Intégration Azure OpenAI dans django

Pour configurer et paramétrer le service Azure OpenAI, qui fait appel à l’api OpenAI, nous devons suivre plusieurs étapes clés :

1.Installation de dépendance pour l’intégration du service dans Django

Installer la bibliothèque OpenAI avec la commande suivante :

<pip install openai>

2. Création d'un compte Azure :

Tout d'abord, il est nécessaire de créer un compte sur le portail Azure. Cela permettra d'accéder à tous les services Azure, y compris Azure OpenAI.

3. Création d'une Instance Azure OpenAI :

Une fois le compte créé, l'étape suivante consiste à créer une instance d'Azure OpenAI. Cela implique de naviguer dans le portail Azure, de sélectionner l'option pour créer une nouvelle ressource et de choisir Azure OpenAI parmi les options disponibles.

4. Définition du modèle choisi :

Lors de la création de cette instance, il faut définir le modèle souhaité. Dans notre cas, nous optons pour le modèle GPT4-o. Cette étape est essentielle car elle détermine les capacités du service d'IA que nous allons utiliser.

5. Récupération des informations de l'instance :

Après avoir créé l'instance, il est nécessaire de noter certaines informations clés qui seront utilisées dans notre application :

• API version

• API key

• Nom du déploiement

• Nom de la ressource

6. Intégration dans l'Application Django :

Pour faire appel au modèle dans notre application Django, nous allons récupérer ces informations en utilisant les variables d'environnement. Voici un exemple de code à inclure dans le fichier views.py de Django (les variables étant à inclure pour la sécurité dans un fichier <.env>:

```
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
AZURE_OPENAI_ENDPOINT = os.getenv("AZURE_OPENAI_ENDPOINT")
CHAT_COMPLETIONS_DEPLOYMENT_NAME = os.getenv("CHAT_COMPLETIONS_DEPLOYMENT_NAME")
API_VERSION = os.getenv("api_version")
```

insertion des informations Azure pour la connexion au modèle déployé dans le views.py de Django

7/ . Définition d’un prompt à insérer dans le message à envoyer à l’api

```
messages = [
{ "role": "user", "content": prompt1 },
{ "role": "assistant",
"content": "Je peux vous proposer une recette avec ces ingrédients. Veuillez patienter un moment pendant que je génère la recette..."}]
```

inclure une variable prompt définie en valeur de la clé 'content' de notre rôle

messages : Liste des messages qui simulent une conversation entre l'utilisateur et l'assistant.

- **role :** Indique le rôle de l'expéditeur du message. Les valeurs possibles sont :
  - user : Message de l'utilisateur.
  - assistant : Réponse de l'assistant.
- **content :** Contenu du message.
  - prompt1 : Variable contenant la demande de l'utilisateur (par exemple, une demande de recette dans notre cas).
  - Réponse préliminaire de l'assistant **:** Un message indiquant que l'assistant génère une recette en réponse à la demande.

La définition du rôle assistant est facultative, cela est fait afin de créer un message d’attente en attendant la réponse de l’api.

8/. Définition des informations à envoyer à AzureOpenAI avec ‘openai.ChatCompletion’

```
response = openai.ChatCompletion.create(
            engine=deployment_name,
            messages=messages,
            max_tokens=800,
            temperature=0.7,
            top_p=0.95,
            frequency_penalty=0,
            presence_penalty=0)

text = response['choices'][0]['message']['content'].strip()
```

Appel à l'api avec openai.ChatCompletion.create

Avec openai.ChatCompletion.create nous faisons un appel à l'API ChatCompletion d'OpenAI pour générer une réponse basée sur les messages fournis.

Nous pouvons définir les paramètres pour adapter sa réponse :

- **max_tokens**=800

Description : Ce paramètre définit le nombre maximum de tokens (unités de texte) que la réponse générée peut contenir. Ici, il est fixé à 800, ce qui permet d'obtenir une réponse relativement longue.

Note : Un token peut être un mot entier ou une partie de mot, donc 800 tokens ne correspondent pas nécessairement à 800 mots.

- **temperature**=0.7

Description : La température contrôle la créativité de la réponse. Une valeur plus élevée (proche de 1) rend la sortie plus variée et imprévisible, tandis qu'une valeur plus basse (proche de 0) rend la réponse plus déterministe et conservatrice.

Valeur choisie : 0.7, ce qui permet d'avoir un bon équilibre entre créativité et cohérence.

- **top_p**=0.95

Description : Ce paramètre, également connu sous le nom de "nucleus sampling", permet de contrôler la diversité des réponses. En définissant top_p à 0.95, vous indiquez au modèle de ne considérer que les tokens qui représentent 95 % de la probabilité cumulée, ce qui peut générer des réponses plus variées tout en maintenant la pertinence.

Relation avec la température : top_p et temperature peuvent être utilisés ensemble pour influencer la créativité et la cohérence des réponses.

- **frequency_penalty**=0

Description : Ce paramètre détermine combien le modèle doit éviter de répéter les mêmes tokens dans la réponse. Une valeur de 0 signifie qu'il n'y a aucune pénalité pour la répétition, ce qui signifie que le modèle peut répéter des mots ou des phrases sans restriction.

presence_penalty=0

- Description : Ce paramètre contrôle l'introduction de nouveaux tokens. Une valeur de 0 signifie qu'il n'y a aucune pénalité pour introduire de nouveaux concepts ou idées dans la réponse. Cela permet au modèle d'être plus libre dans sa génération.

Retour de l’api :

Le retour de l’api est contenu dans cette variable response, qu’il nous faudra ensuite extraire et nettoyer pour avoir uniquement les informations désirées et transmettre cette nouvelle variable ‘text’ au template de notre view Django.