

**génère une version mis à jour du guide technique "**[**AnalylitV4.1.md**](http://AnalylitV4.1.md)**"**

Absolument ! Voici la version mise à jour de votre guide technique, AnalyLitV4.1.md, qui intègre les dernières modifications et corrections.

**Guide Technique : AnalyLit V4.1**

Ce document détaille l'architecture technique, les composants clés et les flux de données de la version 4.1 de l'application AnalyLit. Cette version introduit une modularité accrue avec le répertoire utils, de nouvelles fonctionnalités d'import et d'analyse, et des corrections de bugs.

**1. Architecture Générale**

AnalyLit V4.1 est une application web conçue pour automatiser les revues de littérature scientifique. Son architecture repose sur plusieurs composants principaux :

* **Frontend (Single Page Application)** : Une interface utilisateur dynamique construite avec **HTML5**, **CSS3** (style.css), et **JavaScript** (app.js). Elle communique avec le backend via des appels API REST et des WebSockets.
* **Backend (Serveur Flask)** : Le fichier server\_v4\_complete.py agit comme le chef d'orchestre. Il expose une **API RESTful** pour gérer les projets, les données et lancer des tâches. Il gère également les connexions **WebSocket** pour les notifications en temps réel.
* **Workers & File d'attente (Redis & RQ)** : Pour les opérations longues (recherche, analyse IA, import), le serveur délègue le travail à des workers via une file d'attente gérée par **Redis** et la bibliothèque **RQ (Redis Queue)**. La logique de ces tâches est définie dans tasks\_v4\_complete.py.
* **Base de Données (SQLite)** : Un fichier database.db centralise toutes les données persistantes : projets, articles, extractions, profils, etc.
* **Modules Utilitaires (utils/)** : Un nouveau répertoire contenant la logique métier réutilisable, notamment pour l'import de données (importers.py), les analyses (analysis.py), et la génération de rapports (reporting.py).
* **Moteur d'IA (Ollama)** : L'application interagit avec un service Ollama local pour toutes les tâches de traitement du langage naturel (NLP).

**2. Composants Clés et Fichiers**

**Backend (server\_v4\_complete.py)**

C'est le cœur de l'application. Ses responsabilités principales sont :

1. **Initialisation de l'application Flask** et du serveur WebSocket (SocketIO).
2. **Initialisation de la base de données (init\_db)**: Crée le schéma SQLite au premier lancement, incluant les tables projects, search\_results, extractions, analysis\_profiles, et les nouvelles extraction\_grids et chat\_messages.
3. **Exposition des endpoints de l'API** via un Blueprint Flask (api\_bp).
4. **Mise en file d'attente des tâches** : Lorsqu'un endpoint demandant un long traitement est appelé (ex: /api/projects/<id>/run), le serveur ne fait que mettre en file d'attente la tâche correspondante (ex: process\_single\_article\_task) dans Redis et renvoie immédiatement une réponse 202 Accepted.
5. **Gestion des WebSockets** : Gère la connexion des clients et leur inscription à des "rooms" spécifiques à un projet pour recevoir des notifications ciblées.

**Endpoint clé ajouté :**

* POST /api/projects/<id>/import-zotero-file : Reçoit un fichier JSON exporté de Zotero, le lit en mémoire et met en file d'attente la tâche import\_from\_zotero\_file\_task pour un traitement asynchrone.

**Tâches Asynchrones (tasks\_v4\_complete.py)**

Ce fichier contient la logique de tous les processus longs qui ne doivent pas bloquer le serveur.

* **Interaction avec les APIs externes** : Fonctions pour interroger PubMed, arXiv, CrossRef, etc. (DatabaseManager).
* **Traitement IA** : La fonction call\_ollama\_api est le point d'entrée unique pour communiquer avec Ollama. Elle gère les formats de sortie (JSON, texte) et les tentatives multiples.
* **Nouvelles tâches notables** :
  + import\_from\_zotero\_file\_task: Utilise la classe ZoteroAbstractExtractor du module importers.py pour parser le fichier JSON, extraire les données et les insérer dans la base de données.
  + generate\_prisma\_diagram\_task: Utilise le module AdvancedPRISMAFlowExtractor pour générer un diagramme PRISMA, sauvegarder l'image et stocker son chemin dans la base de données.
  + run\_discussion\_generation\_task et run\_knowledge\_graph\_task: Appellent désormais les fonctions correspondantes depuis utils/analysis.py.

**Modules Utilitaires (utils/)**

La modularisation dans le répertoire utils permet de séparer clairement les préoccupations.

* **importers.py** : Contient la classe ZoteroAbstractExtractor.
  + **Rôle** : Charger un fichier CSL JSON de Zotero, en extraire les références, nettoyer les données (HTML, etc.), et les normaliser dans un format standard pour l'application.
  + **Robustesse** : La vérification isinstance(item.get("PMID"), (str, int, float)) a été ajoutée pour gérer les différents types de données possibles pour un PMID dans les exports Zotero.
* **analysis.py (Corrigé)** : Contient les fonctions d'analyse avancée.
  + **Rôle** : Fournir une logique pure pour transformer des données (DataFrames) en prompts complexes et traiter les résultats de l'IA.
  + **Correction apportée** : Les fonctions (generate\_discussion\_draft, generate\_knowledge\_graph\_data, etc.) reçoivent maintenant call\_ollama\_func et model comme arguments. Cela résout le problème des variables non définies et renforce l'indépendance du module : il n'a plus besoin de connaître l'implémentation de l'appel à l'IA, il a juste besoin d'une fonction qui le fait.

**Frontend (app.js, index.html, style.css)**

* **index.html**
  + **Rôle** : Fournir la structure de base de la SPA avec des conteneurs vides (<div id="projectsList">, <div id="projectDetail">, etc.) qui seront remplis dynamiquement par JavaScript.
  + **Modification nécessaire** : Le fichier doit être mis à jour pour inclure les nouveaux éléments d'interface, comme le formulaire <form id="zoteroImportForm">. Sans cette modification, les nouvelles fonctionnalités du backend ne sont pas accessibles à l'utilisateur.
* **app.js**
  + **Rôle** : Contenir toute la logique de l'interface utilisateur.
  + **Gestion d'état** : L'objet global appState centralise l'état actuel de l'application (projet sélectionné, listes de résultats, etc.).
  + **Gestionnaires d'événements** : setupEventListeners attache les listeners aux éléments statiques. Un listener délégué sur document.body gère les clics sur les éléments dynamiques via les attributs data-action.
  + **Interaction Backend** : La fonction fetchAPI centralise tous les appels HTTP vers le serveur Flask.
  + **Logique d'import** : Un nouvel écouteur d'événement pour le submit du formulaire zoteroImportForm a été ajouté. Il utilise FormData pour envoyer le fichier via fetchAPI à l'endpoint correspondant.
* **style.css**
  + **Rôle** : Définir l'apparence de l'application en utilisant le "Perplexity Design System" comme base.
  + **Technique** : Utilisation intensive des variables CSS (--var-name) pour une thématisation facile (modes clair/sombre) et une maintenance simplifiée des couleurs, polices et espacements.

**3. Flux de Données Typique (Exemple : Import Zotero)**

1. **Utilisateur** : Sélectionne un fichier export.json dans le nouveau formulaire d'import sur la page "Import" et clique sur "Importer".
2. **app.js** : Le listener sur zoteroImportForm s'active. Il crée un objet FormData, y attache le fichier, et appelle fetchAPI pour envoyer une requête POST à l'endpoint /api/projects/{id}/import-zotero-file.
3. **server\_v4\_complete.py** : L'endpoint reçoit la requête, lit le contenu du fichier, et met en file d'attente la tâche import\_from\_zotero\_file\_task dans Redis, en lui passant l'ID du projet et le contenu du JSON. Le serveur répond immédiatement 202 Accepted.
4. **Worker (RQ)** : Un worker récupère la tâche depuis la file Redis.
5. **tasks\_v4\_complete.py** : La fonction import\_from\_zotero\_file\_task s'exécute.
   * Elle utilise ZoteroAbstractExtractor pour parser le JSON.
   * Elle boucle sur les enregistrements normalisés et les insère dans la table search\_results de la base de données.
   * Une fois terminée, elle envoie une notification WebSocket via send\_project\_notification.
6. **app.js** : Le client WebSocket reçoit l'événement notification de type import\_completed. Le handleWebSocketNotification déclenche un rafraîchissement des données du projet (selectProject(id, true)), mettant ainsi à jour l'interface avec les nouveaux articles importés.

Ce flux asynchrone garantit que l'interface utilisateur reste réactive même pendant les imports volumineux.

⁂

1. [importers.py](http://importers.py)

1. [analysis.py](http://analysis.py)

1. style.css

1. index.html

1. app.js

1. tasks\_v4\_complete.py

1. requirements.txt

1. server\_v4\_complete.py