Rapport Technique — Beatothèque

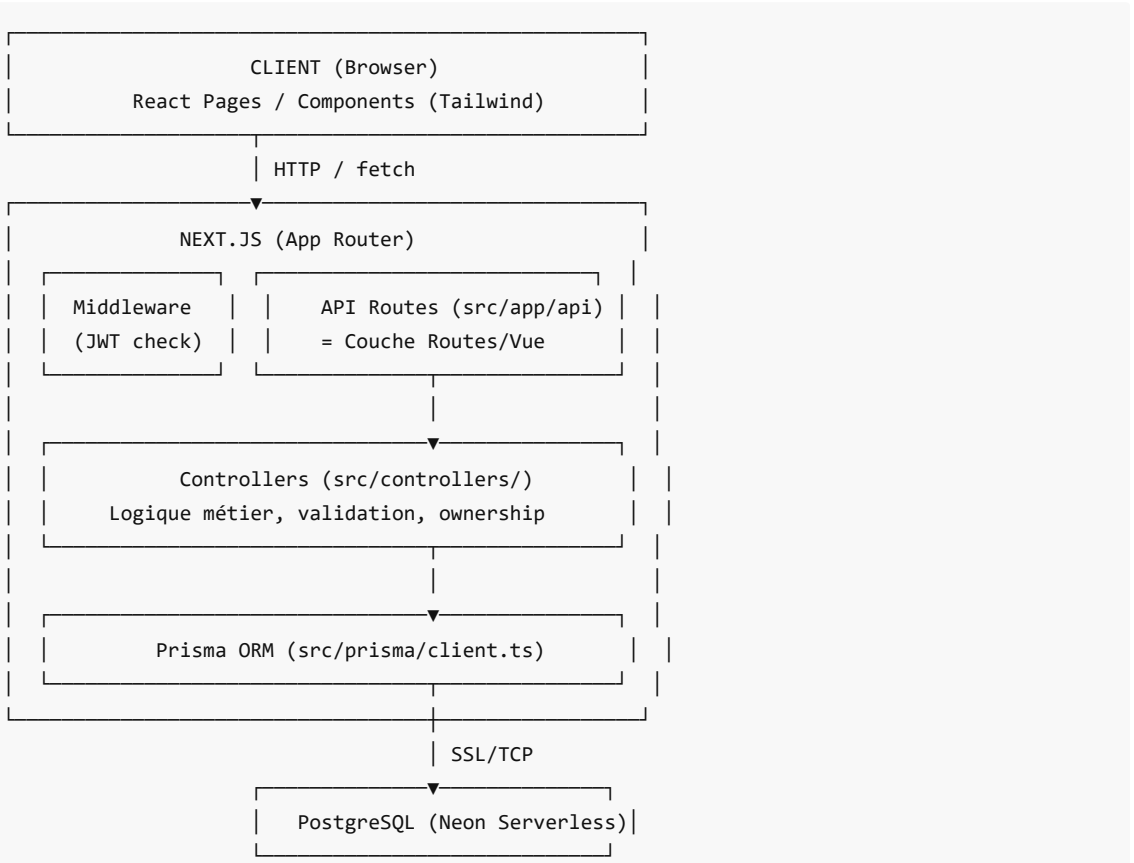
Services Web — Hiver 2026 | Lab 2 + Lab 3

---

1. Architecture

1.1 Vue d'ensemble

Beatothèque est une application web full-stack développée selon le parcours B (Next.js Full-Stack, App Router). L'architecture suit le modèle MVC adapté à Next.js :



1.2 Choix techniques justifiés

Next.js 14 App Router a été choisi pour permettre le rendu côté serveur (SSR), le routing natif et la colocation des API routes avec les pages, réduisant la complexité d'infrastructure.

Prisma ORM offre une sécurité de type bout-en-bout entre le schéma de base de données et le code TypeScript, éliminant les erreurs de requêtes SQL à la compilation.

Neon (Serverless PostgreSQL) est compatible avec les environnements serverless (Edge functions, Vercel) car il gère le pool de connexions au niveau de la plateforme, évitant l'épuisement de connexions au redémarrage des fonctions.

jose a été préféré à `jsonwebtoken` car cette bibliothèque est compatible avec l'Edge Runtime de Next.js (le middleware s'exécute dans un environnement non-Node.js).

Tailwind CSS permet un développement UI rapide et cohérent sans fichiers CSS externes, avec un bundle final optimisé grâce au tree-shaking.

1.3 Structure MVC

Couche	Dossier	Rôle
Model	`backend/models/`, `prisma/schema.prisma`	Définitions TypeScript + schéma DB
View	`src/app/` (pages), `frontend/components/`	Interface React + composants
Controller	`backend/controllers/`	Logique métier pure
Routes	`src/app/api/`, `backend/routes/index.ts`	Endpoints REST + référence

---

2. Authentification

2.1 Justification du choix : JWT Personnalisé (Option 3)

Le JWT personnalisé a été choisi plutôt que NextAuth pour trois raisons :

1. **Contrôle total** sur la structure du payload et la durée de vie des tokens.
2. **Légèreté** : pas de dépendance externe volumineuse pour un système simple.
3. **Apprentissage** : implémenter le cycle complet (génération, vérification, révocation par expiration) renforce la compréhension des mécanismes d'auth.

2.2 Flux d'authentification

```
Inscription/Connexion
      │
      ▼
POST /api/auth/register ou /login
      │
      ├─ Validation Zod des données
      ├─ Hash bcrypt du mot de passe (coût 12)
      ├─ Génération JWT (jose, HS256, 7 jours)
      │
      ▼
Cookie httpOnly "beatotheque_token"
  ├─ httpOnly: true   → inaccessible à JavaScript (protection XSS)
  ├─ secure: true     → HTTPS uniquement en production
  └─ sameSite: lax    → protection CSRF de base

Requête protégée
      │
      ├─ Middleware Next.js vérifie le cookie (pages /dashboard)
      ├─ requireAuth() dans les controllers (routes API)
      │
      ▼
  Accès accordé ou 401/302
```

2.3 Règles d'accès (Ownership)

Chaque ressource modifiable vérifie que `userId` de la ressource correspond à `sub` du JWT. Les controllers `beat.controller.ts` et `license.controller.ts` implémentent ces vérifications avant toute mutation.

2.4 Sécurité du mot de passe

bcryptjs avec un coût de 12 est utilisé pour le hachage. Ce coût représente ~250ms par hachage sur un serveur moderne, rendant les attaques par force brute computationnellement coûteuses. Les messages d'erreur de connexion sont volontairement génériques ("Email ou mot de passe incorrect") pour éviter l'énumération d'emails.

---

3. Défis rencontrés et solutions

Défi 1 : Compatibilité Edge Runtime / jsonwebtoken

Problème : La bibliothèque `jsonwebtoken` utilise des APIs Node.js (`crypto`) non disponibles dans l'Edge Runtime de Next.js, où s'exécute le middleware.

Solution : Remplacement par `jose` , qui utilise l'API Web Crypto standard, compatible aussi bien avec Node.js que l'Edge Runtime.

Défi 2 : Connexions PostgreSQL et hot-reload

Problème : En développement, Next.js recharge les modules à chaque modification. Sans précaution, chaque rechargement crée une nouvelle instance PrismaClient, épuisant rapidement le pool de connexions de Neon (max 10 connexions par défaut).

Solution : Pattern Singleton sur `globalThis` dans `src/prisma/client.ts` , qui réutilise l'instance existante entre les rechargements.

Défi 3 : Decimal Prisma → JSON

Problème : Prisma retourne les champs `Decimal` comme des objets JavaScript spéciaux qui ne se sérialisent pas correctement en JSON (`{}`).

Solution : Utilisation de `Number(beat.price)` dans les réponses côté client, et déclaration des types en `number` dans les interfaces TypeScript front-end.

Défi 4 : Protection des routes côté middleware

Problème : Le middleware s'exécute avant le rendu des pages et doit vérifier le JWT sans dépendances Node.js.

Solution : Extraction de la logique de vérification JWT dans `backend/lib/jwt.ts` avec `jose` , appelé directement dans `middleware.ts` . En cas d'échec, redirection vers `/login?from=<pathname>` pour reprendre la navigation après connexion.

Défi 5 : Système de fichiers Vercel non persistant

Problème : L'upload de fichiers audio utilisait `fs.writeFile` pour écrire dans `public/uploads/` . Sur Vercel, le système de fichiers est en lecture seule en production — les fichiers écrits sont perdus dès que la fonction serverless se termine.

Solution : Migration vers **Vercel Blob** (`@vercel/blob`). L'API `put()` envoie le fichier directement vers un stockage objet persistant et retourne une URL publique permanente, sans modifier l'interface utilisateur.

---

4. Améliorations futures

4.1 Fonctionnalités

- **Système de paiement** : intégration Stripe pour l'achat de licences directement dans l'application.
- **Recherche full-text** : recherche par titre, artiste ou tags avec PostgreSQL `tsvector` ou Elasticsearch.
- **Lecteur audio intégré** : prévisualisation des beats directement dans le catalogue sans téléchargement.
- **Profil producteur public** : page publique par utilisateur regroupant tous ses beats.

4.2 Sécurité et performance

- **Refresh token** : implémenter un mécanisme de rotation de tokens pour les sessions longues sans re-authentification.
- **Rate limiting** : limiter les tentatives de connexion par IP (ex: `upstash/ratelimit`) pour prévenir les attaques par force brute.
- **Révocation de tokens** : maintenir une blocklist Redis pour invalider les tokens avant expiration (utile pour la déconnexion immédiate).
- **Tests automatisés** : suite de tests d'intégration avec Jest et `supertest` pour les endpoints API critiques.

4.3 Infrastructure

- **CI/CD** : pipeline GitHub Actions pour linting, build et tests à chaque pull request.
- **Monitoring** : intégration Sentry pour la capture d'erreurs en production.
- **Mise à jour Next.js** : migration vers Next.js 15 pour corriger la vulnérabilité de sécurité signalée (décembre 2025).

Fin du rapport technique — Beatothèque | Services Web Hiver 2026