Réalisation d’une application de type RAG

**      **

Certification : Développeur d’application en Intelligence Artificiel

Marseille – Nice

Bloc de compétences 3 - E4

Réaliser une application intégrant un service d’intelligence artificielle

LOMBARDI Joachim

Table des matières

[A6. Compétence : Conception d'une application intégrant un service d'IA 3](#_Toc193039771)

[C14. Analyse du Besoin et Spécifications Fonctionnelles 3](#_Toc193039772)

[C15. Conception du Cadre Technique 3](#_Toc193039773)

[I. Les éléments clés de l’architecture Django 4](#_Toc193039774)

[A7. Compétence : Développement des interfaces et fonctionnalités 10](#_Toc193039775)

[C16. Coordination de la Réalisation Technique 10](#_Toc193039776)

[A. Conduite agile de projet. 10](#_Toc193039777)

[C17. Développement des Composants et Interfaces 14](#_Toc193039778)

[Sécurisation 15](#_Toc193039779)

[A8. Compétence : Développement des fonctions de tests et de contrôle 18](#_Toc193039780)

[C18. Automatisation des Tests avec Intégration Continue 18](#_Toc193039781)

[Tests 18](#_Toc193039782)

[C19. Création d'un Processus de Livraison Continue 20](#_Toc193039783)

[A. Outils de conteneurisation 20](#_Toc193039784)

[1) Docker 20](#_Toc193039785)

[2) Podman 21](#_Toc193039786)

[3) Kubernetes (K8s) 21](#_Toc193039787)

[B. Conteneurisation 25](#_Toc193039788)

[1) Service django 25](#_Toc193039789)

[2) Service elasticsearch 26](#_Toc193039790)

[3) Service elasticsearch\_exporter 26](#_Toc193039791)

[4) Service db 26](#_Toc193039792)

[5) Service postgres\_exporter 26](#_Toc193039793)

[6) Service ollama 27](#_Toc193039794)

[7) Service prometheus 27](#_Toc193039795)

[8) Service grafana 27](#_Toc193039796)

[9) Service renderer 27](#_Toc193039797)

[10) Service uptime-kuma 27](#_Toc193039798)

[11) Réseau networks 27](#_Toc193039799)

[Conclusion 28](#_Toc193039800)

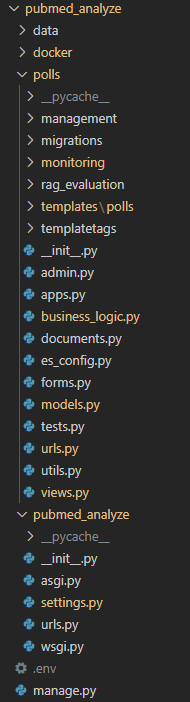
# A6. Compétence : Conception d'une application intégrant un service d'IA

## C14. Analyse du Besoin et Spécifications Fonctionnelles

Il s’agit d’une application permettant de poser des questions sur des articles médicaux issus du site internet PubMed. La méthode Merise a été utilisée pour réaliser la base de données. L’interface utilisateur est simple et lisible. Des messages d’erreurs ou de succès sont affichés afin d’indiquer à l’utilisateur si sa démarche a été réalisée correctement.

## C15. Conception du Cadre Technique

L’architecture du projet est celle d’un projet Django à savoir une architecture MVT (Model-View-Template)



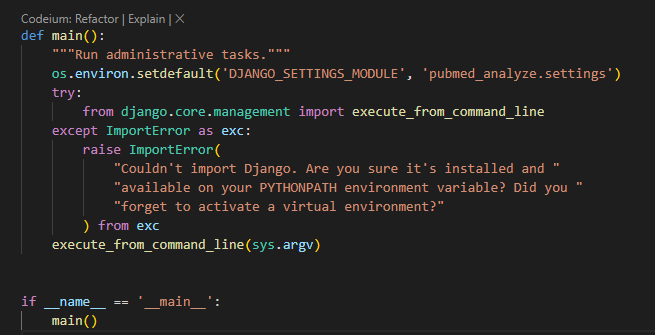
*Figure 1. Arborescence du projet pubmed\_analyze*

### Les éléments clés de l’architecture Django

#### manage.py

Un script utilisé pour exécuter des commandes Django comme :

* *python manage.py runserver* (lancer le serveur),
* *python manage.py migrate* (appliquer les migrations),
* *python manage.py createsuperuser* (créer un admin).

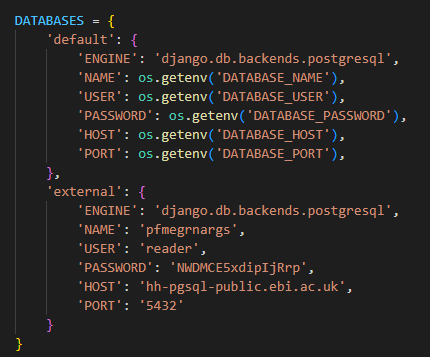


*Figure 2. manage.py*

#### 2. settings.py (Configuration du projet)

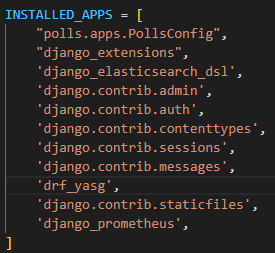
Ce fichier contient les paramètres globaux du projet Django :

* Configuration de la base de données (PostgreSQL, MySQL, SQLite…).



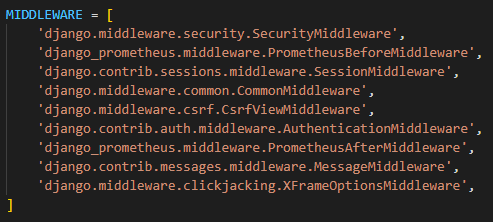
*Figure 3. settings.py -- bases de données*

* Applications installées (INSTALLED\_APPS).



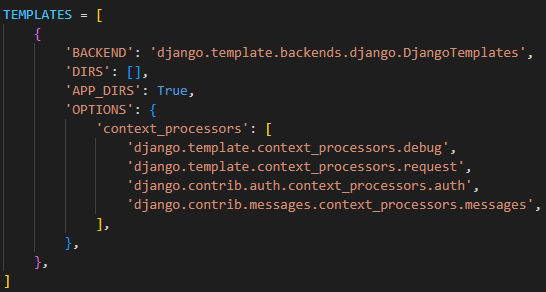
*Figure 4. settings.py -- Installed apps*

* Middleware (MIDDLEWARE) pour gérer les requêtes.



*Figure 5. settings.py -- Middlewares*

* Templates et fichiers statiques.



*Figure 6. settings.py -- Templates*

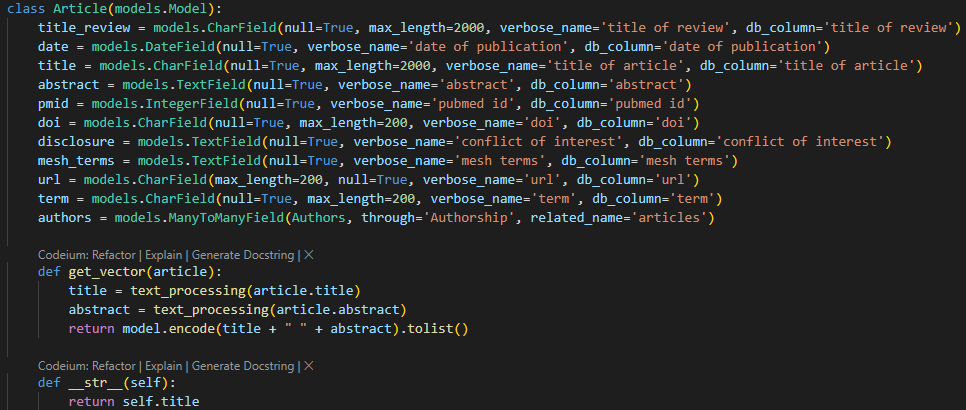
#### 3. urls.py (Gestion des URL)

Ce fichier définit le routage de l’application



*Figure 7. urls.py*

#### 4. models.py (Gestion de la base de données - ORM)

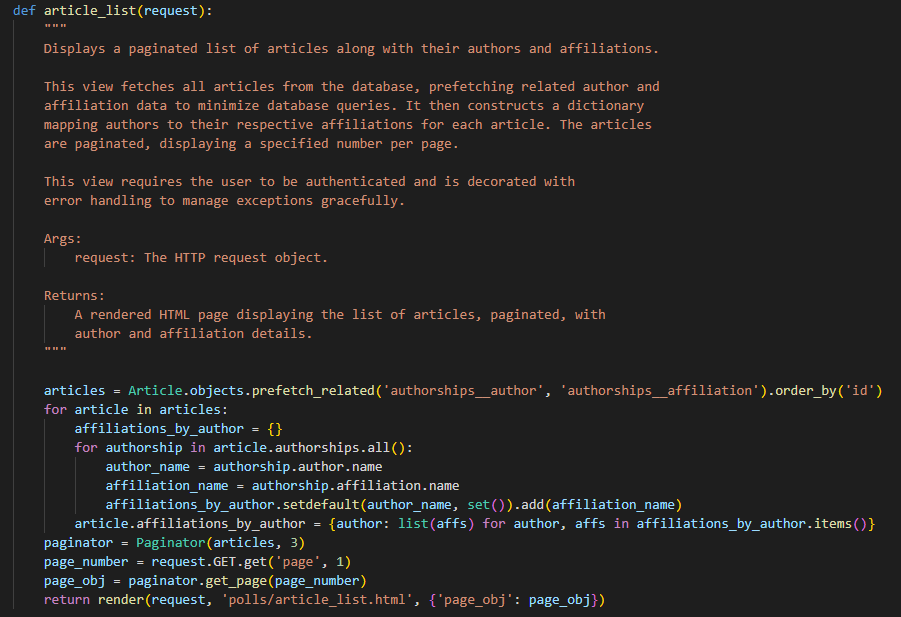


*Figure 8. models.py -- Articles*

Les modèles définissent la structure de la base de données en Python. Django génère ensuite automatiquement les tables SQL.

#### 5. views.py (Logique de l'application)

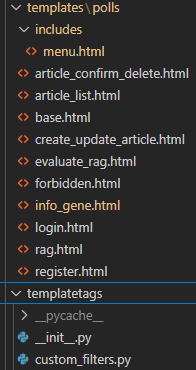
Les vues définissent la logique des pages web ou API.



*Figure 9. view.py -- article\_list*

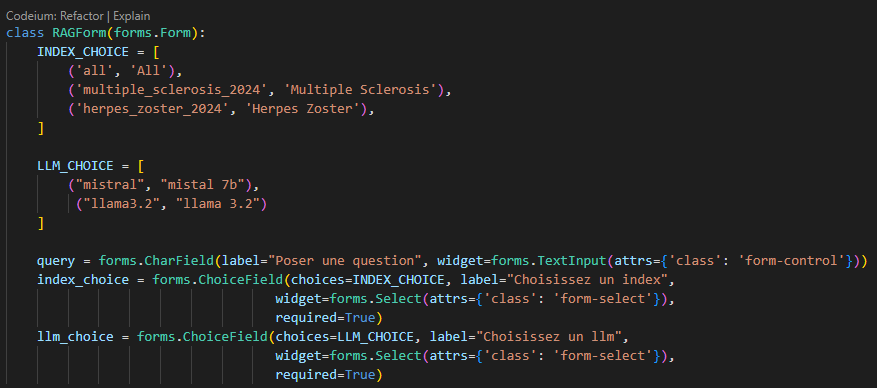
#### 6. templates/ (Système de Templates - HTML)

Django utilise un moteur de templates pour générer du HTML dynamique.



*Figure 10. Arborescence des templates*

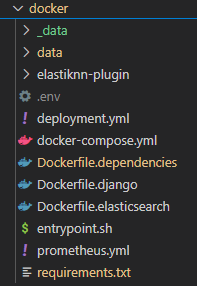
#### 7. forms.py (Gestion des formulaires)



*Figure 11. forms.py – RAGForm*

Il y a également des éléments spécifiques à mon projet :

#### 8. Docker (Gestion des conteneurs)

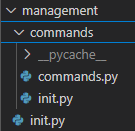


*Figure 12. Fichier docker*

C’est aussi là qu’est stocké mon fichier requirements.txt avec toutes les bibliothèques.

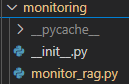
#### 9. Management (Commandes Django)

Elles permettent d’exécuter des fonctions dans l’invite de commande PowerShell (sans requête http)



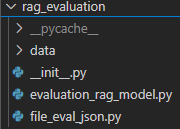
*Figure 13. Fichier management*

#### 10. Monitoring (Récupère les métriques du RAG)



*Figure 14. Fichier monitoring*

#### 11. RAG\_evaluation (Evalue le RAG)



*Figure 15. rag\_evaluation*

# A7. Compétence : Développement des interfaces et fonctionnalités

## C16. Coordination de la Réalisation Technique

### Conduite agile de projet.

La gestion du projet s’articule autour des méthodes agiles. Voici les principales méthodes de gestion utilisées :

#### La méthode SCRUM

Scrum est une méthodologie Agile qui vise à améliorer la flexibilité, la collaboration et l’efficacité dans le développement de produits, notamment en IT et en Machine Learning. Il repose sur des cycles courts appelés sprints, permettant une adaptation rapide aux changements.

Principaux Rôles dans Scrum :

* Scrum Master → Facilite le processus, élimine les obstacles et assure le respect des principes Scrum.
* Product Owner → Définit les besoins du produit et priorise les tâches dans le backlog.
* Équipe de développement → Développe, teste et livre les fonctionnalités du produit.

Événements Clés de Scrum :

* Sprint Planning → Planification des tâches à accomplir pendant le sprint (1 à 4 semaines).
* Daily Scrum (Stand-up Meeting) → Réunion quotidienne de 15 minutes pour synchroniser l’équipe.
* Sprint Review → Présentation des fonctionnalités développées à la fin du sprint.
* Sprint Retrospective → Analyse des points forts et axes d’amélioration pour le sprint suivant.

Artefacts Scrum :

* Product Backlog → Liste des fonctionnalités à développer.
* Sprint Backlog → Liste des tâches sélectionnées pour un sprint.
* Increment → Version améliorée du produit après chaque sprint.

SCRUM est la méthode agile la plus populaire.

#### Kanban

Kanban est une méthode Agile similaire à Scrum mais plus flexible et sans cycles fixes (sprints). Elle se concentre sur la gestion des flux de travail et l'amélioration continue.

Principaux Concepts :

* Tableau Kanban avec trois colonnes principales : À faire, En cours, Terminé.
* Limite de travail en cours (WIP) : Limiter le nombre de tâches dans une colonne pour éviter la surcharge.
* Amélioration continue : Revue régulière des processus pour améliorer le flux de travail.

Avantages :

* Flexibilité et adaptabilité pour les équipes en évolution.

Suivi visuel des tâches en temps réel.Outils populaires :

* Trello (qui propose aussi un mode Kanban)
* Jira

#### XP (Extreme Programming) : Méthode Agile Technique.

XP (Extreme Programming) se concentre sur l'amélioration de la qualité du code et l'agilité technique. C'est une approche fortement utilisée pour les projets de développement logiciel.

Principaux Concepts :

* Développement itératif : Des cycles de développement courts pour livrer rapidement des versions du produit.
* Code de qualité : Utilisation de pratiques comme la revue de code, les tests automatisés et l'intégration continue.

Avantages :

* Haute qualité du code.
* Adaptation rapide aux besoins du client.

Outils populaires :

* Jenkins (pour l'intégration continue)
* GitLab ou GitHub (gestion du code)

La méthode SCRUM est la méthode la plus pratique lors de travail en équipe car elle permet de visualiser et corriger l’évolution du projet à tout instant, cependant dans ce cas précis le projet à été réalisé en solo à titre individuel ainsi il n’y a jamais eu de présentation au client ni à l’équipe. A noter, que ces méthodes peuvent être complémentaires car Kanban permet de visualiser le SCRUM s’il y en a un. Ci-dessous une présentation des principaux outils de visualisation pour gérer les tâches :D.

#### Trello

Trello est un outil de gestion de projet visuel, basé sur la méthodologie Kanban. Il est souvent utilisé pour gérer les sprints Scrum de manière simple et intuitive.

Structure d’un Tableau Trello

* Tableau → Représente un projet.
* Listes → Représentent les étapes du projet (ex: À faire, En cours, Terminé).
* Cartes → Représentent les tâches spécifiques (ex: “Déployer un modèle ML”).
* Étiquettes, deadlines et membres assignés → Facilitent l’organisation des tâches.

Exemple d’Utilisation de Trello dans Scrum

* Product Owner ajoute les tâches dans le Product Backlog.
* L’équipe choisit les tâches pour le sprint et les place dans la liste "À faire".
* Pendant le sprint, les tâches passent par "En cours" puis "Terminé" après validation.

#### Jira

Jira, développé par Atlassian, est l'un des outils les plus populaires dans le domaine Agile et Scrum. Il est particulièrement adapté pour des projets logiciels ou des équipes de développement.

Caractéristiques principales :

* Suivi détaillé des sprints et des backlogs.
* Possibilité d’intégrer des tableaux Kanban et Scrum pour la gestion des tâches.
* Outils de gestion de version et de test intégrés.

Avantages :

* Très adapté aux équipes techniques.
* Suivi détaillé des problèmes, tâches et bugs.

Inconvénients :

* Peut-être complexe pour des projets non techniques.
* Courbe d’apprentissage.

#### Asana : Outil de Gestion de Projet Collaboratif

Asana est une alternative populaire pour la gestion des tâches et la collaboration d’équipe. Il peut être utilisé pour différents types de projets, y compris ceux en Agile et en Scrum.

Caractéristiques principales :

* Suivi des tâches avec des listes de tâches ou des tableaux Kanban.
* Assignation des tâches, priorisation, et suivi des progrès.
* Collaboration en temps réel avec commentaires et pièces jointes.

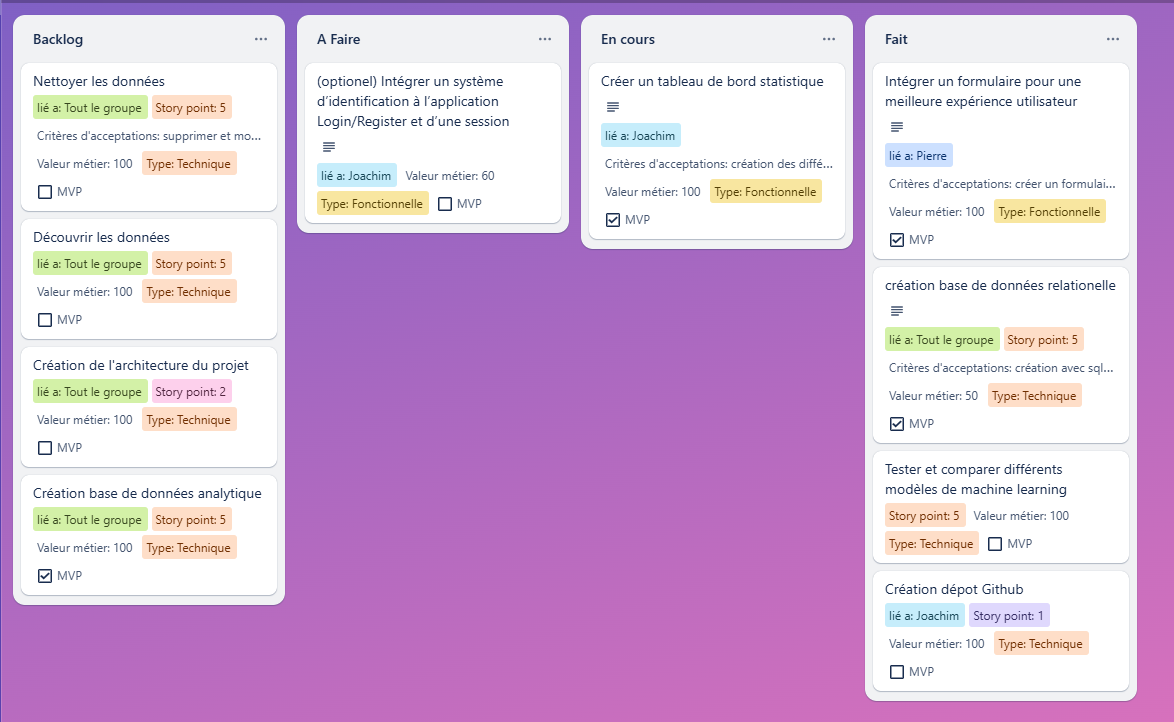
Avantages :

* Interface conviviale et flexible.
* Collaboration fluide pour les équipes.

Inconvénients :

* Moins de fonctionnalités spécifiques pour les équipes de développement logiciel.
* Manque de certaines intégrations avancées.

Trello est l’outil le plus populaire, il est simple et complet de plus je l’ai déjà utilisé dans le cadre de ma formation. Voici un Trello réalisé dans un projet antérieur :



*Figure 16. Kanban*

Les dates n’y sont pas présentes, dans sa version gratuite, mais on peut y voir les différentes tâches à effectuer, celui ou ceux qui ont à l’effectuer, les story points représentant l’effort à fournir pour réaliser la tâche et les valeurs métiers désignant l’importance de la tâche pour le client ou l’entreprise. Également, si la tâche est essentielle au MVP (Produit Minimum Viable) à savoir une version contenant les parties essentielles du projet.

## C17. Développement des Composants et Interfaces

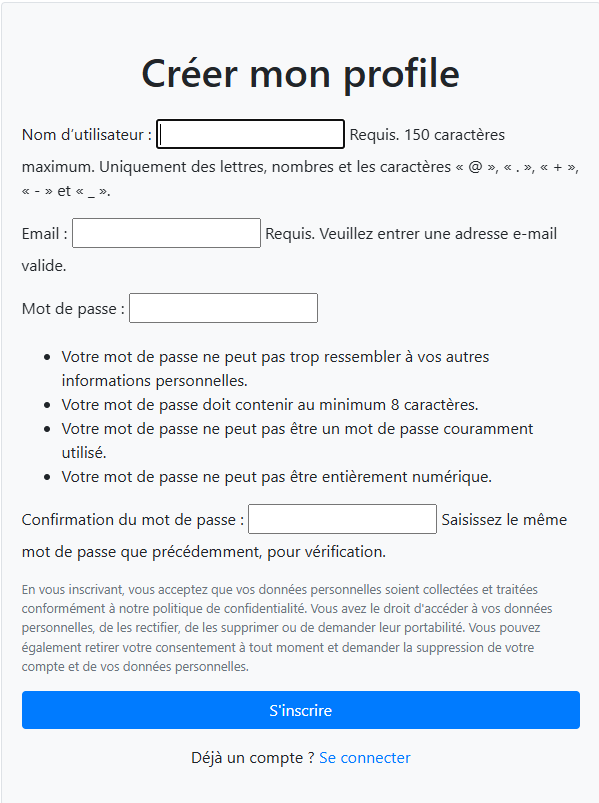
Les prérequis pour installer l’environnement de développement sont indiqués sur le README : <https://github.com/JoachimLombardi/Projet_chef_d-oeuvre/blob/master/README.md>

Le Front-End a été développé avec Django + bootstrap.



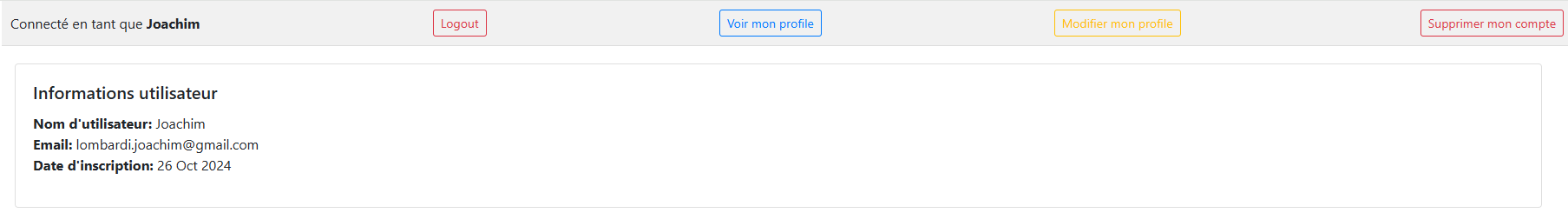
*Figure 17. Extrait du fichier base.html montrant l'appel à Bootstrap*

L’application respecte les règles du RGPD. Lors de l’inscription l’utilisateur est informé de ses droits.

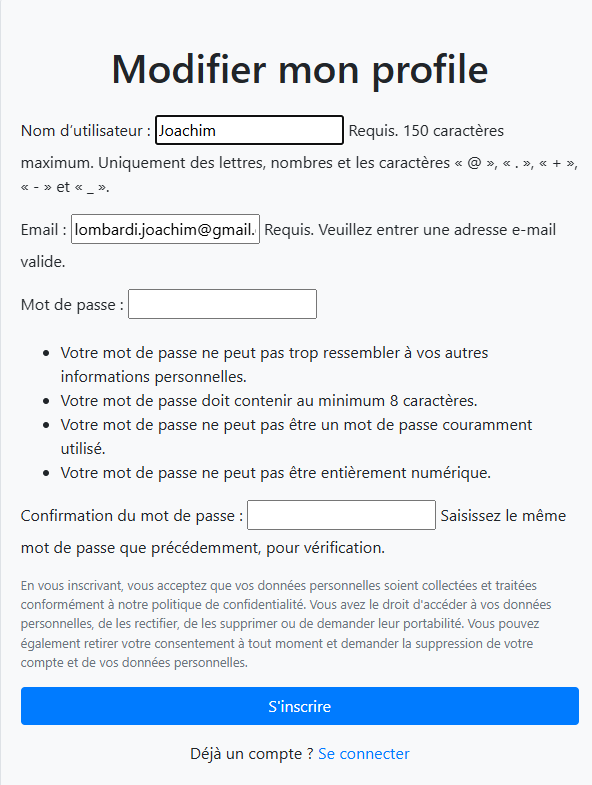


*Figure 18. Formulaire d'inscription*

Une fois connecté, l’utilisateur peut librement consulter, modifier ou supprimer ses informations personnelles.



*Figure 19. Ecran d'affichage des informations personnelles*

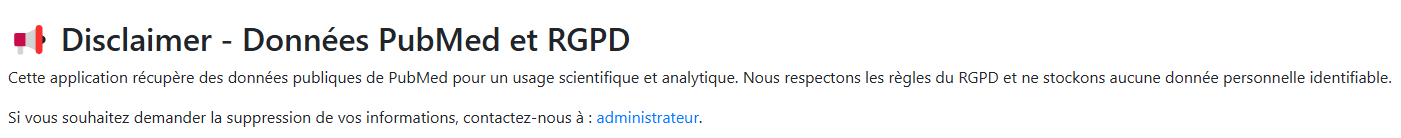


*Figure 20. Formulaire d'édition des informations personnelles*



*Figure 21. Formulaire de confirmation de suppression du compte personnelle.*

En ce qui concerne les données issues des articles scrappés sur PubMed, un disclaimer informatif indique l’adresse email de la personne à contacter pour la suppression des articles désirés.



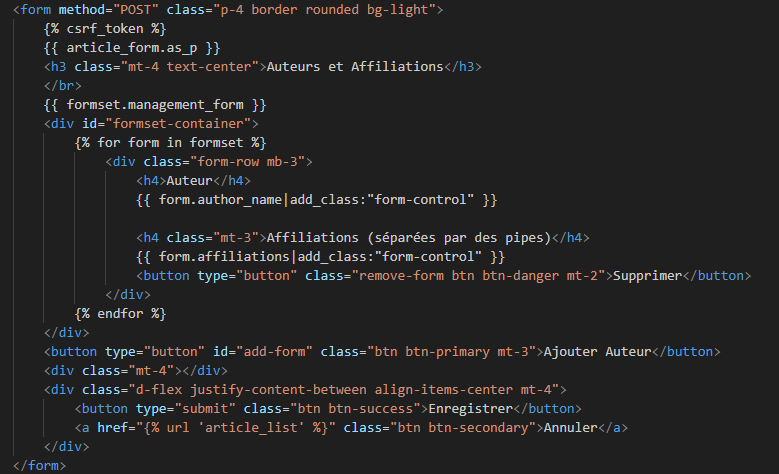
*Figure 22. Disclaimer informatif.*

L’application est également sécurisée par Django comme indiqué dans le rapport E1.

### Sécurisation

#### HTML

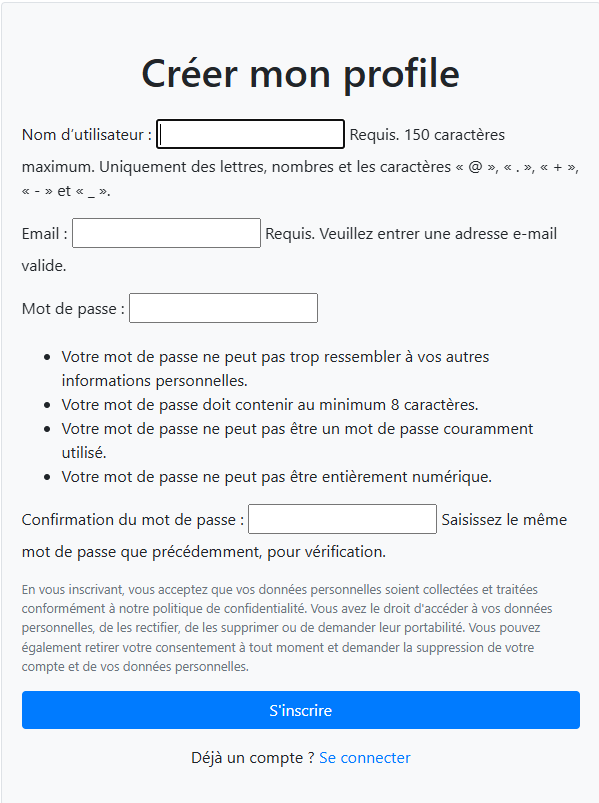
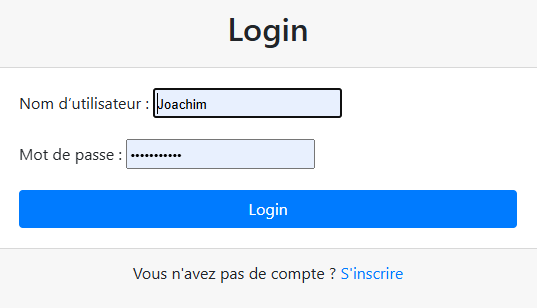
Les formulaires HTML sont protégés contre les attaques CSRF via un jeton placé dans chaque formulaire.



*Figure 4. Extrait du HTML create\_update.*

Ci-dessus, le jeton CRSF, placé juste en-dessous de la balise d’ouverture form.

#### Ecran d’inscription/connexion

*Figure 5. Ecran d’inscription Figure 6. Ecran de connexion*

L’accès au site nécessite un login et mot de passe créés lors de l’inscription.

Pour créer un utilisateur, on importe la classe *UserCreationForm* de Django. La classe *UserCreationForm* valide automatiquement les champs *username*, *password1*, et *password2*, garantissant qu'ils respectent les exigences de Django, comme l'unicité du nom d'utilisateur et la correspondance des mots de passe. Pour le login, on utilise *l’AuthenticationForm*, Il vérifie automatiquement le nom d'utilisateur et le mot de passe, et gère les erreurs d'authentification. On utilise la fonction *authenticate* de Django qui vérifie que les credentials soient valides. On utilise ensuite la fonction *auth\_login* pour persister les éléments de l’utilisateur sur chaque requête. Django utilise l'algorithme de hachage PBKDF2 (Password-Based Key Derivation Function 2) pour hasher les mots de passe. Cet algorithme applique un certain nombre de tours de hachage (par défaut 100 000) pour rendre le hachage plus lent et plus difficile à attaquer par des attaques par force brute. En plus du hachage, un "sel" (un nombre aléatoire unique) est généré pour chaque mot de passe. Le sel est ajouté au mot de passe avant le hachage pour empêcher les attaques cryptographiques comme les attaques par tables de hachage pré-calculées ou par dictionnaire. Le mot de passe haché et le sel sont stockés dans la base de données. Le mot de passe en clair n'est jamais enregistré.

##### Attaque par table de Hashage pré-calculées (RainBow tables)

Les tables de hachage pré-calculées, ou rainbow tables, sont des tables qui contiennent des hachages pour une vaste collection de mots de passe courants des millions voir des milliards. Ces tables permettent à un attaquant de rechercher rapidement un hachage dans la table pour trouver le mot de passe original sans avoir à recalculer le hachage pour chaque mot de passe.

Rôle du salage : L'ajout d'un "sel" unique à chaque mot de passe empêche les attaques par rainbow table. Même si deux utilisateurs ont le même mot de passe, leurs hachages seront différents en raison du sel. Cela rend inutile l'utilisation de rainbow tables, car l'attaquant ne pourra pas prédéfinir tous les hachages possibles.

##### Attaque par dictionnaire

L'attaquant possède un fichier contenant des millions de mots de passe communs (ex : "123456", "password", "qwerty", etc.). Il hache chaque mot de passe avec le même algorithme que celui utilisé par le système cible. Il compare les hachages obtenus avec le hachage qu'il veut casser. Si un hachage correspond, il a trouvé le mot de passe. Ici le salage ne sert à rien car il utilise le même algorithme en revanche, il faut tester les mots de passe un par un ce qui est très coûteux.

Aujourd’hui, la plupart des systèmes utilisent un sel ce qui rend l’attaque par Rainbow tables inefficace. Ces deux techniques montrent l’intérêt de ne pas choisir de mot de passe simple.

Pour se connecter à mon application, l’utilisateur s’inscrit pour se connecter au site en renseignant les champs suivants :

* Nom d’utilisateur
* Email
* Password
* Password (répéter)

Pour se connecter, il renseigne son nom d’utilisateur et son mot de passe. Il a accès à toutes les pages publiques. Les pages privées nécessitent un accès administrateur créé via une commande Django.

Quand un utilisateur se connecte, Django re-hache le mot de passe fourni et compare le résultat avec celui stocké en base de données.

Compétence validée : Développer une API mettant à disposition le jeu de données en utilisant l’architecture REST afin de permettre l’exploitation du jeu de données par les autres composants du projet.

L’utilisation de l’application est facile et agréable. Des jeux de couleurs sont utilisés pour différencier les différents boutons (lecture, création/modification, suppression). Le menu latéral surligne la catégorie sélectionnée. Des messages d’erreur ou de succès aux couleurs contrastées indiquent à l’utilisateur l’efficacité de son action. Les données sont affichées dans un cadre clair et plaisant à lire. La navigation est intuitive et permet d’accéder à toutes les parties de l’application et de revenir en arrière à tout instant.

# A8. Compétence : Développement des fonctions de tests et de contrôle

## C18. Automatisation des Tests avec Intégration Continue

### Tests

Django fournit ses propres outils de tests. Afin de réaliser les tests, on utilise la classe TestCase qui permet d’écrire des tests unitaires. TestCase est compatible avec les outils de test standard de Python. Django utilise une base de données en mémoire (SQLite) pour accélérer les tests. TestCase fournit un client HTTP intégré pour simuler des requêtes (GET, POST, etc.) et tester les vues et les réponses.

J’utilise aussi la bibliothèque MAgicMock. Conçue pour faciliter la création de faux objets (mocks) dans les tests. Ces objets simulés peuvent être utilisés pour remplacer des fonctions, des méthodes ou des objets réels lors de tests unitaires, afin d'isoler les fonctionnalités et de vérifier les interactions.

#### Test de scrapping

Lors de la définition de la fonction de test, le décorateur patch permet de remplacer des objets ou des fonctions par des Mocks, les arguments passés sont des fonctions à l’intérieur des décorateurs. Les fonctions correspondantes aux arguments de la fonction de test, vont respectivement du décorateur le plus bas au plus haut. Dans ce test, je lui passe la fonction pour initialiser BeautifulSoup (récupération du HTML correspondant à l’URL ) et la fonction pour récupérer toutes les URL liées à mes filtres.

Je crée un objet avec MagiMock qui représente beautifulsoup avec ses attributs. A l’intérieur un deuxième objet MagicMock simule la page html, vérifie la présence de la balise html passé dans lambda et retourne le texte inscrit dans le deuxième MagicMock. L’attribut side\_effect indique ce que le Mock est censé ressortir lors de l’appel.

Ensuite vient l’évaluation du résultat en JSON via les méthodes héritées de TestCase. Dans ce test, via les méthodes de type Assert, on vérifie si le JSON est conforme, comme la présence des différentes clés ou bien le fait que le JSON contient une liste.

#### Test d’import en base

Ici, on vérifie que les données sont bien présentes dans la base de données temporaire. On fait un appel à l’objet en base et on vérifie que la fonction d’import en base de données ait bien importé l’article en regardant un par un ses attributs.

#### Test du CRUD

##### Test du read

L’utilisation du décorateur *classmethod* permet d’initialiser un objet article pour tous les tests évitant de ré exécuter la fonction pour chaque test. Les tests vérifient que l’objet est lisible pour l’url correspondante et que son affichage utilise le template correspondant

##### Test du create et de l’update

Les tests sont assez similaires avec la vérification de l’URL. L’attribut « refesh\_from\_db » permet de récupérer le dernier objet après la mise à jour de la base de données.

##### Test du delete

On vérifie que l’URL correspondante est bonne et que l’objet a bien été supprimé de la base de données.

Compétence validée : Programmer les tests automatisés d’un modèle d’intelligence artificielle en définissant les règles de validation des jeux de données, des étapes de préparation des données, d'entraînement, d’évaluation et de validation du modèle pour permettre son intégration en continu et garantir un niveau de qualité élevé.

#### Test du RAG

Dans la méthode de classe on utilise *cls* car il permet de modifier des attributs pour toutes les instances de classe et non une seule. On définit deux fonctions que l’on va « Mocker » : Search et encode. On attribue une valeur fictive à l’encodeur fictif. Le *Mock\_response* va créer une réponse fictive pour chaque réponse du moteur de recherche avec l’id, le titre et l’abstract de l’article.

Lorsqu’une recherche est exécutée le *Mock\_search* va renvoyer la réponse fictive du *Mock\_response*. Cela revient à simuler une recherche *elasticsearch*.

*RequestFactory* permet de créer une requête POST avec la query, l’index et l’URL du RAG. La requête est associée à l’utilisateur car il faut se connecter pour utiliser le RAG. Enfin, la requête est envoyée à la vue du RAG avec la requête fictive et on teste s’il n’y a pas eu d’erreur.

Ici le principe n’est pas réellement de tester si l’application fonctionne parfaitement mais simplement d’effectuer des tests sur certaines parties pour notamment découvrir les Mocks. Le code pourra être amélioré par la suite.

# C19. Création d'un Processus de Livraison Continue

Afin de déployer ma solution, j’ai besoin d’encapsuler tout ce dont mon application a besoin pour fonctionner : le code, les bibliothèques, les dépendances et les configurations. Pour réaliser ça, j’utilise des conteneurs. Cela va lui permettre de marcher aussi bien sur un serveur de production qu’en local. Dans un conteneur, l’environnement et les ressources restent les mêmes quelle que soit la machine sur laquelle il est déployé. Les conteneurs permettent également d es architectures de micro services ici : Django, PostgreSQL, elasticsearch… où chaque service est exécuté dans son propre conteneur. Voici les principaux outils de conteneurisation.

## Outils de conteneurisation

### Docker

Docker est une plateforme open-source permettant de créer, de déployer et d'exécuter des applications dans des conteneurs.

Ces principales caractéristiques sont :

* Portabilité :
  + Les conteneurs Docker fonctionnent sur n’importe quel environnement avec Docker installé (Windows, MacOs, Linux).
* Isolation :
  + Chaque conteneur est isolé, avec ses propres ressources (CPU, mémoire, stockage) et ne partage pas ses processus avec d'autres conteneurs.
* Efficacité :
  + Contrairement aux machines virtuelles (VM), Docker utilise le même noyau que l’hôte, ce qui réduit les besoins en ressources.
* Écosystème riche :
  + Docker Hub propose des milliers d’images prêtes à l’emploi (bases de données, serveurs web, environnements de développement).

Les principaux composants de docker sont :

* Les images : elles contiennent tout ce dont une application a besoin pour fonctionner.
* Les conteneurs : Instances exécutables des images Docker. Ils encapsulent l’application avec leurs dépendances dans un environnement isolé.
* Dockerfile : Un fichier texte contenant les instructions pour construire une image Docker.
* Docker Compose : Un outil pour définir et gérer plusieurs conteneurs à l’aide d’un fichier docker-compose.yml.
* Docker Hub : Une plateforme pour partager et télécharger des images Docker.

### Podman

Comme Docker, Podman est un outil de conteneurisation qui s'adresse aux environnements nécessitant une sécurité renforcée et il nécessite moins de dépendance que Docker.

### Kubernetes (K8s)

Permet l’orchestration de micro services dans des environnements cloud (AWS, GCP, Azure) et le déploiement d'applications complexes avec plusieurs conteneurs.

J’ai choisi d’utiliser Docker avec Azure car j’ai plusieurs conteneurs à déployer sur Azure. Pour docker, je l’utilise en entreprise et je suis plus familier.

Azure est une plateforme de cloud computing développée par Microsoft. Elle offre une large gamme de services pour héberger, gérer et déployer des applications et infrastructures à l’échelle mondiale.

Voici les principales catégories proposées par Azure :

* Informatique et Virtualisation
  + Azure Virtual Machines (VMs) : Machines virtuelles pour exécuter des applications sur Linux ou Windows.
  + Azure Kubernetes Service (AKS) : Service managé pour exécuter des applications en conteneurs avec Kubernetes.
  + Azure Functions : Permet d’exécuter du code sans gérer de serveurs (serverless computing).
* Stockage et Bases de Données
  + Azure Blob Storage : Stockage d’objets évolutif pour les fichiers, images, vidéos, etc.
  + Azure SQL Database : Base de données relationnelle managée (SQL Server dans le cloud).
  + Azure Cosmos DB : Base de données NoSQL haute performance.
* Réseau et Sécurité
  + Azure Virtual Network (VNet) : Création de réseaux privés dans Azure.
  + Azure Firewall : Protection avancée contre les attaques réseau.
  + Azure DDoS Protection : Sécurisation contre les attaques DDoS.
* Intelligence Artificielle et Big Data
  + Azure AI Services : Inclut des services IA comme Cognitive Services (vision, reconnaissance vocale, NLP).
  + Azure Machine Learning : Développement et déploiement de modèles de machine learning.
  + Azure OpenAI Service : Accès aux modèles GPT, comme ceux de ChatGPT.
* Développement et DevOps
  + Azure DevOps : Outils pour CI/CD, gestion de code, et pipelines automatisés.
  + Azure App Service : Déploiement rapide d’applications web et mobiles.
  + Azure Logic Apps : Automatisation des workflows d’entreprise.
* Analyse de Données
  + Azure Synapse Analytics: Plateforme de Big Data et Data Warehousing.
  + Azure Data Factory : Orchestration de flux de données.
  + Azure Databricks : Analytique avancée et Machine Learning basé sur Apache Spark.

Comme indiqué précédemment, j’utilise Azure. J’ai automatisé le déploiement dans le fichier déployment.yaml.

Ce fichier gère le déploiement et la gestion des Pods. Il assure que le bon nombre de réplicas (instances) tourne et peut redémarrer les Pods en cas d’échec.

**1. Instructions d'authentification et d'accès à Azure**

Ces lignes contiennent des commandes de base pour se connecter à Azure, créer un groupe de ressources, et se connecter à un registre Azure Container Registry (ACR) :

* **az login** : Ouvre une session Azure CLI.
* **az group create** : Crée un groupe de ressources dans Azure, ici appelé devia25 à l'emplacement francecentral.
* **az acr login** : Se connecte à un registre Docker Azure, ici nommé pubmedregistry, pour permettre de pousser et récupérer des images Docker.

**2. Spécifications de ressources (apiVersion, location, name, properties)**

* apiVersion: 2021-09-01
* location: francecentral
* name: pubmed-container-group

Ces lignes définissent la version de l'API (ici 2021-09-01) utilisée pour créer des ressources dans Azure. La ressource est située dans la région francecentral et son nom est pubmed-container-group.

**3. Définition des conteneurs (dans la clé containers)**

Cette section spécifie les différents services (conteneurs) qui sont utilisés dans ce groupe de conteneurs.

**- Django (Application Web)**

* **name: django** : Nom du conteneur (le service est l'application Django).
* **image: joachimlombardi/django:latest** : Image Docker à utiliser, ici une image Docker personnalisée pour Django.
* **ressources** : Les ressources allouées au conteneur : 0.5 vCPU et 1 Go de mémoire.
* **ports** : Le port exposé du conteneur, ici le port 8000 pour l'application Django.
* **environmentVariables** : Variables d'environnement passées au conteneur, telles que les paramètres Django, la connexion à la base de données et autres informations sensibles (ex : clés API).

**PostgreSQL (Base de données)**

* **name: postgres** : Nom du conteneur pour la base de données PostgreSQL.
* **image: joachimlombardi/postgres:16** : Image Docker de PostgreSQL (version 16).
* **ressources** : 0.5 vCPU et 1 Go de mémoire pour PostgreSQL.
* **ports** : Exposition du port 5432 pour la base de données PostgreSQL.

**- Elasticsearch**

* **name: elasticsearch** : Nom du conteneur pour Elasticsearch.
* **image: joachimlombardi/elasticsearch:latest** : Image Docker pour Elasticsearch.
* **ressources** : 1 vCPU et 2 Go de mémoire pour Elasticsearch.
* **ports** : Exposition du port 9200 pour Elasticsearch.

**- Grafana**

* **name: grafana** : Nom du conteneur pour Grafana, une plateforme d’analyse et de visualisation.
* **image: joachimlombardi/grafana:latest** : Image Docker pour Grafana.
* **ressources** : 0.25 vCPU et 0.5 Go de mémoire pour Grafana.
* **ports** : Exposition du port 3000 pour Grafana.

**- Autres services**

Des services similaires sont définis pour **Ollama**, **Prometheus**, et **Uptime Kuma**, chaque service ayant ses propres ressources, ports et variables d'environnement. Ces services sont utilisés pour la surveillance, la gestion des performances, et les alertes de l'application.

**4. Propriétés globales du groupe de conteneurs**

* **osType: Linux** : Le système d'exploitation utilisé pour ces conteneurs est Linux.
* **restartPolicy: Always** : Cela indique que les conteneurs doivent être redémarrés automatiquement en cas de défaillance.
* **ipAddress** : La configuration de l'adresse IP publique pour les conteneurs avec un label DNS pubmed-containers, permettant d'accéder à l'application via une adresse web.
* **ports** : Les ports TCP exposés pour les différents services, comme l'application Django (8000), Grafana (3000), et Uptime Kuma (3001).

Le déploiement se fait lors du push sur le cicd, on définit donc un fichier dans django.yml :

**1. deploy:**

Cette section décrit le processus de déploiement utilisé dans un pipeline CI/CD (intégration continue / déploiement continu).

* **runs-on: ubuntu-latest** : Cette ligne indique que le job de déploiement sera exécuté sur une machine virtuelle avec Ubuntu comme système d'exploitation.
* **needs: test** : Cette ligne signifie que le job de déploiement ne s'exécutera que si le job "test" a réussi. Cela permet de garantir que les tests passent avant d'envoyer l'application en production.

**2. Environnement de déploiement (env:)**

Les variables d'environnement suivantes sont définies pour configurer l'application sur azure :

* **ALLOWED\_HOSTS, DATABASE\_URL, etc.** : Ces variables sont des secrets stockés dans GitHub et récupérés via *secrets*. Elles permettent de sécuriser l'accès à la base de données et à d'autres ressources sensibles, telles que les clés secrètes ou les informations sur la base de données.

Voici les principales variables d'environnement :

* **ALLOWED\_HOSTS** : Liste des hôtes autorisés à accéder à l'application Django.
* **DATABASE\_URL**, **DATABASE\_HOST**, **DATABASE\_USER**, **DATABASE\_PASSWORD**, **DATABASE\_NAME**, **DATABASE\_PORT** : Informations de connexion à la base de données.
* **DJANGO\_SECRET\_KEY** : La clé secrète Django utilisée pour la cryptographie et la sécurité.
* **DJANGO\_SETTINGS\_MODULE** : Indique quel fichier de paramètres Django est utilisé. Ici, c'est *pubmed\_analyze.settings*.

**3.Étapes du Job (steps)**

Cette section définit les différentes étapes à suivre dans le job de déploiement. Chaque étape correspond à une action spécifique.

**- Checkout code**

Cette étape extrait le code du dépôt GitHub afin qu'il soit disponible pour les étapes suivantes.

**- Set up Python**

Cette étape configure l'environnement Python. Ici, Python 3.11.8 est spécifié pour l'environnement.

**- Install dependencies**

Cette étape installe les dépendances Python nécessaires à partir du fichier *requirements.txt*, qui contient toutes les bibliothèques nécessaires pour le projet.

**- Deploy to Azure (or your cloud provider)**

Cette étape est utilisée pour déployer l'application sur un fournisseur de cloud (par exemple, Azure). Elle utilise la commande *Azure CLI az webapp up* pour déployer l'application sur Azure App Service.

**- Run migrations**

Cette étape exécute les migrations Django pour mettre à jour la base de données avec les dernières modifications des modèles. Cela garantit que la base de données est synchronisée avec le code de l'application après le déploiement.

**- Restart application**

Cette étape redémarre l'application une fois le déploiement terminé. Cela peut être nécessaire pour appliquer toutes les modifications ou pour résoudre d'éventuels problèmes de performance.

## Conteneurisation

Pour préparer mon application au déploiement, j’ai conteneurisé chaque micro-services via le fichier *docker-compose.yaml .*

### Service django

D’abord, je crée le conteneur principal qui va contenir Django et toutes les dépendances dont j’ai besoin pour exécuter mon application.

Pour construire mon image, j’utilise un Dockerfile : *Dockerfile.django.* Il va faire appel à une autre image que j’ai défini dans Dockerfile.dependencies et que j’ai envoyé sur Docker Hub qui contient toutes les bibliothèques, ainsi que le langage nécessaire pour faire fonctionner mon application. Cela me permet d’éviter de réinstaller toutes les bibliothèques et Python lorsque je reconstruis les conteneurs. Ensuite je crée et je construis un répertoire de travail dans lequel je copie les fichiers de mon application. J’utilise un script Shell pour exécuter toutes les commandes Django nécessaire à l’initialisation de mon application : la création de la base de données, celle du super utilisateur (admin), le remplissage de la base de données et l’indexation des articles dans elasticsearch. Enfin, le script lance le serveur Django.

Après avoir créé l’image django, on définit les ports, on lui passe toutes les variables d’environnements, on monte le code source local dans le conteneur pour synchroniser les modifications et on indique que ce conteneur dépend d’autres conteneurs, notamment elasticsearch, prometheus, db (PostgreSQL), et ollama.

Ci-dessous le lien vers mon docker-compose.

<https://github.com/JoachimLombardi/Projet_chef_d-oeuvre/blob/master/pubmed_analyze/docker/docker-compose.yml>

### Service elasticsearch

Ce conteneur va fournir Elasticsearch, définir ses ports. Le configurer en mono nœud et définir les options mémoire pour JAVA. Enfin, il va monter un répertoire local. Il va également construire une image personnalisée d’Elasticsearch, via Dockerfile.elasticsearch, qui inclut le plugin Elastiknn, un plugin conçu pour effectuer des recherches de similarité approximative (ANN, Approximate Nearest Neighbors) dans Elasticsearch. Le Dockerfile va ajouter le plugin à Elasticsearch. L’avantage de ANN par rapport à KNN c’est les temps de calculs permettant une vitesse accrue en sacrifiant légèrement la précision.

### Service elasticsearch\_exporter

Il va exporter les métriques pour Prometheus, définir le port et configurer l’URL d’accès à Elasticsearch.

### Service db

Fournit la base de données PostgreSQL, définit l’environnement et le port et monte le répertoire.

### Service postgres\_exporter

Exporte les métriques PostgreSQL pour Prometheus. Définit le port pour les métriques.

### Service ollama

Service utilisant l’image d’ollama, définit les ports et monte le répertoire.

### Service prometheus

Service de surveillance qui collecte et stocke les métriques des services comme PostgreSQL ou Elasticsearch. Définit le port pour accéder à l'interface de Prometheus. Monte un fichier de configuration prometheus.yml et un répertoire pour les données.

Le fichier prometheus.yml est une configuration pour Prometheus. Prometheus va interroger toutes les cibles définies dans scrape\_configs toutes les 15 secondes pour récupérer les métriques.

### Service grafana

Service pour fournir des visualisations des métriques collectées via Prometheus. Définit le port. Il configure un service de rendu d'images pour les graphiques via le service renderer.

### Service renderer

Service complémentaire pour Grafana permettant de générer des images, des graphiques.

### Service uptime-kuma

Service de surveillance pour vérifier la disponibilité des autres services. Définit le port. Monte des répertoires locaux pour les données et l'accès au socket Docker. Monter le Docker socket dans le conteneur Uptime Kuma permet à cet outil de surveiller et de gérer les conteneurs Docker directement.

### Réseau networks

Un seul réseau est défini sous le nom django\_network. Tous les services sont connectés à ce réseau par défaut, ce qui leur permet de communiquer via leurs noms de conteneur.

# Conclusion

On a réalisé l’architecture complète de l’application. Les interfaces utilisateurs sont présentes et fonctionnelles. La gestion de projet, elle, a été présentée dans le cadre d’un autre projet car ce projet a été réalisé seul. La sécurité des formulaires a été implémenté. Tous les tests ont été réalisés. Les outils et le processus de conteneurisation ont été définis.