Rapport de projet : MediaTracker

Alexandre BOISFER

13 janvier 2026

Table des matières

I Introduction et contexte

Ce document présente le rapport de réalisation du projet **MediaTracker**, développé dans le cadre du module SAE "Développement d'application web interactive" (FISA S5). Au-delà de l'exercice académique, ce projet incarne une volonté de concevoir une solution professionnelle et pérenne répondant à une problématique quotidienne de consommation culturelle.

I.1 Cadre du projet et problématique

À l'ère du numérique, notre consommation de divertissement est fragmentée sur de multiples plateformes (Netflix, Steam, Spotify, Kindle). Il devient complexe de conserver un historique centralisé de notre vie culturelle ou de gérer simplement une liste d'envies sans basculer d'une application à l'autre. L'ambition de *MediaTracker* est de devenir ce point de convergence unique : un véritable "compagnon médiatique".

Si la vision à long terme inclut la gestion des livres, des jeux vidéo et de la musique, cette première version (v1) se concentre spécifiquement sur l'univers audiovisuel (films et séries).

I.2 Objectifs pédagogiques et personnels

Ce projet représente un jalon important dans mon parcours d'ingénieur pour plusieurs raisons :

- **Montée en compétences techniques :** L'objectif est de maîtriser des technologies de pointe, notamment l'implémentation complète d'une API GraphQL (découverte totale) et l'utilisation des dernières avancées de Next.js 16.
- **Professionnalisation :** Passer du statut de "projet étudiant" à celui de "produit portfolio" complet, respectant les standards de l'industrie (architecture propre, typage strict, CI/CD).
- **Validation des acquis :** Démontrer ma capacité à mener un projet Full-Stack complexe en autonomie, de la conception de la base de données jusqu'au déploiement.

I.3 Périmètre fonctionnel

Pour cette première itération, les fonctionnalités ont été priorisées pour garantir une expérience utilisateur fluide et aboutie :

- **Gestion de collection :** Recherche via l'API TVDB, ajout à la bibliothèque et suppression d'œuvres.
- **Suivi de progression :** Marquage des épisodes vus et suivi global des séries.
- **Fonctions sociales :** Système de notation et de commentaires pour chaque média.

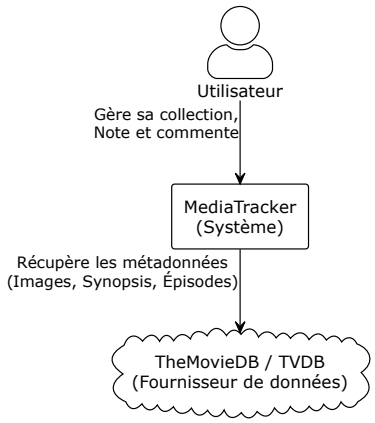

FIGURE 1 – Diagramme de contexte (niveau 0)

I.4 Méthodologie et organisation

La gestion de ce projet en solo a nécessité une rigueur organisationnelle stricte pour tenir les délais impartis. J'ai adopté une approche itérative inspirée des méthodes SCRUM :

- **Planification par sprints :** Le travail a été découpé en blocs fonctionnels distincts (infrastructure, backend core, UI integration), visibles sur le diagramme de Gantt (Figure 2).
- **Processus de qualité :** Chaque fin de cycle inclut une phase de"Self-Review" et de refactoring, garantissant que la dette technique ne s'accumule pas.
- **Convention de code :** L'application stricte des *Conventional Commits* permet une traçabilité complète des évolutions et des correctifs.

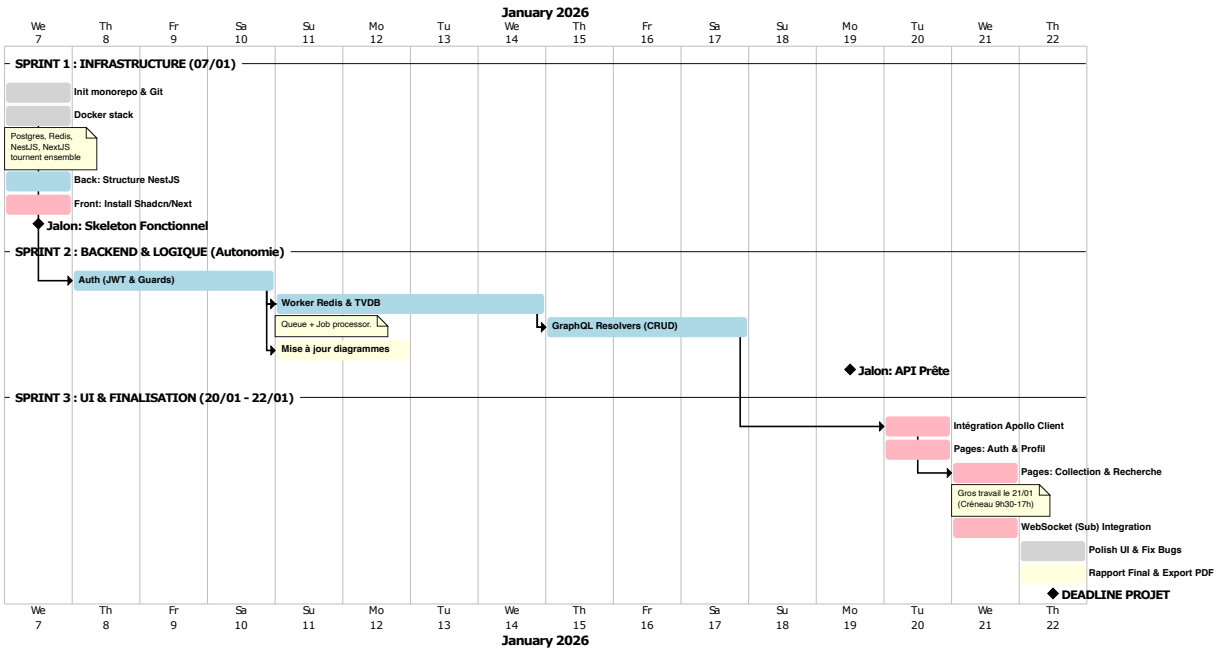

FIGURE 2 – Planning prévisionnel et découpage en sprints

II Analyse et conception

Cette partie détaille la traduction du besoin métier en spécifications techniques et visuelles avant le début du développement.

II.1 Besoins fonctionnels

Liste des fonctionnalités clés attendues par l'utilisateur : authentification sécurisée, gestion CRUD des médias (films, séries, animés), et interactions sociales (système de favoris, notes, statuts de visionnage).

II.2 Modélisation (UML)

Présentation des diagrammes de conception. Analyse de la structure de données choisie pour découpler le catalogue global des données utilisateur.

III Justification des choix technologiques

Le choix de l'architecture est une décision qui impacte non seulement la performance de l'application, mais aussi sa maintenabilité et sa sécurité sur le long terme. Cette section détaille les arbitrages réalisés pour le backend, le frontend et la base de données.

III.1 Architecture backend : le choix de NestJS

Dans le cadre de ce projet, j'ai évalué quatre technologies majeures : NestJS, Python FastAPI, Java Spring Boot et PHP Symfony. L'analyse repose sur une matrice de décision pondérée (Weighted Decision Matrix) basée sur les contraintes du projet : architecture micro-services, API GraphQL, et rigueur de code.

Analyse par critère

- **Structure & rigueur :**
  - **NestJS (5/5) :** Impose une architecture modulaire stricte (inspirée d'Angular) avec injection de dépendances. Idéal pour prévenir la dette technique.
  - **Spring Boot (5/5) :** La référence industrielle, mais au prix d'une verbosité importante.
  - **Symfony (4/5) :** Excellent respect du MVC, mais plus permissif que Nest ou Spring.
  - **FastAPI (3/5) :** N'impose aucune architecture. Risque élevé de code désorganisé sans expert.
- **Sûreté du typage :**
  - **NestJS (5/5) :** TypeScript strict vérifié à la compilation. Élimine une classe entière de bugs.
  - **Spring Boot (5/5) :** Typage statique fort (Java). Robustesse maximale.
  - **Symfony (3/5) :** PHP 8 a beaucoup progressé sur le typage, mais reste un langage permissif à la base. Le typage strict doit être activé volontairement fichier par fichier.
  - **FastAPI (2/5) :** Les "Type Hints" Python sont ignorés à l'exécution.
- **Synergie frontend :**
  - **NestJS (5/5) :** Avantage décisif. Partage des interfaces (DTO) via un monorepo (full TypeScript).
  - **Autres (1/5) :** Rupture technologique imposant une duplication des modèles de données.
- **Performance asynchrone & I/O :**
  - **NestJS (5/5) :** Node.js est conçu autour d'une boucle d'événement non bloquante. Parfait pour gérer des milliers de connexions simultanées (cas typique des Subscriptions GraphQL) avec peu de ressources.
  - **FastAPI (4/5) :** Utilise `async/await` natif de Python (ASGI). Très performant, mais certaines librairies Python tierces sont encore bloquantes, ce qui peut paralyser l'API.
  - **Spring Boot (3/5) :** Modèle classique "un thread par requête". Robuste mais gourmand en mémoire. Le mode réactif (WebFlux) existe mais complexifie énormément le développement.
  - **Symfony (2/5) :** PHP est synchrone par nature. Gérer de l'asynchrone réel (WebSockets) demande des outils tiers complexes (Swoole, ReactPHP) ou des serveurs d'application spécifiques (FrankenPHP).

TABLE 1 – Matrice de décision pondérée pour le choix du backend

Critères de sélection	Poids	NestJS		FastAPI		Spring Boot		Symfony	
	(1–5)	Note	Score	Note	Score	Note	Score	Note	Score
Structure & rigueur	5	5	25	3	15	5	25	4	20
Sûreté du typage	4	5	20	2	8	5	20	3	12
Synergie frontend	5	5	25	1	5	1	5	1	5
Support GraphQL / WebSocket	4	5	20	3	12	4	16	3	12
Performance asynchrone	3	5	15	4	12	3	9	2	6
Efficacité ressources	2	4	8	4	8	2	4	3	6
SCORE TOTAL	**23**	**113**		**60**		**79**		**61**	

Légende : Note (1–5), Score (Note x Poids). NestJS (Node) vs FastAPI (Python) vs Spring Boot (Java) vs Symfony (PHP).

Conclusion backend : Bien que Spring Boot soit robuste, **NestJS** s'impose (score : 113) grâce à sa synergie totale avec le frontend et son architecture événementielle native.

III.2 Architecture frontend : le choix de Next.js

Contrairement aux recommandations du cours orientant vers Vue.js, j'ai opté pour **Next.js (React)**.

TABLE 2 – Comparatif frontend simplifié

Critère	Next.js (Retenu)	Vue.js (Cours)
Rendu	Server Components (RSC)	SPA / Options API
Typage	TypeScript Natif (Strict)	Prop Types (Moins strict)
Écosystème UI	Shadcn/UI (Moderne)	Vuetify / Bootstrap

Ce choix permet d'utiliser les *React Server Components* pour optimiser le chargement initial et le SEO, tout en tirant profit de l'écosystème React (Shadcn/UI, Aceternity UI) pour l'interface.

III.3 Base de données et ORM

J'ai choisi le couple **PostgreSQL + Prisma** plutôt que MSSQL.

- **PostgreSQL :** Standard open-source, images Docker très légères (contrairement à MSSQL Server qui est lourd en local).
- **Prisma :** Remplace les requêtes SQL manuelles par un client typé auto-généré. Il garantit que si le schéma de base de données change, le code ne compile plus (Type Safety), évitant les erreurs au runtime.

IV Architecture Technique

Description de l'infrastructure logicielle mise en place pour soutenir le développement et l'exécution de l'application.

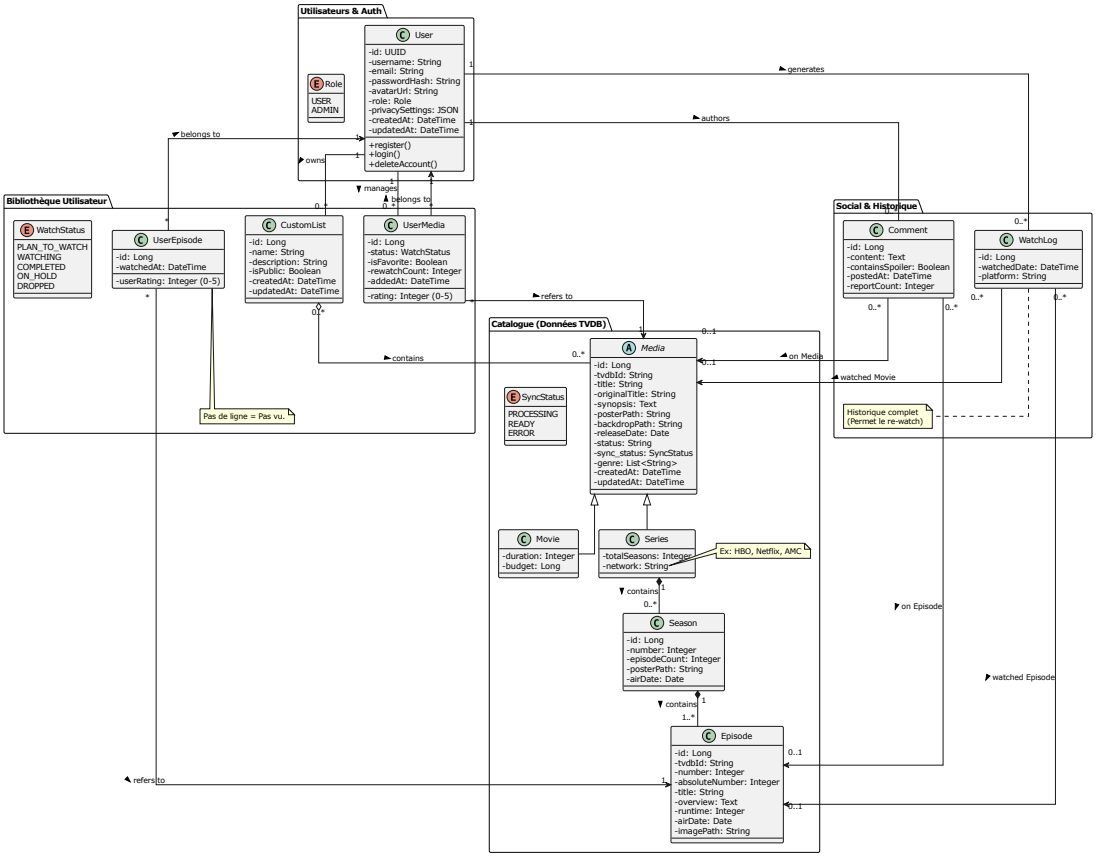

FIGURE 3 – Diagramme de classes

IV.1 Structure monorepo

Présentation de l'organisation du code en un dépôt unique contenant backend, frontend et documentation. Avantages liés au partage de configuration (ESLint, TSConfig) et à l'atomicité des commits.

IV.2 Protocole de communication (GraphQL)

Justification de l'utilisation de GraphQL face à REST. Explication des avantages : réduction de l'*over-fetching*, typage fort via l'introspection, et génération automatique du code client.

V Sécurité et Conformité RGPD

La sécurité des données utilisateurs est un enjeu central du projet MediaTracker. L'architecture a été pensée pour minimiser les vecteurs d'attaque tout en garantissant la souveraineté des données.

V.1 Stratégie d'authentification (NextAuth & JWT)

L'authentification repose sur une architecture hybride. Le frontend (Next.js) utilise la librairie **NextAuth.js** (v5) qui agit comme une couche d'abstraction sécurisée.

- **Credential provider :** NextAuth capture les identifiants, les valide via un schéma **Zod** (pour garantir le format email/password avant même l'envoi), et interroge l'API GraphQL via une mutation dédiée.
- **Gestion de session :** Le token JWT (JSON Web Token) généré par le backend NestJS est encapsulé dans un cookie chiffré et géré automatiquement par NextAuth (`HttpOnly`, `Secure`).
- **Payload :** Le JWT contient l'ID utilisateur et son rôle (`USER` ou `ADMIN`), permettant au backend de vérifier les droits sans interroger la base de données à chaque requête (stateless).

V.2 Hachage des mots de passe

Contrairement aux solutions vieillissantes comme MD5 ou SHA-256, et même préférablement à bcrypt, j'ai opté pour **argon2** (via la librairie `argon2`).

- **Justification :** Argon2 est l'algorithme lauréat de la "Password Hashing Competition".
- Il est conçu pour résister aux attaques par force brute sur GPU (Memory-Hard Function), offrant un niveau de sécurité maximal pour les comptes utilisateurs en 2026.

V.3 Gestion des données personnelles (RGPD)

La conformité au Règlement Général sur la Protection des Données est assurée par deux mécanismes clés :

1. **Droit à la portabilité :** L'utilisateur peut demander l'export complet de ses données via le cas d'utilisation.
2. Le système génère un fichier structuré au format **JSON**, lisible par une machine et facilement réutilisable.
3. **Droit à l'oubli (suppression) :** Actuellement, la suppression d'un compte entraîne une suppression physique définitive (*Hard Delete*) de l'utilisateur et de toutes ses données liées (listes, notes, historique) grâce aux contraintes d'intégrité référentielle (`ON DELETE CASCADE`) de la base de données. *Perspective d'évolution :* Une approche par "Soft Delete" (marquage inactif pendant 30 jours avant purge) est envisagée pour permettre la restauration en cas d'erreur, mais n'a pas été retenue dans cette version v1 pour privilégier la minimisation des données stockées.

VI Implémentation et qualité

Détails sur la phase de développement, les défis techniques rencontrés et les solutions d'ingénierie logicielle appliquées.

VI.1 Stratégie de qualité code (static analysis)

Compte tenu des contraintes temporelles strictes du projet, la stratégie de qualité s'est concentrée sur l'analyse statique et la prévention des bugs en amont, plutôt que sur une couverture de tests unitaires a posteriori.

- **Typage strict (TypeScript) :** L'utilisation de TypeScript en mode `strict` sur toute la stack garantit la cohérence des structures de données.
- Le partage des types DTO entre le back et le front empêche les erreurs d'incompatibilité API.
- **Linter unifié (Biome) :** L'outil Biome remplace ESLint et Prettier pour imposer des standards de code rigoureux.
- **Hooks de pré-commit :** L'outil `lint-staged` empêche physiquement tout commit ne respectant pas les règles de formatage ou contenant des erreurs de syntaxe, agissant comme une première barrière de qualité (quality gate).

VI.2 Gestion des erreurs standardisée

L'API backend implémente un mappage précis des erreurs de base de données vers des exceptions HTTP compréhensibles. Par exemple, le code d'erreur Prisma `P2002` (violation d'unicité) est intercepté dans les services et transformé en une `ConflictException` (HTTP 409) avec un message clair ("Nom d'utilisateur ou adresse mail déjà utilisé"). Cela permet au frontend d'afficher des feedbacks précis à l'utilisateur sans exposer la structure interne de la base de données.

VI.3 Développement backend (NestJS)

Focus sur l'utilisation des décorateurs et des modules. Explication de la sécurisation de l'API via les Guards JWT et la stratégie "private by default" (décorateur `@Public`).

Contrôleur NestJS : MediaController

```typescript
@Controller('media')
export class MediaController {
    @Get(':id')
    findOne(@Param('id') id: string) {
        return this.mediaService.findById(id);
    }
}
```

VI.4 Développement frontend (Next.js)

Détails sur l'intégration des composants UI (Shadcn/UI), la gestion du cache avec Apollo Client et l'expérience utilisateur (UX).

VI.5 Qualité et outillage (innovation)

Présentation de la démarche d'innovation technique (BC 1.5). Utilisation de **Bun** comme runtime haute performance et de **Biome** pour un linting/formatting ultra-rapide, garantissant une "expérience développeur"

moderne.

VI.6 Déploiement et administration

Description de la conteneurisation via Docker et Docker Compose. Gestion des variables d'environnement et administration des données via prisma studio.

VII Bilan et perspectives

Rétrospective critique sur le projet, analyse des réussites et des axes d'amélioration.

VII.1 Retour sur la stack technique

Analyse du ratio coût/bénéfice de la stack choisie. Évaluation de la productivité gagnée grâce à la synergie TypeScript Backend/Frontend versus la complexité initiale de configuration.

VII.2 Auto-évaluation des compétences

Tableau de synthèse validant l'acquisition des compétences du référentiel SAE (Niveaux Notion, Application, Maîtrise, Expertise).

VIII Conclusion

Synthèse finale du projet MediaTracker. Rappel de la réponse à la problématique initiale d'unification des bibliothèques multimédias et ouverture sur les évolutions futures possibles (Mobile, IA).

Table des figures

Liste des tableaux

Références

[1] NestJS. Nestjs - a progressive node.js framework. https://nestjs.fr/, 2025. Consulté le 11 janvier 2026.

[2] Équipe pédagogique FISA. *Support de cours : Développement web d'application web interactive*. INSA / UPHF, 2025.