui fournit un soutien continu et de nombreuses bibliothèques open source, facilitant l'ajout de nouvelles fonctionnalités.
- **API RESTful** : Avec **Express.js**, nous pouvons créer une API RESTful rapidement et de manière flexible, facilitant la gestion des interactions entre le front-end et la base de données.

b. Front-end : React.js

- **Justification du choix** :
  - **Interactivité et réactivité** : **React.js** permet de créer une interface utilisateur dynamique avec un rafraîchissement instantané des données sans recharger la page entière, idéal pour un suivi en temps réel des streams et la gestion du stock.
  - **Component-based architecture** : La structure basée sur les composants de React.js permet une meilleure modularité du code, facilitant ainsi les modifications ou ajouts futurs.
  - **Performance** : React.js optimise le DOM via son "Virtual DOM", garantissant une bonne performance même pour les interfaces riches en données.

c. Base de Données : MySQL

- **Justification du choix** :
  - **Modèle relationnel** : Les relations complexes entre les données (streamers, matériel, monitoring) sont bien adaptées à une base relationnelle telle que **MySQL**. Ce modèle garantit l'intégrité des données et une gestion efficace des jointures.
  - **Fiabilité et scalabilité** : MySQL est largement utilisé et optimisé pour des applications évolutives. C'est une solution mature, avec de nombreux outils de gestion et de surveillance disponibles.
  - **Sécurité des transactions** : MySQL garantit la sécurité des transactions et permet de gérer les droits d'accès aux différentes parties de la base de données.

d. Authentification : JSON Web Tokens (JWT)

- **Justification du choix** :
    - **Simplicité et sécurité** : **JWT** offre un moyen sécurisé et simple d'authentifier les utilisateurs via des tokens. Chaque requête API utilise un token JWT, ce qui permet de s'assurer que seules les personnes authentifiées peuvent accéder aux fonctionnalités sensibles.
    - **Stateless** : L'utilisation de JWT rend le système d'authentification "stateless", permettant de déployer facilement l'application sur plusieurs serveurs sans avoir à gérer une session côté serveur.

e. Serveur Web : NGINX

- **Justification du choix** :
    - **Performance** : **NGINX** est un serveur web performant qui peut servir le contenu statique tout en agissant comme un reverse proxy pour notre serveur Node.js.
    - **Gestion des connexions** : Il gère très bien les connexions simultanées et la répartition des charges, ce qui est essentiel pour une application avec plusieurs utilisateurs en temps réel.

f. Gestion des processus : PM2

- **Justification du choix** :
    - **Surveillance et redémarrage automatique** : **PM2** est utilisé pour gérer les processus Node.js, assurant la disponibilité constante de l'application en redémarrant automatiquement les processus en cas de panne.

2. Mise en Place de l'Environnement de Travail

a. Environnement de Développement

- **Choix de l'IDE** : **Visual Studio Code** a été utilisé comme éditeur de code en raison de ses nombreuses extensions pour Node.js et React.js, ainsi que son support intégré de Git et des outils de débogage.
- **Gestionnaire de packages** : Utilisation de **NPM** (Node Package Manager) pour la gestion des dépendances du projet, tant pour le back-end (Express.js) que pour le front-end (React.js). Cela a permis une installation rapide et facile des bibliothèques et des modules nécessaires au projet.

- **Gestion de version** : **Git** a été utilisé pour le contrôle de version, hébergé sur une plateforme telle que **GitHub**, permettant une collaboration efficace, la gestion des branches et des pull requests pour des révisions de code.

b. Configuration de la Base de Données

- **Choix d'un serveur local MySQL** (ou Postgres selon la préférence) pour le développement, configuré pour permettre la migration des schémas de base de données via un outil de migration tel que **Knex.js**.
- **Scripts d'initialisation** : Des scripts SQL ont été mis en place pour créer les tables nécessaires (matériel, streamers, monitoring) et assurer l'intégrité des données à chaque démarrage du projet.

c. Environnement de Production

- **Déploiement avec Docker** : Mise en place de conteneurs **Docker** pour simplifier le déploiement en production et assurer que l'environnement local soit proche de celui de production. Cela garantit une plus grande cohérence des résultats entre les différentes machines de développement.
- **Sécurité des accès à la base de données** : Utilisation de **variables d'environnement** via un fichier `.env` pour stocker les informations sensibles comme les identifiants de connexion à la base de données, évitant ainsi de les exposer dans le code.

3. Mécanismes de Sécurité

a. Sécurisation des Formulaires

- **Validation côté client** : Chaque formulaire (ajout de matériel, mise à jour des informations de streamer) inclut une validation front-end pour s'assurer que les champs requis sont correctement remplis avant d'envoyer les données au serveur.
- **Validation côté serveur** : En plus de la validation front-end, des vérifications côté serveur sont effectuées pour s'assurer que les données envoyées par l'utilisateur sont conformes (formats corrects, valeurs attendues, etc.), évitant ainsi des injections SQL ou des erreurs de type.

b. Prévention des Attaques XSS et CSRF

- **Protection contre le XSS (Cross-Site Scripting)** : Les données entrées par l'utilisateur sont correctement échappées (via des bibliothèques comme **Helmet.js** ou des techniques de sanitation des inputs) avant d'être affichées sur la page web, empêchant ainsi l'exécution de scripts malveillants.
- **Protection contre le CSRF (Cross-Site Request Forgery)** : L'utilisation de **CSRF tokens** garantit que chaque formulaire et chaque requête POST est sécurisé contre les attaques de type CSRF, où un utilisateur non autorisé pourrait tenter d'effectuer des actions à la place d'un autre utilisateur.

c. Authentification et Autorisation

- **JWT** : Chaque utilisateur authentifié reçoit un **token JWT** signé, qui est ensuite envoyé dans l'en-tête de chaque requête API pour vérifier son identité. Cela garantit que seules les personnes autorisées peuvent accéder ou modifier les données sensibles (matériel, planning des streamers, monitoring).
- **Role-based Access Control (RBAC)** : Un système de permissions a été mis en place pour distinguer les **administrateurs** (qui peuvent gérer les données du système) des **streamers** (qui ont un accès limité à leur propre planning et thématique).

d. Chiffrement des Mots de Passe

- Les mots de passe des utilisateurs sont **hachés** avant d'être stockés dans la base de données en utilisant une fonction de hachage sécurisée comme **bcrypt**, ce qui empêche qu'ils soient récupérés en clair en cas de fuite de données.

e. Sécurisation des Communications

- **HTTPS** : Les échanges de données entre le serveur et le client sont sécurisés via un certificat SSL pour assurer que toutes les communications soient cryptées et sécurisées, particulièrement pour les actions sensibles comme l'authentification et la gestion des données.

f. Mise en place d'un Pare-feu

- Utilisation d'un **pare-feu** sur le serveur de production pour limiter l'accès aux seules adresses IP autorisées et filtrer les tentatives d'accès non autorisées.

En résumé, les choix technologiques et les mécanismes de sécurité mis en place pour cette application garantissent une gestion fluide des données, tout en assurant une protection maximale des utilisateurs et des informations critiques.

Veille Technologique sur les Vulnérabilités de Sécurité

1. Introduction à la Veille Technologique

La veille technologique consiste à surveiller et analyser les évolutions technologiques dans un domaine spécifique pour rester à jour sur les nouvelles tendances, découvertes, et menaces. En matière de sécurité, cette veille est cruciale pour protéger les systèmes et les données contre des vulnérabilités émergentes. Dans le cadre du développement de notre application pour le Z-Event, j'ai effectué une veille régulière sur les **vulnérabilités de sécurité**, afin de garantir que l'application reste sécurisée face aux menaces en constante évolution.

2. Objectif de la Veille

L'objectif principal de cette veille était d'identifier les **nouvelles failles de sécurité** dans les technologies utilisées dans notre projet (Node.js, React.js, MySQL, JWT) et de comprendre les moyens de les atténuer. Cela permettrait de protéger les données sensibles telles que les informations sur les streamers, les plannings, le stock de matériel, ainsi que l'intégrité des flux en direct.

3. Méthodologie de la Veille

a. Sources d'Information

La veille technologique a été effectuée en s'appuyant sur plusieurs sources fiables dans le domaine de la cybersécurité :

- **CVE (Common Vulnerabilities and Exposures)** : La base de données CVE est une source essentielle pour obtenir des informations sur les vulnérabilités connues dans les bibliothèques et logiciels utilisés.
- **OWASP (Open Web Application Security Project)** : OWASP publie régulièrement des rapports et des documents sur les vulnérabilités les plus courantes et les bonnes pratiques pour les éviter, notamment avec la **OWASP Top 10** qui liste les risques les plus critiques pour les applications web.
- **NVD (National Vulnerability Database)** : La NVD est une autre base de données importante qui fournit des détails techniques sur les failles et leurs correctifs.
- **Blogs de Sécurité** : Les blogs comme **KrebsOnSecurity** ou les publications de **SANS Institute** fournissent des rapports détaillés sur les nouvelles menaces et les incidents de sécurité récents.
- **Groupes de discussion et forums** : Suivi de discussions sur **Stack Overflow**, **GitHub**, et des forums spécialisés dans la sécurité informatique pour être informé des vulnérabilités découvertes par la communauté.

b. Technologies Ciblées

L'effort de veille a principalement porté sur les technologies suivantes :

- **Node.js** et ses packages NPM : En raison de la popularité de Node.js, de nombreuses vulnérabilités sont régulièrement découvertes dans les modules NPM, ce qui peut affecter la sécurité globale de l'application.
- **React.js** : Le framework front-end peut être vulnérable aux attaques de type **Cross-Site Scripting (XSS)** ou autres exploits front-end.
- **MySQL** : Les vulnérabilités potentielles autour des injections SQL ou des mauvaises configurations peuvent compromettre la sécurité de la base de données.
- **JSON Web Tokens (JWT)** : Les failles dans la gestion des tokens peuvent permettre à des attaquants d'usurper des identités ou d'accéder à des données sensibles.

4. Résultats de la Veille Technologique

a. Vulnérabilités Identifiées

1. **Node.js et NPM** :
    a. **Prototype Pollution dans certains packages NPM** : Une vulnérabilité a été découverte dans plusieurs packages NPM où un attaquant pouvait exploiter une faille de **prototype pollution**, affectant les objets JavaScript utilisés dans l'application. Cette faille permettait à un attaquant de modifier des propriétés d'objets partagés au sein de l'application, pouvant entraîner des comportements imprévus et dangereux.
    b. **Solution** : S'assurer que tous les packages NPM sont régulièrement mis à jour. Utilisation de la commande `npm audit` pour détecter les vulnérabilités dans les dépendances du projet, et implémentation des correctifs recommandés.
2. **React.js** :
    a. **Cross-Site Scripting (XSS)** : Le rendu dynamique dans React peut être vulnérable à des attaques XSS si les entrées utilisateur ne sont pas correctement échappées. Par exemple, l'injection de scripts malveillants via des formulaires ou des champs texte pourrait compromettre la sécurité des utilisateurs.
    b. **Solution** : Utilisation de la fonction `dangerouslySetInnerHTML` a été strictement évitée dans le code, et toute donnée utilisateur a été systématiquement échappée. De plus, des outils de sécurité comme **Helmet.js** ont été intégrés pour renforcer la sécurité des en-têtes HTTP.
3. **MySQL** :
    a. **Injection SQL** : Même avec des ORM (Object Relational Mapping), certaines vulnérabilités d'injection SQL peuvent persister si les requêtes ne sont pas correctement paramétrées.
    b. **Solution** : Utilisation systématique de **requêtes paramétrées** via l'ORM, ou des fonctions de protection contre les injections SQL. Les requêtes brutes ont été minimisées et encadrées par des vérifications strictes des entrées utilisateurs.
4. **JWT (JSON Web Tokens)** :
    a. **Vulnérabilité aux attaques de relecture (Replay Attack)** : Un attaquant peut réutiliser un token JWT non expiré pour accéder à des ressources même après la déconnexion de l'utilisateur. Cela représente un risque si les tokens ne sont pas correctement invalidés.
    b. **Solution** : Implémentation d'une courte durée de vie pour les tokens JWT avec des **tokens de rafraîchissement**. De plus, un mécanisme de

blacklist pour les tokens révoqués a été mis en place, évitant ainsi que les anciens tokens soient réutilisés.

b. Mesures Préventives

1. **Mise à jour régulière des dépendances** : Les dépendances du projet (Node.js, packages NPM, modules de sécurité) ont été régulièrement mises à jour pour intégrer les derniers correctifs de sécurité.
2. **Audit de sécurité automatisé** : Utilisation d'outils comme **npm audit** et **Snyk** pour effectuer des audits réguliers des vulnérabilités dans les packages tiers.
3. **Amélioration des pratiques de gestion des sessions** : Les sessions d'utilisateur et les tokens JWT ont été configurés avec une durée de vie courte et des rafraîchissements sécurisés pour éviter les abus.
4. **Politiques de sécurité des en-têtes HTTP** : Des en-têtes de sécurité comme **Content Security Policy (CSP)** et **Strict-Transport-Security (HSTS)** ont été implémentés avec **Helmet.js** pour éviter les attaques courantes comme XSS ou le clickjacking.

5. Impact de la Veille sur le Projet

Grâce à cette veille technologique, plusieurs améliorations ont été apportées à notre application :

- **Renforcement des mécanismes d'authentification** : Grâce à l'identification des failles JWT, nous avons optimisé la gestion des tokens pour éviter les abus.
- **Amélioration du front-end sécurisé** : L'application React a été davantage sécurisée contre les attaques XSS en évitant l'injection directe de données non filtrées dans le DOM.
- **Correction des vulnérabilités identifiées** : Les paquets Node.js affectés par des vulnérabilités critiques ont été mis à jour, réduisant ainsi les risques pour l'intégrité du système.

6. Conclusion

La veille technologique sur les vulnérabilités de sécurité est une pratique indispensable pour tout projet de développement. Elle nous a permis de corriger des failles potentielles et de mettre en place des stratégies proactives pour sécuriser notre application. Dans le contexte du Z-Event, où la gestion des données des streamers, du

matériel et des performances de streaming est essentielle, cette veille nous a aidés à protéger l'ensemble du système contre des menaces nouvelles et évolutives

Situation de Travail Nécessitant une Recherche

Lors du développement de l'application pour le Z-Event, une situation complexe s'est présentée concernant la **gestion des tokens JWT (JSON Web Tokens)**. Nous avons constaté que même après qu'un utilisateur se soit déconnecté, son token restait valide et pouvait être utilisé pour accéder à des données sensibles. Cela représentait une vulnérabilité de **replay attack** (attaque par relecture), où un attaquant pourrait réutiliser un ancien token JWT non expiré pour accéder aux ressources.

Pour résoudre ce problème, nous avons effectué une recherche en ligne et avons trouvé un article utile sur un site anglophone expliquant comment **révoquer ou invalider un JWT après déconnexion**.

Source du Site Anglophone

Nous avons trouvé des informations sur le site **Auth0**, un fournisseur de services d'authentification bien connu. L'article qui nous a aidé se trouvait à cette URL : Auth0 - How to securely implement JWT tokens.

Extrait du Site en Anglais

Voici l'extrait spécifique qui nous a été utile :

"One common way to invalidate JWTs after logout is by implementing a blacklist. This involves storing a list of tokens that are no longer valid in a persistent store, such as a database or an in-memory store like Redis. When a token is presented for validation, it is checked against this blacklist to determine whether it has been revoked."

Traduction en Français

"Une méthode courante pour invalider les JWT après une déconnexion est de mettre en place une liste noire. Cela implique de stocker une liste de tokens qui ne sont plus valides dans un stockage persistant, comme une base de données ou un stockage en

mémoire tel que Redis. Lorsqu'un token est présenté pour validation, il est comparé à cette liste noire pour déterminer s'il a été révoqué."

Mise en Pratique

Grâce à cette information, nous avons implémenté une **liste noire (blacklist)** dans notre base de données pour stocker les tokens révoqués après la déconnexion des utilisateurs. Cette solution nous a permis de renforcer la sécurité de notre application en s'assurant qu'aucun token expiré ou révoqué ne puisse être utilisé pour accéder à des données sensibles.

Le processus de vérification des tokens est désormais plus sûr, car chaque token est contrôlé par rapport à cette liste noire avant d'autoriser l'accès aux ressources protégées.