

CONSTANT AIDAM

Music Video Tracker

Dossier Projet - Concepteur Développeur d'Applications

Sommaire

[1. Introduction](#)

[2. Compétences mises en oeuvre](#)

[3. Expression des besoins, objectifs et limites](#)

[4. Environnement technique](#)

[a. Environnement et approche de développement](#)

[b. Architecture du Projet](#)

[c. Technologies Utilisées](#)

[5. Conception de l'application](#)

[a. Principaux Cas d'Utilisation](#)

[b. Maquettes Des Interfaces Utilisateur](#)

[c. Modélisation des Données Applicative](#)

[d. Django Rest Framework](#)

[e. API Analytique](#)

[f. Modélisation du Datawarehouse](#)

[g. Consultation Des Statistiques](#)

[i. Sécurité de l'Application](#)

[j. Routes de l'Application](#)

[6. Gestion de projet](#)

[7. Réalisations présentant la mise en oeuvre des compétences](#)

[8. Jeu d'essai](#)

[9. Veille de Sécurité](#)

[10. Annexes](#)

1. Introduction

a. English Version

This project aims to develop an analytical web application for a blog specializing in music video clips. The application will track, discover, and analyze the performance of music videos shared by the blog. In the medium term, it will guide the blog's editorial line based on clip statistics, and in the long term, it will facilitate the creation of a database of directors and technical staff of the clips. Key features include a main page for clip management, performance analysis tools, a random clip discovery page, and a secure authentication system. The technical architecture relies on modern technologies such as Vite, React, Django REST Framework, and PostgreSQL, ensuring robustness, scalability, and security.

b. Version Française

Ce projet a pour objectif de développer une application web analytique pour un blog spécialisé dans les clips musicaux. L'application permettra de suivre, découvrir et analyser les performances des clips musicaux partagés par le blog. À moyen terme, elle orientera la ligne éditoriale du blog en fonction des statistiques des clips, et à long terme, elle facilitera la création d'une base de données des réalisateurs et du staff technique des clips. Les fonctionnalités clés incluent une page principale pour la gestion des clips, des outils d'analyse de performance, une page de découverte aléatoire de clips, et un système d'authentification sécurisée. L'architecture technique repose sur des technologies modernes telles que Vite, React, Django REST Framework, et PostgreSQL, assurant robustesse, évolutivité et sécurité.

2. Compétences mises en oeuvre

a. Activité 1 : Développer une application sécurisée

i. Développer des interfaces utilisateur

Implémentation : Utilisation de Vite et React pour créer des interfaces utilisateur réactives et intuitives. Les maquettes de la page principale, de la page de focus sur les performances, et de la page discovery sont développées pour offrir une navigation fluide et une expérience utilisateur optimale.

Approche Sécurisée : Intégration de bonnes pratiques pour sécuriser les interfaces via des validations côté client et l'utilisation de bibliothèques sécurisées.

ii. Développer des composants métier d'une application

Implémentation : Développement de composants backend en utilisant Django REST Framework pour la gestion des opérations CRUD et l'authentification. Les composants incluent des endpoints sécurisés pour la création, la récupération, la mise à jour et la suppression de clips et de profils utilisateurs.

Approche Sécurisée : Application des principes de sécurité tels que l'authentification JWT et les permissions utilisateur pour garantir que seules les actions autorisées peuvent être effectuées.

iii. Contribuer à la gestion d'un projet informatique

Implémentation : Utilisation de GitHub pour la gestion de projet et la gestion de version. Application des bonnes pratiques appliquées en entreprise pour suivre les progrès du projet, et ajuster les priorités en fonction des besoins.

Approche Sécurisée : Mise en place d'un processus de revue de code pour garantir la qualité et la sécurité du code avant le déploiement. Étant développeur solo, ces revues s'effectuent en fonction des disponibilités avec des collègues, formateurs ou des LLM spécialisés type Chat GPT.

b. Activité 2 : Concevoir et développer une application sécurisée organisée en couches

i. Analyser les besoins et maquetter une application

Implémentation : Analyse des besoins fonctionnels et non fonctionnels du projet. Création de maquettes pour les différentes pages de l'application, y compris les fonctionnalités de gestion des clips, l'analyse des performances, et la découverte de vidéos.

Approche Sécurisée : Validation des maquettes avec les parties prenantes pour garantir que toutes les exigences de sécurité et de confidentialité sont prises en compte dès le début du projet.

ii. Définir l'architecture logicielle d'une application

Implémentation : Définition d'une architecture en couches comprenant le frontend (React), le backend (Django REST Framework), l'API analytique (Flask), et les services de stockage et de gestion de données (PostgreSQL, BigQuery, dbt, Kestra).

Approche Sécurisée : Conception d'une architecture modulaire avec des couches séparées pour faciliter la gestion des permissions et des accès sécurisés entre les composants.

iii. Concevoir et mettre en place une base de données relationnelle

Implémentation : Conception d'une base de données relationnelle avec PostgreSQL pour stocker les informations de base sur les clips et les utilisateurs. Création de schémas de base de données pour soutenir les besoins fonctionnels de l'application.

Approche Sécurisée : Utilisation de l'orm de Django en interface avec la base de données pour bénéficier des garanties en termes de sécurité et bonnes pratiques du framework.

iv. Développer des composants d'accès aux données SQL et NoSQL

Implémentation : Développement d'une API backend et d'une API analytique pour l'accès aux données de la base de données relationnelle et du datawarehouse pour les analyses de données. Utilisation de dbt Cloud pour la transformation des données.

Approche Sécurisée : Implémentation d'un système d'authentification et de routes privées, isolation des clés API externes et identifiants BigQuery en dehors du backend.

c. Activité 3 : Préparer le déploiement d'une application sécurisée

i. Préparer et exécuter les plans de tests d'une application

Implémentation : Élaboration de plans de test détaillés pour les tests fonctionnels, de performance et de sécurité.

Approche Sécurisée : Mise en place d'une stratégie de tests automatisés pour détecter les régressions et les failles de sécurité, ainsi qu'une stratégie de déploiement continu pour assurer la qualité et la sécurité du code à chaque mise à jour.

3. Expression des besoins, objectifs et limites

a. Contexte et Objectifs

i. Contexte :

Le projet concerne la création d'une application web analytique pour un média/blog spécialisé dans les clips musicaux. Le blog est dédié à la découverte musicale et à la promotion des arts visuels dans les projets musicaux. Il offre régulièrement des playlists mettant en avant des nouvelles sorties de clips musicaux d'artistes émergents et peu connus, ainsi que des focus et des interviews sur les métiers du clip musical. La volonté est de créer une application web annexe et indépendante qui viendrait en complément et en support de l'activité du blog.

ii. Objectifs :

1. Objectifs immédiats :

- Permettre aux utilisateurs de garder une trace des clips musicaux partagés par le blog.
- Offrir un outil de découverte de nouveaux clips et de suivi de la progression des artistes.
- Fournir des statistiques sur les performances des clips.
- Assurer la sécurité et la gestion des accès via une authentification utilisateur.

2. Objectifs à moyen terme :

Mettre en place un suivi des performances des clips pour orienter la ligne éditoriale du blog. Cela inclut la possibilité de revenir sur des clips qui deviennent populaires plusieurs mois après leur publication et de décider quels clips mettre en avant en fonction de leur performance.

Créer des rapports d'analyse basés sur les statistiques des clips pour optimiser les choix éditoriaux et les futures playlists.

3. Objectifs à long terme :

Développer une base de données des réalisateurs et du staff technique des clips en extrayant et en analysant les crédits dans les descriptions des vidéos YouTube. Cela permettra de calculer des statistiques par réalisateur et staff technique, et d'orienter la ligne éditoriale en fonction des contributions des professionnels.

Intégrer des fonctionnalités avancées d'analyse et de visualisation des données pour améliorer la prise de décision et l'orientation éditoriale du blog.

b. Public Cible

- Utilisateurs/Collaborateurs du blog : Gestion des clips musicaux, découverte de nouveaux clips, et analyse des performances.
- Développeurs et Analystes : Maintenance et évolution de l'application.

c. Fonctionnalités Requises

i. Page Principale

Affichage d'un tableau répertoriant tous les clips enregistrés avec les informations suivantes :

- Titre du clip
- Chaîne YouTube de l'artiste
- Date d'ajout
- Miniature
- Bouton permettant d'ouvrir un menu pour l'ajout de nouvelles vidéos via une URL YouTube.

ii. Menu d'Ajout de Vidéo

Formulaire permettant l'ajout de nouvelles vidéos en entrant une URL YouTube. Validation de l'URL pour s'assurer qu'elle pointe vers une vidéo valide sur YouTube.

iii. Analyse des Performances

- Architecture analytique en arrière-plan pour récupérer les données des vidéos enregistrées une fois par jour.
- Calcul et affichage des statistiques de performances des clips (par exemple : vues, likes, commentaires) sur une période des 7 derniers jours.

iv. Page Focus sur les Performances

- Affichage des détails des performances d'un clip spécifique.
- Présentation des statistiques pour les 7 derniers jours.
- Option d'affichage de détails supplémentaires (par exemple : évolution quotidienne des vues, engagements).



v. Page Discovery

- Design simple pour découvrir une vidéo aléatoire parmi les vidéos soumises par les utilisateurs.
- Bouton permettant de lancer une vidéo au hasard.

vi. Authentification

- Système d'authentification pour sécuriser l'accès aux pages principales, d'analyse des performances et de découverte.
- Gestion des utilisateurs : création de compte, connexion, déconnexion, changement de mot de passe.

d. Interface Utilisateur

i. Conception

- Interface claire et intuitive, en ligne avec les lignes directrices du blog.
- Design adaptatif pour une utilisation sur différents appareils (ordinateurs, tablettes, smartphones).

ii. Ergonomie

- Navigation fluide entre les pages et fonctionnalités.
- Interface utilisateur conviviale pour l'ajout de vidéos et la consultation des performances.

e. Limites

- Budget :

Le budget doit rester bas, ce qui implique des choix technologiques et de fonctionnalités optimisées pour minimiser les coûts.

- Constraintes de Temps :

Projet développé en tant que développeur full stack solo, ce qui limite les ressources disponibles pour le développement et les tests.

- Scalabilité :

La solution doit être conçue en tenant compte de la possibilité d'extension future, bien que les fonctionnalités initiales se concentreront sur les besoins immédiats.

- Ressources :

Le développement doit être réalisé en respectant les capacités du développeur solo, avec des ressources limitées pour l'intégration et l'analyse avancée.

4. Environnement technique

a. Environnement et approche de développement

Dans le contexte d'une formation en alternance, partagée entre périodes de cours et période d'entreprise, il a été important de mettre en place un environnement de développement qui s'adapterait bien au rythme instable et au temps parfois très limité à consacrer au projet.

Travaillant dans différentes conditions selon le contexte école/entreprise, j'ai essayé dès le départ d'organiser mon environnement de développement de manière à ce qu'il soit très adaptable. Pour chaque couche de l'application, l'objectif était de pouvoir continuer à contribuer au développement sans difficultés selon que je suis sur un environnement Mac OS en entreprise ou Ubuntu pendant les périodes de cours. Cette approche me permettrait également d'appliquer des bonnes pratiques qui pourraient faciliter l'ajout de contributeurs au projet.

Cela a donc impliqué l'utilisation de Git, Docker et de l'extension Dev Container via Visual Studio Code pour développer dans des environnements conteneurisés. Cette approche m'a permis de pouvoir avancer efficacement sur ce projet sans être dépendant d'une machine, d'un OS ou d'un environnement précis.

b. Architecture du Projet

Sur le plan technique, en me basant sur l'expression de besoin, en prenant compte de mes appétences technologiques et dans la continuité de ma recherche d'efficacité et de modularité dans mon environnement de développement, j'ai choisi de d'appliquer le principe de "separation of concerns" à l'architecture de mon projet.

Cette stratégie m'a permis d'appréhender chaque couche de l'application comme un projet indépendant et de pouvoir passer rapidement de l'un à l'autre en fonction du

contexte, des notions abordées pendant les périodes de cours et de ma montée en compétence. Tout en gardant les aspects de scalabilité, de budget et de ressources à l'esprit.

Les diagrammes ci-dessous présentent une vue d'ensemble de l'architecture multi-couches visée. L'idée principale est de pouvoir bénéficier d'un backend simple et robuste pour gérer les opérations critiques et développer une API analytique plus flexible en parallèle pour effectuer les tâches plus complexes mais décorrélées des fonctionnalités vitales de l'application.



Fig. .1. Diagramme de Composants du projet | Généré avec l'aide de PlantText UML Editor.

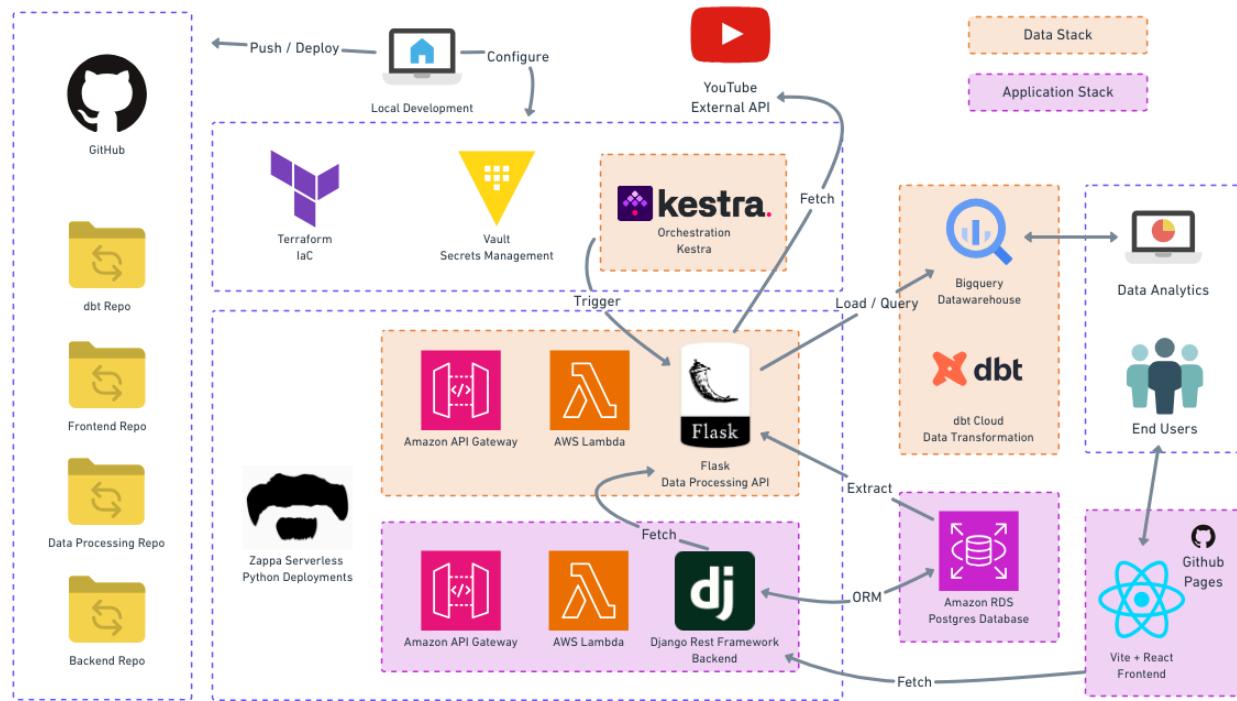


Fig. .2. Architecture cible du système | Crée avec Whimsical.

c. Technologies Utilisées

i. Stack Applicative :

1. Front-end :

Technologies : Vite, React, JS

Détails : Vite est utilisé pour le développement rapide du frontend avec React et Javascript pour construire les interfaces utilisateur interactives. React permet une gestion efficace de l'état de l'application et une expérience utilisateur fluide. Pour l'aspect style et design je me suis principalement basé sur l'utilisation de composants Shadcn/ui dont la

philosophie modulaire correspond bien à React et dont l'intégration avec Tailwind m'a permis d'adopter des pratiques plus solides dans la gestion du design et du style des interfaces.

2. Back-end :

Technologies : Django REST Framework

Détails : Django REST Framework (DRF) est utilisé pour construire les API nécessaires à la gestion des clips, des utilisateurs, et des statistiques. DRF fournit une interface robuste, sécurisée et évolutive pour la création, la récupération, la mise à jour et la suppression des données.

3. Base de Données :

Technologies : Base de données PostgreSQL

Détails : Base de données relationnelle open source utilisée dans le projet pour stocker les données critiques de l'application, telles que les informations sur les utilisateurs et les clips. Le choix se justifie par sa fiabilité, sa richesse fonctionnelle et ses capacités d'extensibilité en vue des évolutions et du déploiement du projet.

ii. Stack Data :

1. API Analytique :

Technologies : Flask

Détails : Flask est utilisé pour l'API analytique qui récupère et traite les données de performance des vidéos. Elle interagit avec les services externes pour collecter des données et calculer les statistiques. Flask est un micro-framework léger et flexible, idéal pour développer des services RESTful. Il facilite l'intégration avec divers services externes pour collecter des données. Cette API joue un rôle crucial dans la fourniture de données de performance pour le reporting et la prise de décision. Alternative à Django,

cette API indépendante offre plus de flexibilité et d'efficacité pour le développement de fonctionnalités annexes aux logiques purement backend.

2. Datawarehouse :

Technologies : BigQuery

Détails : BigQuery est utilisé pour le stockage, la transformation et l'analyse des données générées par les clips et leurs performances. Il permet des requêtes SQL rapides et efficaces sur de grands ensembles de données. Cette base de données secondaire permet d'enrichir les données transactionnelles (OLTP) du backend et de les re-modéliser en suivant une logique analytique (OLAP).

3. Data Transformation :

Technologies : dbt Cloud

Détails : dbt (Data Build Tool) est utilisé pour la transformation des données dans le data warehouse. Il permet de créer des modèles de données reproductibles et maintenables en utilisant SQL. dbt facilite le nettoyage, l'enrichissement et la transformation des données en pipelines de données automatisés, permettant des analyses plus approfondies et des rapports précis. L'intégration avec BigQuery et l'automatisation des tâches de transformation rendent le processus de préparation des données plus efficace. Il facilite également certaines bonnes pratiques de développement telles que le versioning et les tests.

4. Orchestration :

Technologies : Kestra (self-hosted)

Détails : Kestra est utilisé pour orchestrer les workflows de données, y compris les tâches d'extraction et de chargement (Extract and Load) entre les différents composants

du système. Il permet de planifier, automatiser et surveiller les processus de traitement des données de manière flexible et visuelle. Kestra facilite la coordination des tâches complexes et la gestion des dépendances, assurant ainsi une exécution fluide des processus de données. Son paradigme déclaratif facilite l'adoption d'une approche DevOps avec la possibilité de configurer les workflows via Terraform.

iii. Stack DevOps :

1. Versioning and CI/CD :

Technologies : Github

Détails : Github est utilisé pour le versioning du code source et la gestion des dépôts de code. Il facilite le suivi des modifications et l'intégration continue/déploiement continu (CI/CD) grâce à des workflows automatisés. Les fonctionnalités de pull requests et de gestion des branches permettent une gestion efficace des versions et des déploiements.

2. Déploiement Python :

Technologies : Zappa

Détails : Zappa est utilisé pour le déploiement des applications Python, notamment celles développées avec Django ou Flask, sur AWS Lambda. Zappa simplifie le déploiement d'applications dites "serverless", en gérant les aspects liés à l'infrastructure et en assurant la scalabilité automatique. Cette approche serverless réduit les coûts d'infrastructure et simplifie la gestion des environnements de déploiement.

3. Secrets Management :

Technologies : HCP Vault (self-hosted dans une EC2)

Détails : HCP Vault est utilisé pour gérer et sécuriser les secrets et les configurations sensibles au sein de l'infrastructure. Il permet de stocker, de gérer et de distribuer des

secrets tels que les clés API et les mots de passe de manière sécurisée. La gestion des secrets est essentielle pour protéger les informations sensibles et assurer la conformité aux bonnes pratiques de sécurité.

4. Infrastructure as Code :

Techonologies : Terraform

Détails : Terraform est utilisé pour la gestion de l'infrastructure en tant que code (IaC), permettant de définir, de provisionner et de gérer les ressources cloud de manière déclarative. Son intégration avec Vault, Kestra et Google Cloud Platform (GCP) assure une cohérence entre les environnements de développement, de test et de production, tout en automatisant le déploiement de certaines ressources.

5. Conception de l'application

a. Principaux Cas d'Utilisation

En m'appuyant sur l'expression de besoin, j'ai pu proposer une première conceptualisation des cas d'utilisation les plus importants :

- Cas d'Utilisation 1 : Ajouter un nouveau clip.
- Cas d'Utilisation 2 : Supprimer un clip.
- Cas d'Utilisation 3 : Voir la liste des clips ajoutés.
- Cas d'Utilisation 4 : Consulter les performances d'un clip spécifique.
- Cas d'Utilisation 5 : Découvrir un clip aléatoire.

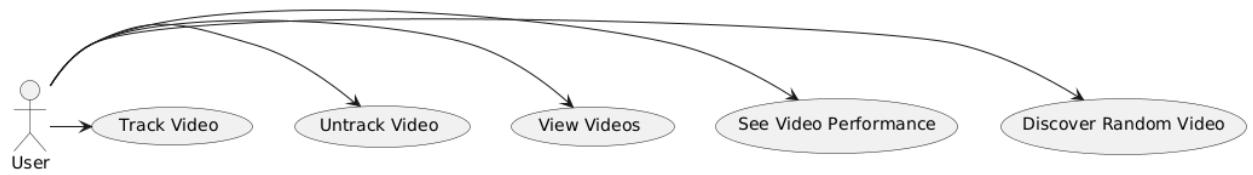


Fig. 3. Diagramme de Cas d'Usages | Généré avec l'aide de PlantText UML Editor.

b. Maquettes Des Interfaces Utilisateur

Les principaux cas d'usages étant identifiés, j'ai ensuite élaboré les maquettes des interfaces les plus importantes de l'application.

i. Page Principale et Navigation

Affichage : Tableau des clips avec titre, chaîne, date d'ajout, miniature.

Interactivité : Bouton pour ouvrir un menu d'ajout des vidéos via une URL YouTube.

Formulaire : Champs pour entrer une URL YouTube.

Validation : Vérification de l'URL pour s'assurer de sa validité.

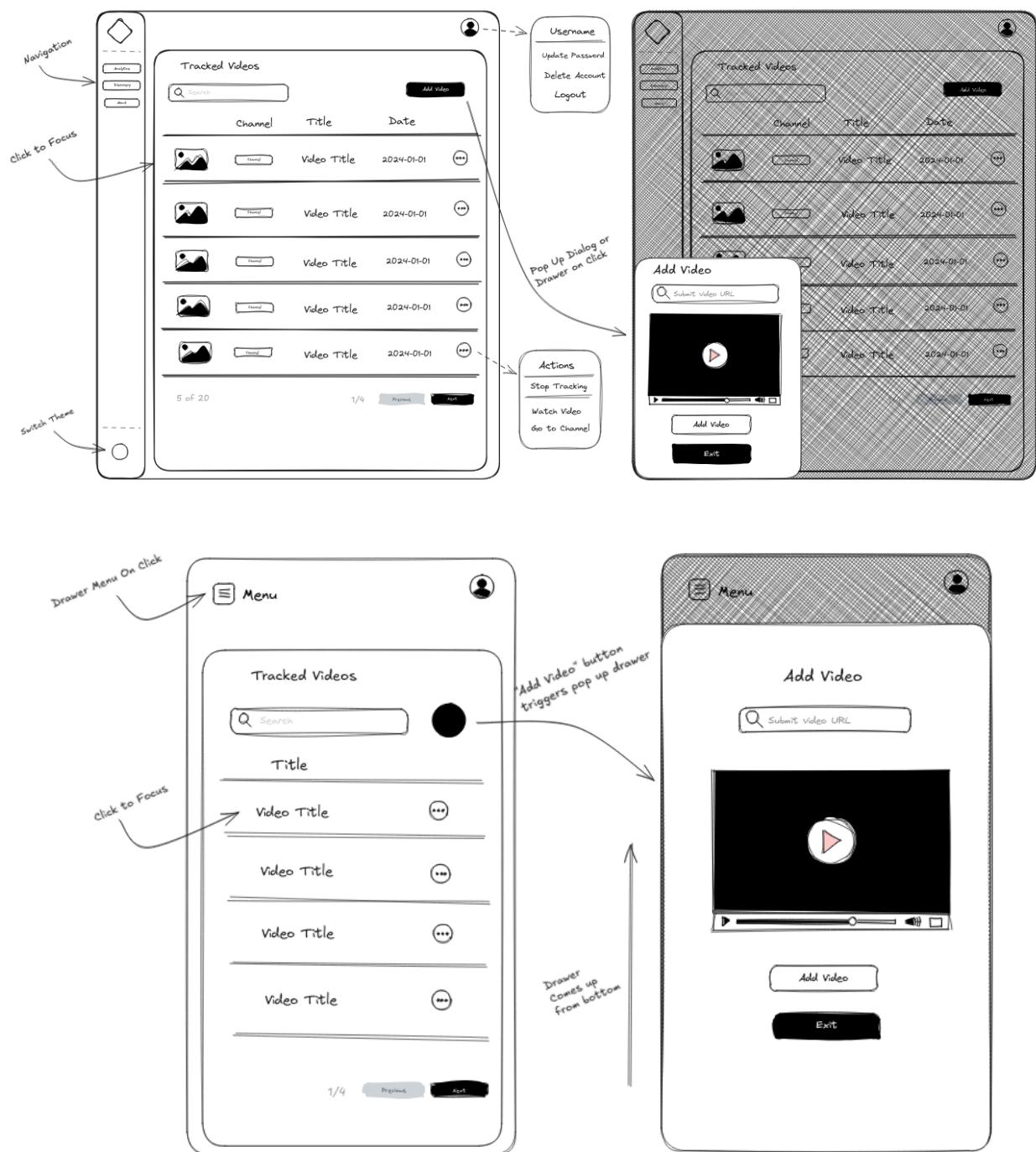


Fig. 4. Maquettes de l'Interface Desktop et Mobile de la page principale | Crées avec Excalidraw.

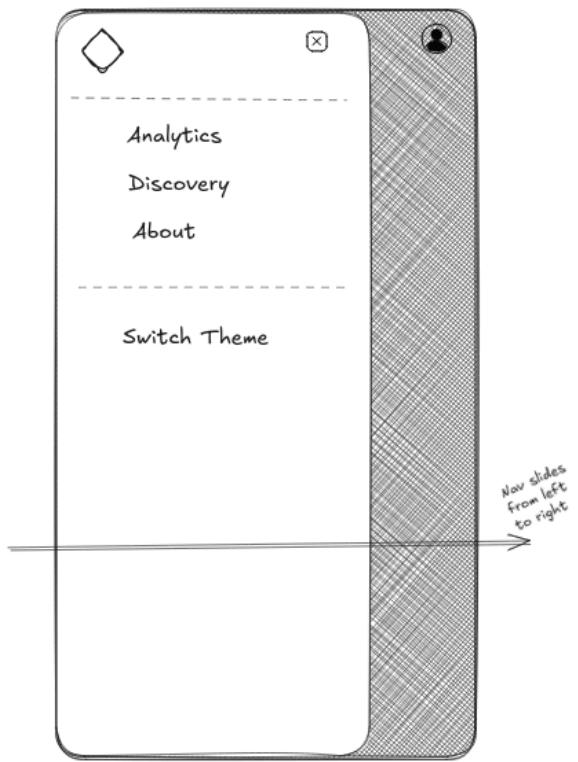


Fig. 5. Maquettes de l'Interface Mobile du menu de navigation | Créeé avec Excalidraw.

ii. Page Focus sur les Performances

Affichage : Détails des performances d'un clip, statistiques des 7 derniers jours.

Graphiques : Évolution quotidienne des vues et engagements.

Maquettes :

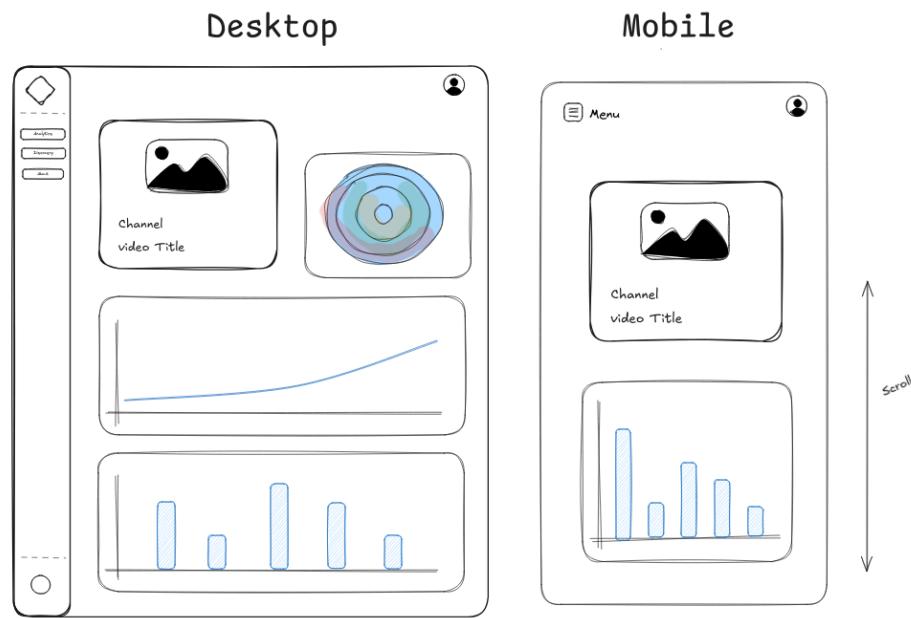
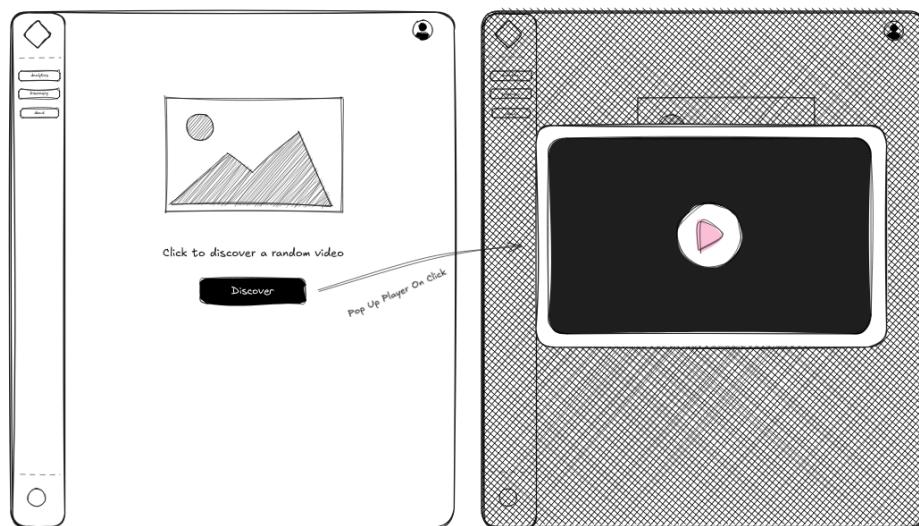


Fig. 6. Maquettes de l'Interface Desktop et Mobile de la page Focus | Crées avec Excalidraw.

iii. Page Discovery

Mécanisme : Découverte aléatoire de clips parmi ceux soumis par les utilisateurs.

Interaction : Bouton pour lancer une vidéo au hasard.



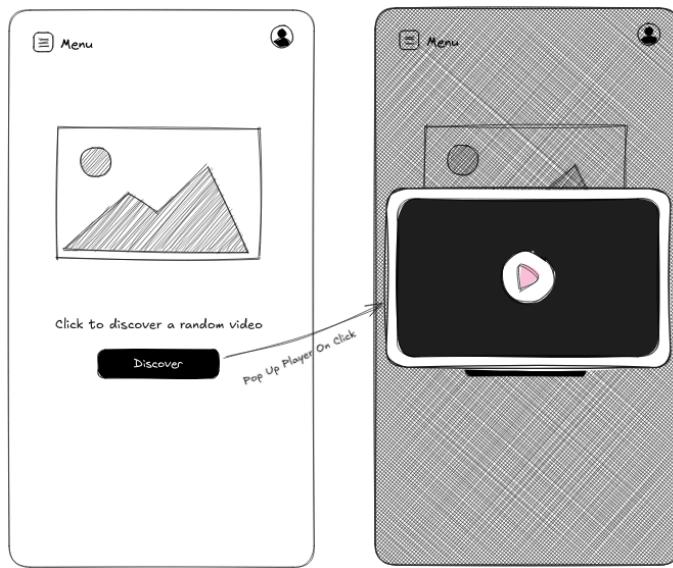


Fig. 7. Maquettes de l'Interface Desktop et Mobile de la page Discovery | Crées avec Excalidraw.

c. Modélisation des Données Applicative

Avec une bonne idée des interfaces à alimenter, j'ai pu appréhender la conceptualisation du backend de l'application. En restant fidèle à l'architecture multi-tier du projet, j'ai cherché à élaborer le modèle de données minimal pour permettre la mise en place des fonctionnalités identifiées.

Cela m'a permis de distinguer trois entités principales :

- User : La représentation de l'utilisateur de l'application, permet d'implémenter une logique d'authentification et de droits.
- Source : La représentation d'un clip musical, un clip est identifié par une URL. C'est l'information minimale dont on a besoin de la part de l'utilisateur. Celle-ci pourra être enrichie via l'API analytique dans un second temps. Dans la phase de conception, la notion de "Source" a été choisie pour décorreler l'entité de

YouTube et faciliter la prise en charge éventuelle de nouvelles sources à l'avenir, comme Spotify ou Vimeo par exemple.

- UserSource : La représentation de la relation entre un User et une Source. Un User peut être lié à plusieurs Sources et une Source peut être liée à plusieurs Users.

i. Diagramme de classes

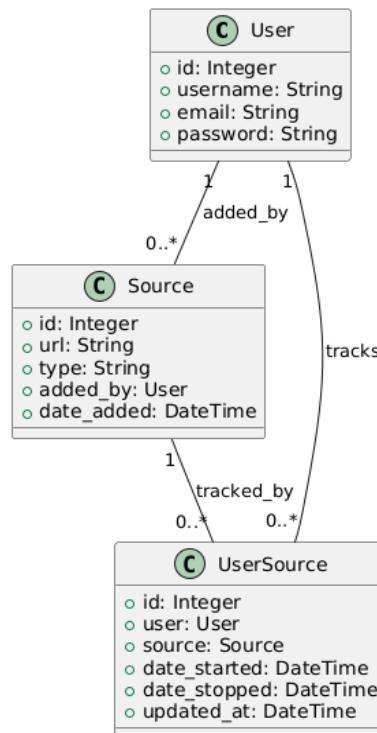


Fig. 8. Diagramme de Classes du backend | Généré avec l'aide de PlantText UML Editor.

Ce diagramme m'a servi de référence pour la mise en œuvre des fonctionnalités et assure que la structure des données est bien alignée avec les besoins fonctionnels, tout en offrant une base solide pour l'évolution future de l'application.

ii. Modélisation de la Base de Données Applicative

Pour mettre en œuvre le modèle de données évoqué dans Django, il est essentiel de comprendre le modèle architectural MVT (Model-View-Template) utilisé par le

framework. Le MVT est une variante du modèle MVC (Model-View-Controller) qui simplifie la gestion des données, de la présentation et des interactions utilisateur dans les applications web.

Dans Django, le modèle représente la structure des données et est défini en utilisant des classes Python qui héritent d'une classe Model pré définie. Chaque classe modèle correspond à une table dans la base de données. Pour notre cas, nous créerons des modèles pour les entités User, Source, et UserSource. Ces modèles définissent les attributs nécessaires et les relations entre les entités, telles que les relations many-to-many entre User et Source :

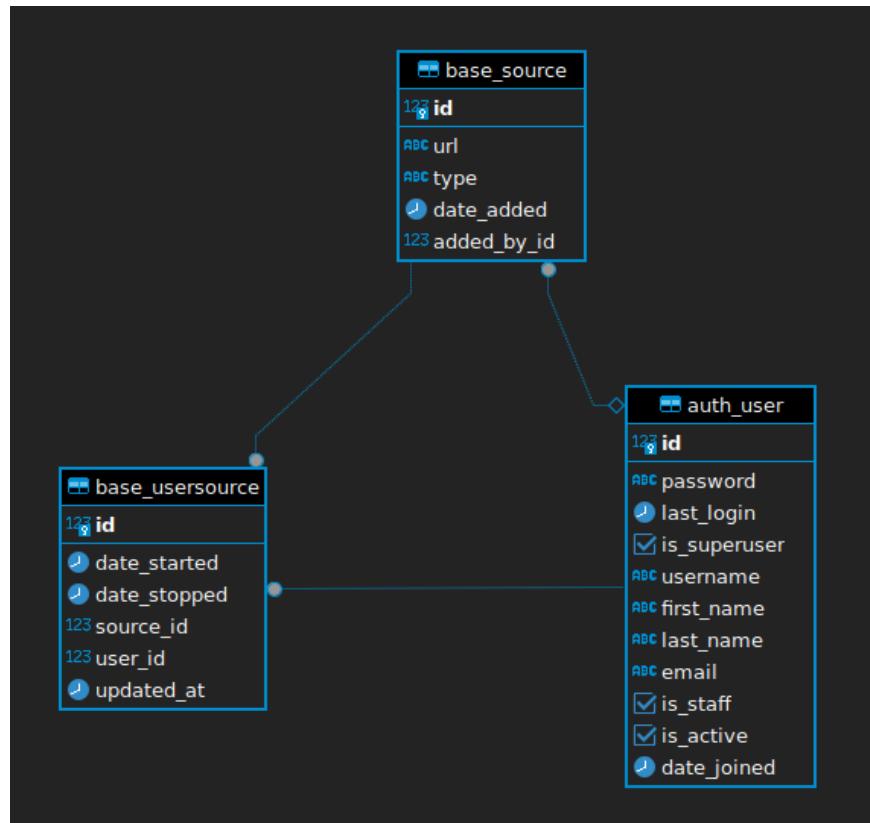


Fig. 9. Diagramme Entité Relation de la BDD backend | Généré avec DBeaver.

La vue en Django est responsable de la logique métier et de la gestion des requêtes HTTP. Elle récupère les données du modèle, les traite, et les envoie à un template pour le rendu. Les vues sont définies dans des fonctions ou des classes dans le

fichier views.py et peuvent interagir avec les modèles pour gérer les opérations CRUD (Create, Read, Update, Delete).

d. Django Rest Framework

Cependant dans notre contexte où Django est utilisé avec Django REST Framework (DRF), le modèle architectural MVT est adapté pour exposer des API plutôt que des templates HTML, ce qui permet une architecture avec un frontend React. DRF est un puissant framework complémentaire à Django qui facilite la création d'API RESTful pour gérer les interactions entre le backend et le frontend.

En utilisant DRF, on définit des "serializers" qui transforment les données du modèle en formats JSON, adaptés pour la communication avec le frontend React. Les viewsets et routers de DRF permettent de créer des endpoints API pour les opérations CRUD (Create, Read, Update, Delete) sur les entités telles que User, Source, et UserSource. Ces endpoints sont ensuite consommés par le frontend React pour afficher et interagir avec les données. Cela me permet donc d'intégrer la robustesse du framework en tant que backend dans un architecture modulaire.

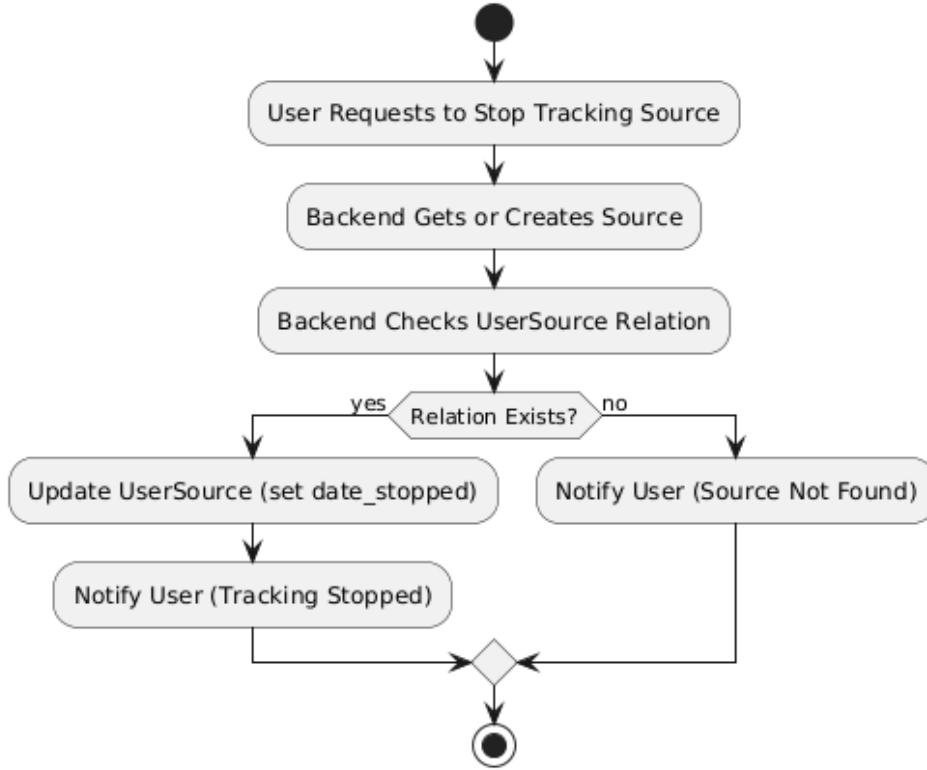


Fig. 10. Diagramme d'Activité de la fonctionnalité "Stop Tracking" du backend | Généré avec l'aide de PlantText UML Editor.

e. API Analytique

Pour optimiser l'efficacité et la modularité de notre backend tout en respectant l'architecture multi-tier, nous avons choisi de déléguer les tâches de traitement des données à une API Flask distincte. Python, étant un langage très apprécié pour le traitement de données en raison de sa richesse en bibliothèques spécialisées (comme Pandas ou SQL Alchemy), il s'avère particulièrement adapté pour ces tâches. Flask, avec sa légèreté et sa flexibilité, nous permet de développer une API dédiée au traitement des données qui peut évoluer indépendamment du backend Django. Cette séparation garantit non seulement une meilleure organisation et maintenabilité du code, mais aussi une spécialisation des outils pour des tâches spécifiques.

l'API analytique aura deux fonctions principales :

- Enrichissement de la donnée : extraire les données (Source et UserSource) de la base applicative pour les charger dans un datawarehouse, et les enrichir en requêtant l'API YouTube de manière journalière. Des endpoints seront créés pour chaque tâche.
- Point d'accès aux données enrichies : des endpoints seront créés pour fournir des données enrichies au frontend via le backend.

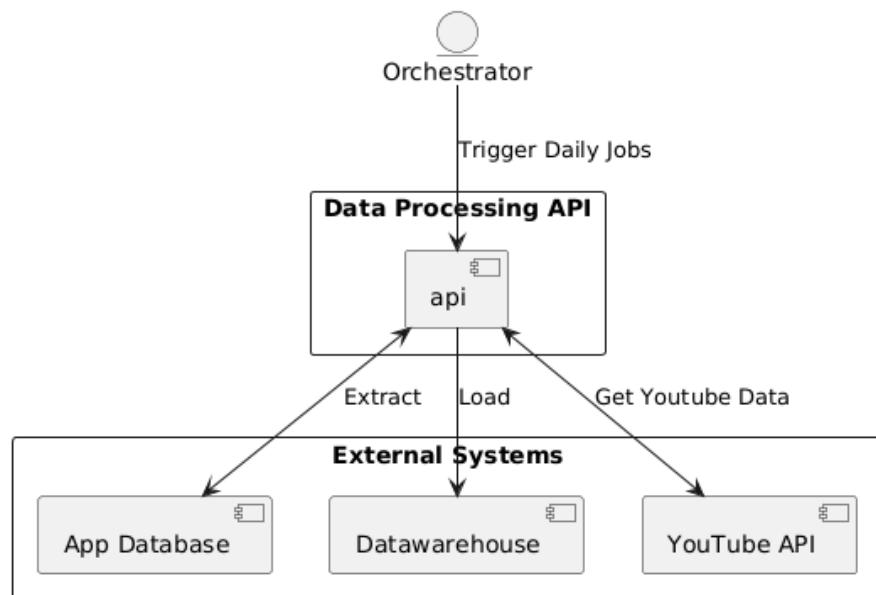


Fig. 11. Diagramme de Flux de Données API analytique | Généré avec l'aide de PlantText UML Editor.

f. Modélisation du Datawarehouse

Une fois les données brutes chargées dans le data warehouse, nous utiliserons dbt Cloud pour effectuer la transformation et la modélisation des données selon la méthode de Kimball.

La méthode de Kimball, qui est un des paradigmes les plus répandus pour la conception de data warehouses, repose sur la création de schémas en étoile ou en flocon pour structurer les données de manière à optimiser les requêtes analytiques et

faciliter l'exploration des données. En appliquant cette méthode, nous organisons les données en faits et dimensions, ce qui améliore la clarté et la performance des analyses. Grâce à dbt Cloud, nous automatisons et versionnons ces transformations, assurant ainsi une approche rigoureuse et reproductible pour la gestion de nos données analytiques qui pourront être servies par l'API Flask.

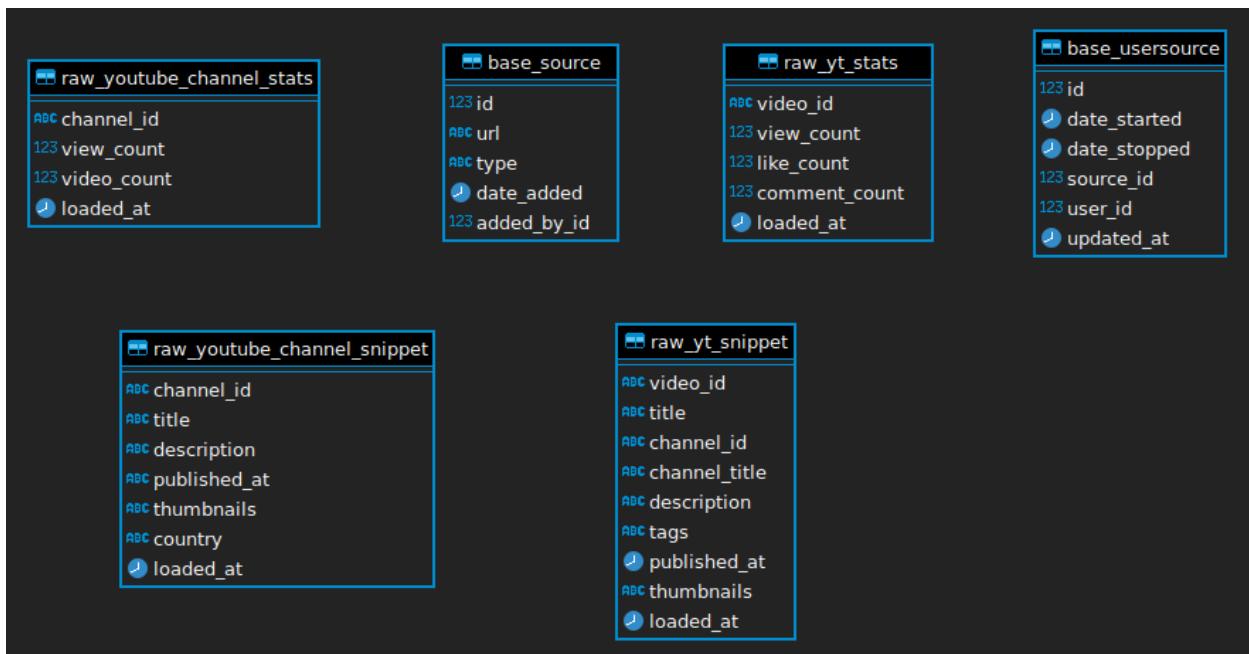


Fig. 12. Diagramme Entité Relation des données brutes chargées dans le Datawarehouse | Généré avec DBeaver.

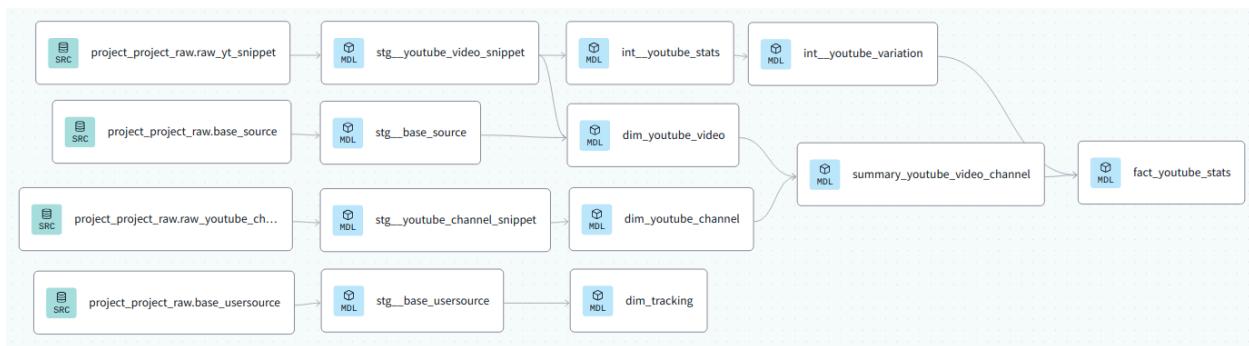


Fig. 13. Séquences de transformations des données brutes dbt | Généré avec dbt.



Fig. 14. Diagramme Entité Relation du modèle de données cible dans le Datawarehouse | Généré avec DBeaver.

g. Consultation Des Statistiques

Avec toutes les briques en place, la fonctionnalité de consultation des statistiques devient réalisable, toutes les couches se mettent en place et jouent leur rôle pour répondre au besoin. La fonctionnalité de consultation des statistiques permet aux utilisateurs de visualiser les performances des clips musicaux dans le temps. Ce processus commence par la collecte quotidienne des données par l'API Flask, elles sont enrichies et stockées dans le data warehouse. Une fois les données transformées, elles sont disponibles pour des analyses détaillées.

Le backend Django, utilisant Django REST Framework, expose des endpoints API qui récupèrent les données transformées du data warehouse via l'API Flask et les transmettent au frontend React. Ce dernier affiche les statistiques de manière interactive, offrant aux utilisateurs une vue claire des performances des clips, telles que les vues, les likes, et les commentaires. Ainsi, chaque composant du système joue un rôle crucial, assurant une intégration fluide et une réponse rapide aux demandes des utilisateurs.

Diagramme de séquence illustrant la récupération des statistiques d'un clip pour un utilisateur donné :

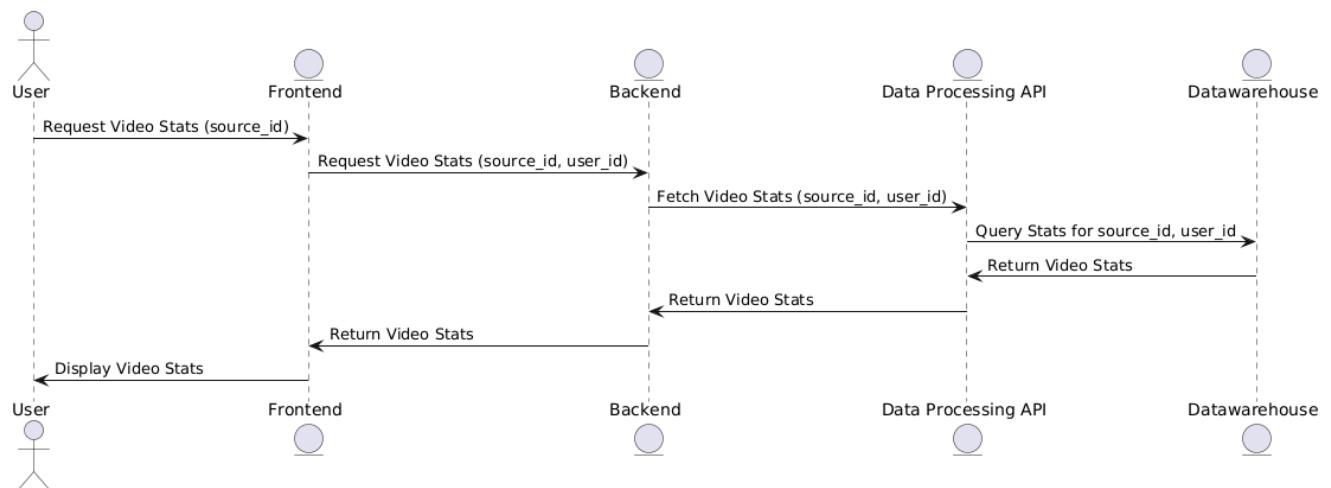


Fig. 15. Diagramme de séquence : récupération des statistiques d'un clip par un utilisateur | Généré avec l'aide de PlantText UML Editor.

h. Sécurité de l'Application

Dans l'architecture actuelle, plusieurs mesures de sécurité sont intégrées pour assurer la protection des données et des systèmes. Le frontend React utilise JSON Web Tokens (JWT) pour gérer l'authentification des utilisateurs. Les tokens JWT permettent de maintenir l'état de connexion tout en facilitant la communication sécurisée entre le frontend et le backend Django REST Framework (DRF). Le backend Django, pour sa part,

applique une authentification basée sur JWT pour vérifier les requêtes des utilisateurs et accorder les accès appropriés en fonction des rôles et des autorisations définis.

La communication entre le backend Django et l'API analytique Flask est sécurisée via des clés API et des protocoles de chiffrement tels que HTTPS, garantissant que les échanges de données restent confidentiels et protégés contre les interceptions non autorisées. En outre, la gestion des secrets, y compris les clés API et les informations d'accès sensibles, est réalisée par HCP Vault, qui est hébergé en version open source sur une instance EC2, offrant ainsi un contrôle rigoureux sur leur stockage et leur utilisation.

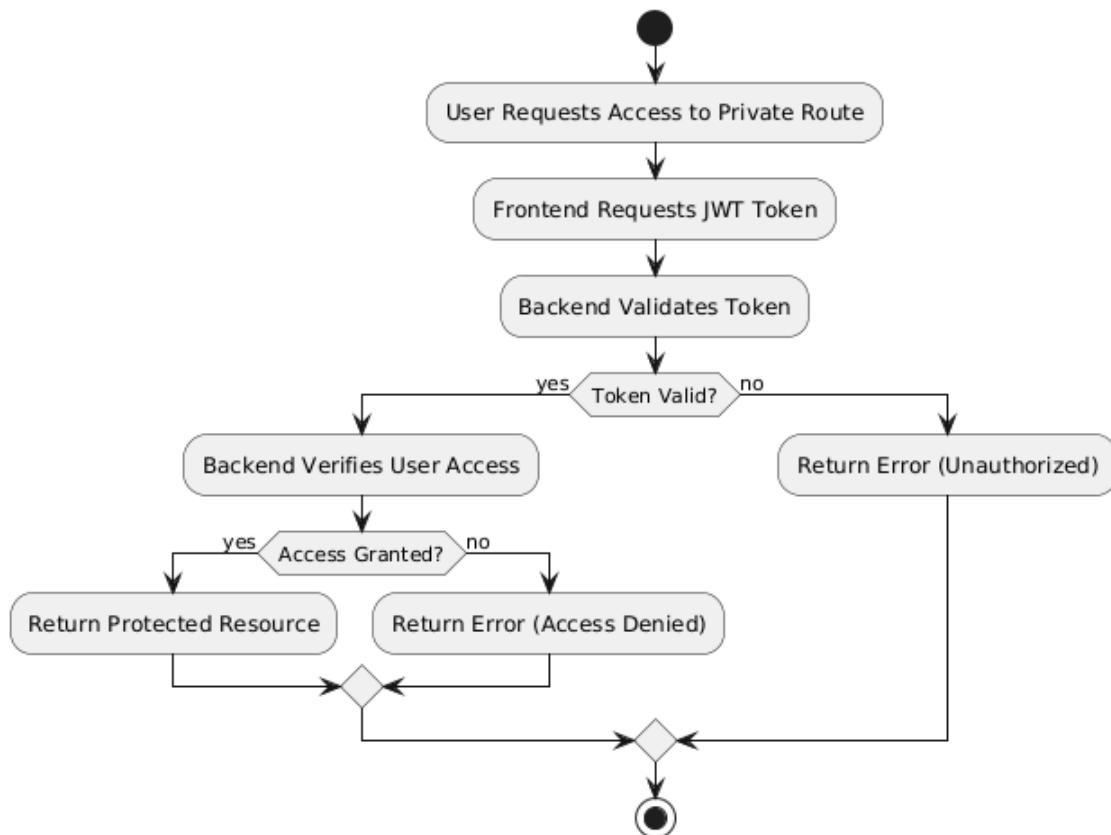


Fig. 16. Diagramme d'Activité : Fonctionnement des routes privées dans la couche backend I
Généré avec l'aide de PlantText UML Editor.



i. Routes de l'Application



Fig. 17. Routes API backend | Généré avec l'aide de PlantText UML Editor.



Fig. 18. Routes API Analytique | Généré avec l'aide de PlantText UML Editor.

j. Plan de Déploiement

i. Déploiement en Beta

Le projet est pré-déployé en version beta avec des solutions adaptées pour une gestion économique. Le backend Flask et Django est déployé via Zappa sur AWS Lambda, tandis que le frontend est hébergé gratuitement sur GitHub Pages. Les bases de données utilisent un plan gratuit Aiven pour PostgreSQL et un plan gratuit Google Cloud Platform (GCP) pour BigQuery, permettant une gestion économique tout en facilitant les tests. Quelques tests sont automatisés mais la majorité sont manuels et les commandes de déploiement sont effectuées en local.

ii. Déploiement en Production

Pour un déploiement robuste en production, des workflows CI/CD seront mis en place avec GitHub Actions pour automatiser les tests et les déploiements de Flask et Django, ainsi que pour le frontend. Les ressources seront provisionnées à l'aide de Terraform, incluant PostgreSQL RDS sur AWS et BigQuery sur GCP. Le provisionnement inclura aussi une instance Kestra pré configurée pour orchestrer les workflows de données. Enfin un VPC sera également configuré avec Terraform pour améliorer la sécurité et l'isolation des déploiements Zappa et des bases de données RDS dans l'écosystème AWS. Tout ceci avec des rôles et identifiants créés de manière dynamique et sécurisée dans HCP Vault au lancement de la configuration.

6. Gestion de Projet

a. Mon Profil

J'ai intégré la formation et le projet avec un profil Data Analyst avec des compétences en SQL et Python orienté data. Au fil de l'année d'alternance, j'ai progressivement évolué vers un rôle de développeur full stack, acquérant des compétences supplémentaires en développement frontend et backend. Cette évolution m'a permis de gérer le projet de manière plus holistique, intégrant des aspects techniques variés tout en développant une expertise approfondie en data engineering.

b. Définition des Rôles

En tant que développeur solo, j'ai pris en charge l'ensemble des responsabilités du projet, allant de la conception initiale à l'implémentation et au déploiement des fonctionnalités. Pour cette raison, j'ai parfois choisi d'utiliser des outils robustes et plus ou moins opiniâtres tels que Django, dbt, React, Shadcn/UI ou Zappa. Cette sélection

d'outils m'a permis de suivre des pratiques éprouvées tout en optimisant le temps de développement.

c. Outils et Méthodes de Gestion de Projets

Inspiré par les pratiques de gestion de projet agiles appliquées dans mon entreprise d'accueil, j'ai adapté une méthodologie agile au rythme particulier de l'alternance. Bien que la cadence des sprints et la gestion des tâches aient dû être ajustées en fonction des périodes de cours et de travail, les principes agiles de flexibilité et de priorisation des tâches ont guidé l'organisation du projet. Cette approche a permis de maintenir un suivi régulier, d'adapter les priorités en fonction des besoins du projet et de garantir un développement cohérent malgré les interruptions.

Pour la gestion de projet, j'ai utilisé GitHub Projects, qui m'a permis de planifier et de suivre l'avancement des tâches de manière efficace. Cet outil a facilité la gestion des sprints, la priorisation des tâches et le suivi de la charge de travail.

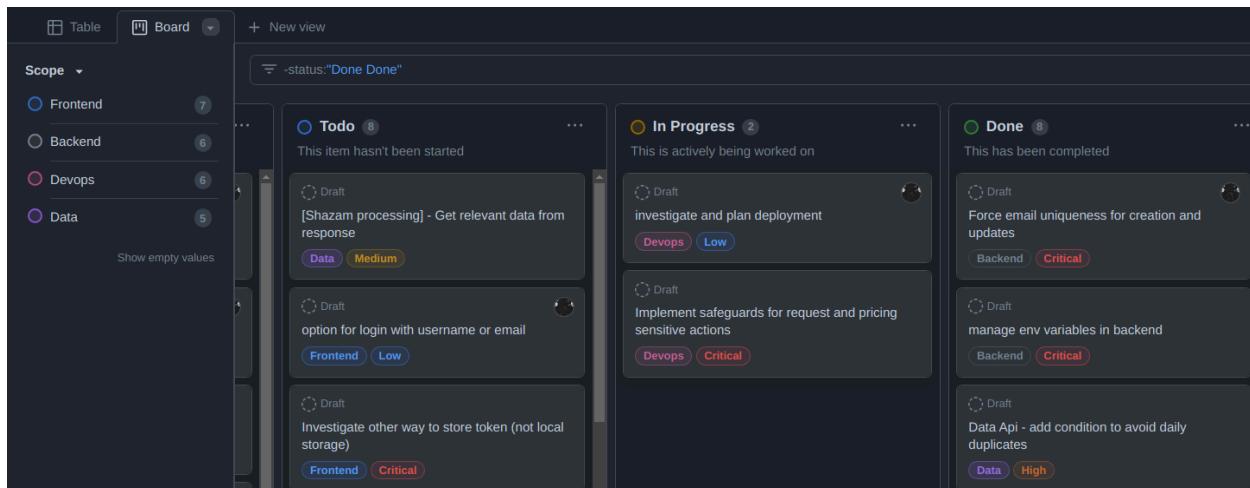


Fig. 19. Interface Kanban sur GitHub Projects | Capture d'écran du site github.com.

7. Réalisations présentant la mise en oeuvre des compétences

a. Réalisation 1 :

Compétence principale : Développer des interfaces utilisateur

Réalisation : Fonctionnalité d'ajout d'un clip musical via une URL YouTube

Couche concernée : Frontend

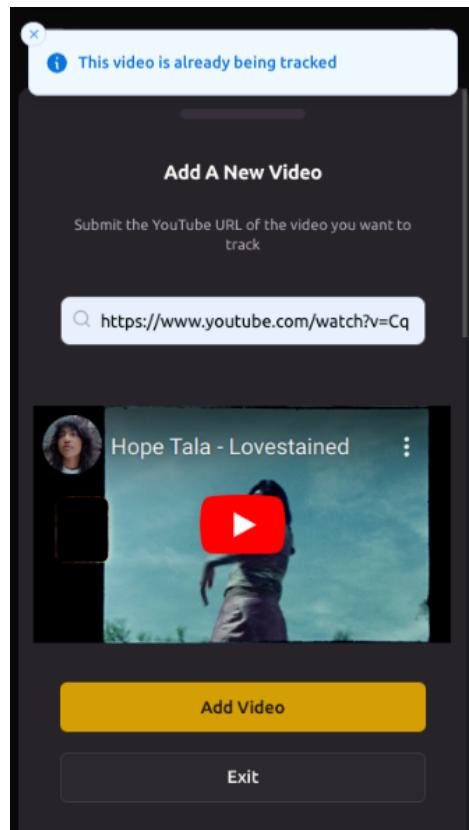


Fig. 20. Interface utilisateur du menu d'ajout d'une vidéo sur mobile, avec exemple d'un toaster d'information "This video is already being tracked" | Capture d'écran du site web.

La fonctionnalité centrale du projet permet aux utilisateurs d'ajouter des vidéos YouTube à une base de données, en les associant à leur compte utilisateur. Côté Frontend, cette fonctionnalité est conçue pour être à la fois intuitive et robuste, en utilisant des outils modernes pour une expérience utilisateur fluide et une gestion efficace des données.

i. Conception et Interface Utilisateur

Pour créer une interface utilisateur à la fois légère et personnalisable, j'ai utilisé les composants Shadcn/ui intégrés avec Tailwind CSS. Voici les avantages et la manière dont ces technologies sont employées :

- Légèreté et Performance : Les composants de Shadcn/ui sont conçus pour être performants et légers, ce qui aide à réduire la taille du bundle JavaScript. Combiné avec Tailwind CSS, cela permet une personnalisation facile tout en maintenant des temps de chargement rapides.

Installation

CLI Manual

- 1 Run the following command:

```
npx shadcn-ui@latest add toast
```



- 2 Add the Toaster component

app/layout.tsx

```
import { Toaster } from "@/components/ui/toaster"

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <head />
      <body>
        <main>{children}</main>
        <Toaster />
      </body>
    </html>
  )
}
```

Fig. 21. Instruction d'installation et d'utilisation d'un composant Shadcn/ui | Capture d'écran du site Shadcn/ui.

- Personnalisation et Thématisation : Grâce à la compatibilité avec Tailwind, j'ai pu ajuster rapidement les styles et les thèmes des composants en utilisant les classes utilitaires de Tailwind. Cela permet de créer une interface cohérente et adaptée aux besoins spécifiques du projet.

```

120  @layer base {
121    :root [|
122      --background: 276 6% 17%;
123      --foreground: 60 20% 98%;
124      --muted: 276 12% 21%;
125      --muted-foreground: 276 12% 71%;
126      --popover: 276 6% 14%;
127      --popover-foreground: 0 0% 100%;
128      --card: 276 6% 15%;
129      --card-foreground: 0 0% 100%;
130      --border: 0 0% 22%;
131      --input: 0 0% 25%;
132      --primary: 45 100% 45%;
133      --primary-foreground: 45 100% 5%;
134      --secondary: 45 30% 25%;
135      --secondary-foreground: 45 30% 85%;
136      --accent: 276 6% 32%;
137      --accent-foreground: 276 6% 92%;
138      --destructive: 10 100% 55%;
139      --destructive-foreground: 0 0% 100%;
140      --ring: 45 100% 45%;
141      --radius: 0.5rem;
142      --chart-1: 220 70% 50%;
143      --chart-2: 160 60% 45%;
144      --chart-3: 30 80% 55%;
145      --chart-4: 280 65% 60%;
146      --chart-5: 340 75% 55%;|
147    ]

```

Fig. 22. Exemple d'un thème de couleur utilisable avec Tailwind | Capture d'écran depuis un script du projet frontend.

- Conformité aux Meilleures Pratiques : Shadcn/ui suit des pratiques robustes en matière de conception d'interface utilisateur, garantissant une expérience utilisateur intuitive et accessible. Les composants sont conçus pour une intégration facile avec React et une personnalisation flexible.

ii. Gestion de l'État et des Effets avec React

Dans le cadre de cette fonctionnalité, j'ai utilisé des composants fonctionnels React et des hooks pour gérer l'état et les effets :

- Composants Fonctionnels : Les composants tels que DrawerComponent et Addsource sont des fonctions qui acceptent des props et retournent des éléments React. Cette approche permet de maintenir le code simple et déclaratif.

```

11  const HomePage = () => [
12
13  !const [sources, setSources, setSourcesUpdateNeeded] = useTrackedSources();
14  const [userSources, setUserSources, setUserSourcesUpdateNeeded] = useUserSources();
15
16  return (
17    <>
18      <Navbar>
19        <Dashboard2>
20          <DataTableDemo2
21            sources={sources}
22            userSources={userSources}
23            setSourcesUpdateNeeded={setSourcesUpdateNeeded}
24            setUserSourcesUpdateNeeded={setUserSourcesUpdateNeeded}
25          >
26            <DrawerComponent>
27              <Addsource
28                sources={sources}
29                setSourcesUpdateNeeded={setSourcesUpdateNeeded}
30                setUserSourcesUpdateNeeded={setUserSourcesUpdateNeeded}
31              />
32            </DrawerComponent>
33            </DataTableDemo2>
34          </Dashboard2>
35        </Navbar>
36      </>
37    )
38 ]
39
40 export default HomePage

```

Fig. 23. Le composant HomePage qui héberge la fonctionnalité | Capture d'écran depuis un script du projet frontend.

- Hooks : J'ai utilisé des hooks comme *useState* pour gérer les états locaux, *useEffect* pour synchroniser les effets de bord, et *useContext* pour partager des données globales comme les jetons d'authentification. Par exemple, le composant *Addsource* utilise *useState* pour gérer les valeurs des entrées utilisateur et les états de validation.
- Gestion des Formulaires : Le formulaire de saisie de l'URL utilise *useState* pour suivre les entrées de l'utilisateur et *handleSetUrl* pour valider et nettoyer l'URL avant de la soumettre.

```

78 // Function to set the YouTube URL
79 const handleSetUrl = async () => {
80     let cleanedUrl = cleanUrl(inputValue);
81
82     if (!cleanedUrl) {
83         // Check if the input URL is a shortened URL and resolve it
84         const finalUrl = await getFinalUrl(inputValue);
85         if (finalUrl) {
86             cleanedUrl = cleanUrl(finalUrl);
87         }
88     }
89
90     if (cleanedUrl) {
91         setYoutubeUrl(cleanedUrl);
92         setIsValidUrl(true);
93     } else {
94         setIsValidUrl(false);
95     }
96 };

```

Fig. 24. Fonction `handleSetUrl` | Capture d'écran depuis un script du projet frontend.

iii. Gestion des Requêtes HTTP avec Axios

Pour interagir avec le backend, nous utilisons Axios, une bibliothèque promise-based pour les requêtes HTTP. Voici comment Axios est intégré dans la gestion des requêtes POST pour l'ajout de vidéos :

- Création d'une Instance Axios : Nous créons une instance Axios (`axiosInstance`) avec une URL de base et des en-têtes d'autorisation. Cette instance est configurée pour inclure le jeton d'authentification dans chaque requête, ce qui facilite la communication sécurisée avec l'API.

```

39 import axios from 'axios';
40 import jwt_decode from 'jwt-decode';
41 import dayjs from 'dayjs';
42 import { DJ_BASE_URL } from '../config';
43
44 const baseURL = DJ_BASE_URL;
45
46 let authTokens = localStorage.getItem('authTokens') ? JSON.parse(localStorage.getItem('authTokens')) : null;
47 let isRefreshing = false;
48 let refreshSubscribers = [];
49
50 const axiosInstance = axios.create({
51     baseURL,
52     headers: { Authorization: `Bearer ${authTokens?.access}` },
53 });

```

Fig. 25. Script Axios/Instance | Capture d'écran depuis un script du projet frontend.

- Intercepteurs de Requêtes : Des intercepteurs sont utilisés pour gérer les requêtes avant leur envoi :
 - Vérification de la validité du jeton : Avant chaque requête, un intercepteur vérifie si le jeton d'accès est expiré. Si c'est le cas, il tente de rafraîchir le jeton en envoyant une requête à l'API de rafraîchissement. Cela garantit que chaque requête est envoyée avec un jeton valide, minimisant les interruptions dues à l'expiration des jetons.
- Intercepteurs de Réponses : Un autre intercepteur gère les réponses des requêtes :
 - Gestion des Erreurs 401 : Si une réponse 401 (non autorisé) est reçue, l'intercepteur tente de rafraîchir le jeton et de réessayer la requête originale. Cela permet de gérer les erreurs d'authentification sans nécessiter une intervention manuelle de l'utilisateur.
- Gestion des requêtes POST : Lorsqu'une vidéo est ajoutée, une requête POST est envoyée à l'API avec l'URL de la vidéo YouTube. L'URL est d'abord nettoyée et validée avant d'être envoyée. Axios gère la soumission de cette requête et fournit des retours instantanés à l'utilisateur via des messages de toast, informant de la réussite ou de l'échec de l'ajout.

```

119
120     // Use toast.promise to handle the promise
121     toast.promise(
122       api.post('/api/usersources/', { source_url: url }, {
123         headers: {
124           'Content-Type': 'application/json',
125         },
126       }),
127       {
128         loading: 'Loading...',
129         success: (response) => {
130           const newSource = {
131             id: response.data.id,
132             url: url,
133           };
134           setSourcesUpdateNeeded(true);
135           setUserSourcesUpdateNeeded(true);
136           setIsSubmitted(true);
137           return `The video has been added`;
138         },
139         error: 'Error adding source',
140       }
141     );
142   };

```

Fig. 26. Utilisation de `toast.promise` | Capture d'écran depuis un script du projet frontend.

iv. Conclusion

En utilisant Shadcn/ui avec Tailwind CSS, j'ai pu créer des interfaces utilisateur légères, personnalisables et performantes, tout en suivant les meilleures pratiques en matière de conception. React facilite la gestion de l'état et des effets, rendant le code plus simple et maintenable. Axios, avec ses intercepteurs et sa gestion des requêtes, assure une communication fluide avec le backend, tout en gérant les jetons d'authentification de manière efficace. Cette approche intégrée garantit une fonctionnalité d'ajout de vidéos YouTube robuste, réactive et facile à utiliser pour les utilisateurs.

b. Réalisation 2

Compétence principale : Développer des composants métier d'une application

Réalisation : Inscription utilisateur et validation par email

Couche concernée : Backend

Le processus d'authentification des utilisateurs et de validation par email est essentiel pour assurer la sécurité et la vérification des comptes du backend Django. Nous allons examiner en détail le mécanisme mis en place dans le cadre de ce projet, du concept théorique aux détails de mise en œuvre.

i. Concepts Théoriques

1. Authentification des Utilisateurs

L'authentification des utilisateurs est le processus de vérification de l'identité d'un utilisateur. Dans Django, cela est généralement géré par le modèle *User* et les vues associées. Une fois qu'un utilisateur est authentifié, il peut accéder aux ressources protégées de l'application.

2. Validation par Email

La validation par email est une méthode utilisée pour vérifier l'adresse email d'un utilisateur après l'inscription. Cela permet de s'assurer que l'email fourni est valide et appartient réellement à l'utilisateur qui souhaite s'inscrire.

ii. Mise en Oeuvre Technique

1. Configuration de Django

Fichiers de Configuration (*settings.py*)

- Authentification JWT : La configuration de Django utilise rest_framework_simplejwt pour gérer les tokens JWT (JSON Web Tokens). Les tokens JWT sont utilisés pour authentifier les utilisateurs et leur permettre d'accéder aux API sécurisées.

```

249 REST_FRAMEWORK = {
250     'DEFAULT_AUTHENTICATION_CLASSES': (
251         'rest_framework_simplejwt.authentication.JWTAuthentication',
252     )
253 }
```

Fig. 27. Configuration Django Rest Framework | Capture d'écran depuis un script du projet backend.

- Paramètres JWT : Les paramètres de SIMPLE_JWT définissent la durée de vie des tokens, la rotation des tokens et les serializers utilisés pour gérer les tokens JWT.

```

238 SIMPLE_JWT = [
239     "ACCESS_TOKEN_LIFETIME": timedelta(hours=2),
240     "REFRESH_TOKEN_LIFETIME": timedelta(days=90),
241     "ROTATE_REFRESH_TOKENS": True,
242     "BLACKLIST_AFTER_ROTATION": True,
243     "AUTH_HEADER_TYPES": ("Bearer",),
244     "AUTH_HEADER_NAME": "HTTP_AUTHORIZATION",
245     "TOKEN_OBTAIN_SERIALIZER": "base.api.views.MyTokenObtainPairSerializer",
246 ]
```

Fig. 28. Configuration JWT | Capture d'écran depuis un script du projet backend.

- Paramètres d'Email : Configurations SMTP pour l'envoi d'emails, utilisés pour l'envoi des liens de validation et de réinitialisation de mot de passe.

```

206 # smtp server settings
207 EMAIL_BACKEND = from_vault('EMAIL_BACKEND')
208 EMAIL_HOST = from_vault('EMAIL_HOST')
209 EMAIL_PORT = from_vault('EMAIL_PORT')
210 EMAIL_USE_TLS = from_vault('EMAIL_USE_TLS')
211 EMAIL_HOST_USER = from_vault('EMAIL_HOST_USER')
212 EMAIL_HOST_PASSWORD = from_vault(['EMAIL_HOST_PASSWORD'])
213 DEFAULT_FROM_EMAIL = from_vault('DEFAULT_FROM_EMAIL')
```

Fig. 29. Configuration Serveur Email | Capture d'écran depuis un script du projet backend.

2. Logique d'Authentification et de Validation

- Enregistrement d'un Utilisateur : Lorsqu'un utilisateur s'inscrit, un nouvel utilisateur est créé avec `user.is_active` défini sur `False`, indiquant que le compte n'est pas encore activé. Un email est envoyé avec un lien d'activation contenant un token unique.

```
82 @api_view(['POST'])
83 def register(request):
84     serializer = UserSerializer(data=request.data)
85     if serializer.is_valid():
86         user = serializer.save()
87         user.is_active = False
88         user.save()

89         token = PasswordResetTokenGenerator().make_token(user)
90         uid = urlsafe_base64_encode(force_bytes(user.pk))
91         # activation_link = f'http://localhost:5173/activate/{uid}/{token}'
92         endpoint = f'{uid}/{token}'
93         activation_link = f'{BASE_URL_FRONT}/#/activate/{endpoint}'

94         send_mail(
95             'Activate your account',
96             f'Hello {user.username}, Click the link to activate your account: {activation_link}',
97             'from@example.com',
98             [user.email],
99         )
100    )
101
102    return Response({'detail': 'Verification email has been sent'}, status=status.HTTP_201_CREATED)
103
104    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Fig. 30. Fonction liée à la route `register` | Capture d'écran depuis un script du projet backend.

- Activation du Compte : Lorsque l'utilisateur clique sur le lien d'activation, une requête est envoyée avec `l'uid` et le `token`. Ces informations sont utilisées pour activer le compte.

```

105 @api_view(['GET'])
106 def activate_account(request, uidb64, token):
107     try:
108         uid = force_str(urlssafe_base64_decode(uidb64))
109         user = User.objects.get(pk=uid)
110
111         if not PasswordResetTokenGenerator().check_token(user, token):
112             return Response({'detail': 'Token is invalid'}, status=status.HTTP_400_BAD_REQUEST)
113
114         user.is_active = True
115         user.save()
116
117         return Response({'detail': 'Account activated.'})
118
119     except (TypeError, ValueError, OverflowError, User.DoesNotExist):
120         return Response({'detail': 'Something went wrong.'}, status=status.HTTP_400_BAD_REQUEST)

```

Fig. 31. Fonction liée à la route *activate_account* | Capture d'écran depuis un script du projet backend.

3. Réinitialisation du Mot de Passe

- Demande de Réinitialisation : Lorsqu'un utilisateur demande une réinitialisation de mot de passe, un email contenant un lien de réinitialisation est envoyé. Si l'utilisateur n'a pas activé son compte, un nouveau lien de validation est envoyé.
- Confirmation de Réinitialisation : Lorsqu'un utilisateur clique sur le lien de réinitialisation, une requête est envoyée avec l'*uid* et le *token*. Si le token est valide, le mot de passe de l'utilisateur est réinitialisé.

iii. Conclusion

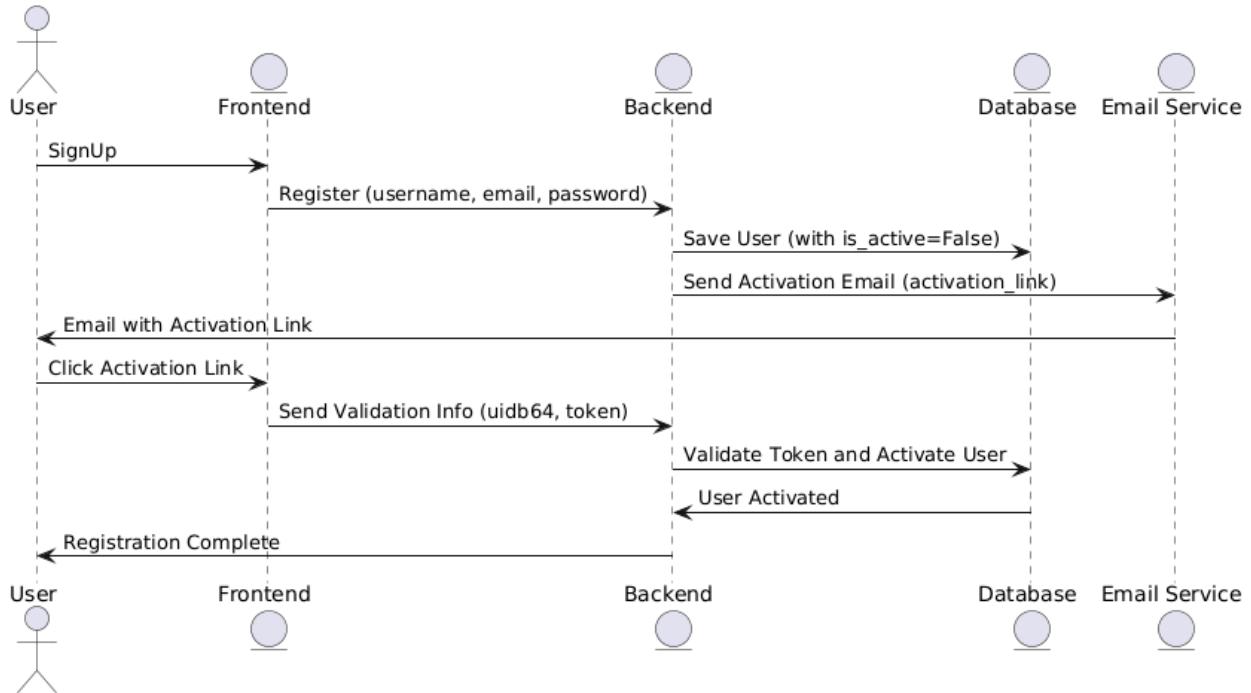


Fig. 32. Diagramme de séquence : Inscription et Validation d'Email | Généré avec l'aide de PlantText UML Editor.

Ce processus d'authentification et de validation par email assure que les utilisateurs sont correctement vérifiés avant de pouvoir accéder pleinement aux fonctionnalités de l'application. L'utilisation de tokens JWT pour l'authentification et des tokens pour la validation par email garantit la sécurité et la fiabilité du système d'authentification. Les détails fournis dans ce guide montrent comment Django et ses extensions facilitent la gestion des utilisateurs et la sécurisation des opérations critiques.

c. Réalisation 3

Compétence principale : Développer des composants d'accès aux données

Réalisation : Fonctionnalité Extract & Load

Couche Concernée : API Analytique

L'API de traitement des données joue un rôle crucial dans la gestion du cycle de vie des données entre les bases de données PostgreSQL et BigQuery. La fonctionnalité clé de l'API comprend l'extraction et le chargement des données depuis PostgreSQL vers BigQuery, ainsi que l'enrichissement et la mise à jour des données avec des sources externes comme YouTube. Voici un aperçu détaillé des processus d'extraction et de chargement des données.

i. Structure de l'API de Traitement des Données

L'API de traitement des données est construite en utilisant Flask et gère diverses tâches liées à l'extraction et au chargement des données. La fonctionnalité principale est divisée en :

- Extraction des Données depuis PostgreSQL
- Chargement des Données dans BigQuery
- Enrichissement des Données à partir des APIs Externes

Pour cette explication, nous nous concentrerons sur les parties d'extraction et de chargement des données.

1. Extraction des Données depuis PostgreSQL

Le processus commence par l'extraction des données de la base de données PostgreSQL. Cela est accompli dans le module `data_loader.py` via la fonction `load_pg_table`.

- Fonction : `load_pg_table`

But : Extrait les données d'une table PostgreSQL spécifiée et les retourne sous forme de DataFrame.

Processus :

- Établit une connexion à la base de données PostgreSQL en utilisant SQLAlchemy (fonction *create_db_engine*).

```

114     from decouple import config
115     from sqlalchemy import create_engine
116
117     def create_db_engine():
118         db_uri = config('PG_URI')
119         engine = create_engine(db_uri)
120
121     return engine

```

Fig. 33. Fonction *create_db_engine* | Capture d'écran depuis un script du projet API analytique.

- Exécute une requête SQL pour sélectionner toutes les données de la table donnée.
- Lit le résultat de la requête dans un DataFrame à l'aide de *pandas*.

```

1  import pandas as pd
2  import logging
3  from google.cloud import bigquery
4  from modules.db_utils import create_bq_client, create_db_engine
5
6  # Set up logging
7  logging.basicConfig(level=logging.INFO)
8  logger = logging.getLogger(__name__)
9
10 def load_pg_table(table_name):
11     try:
12         engine = create_db_engine()
13         query = f'SELECT * FROM "{table_name}"'
14         df = pd.read_sql_query(query, con=engine)
15         return df
16     except Exception as e:
17         logger.error(f"Error loading data from PostgreSQL: {e}")
18         raise

```

Fig. 34. Fonction *load_pg_table* | Capture d'écran depuis un script du projet API analytique.

Bibliothèques utilisées :

- SQLAlchemy : Gère la connexion à PostgreSQL et l'exécution des requêtes SQL.

- Pandas : Utilisé pour lire les résultats de la requête SQL et les convertir en DataFrame.

Modularisation : La connexion à la base de données est encapsulée dans la fonction `create_db_engine`, ce qui permet de réutiliser cette connexion dans d'autres parties du code sans duplication.

Sécurité : Les requêtes SQL sont paramétrées pour éviter les injections SQL. Les exceptions sont capturées et enregistrées pour faciliter le diagnostic tout en évitant l'exposition des détails techniques aux utilisateurs finaux.

2. Chargement des Données dans BigQuery

Une fois les données extraites, elles doivent être chargées dans BigQuery. Cela est géré par les fonctions `load_bigquery_table` et `load_pg_to_bq` dans `data_loader.py`.

- Fonction : `load_bigquery_table`

But : Charger un DataFrame dans une table BigQuery spécifiée.

Processus :

- Crée un client BigQuery.
- Configure le schéma et la disposition d'écriture (comment les données doivent être écrites dans la table) pour le job BigQuery.
- Charge le DataFrame dans la table BigQuery et attend la fin du job.

```

20  def load_bigquery_table(df, table_name, schema, w_disposition):
21      try:
22          bigquery_client = create_bq_client()
23          dataset_id = 'project_project_raw' # Use relevant dataset ID
24          table_ref = bigquery_client.dataset(dataset_id).table(table_name)
25          job_config = bigquery.LoadJobConfig()
26
27          # Specify column types for each table
28          job_config.schema = schema
29
30          job_config.write_disposition = w_disposition # 'WRITE_TRUNCATE' # 'WRITE_APPEND' # Change as needed
31          bigquery_job = bigquery_client.load_table_from_dataframe(df, table_ref, job_config=job_config)
32          bigquery_job.result() # Wait for the job to complete
33          logger.info(f'Data loaded into BigQuery table: {table_name}')
34      except Exception as e:
35          logger.error(f'Error loading data into BigQuery: {e}')
36          raise

```

Fig. 35. Fonction *load_bigquery_table* | Capture d'écran depuis un script du projet API analytique.

- Fonction : `load_pg_to_bq`

But : Faciliter le processus de chargement des données de PostgreSQL vers BigQuery.

Processus :

- Appelle `load_pg_table` pour extraire les données de PostgreSQL.
- Utilise `load_bigquery_table` pour charger les données dans BigQuery.

Bibliothèques utilisées :

- Google Cloud BigQuery : Utilisé pour créer un client BigQuery et charger des données dans BigQuery.
- Pandas : Utilisé pour manipuler les DataFrames avant le chargement.

Modularisation : Les fonctions de chargement de données et de gestion des jobs BigQuery sont séparées pour une meilleure réutilisabilité et une maintenance facilitée. Le code est divisé en modules qui encapsulent des tâches spécifiques.

Sécurité : Les configurations des jobs BigQuery sont sécurisées pour garantir que les données sont traitées de manière appropriée. Les erreurs sont gérées et enregistrées pour une analyse approfondie, évitant les fuites d'informations.

ii. Enrichissement des Données et Transformation

En plus de l'extraction et du chargement internes, l'API enrichit également les données avec des informations provenant de sources externes, spécifiquement YouTube. Cela implique plusieurs étapes :

- Récupération des Données YouTube : Le module `external_apis_service.py` gère les interactions avec l'API YouTube pour récupérer des informations sur les vidéos et les chaînes.
- Enrichissement des Données : Les données récupérées depuis YouTube sont traitées et transformées avant d'être chargées dans BigQuery.
- Fonction : `get_yt_info`

But : Récupérer les données YouTube par lots et les renvoyer sous forme de `DataFrame`.

Processus :

- Créer une session API YouTube.
- Récupérer les données par lots pour éviter de dépasser les limites de l'API.
- Convertir les données en `DataFrame` et ajouter un horodatage.

```

196 def get_yt_info(get_yt_data_func, id_batches):
197     # id_batches = get_video_id_batches()
198     youtube = create_yt_session()
199
200     video_data = []
201
202     for batch in id_batches:
203         video_data.extend(get_yt_data_func(youtube, batch))
204
205     # Create a DataFrame from the video data
206     df_final = pd.DataFrame(video_data)
207
208     # Add a timestamp column to the DataFrame
209     df_final['loaded_at'] = datetime.datetime.now().isoformat()
210
211     print('retrieved youtube info')
212     # print(df_final.head())
213     return df_final

```

Fig. 36. Fonction `get_yt_info` | Capture d'écran depuis un script du projet API analytique.

- Fonction : `yt_snippet` et `yt_stats`

But : Récupérer et traiter les extraits de vidéos YouTube et les statistiques.

Processus :

- Récupérer les IDs des vidéos depuis les URLs YouTube.
- Utiliser l'API pour obtenir les extraits ou les statistiques des vidéos.
- Transformer les données au format approprié et les renvoyer sous forme de DataFrame.

```

33     def get_yt_stats(youtube, video_ids):
34         '''Fetches YouTube statistics data for the given video IDs.'''
35         video_data = []
36
37         yt_data = youtube.videos().list(part=['statistics'], id=video_ids).execute()
38
39         for item in yt_data.get('items', []):
40             stats = item.get('statistics', {})
41             video_data.append({
42                 'video_id': item['id'],
43                 'view_count': stats.get('viewCount', ''),
44                 'like_count': stats.get('likeCount', ''),
45                 'comment_count': stats.get('commentCount', ''),
46             })
47
48     return video_data

```

Fig. 36. Fonction `get_yt_stats` | Capture d'écran depuis un script du projet API analytique.

```

95     def yt_stats():
96
97         id_batches = get_video_id_batches()
98         df = get_yt_info(get_yt_stats, id_batches)
99
100        # handle types
101        df['video_id'] = df['video_id'].astype(str)
102        df['view_count'] = df['view_count'].astype(int)
103        df['like_count'] = df['like_count'].astype(int)
104        df['comment_count'] = df['comment_count'].astype(int)
105        df['loaded_at'] = pd.to_datetime(df['loaded_at'])
106

```

Fig. 37. Fonction `yt_stats` | Capture d'écran depuis un script du projet API analytique.

iii. Contexte des Appels API

- But : Déclencher le processus de chargement des données depuis PostgreSQL vers BigQuery et depuis l'API YouTube.
- Utilisation : Intégrée dans l'orchestrateur Kestra pour automatiser les appels sous forme de workflows conditionnés.

8. Jeu d'essai

Le jeu de test pour la fonctionnalité d'ajout de vidéos YouTube est conçu pour s'assurer que toutes les variantes possibles d'URL sont correctement gérées par l'application. Chaque scénario est testé en suivant une série d'étapes prédefinies pour vérifier que l'application réagit de manière appropriée aux différents types d'entrées fournies par l'utilisateur.

Scénarios de Test :

- Ajouter une URL YouTube valide, format classique
- Ajouter une URL YouTube valide, format raccourci
- Ajouter une URL YouTube valide, format mobile
- Ajouter une URL YouTube invalide
- Ajouter autre chose qu'une URL
- Ajouter une URL déjà suivie
- Erreur lors de l'ajout d'une URL valide

Chaque test est conçu pour valider une fonctionnalité spécifique de l'application et garantir que le comportement attendu est conforme aux spécifications. Les résultats sont enregistrés pour chaque test, et des messages de confirmation ou d'erreur sont observés pour évaluer le bon fonctionnement de la fonctionnalité d'ajout de vidéos.

ID	Titre du test	Objectif	Process	Résultats Attendus	Données de test	État du test	Commentaires
1	Ajouter une URL YouTube valide, format classique	Vérifier que l'utilisateur peut ajouter une vidéo YouTube valide à son compte.	Ouvrir le menu d'ajout d'une video; entrer l'URL à tester en input; cliquer sur "Check URL"; cliquer sur "ADD Video".	Success Toaster : "The video has been added" + Cleaned URL displayed in console.	https://www.youtube.com/watch?v=dQw4w9WgXcQ	OK	RAS
2	Ajouter une URL YouTube valide, format raccourci	Vérifier que l'application gère correctement les URL YouTube raccourcies.	Même process.	Success Toaster : "The video has been added" + Cleaned URL displayed in console.	https://youtu.be/dQw4w9WgXcQ	OK	RAS
3	Ajouter une URL YouTube valide, format mobile	Vérifier que l'application gère les URL YouTube mobiles.	Même process.	Success Toaster : "The video has been added" + Cleaned URL displayed in console.	https://m.youtube.com/watch?v=dQw4w9WgXcQ	OK	RAS
4	Ajouter une URL YouTube invalide	Vérifier que l'application affiche un message d'erreur pour une URL invalide.	Même process.	Error Toaster : "Invalid URL"	https://notyoutube.com/video	OK	RAS
5	Ajouter autre chose qu'une URL	Vérifier que l'application affiche un message quand l'input ne correspond pas à une URL.	Même process.	Error Toaster : "Invalid URL"	Not a URL ///	OK	RAS
7	Ajouter une URL déjà suivie	Vérifier que l'application informe l'utilisateur si la vidéo est déjà suivie.	Même process.	Info Toaster : "This video is already being tracked"	https://www.youtube.com/watch?v=[video_id d'une video déjà suivie]	OK	RAS
8	Erreur lors de l'ajout d'une URL valide	Vérifier que l'application informe l'utilisateur qu'une erreur s'est produite.	Même process après avoir déconnecté le Backend du Frontend. (En mettant une Base URL invalide par	Error Toaster : "Error adding source"	https://www.youtube.com/watch?v=dQw4w9WgXcQ	OK	RAS

Fig. 36. Jeu d'Essai de la fonctionnalité d'ajout de vidéo | Capture d'écran depuis un tableau Google Sheets.

9. Veille de sécurité

Dans le cadre de ce projet, j'ai effectué une veille de sécurité approfondie pour identifier les vulnérabilités potentielles et les risques associés à l'architecture en place. Cette veille a conduit à la détection de plusieurs points d'attention critiques, chacun nécessitant des mesures spécifiques pour garantir la sécurité et l'intégrité de l'application.

a. Sécurité du Frontend : Stockage des Tokens en Local Storage

Actuellement, l'application stocke les tokens d'authentification dans le *localStorage* du navigateur. Bien que cette approche soit simple à implémenter, elle présente des risques importants, notamment en cas de vulnérabilité de type *Cross-Site Scripting (XSS)*. Si un attaquant parvient à injecter du code malveillant dans l'application, il pourrait accéder aux tokens stockés, compromettant ainsi l'intégrité des sessions utilisateurs.

Mesures proposées :

- Utiliser les Cookies sécurisés avec le flag *HttpOnly* : Cette approche permet de stocker les tokens dans des cookies qui ne sont pas accessibles via JavaScript, réduisant ainsi l'exposition aux attaques XSS.
- Implémenter une politique de *Content Security Policy (CSP)* stricte : Pour limiter les sources de scripts exécutables, minimisant ainsi les risques d'exploitation XSS.
- Activer HTTPS sur l'ensemble de l'application : Pour garantir que toutes les communications entre le frontend et le backend sont chiffrées, empêchant ainsi toute interception des tokens.

b. Risques liés à l'Architecture Multi-couches

L'architecture de l'application est organisée en différentes couches indépendantes (frontend, backend, API analytique, base de données). Bien que cela favorise la modularité et la maintenance, cette séparation introduit des risques, principalement liés à la sécurité des communications inter-services.

Vulnérabilités potentielles :

- Interception des communications : Les échanges entre les différentes couches, s'ils ne sont pas correctement sécurisés, peuvent être vulnérables à des attaques de type *Man-in-the-Middle* (MITM).
- Fuite de données sensibles : Si les données échangées entre les services ne sont pas chiffrées, elles peuvent être lues par des acteurs malveillants.

Mesures proposées :

- Chiffrement des communications : Utiliser *TLS* (Transport Layer Security) pour sécuriser les échanges entre les services (APIs, frontend, bases de données).
- Utilisation de certificats *SSL/TLS* validés : Pour garantir l'authenticité des serveurs et prévenir les attaques *MITM*.
- Segmenter les réseaux avec des *VPCs* (Virtual Private Clouds) : Pour isoler les différentes couches et limiter l'accès aux services internes uniquement aux parties autorisées.

c. Sécurisation via AWS et Zappa Serverless

L'utilisation d'*AWS* et de *Zappa* pour déployer des services en mode serverless présente des avantages en termes de scalabilité et de coût. Cependant, cela introduit également des défis de sécurité spécifiques.

Mesures proposées :

- *API Gateway* avec clé API : Configurer les *API Gateway* pour exiger une clé API, limitant ainsi l'accès aux seules requêtes autorisées.
- Définition de *rôles AWS* restreints (*least privilege*) : Assigner aux services des rôles avec les permissions minimales nécessaires, limitant ainsi les dégâts potentiels en cas de compromission.
- Optimisation de la configuration du *VPC* : Utiliser des sous-réseaux privés et des groupes de sécurité pour contrôler strictement l'accès aux services *AWS* et minimiser l'exposition des services critiques.

d. Risques liés au Déploiement Serverless et Gestion des Coûts

Le déploiement de l'API via des *fonctions Lambda* sur *AWS* est une solution économique pour des applications peu sollicitées. Cependant, en cas d'augmentation soudaine du trafic, par exemple lors d'une attaque *DDoS* (Distributed Denial of Service), cela pourrait entraîner des coûts significatifs.

Mesures proposées :

- Mise en place de limites de facturation : Configurer des alertes et des plafonds de facturation pour éviter des dépenses imprévues en cas de pic de requêtes.
- Utilisation de *WAF* (Web Application Firewall) : Pour filtrer et bloquer les requêtes malveillantes avant qu'elles n'atteignent les fonctions Lambda.
- Activation de la protection contre les *DDoS* avec *AWS Shield* : Pour atténuer les attaques visant à submerger les services et augmenter les coûts.

10. Annexes

a. Sources

Vite : <https://vitejs.dev/>

React : <https://fr.react.dev/>

Tailwind : <https://tailwindcss.com/>

Shadcn/ui : <https://ui.shadcn.com/>; <https://ui.jln.dev/>

Django : <https://www.djangoproject.com/>

Django REST Framework (DRF) : <https://www.django-rest-framework.org/>

PostgreSQL : <https://www.postgresql.org/>

BigQuery : <https://cloud.google.com/bigquery?hl=fr>

Flask : <https://flask.palletsprojects.com/en/3.0.x/>

dbt Cloud : <https://www.getdbt.com/product/dbt-cloud>

Kestra : <https://kestra.io/>

GitHub : <https://github.com/>

Zappa : <https://github.com/zappa/Zappa>

Terraform : <https://www.terraform.io/>

HCP Vault : <https://developer.hashicorp.com/hcp/docs/vault/what-is-hcp-vault>

Docker : <https://www.docker.com/>

VS Code Dev Containers : <https://code.visualstudio.com/docs/devcontainers/containers>

GitHub Pages : <https://pages.github.com/>

Aiven : <https://aiven.io/>

PlantText UML Editor : <https://www.planttext.com/>

Whimsical : <https://whimsical.com/>

ChatGPT : <https://openai.com/chatgpt/>

b. Illustrations

The screenshot shows a dark-themed application window titled 'Analytics > Tracked Videos'. At the top, there are navigation icons for back, forward, and search, along with a user profile icon. Below the header, a section titled 'Tracked Videos' with the sub-instruction 'Manage your tracked videos and add new ones' is visible. A search bar labeled 'Filter titles...' is positioned above a table. The table has columns for 'Author Name', 'Title', and 'Date added'. The data is sorted by date added in descending order (indicated by an upward arrow). Six video entries are listed:

Author Name	Title	Date added ↑
Tems	Tems - Gangsta (Official Visualizer)	2024-06-13
Snoh Aalegra	Snoh Aalegra - DYING 4 YOUR LOVE	2024-06-14
Cleo Sol	Cleo Sol - Butterfly (Official Video)	2024-06-14
Hope Tala	Hope Tala - Lovestained	2024-06-14
HojeanVEVO	Hojean - Hold Me	2024-07-23
Samaria	Samaria - Out the Way [Official Video]	2024-07-23

At the bottom right of the table area, there are 'Columns' and 'Add Video' buttons.

Annexe 1 : Page principale de l'application format Desktop | Capture d'écran du site web.

The screenshot shows a dark-themed application window titled 'Tracked Videos'. At the top, there are navigation icons for back, forward, and search, along with a user profile icon. Below the header, a section titled 'Tracked Videos' with the sub-instruction 'Manage your tracked videos and add new ones' is visible. A search bar labeled 'Filter titles...' is positioned above a table. The table has a single column for 'Title'. Five video entries are listed:

Title
Mustafa - Name of God
J.Tajor - Like I Do (Official Music Video)
Cruel Santino - Rapid Fire (Official Video) feat Shane Eagle, Tomi Agape & Amaaræ
RUSSELLI - RITA'S FREESTYLE (Official Music Video)

Annexe 2 : Page principale de l'application format Mobile | Capture d'écran du site web.

Tracked Videos

Manage your tracked videos and add new ones

Filter titles...

Author Name Title Date added

Author Name	Title	Date added
Mustafa	Mustafa - Name of God	2024-08-07
JTajorVEVO	J.Tajor - Like I Do (Official Music Video)	2024-08-05
Cruel Santino	Cruel Santino - Rapid Fire (Official Video) feat Shane Eagle, Tomi Agape & Amaarae	2024-08-04
RUSSELLI	RUSSELLI - RITA'S FREESTYLE (Official Music Video)	2024-07-31
Daniela Andrade	Daniela Andrade - Tamale (Official Video)	2024-07-25
Hojean	Hojean - Hold Me	2024-07-23

Add A New Video

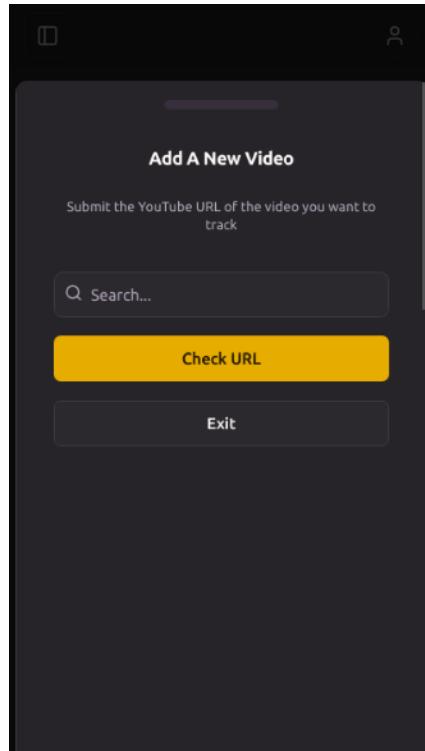
Submit the YouTube URL of the video you want to track

Search...

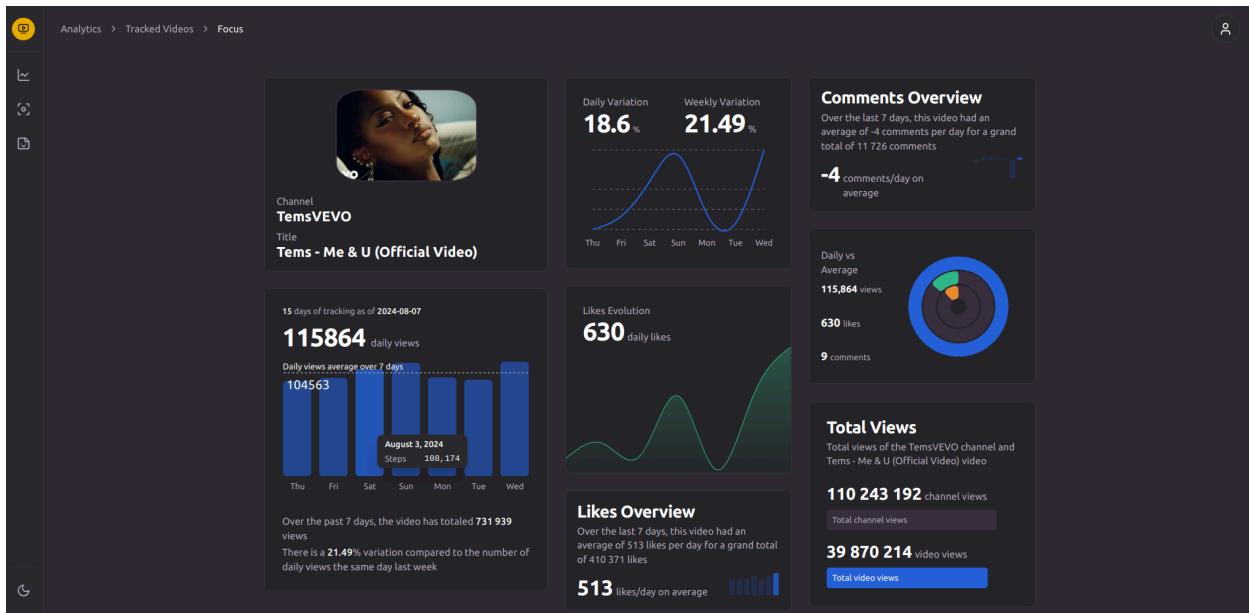
Check URL

Exit

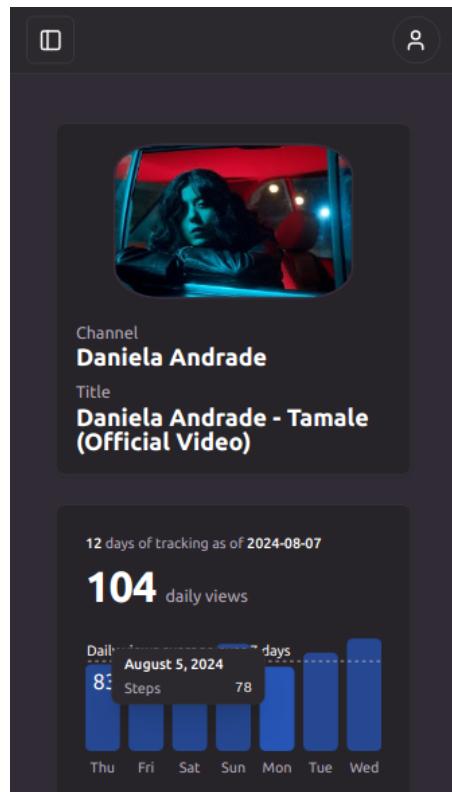
Annexe 3 : Menu d'ajout d'une vidéo format Desktop | Capture d'écran du site web.



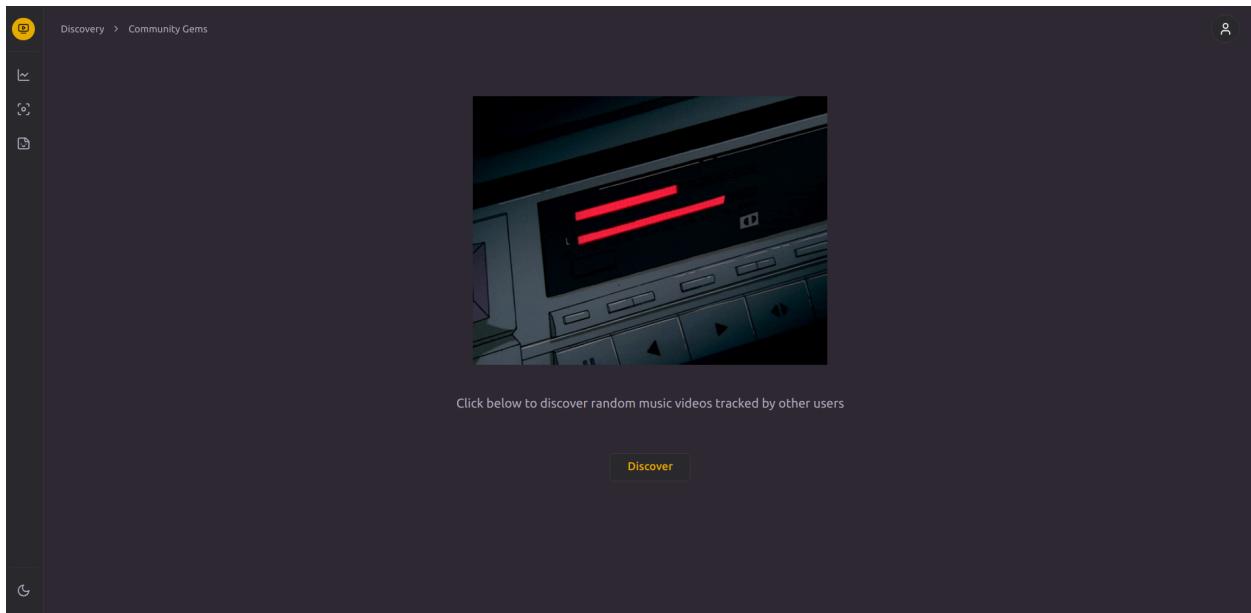
Annexe 4 : Menu d'ajout d'une vidéo format Mobile | Capture d'écran du site web.



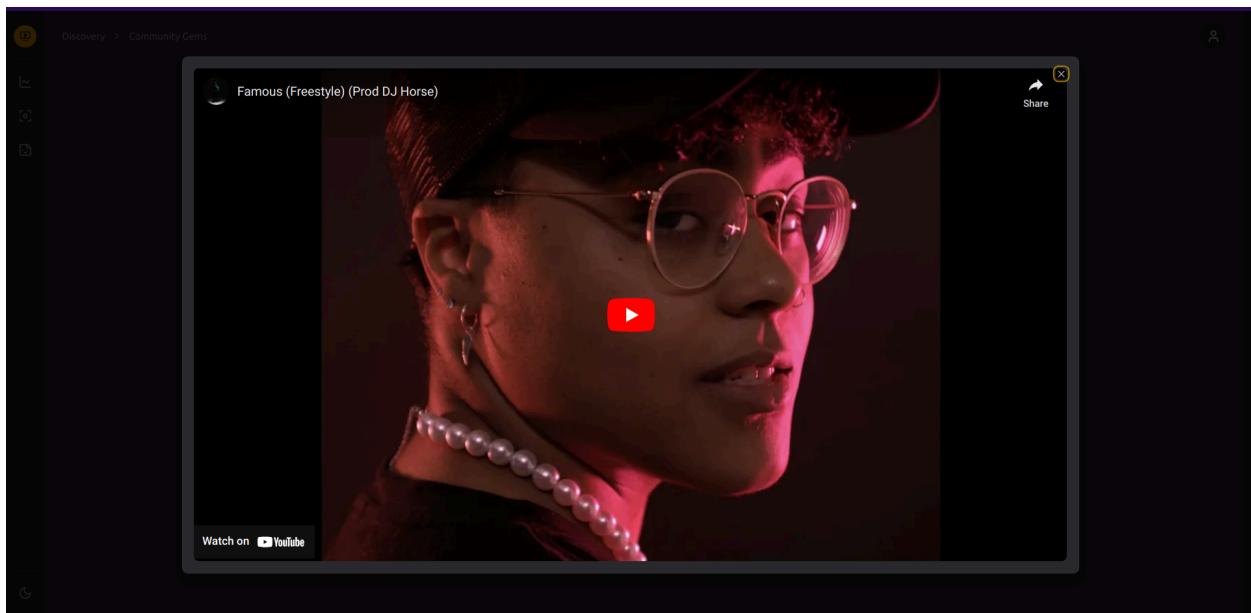
Annexe 5 : Page Focus format Desktop | Capture d'écran du site web.



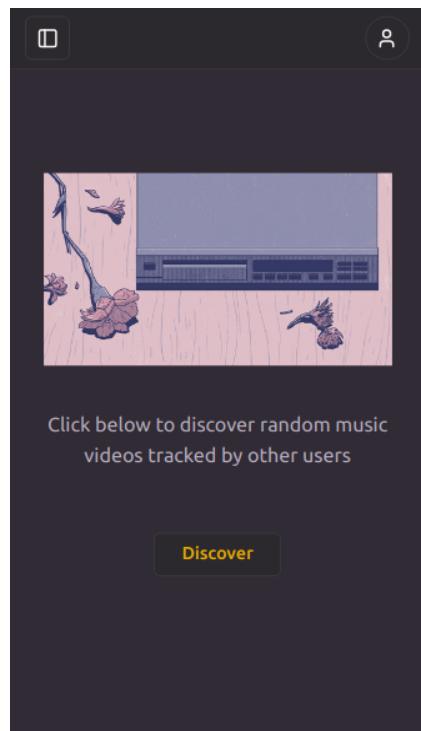
Annexe 6 : Page Focus format Desktop | Capture d'écran du site web.



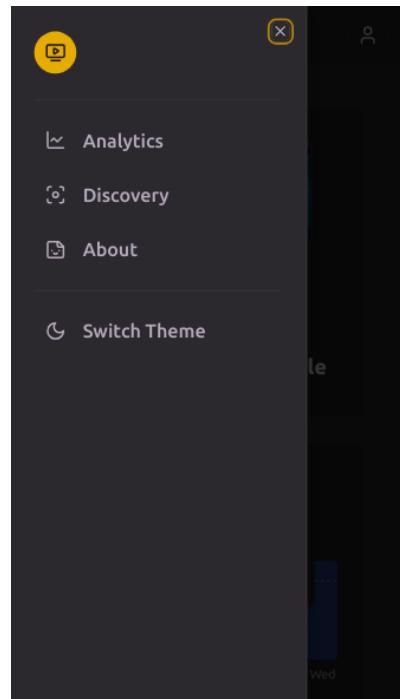
Annexe 7 : Page Discovery format Desktop | Capture d'écran du site web.



Annexe 8 : Page Discovery format lecteur Desktop | Capture d'écran du site web.



Annexe 9 : Page Discovery format Mobile | Capture d'écran du site web.



Annexe 10 : Barre de navigation format Mobile | Capture d'écran du site web.

The screenshot shows the dbt Cloud interface. On the left, there's a sidebar with navigation links: Dashboard, Develop (selected), Deploy, Explore, Documentation, Classic navigation, Leave feedback, Help & Guides, and Selfdata. The main area has tabs for 'develop' and 'Change branch'. Under 'Version control', there's a button to 'Create a pull request on GitHub'. The 'File explorer' sidebar lists files like dbt-project-project, analyses, dbt_packages, macros, models (including datawarehouse, intermediate, and models subfolders), and sources.yml. The central workspace shows a code editor with a SQL script for 'int_youtube_stats.sql'. The script includes imports from 'video_stats' and 'channel_stats', joins between them, and a final select clause with date range aliases. Below the code editor is a toolbar with buttons for Preview, Compile, Build, Format, Results, Code quality, Compiled code, and Lineage. The Lineage tab is selected, displaying a dependency graph. The graph shows three raw source nodes: 'project_project_raw_raw_yt_stats', 'project_project_raw_raw_youtube_ch...', and 'project_project_raw_raw_yt_snippet'. These nodes feed into three intermediate nodes: 'stg_youtube_video_stats', 'stg_youtube_channel_stats', and 'stg_youtube_video_snippet'. These three intermediate nodes all point to a single 'int_youtube_stats' node. From 'int_youtube_stats', arrows point to two more intermediate nodes: 'int_youtube_variation' and 'fact_youtube_stats'. Finally, 'int_youtube_variation' and 'fact_youtube_stats' both point to a single '2+int_youtube_stats+2' node.

```

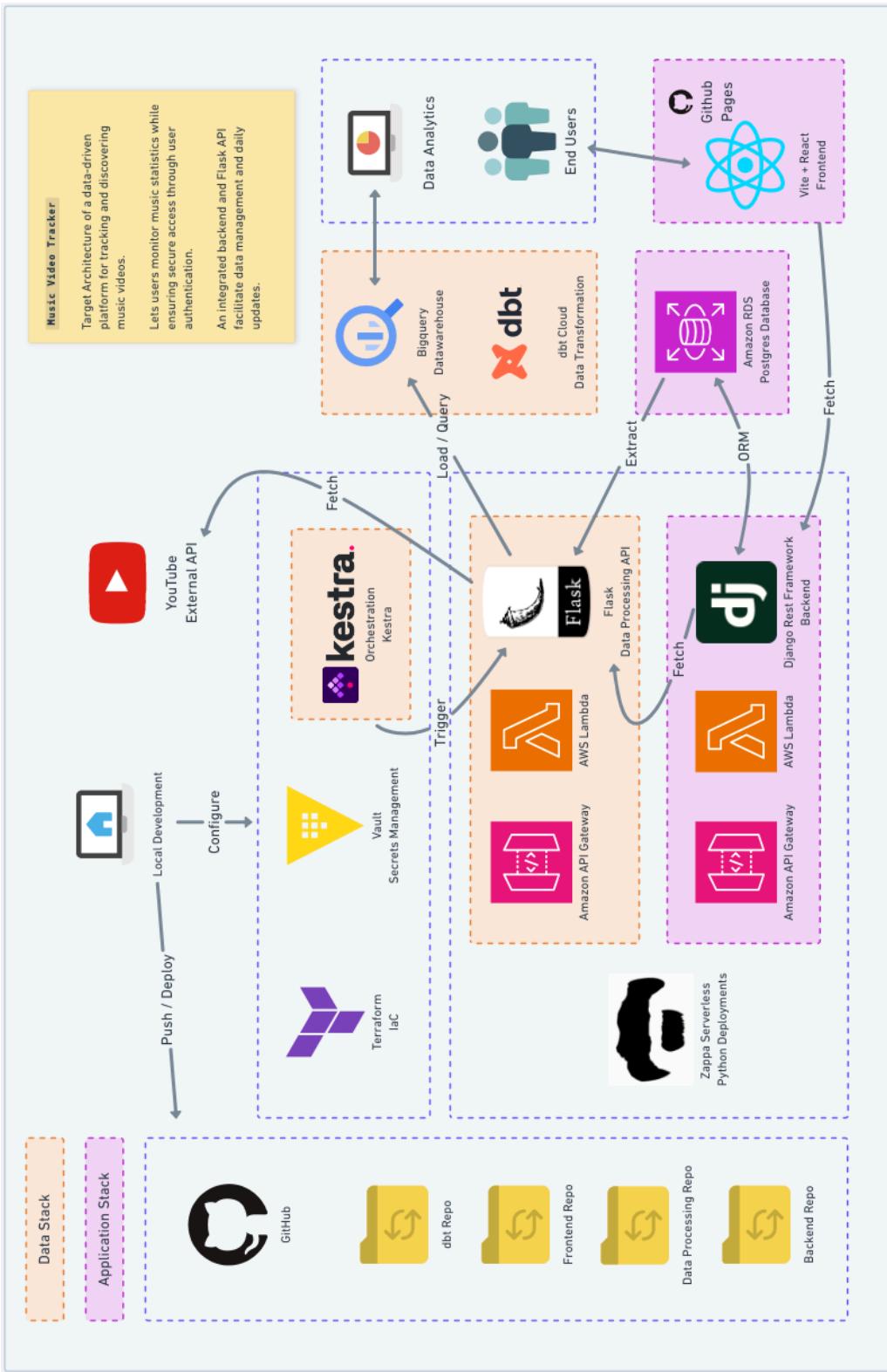
models : intermediate > int_youtube_stats.sql
  6
  7   , youtube_stats as (
  8     select
  9       vs.youtube_video_id,
 10      cs.youtube_channel_id,
 11      vs.youtube_video_stats_ref_day as ref_day,
 12      vs.youtube_video_view_count,
 13      vs.youtube_video_like_count,
 14      vs.youtube_video_comment_count,
 15      cs.youtube_channel_view_count,
 16      cs.youtube_channel_video_count
 17   from video_stats vs
 18   inner join video_snippet sn on vs.youtube_video_id = sn.youtube_video_id
 19   inner join channel_stats cs
 20     on sn.youtube_video_channel_id = cs.youtube_channel_id
 21     and vs.youtube_video_stats_ref_day = cs.youtube_channel_stats_ref_day
 22 )
 23
 24   , date_ref as (
 25     select
 26       *
 27     , date_sub(ref_day, interval 1 day) as previous_day,
 28     date_sub(ref_day, interval 7 day) as previous_week,
 29     date_sub(ref_day, interval 1 month) as previous_month,
 30     date_sub(ref_day, interval 1 year) as previous_year
 31   from youtube_stats
 32 )

```

Annexe 11 : IDE en ligne de dbt Cloud avec exemple de transformation sql | Capture d'écran du site dbt Cloud.



Annexe 12 : Interface de monitoring des jobs de transformation dbt Cloud | Capture d'écran du site dbt Cloud.



Annexe 13 : Architecture cible grand format | Crée avec Whimsical.

Flows

Search data.processing Labels Filters

202 Executions

Id	Labels	Namespace	Last execution date	Last status	Execution statistics	Triggers
api_check		data .processing	Sun, Aug 11, 2024 8:29 PM	Success		
api_load_base_source		data .processing	Sun, Aug 11, 2024 8:30 PM	Success		
api_load_base_usersource		data .processing	Sun, Aug 11, 2024 8:30 PM	Success		
api_load_youtube_channel_snippets		data .processing	Sun, Aug 11, 2024 8:30 PM	Success		
api_load_youtube_channel_stats		data .processing	Sun, Aug 11, 2024 8:30 PM	Success		
api_load_youtube_snippets		data .processing	Sun, Aug 11, 2024 8:30 PM	Success		
api_load_youtube_stats		data .processing	Sun, Aug 11, 2024 8:30 PM	Success		

Total: 7

Namespaces

Blueprints

Plugins

Administration

Settings

0.17/13

25 per page

Total: 7

Annexe 14 : Interface utilisateur de l'orchestrator Kestra | Capture d'écran de l'UI.

```

kestra_flows > ! api_load_base_source.yml
 1   id: api_load_base_source
 2   namespace: data.processing
 3
 4   tasks:
 5     - id: api
 6       type: io.kestra.plugin.core.http.Request
 7       uri: "{{ globals['api-uri'] }}/load_data/base_source"
 8       method: "GET"
 9       headers:
10         Authorization: "{{ globals['api-key'] }}"
11
12     - id: check_status
13       type: io.kestra.plugin.core.flow.If
14       condition: "{{ outputs.api.code != 200 }}"
15       then:
16         - id: unhealthy
17           type: io.kestra.plugin.core.log.Log
18           message: Server unhealthy!!! Response {{ outputs.api.body }}
19       else:
20         - id: healthy
21           type: io.kestra.plugin.core.log.Log
22           message: Everything is fine! Response {{ outputs.api.body }}
23
24   triggers:
25     - id: listenFlow
26       type: io.kestra.plugin.core.trigger.Flow
27       conditions:
28         - type: io.kestra.plugin.core.condition.ExecutionFlowCondition
29           namespace: data.processing
30           flowId: api_check
31         - type: io.kestra.plugin.core.condition.ExecutionStatusCondition
32           in:
33             - SUCCESS

```

Annexe 15 : Exemple de workflow Kestra | Capture d'écran d'un script de flow Kestra.

```

base>api>tests> test_views.py > ...
  1 from django.urls import reverse
  2 from rest_framework.test import APITestCase, APIClient
  3 from rest_framework import status
  4 from django.contrib.auth.models import User
  5 from base.models import Source, UserSource
  6 from rest_framework_simplejwt.tokens import RefreshToken
  7
  8 class UserTests(APITestCase):
  9     def setup(self):
 10         self.client = APIClient()
 11         self.user = User.objects.create_user(username='testuser', email='user@example.com', password='testpass123')
 12         self.url = reverse('user-list')
 13
 14     def test_register_user(self):
 15         data = {
 16             'username': 'newuser',
 17             'email': 'newuser@example.com',
 18             'password': 'newpass123'
 19         }
 20         response = self.client.post(reverse('register'), data, format='json')
 21         self.assertEqual(response.status_code, status.HTTP_201_CREATED)
 22
 23     def test_login_user(self):
 24         data = {
 25             'username': 'testuser',
 26             'password': 'testpass123'
 27         }
 28         response = self.client.post(reverse('token_obtain_pair'), data, format='json')
 29         self.assertEqual(response.status_code, status.HTTP_200_OK)
 30
 31     def get_token_for_user(self, user):
 32         refresh = RefreshToken.for_user(user)
 33         return {
 34             'access': str(refresh.access_token),
 35         }

```

Annexe 16 : Exemple de tests sur les vues du backend avec Pytest et Django | Capture d'écran d'un script du backend.

```

=====
===== test session starts =====
=====
platform linux -- Python 3.10.12, pytest-8.3.2, pluggy-1.5.0
django: version: 5.0.2, settings: backend.test_settings (from ini)
rootdir: /home/constant/Documents/project-project/test_zappa/project-project
configfile: pytest.ini
plugins: django-4.8.0
collected 15 items

base/api/tests/test_database.py . [ 6%]
base/api/tests/test_models.py ..... [ 53%]
base/api/tests/test_serializers.py .. [ 66%]
base/api/tests/test_views.py .... [100%]

===== 15 passed in 5.98s =====
=====
```

Annexe 17 : Résultat des tests avec la commande `pytest` côté backend | Capture d'écran du terminal.

```

src > components > __tests__ > PasswordResetRequestForm.test.jsx > ...
1 import React from 'react';
2 import { render, screen, fireEvent } from '@testing-library/react';
3 import '@testing-library/jest-dom';
4 import { HashRouter } from 'react-router-dom';
5 import { PasswordResetRequestForm } from '../LoginFormComponent';
6
7 describe('PasswordResetRequestForm Component', () => {
8   const requestPasswordResetMock = jest.fn();
9
10  beforeEach(() => {
11    render(
12      <HashRouter>
13        <PasswordResetRequestForm requestPasswordReset={requestPasswordResetMock}>
14        </HashRouter>
15    );
16  });
17
18  test('renders PasswordResetRequestForm with fields and button', () => {
19    expect(screen.getByRole('heading', { name: /Request Password Reset/i })).toBeInTheDocument();
20    expect(screen.getByText(/Enter your email below to update your password/i)).toBeInTheDocument();
21    expect(screen.getByLabelText(/Email/i)).toBeInTheDocument();
22    expect(screen.getByRole('button', { name: /Submit/i })).toBeInTheDocument();
23  });
24
25  test('calls requestPasswordReset on form submission', () => {
26    fireEvent.change(screen.getByPlaceholderText(/m@example.com/i), { target: { value: 'test@example.com' } });
27
28    fireEvent.submit(screen.getByRole('button', { name: /Submit/i }));
29
30    expect(requestPasswordResetMock).toHaveBeenCalled();
31  });
32});
```

Annexe 18 : Exemple de tests sur le composant `PasswordResetRequestForm` en utilisant Jest | Capture d'écran d'un script du frontend.

```
RUNS  src/components/_tests_/SignUpForm.test.jsx
RUNS  src/components/_tests_/LoginForm.test.jsx
RUNS  src/components/_tests_/PasswordResetConfirmForm.test.jsx
RUNS  src/components/_tests_/PasswordResetRequestForm.test.jsx

RUNS  src/components/_tests_/SignUpFormOtp.test.jsx
RUNS  src/components/_tests_/SignUpForm.test.jsx
RUNS  src/components/_tests_/LoginForm.test.jsx
RUNS  src/components/_tests_/PasswordResetConfirmForm.test.jsx
PASS   src/components/_tests_/SignUpFormOtp.test.jsx

RUNS  src/components/_tests_/SignUpFormOtp.test.jsx
RUNS  src/components/_tests_/SignUpForm.test.jsx
RUNS  src/components/_tests_/LoginForm.test.jsx
RUNS  src/components/_tests_/PasswordResetConfirmForm.test.jsx
PASS   src/components/_tests_/PasswordResetConfirmForm.test.jsx

RUNS  src/components/_tests_/SignUpFormOtp.test.jsx
RUNS  src/components/_tests_/SignUpForm.test.jsx
RUNS  src/components/_tests_/LoginForm.test.jsx
RUNS  src/components/_tests_/PasswordResetConfirmForm.test.jsx
PASS   src/components/_tests_/LoginForm.test.jsx

RUNS  src/components/_tests_/SignUpFormOtp.test.jsx
RUNS  src/components/_tests_/SignUpForm.test.jsx
RUNS  src/components/_tests_/LoginForm.test.jsx
RUNS  src/components/_tests_/PasswordResetConfirmForm.test.jsx
PASS   src/components/_tests_/SignUpForm.test.jsx

RUNS  src/components/_tests_/SignUpFormOtp.test.jsx
RUNS  src/components/_tests_/SignUpForm.test.jsx
RUNS  src/components/_tests_/LoginForm.test.jsx
RUNS  src/components/_tests_/PasswordResetConfirmForm.test.jsx

Test Suites: 6 passed, 6 total
Tests:       16 passed, 16 total
Snapshots:  0 total
Time:        2.584 s
Ran all test suites.
```

Annexe 19 : Résultat des tests avec la commande *npm test* côté frontend | Capture d'écran du terminal.