Rapport de Projet
API Data Lake

Étudiant : Rick Georges YELEUMEU
Formation : M1 Data Engineering and AI

May 21, 2025

Contents

1. Introduction

Ce projet s'inscrit dans le cadre du cours de Data Integration. Il vise à concevoir une API RESTful capable d'exposer les données d'un Data Lake simulé. Le projet est réalisé avec Django, Django REST Framework, SQLite, JWT pour la sécurisation, et Streamlit pour une interface de test automatique.

2. Architecture Générale

L'architecture repose sur les principes classiques d'une API moderne :

- Backend Django (REST API)
- Base de données SQLite (mode local)
- Authentification JWT
- Fichiers CSV comme source simulée (bronze layer)
- Interface de test avec Streamlit

Un schéma est proposé en Figure 1.

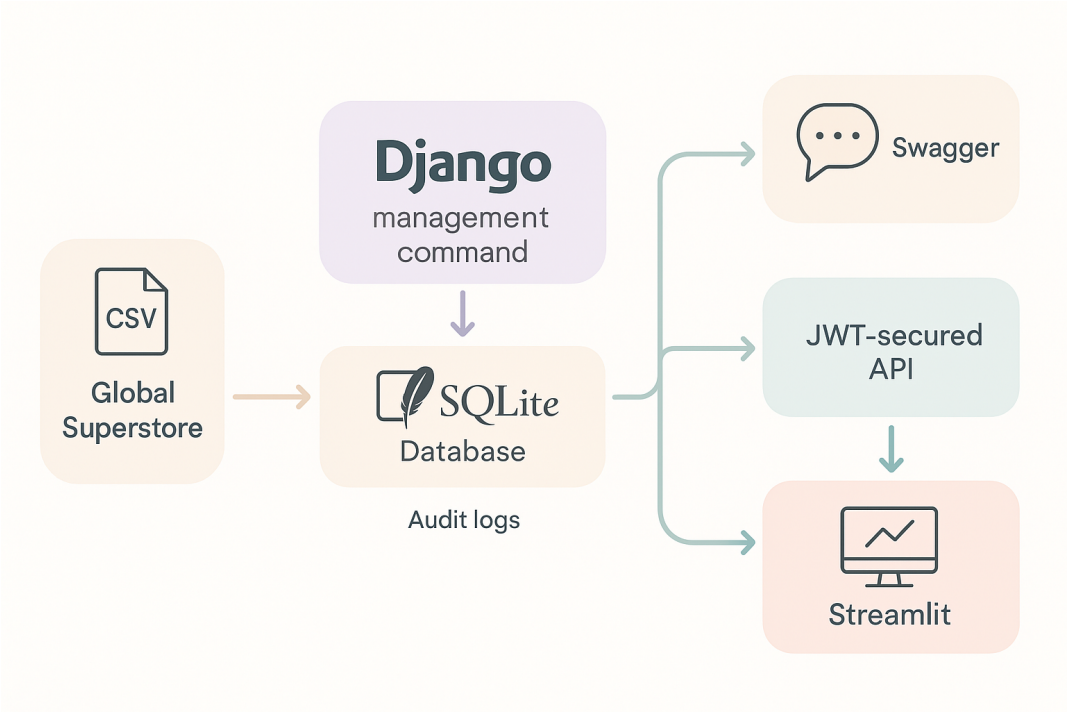

Figure 1: Architecture technique de l'API Data Lake

3. Modèle de Données

Le modèle principal à exposer est la table `Transaction`. Les champs sont nombreux : `customer_name`, `amount`, `country`, `status`, `rating`, `payment_method`, etc. Les données sont chargées depuis un fichier CSV enrichi via une commande personnalisée Django car accès impossible aux données bancaires utilisées lors des tps précédent ceci du à un problème avec Docker .

4. Sécurité et Gestion des Accès

- Authentification JWT à l'aide de `simplejwt`
- Superuser créé pour accéder à l'interface admin
- Middleware d'audit pour logger chaque appel à l'API
- Endpoints `/grant-access`, `/revoke-access` pour simuler des droits

5. Fonctionnalités de l'API REST

- `/transactions/` avec pagination, filtres dynamiques, projection de champs
- `/search/?q=` : recherche full-text multi-champs
- `/metrics/` : voir section suivante
- `/repush/` et `/train-ml/` : simulations de flux et modèle ML
- `/audit/`, `/lineage/` : traçabilité et journalisation

Toutes les routes sont documentées dans Swagger : `http://127.0.0.1:8000/swagger/`

6. Métriques Analytiques

Une dizaine de métriques sont exposées :

- `/metrics/avg-spend/`, `/metrics/avg-rating/`
- `/metrics/sales-per-country/`, `/metrics/daily-activity/`
- `/metrics/recent-spend/` (5 min)
- `/metrics/top-products/?x=5`, `/metrics/total-per-user/`

Chacune est calculée dynamiquement via Django ORM.

7. Traçabilité et Audit

Un middleware personnalisé capture tous les appels à l'API :

- utilisateur, méthode, endpoint, timestamp, body
- accessible via `/audit/` et `/lineage/<id>/`
- permet de démontrer la conformité RGPD et ISO27001 (simulée)

8. Fonctionnalités Avancées

Ces fonctionnalités illustrent des cas réels de pipelines Big Data :

- **Search full-text** : sans Elasticsearch, mais efficace
- **Repush Kafka simulé** : réenvoi de données dans un pipeline
- **Train ML simulé** : démarre une fonction symbolique d'entraînement

9. Interface de Test Automatique (Streamlit)

Une application Streamlit dédiée a été construite pour :

- Se connecter via JWT (`user:admin password:rootapi123`)
- Tester tous les endpoints
- Explorer les données en JSON ou DataFrame
- Lancer des tests : audit, lineage, repush, ML

Elle est autoportée (`streamlit run app.py`) et présente dans le livrable.

10. Tests et Scénarios de Validation

Le projet a été conçu pour passer les tests suivants :

- Authentification valide / invalide
- Transactions filtrées et projections
- Métriques cohérentes / mauvais paramètres
- Recherche efficace / cas déchec
- Audit précis / lineage logique

Tous ces tests sont accessibles via Streamlit, et loggés automatiquement.

11. Difficultés rencontrées et posture d'apprentissage

Le projet a présenté des difficultés réelles :

- **Problèmes de migration Django** : tables absentes ou champs ignorés, dus à des oublis de `makemigrations` ou `migrate`. Cela m'a appris à vérifier l'ordre des commandes, et parfois à purger la base.
- **Fichier CSV non conforme** : colonnes manquantes, encodage incorrect, dates sans timezone. Cela a impliqué d'inspecter la structure du fichier, et d'ajuster le parser avec `make_aware` et des try/except judicieux.
- **Middleware non fonctionnel** : corps de la requête inaccessible ("already consumed"). Cela m'a forcé à comprendre les limites du parsing de requêtes dans Django, et à chercher des solutions pédagogiques (ex: GET avec `request.GET`).
- **Swagger et routage 404** : erreurs de chemin, noms d'endpoint incohérents. Cela m'a appris l'importance de tester chaque route, et d'utiliser `reverse()` ou des noms explicites.

12. Limites et Améliorations

- Pas de base PostgreSQL (production)
- Pas de job Kafka réel (seulement simulé)
- Pas de gestion des versions multiples des transactions
- Absence de CI/CD ou de déploiement Docker

Mais l'architecture est préte à être industrialisée.

13. Conclusion

Le projet a permis d'implémenter un pipeline complet d'exposition de données de type Data Lake. Il m'a permis de consolider mes compétences en Django, API REST, audit, Streamlit et métriques. Le code est stable, testable, et prêt à être déployé.
Je suis capable de l'améliorer, de le défendre et de le maintenir.