R5-8 - Qualité de développement

DOCUMENTATION TECHNIQUE – Projet de Gestion de Bibliothèque

Projet : Système de Gestion de Bibliothèque Universitaire

Groupe : Naharro Guerby, Bonnard Nathan, Le Bastard Théo

Date de rendu : 13/02/2026

SOMMAIRE

I. Introduction

Ce projet s'inscrit dans le cadre de la modernisation des infrastructures numériques de l'université. L'objectif est de réaliser une **application web complète (Full Stack)** permettant la gestion automatisée d'une bibliothèque universitaire.

Le projet servira de support pour évaluer :

- La maîtrise de l'architecture client-serveur (React/Django).
- La qualité du code et la gestion de base de données.
- La capacité à déployer une solution conteneurisée (Docker).

Équipe de développement :

- Naharro Guerby
- Bonnard Nathan
- Le Bastard Théo

II. ARCHITECTURE GÉNÉRALE

Le projet repose sur une architecture **n-tiers** (Client-Serveur) conteneurisée, garantissant une séparation claire des responsabilités et une facilité de déploiement.

A. Schéma d'Architecture

L'application est composée de trois services distincts orchestrés par Docker Compose :

1. **Frontend (Client)** : Application React.js (SPA) servie via Vite. Elle communique avec le Backend via des appels HTTP (API REST).
2. **Backend (Serveur)** : Application Django exposant une API REST. Elle traite la logique métier et communique avec la base de données.
3. **Base de Données** : Serveur PostgreSQL stockant les données de manière persistante.

B. Flux de Données

- **Requête** : Le Client (Navigateur) envoie une requête HTTP (ex: GET /api/books/) au Serveur.
- **Traitement** : Django reçoit la requête, interroge PostgreSQL via l'ORM, sérialise les données en JSON et renvoie la réponse.
- **Affichage** : React reçoit le JSON et met à jour l'interface utilisateur (DOM) sans recharger la page.

III. BACKEND (API & LOGIQUE MÉTIER)

A. Technologies

- **Langage :** Python 3.11
- **Framework :** Django 5.0
- **API Toolkit :** Django REST Framework (DRF)
- **Authentification :** Token Authentication (DRF)

B. Structure du Code (/backend/bibliotheque/)

- **models.py** : Définition des objets (ORM) mappés en base de données.
- **serializers.py** : Transformation des objets Python (Models) en JSON et validation des données entrantes.
- **views.py** : Logique de contrôle (ViewSets). Gère les requêtes GET, POST, PUT, DELETE.
- **urls.py** : Routage des URL API (ex: /api/books/ -> BookViewSet).

C. Points de Terminaison (Endpoints API)

Méthode	Endpoint	Description	Accès
POST	/api/login/	Authentification et obtention du Token	Public
GET	/api/books/	Liste des livres (avec filtres)	Public
POST	/api/books/	Ajouter un livre	Admin
POST	/api/books/{id}/borrow/	Emprunter un livre	Authentifié

GET	/api/my-loans/	Liste des emprunts de l'utilisateur	Authentifié
GET	/api/dashboard/	Statistiques globales	Admin

IV. FRONTEND (INTERFACE UTILISATEUR)

A. Technologies

- **Framework :** React 18
- **Build Tool :** Vite (pour des performances de développement accrues)
- **Langage :** TypeScript (TSX) pour la robustesse du typage.
- **UI Library :** Material UI (MUI v6) pour les composants graphiques.
- **HTTP Client :** Axios.

B. Organisation des Dossiers (/frontend/src/)

1. **components/** : Éléments réutilisables (Navbar, BookCard, Popups).
2. **pages/** : Vues principales (HomePage, Dashboard, MyLoans).
3. **services/** : Fichier api.ts centralisant tous les appels Axios vers le Backend.
4. **context/** : Gestion de l'état global (AuthContext pour l'utilisateur connecté).

V. BASE DE DONNÉES

A. Système de Gestion de Base de Données (SGBD)

- **Serveur :** PostgreSQL 16
- **Persistance :** Volume Docker postgres_data pour conserver les données après arrêt des conteneurs.

B. Modèle Conceptuel de Données (MCD)

L'application utilise 4 tables principales :

1. **Users (auth_user étendu)**
    - id (PK), username, password, email
    - role (FK vers Role) : Définit les permissions (Admin, Prof, Élève).
2. **Books (api_book)**
    - id (PK)
    - title, author, isbn, editor, page_count (Varchar/Int)
    - stock (Int) : Nombre d'exemplaires disponibles.
    - category (FK vers Category).
3. **Loans (api_loan)**
    - id (PK)
    - user (FK vers User)
    - book (FK vers Book)
    - loan_date (Date) : Date de création.
    - return_date (Date) : Date limite calculée.
    - status (Varchar) : "En cours" ou "Retourné".
4. **Categories (api_category)**
    - id (PK), name (Varchar).

VI. DÉPLOIEMENT & INFRASTRUCTURE

L'application est entièrement "Dockerisée" pour assurer la portabilité.

A. Docker Compose (docker-compose.yml)

Le fichier orchestre le lancement simultané des 3 services :

- **Service db** : Initialise PostgreSQL avec les variables d'environnement (User/Pass).
- **Service web (Backend)** : Construit l'image Python, attend la DB, lance les migrations et le serveur de développement.
    - *Commande spéciale Windows :* Utilisation de sh -c pour contourner les problèmes de fins de ligne (CRLF).
- **Service frontend** : Construit l'image Node.js et sert l'application sur le port 5173.

VII. SÉCURITÉ ET PERFORMANCES

A. Mesures de Sécurité

- **Mots de passe :** Hachés via PBKDF2 (Standard Django).
- **CORS (Cross-Origin Resource Sharing) :** Configuré pour n'accepter que les requêtes venant du Frontend (localhost:5173).
- **Injection SQL :** Prévenue par l'utilisation de l'ORM Django.

B. Gestion des Erreurs

- Le Frontend intercepte les erreurs 401 (Non autorisé) et redirige vers le Login.
- Le Backend renvoie des codes d'erreur explicites (400 Bad Request) si le stock est épuisé lors d'une tentative d'emprunt.
