COMMUNAUTÉ DE PRATIQUE
IA en Santé
De la **recherche** à l'**application**

Répertoire des expertises de la CPIAS

Guide du développeur

2023-12-04

Table des matières

1. Introduction **3**
1.1. Présentation du logiciel 3
1.2. Présentation du guide du développeur 3
2. Interface utilisateur **3**
2.1. Technologies utilisées 3
2.1.1. React 3
2.1.2. Typescript 3
2.1.3. Chakra UI 3
2.2. Structure du projet 4
2.3. Variables environnement 4
2.4. Composants fonctionnels 4
2.5. Navigation 5
2.5.1. Fichier Routes.tsx 5
2.5.2. Navigation à partir des composants 6
2.6. Communication avec le serveur 7
2.7. Interface réactive 7
2.8. Lancement de l'application 9
2.8.1. Prérequis 9
2.8.2. Lancement de l'application 9
2.9. Déploiement 9
2.9.1. Prérequis 9
2.9.2. Compilation et déploiement 9
3. Serveur **11**
3.1. Technologies utilisées 11
3.1.1. Python (langage de programmation): 11
3.1.2. Flask (Cadriciel Web): 11
3.1.3. Gunicorn (Serveur WSGI): 11
3.1.4. Nginx (Serveur mandataire inversé): 11
3.1.5. SQLAlchemy (Base de données ORM): 11
3.1.6. ChromaDB (Stockage d'Embeddings): 11
3.1.7. spaCy (Traitement du langage naturel): 12
3.1.8. LangChain (Interaction avec les Grands Modèles de Langage): 12
3.1.9. ZeroMQ (bibliothèque de messagerie): 12
3.1.10. Ollama App (Exécuteur Local de LLMs): 12
3.1.11. KeyBert (Extraction de mots-clés): 12
3.1.12. Transformateurs de phrases (calcul des embeddings): 12
3.1.13. Deep Translator (traduction de texte): 12

3.2. Structure du projet 12
3.3. Variables environnement 14
3.4. Architecture du serveur 15
3.4.1. Serveur Web 15
3.4.2. Base de données 16
3.4.3. Intelligence Artificielle (IA) 17
3.4.4. Serveur LLM (Ollama) 18
3.4.5. Service de messagerie (ZeroMQ) 19
3.5. Lancement du serveur 21
3.5.1. Prérequis 21
3.5.2. Lancement du serveur 21
3.6. Déploiement 22
3.6.1. Prérequis 22
3.6.2. Configuration de l'instance EC2 22
3.6.3. Compilation et déploiement 23

1. Introduction

1.1. Présentation du logiciel

Le logiciel a été développé pour la Communauté de pratique IA en santé (CPIAS) et l'École de l'intelligence artificielle en santé du CHUM (ÉIAS). Il consiste en une plateforme web permettant de faciliter la collaboration entre les professionnels de la santé et les experts en IA. Cet outil comprend un moteur de recherche permettant de trouver des membres de la CPIAS en fonction de leurs expertises en IA et en santé. De plus, la plateforme offre un système de recommandation d'expertise pour former des équipes multidisciplinaires. Cette application permettra de favoriser l'intégration de l'IA dans le domaine de la santé, en répertoriant et en mettant en relation les acteurs clés.

1.2. Présentation du guide du développeur

Ce guide vise à fournir une compréhension approfondie de l'architecture, des conventions de codage, et des bonnes pratiques associées à l'application. Qu'il s'agisse du développement de l'interface utilisateur ou du serveur, ce document offre des informations pertinentes et des conseils pour chaque technologie utilisée.

2. Interface utilisateur

2.1. Technologies utilisées

L'interface utilisateur du Répertoire des expertises de la CPIAS est développée avec le cadriciel React, le langage Typescript et la bibliothèque Chakra UI. Ces choix technologiques offrent un environnement propice à la création d'applications robustes et faciles à maintenir.

2.1.1. React

React est un cadriciel JavaScript/Typescript qui simplifie la création d'interfaces utilisateur interactives. Basé sur le concept de composants réutilisables, React permet de structurer l'application de manière modulaire. Cela facilite le développement, la maintenance et l'extension de l'interface utilisateur.

2.1.2. Typescript

TypeScript est un sur-ensemble de JavaScript qui ajoute des fonctionnalités de typage statique au langage. L'utilisation de TypeScript apporte des avantages significatifs en termes de détection d'erreurs pendant la phase de développement, ce qui contribue à un code source plus fiable et facile à comprendre.

2.1.3. Chakra UI

Chakra UI est une bibliothèque de composants React qui simplifie la création d'interfaces utilisateur cohérentes et esthétiques. Elle offre des composants préconstruits, configurables et stylisés, permettant un développement plus rapide et une gestion simplifiée de la conception.

2.2. Structure du projet

La structure des dossiers et des fichiers du code source de l'interface utilisateur est détaillée ci-dessous.

- ***node_modules*** : répertoire généré par Node.js, qui stocke les dépendances du projet.
- ***public*** : répertoire des fichiers statiques tels que les images, les documents et les icônes
- ***src*** : répertoire source principal de l'application. Contient les composants, les pages et les fonctions utilitaires
    - ***components*** : répertoire contenant les composants fonctionnels réutilisables.
    - ***models*** : répertoire contenant la définition des interfaces.
    - ***pages*** : répertoire contenant les composants spécifiques à chaque page de l'application.
    - ***utils*** : répertoire contenant les utilitaires ou des fonctions réutilisables
    - ***theme*** : répertoire contenant des fichiers liés au thème de l'application, tels que les couleurs et la configuration du thème Chakra UI.
- ***tsconfig.json*** : Configuration Typescript pour spécifier comment le code doit être compilé.
- ***package.json/package-lock.json*** : Fichiers de configuration pour les dépendances du projet et des scripts associés.
- ***App.tsx*** : Fichier de composant principal
- ***Routes.tsx*** : Fichier contenant la définition des routes de l'application.
- ***.env*** : Fichier de configuration où sont stockés les variables environnement.
- ***.eslintrc.json*** : Configuration pour ESLint, un outil d'analyse statique du code JavaScript et/ou TypeScript.

2.3. Variables environnement

Pour assurer la configuration flexible de l'interface utilisateur du Répertoire des expertises de la CPIAS, trois variables d'environnement sont utilisées. Ces variables fournissent des informations sensibles, telles que les URL du serveur, les clés d'API et le mot de passe du mode administrateur. Un exemple du fichier environnement est inclus dans le code source sous le nom *.env.example*.

Les variables d'environnement peuvent être accédées dans le code en utilisant la syntaxe *process.env.REACT_APP_NOM_VARIABLE*. Un exemple d'utilisation des variables environnement est présenté à l'extrait de code 4.

2.4. Composants fonctionnels

En React, un composant fonctionnel est une fonction qui retourne une partie de l'interface utilisateur devant être rendue. En termes plus techniques, un composant fonctionnel contient du code permettant de retourner un élément JSX (*JavaScript XML*). Les composants fonctionnels sont contenus dans les fichiers .tsx et peuvent accepter des paramètres en entrées..

L'utilisation de composants fonctionnels a pour objectif de diviser l'interface utilisateur en unités autonomes et réutilisables. Un exemple de la syntaxe de la composante *UserGuide* est présenté à l'Extrait de code 1. Dans cet exemple, la composante est définie avec une interface *UserGuideProps*, spécifiant la structure des propriétés attendues. Dans cet exemple, cette interface contient deux chaînes de caractères : un titre et une description. Ces propriétés sont ensuite utilisées dans le rendu de la composante, où elles sont affichées dans un conteneur *Flex*.

```
//...imports...

interface UserGuideProps{ title: string; description: string; }

const UserGuide: React.FC<UserGuideProps> = ({
    title,
    description
}) => {
  return (
     <Flex>
       <Text>{title}</Text>
       <Text>{description}</Text>
     </Flex>
  );
};
```

Extrait de code 1 : Déclaration d'un composant fonctionnel

La documentation officielle des composants et des propriétés offre plusieurs exemples et cas d'utilisation sur ce concept.

2.5. Navigation

La gestion de la navigation dans l'application revêt une importance particulière pour assurer une expérience utilisateur fluide et intuitive. Dans le cas du Répertoire des expertises de la CPIAS, la librairie *React Router* est utilisée pour définir les routes et permettre aux utilisateurs de passer d'une page à l'autre.

2.5.1. Fichier Routes.tsx

Dans le fichier *Routes.tsx*, les routes sont mises en place à l'aide de *React Router*. L'utilisation de *HashRouter* définit le contexte de navigation et simplifie la gestion des routes. Chacune d'elle est déclarée à l'intérieur dans ce fichier et est associée à un composant spécifique, qui est rendu lorsqu'une route est atteinte. Un exemple de déclaration du composant *Router* est présenté à l'Extrait de code 2.

```
//...imports...

const Router: React.FC = () => {
  return (
    <HashRouter>
       <Routes>
          <Route path="/" element={<Navigate to="/accueil" replace />} />
          <Route path="/accueil" element={<HomePage />} />
          <Route path="/membres" element={<MembersPage />} />
          <Route path="*" element={<NotFoundPage />} />
          {/*...Other routes...*/}
       </Routes>
    </HashRouter>
  );
};
```

Extrait de code 2 : Déclaration du fichier *Routes.tsx*

La documentation officielle de *React Router* fournit plusieurs explications sur les concepts fondamentaux de la gestion des routes en React.

2.5.2. Navigation à partir des composants

La bibliothèque *React Router* propose un crochet (*hook*), qui permet à un composant d'accéder à l'état de la navigation de l'application. Ce crochet, nommé *useNavigate*, permet de rediriger un utilisateur vers une autre route de manière programmatique.

```
//...imports...

import { useNavigate } from 'react-router-dom';

const HomePage: React.FC = () => {
  const navigate = useNavigate(); //Déclaration du hook

  return (
    <Button onClick={() => navigate('/membres')}>
      {'Aller à la page des membres'}
    </Button>
  );
};
```

Extrait de code 3 : Utilisation de *useNavigate()*

L'Extrait de code 3 montre une déclaration simplifiée du composant *HomePage*, qui affiche un bouton. Un clic sur ce bouton redirige l'utilisateur vers la page des membres grâce au crochet *useNavigate*.

2.6. Communication avec le serveur

La communication entre l'interface utilisateur et le serveur est cruciale pour assurer le bon fonctionnement de l'application. Dans le Répertoire des expertises de la CPIAS, cette interaction est réalisée à l'aide de la bibliothèque Axios, qui simplifie les requêtes HTTP et assure une communication efficace entre avec le serveur.

```
import axios from 'axios';

const API_HOST = process.env.REACT_APP_SERVER_URL;
const API_KEY = process.env.REACT_APP_API_KEY;

const fetchMembers = async () => {
    try {
        const response = await axios.get(`${API_HOST}/users`, {
            headers: {
                'Authorization': `${API_KEY}`
            }
        });
        console.log(response.data);
    } catch (error) {
        console.error('Error while fetching members: ', error);
    }
};
```

Extrait de code 4 : Utilisation de la méthode *GET* de la bibliothèque Axios

L'utilisation des méthodes les plus fréquemment utilisées du protocole HTTP (*GET, PUT, POST, PATCH, DELETE*) se fait de manière très simple avec Axios. L'Extrait de code 4 présente un exemple d'obtention de la liste des membres, grâce à la méthode *GET*. Cette méthode prend en paramètre *API_HOST*, qui correspond à la variable environnement de l'URL du serveur. Pour augmenter la sécurité, le serveur requiert une clé dans l'en-tête des requêtes. Celle-ci doit être passée à la propriété *Authorization* de l'en-tête. Il est fortement conseillé d'encadrer le code avec une structure *try/catch* afin de gérer les erreurs qui pourraient survenir si le serveur ne répond pas conformément aux attentes.

2.7. Interface réactive

L'interface réactive est essentielle pour garantir une expérience utilisateur optimale, quelle que soit la taille de l'écran utilisé. La bibliothèque Chakra UI facilite la création d'une interface réactive avec une approche basée sur les points de rupture. Les points de rupture par défaut sont présentés au Tableau 1.

Tableau 1 : Points de rupture réactifs de Chakra UI

Point de rupture	unité (em)	unité (px)
base	0	0
sm	30	~480
md	48	~768
lg	62	~992
xl	80	~1280
2xl	96	~1536

Pour créer une interface utilisateur réactive avec Chakra UI, il suffit de spécifier plusieurs valeurs pour une propriété de style. Par exemple, pour rendre la largeur d'un conteneur *Flex* réactive, il est possible de passer plusieurs valeurs, comme présenté à l'Extrait de code 5.

```
<Flex width={{ base: "250px", md: "350px", lg: "500px" }}>
    {/*Content*/}
</Flex>
```

Extrait de code 5 : Déclaration d'un conteneur *Flex* avec une largeur réactive

Cette syntaxe peut être comprise comme suit : lorsque la largeur de l'écran est supérieure à 0 em, la largeur du conteneur *Flex* est fixée à 250px ; si la largeur de l'écran est supérieure à 48 em, la largeur devient 350px ; enfin, si la largeur de l'écran dépasse 62 em, la largeur du conteneur est de 500px.

De manière similaire, il est possible d'afficher un conteneur de manière conditionnelle, selon la taille de l'écran. Dans l'exemple de l'Extrait de code 6, le conteneur *Box* n'est pas affiché lorsque la largeur de l'écran est entre 0em et 62em. À partir d'un écran de 62em, l'affichage est réalisé selon le paramètre *flex*.

```
<Box display={{ base: "none", lg: "flex" }}>
    {/*Content*/}
</Box>
```

Extrait de code 6 : Déclaration d'un conteneur *Box* avec un affichage conditionnel

Les styles réactifs sont détaillés avec plusieurs exemples dans la documentation officielle de Chakra UI.

2.8. Lancement de l'application

2.8.1. Prérequis

Avant de procéder au lancement de l'application, il est nécessaire de s'assurer que les éléments suivants sont correctement installés ou configurés :

- Node.js (version 16.0 et plus)
- Node Package Manager (npm), qui est généralement inclus avec l'installation de Node.js.
- Fichier *.env* avec les variables présentées à la section 2.3.

2.8.2. Lancement de l'application

Les étapes pour lancer l'application sont décrites dans le fichier *README.md* du répertoire *Frontend*.

1. Installer les dépendances du projet en exécutant la commande suivante dans un terminal ouvert dans le répertoire *Frontend* :

```
npm ci
```

2. Lancer la compilation de l'application en exécutant la commande suivante :

```
npm start
```

3. Accéder à l'application en ouvrant le navigateur et en accédant à l'URL http://localhost:3000.

4. Lors du développement, exécuter la commande suivante pour rouler l'analyseur de code statique ESlint pour détecter et régler les défauts dans la qualité du code.

```
npm run lint        //Détecter les défauts
npm run lint:fix    //Régler les défauts
```

2.9. Déploiement

Les étapes de déploiement ci-dessous permettent de déployer l'interface utilisateur sur une instance EC2 d'Amazon Web Services.

2.9.1. Prérequis

- Instance EC2 avec un minimum de 4 Go de mémoire vive
- Nginx installé sur l'instance EC2

2.9.2. Compilation et déploiement

1. Cloner le répertoire Git de l'application dans l'instance EC2.
2. Aller dans le répertoire *./Frontend*

3. Lancer la construction de l'application avec la commande suivante :

```
npm run build
```

4. Copier le contenu du répertoire *./build* vers le répertoire *./var/www/cpias*
5. Lancer Nginx avec la commande suivante :

```
sudo systemctl restart build
```

Un tutoriel complet est présenté dans la vidéo *Deploy a NodeJS React app to AWS EC2*.

3. Serveur

3.1. Technologies utilisées

Lors du développement du serveur du Répertoire des expertises de la CPIAS, une sélection méticuleuse des technologies a été entreprise pour garantir l'efficacité, la robustesse, la maintenabilité et l'évolutivité de l'application développée.

En effet, les technologies suivantes ont été utilisées pour implémenter les différentes fonctionnalités de l'application:

3.1.1. Python (langage de programmation):

Python est un langage de programmation polyvalent et de haut niveau choisi pour sa lisibilité et sa facilité de développement.

3.1.2. Flask (Cadriciel Web):

Flask est un cadriciel web léger et flexible pour Python. Il facilite le développement d'applications web et d'API avec simplicité et évolutivité.

3.1.3. Gunicorn (Serveur WSGI):

Gunicorn (Green Unicorn) est un serveur WSGI (Web Server Gateway Interface ou Interface de Passerelle de Serveur Web) qui sert de pont entre l'application Flask et le serveur web (Nginx). Il aide à gérer efficacement les requêtes concurrentes.

3.1.4. Nginx (Serveur mandataire inversé):

Nginx agit comme un serveur mandataire inversé, transmettant les demandes des clients à l'application Flask servie par Gunicorn. Il améliore la sécurité, l'équilibrage de la charge et gère efficacement le contenu statique.

3.1.5. SQLAlchemy (Base de données ORM):

SQLAlchemy est une bibliothèque ORM (Object-Relational Mapping ou Mappage Objet-Relationnel) pour Python. Elle abstrait les interactions avec les bases de données, ce qui permet de travailler avec des bases de données en utilisant des objets et des requêtes Python.

3.1.6. ChromaDB (Stockage d'Embeddings):

ChromaDB est une base de données d'embeddings (également connue sous le nom de base de données vectorielles) qui stocke les embeddings permettant d'effectuer des recherches par voisinage le plus proche plutôt que par sous-chaîne comme dans une base de données traditionnelle. Cela peut s'avérer crucial pour des tâches telles que les recherches de similarité ou les recommandations.

3.1.7. spaCy (Traitement du langage naturel):

spaCy est une puissante bibliothèque NLP (Natural Language Processing ou Traitement du Langage Naturel) en Python. Elle est utilisée pour des tâches de traitement du langage telles que la tokenisation, l'étiquetage de la partie du discours et la reconnaissance des entités nommées. Elle prend en charge plusieurs langues, dont l'anglais et le français.

3.1.8. LangChain (Interaction avec les Grands Modèles de Langage):

LangChain est un cadriciel conçu pour simplifier la création d'applications utilisant de grands modèles de langage (LLM). Il fournit une interface pour gérer et communiquer avec différents LLMs.

3.1.9. ZeroMQ (bibliothèque de messagerie):

ZeroMQ est une bibliothèque de messagerie légère qui facilite la communication entre différents processus. Dans notre cas, elle permet de connecter le processus Flask avec le processus LLM.

3.1.10. Ollama App (Exécuteur Local de LLMs):

Ollama est une application utilisée pour exécuter localement de grands modèles de langage. Elle a été utilisée pour exécuter le LLM nommé "mistral:instruct".

3.1.11. KeyBert (Extraction de mots-clés):

KeyBERT est une technique d'extraction de mots-clés qui s'appuie sur les embeddings BERT pour créer des mots-clés et des phrases-clés qui sont les plus similaires à un document.

3.1.12. Transformateurs de phrases (calcul des embeddings):

Des transformateurs de phrases ont été utilisés pour calculer des embeddings à partir de textes. Des modèles tels que "all-mpnet-base-v2" et "camembert/camembert-large" ont été utilisés pour générer les représentations vectorielles des phrases.

3.1.13. Deep Translator (traduction de texte):

La bibliothèque Deep Translator est utilisée pour la traduction de textes. Il s'agit d'un outil flexible, gratuit et illimité qui permet de traduire entre différentes langues de manière simple en utilisant plusieurs traducteurs.

Ces technologies contribuent collectivement au développement d'un serveur robuste pour cette application, incorporant des fonctionnalités avancées telles que le traitement du langage naturel, l'interaction de grands modèles de langage et le traitement efficace des données par le biais des embeddings et des bases de données.

3.2. Structure du projet

La structure des dossiers et des fichiers du code source du serveur est détaillée ci-dessous.

- ***database*** : répertoire généré par l'application lors de son premier lancement. Il contient la base de données ***SQLite***.

- ***resources*** : répertoire de fichiers statiques tels que les photos de profils des utilisateurs, le fichier ***users.csv*** générés par le formulaire d'inscription à la CPIAS et le fichier ***users.json*** contenant des informations publiques supplémentaires sur les utilisateurs qui ont été extraites de leurs comptes LinkedIn.
- ***src*** : répertoire source principal du serveur. Il contient tous les classes et fichiers nécessaires au bon fonctionnement de l'application.
    - ***ai.py*** : contient toute la logique du système de recommandation d'experts et d'extraction de mots clés à partir de leurs expertises en utilisant l'intelligence artificielle. Il contient aussi la logique permettant de gérer la base de données vectorielle ***ChromaDB.***
    - ***ai_models.py***: contient un modèle utilisé par le parser de l'output du système de recommandation d'experts.
    - ***app.py*** : contient toute la logique du serveur ***Flask***, qui est responsable du traitement de toutes les requêtes reçues du client.
    - ***database.py*** : contient toute la logique liée à la gestion de la base de données ***SQLite*** contenant les profils d'experts, ainsi que le fichier ***users.csv***.
    - ***decorators.py*** : contient un décorateur permettant de vérifier que l'en-tête "Authorization" de la requête entrante contient une clé API valide.
    - ***gunicorn_config.py*** : contient toute la configuration utilisée par ***Gunicorn*** qui sert de pont entre le serveur ***Flask*** et ***Nginx***.
    - ***server.conf*** : contient toute la configuration du serveur ***Nginx***.
    - ***server.service*** : c'est un fichier de configuration d'unité permettant de créer un service de serveur contrôlé et supervisé par ***systemd***, qui se lance au démarrage d'***Ubuntu***.
    - ***settings.py*** : contient tous les paramètres utilisés par le serveur dans un seul endroit central.
    - **wsgi.py** : fichier utilisé par ***Gunicorn*** pour initialiser et lancer le serveur ***Flask***.
- ***templates*** : répertoire qui est traditionnellement utilisé par le serveur ***Flask*** pour stocker des modèles *HTML* pour le rendu de contenu dynamique dans les applications web. Il contient une page ***index.html*** utilisée pour montrer le bon fonctionnement du serveur.
- ***tls_ssl*** : répertoire contenant les fichiers ***self-signed.conf*** et ***ssl-params.conf*** qui sont généralement utilisés par ***Nginx*** pour configurer les paramètres ***SSL/TLS***, en particulier dans le contexte de la mise en place de connexions sécurisées pour **HTTPS**.
    - ***self-signed.conf :*** ce fichier contient les paramètres de configuration d'un certificat SSL/TLS auto-signé. À utiliser seulement dans des environnements de **développement** ou de **test** pour permettre des connexions ***HTTPS*** cryptées. Dans un environnement de **production**, il est fortement recommandé d'utiliser un certificat signé par une autorité de certification (CA) de confiance. Le fichier ***self-signed.conf*** inclut des directives permettant de spécifier les chemins d'accès au certificat auto-signé et à la clé privée.
    - ***ssl-params.conf :*** ce fichier contient les paramètres ***SSL/TLS*** qui renforcent la sécurité de la connexion ***HTTPS***. Il contient les directives permettant de configurer

divers paramètres SSL, tels que les protocoles cryptographiques préférés, les algorithmes de chiffrement et d'autres paramètres liés à la sécurité. Il est essentiel de configurer correctement les paramètres SSL pour garantir la sécurité et la solidité de la communication cryptée entre le client et le serveur. Les paramètres de ce fichier visent à optimiser la position de sécurité de l'implémentation ***SSL/TLS***.

- ***vector_store*** : répertoire généré par l'application lors de son premier lancement. Il contient la base de données vectorielle ***ChromaDB.***
- ***.env*** : Fichier de configuration où sont stockés les variables environnement.
- ***requirements.txt*** : fichier contenant toutes les dépendances qui doivent être installées préalablement dans le système hôte pour que le serveur puisse fonctionner correctement.
- ***setup.sh*** : un script qui automatise l'installation du serveur à partir de zéro sur une instance ***Ubuntu***.

3.3. Variables environnement

Comme indiqué ci-dessus pour l'interface utilisateur, l'utilisation de variables d'environnement offre une approche flexible et sécurisée pour configurer et contrôler le comportement d'un serveur sans nécessiter des modifications directes dans le code source. Cela facilite la gestion, la maintenance et l'évolutivité des applications serveur.

En effet, le serveur de l'application a besoin de certaines variables d'environnement afin de fonctionner correctement. Ces variables doivent être enregistrées dans un fichier ***.env*** à la racine du projet. Un exemple du fichier environnement est inclus dans le code source sous le nom ***.env.example***.

Les variables d'environnement peuvent être accédées dans le code en utilisant par exemple la bibliothèque ***dotenv*** comme suit:

```
import os
from dotenv import load_dotenv
from flask import request, jsonify

load_dotenv()


def require_api_key(view_func):
   def decorated(*args, **kwargs):
       api_key = request.headers.get('Authorization')

       if api_key is not None and api_key == os.getenv('API_KEY'):
           return view_func(*args, **kwargs)
       else:
           return jsonify({'error': 'Unauthorized'}), 401

   return decorated
```

Extrait de code 7 : Accès aux variables d'environnement via la bibliothèque ***dotenv.***

3.4. Architecture du serveur

Le serveur a été développé sur une base modulaire pour faciliter l'extensibilité future du système. En fait, chaque composant du serveur fonctionne indépendamment et peut être facilement remplacé par un nouveau composant si cela est nécessaire.

Le serveur se compose principalement des modules suivants:

3.4.1. Serveur Web

Dans l'architecture du serveur web mise en œuvre, Flask, Gunicorn et Nginx collaborent pour assurer un fonctionnement robuste et efficace.

1. **Flask :** il est le cadriciel web responsable de la gestion de la logique commerciale de l'application. Il reçoit les requêtes HTTP entrantes, les traite et génère les réponses appropriées. Flask est léger et conçu pour être flexible, ce qui en fait un excellent choix pour le développement d'applications web et d'API.

2. **Gunicorn :** il agit comme un serveur WSGI. Son rôle principal est de servir d'interface entre l'application Flask et le serveur web externe (dans ce cas, Nginx). Gunicorn gère plusieurs processus de travailleurs afin de traiter efficacement les requêtes simultanées. Lorsqu'une requête est reçue par Nginx, elle est transmise à Gunicorn qui, à son tour, la transmet à l'application Flask s'exécutant dans l'un de ses processus de travailleurs.

3. **Nginx :** il sert de serveur mandataire inversé, agissant en tant qu'intermédiaire entre les clients externes et l'application Flask. Il est configuré pour écouter les requêtes entrantes sur les ports 80 (HTTP) et 443 (HTTPS). Lorsqu'un client envoie une requête, Nginx la reçoit et décide comment la traiter en fonction des blocs de serveur définis dans sa configuration.

    a. Pour HTTPS (port 443), le premier bloc serveur gère les connexions sécurisées. Il inclut les configurations SSL (référencées dans ***self-signed.conf*** et ***ssl-params.conf***), spécifie le nom du serveur (domaine ou IP), et définit un bloc de localisation pour router les requêtes vers l'application Flask via Gunicorn. La directive ***proxy_pass*** de ce bloc de localisation dirige les requêtes vers le socket Unix où Gunicorn est à l'écoute (voir extrait de code 8).

    b. Pour HTTP (port 80), le deuxième bloc serveur est configuré pour rediriger tout le trafic HTTP vers la version sécurisée HTTPS. Cela renforce la sécurité en garantissant que la communication est cryptée (voir extrait de code 8).

En résumé, le flux de travail se présente comme suit:

1. Un client adresse une requête HTTP ou HTTPS au serveur Nginx.
2. Nginx, en fonction de sa configuration, décide comment traiter la demande.
    - Pour HTTP, il redirige vers la version HTTPS.
    - Pour HTTPS, il transmet la demande à l'application Flask via Gunicorn.
3. Gunicorn, agissant en tant que serveur WSGI, dirige la requête vers le processus travailleur approprié de Flask.
4. Flask traite la demande, exécute la logique nécessaire et génère une réponse.

5. Gunicorn renvoie la réponse à Nginx.
6. Nginx délivre à son tour la réponse au client.

```
server {
       listen 443 ssl;
       listen [::]:443 ssl;

       # Change this if you have a CA signed cert.
       include snippets/self-signed.conf;
       include snippets/ssl-params.conf;

       # Define the server_name (your domain or IP)
       # server_name your_domain.com www.your_domain.com;

       # Location block for "/*" routes
       location / {
           proxy_pass http://unix:/home/ubuntu/project_4/Backend/src/server.sock;
           include /etc/nginx/proxy_params;
       }
}

server {
   listen 80;
   listen [::]:80;

   # Redirect all HTTP traffic to HTTPS
   return 301 https://$host$request_uri;
}
```

Extrait de code 8: Configuration du serveur mandataire inversé Nginx.

3.4.2. Base de données

Il y a deux bases de données : une base de données ***SQLite*** stockant les profils d'experts et une base de données ***ChromaDB*** contenant les embeddings des compétences des experts. La base de données *SQLite* prend en charge les requêtes des clients, tandis que la base de données *ChromaDB* est utilisée par le module d'intelligence artificielle pour les recherches de similarités.

1. **Base de données *SQLite* (profils d'experts)**

   Requêtes des clients: Le serveur web utilise la bibliothèque ***SQLAlchemy*** pour interagir avec la base de données *SQLite* lorsqu'il répond aux requêtes des clients. Par exemple, lorsqu'un client demande des informations sur un expert spécifique ou une liste d'experts correspondant à certains critères, le serveur web exécute des requêtes *SQL* à l'aide de *SQLAlchemy* pour extraire les données pertinentes de la base de données *SQLite*. Ces données peuvent inclure des détails sur le profil de l'expert, ses compétences et d'autres informations pertinentes.

2. **Base de données *ChromaDB* (Embeddings de compétences expertes)**

    a. Module IA (recherche de similarité): La base de données *ChromaDB* répond exclusivement aux besoins du module IA en matière de recherche de similarité. *ChromaDB* contient des embeddings de compétences d'experts, qui sont des représentations vectorielles des compétences apprises à partir des profils d'experts. Lors d'une recherche de similarité, le module d'intelligence artificielle interroge *ChromaDB* pour trouver des experts dont les compétences sont proches ou similaires à une requête donnée, ce qui permet de recommander efficacement des experts ayant une expertise similaire.

    b. En isolant la base de données *ChromaDB* pour les recherches de similarité, on crée un espace dédié au module d'intelligence artificielle pour qu'il puisse effectuer ses tâches spécialisées de manière efficace. Cette isolation permet au module IA de se concentrer sur les calculs de similarité sans affecter la charge de travail transactionnelle de la base de données *SQLite* utilisée pour les interactions avec les clients.

3.4.3. Intelligence Artificielle (IA)

Les fonctionnalités principales de ce module se résume comme suit:

1. **Recommandations d'experts**

    - Génère des recommandations d'experts basées sur des requêtes ou des questions données en entrée à un prompt adapté à cette tâche.
    - Utilise le LLM nommé ***mistral:instruct*** (***7B*** de paramètres) qui est une version peaufinée pour mieux traiter des requêtes en forme d'instruction. Il est basée sur le LLM ***Mistral-7B-v0.1.***
    - Gère la récupération et le traitement des compétences et des recommandations d'experts à partir du fichier CSV généré par le formulaire d'inscription à la CPIAS.
    - Gère le stockage et la récupération des profils d'experts à l'aide d'une base de données vectorielle calculée en utilisant le transformateur de phrase ***all-mpnet-base-v2*** qui peut être utilisé pour des tâches telles que le clustering ou la recherche sémantique.

2. **Extraction de mots-clés**

    - Extrait les mots-clés les plus pertinents à partir des compétences des experts.
    - Demande au LLM ***mistral:instruct*** d'extraire les mots-clés en lui fournissant les compétences de l'expert en entrée à un prompt adapté à cette tâche.
    - Une fois les mots-clés du LLM sont obtenus, on les passe au ***KeyBert*** qui utilise la similitude du cosinus pour ne retenir que les mots-clés les plus pertinents (les trois meilleurs mots-clés selon leurs scores de similitude). Pour ce faire, KeyBert utilise le transformateur de phrase ***camembert/camembert-large*** pour calculer les embeddings des compétences de l'expert et des mots-clés candidats générés par le LLM. Le choix de ce transformateur de phrases s'explique par le fait qu'il est entraîné à traiter des textes rédigés en français.

3. **Traitement du langage naturel :** Utilise spaCy pour des tâches de traitement du langage naturel, y compris la tokenisation du texte et la suppression des mots superflus.

4. **Traduction :** Utilise l'API Google Translator pour la traduction de textes entre les langues. Etant donné que les textes contenus dans la base de données CSV sont parfois en anglais ou en français. On les traduit systématiquement en anglais pour standardiser les textes, et aussi parce que le LLM ***mistral:instruct*** est plus performant en anglais qu'en français. Ensuite, une fois la requête traitée, la réponse finale est traduite en français avant d'être renvoyée au client.

3.4.4. Serveur LLM (Ollama)

Le serveur LLM est assuré par l'application Ollama qui doit être installée sur la machine hôte. Elle sert d'interface conviviale pour l'exécution locale des LLMs, en fournissant un moyen direct d'interagir avec différents modèles et en faisant abstraction des complexités de l'installation du serveur. Elle agit comme un intermédiaire entre le module IA et les LLMs permettant de traiter efficacement des requêtes locales.

Le choix d'Ollama s'explique par les points suivants:

1. **Exécution locale :** il permet d'exécuter localement le LLM ***mistral:instruct***, en agissant comme un serveur autonome, indépendant du serveur web.

2. **Fonctionnement en tant que serveur :** Ollama fonctionne comme un serveur qui peut recevoir des requêtes de divers composants, tels que le module IA. Il est conçu pour traiter les requêtes liées aux tâches de traitement de langues naturelles.

3. **Compatibilité avec plusieurs LLM :** Ollama est compatible avec une liste de LLMs prédéfinis, dont Mistral et Llama2. Ces LLMs peuvent être extraits d'un dépôt, ce qui permet une certaine flexibilité dans le choix du modèle de langage à utiliser. Pour ce faire il suffit d'exécuter la commande suivante:

```
ollama pull <nom_du_model>
```

   Une liste des LLMs pris en charge par Ollama est disponible dans ce lien: https://ollama.ai/library

   Afin d'utiliser un nouveau modèle téléchargé par Ollama, il suffit de changer cette variable dans le fichier ***settings.py :***

```
SERVER_SETTINGS = {
   # autres paramètres du serveur...

   "expert_recommendation_llm_model": "nom_du_nouveau_LLM",

   # suite des paramètres du serveur...
}
```

Extrait de code 9: Mettre à jour le LLM utilisé par le serveur.

On peut à tout moment savoir la liste des LLMs téléchargés localement sur la machine hôte en exécutant la commande suivante:

```
ollama list
```

En effet, Ollama facilite le passage d'un LLM à l'autre sans modifier le code source du module IA. Pour ce faire, il fournit un dépôt de LLM compatibles qui peuvent être facilement extraits et utilisés. Il permet d'expérimenter différents modèles sans avoir à modifier le code en profondeur.

4. **Modèle requête-réponse :** L'interaction entre le module IA et Ollama suit un modèle requête-réponse. Le module IA envoie des requêtes à Ollama, en spécifiant les tâches de traitement linguistique, et Ollama traite ces requêtes et renvoie les réponses appropriées.

   Ollama est accessible sur cette adresse: http://localhost:11434 comme le montre l'extrait du code suivant:

```
@staticmethod
def __get_expert_recommendation_llm(expert_recommendation_llm_model: str, temperature:
float = 0.0) -> Ollama:
   # documentation...

   return Ollama(base_url='http://localhost:11434',
                 model=expert_recommendation_llm_model,
                 temperature=temperature)
```

Extrait de code 10: Mettre à jour le LLM utilisé par le serveur.

3.4.5. Service de messagerie (*ZeroMQ*)

La principale raison d'utiliser ***ZeroMQ*** est que le module AI utilise généralement la bibliothèque ***PyTorch***, qui utilise à son tour la bibliothèque ***CUDA*** de Nvidia. Le runtime CUDA ne supporte pas la méthode de lancement "***fork***", donc les méthodes de lancement "***spawn***" ou "***forkserver***" sont nécessaires pour utiliser CUDA dans les sous-processus.

Et comme le serveur web utilise ***Gunicorn*** qui crée des sous-processus travailleurs à l'aide de la méthode "fork", il n'y avait pas d'autre choix que de lancer le module IA dans un processus séparé indépendant du processus du serveur web, d'où la nécessité d'utiliser une solution de communication inter-processus pour synchroniser le processus du serveur web et le processus IA qui s'exécute en parallèle.

ZeroMQ est un choix idéal pour synchroniser des processus séparés en raison de sa conception légère et asynchrone qui prend en charge différents modèles de messagerie. Il est multiplateforme, indépendant des langages de programmation et connu pour ses performances élevées, sa fiabilité et sa tolérance aux pannes. Grâce à son architecture décentralisée, à son évolutivité et au soutien de la communauté, ZeroMQ simplifie la communication inter-processus, ce qui le rend efficace et polyvalent pour un large éventail d'applications.

Le module IA utilise ZeroMQ comme pont de communication pour permettre au serveur Flask ou à la base de données d'invoquer dynamiquement les méthodes de la classe LLM. La classe LLM possède plusieurs méthodes privées et publiques qui exécutent diverses tâches de traitement du langage naturel.

Voyons comment fonctionne l'intégration de ZeroMQ.

1. **Initialisation du contexte et de la socket**

   La méthode ***__init_zmq*** initialise un contexte ZeroMQ et lie une socket à une adresse spécifiée (***SERVER_SETTINGS["zeromq_response_address"]***).

2. **Invocation dynamique de méthodes**

   La méthode ***__call_method*** invoque dynamiquement une méthode de la classe LLM en fonction du nom de la méthode et des arguments fournis. Elle utilise la fonction ***getattr*** de Python pour rechercher et appeler dynamiquement la méthode.

3. **Boucle de traitement**

   La méthode ***start_llm_processing*** entre dans une boucle de traitement qui écoute continuellement les demandes du serveur Flask ou de la base de données via ZeroMQ. Elle reçoit une requête codée en JSON, extrait le nom de la méthode et les arguments, et invoque dynamiquement la méthode correspondante à l'aide de ***__call_method***. Le résultat ou le message d'erreur est ensuite renvoyé au demandeur.

4. **Arrêt du processeur LLM**

   La méthode ***stop_llm_processing*** arrête le processeur LLM de manière élégante. Elle attribue la valeur ***False*** à l'indicateur ***is_available***, ferme le socket ZeroMQ et met fin au contexte ZeroMQ.

5. **Méthode de requête**

   La méthode statique ***query_llm*** permet aux composants externes d'interagir avec la classe LLM. Elle envoie une requête codée en JSON au processus LLM en utilisant un socket ZeroMQ séparé (***zeromq_request_address***) et reçoit la réponse.

6. **Comment ZeroMQ est utilisé**

   - Le socket ***ZeroMQ REP*** dans la classe LLM écoute les demandes.
   - Le serveur Flask ou la base de données utilise une socket ***REQ*** pour envoyer des demandes codées en JSON à la classe LLM.
   - La classe LLM reçoit les demandes, invoque dynamiquement la méthode spécifiée et renvoie le résultat ou le message d'erreur.

7. **Exemple d'utilisation**

```
# Example: Query for Keywords
arguments = ['document_text']
result = LLM.query_llm('get_keywords', arguments)
```

```
print(result)
```

8. **Notes**

- Le dictionnaire ***SERVER_SETTINGS*** contient les paramètres de configuration, y compris les adresses ZeroMQ:

```
SERVER_SETTINGS = {
   # autres paramètres du serveur...

   "zeromq_request_address": "tcp://localhost:5555",
   "zeromq_response_address": "tcp://*:5555"
}
```

- L'indicateur ***is_available*** est utilisé pour vérifier si le LLM a été initialisé avec succès avant de traiter les demandes.
- La gestion des erreurs est implémentée pour diverses exceptions potentielles.
- La méthode ***start_llm_processing*** tourne en boucle jusqu'à ce que ***is_available*** prenne la valeur False (indiquant un arrêt).

3.5. Lancement du serveur

3.5.1. Prérequis

Avant de procéder au lancement du serveur, il est nécessaire de satisfaire les prérequis suivants:

- Système d'exploitation: Ubuntu 22.04 avec une architecture 64 bits.
- Spécifications matérielles:
    - Avoir au minimum 60 Go d'espace de stockage.
    - Avoir au minimum 16 Go de RAM.
    - Avoir une carte graphique Nvidia compatible avec la bibliothèque CUDA avec au minimum 16 Go de VRAM.
- Python 3.10.
- Fichier ***.env*** avec les variables présentées à la section 3.3.

3.5.2. Lancement du serveur

Afin de démarrer le serveur dans un environnement de développement, il faut suivre les étapes suivantes:

1. Créer un environnement virtuel à la racine du projet en exécutant la commande suivante dans un terminal ouvert dans le répertoire ***Backend*** :

```
python -m venv venv
source venv/bin/activate
```

2. Installer les dépendances du serveur:

```
pip install -r requirements.txt
```

3. Installer l'application ***Ollama*** afin de pouvoir rouler le LLM localement:

```
curl https://ollama.ai/install.sh | sh
```

4. Télécharger le LLM ***mistral:instruct :***

```
ollama pull mistral:instruct
```

5. Lancer le serveur en exécutant la commande suivante dans un terminal ouvert dans le répertoire ***src:***

```
sudo ../venv/bin/python wsgi.py
```

6. Accéder à serveur en ouvrant le navigateur et en accédant à l'URL http://localhost:80.

3.6. Déploiement

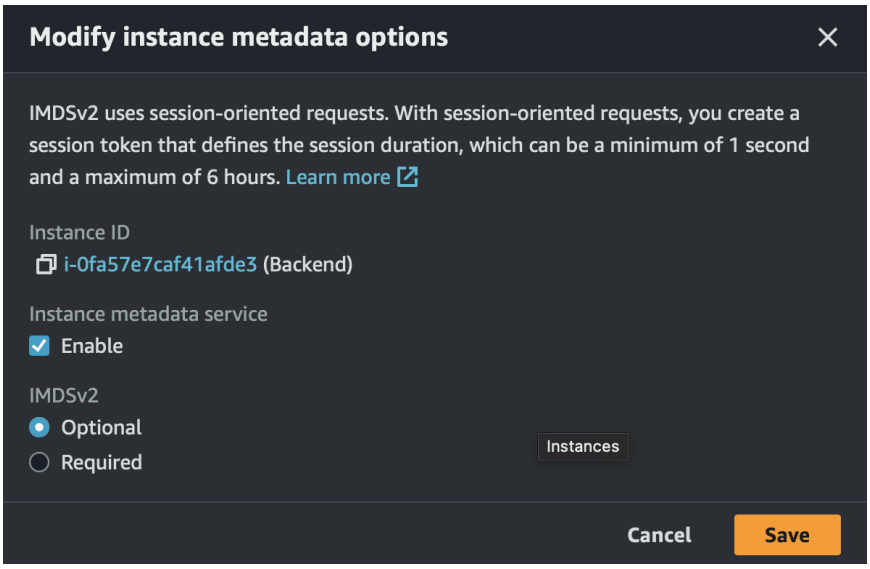
Les étapes de déploiement ci-dessous permettent de déployer le serveur sur une instance EC2 d'Amazon Web Services.

3.6.1. Prérequis

- Instance EC2 de type ***g4dn.xlarge*** ou supérieure.
- Système d'exploitation: ***Ubuntu Server 22.04 LTS*** (64 bits).
- Espace de stockage: ***60 Go.***

3.6.2. Configuration de l'instance EC2

1. Configuration des métadonnées de l'instance: les métadonnées de l'instance doivent être configurées de la manière suivante afin de permettre au script d'installation de récupérer automatiquement le nom d'hôte public de l'instance nécessaire à la génération des certificats ***TLS-SSL*** auto-signés:



2. Configuration des règles entrantes: l'instance doit avoir les ports ouverts suivants:
   - ***Port 22***, protocole ***TCP***: pour se connecter à l'instance en ***SSH***.
   - ***Port 443***, protocole ***TCP***: pour communiquer avec le serveur en ***HTTPS***.

3.6.3. Compilation et déploiement

1. Cloner le répertoire Git de l'application dans l'instance EC2.

```
git clone https://adresse_du_répertoire_du_projet.git
```

2. Aller dans le répertoire *Backend.*

3. Lancer l'installation du serveur avec la commande suivante :

```
sudo ./setup.sh
```

4. Redémarrer l'instance.

5. Accéder à serveur en ouvrant le navigateur et en accédant à l'URL correspondant à l'adresse publique IPv4 de l'instance EC2 créée.