Projet : Backend

Jilen SEBAMALAINATHAN

Nathan REMOND

Contexte du projet :

Ce projet a pour but de concevoir, développer et documenter un back-end complet sur la thématique Steam/ Jeux vidéo. Nous avons choisi cette thématique car nous sommes tous deux passionnés de jeux vidéo et sommes familier avec la plateforme Steam.

Objectifs :

Les objectifs de ce projet sont :

- Avoir une architecture contenant les dossiers Routes, Controllers, Models, Middlewares et Config. Elle doit aussi comporter un cors configuré, un Rate Limiter et un JWT Auth complet (Inscription, Login, Refresh token, Gestion de rôles.
- Utiliser 2 bases de données (PostgreSQL et MongoDB).
- Avoir de la sécurité comme une gestion stricte des tokens JWT, un hashage des mots de passes avec bcrypt, une validation des données, un rate limiting, un CORS, une vérification des permissions et rôles lors des requêtes, ne pas offrir une pinte et une gestion des erreurs centralisée.
- Avoir des tests d'intégrations complets avec postman et jest/supertest.
- Avoir de la documentation (endpoints avec swagger, codes de réponses et erreurs, ...).

Acteurs/ roles

Nathan : Conception, création des bases, CRUD (user), sécurisation de l'API, les test Postman et supertest

Jilen : Conception, CRUD (game, comment, category et profiles), documentation avec swagger

Scénarios utilisateur

L'utilisateur peut créer un compte, ajouter un commentaire (il faut être connecté), afficher les commentaires de chaque utilisateur (il faut être connecté), afficher les jeux avec leurs catégories (il faut être connecté), afficher le profil d'un autre utilisateur (il faut être connecté).

Cas d'usage clé

Le cas d'usage clé de notre projet est de parcourir un catalogue de jeu, voir les détails du jeu, on peut payer, mettre le jeu dans le profil.

Analyse des sécurités mises en place

Le CORS permet d'éviter qu'un site malveillant n'appelle l'API depuis un domaine non autorisé donc il contrôle quels domaines sont autorisés à effectuer des requêtes vers l'API.

Le rate-limiting limite le nombre de requêtes qu'un utilisateur peut effectuer dans une courte donc ça permet de se protéger des DDOS

Le Token JWT permet d'ajouter des serveurs d'un type donné et de ne pas gérer les sessions côté serveur

Les rôles et permissions permettent d'établir des règles sur qui peut accéder à quoi en fonction de leurs rôles.

Le Hashing permet que les mots de passes soient protégés si la base fuit.

Le middleware de validation vérifie que les données sont semblables et de protéger contre les injections, les payloads malformés et garantir la cohérence des données.

La gestion des erreurs permet d'éviter les crashs serveur et produires des réponses cohérentes.

La politique de refresh Token est de maintenir la session ouverte sans exposer des tokens longue durée dans le front.

Justification globale des choix

CORS : permet de sécuriser quelles origines peuvent consommer l'API.

Rate limiting : essentiel pour un store, protection contre brute-force, spidering, bots d'achat, DDoS.

JWT : idéal pour un Game Store en évolution

Rôle/permission : nécessaire pour admin/user car le projet a plusieurs profils avec des droits différents.

 Hashing fort : norme de sécurité moderne et empêche le vol de comptes même si la base fuit.

 Validation middleware : évite les erreurs utilisateurs et empêche les requêtes incorrectes d'arriver au cœur du système.

Gestion d'erreurs centralisée : facilite le debugging et la communication front/back.

 Politique de refresh token : conserve une session persistante sans exposer un token long.

Technologies

JavaScript

Pourquoi cette technologie ?

- Langage universel : même langage pour le front et le back.
- Apprentissage rapide, standard pour les apps web modernes.

Rôle

- Langage du frontend.
- Langage utilisé côté backend (Node.js).

Avantages / limites

Avantages

- Rapidité de développement.
- Très riche en frameworks (Express).

Limites

- Bugs runtime possibles si absence de validation stricte.

Intégration technique

- Utilisé côté client (JS natif ou framework).
- Utilisé côté serveur via Node.js.

Node.js

Pourquoi cette technologie ?

- Idéal pour API performantes.
- Très accessible si l'équipe maîtrise JavaScript.

Rôle

- Sert de backend principal.
- Gère l'API REST.

Avantages / limites

Avantages

- Large écosystème (Express, Mongoose, pg...).
- Parfait pour microservices si besoin.

Limites

- Nécessite rigueur dans gestion async.

Intégration technique

- Framework probable : Express.js.
- Gère les middlewares : CORS, rate limiting, validation, erreurs.
- Connexions DB :
  - PostgreSQL via pg
  - MongoDB via mongoose

PostgreSQL

Pourquoi cette technologie ?

- Solide base relationnelle ACID.
- Parfaite pour données structurées.

Rôle

- Stockage des données critiques

Avantages / limites

Avantages

- Haute fiabilité et transactions.
- Relations et contraintes fortes.

Limites

- Moins flexible que MongoDB pour données non structurées.

Intégration technique

- Connecté via Node.js avec le client pg.
- Les services écrivent et lisent via requêtes SQL.

4. MongoDB

Pourquoi cette technologie ?

- Base NoSQL très flexible.
- Adaptée aux données évolutives, non structurées ou volumineuses.

Rôle

- Stocke les données

Avantages / limites

Avantages

- Schéma flexible.
- Très bonne scalabilité.

Limites

- Pas de transactions strictes comme PostgreSQL.

Intégration technique

- Utilisé via Mongoose.
- Collections.

Postman

Pourquoi cette technologie ?

- Outil standard pour tester les API.
- Facile pour documenter, créer des collections et automatiser les tests.

Rôle

- Tester les endpoints Node.js.
- Servir de documentation technique pour l'équipe.

Avantages / limites

Avantages

- Interface intuitive.
- Tests automatisables.
- Exportable pour toute l'équipe.

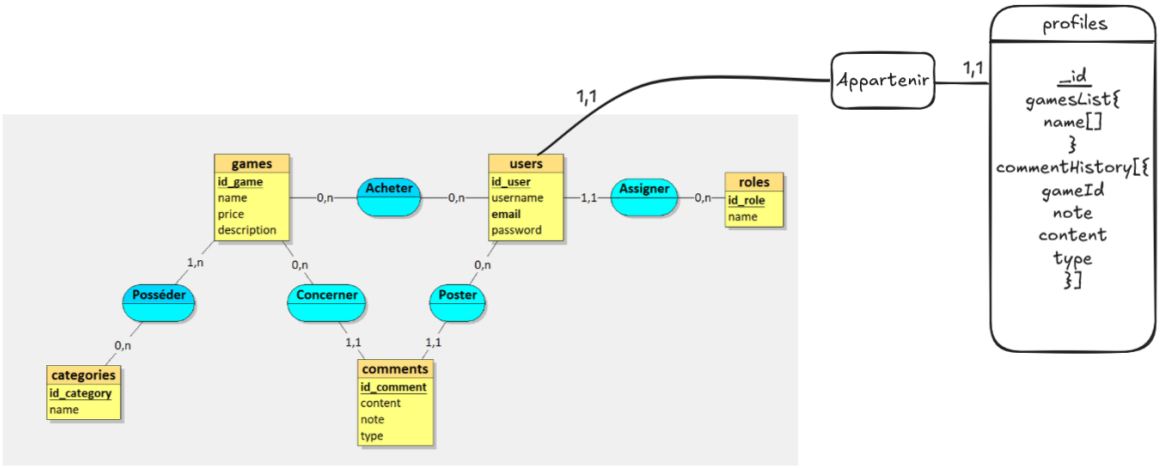
Limites

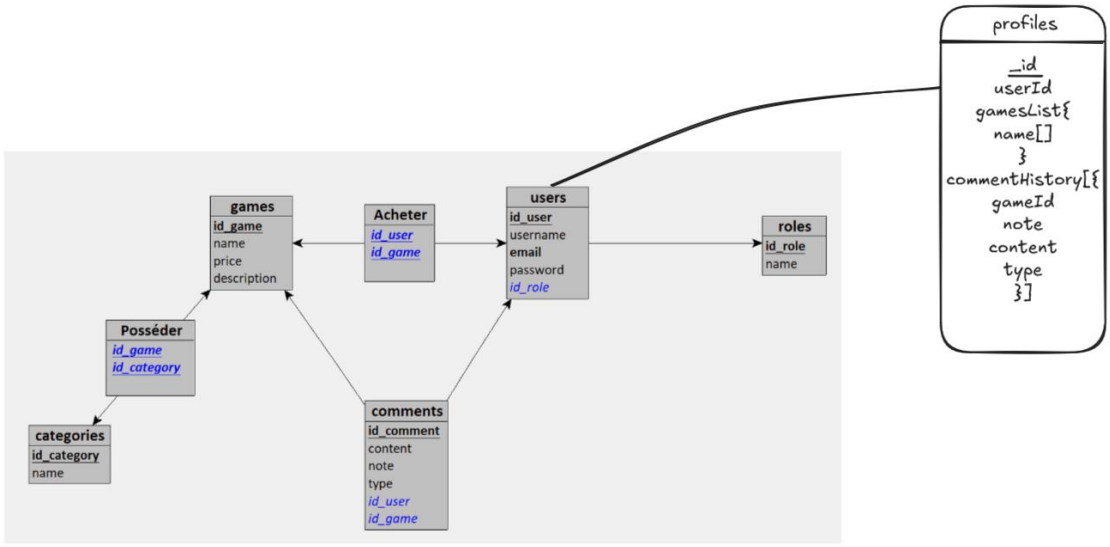
- Nécessite version pro pour certaines fonctions.

Intégration technique

- Collections organisées par modules.
- Tests automatisés dans Postman

MCD et MLD





Comme vous pouvez le constater, les tables users, roles, comments, categories, games sont en PostgreSQL car ces données relationnelles sont structurées, nécessitent des contraintes d'intégrité et ils ont beaucoup de relations entre elles. La table profiles est en MongoDB car ces données sont plus flexibles, ils sont spécifiques à l'utilisateur et ils sont évolutifs dans le temps.

Explication de code

Routes :

```
router.get("/", authenticateToken, CategoryController.listCategories);
```

Dans ce code, on définit une route GET pour lister toutes les catégories d'un jeu qu'on a défini dans le Controller tout en vérifiant si l'utilisateur est connecté avec le middleware authenticateToken.

Controller :

```
class CategoryController {
    static async listCategories(req, res, next) {
        try {
            const categories = await Category.findAll();
            res.status(200).json(categories);
        } catch (err) {
            next(err);
        }
    }
```

Dans ce code, nous définissons une class CategoryController et à l'intérieur, nous mettons une fonction asynchrone qui permet de lister toutes les catégories.

Model PostgreSQL :

```
class Category{

    static async findAll(){
        const result = await pool.query("SELECT * FROM categories");
        return result.rows;
    }
```

Dans ce code, nous définissons une class Category et à l'intérieur, nous mettons une fonction asynchrone qui permet avec une requête SQL, de lister toutes les catégories.

Model MondoDB :

```
const profileSchema = new mongoose.Schema({
  userId: {
    type: Number,
    required: true,
    unique: true,
  },
```

Dans ce code, nous définissons une contante qui prend un nouveau Schéma Mongoose pour la collection Profiles et il a comme parametres userId avec son type, s'il est requis et s'il est unique.

Middleware JWT :

```
export function authenticateToken(req, res, next) {
    const authHeader = req.headers["authorization"];
    const token = authHeader && authHeader.split(" ")[1];

    if (!token) {
        return res.status(401).json({ message: "Accès refusé : aucun token fourni" });
    }

    jwt.verify(token, JWT_SECRET, (e, user) => {
        if (e) {
            return res.status(403).json({ message: "Token invalide ou expiré" });
        }

        req.user = user;
        next();
    });
}
```

Dans ce code, nous définissons une fonction qui vérifie si l'utilisateur a un token et un code secret "JWT_SECRET" ainsi que les informations de l'utilisateur pour savoir si l'utilisateur est connecté.

Rate limiter :

```
const limiter = rateLimit({
    windowMs: 60*1000,
    max: 10,
    message: {
        status: 429,
        message: "Trop de requêtes. Réessayez dans une minute."
    },
    standardHeaders: true,
    legacyHeaders: false
});

if (process.env.NODE_ENV !== "test") {
    app.use(limiter);
}
```

Ce code permet de limiter les requêtes d'un utilisateur pour empêcher le DDOS et quand nous sommes dans la base test, il n'y aura pas de limiter.

Tests d'intégration :

```
describe("Tests sécurisés /api/games", () => {

    test("GET /api/games sans token => 401", async () => {
        const res = await request(app).get("/api/games");
        expect(res.status).toBe(401);
    });
```

Ce test permet de vérifier que sans token, la route GET /api/games renvoie un statut 401.

```
 test("GET /api/games avec token user => 200", async () => {
    const res = await request(app)
      .get("/api/games")
      .set("Authorization", `Bearer ${userToken}`);
    expect(res.status).toBe(200);
    expect(res.body.length).toBeGreaterThan(0);
 });
```

Ce test permet de vérifier qu'avec un token , la route GET/ api/games renvoie un statut 200 et qu'elle renvoie des données.

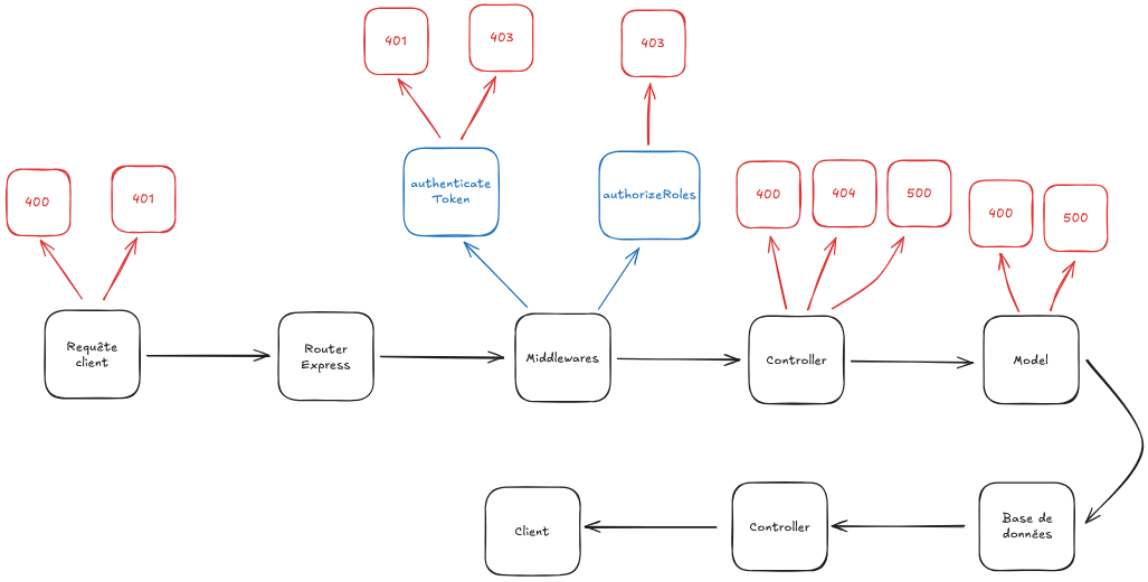
Swagger :

```
import swaggerJsdoc from "swagger-jsdoc";

export const swaggerSpec = swaggerJsdoc({
    definition: {
        openapi: "3.0.0",
        info: {
            title: "TP Express - Game Store API",
            version: "1.0.0"
        }
    },
    apis: ["./src/routes/*.js"]
});
```

Dans ce code, on appelle la fonction sawggerJsdoc avec les méta-informations de ton API.

Schéma des échanges entre API et bases :



Documentation Swagger :

```
/**
 * @openapi
 * /api/games/{id}:
 *   put:
 *     summary: Met à jour un jeu par son id
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *         description: L'id du jeu
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             type: object
 *             properties:
 *               name:
 *                 type: string
 *               price:
 *                 type: number
 *             description: Le nom et le prix du jeu
 *     responses:
 *       200:
 *         description: Jeu mis à jour
 *       404:
 *         description: Jeu non trouvé
 */
router.put("/:id", authenticateToken, authorizeRoles("admin"), GameController.updateGame);
```

Dans cette documentation Swagger, nous définissons sa route et ce qu'elle fait avec ses paramètres et ses réponses correctes et incorrectes.

Vulgarisation des technologies:

Express.js : C’est un framework node.js qui facilite la création de serveurs et d’API

REST : Style d’architecture pour créer des API

POO : C’est une méthode de programmation qui organise le code avec des objets et des classes.

MVC : C’est un type d’architecture qui sépare le dossier model, view, controller

PostgreSQL : c’est une base de données relationnelle qui permet de stocker des données avec des tables et des relations.

MongoDB : C’est une base de données NoSQL qui stockent les données sous forme d’un document JSON

CORS : C’est une sécurité qui contrôle quels sites peuvent accéder à ton API

Rate limiter : C’est une sécurité qui permet de limiter le nombre de requêtes qu’un utilisateur peut faire dans un intervalle de temps

JWT : C’est une méthode pour identifier un utilisateur avec un token signé

Swagger : Cela permet à l'utilisateur de demander exactement les données qu’il a besoin