



# Guide de référence Backend

## [Introduction](#)

### [Structure du Projet](#)

### [Composants Principaux](#)

#### [1. main.py](#)

#### [2. core/config.py](#)

#### [3. core/security.py](#)

#### [4. core/token\\_manager.py](#)

#### [5. endpoints/classify\\_intents.py](#)

#### [6- endpoints/general\\_qst.py](#)

#### [7- endpoints/general\\_v1.py](#)

#### [8- endpoints/request\\_data.py](#)

#### [9- utils/file\\_watcher.py](#)

#### [10- utils/logging\\_config.py](#)

#### [11- schemas.py](#)

### [Mécanisme de Fonctionnement des Appels API](#)

## [Modèles NLP et LLM](#)

### [Modèles utilisés](#)

### [Implémentation](#)

#### [init\\_models.sh](#)

### [services/functions.py](#)

#### [Chargement des variables d'environnement](#)

#### [Chargement des modèles NLP](#)

#### [Fonctionnalités principales](#)

### [Méthodologie de génération d'Embeddings](#)

### [gen\\_embed.py](#)

#### [Description des Fonctions](#)

#### [Instructions d'Utilisation dans le Terminal](#)

### [Méthodologie génération de tokens pour les datasets](#)

### [token\\_gen.py](#)

#### **[Fonctions](#)**

#### **[Utilisation du script dans le terminal](#)**

### [Déploiement et Lancement de l'API](#)

#### [Explication des Options d' `uvicorn` :](#)

## [Déploiement avec Docker \(Pour la production\)](#)

### [Sans construire d'image](#)

#### [Description des fonctions dans le Dockerfile :](#)

#### [Gestion des erreurs](#)

### [Mise à jour des données pour la dataset de requête de données](#)

#### [Comment utiliser la fonction update\\_tags.py](#)

#### [Exemple d'utilisation](#)

### [Exécution des scripts](#)

## [Utilisation de l'API pour d'autres portails](#)

### **[Simulation pour le portail Academia Raqmya :](#)**

#### [1 - Collecte de données](#)

#### [2 - Vectorisation de données \(Embeddings\)](#)

#### [3 - Génération de token](#)

#### [4 - Exemple de requête :](#)

## [API Docs](#)

### [Introduction](#)

#### **[Conventions](#)**

### **[POINTS DE TERMINAISON PRIS EN CHARGE](#)**

### [Codes de statut](#)

#### [Code de réussite](#)

#### [Codes d'erreur](#)

## [Points de terminaison \(Endpoints\)](#)

### [Exemples de code](#)

Pour le portail [data.gov.ma](https://data.gov.ma) vous pouvez utiliser les trois `endpoints` comme ce qui suit:

[Endpoints:](#)

Pour d'autre dataset ou portails vous pouvez utiliser cet `endpoint` :

[Endpoint:](#)

## Introduction

Cette documentation détaille le code et le mécanisme de l'application FastAPI. FastAPI est un framework web moderne, rapide (haute performance) pour la création d'APIs avec Python 3.6+ basé sur les annotations de type standard.

## Structure du Projet

Voici une vue d'ensemble de la structure de notre projet FastAPI :

```
.
├── app.log
├── config.env
├── core
│   ├── config.py
│   ├── __init__.py
│   ├── security.py
│   └── token_manager.py
├── datasets
│   ├── data_ar.json
│   ├── data_fr.json
│   └── tags.json
├── docker-compose.yml
├── Dockerfile
├── embeddings
│   ├── faiss_answers_ar.faiss
│   ├── faiss_answers_fr.faiss
│   └── faiss_nom_tags_paraphrase_multi.faiss
├── endpoints
│   ├── classify_intents.py
│   ├── general_qst.py
│   ├── general_v1.py
│   ├── __init__.py
│   └── request_data.py
├── folder_structure.txt
├── gen_embed.py
├── init_models.sh
├── __init__.py
├── main.py
├── requirements.txt
├── run_api.sh
├── schemas.py
├── services
│   ├── functions.py
│   ├── __init__.py
├── token_gen.py
├── tokens.env
├── update_tags.py
└── utils
    ├── file_watcher.py
    ├── __init__.py
    └── logging_config.py
```

## Composants Principaux

## 1. main.py

Ce fichier est le point d'entrée principal de l'application FastAPI. Il configure l'application, définit les middlewares, gère les exceptions, et inclut les différents routeurs. Voici un aperçu des principales fonctionnalités :

1. Configuration initiale : Chargement de la configuration et initialisation des tokens `tokens.env` et `config.env`.
2. Gestion du cycle de vie : Utilisation de `@asynccontextmanager` pour gérer le démarrage et l'arrêt de l'application.
3. Gestion des exceptions : Définition d'un gestionnaire personnalisé pour les exceptions HTTP.
4. Middleware de journalisation : Enregistrement de l'utilisation de la mémoire avant et après chaque requête.
5. Routes de base : Définition des routes pour la vérification de l'état de l'API ( `"/health"` ) et la racine ( `"/"` ).
6. Configuration CORS : Mise en place du middleware CORS pour gérer les requêtes `cross-origin`.
7. Avant et après chaque requête, l'application enregistre l'utilisation de la mémoire du serveur à des fins de diagnostic et de surveillance.
8. Inclusion des routeurs : Ajout des différents routeurs pour les fonctionnalités spécifiques de l'API.
9. Lancement de l'application : Utilisation `uvicorn` pour exécuter l'application si le script est exécuté directement.

Ce fichier joue un rôle crucial dans la structuration et le fonctionnement global de l'API.

```
from fastapi import FastAPI, Request, HTTPException
from fastapi.responses import JSONResponse
from core.config import load_configuration, initialize_tokens, start_file_watcher
from fastapi.middleware.cors import CORSMiddleware
from endpoints.general_qst import router as general_qst_router
from endpoints.request_data import router as request_data_router
from endpoints.general_v1 import router as general_v1_router
from endpoints.classify_intents import router as classify_intents_router
import threading
import asyncio
from contextlib import asynccontextmanager
import psutil
from utils.logging_config import logger

# Création d'une instance de l'application FastAPI
app = FastAPI()

# Middleware pour la gestion des CORS (Cross-Origin Resource Sharing)
# Utilisé pour contrôler quelles ressources peuvent être partagées entre différentes origines

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Charger la configuration de l'application
    await load_configuration()
    # Initialiser les tokens d'authentification ou autres
    await initialize_tokens()

    # Lancer un thread pour surveiller les changements de fichiers
    watcher_thread = threading.Thread(target=lambda: asyncio.run(start_file_watcher()), daemon=True)
    watcher_thread.start()

    yield # Le yield indique que l'application est en cours d'exécution

    # Des actions de nettoyage peuvent être placées ici si nécessaire
    logger.info("Application is cleaning up resources.") # Log de nettoyage des ressources
```

```

# Définir la durée de vie de l'application FastAPI
app = FastAPI(lifespan=lifespan)

# Gestionnaire d'exception personnalisé pour les erreurs HTTP
@app.exception_handler(HTTPException)
async def custom_http_exception_handler(request: Request, exc: HTTPException):
    # Enregistrer les erreurs HTTP dans les logs avec l'adresse IP du client
    logger.error(f"HTTP error: {exc.detail} from IP: {request.client.host}")
    if exc.status_code == 400:

        return JSONResponse(
            status_code=exc.status_code,
            content={"message": "Invalid request data"},
        )
    elif exc.status_code == 500:

        return JSONResponse(
            status_code=exc.status_code,
            content={"message": "Internal Server Error"},
        )
    else:
        # Pour toutes les autres erreurs HTTP, renvoyer le message d'erreur par défaut
        return JSONResponse(
            status_code=exc.status_code,
            content={"message": exc.detail},
        )

# Middleware pour enregistrer l'utilisation de la mémoire avant et après chaque requête HTTP
@app.middleware("http")
async def log_memory_usage(request: Request, call_next):
    # Enregistrer l'utilisation de la mémoire avant l'appel de l'API
    mem = psutil.virtual_memory()
    logger.info(f"Memory Usage: {mem.percent}% used, {mem.available / (1024 * 1024)} MB avail

    response = await call_next(request) # Appel de la requête

    # Enregistrer l'utilisation de la mémoire après l'appel de l'API
    mem = psutil.virtual_memory()
    logger.info(f"Memory Usage after request: {mem.percent}% used, {mem.available / (1024 * 1

    return response

# Route pour vérifier l'état de santé de l'API
@app.get("/health")
async def health_check():
    return {"status": "OK", "message": "API is running"}

# Route de base pour afficher un message de bienvenue
@app.get("/")
async def root():
    return {"message": "Bienvenue #ADD "}

# Route pour gérer l'absence d'icône favicon
@app.get("/favicon.ico")
async def favicon():
    return {"message": "No ico avaible."}

```

```
# Configuration des origines autorisées pour les requêtes CORS
origins = ["http://localhost:3000", "http://127.0.0.1:5500"]
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Autoriser toutes les origines
    allow_credentials=True, # Autoriser l'envoi des cookies avec les requêtes
    allow_methods=["*"], # Autoriser toutes les méthodes HTTP (GET, POST, etc.)
    allow_headers=["*"], # Autoriser tous les en-têtes
)

# Inclusion des routeurs pour gérer différents points de terminaison
app.include_router(general_qst_router, prefix="/api")
app.include_router(request_data_router, prefix="/api")
app.include_router(general_v1_router, prefix="/api")
app.include_router(classify_intents_router, prefix="/api")

# Point d'entrée de l'application pour lancer le serveur avec uvicorn
if __name__ == '__main__':
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=5000) # Démarrer l'application sur le port 5000
```

## 2. core/config.py

Voici une description détaillée du code présenté :

Ce code est un module de configuration pour une application Python, probablement une API FastAPI. Il gère le chargement de la configuration, l'initialisation des tokens, et la surveillance des fichiers de configuration. Voici les principales fonctionnalités :

- **Importations** : Le code importe diverses bibliothèques nécessaires pour la gestion de l'environnement, le chiffrement, la journalisation et la surveillance des fichiers.
- **Variables globales** : `fernet_key` et `API_KEY` sont déclarées comme variables globales.
- **Fonction `load_configuration()`** : Cette fonction asynchrone charge les variables d'environnement à partir des fichiers `config.env` et `tokens.env`. Elle initialise également la clé Fernet pour le chiffrement.
- **Fonction `initialize_tokens()`** : Cette fonction asynchrone lit le fichier `tokens.env`, extrait les paires clé-valeur, et met à jour les tokens valides.
- **Fonction `start_file_watcher()`** : Cette fonction asynchrone met en place un observateur de fichiers qui surveille les changements dans le répertoire `/app`. Elle utilise un `PollingObserver` pour une compatibilité accrue.

Le code met l'accent sur la sécurité (utilisation de Fernet pour le chiffrement) et la flexibilité (chargement dynamique de la configuration). Il gère également les exceptions de manière robuste et utilise la journalisation pour suivre les événements importants.

```
import os
from dotenv import load_dotenv
from utils.logging_config import logger
from cryptography.fernet import Fernet
import asyncio
from watchdog.observers import Observer
from utils.file_watcher import TokenFileHandler
from core.token_manager import update_current_valid_token, update_cipher_suite
from watchdog.observers.polling import PollingObserver

# Variables globales pour stocker la clé Fernet et l'API Key
fernet_key = None
API_KEY = None

# Fonction asynchrone pour charger la configuration de l'application
```

```

async def load_configuration():
    global fernet_key, API_KEY # Indiquer que ces variables sont globales

    try:
        # Charger les fichiers d'environnement config.env et tokens.env
        load_dotenv("config.env")
        load_dotenv("tokens.env")
    except Exception as e:
        # Log d'erreur si le chargement échoue
        logger.error(f"Failed to load environment variables: {e}")
        raise # Relancer l'exception pour gérer les erreurs plus haut dans la chaîne

    try:
        # Récupérer les clés d'environnement FERNET_KEY et API_KEY
        fernet_key = os.getenv("FERNET_KEY")
        API_KEY = os.getenv("API_KEY")

        # Vérifier que FERNET_KEY est bien définie
        if fernet_key is None:
            raise ValueError("FERNET_KEY environment variable is not set.")

        # Vérifier que API_KEY est bien définie
        if API_KEY is None:
            raise ValueError("API_KEY environment variable is not set.")

        # Créer une suite de chiffrement à partir de la clé Fernet
        cipher_suite = Fernet(fernet_key)
        update_cipher_suite(cipher_suite) # Mettre à jour la suite de chiffrement
        # Log d'information sur le succès du chargement
        logger.info("Configuration loaded successfully.")

    except ValueError as ve:
        # Log et gestion des erreurs de type ValueError
        logger.error(ve)
        raise # Relancer l'exception
    except Exception as e:
        # Log d'erreurs inattendues lors du chargement de la configuration
        logger.error(f"An unexpected error occurred during configuration loading: {e}")
        raise # Relancer l'exception

# Fonction asynchrone pour initialiser les tokens depuis le fichier tokens.env
async def initialize_tokens():
    env_file = "tokens.env" # Fichier contenant les tokens
    new_tokens = {} # Dictionnaire pour stocker les nouveaux tokens

    try:
        # Ouvrir le fichier tokens.env et lire les lignes
        with open(env_file, 'r') as file:
            lines = file.readlines()

        # Boucle sur chaque ligne du fichier
        for line in lines:
            line = line.strip() # Retirer les espaces en début et fin de ligne
            # Ne traiter que les lignes non vides avec des paires clé-valeur
            if line and '=' in line:
                key, value = line.split('=', 1) # Diviser uniquement au premier '='
                # Ajouter la paire clé-valeur au dictionnaire
                new_tokens[key.strip()] = value.strip()

```

```

        logger.info(f"Loaded token: {key.strip()} into current_valid_token ")

    # Vérifier si aucun token n'a été trouvé
    if not new_tokens:
        logger.warning("No tokens found in tokens.env")

    # Mettre à jour les tokens dans le gestionnaire de tokens
    update_current_valid_token(new_tokens)

except FileNotFoundError:
    # Log d'erreur si le fichier tokens.env n'est pas trouvé
    logger.error(f"{env_file} not found.")
except Exception as e:
    # Log pour toute autre erreur lors de la lecture des tokens
    logger.error(f"An error occurred while reading tokens: {e}")

# Fonction asynchrone pour démarrer l'observateur de fichiers
async def start_file_watcher():
    # Créer un gestionnaire d'événements pour surveiller les modifications du fichier de tokens
    event_handler = TokenFileHandler(initialize_tokens, load_configuration)
    # Utiliser un observateur de type PollingObserver pour surveiller les changements
    observer = PollingObserver()
    # Planifier la surveillance sur le répertoire /app
    observer.schedule(event_handler, path='/app', recursive=False)
    observer.start() # Démarrer l'observateur

    try:
        # Boucle infinie pour garder l'observateur actif
        while True:
            await asyncio.sleep(1) # Pause d'une seconde entre chaque itération
    except KeyboardInterrupt:
        # Stopper l'observateur en cas d'interruption clavier
        observer.stop()
    observer.join() # Attendre la terminaison propre de l'observateur

```

### 3. core/security.py

Voici une description détaillée du code de sécurité présenté :

- **Importations** : Le code importe les modules nécessaires de FastAPI, os, cryptography, dotenv, et un module de journalisation personnalisé.
- **Chargement des variables d'environnement** : Il tente de charger les variables d'environnement à partir du fichier "config.env" et récupère la clé API.
- **Gestion des erreurs** : En cas d'échec du chargement des variables d'environnement, une exception est levée et enregistrée.
- **Configuration de l'en-tête API** : Un objet APIKeyHeader est créé pour gérer l'authentification via l'en-tête "X-API-Key".

Fonctions de chiffrement et déchiffrement :

- **encrypt\_string** : Chiffre une chaîne d'entrée en utilisant une suite de chiffrement Fernet.
- **decrypt\_string** : Déchiffre un texte chiffré en utilisant la même suite de chiffrement.
- **Vérification de la clé API** : La fonction asynchrone **verify\_api\_key** vérifie si la clé API fournie correspond à celle stockée dans les variables d'environnement. Si ce n'est pas le cas, une exception HTTP 403 (Forbidden) est levée.

Ce code met en place un système de sécurité robuste pour l'API, en utilisant le chiffrement Fernet pour la protection des données sensibles et en vérifiant l'authenticité des requêtes via une clé API.



```

from fastapi import HTTPException, Depends
from fastapi.security import APIKeyHeader
import os
from cryptography.fernet import Fernet
from dotenv import load_dotenv
from utils.logging_config import logger

# Charger les variables d'environnement à partir du fichier config.env
try:
    load_dotenv("config.env") # Charger les variables depuis config.env
    API_KEY = os.getenv("API_KEY") # Récupérer l'API key de l'environnement
except Exception as e:
    # Enregistrer une erreur si les variables d'environnement ne sont pas chargées correctement
    logger.error(f"Failed to load environment variables: {e}")
    # Lever une exception en cas d'échec
    raise Exception(f"Failed to load environment variables: {e}")

# Définir le header de sécurité pour l'API key
# Utiliser un header nommé "X-API-Key" pour authentifier les requêtes
api_key_header = APIKeyHeader(name="X-API-Key")

# Fonction pour chiffrer une chaîne de caractères
def encrypt_string(input_string, cipher_suite):
    # Chiffrer la chaîne d'entrée en utilisant la suite de chiffrement fournie
    encrypted_text = cipher_suite.encrypt(input_string.encode())
    return encrypted_text # Retourner le texte chiffré

# Fonction pour déchiffrer une chaîne de caractères
def decrypt_string(encrypted_text, cipher_suite):
    # Déchiffrer le texte chiffré en utilisant la suite de chiffrement fournie
    decrypted_text = cipher_suite.decrypt(encrypted_text).decode()
    return decrypted_text # Retourner le texte déchiffré

# Fonction pour vérifier l'API key envoyée dans les requêtes
async def verify_api_key(api_key: str = Depends(api_key_header)):
    # Comparer l'API key envoyée avec celle stockée dans les variables d'environnement
    if api_key != API_KEY:
        # Si elles ne correspondent pas, lever une exception HTTP 403 (Forbidden)
        raise HTTPException(status_code=403, detail="Forbidden")
    return api_key # Retourner l'API key si elle est valide

```

## 4. core/token\_manager.py

Voici une description détaillée du code du gestionnaire de tokens :

- Variables globales :
  - `current_valid_token` : Un dictionnaire pour stocker les tokens valides actuels.
  - `cipher_suite` : Une variable pour stocker la suite de chiffrement.
- Fonctions :
  - **`get_current_valid_token()`** : Retourne le dictionnaire des tokens valides actuels.
  - **`update_current_valid_token(new_tokens)`** : Met à jour le dictionnaire des tokens valides avec de nouveaux tokens.
  - **`get_cipher_suite()`** : Retourne la suite de chiffrement actuelle.
  - **`update_cipher_suite(new_cipher_suite)`** : Met à jour la suite de chiffrement avec une nouvelle instance.

Ce module gère les tokens d'authentification et la suite de chiffrement de manière centralisée, permettant un accès et une mise à jour faciles depuis d'autres parties de l'application.



```
# token_manager.py

# Variable globale pour stocker les tokens valides actuels
current_valid_token = {}
# Variable globale pour stocker la suite de chiffrement
cipher_suite = None

# Fonction pour obtenir le token valide actuel
def get_current_valid_token():
    return current_valid_token # Retourner le dictionnaire des tokens valides actuels

# Fonction pour mettre à jour les tokens valides actuels
def update_current_valid_token(new_tokens):
    global current_valid_token # Indiquer que la variable est globale
    current_valid_token = new_tokens # Mettre à jour les tokens avec les nouveaux tokens fournis

# Fonction pour obtenir la suite de chiffrement actuelle
def get_cipher_suite():
    return cipher_suite # Retourner la suite de chiffrement actuelle

# Fonction pour mettre à jour la suite de chiffrement
def update_cipher_suite(new_cipher_suite):
    global cipher_suite # Indiquer que la variable est globale
    cipher_suite = new_cipher_suite
```

## 5. endpoints/classify\_intents.py

Voici une description détaillée du code dans classify\_intents.py :

- **Importations** : Le code importe les modules nécessaires de FastAPI, les schémas personnalisés, les fonctions de sécurité, les services, et la configuration de journalisation.
- **Création du routeur** : Un objet APIRouter est créé pour gérer les routes de l'API.
- **Définition de la route** : Une route POST "/classify\_intent\_v4" est définie avec le décorateur @router.post.
- **Fonction de classification** : La fonction asynchrone classify\_v4 est définie avec les paramètres suivants :
  - request : Un objet ClassifyRequest contenant le texte à classifier et la langue.
  - http\_request : L'objet Request de FastAPI pour accéder aux informations de la requête HTTP.
  - api\_key : La clé API vérifiée par la fonction verify\_api\_key.
- **Gestion des erreurs** : Le code utilise un bloc try-except pour gérer différents types d'erreurs :
  - ValidationError : Pour les erreurs de validation des données de requête.
  - ValueError : Pour les erreurs de valeurs invalides.
  - Exception générale : Pour toute autre erreur inattendue.
- **Vérification du token** : Le code vérifie si le token fourni correspond au token valide actuel pour "open\_data".
- **Classification** : La fonction classify\_intent\_v4 est appelée avec le texte et la langue fournis.
- **Journalisation** : Les informations de la requête et les erreurs sont enregistrées à l'aide du logger.
- **Réponse** : La fonction renvoie le résultat de la classification ou lève une exception HTTPException en cas d'erreur.

```
from fastapi import APIRouter, HTTPException, Request, Depends
from schemas import ClassifyRequest
from core.security import verify_api_key
from services.functions import classify_intent_v4
```

```

from utils.logging_config import logger
from pydantic import ValidationError
from core.token_manager import get_current_valid_token

# Créer un routeur pour définir les routes liées à la classification des intentions
router = APIRouter()

# Définir une route POST pour classifiez les intentions en version 4
@router.post("/classify_intent_v4")
async def classify_v4(request: ClassifyRequest, http_request: Request, api_key: str = Depends(validate_api_key)):
    try:
        # Extraire les informations de la requête
        text = request.text # Le texte à classifiez
        lang = request.lang # La langue du texte
        token = request.token # Le token fourni par le client

        # Récupérer l'adresse IP du client qui fait la requête
        client_ip = http_request.client.host

        if len(text) >= 2000:
            logger.error(f"Length exceeded from client ip {client_ip}")
            return {"output": "Max caractères 2000"}

        # Récupérer le token valide actuel
        current_valid_token = get_current_valid_token()

        # Vérifier que la clé "open_data" existe dans les tokens valides
        if "open_data" not in current_valid_token:
            logger.error(f"Token key 'open_data' not found from client ip {client_ip}")
            raise HTTPException(status_code=403, detail="Token not found")

        # Vérifier si le token fourni correspond au token valide actuel
        if current_valid_token["open_data"] != token:
            logger.error(f"Invalid token received {token} from client IP {client_ip}")
            raise HTTPException(status_code=403, detail="Invalid token")

        # Appeler la fonction de classification (placeholder pour la fonction réelle)
        response = classify_intent_v4(text, lang)

        # Enregistrer une log indiquant que la requête a été traitée avec succès
        logger.info(f"POST /classify_intent_v4 HTTP/1.1 200 OK FROM IP: {client_ip}")

        return response # Retourner la réponse obtenue

    # Gestion des erreurs de validation des données de la requête
    except ValidationError as e:
        logger.exception(f"Validation error: {e}")
        raise HTTPException(status_code=400, detail="Invalid request data")

    # Gestion des erreurs de type valeur incorrecte dans l'entrée
    except ValueError as e:
        logger.exception(f"Value error: {e}")
        raise HTTPException(status_code=400, detail="Invalid input data")

    # Gestion des erreurs inattendues
    except Exception as e:

```

```
logger.exception(f"Unexpected error: {e}")
raise HTTPException(status_code=500, detail="Internal Server Error")
```

## 6- endpoints/general\_qst.py

Voici une description détaillée du code fourni pour le endpoint `/general_qst` :

- **Importations** : Le code importe les modules nécessaires de FastAPI, les schémas personnalisés, les fonctions de sécurité, les services, et la configuration de journalisation.
- **Création du routeur** : Un objet `APIRouter` est créé pour gérer les routes de l'API.
- **Définition de la route** : Une route `POST /general_qst` est définie avec le décorateur `@router.post`.
- **Fonction principale** : La fonction asynchrone `general_qst` est définie avec les paramètres suivants :
  - `request` : Un objet `GeneralEqst` contenant le texte et le token.
  - `http_request` : L'objet `Request` de FastAPI pour accéder aux informations de la requête HTTP.
  - `api_key` : La clé API vérifiée par la fonction `verify_api_key`.
- **Gestion des erreurs** : Le code utilise un bloc `try-except` pour gérer différents types d'erreurs :
  - `ValidationError` : Pour les erreurs de validation des données de requête.
  - `ValueError` : Pour les erreurs de valeurs invalides.
  - Exception générale : Pour toute autre erreur inattendue.
- **Vérification du token** : Le code vérifie si le token fourni est présent dans les valeurs des tokens valides actuels.
- **Déchiffrement** : Le token est déchiffré en utilisant la suite de chiffrement obtenue via `get_cipher_suite()`.
- **Traitement de la requête** : La fonction `general_qst_v1` est appelée avec le texte et la chaîne traduite (déchiffrée).
- **Journalisation** : Les informations de la requête et les erreurs sont enregistrées à l'aide du `logger`.
- **Réponse** : La fonction renvoie le résultat du traitement dans un dictionnaire avec la clé `"output"`, ou lève une exception `HTTPException` en cas d'erreur.

Ce code met en place un endpoint sécurisé pour traiter des requêtes générales, avec une vérification du token, un déchiffrement des données, et une gestion appropriée des erreurs.

```
from fastapi import APIRouter, HTTPException, Request, Depends
from schemas import GeneralEqst
from core.security import verify_api_key
from pydantic import ValidationError
from services.functions import general_qst_v1
from utils.logging_config import logger
from core.token_manager import get_current_valid_token, get_cipher_suite
from core.security import decrypt_string

# Création d'un routeur FastAPI pour gérer les routes de l'API
router = APIRouter()

# Définir une route POST pour "/general_qst"
@router.post("/general_qst")
async def general_qst(request: GeneralEqst, http_request: Request, api_key: str = Depends(verify
    try:
        # Extraire les données de la requête (texte et token)
        text = request.text
        token = request.token

        # Récupérer l'adresse IP du client qui fait la requête
        client_ip = http_request.client.host

        if len(text) >= 2000:
```

```

        logger.error(f"Length exceeded from client ip {client_ip}")
        return {"output": "Max caractères 2000"}

# Récupérer le token valide actuel et la suite de chiffrement
current_valid_token = get_current_valid_token()
cipher_suite = get_cipher_suite()

# Vérifier si le token est dans les tokens valides
if token not in current_valid_token.values():
    # Log si le token est inconnu ou invalide
    logger.error(f"Unknown token: {token} from IP: {client_ip}")
    raise HTTPException(status_code=403, detail="Could not authenticate token")

# Déchiffrer le token à l'aide de la suite de chiffrement
translated_string = decrypt_string(token, cipher_suite)

# Appeler la fonction principale pour traiter la question générale (Placeholder)
response = general_qst_v1(text, translated_string)

# Log une entrée de succès lorsque la requête est correctement traitée
logger.info(f"POST /general_qst HTTP/1.1 200 OK FROM IP: {client_ip}")

# Retourner la réponse sous forme de dictionnaire
return {"output": response}

# Gérer les erreurs de validation des données de la requête
except ValidationError as e:
    logger.exception(f"Validation error: {e}")
    raise HTTPException(status_code=400, detail="Invalid request data")

# Gérer les erreurs de type valeur incorrecte dans l'entrée
except ValueError as e:
    logger.exception(f"Value error: {e}")
    raise HTTPException(status_code=400, detail="Invalid input data")

# Gérer les erreurs inattendues
except Exception as e:
    logger.exception(f"Unexpected error: {e}")
    raise HTTPException(status_code=500, detail="Internal Server Error")

```

## 7- endpoints/general\_v1.py

Voici une description détaillée du code fourni pour le endpoint `/gener_v1` :

- **Importations** : Le code importe les modules nécessaires de FastAPI, les schémas personnalisés, les fonctions de sécurité, les services, et la configuration de journalisation.
- **Création du routeur** : Un objet `APIRouter` est créé pour gérer les routes de l'API.
- **Définition de la route** : Une route `POST /gener_v1` est définie avec le décorateur `@router.post`.
- **Fonction principale** : La fonction asynchrone `gener_v1` est définie avec les paramètres suivants :
  - `request` : Un objet `ClassifyRequest` contenant le texte, la langue et le token.
  - `http_request` : L'objet `Request` de FastAPI pour accéder aux informations de la requête HTTP.
  - `api_key` : La clé API vérifiée par la fonction `verify_api_key`.
- **Gestion des erreurs** : Le code utilise un bloc `try-except` pour gérer différents types d'erreurs :
  - `ValidationError` : Pour les erreurs de validation des données de requête.
  - `ValueError` : Pour les erreurs de valeurs invalides.

- Exception générale : Pour toute autre erreur inattendue.
- **Vérification du token** : Le code vérifie si le token fourni correspond au token valide actuel pour "open\_data".
- **Traitement de la requête** : La fonction general\_v1 est appelée avec le texte et la langue fournis.
- **Journalisation** : Les informations de la requête et les erreurs sont enregistrées à l'aide du logger.
- **Réponse** : La fonction renvoie le résultat du traitement dans un dictionnaire avec la clé "output", ou lève une exception HTTPException en cas d'erreur.

Ce code met en place un endpoint sécurisé pour traiter des requêtes générales, avec une vérification du token et une gestion appropriée des erreurs. Il utilise la fonction general\_v1 pour le traitement principal de la requête.

```
from fastapi import APIRouter, HTTPException, Request, Depends
from schemas import ClassifyRequest
from core.security import verify_api_key
from pydantic import ValidationError
from services.functions import general_v1
from utils.logging_config import logger
from core.token_manager import get_current_valid_token

# Création d'un routeur FastAPI pour définir les routes de l'API
router = APIRouter()

# Définir une route POST pour "/gener_v1"
@router.post("/gener_v1")
async def gener_v1(request: ClassifyRequest, http_request: Request, api_key: str = Depends(verify_api_key)):
    try:
        # Extraire les informations de la requête (texte, langue, token)
        text = request.text
        lang = request.lang
        token = request.token

        # Récupérer l'adresse IP du client qui fait la requête
        client_ip = http_request.client.host

        if len(text) >= 2000:
            logger.error(f"Length exceeded from client ip {client_ip}")
            return {"output": "Max caractères 2000"}

        # Récupérer le token valide actuel
        current_valid_token = get_current_valid_token()

        # Vérifier si la clé "open_data" existe dans les tokens valides
        if "open_data" not in current_valid_token:
            # Log d'erreur si la clé "open_data" n'est pas trouvée
            logger.error(f"Token key 'open_data' not found from client ip {client_ip}")
            raise HTTPException(status_code=403, detail="Token not found")

        # Vérifier si le token envoyé correspond au token valide actuel
        if current_valid_token["open_data"] != token:
            # Log d'erreur si le token est invalide
            logger.error(f"Invalid token received {token} from client IP {client_ip}")
            raise HTTPException(status_code=403, detail="Invalid token")

        # Appeler la fonction principale pour traiter la requête
        response = general_v1(text, lang)

        # Log indiquant que la requête a été traitée avec succès
        logger.info(f"POST /genere_v1 HTTP/1.1 200 OK FROM IP: {client_ip}")
```

```

        # Retourner la réponse sous forme de dictionnaire
        return {"output": response}

# Gérer les erreurs de validation des données de la requête
except ValidationError as e:
    logger.exception(f"Validation error: {e}")
    raise HTTPException(status_code=400, detail="Invalid request data")

# Gérer les erreurs de type valeur incorrecte dans l'entrée
except ValueError as e:
    logger.exception(f"Value error: {e}")
    raise HTTPException(status_code=400, detail="Invalid input data")

# Gérer les erreurs inattendues
except Exception as e:
    logger.exception(f"Unexpected error: {e}")
    raise HTTPException(status_code=500, detail="Internal Server Error")

```

## 8- endpoints/request\_data.py

Voici une description détaillée du code fourni pour le endpoint `/req_data_v2` :

- **Importations** : Le code importe les modules nécessaires de FastAPI, les schémas personnalisés, les fonctions de sécurité, les services, et la configuration de journalisation.
- **Création du routeur** : Un objet `APIRouter` est créé pour gérer les routes de l'API.
- **Définition de la route** : Une route POST `/req_data_v2` est définie avec le décorateur `@router.post`.
- **Fonction principale** : La fonction asynchrone `req_data` est définie avec les paramètres suivants :
  - `request` : Un objet `ClassifyRequest` contenant le texte, la langue et le token.
  - `http_request` : L'objet `Request` de FastAPI pour accéder aux informations de la requête HTTP.
  - `api_key` : La clé API vérifiée par la fonction `verify_api_key`.
- **Gestion des erreurs** : Le code utilise un bloc `try-except` pour gérer différents types d'erreurs :
  - `ValidationError` : Pour les erreurs de validation des données de requête.
  - `ValueError` : Pour les erreurs de valeurs invalides.
  - Exception générale : Pour toute autre erreur inattendue.
- **Vérification du token** : Le code vérifie si le token fourni correspond au token valide actuel pour `"open_data"`.
- **Traitement de la requête** : La fonction `request_data_v2` est appelée avec le texte et la langue fournis.
- **Journalisation** : Les informations de la requête et les erreurs sont enregistrées à l'aide du logger.
- **Réponse** : La fonction renvoie le résultat du traitement dans un dictionnaire avec la clé `"output"`, ou lève une exception `HTTPException` en cas d'erreur.

Ce code met en place un endpoint sécurisé pour traiter des requêtes de données, avec une vérification du token et une gestion appropriée des erreurs. Il utilise la fonction `request_data_v2` pour le traitement principal de la requête.

```

from fastapi import APIRouter, HTTPException, Request, Depends
from schemas import ClassifyRequest
from pydantic import ValidationError
from services.functions import request_data_v2
from core.security import verify_api_key
from utils.logging_config import logger
from core.token_manager import get_current_valid_token

# Création d'un routeur FastAPI pour définir les routes de l'API
router = APIRouter()

```



```

# Définir une route POST pour "/req_data_v2"
@router.post("/req_data_v2")
async def req_data(request: ClassifyRequest, http_request: Request, api_key: str = Depends(verif
try:
    # Extraire les informations de la requête (texte, langue, token)
    text = request.text
    lang = request.lang
    token = request.token
    client_ip = http_request.client.host

    if len(text) >= 2000:
        logger.error(f"Length exceeded from client ip {client_ip}")
        return {"output": "Max caractères 2000"}

    # Récupérer l'adresse IP du client qui fait la requête
    client_ip = http_request.client.host

    # Récupérer le token valide actuel
    current_valid_token = get_current_valid_token()

    # Vérifier si la clé "open_data" existe dans les tokens valides
    if "open_data" not in current_valid_token:
        # Log d'erreur si la clé "open_data" n'est pas trouvée
        logger.error(f"Token key 'open_data' not found from client ip {client_ip}")
        raise HTTPException(status_code=403, detail="Token not found")

    # Vérifier si le token envoyé correspond au token valide actuel
    if current_valid_token["open_data"] != token:
        # Log d'erreur si le token est invalide
        logger.error(f"Invalid token received {token} from client IP {client_ip}")
        raise HTTPException(status_code=403, detail="Invalid token")

    # Appeler la fonction principale pour traiter la requête
    response = request_data_v2(text, lang)

    # Log indiquant que la requête a été traitée avec succès
    logger.info(f"POST /req_data_v2 HTTP/1.1 200 OK FROM IP: {client_ip}")

    # Retourner la réponse sous forme de dictionnaire
    return {"output": response}

# Gérer les erreurs de validation des données de la requête
except ValidationError as e:
    logger.exception(f"Validation error: {e}")
    raise HTTPException(status_code=400, detail="Invalid request data")

# Gérer les erreurs de type valeur incorrecte dans l'entrée
except ValueError as e:
    logger.exception(f"Value error: {e}")
    raise HTTPException(status_code=400, detail="Invalid input data")

# Gérer les erreurs inattendues
except Exception as e:
    logger.exception(f"Unexpected error: {e}")
    raise HTTPException(status_code=500, detail="Internal Server Error")

```

## 9- utils/file\_watcher.py

Le fichier file\_watcher.py contient une classe TokenFileHandler qui hérite de FileSystemEventHandler. Cette classe est conçue pour surveiller les modifications de fichiers spécifiques et déclencher des actions en conséquence. Voici une description détaillée de son fonctionnement :

- **Importations** : Le code importe FileSystemEventHandler de watchdog.events, asyncio pour la gestion asynchrone, et logger de utils.logging\_config pour la journalisation.
- **Classe TokenFileHandler** : Cette classe étend FileSystemEventHandler pour gérer les événements du système de fichiers.
- **Méthode \_\_init\_\_** :
  - Initialise la classe parent avec super().\_\_init\_\_().
  - Stocke la boucle d'événements asyncio actuelle.
  - Prend deux fonctions en paramètres : initialize\_tokens et load\_configuration.
- **Méthode on\_modified** : Cette méthode est appelée lorsqu'un fichier est modifié.
  - Si le fichier modifié est "tokens.env" :
    - Enregistre un message de log.
    - Exécute de manière asynchrone la fonction initialize\_tokens().
  - Si le fichier modifié est "config.env" :
    - Enregistre un message de log.
    - Exécute de manière asynchrone la fonction load\_configuration().

Cette classe permet de réagir en temps réel aux modifications des fichiers de configuration et de tokens, assurant ainsi que l'application utilise toujours les données les plus à jour sans nécessiter de redémarrage.

```
from watchdog.events import FileSystemEventHandler
import asyncio
from utils.logging_config import logger

# Définir une classe pour gérer les événements du système de fichiers
class TokenFileHandler(FileSystemEventHandler):
    def __init__(self, initialize_tokens, load_configuration):
        super().__init__()
        # Initialiser la boucle d'événements asyncio
        self.loop = asyncio.get_event_loop()
        # Stocker les fonctions pour initialiser les tokens et charger la configuration
        self.initialize_tokens = initialize_tokens
        self.load_configuration = load_configuration

    # Méthode appelée lorsque le système de fichiers détecte une modification
    def on_modified(self, event):
        # Vérifier si le fichier modifié est "tokens.env"
        if event.src_path.endswith("tokens.env"):
            logger.info("tokens.env has been modified, reloading tokens...")
            # Recharger les tokens de manière asynchrone
            asyncio.run_coroutine_threadsafe(self.initialize_tokens(), self.loop)
        # Vérifier si le fichier modifié est "config.env"
        elif event.src_path.endswith("config.env"):
            logger.info("config.env has been modified, reloading configuration...")
            # Recharger la configuration de manière asynchrone
            asyncio.run_coroutine_threadsafe(self.load_configuration(), self.loop)
```

## 10- utils/logging\_config.py

Le fichier `logging_config.py` configure le système de journalisation pour l'application. Voici une description détaillée du code :

- **Importations** : Le code importe les modules `logging` pour la journalisation et `os` pour les opérations liées au système de fichiers.
- **Fonction `setup_logging`** : Cette fonction configure le logger avec les paramètres suivants :
  - Crée un logger avec un nom spécifique.
  - Définit le niveau de journalisation à `DEBUG`.
  - Crée un formateur qui inclut la date, l'heure, le niveau de log et le message.
  - Configure un gestionnaire de console pour afficher les logs dans le terminal.
  - Configure un gestionnaire de fichier pour écrire les logs dans un fichier (`app.log` par défaut).
  - Assure que le fichier de log existe, le crée s'il n'existe pas.
- **Création de l'instance du logger** : Le code crée une instance du logger en utilisant la fonction `setup_logging()`.
- **Gestion des avertissements** : Le commentaire suggère d'ignorer les avertissements dans le terminal, bien que le code pour cela ne soit pas inclus dans l'extrait fourni.

Cette configuration permet une journalisation détaillée et flexible, avec des logs envoyés à la fois à la console et à un fichier, facilitant ainsi le débogage et le suivi de l'application.

```
import logging
import os

def setup_logging(log_file='app.log'):
    # Créer un logger
    logger = logging.getLogger(__name__)
    logger.setLevel(logging.DEBUG) # Définir le niveau de journalisation

    # Créer un formateur qui inclut la date et l'heure
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

    # Créer un gestionnaire de console
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    # Créer un gestionnaire de fichier, en ajoutant au fichier journal
    file_handler = logging.FileHandler(log_file, mode='a') # 'a' pour mode ajout
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    # S'assurer que le fichier journal se trouve dans le même répertoire que le script
    if not os.path.exists(log_file):
        with open(log_file, 'w'): # Créer le fichier s'il n'existe pas
            pass

    return logger

# Créer une instance de logger en utilisant la fonction setup_logging
logger = setup_logging()
```

## 11- `schemas.py`

Le fichier `schemas.py` définit les modèles de données utilisés pour valider les requêtes entrantes dans l'application FastAPI. Voici une description détaillée du code :

- **Importation** : Le code importe BaseModel de pydantic, qui est utilisé pour créer des modèles de données avec validation.
- **ClassifyRequest** : Cette classe définit le modèle pour les requêtes de classification :
  - text (str) : Le texte à classifier (obligatoire).
  - lang (str) : La langue du texte, avec 'fr' (français) comme valeur par défaut.
  - token (str) : Le jeton d'authentification (obligatoire).
- **GeneralEqst** : Cette classe définit le modèle pour les questions générales :
  - text (str) : Le texte de la question (obligatoire).
  - token (str) : Le jeton d'authentification (obligatoire).

Ces modèles Pydantic permettent une validation automatique des données entrantes, assurant que les requêtes reçues par l'API sont correctement structurées et contiennent toutes les informations nécessaires. Si une requête ne correspond pas à ces modèles, FastAPI générera automatiquement une erreur de validation.

```
from pydantic import BaseModel

# Modèle de données pour la classification de requêtes
class ClassifyRequest(BaseModel):
    text: str # Le texte à classifier
    lang: str = 'fr' # La langue du texte, par défaut 'fr' (français)
    token: str # Le jeton d'authentification

# Modèle de données pour les questions générales
class GeneralEqst(BaseModel):
    text: str # Le texte de la question
    token: str # Le jeton d'authentification
```

## Mécanisme de Fonctionnement des Appels API

1. Lorsqu'une requête arrive, FastAPI la dirige vers la route appropriée définie dans `main.py`.
2. La clé d'API est vérifiée dans les en-têtes ( `"X-API-Key"` ) pour s'assurer qu'elle correspond à celle définie dans le fichier de configuration `config.env`.
3. Les données de la requête sont validées en utilisant les schémas Pydantic définis dans `schemas.py`.
4. Le token `open_data` est vérifié pour s'assurer qu'il correspond à celui stocké dans `tokens.env`.
5. Les fonctions NLP (Traitement du Langage Naturel) et LLM (Modèles de Langage) sont exécutées après avoir été importées depuis le fichier `services/functions.py`. Voir section Modèles NLP et LLM.
6. La réponse est renvoyée au client, généralement sous forme de JSON, comme spécifié dans le Guide de Référence de l'API.

## Modèles NLP et LLM

### Modèles utilisés

Dans notre application, nous utilisons plusieurs modèles NLP et LLM pour diverses tâches :

- **sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 avec FAISS embeddings** : Ce modèle est utilisé pour la recherche d'informations dans la documentation et pour identifier les mots-clés des requêtes. Il prend en charge près de 50 langues, y compris l'arabe et le français, et permet une recherche multilingue efficace. Les embeddings générés par ce modèle sont indexés à l'aide de **FAISS** (Facebook AI Similarity Search), une bibliothèque open-source dédiée à la recherche rapide de similarités sur de grands ensembles de vecteurs. FAISS permet d'effectuer des recherches par similarité de manière très efficace, même dans des espaces de très haute dimension. Ces embeddings sont stockés dans le dossier **embeddings** pour une récupération rapide lors des requêtes. [Suivez ce lien pour plus d'informations.](#)

- **Helsinki-NLP/opus-mt-ar-fr** : Ce modèle de traduction est utilisé pour la génération textuelle, spécifiquement pour la traduction de l'arabe vers le français. [Suivez ce lien pour plus d'informations.](#)
- **spaCy fr** : Utilisé pour l'identification des mots-clés et la correction des phrases en français, optimisant ainsi la qualité des requêtes utilisateur.
- **tferhan/finetuned\_camb\_intents** : Un modèle basé sur CamemBERT, utilisé pour la classification des intentions des utilisateurs. Il distingue si une question concerne des informations générales (comme des coordonnées, des adresses, etc.) ou une demande spécifique de données. [Suivez ce lien pour plus d'informations.](#)

## Implémentation

Ces modèles sont initialisés au lancement de l'application afin d'être disponibles localement sans avoir à les télécharger à chaque fois. Cela est géré dans le fichier `init_models.sh`, et ils sont ensuite chargés dans `services/functions.py` comme suit.

### init\_models.sh

Ce script shell (`init_models.sh`) est utilisé pour initialiser l'environnement de modèles NLP dans une application. Il installe **Git LFS** si nécessaire, télécharge le modèle français de **spaCy**, et clone des répertoires de modèles depuis **Hugging Face**. Ensuite, il supprime certains fichiers spécifiques dans ces répertoires s'ils existent pour garder juste le nécessaire et optimiser la mémoire. Les étapes principales sont :

1. Installation de Git LFS pour gérer les grands fichiers dans les répertoires Git.
2. Téléchargement du modèle **spaCy** pour le français.
3. Création d'un répertoire de modèles s'il n'existe pas.
4. Clonage de modèles depuis Hugging Face.
5. Suppression de fichiers spécifiques (ex. : `tf_model.h5`) dans les répertoires clonés.

```
#!/bin/bash

# Fonction pour installer Git LFS (Large File Storage)
install_git_lfs() {
    # Vérifie si apt-get est disponible pour installer Git LFS
    if command -v apt-get > /dev/null; then
        echo "Installation de Git LFS avec apt-get..."
        apt-get update
        apt-get install -y git-lfs
    # Si apt-get n'est pas disponible, vérifie si yum peut être utilisé
    elif command -v yum > /dev/null; then
        echo "Installation de Git LFS avec yum..."
        yum install -y git-lfs
    # Si ni apt-get ni yum ne sont disponibles, affiche une erreur et quitte
    else
        echo "Erreur : apt-get ou yum introuvables. Impossible d'installer Git LFS."
        exit 1
    fi

    # Initialise Git LFS, et si cela échoue, affiche une erreur et quitte
    git lfs install || { echo "Échec de l'initialisation de Git LFS."; exit 1; }
}

# Vérifie si Git LFS est déjà installé
if ! command -v git-lfs > /dev/null; then
    # Si Git LFS n'est pas installé, appelle la fonction pour l'installer
    install_git_lfs
else
    # Sinon, informe que Git LFS est déjà installé
    echo "Git LFS est déjà installé."
```

```

fi

# Téléchargement du modèle français spaCy
echo "Téléchargement du modèle français de Spacy..."
python -m spacy download fr_core_news_md || { echo "Échec du téléchargement du modèle Spacy."; e

# Crée le répertoire 'models' s'il n'existe pas déjà
mkdir -p models || { echo "Échec de la création du répertoire models."; exit 1; }
# Change de répertoire vers 'models'
pushd models || exit

# Clone les répertoires des modèles s'ils n'existent pas déjà
echo "Clonage des dépôts de modèles..."

# Déclare un tableau associatif avec les noms des répertoires et leurs URLs correspondantes
declare -A repos=(
  ["opus-mt-ar-fr"]="https://huggingface.co/Helsinki-NLP/opus-mt-ar-fr"
  ["finetuned_camb_intents"]="https://huggingface.co/tferhan/finetuned_camb_intents"
  ["paraphrase-multilingual-MiniLM-L12-v2"]=
  "https://huggingface.co/sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
)

# Boucle sur chaque modèle et clone le dépôt s'il n'existe pas
for repo in "${!repos[@]}"; do
  if [ ! -d "$repo" ]; then
    # Si le répertoire du modèle n'existe pas, le cloner depuis Hugging Face
    git clone "${repos[$repo]}" "$repo" || { echo "Échec du clonage de $repo"; exit 1; }
  else
    # Si le répertoire existe déjà, passer au modèle suivant
    echo "$repo existe déjà, clonage ignoré."
  fi
done

# Fonction pour nettoyer les fichiers spécifiques dans un répertoire donné
cleanup_files() {
  local dir="$1" # Nom du répertoire
  shift          # Shift pour traiter les fichiers restants
  cd "$dir" || exit
  # Boucle sur chaque fichier à supprimer dans le répertoire
  for file in "$@"; do
    # Si le fichier existe, le supprimer et afficher un message
    if [ -f "$file" ]; then
      rm "$file"
      echo "Fichier $file supprimé dans $dir."
    else
      # Si le fichier n'existe pas, afficher un message et continuer
      echo "$file n'existe pas dans $dir, suppression ignorée."
    fi
  done
  cd .. || exit
}

# Nettoie des fichiers spécifiques dans les répertoires clonés
cleanup_files "opus-mt-ar-fr" "tf_model.h5"
cleanup_files "paraphrase-multilingual-MiniLM-L12-v2" "tf_model.h5" "model.safetensors"

# Revient au répertoire précédent
popd || exit

```



```
# Message final confirmant la fin du processus
echo "Configuration terminée avec succès !"
```

## services/functions.py

### Chargement des variables d'environnement

Le fichier `.env` est chargé pour récupérer les chemins des datasets, des modèles, des index FAISS et des jetons d'accès nécessaires pour l'exécution. Un ensemble de variables obligatoires est vérifié, et si certaines sont manquantes, une erreur est levée.

```
# Charger les variables d'environnement depuis le fichier 'config.env'
try:
    load_dotenv('config.env')
except Exception as e:

    logger.error(f"An error occurred while loading the config.env file: {e}")
    sys.exit(1)

# Charger les chemins des datasets et des index FAISS à partir des variables d'environnement
try:
    tags_dataset_path = os.getenv("TAGS_DATASET_PATH")
    answers_fr_dataset_path = os.getenv("ANSWERS_FR_DATASET_PATH")
    answers_ar_dataset_path = os.getenv("ANSWERS_AR_DATASET_PATH")

    tags_faiss_index = os.getenv("TAGS_FAISS_INDEX")
    answers_fr_faiss_index = os.getenv("ANSWERS_FR_FAISS_INDEX")
    answers_ar_faiss_index = os.getenv("ANSWERS_AR_FAISS_INDEX")

    sentence_model_path = os.getenv("sentence_model_path")
    translation_model_path = os.getenv("translation_model_path")
    intent_classify_model_path = os.getenv("intent_classify_model_path")

    HF_TOKEN = os.getenv("HF_TOKEN")

    sentence_model_name = os.getenv("SENTENCE_MODEL_NAME")

# Vérifier si des variables d'environnement nécessaires sont absentes
required_vars = [
    "TAGS_DATASET_PATH",
    "ANSWERS_FR_DATASET_PATH",
    "ANSWERS_AR_DATASET_PATH",
    "TAGS_FAISS_INDEX",
    "ANSWERS_FR_FAISS_INDEX",
    "ANSWERS_AR_FAISS_INDEX",
    "HF_TOKEN",
    "SENTENCE_MODEL_NAME",
    "sentence_model_path",
    "translation_model_path",
    "intent_classify_model_path"
]

missing_vars = [var for var in required_vars if os.getenv(var) is None]

if missing_vars:
    # Enregistrer les variables d'environnement manquantes et lever une exception
    logger.error(f"Missing environment variables: {'', '.join(missing_vars)}")
```

```

        raise ValueError(f"Missing environment variables: {'', ' '.join(missing_vars)}")

except Exception as e:

    logger.error(f"An error occurred while loading environment variables: {e}")
    sys.exit(1)

```

## Chargement des modèles NLP

### 1. Classification d'intentions:

- Un pipeline de classification de texte est chargé à partir d'un modèle spécifié par le chemin `intent_classify_model_path`.

### 2. Similarité de phrases:

- Le modèle `SentenceTransformer` est utilisé pour calculer les embeddings et mesurer la similarité entre phrases.

### 3. Traduction:

- Un pipeline de traduction de l'arabe vers le français est créé en utilisant le modèle spécifié par `translation_model_path`.

### 4. Correction orthographique:

- `SpellChecker` est utilisé pour la correction des fautes d'orthographe en français.

### 5. Spacy:

- Le modèle linguistique français `fr_core_news_md` est chargé pour l'analyse de texte (tokenisation, POS tagging).

## Chargement des datasets et index FAISS

Les jeux de données pour les tags, les réponses en français et en arabe sont chargés au format JSON et indexés avec FAISS, permettant une recherche vectorielle rapide.

```

# Charger les modèles et pipelines nécessaires
try:
    # Charger les pipelines et modèles nécessaires pour la classification de texte, la similarité
    nlp_pipeline_class = pipeline("text-classification", intent_classify_model_path)
    model = SentenceTransformer(sentence_model_path, device="cpu")
    translation = pipeline("translation", translation_model_path)

    # Charger le modèle Spacy pour le français et le correcteur orthographique
    spell = SpellChecker(language='fr')
    nlp = spacy.load("fr_core_news_md")
except Exception as e:
    # En cas d'erreur lors du chargement des modèles, afficher l'erreur et quitter le programme
    logger.error(f"An error occurred during model loading: {e}")
    sys.exit(1)

# Charger les jeux de données et les index FAISS associés
try:
    dataset_tags = datasets.load_dataset("json", data_files=[tags_dataset_path], split="train")
    dataset_tags.load_faiss_index("embeddings", tags_faiss_index)
    dataset_answers_fr = datasets.load_dataset("json", data_files=[answers_fr_dataset_path],
    dataset_answers_fr.load_faiss_index("embeddings", answers_fr_faiss_index)
    dataset_answers_ar = datasets.load_dataset("json", data_files=[answers_ar_dataset_path],
    dataset_answers_ar.load_faiss_index("embeddings", answers_ar_faiss_index)
except Exception as e:

```

```
logger.error(f"An error occurred while loading datasets: {e}")
sys.exit(1)
```

## Fonctionnalités principales

### 1. Correction orthographique ( `correct_spelling_french` et `correct_spelling_tokens` ):

- Corrige les fautes d'orthographe en français dans un texte donné. Utilise le tokenizer de Spacy pour diviser le texte en tokens (Ici pour Spacy les tokens sont sous forme d'objets de grammaire comme VERBE, NOM...) et appliquer la correction mot par mot.

```
def correct_spelling_french(text):
    try:
        corrected_words = []
        # Séparer le texte en mots individuels
        for word in text.split():
            # Obtenir la correction orthographique pour chaque mot
            correction = spell.correction(word)
            # Gérer les cas où aucune correction n'est trouvée
            corrected_words.append(correction if correction is not None else word)
        # Joindre les mots corrigés pour reformer le texte
        corrected_text = " ".join(corrected_words)
        return corrected_text
    except Exception as e:
        # En cas d'erreur, enregistrer l'erreur et retourner le texte original
        logger.error(f"An error occurred in correct_spelling_french: {e}")
        return text # Retourner le texte original en cas d'erreur
    return corrected_text

def correct_spelling_tokens(text):
    # Analyser le texte avec Spacy
    doc = nlp(text)
    corr = []
    # Corriger l'orthographe de chaque token
    for t in doc:
        corr.append(correct_spelling_french(t.text))

    # Joindre les tokens corrigés pour reformer le texte
    return " ".join(corr)
```

### 2. Recherche de similarité ( `search` , `search_general_qst` ):

- Effectue une recherche dans la dataset en utilisant la similarité des embeddings. La fonction encode la requête à l'aide du modèle `SentenceTransformer` et renvoie les exemples les plus proches dans l'index FAISS.

```
def search(query, data, k, lang='fr'):
    try:
        # Si la langue est le français, corriger l'orthographe des tokens dans la requête
        if lang == 'fr':
            query = correct_spelling_tokens(query)
        # Encoder la requête en un vecteur de caractéristiques
        query_embedding = model.encode(query)
        # Obtenir les exemples les plus proches du vecteur de requête
        _, retrieved_examples = data.get_nearest_examples("embeddings", query_embedding, k=k)
        return retrieved_examples
    except Exception as e:
        # En cas d'erreur, enregistrer l'erreur et retourner None
        logger.error(f"An error occurred during search: {e}")
        return None
```

```
def search_general_qst(query, data, k):
    try:
        # Encoder la requête en un vecteur de caractéristiques
        query_embedding = model.encode(query)
        # Obtenir les exemples les plus proches du vecteur de requête
        _, retrieved_examples = data.get_nearest_examples("embeddings", query_embedding, k=k)
        return retrieved_examples
    except Exception as e:
        # En cas d'erreur, enregistrer l'erreur et retourner None
        logger.error(f"An error occurred during search: {e}")
        return None
```

### 3. Filtrage des termes non pertinents ( `keep_only_matters` ):

- Filtrer les mots non pertinents (comme les verbes, déterminants, etc.) pour ne conserver que les mots significatifs dans une phrase.

```
def keep_only_matters(text):
    try:
        # Analyser le texte avec le modèle spaCy
        terms = nlp(text)
        # Utiliser une liste pour collecter les tokens pertinents
        filtered_terms = [token.text for token in terms if token.pos_ not in ["VERB", "DET",
        # Joindre les tokens filtrés en une seule chaîne de caractères
        req = ' '.join(filtered_terms)
        return req
    except Exception as e:

        logger.error(f"An error occurred in keep_only_matters: {e}")
        return text
```

#### 4. Création de dataset ( `create_dataset_general` ):

- Charge un dataset et un index FAISS à partir de fichiers JSON. Cette fonction est utilisée pour faciliter l'accès aux jeux de données.

```
def create_dataset_general(data_file_path, faiss_index):
    try:
        # Charger le dataset à partir du fichier JSON spécifié
        dataset = datasets.load_dataset("json", data_files=[data_file_path], split="train")
        # Charger l'index FAISS associé pour le dataset
        dataset.load_faiss_index("embeddings", faiss_index)
        return dataset
    except Exception as e:
        # En cas d'erreur, enregistrer l'erreur et retourner None
        logger.error(f"An error occurred while creating dataset: {e}")
        return None
```

### 5. Réponse à une question générale ( `general_qst_v1` ):

- Obtient les chemins des fichiers de dataset et des index FAISS à partir des variables d'environnement basées sur un token.
- Crée un dataset général en utilisant les chemins récupérés.
- Effectue une recherche dans le dataset pour trouver une réponse au texte fourni.
- Retourne la première réponse trouvée ou un message d'erreur en cas de problème.

```
def general_qst_v1(text, token):
    try:

        dataset_path = os.getenv(f"{token}_DATASET_PATH")
        faiss_index_path = os.getenv(f"{token}_FAISS_INDEX")
        # Créer le dataset général en utilisant les chemins obtenus
        dataset = create_dataset_general(dataset_path, faiss_index_path)

        # Rechercher une réponse dans le dataset en utilisant le texte fourni
        quest = search_general_qst(text, dataset, 1)['text']
        return quest[0]
    except Exception as e:
        # En cas d'erreur, enregistrer l'erreur et retourner un message d'erreur
        logger.info(f"An error occured in general_qst : {e}")
        return f"Erreur lors de la réponse sur la documentation"
```

#### 6. Réponse à une question en fonction de la langue ( `general_v1` )

- Vérifie la langue spécifiée ( `lang` ). Si la langue est le français ( `'fr'` ), la fonction recherche une réponse dans le dataset de réponses en français ( `dataset_answers_fr` ).
- Pour les autres langues (par exemple, l'arabe), elle effectue la recherche dans le dataset de réponses en arabe ( `dataset_answers_ar` ).
- Retourne la première réponse trouvée ou un message d'erreur en cas de problème.

```
def general_v1(text, lang = 'fr'):
    try:
        # Vérifie si la langue est le français
        if lang == 'fr':
            # Effectue une recherche dans le dataset des réponses en français
            quest = search(text, dataset_answers_fr, 1)['text']
            return quest[0]
        else:
            # Effectue une recherche dans le dataset des réponses en arabe
            quest = search(text, dataset_answers_ar, 1)['text']
            return quest[0]
    except Exception as e:
        # En cas d'erreur, enregistrer l'erreur et retourner un message d'erreur
        logger.info(f"An error occured in general_v1 : {e}")
        return f"Erreur lors de la réponse sur la documentation"
```

#### 7. Recherche de données publiques ( `chercher_data` , `format_reponse` ):

- Effectue des requêtes HTTP en appelant l'API de [Ckan](#), pour récupérer des jeux de données publics en fonction de mots-clés donnés. Les résultats sont ensuite formatés pour être renvoyés à l'utilisateur.

```
def chercher_data(mot, lang="fr", titles=None, links=None):
    if titles is None:
        titles = []
    if links is None:
        links = []
    try:
        response = requests.get(f"https://data.gov.ma/data/api/3/action/package_search", params={
            'q': mot,
            'lang': lang
        })
        res_url = f"https://data.gov.ma/data/{lang}/dataset?q={mot}"
        if response.status_code != 200:
            return titles, links, response.url, 0
        result = response.json()
        res = result["result"]
```

```

count = res["count"]
results = result["result"]["results"]
titre_fr = results[0]["title_fr"]
titre_ar = results[0]["title_ar"]
id = results[0]["id"]
if lang == "fr":
    titles.append(titre_fr)
    link = "https://data.gov.ma/data/fr/dataset/" + id
    links.append(link)
else:
    titles.append(titre_ar)
    link = "https://data.gov.ma/data/ar/dataset/" + id
    links.append(link)
return titles, links, res_url, count
except Exception as e:
    logger.error(f"An error occurred in chercher_data: {e}")
    return titles, links, "", 0 # Return empty values in case of an error

```

```

def format_reponse(data, lang="fr"):
    try:
        if lang == 'fr':
            response = f"Ici le lien vers toutes les {data[-1]} données correspondant au"
            response += f"Voici un exemple parmi les résultats trouvés :\n"
            response += f"Titre : {data[0][-1]}\n"
            response += f"Lien : {data[1][-1]}\n"
            return response, data[-1]
        else:
            response = f"البيانات المطابقة للكلمة المطلوبة {data[-1]} هنا الرابط لجميع"
            response += f"إليك مثال من بين النتائج التي تم العثور عليها\n"
            response += f"العنوان: {data[0][-1]}\n"
            response += f"الرابط: {data[1][-1]}\n"
            return response, data[-1]
    except Exception as e:
        logger.error(f"An error occurred in format_reponse: {e}")
        return "Erreur dans le formatage de la réponse"

```

#### 8. Rechercher des données ( req\_dt ):

- Cette fonction appelle la fonction `chercher_data` pour rechercher des données à partir d'une requête donnée.
- Si des résultats sont trouvés (la liste des titres n'est pas vide), la réponse est formatée avec la fonction `format_reponse`.
- Retourne la réponse formatée ou la requête d'origine si aucun résultat n'est trouvé ou en cas d'erreur.

```

def req_dt(query, lang="fr"):
    try:
        rg = chercher_data(query, lang)
        # Si des résultats sont trouvés, formate la réponse
        if len(rg[0]):
            reponse_final = format_reponse(rg, lang)
            return reponse_final
        else:
            # Si aucun résultat, retourne la requête originale
            return query
    except Exception as e:
        # En cas d'erreur, log l'erreur et retourne la requête originale

```



```
logger.error(f"An error occurred in req_dt: {e}")
return query
```

#### 9. Requête de données avancée ( `request_data_v2` ):

- Traite une requête en fonction de la langue spécifiée.
- Pour le français :
  - Utilise Spacy pour analyser le texte et filtre les mots non pertinents (par exemple, verbes, déterminants).
  - Recherche les données correspondantes dans un dataset en utilisant la fonction `search`.
  - Formate les résultats avec `req_dt` et stocke les réponses dans une liste.
  - Retourne le résultat avec le plus grand nombre d'occurrences ou la liste des réponses.
- Pour l'arabe :
  - Utilise directement le texte pour la recherche et suit un processus similaire.
- En cas d'erreur, retourne un message d'erreur.

```
def request_data_v2(text, lang='fr'):
    try:
        reponses = []
        req = ""
        # Si la langue est le français, traite le texte avec le modèle NLP pour éliminer cert
        if lang == 'fr':
            doc = nlp(text)
            for token in doc:
                # Garde uniquement les tokens pertinents
                if token.pos_ not in ["VERB", "DET", "ADP", "PRON"]:
                    req += f"{token.text} "
            # Recherche des tags correspondants
            rs = search(req, dataset_tags, 2)
            if rs:
                dis = rs['text']
                # Formate chaque résultat
                for d in dis:
                    fre = req_dt(d)
                    reponses.append(fre)
            # Obtenir la meilleure réponse selon le nombre
            result_final = get_text_of_max_number(reponses)
            if result_final:
                return result_final
            return reponses
        else:
            # Si la langue est différente, utilise directement la recherche
            rs = search(text, dataset_tags, 2)
            if rs:
                dis = rs['text']
                for d in dis:
                    fre = req_dt(d, 'ar')
                    reponses.append(fre)
            result_final = get_text_of_max_number(reponses)
            if result_final:
                return result_final
    except Exception as e:
        # En cas d'erreur, log l'erreur et retourne un message d'erreur
        logger.error(f"An error occurred in request_data_v2: {e}")
        return "Désolé, un problème s'est produit"
```

#### 10. Obtenir le texte avec le plus grand nombre ( `get_text_of_max_number` ):

- Fonction utilitaire qui sélectionne le texte associé à la plus grande valeur numérique, utilisé pour choisir la meilleure réponse parmi plusieurs.

```
def get_text_of_max_number(data):
    max_number = 0
    max_text = None
    for item in data:
        if max_number < item[-1]:
            max_number = item[-1]
            max_text = item[0]
    return max_text
```

#### 11. Classifier l'intention ( `classify_intent_v4` ):

- Corrige l'orthographe du texte (si la langue est le français) et utilise un modèle NLP pour classer l'intention du texte.
- Si le label classifié est `LABEL_0`, appelle la fonction `general_v1` pour générer une réponse.
- Si un autre label est détecté, appelle la fonction `request_data_v2` pour traiter la demande en fonction de l'intention.
- Pour une langue autre que le français (par exemple, l'arabe), traduit d'abord le texte avant de procéder de la même manière.
- Retourne la réponse générée, la langue utilisée, la fonction exécutée, et le texte d'entrée.
- En cas d'erreur, retourne un message d'erreur indiquant que la classification a échoué.

```
def classify_intent_v4(text, lang='fr'):
    try:
        executed_function = ""
        # Si la langue est le français
        if lang == 'fr':
            # Corriger l'orthographe du texte
            text = correct_spelling_tokens(text)
            # Classifier l'intention avec le pipeline NLP
            label = nlp_pipeline_class(text)[0]['label']
            # Si le label est 'LABEL_0', utiliser la fonction general_v1 pour générer une réponse
            if label == 'LABEL_0':
                response = general_v1(text)
                executed_function = "general_v1"
            else:
                # Sinon, appeler request_data_v2 pour traiter la demande
                response = request_data_v2(text)
                executed_function = "request_data"
            # Retourner la réponse, la langue, la fonction exécutée et le texte traité
            return {
                'output': response,
                'language': lang,
                'executed_function': executed_function,
                'input_text': text
            }
        else:
            # Si la langue est différente (par exemple, arabe), traduire le texte en français
            trans = translation(text)[0]['translation_text']
            # Corriger l'orthographe du texte traduit
            deci = correct_spelling_tokens(trans)
            # Classifier l'intention avec le pipeline NLP
            label = nlp_pipeline_class(deci)[0]['label']
            # Si le label est 'LABEL_0', utiliser la fonction general_v1 pour traiter la requête
            if label == 'LABEL_0':
```

```

        response = general_v1(text, 'ar')
        executed_function = "general_v1"
    else:
        # Sinon, appeler request_data_v2 pour générer la réponse en arabe
        response = request_data_v2(text, 'ar')
        executed_function = "request_data"
    # Retourner la réponse, la langue, la fonction exécutée et le texte traité
    return {
        'output': response,
        'language': lang,
        'executed_function': executed_function,
        'input_text': text
    }
except Exception as e:
    # En cas d'erreur, log l'erreur et retourner un message d'erreur avec les détails
    logger.error(f"An error occurred in classify_intent_v4: {e}")
    return {
        'output': "Erreur lors de la classification de l'intention",
        'language': lang,
        'executed_function': "error",
        'input_text': text
    }
}

```

## Méthodologie de génération d'Embeddings

Pour générer et vectoriser un dataset quelconque ou bien le mettre à jour, on peut exécuter le script

`gen_embed.py` comme suit :

### gen\_embed.py

Voici une description détaillée des fonctions du script `gen_embed.py` ainsi que des instructions pour l'utiliser dans le terminal :

#### Description des Fonctions

1. `check_existing_name(name_data, config_file)`
  - **Objectif** : Vérifie si le nom du dataset existe déjà dans le fichier de configuration.
  - **Détails** : Ouvre le fichier de configuration ( `config_file` ) en lecture. Parcourt chaque ligne pour vérifier si une ligne commence par le nom du dataset avec le suffixe `_DATASET_PATH` . Renvoie `True` si le nom existe, sinon `False` .
2. `prompt_user_for_override(name_data)`
  - **Objectif** : Invite l'utilisateur à décider s'il souhaite remplacer le chemin du dataset existant.
  - **Détails** : Demande à l'utilisateur s'il veut écraser le nom du dataset existant. Accepte les réponses 'y' (oui) ou 'n' (non). Répète la demande jusqu'à obtenir une réponse valide.
3. `validate_path(path_data)`
  - **Objectif** : Valide si le chemin fourni pour le dataset est un fichier existant.
  - **Détails** : Vérifie si le fichier à l'emplacement spécifié par `path_data` existe. Lève une exception `FileNotFoundError` si le fichier n'existe pas.
4. `update_config(name_data, path_data, faiss_path)`
  - **Objectif** : Met à jour ou ajoute les entrées dans le fichier de configuration avec le nouveau chemin du dataset et l'index FAISS.
  - **Détails** : Lit le fichier de configuration et met à jour les chemins pour le dataset et l'index FAISS. Écrit les lignes mises à jour dans le fichier de configuration.
5. `generate_embeddings(name_data, path_data)`
  - **Objectif** : Génère des embeddings pour un dataset et crée un index FAISS.

- **Détails :**
  - Valide le chemin du dataset.
  - Vérifie si le nom du dataset existe déjà et demande à l'utilisateur s'il veut le remplacer.
  - Charge le dataset à partir du fichier JSON.
  - Génère des embeddings en utilisant le modèle SentenceTransformer.
  - Crée un index FAISS avec les embeddings générés.
  - Sauvegarde l'index FAISS sur le disque.
  - Met à jour le fichier de configuration avec le nouveau chemin du dataset et l'index FAISS.

## Instructions d'Utilisation dans le Terminal

Pour utiliser le script `gen_embed.py`, procédez comme suit :

### 1. Préparer les Prérequis :

- Assurez-vous que le fichier de configuration `config.env` contient les chemins corrects pour le modèle SentenceTransformer.
- Placez le fichier JSON contenant les données du dataset à l'emplacement approprié.

### 2. Exécuter le Script :

- Ouvrez un terminal.
- Naviguez vers le répertoire contenant le script `gen_embed.py`.
- Exécutez le script avec les arguments requis pour le nom et le chemin du dataset. Utilisez la commande suivante :

```
python gen_embed.py <name_data> <path_data>
```

- `<name_data>` : Le nom que vous souhaitez donner au dataset.
- `<path_data>` : Le chemin vers le fichier JSON contenant les données du dataset.

#### Exemple :

```
python gen_embed.py my_dataset /path/to/dataset.json
```

### 3. Notes Supplémentaires :

- Si le nom du dataset existe déjà dans le fichier de configuration, vous serez invité à choisir si vous voulez le remplacer ou non.
- Assurez-vous que les chemins fournis sont corrects pour éviter des erreurs lors de la validation du fichier.

Le script générera les embeddings pour le dataset spécifié, créera un index FAISS, et mettra à jour le fichier de configuration en conséquence.

## Méthodologie génération de tokens pour les datasets

Pour générer des tokens hachés en MD5 afin de les utiliser dans les payloads pour définir le type de dataset que vous envisagez d'utiliser, vous devez exécuter le fichier `token_gen.py` comme suit :



Pour le portail [data.gov.ma](https://data.gov.ma), il faut utiliser le token qui correspond nécessairement au mot `open_data`

## token\_gen.py

Voici une description détaillée des fonctions contenues dans le script `gen_token.py` :

## Fonctions

### 1. Charger les variables d'environnement ( `load_dotenv` ):

- **Responsabilité** : Charge les variables d'environnement à partir des fichiers `config.env` et `tokens.env` .
- **Détails** : Utilise `dotenv` pour lire les fichiers de configuration. En cas d'échec, le script enregistre une erreur et s'arrête.

### 2. Créer une suite de chiffrement ( `Fernet` ):

- **Responsabilité** : Initialise la suite de chiffrement avec la clé spécifiée dans les variables d'environnement.
- **Détails** : Utilise `cryptography.fernet.Fernet` pour créer un objet de chiffrement. En cas d'échec, le script enregistre une erreur et s'arrête.

### 3. Chiffrer une chaîne de caractères ( `encrypt_string` ):

- **Responsabilité** : Chiffre la chaîne de caractères fournie en utilisant la suite de chiffrement.
- **Détails** : Convertit la chaîne de caractères en bytes, puis utilise `Fernet` pour effectuer le chiffrement. En cas d'échec, le script enregistre une erreur et s'arrête.

### 4. Écrire un token dans le fichier d'environnement ( `write_token_to_env` ):

- **Responsabilité** : Ajoute ou met à jour un token dans le fichier `tokens.env` .
- **Détails** : Lit le fichier `tokens.env` , vérifie si la clé existe déjà, et met à jour la valeur ou ajoute une nouvelle entrée. En cas d'erreur lors de la lecture ou de l'écriture, le script enregistre une erreur et s'arrête.

### 5. Générer un token ( `generate_token` ):

- **Responsabilité** : Génère un token chiffré pour la chaîne de caractères fournie et l'enregistre dans le fichier d'environnement.
- **Détails** : Utilise `encrypt_string` pour créer un token, puis `write_token_to_env` pour l'enregistrer. En cas d'échec, le script enregistre une erreur et s'arrête.

## Utilisation du script dans le terminal

Pour utiliser ce script dans le terminal, suivez ces étapes :

### 1. Préparer l'environnement :

- Assurez-vous que les fichiers de configuration `config.env` et `tokens.env` sont présents dans le même répertoire que le script.
- Le fichier `config.env` doit contenir la clé `FERNET_KEY` nécessaire pour le chiffrement.

### 2. Exécuter le script :

- Ouvrez un terminal.
- Accédez au répertoire contenant le script `gen_token.py` .
- Exécutez le script en fournissant le texte à chiffrer comme argument. Par exemple :

```
python gen_token.py "open_data"
```

- Le script génère un token chiffré pour le texte fourni et l'enregistre dans le fichier `tokens.env` .

### 3. Vérifier le résultat :

- Après l'exécution du script, le token chiffré sera visible dans le fichier `tokens.env` .
- Le script affiche également le token généré dans le terminal.

## Déploiement et Lancement de l'API

Vous pouvez lancer l'API de deux manières :

### 1. Directement via le fichier `main.py` :

- Ouvrez un terminal et exécutez la commande suivante pour démarrer l'API :

```
python main.py
```

## 2. Via le script `run_api.sh` :

- Le script `run_api.sh` est conçu pour lancer une instance d'`uvicorn`, un serveur ASGI pour les applications Python. Pour utiliser ce script, exécutez la commande suivante :

```
./run_api.sh
```

- Ce script démarre l'API avec `uvicorn` en utilisant les paramètres suivants :

```
uvicorn main:app --host 0.0.0.0 --port 5000
```

**Remarque :** Assurez-vous que le fichier `run_api.sh` a les permissions d'exécution. Vous pouvez définir ces permissions avec la commande suivante :

```
chmod +x run_api.sh
```

## Explication des Options d' `uvicorn` :

- `-host 0.0.0.0` : Permet à l'API d'écouter sur toutes les interfaces réseau disponibles.
- `-port 5000` : Définit le port sur lequel l'API sera accessible.

Avec ces configurations, vous pouvez accéder à votre API en visitant `http://localhost:5000/api` dans votre navigateur ou via des outils de requêtes HTTP comme `curl` ou `Postman`. Et ne pas oublier d'intégrer votre clé d'API dans les headers `"X-API-Key"`.

# Déploiement avec Docker (Pour la production)

Pour déployer et exécuter votre application FastAPI sur Ubuntu en utilisant Docker, suivez ces étapes :

### Préparation de l'environnement Ubuntu:

- Mettez à jour la liste des paquets et installez Docker et d'autres dépendances nécessaires :

```
sudo apt update
sudo apt install -y docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

## Sans construire d'image

Pour un déploiement rapide, vous pouvez récupérer l'image Docker depuis Docker Hub à l'aide du [lien ci-dessous](#), où vous trouverez les dernières versions. Suivez alors cette démarche :

```
sudo docker pull tferhan/fastapi_app:v3
```

Et après, exécutez l'image pour créer un conteneur avec cette commande :

```
sudo docker run -d \\\n  --name api_gov \\\n  -v models:/app/models \\\n  -v embeddings:/app/embeddings \\\n  -v datasets:/app/datasets \\\n  -v ./config.env:/app/config.env \\\n  -v ./tokens.env:/app/tokens.env \\\n  -p 5000:5000 \\\n  --env-file ./config.env \\\n
```



```
--env-file ./tokens.env \\  
tferhan/fastapi_app:v3
```

### Construire l'image Docker:

- Assurez-vous que vous avez copié le fichier de votre application et le Dockerfile sur votre machine virtuelle Ubuntu. Ensuite, construisez l'image Docker à partir du Dockerfile :

```
cd api_gov  
sudo docker build -t fastapi_app .
```

#### 1. Exécuter le conteneur Docker:

- Lancez le conteneur Docker en montant les répertoires nécessaires et en exposant le port 5000 :

```
sudo docker run -d \\  
  --name api_gov \\  
  -v models:/app/models \\  
  -v embeddings:/app/embeddings \\  
  -v datasets:/app/datasets \\  
  -v ./config.env:/app/config.env \\  
  -v ./tokens.env:/app/tokens.env \\  
  -p 5000:5000 \\  
  --env-file ./config.env \\  
  --env-file ./tokens.env \\  
  fastapi_app
```

#### 2. Accéder au conteneur pour des opérations supplémentaires:

- Si vous avez besoin d'exécuter des commandes supplémentaires dans le conteneur, vous pouvez ouvrir une session bash dans le conteneur :

```
sudo docker exec -it api_gov /bin/bash
```

### Description des fonctions dans le Dockerfile :

- FROM python:3.9-slim:**  
Utilise l'image officielle Python 3.9 comme base pour le conteneur.
- ENV PYTHONUNBUFFERED=1:**  
Configure l'environnement pour ne pas mettre en cache les sorties Python, ce qui est utile pour les journaux en temps réel.
- RUN apt-get update && apt-get install -y build-essential python3-dev && apt-get clean:**  
Met à jour les paquets, installe les outils de construction et les dépendances nécessaires pour compiler certains modules Python, puis nettoie les fichiers temporaires.
- RUN pip install --upgrade pip:**  
Met à jour pip à la dernière version pour garantir l'installation des dépendances les plus récentes.
- WORKDIR /app:**  
Définit le répertoire de travail dans le conteneur à `/app`.
- COPY requirements.txt /app/:**  
Copie le fichier `requirements.txt` dans le répertoire de travail du conteneur.

- **RUN pip install --no-cache-dir -r requirements.txt:**  
Installe les dépendances Python spécifiées dans `requirements.txt` sans utiliser le cache.
- **COPY ./app:**  
Copie tout le code de l'application dans le répertoire de travail du conteneur.
- **RUN bash init\_models.sh:**  
Exécute le script `init_models.sh` pour initialiser les modèles nécessaires à l'application (assurez-vous que ce script existe et est correct).
- **EXPOSE 5000:**  
Expose le port 5000 pour que l'application puisse accepter les connexions.
- **ENTRYPOINT ["bash", "run\_api.sh"]:**  
Définit le point d'entrée du conteneur pour exécuter le script `run_api.sh`, qui démarrera l'application FastAPI.

En suivant ces instructions, vous serez en mesure de déployer et d'exécuter votre application FastAPI sur Ubuntu en utilisant Docker.

## Gestion des erreurs

Chaque fonction capture les exceptions possibles et enregistre les erreurs dans les logs à l'aide de la configuration de `logger`. En cas d'erreur, des valeurs par défaut ou des messages d'erreur explicites sont renvoyés pour éviter l'interruption brutale de l'exécution.

## Mise à jour des données pour la dataset de requête de données

Pour une meilleure expérience utilisateur, le chatbot doit fournir les données les plus pertinentes lors des requêtes. Pour cela, les titres des données et les tags présents dans le fichier `tags.json` doivent être mis à jour régulièrement, idéalement après l'ajout d'environ 20 nouvelles données sur le portail [data.gov.ma](https://data.gov.ma). Cette fréquence (20 données) semble adéquate pour éviter de surcharger le serveur, car il n'est pas optimal de mettre à jour le fichier pour l'ajout d'une seule donnée.

Pour effectuer cette mise à jour, il est recommandé d'accéder aux endpoints de CKAN suivants :

- [package\\_list](#)
- [tag\\_list](#)

Ces endpoints contiennent la liste des titres et des tags associés à toutes les données présentes sur le portail. Lors de la mise à jour du fichier `tags.json`, la plupart des données y sont déjà présentes, donc seuls les nouveaux titres ou tags récemment ajoutés doivent être incorporés.

Pour réaliser cette mise à jour, on exécute le script `update_tags.py`, qui est responsable de cette tâche. Une fois la nouvelle liste de tags reçue, on procède à l'exécution du fichier `gen_embed.py` pour générer et vectoriser cette nouvelle dataset, en utilisant l'argument `TAGS` afin de produire un fichier `TAGS.faiss`.



Il est impératif d'exécuter la fonction `gen_embed.py` avec les datasets du fichier `tags.json`, en utilisant l'argument nommé `TAGS`. Sinon, l'application continuera à utiliser l'ancien dataset de `tags.json`, ou pourrait même planter. Cette tâche doit être réalisée par un administrateur ou par quelqu'un ayant déjà lu la documentation.

Le fichier retourné par la fonction

`update_tags.py` sert à modifier les datasets dans le fichier `tags.json`, ainsi qu'à générer les embeddings et remplacer `TAGS.faiss` dans les volumes Docker.

## Comment utiliser la fonction update\_tags.py

La fonction

`update_tags.py` prend en argument le chemin du fichier ancien `tags.json` et le chemin dont vous voulez sauvegarder le

nouveau fichier, et enfin retourne un nouveau fichier, car il est sensible de remplacer directement le dataset ancien par le nouveau dans le chemin.

```
import json
import requests
import re
import argparse
import os

DATASETS_PATH = "./datasets"

def get_new_data():
    # URL pour récupérer la liste des tags et des titres depuis l'API
    link_tags = "https://data.gov.ma/data/api/3/action/tag_list"
    link_titres = "https://data.gov.ma/data/api/3/action/package_list"

    try:
        # Effectuer des requêtes GET pour obtenir les données
        tags_req = requests.get(link_tags)
        titres_req = requests.get(link_titres)
    except Exception as e:
        print(f"Error in fetching data from links : {link_tags} and {link_titres}")
        exit(1)

    # Extraire les résultats au format JSON
    result_tags = tags_req.json()["result"]
    result_titres = titres_req.json()["result"]

    # Remplacer les tirets par des espaces dans les titres
    result_titres = [re.sub("-", " ", titre) for titre in result_titres]

    # Fusionner les listes de tags et de titres
    result_merged = result_tags + result_titres
    return result_merged

def load_json(data_path):
    # Charger le contenu d'un fichier JSON
    try:
        with open(data_path, "r") as f:
            data = json.load(f)
            return data
    except FileNotFoundError:
        print("File not found")
        exit(1)

def add_to_json(data_path, updated_path):
    # Ajouter de nouveaux éléments à un fichier JSON existant
    try:
        count = 0
        # Récupérer les nouvelles données
        data = get_new_data()
        # Charger les données existantes
        origin = load_json(data_path)
        # Compter le nombre d'éléments ajoutés
        count = len(data) - len(origin)
        # Ajouter les nouveaux éléments à la liste d'origine
        for rs in data:
```

```

        if rs not in origin:
            origin.append(rs)
        print(f"Added {count} new items to {data_path}")
        # Enregistrer les données mises à jour dans un nouveau fichier
        with open(updated_path, "w") as f:
            json.dump(origin, f)
        print(f"Updated file saved to {updated_path}")
    except Exception as e:
        print(f"Error in adding to json file : {e}")
        exit(1)

def validate_paths(data_path, updated_path):
    # Ensure both paths are in the datasets directory
    if not (data_path.startswith(DATASETS_PATH) and updated_path.startswith(DATASETS_PATH)):
        print("Both data_path and updated_path must be within the datasets directory.")
        exit(1)

if __name__ == "__main__":
    # Configuration de l'analyseur d'arguments pour la ligne de commande
    parser = argparse.ArgumentParser(description="Add new items to the json file")
    parser.add_argument("data_path", type=str, help="Path to the json file")
    parser.add_argument("updated_path", type=str, help="Path to the updated json file")
    args = parser.parse_args()

    validate_paths(args.data_path, args.updated_path)
    add_to_json(args.data_path, args.updated_path)

```

## Exemple d'utilisation

```
python update_tags.py "./datasets/tags.json" "./datasets/updated_tags.json"
```

Ensuite pour la vectorisation

```
python gen_embed.py "TAGS" "./datasets/updated_tags.json"
```



Une exception sera déclenchée pour vous indiquer qu'un nom TAGS existe déjà pour être remplacé. Écrivez "y" si vous souhaitez poursuivre.



Si vous avez commis une erreur, vous pouvez toujours modifier le chemin des TAGS dans les variables d'environnement `config.env` avec l'ancien chemin.

## Exécution des scripts

L'exécution des scripts de génération de tokens ou de vectorisation doit se faire à l'intérieur du shell du conteneur Docker. Pour y accéder, vous pouvez :

```
sudo docker exec -it nom_du_conteneur /bin/bash
```

## Utilisation de l'API pour d'autres portails

L'API peut en tout cas être utilisée pour d'autres portails, pour la recherche de données approchée (RAG - Retrieved Augmented Generation) avec d'autres API de génération de texte si disponibles, ou bien intégrée avec d'autres applications. Dans ce qui suit, nous allons lister les différentes utilisations de l'endpoint `/api/general_qst`, qui est un endpoint externe de l'API d'Open Data. Mais avant tout, nous devons expliquer comment fonctionne la logique derrière cet endpoint.

Cette API prend en arguments ou en payloads un texte — une question ou une information à chercher — et un token. Ce token est généré par le script `token_gen.py`. Ici, nous allons donner un exemple simple pour simuler l'utilisation de cette API dans des cas d'utilisation réels.

## Simulation pour le portail Academia Raqmya :

Les étudiants cherchent souvent avec des mots-clés qui peuvent ne pas être adaptés à une requête valide pour la recherche de cours en ligne. Par exemple, ils peuvent vouloir exprimer leurs intentions ou leurs objectifs afin de trouver le cours en ligne adéquat pour atteindre leurs buts. Une recherche stricte de cours n'est donc pas la solution optimale. Dans ce cas, on peut utiliser la vectorisation de texte (embeddings) pour effectuer une recherche de similarité entre les phrases, ce qui permet de retourner le cours en ligne approprié pour les étudiants. Découvrons donc la démarche pour l'implémentation de cette solution avec l'endpoint `/api/general_qst`.

### 1 - Collecte de données

Il faut tout d'abord collecter les données que les étudiants vont rechercher, comme les titres des cours en ligne. L'utilisation du Web Scraping ou bien des endpoints similaires du portail peut servir à collecter ces données. Ensuite, il convient de les transformer au format d'une liste `["titre_1", "titre_2", ..., "titre_n"]` et de les enregistrer dans un fichier `.json`, par exemple `academia.json`, dans le répertoire `datasets`.

### 2 - Vectorisation de données (Embeddings)

Par la suite, comme nous l'avons fait pour les données `tags.json` (voir section précédente), il est nécessaire de vectoriser ces données collectées. C'est là qu'intervient le script `gen_embed.py`. Il faut également nommer notre dataset, par exemple **ACADEMIA**.

```
python gen_embed.py "ACADEMIA" "./datasets/academia.json"
```

Une fois exécuté, vous verrez un nouveau fichier

`ACADEMIA.faiss` ajouté dans le répertoire `embeddings/`.

### 3 - Génération de token

Finalement, pour différencier chaque dataset, on doit générer un token qui contient le nom du dataset, ce qui est fait par la fonction

`token_gen.py` comme ceci :

```
python token_gen.py "ACADEMIA"
```

Un token sera retourné et devra être copié puis ajouté dans le payload pour effectuer une requête.

### 4 - Exemple de requête :

L'appel de l'endpoint

`/api/general_qst` peut être effectué de la manière suivante :

```
curl -X POST "http://chatbot.data.gov.ma:5000/api/general_qst" -H "X-API-Key: datagovma"
-H "Content-Type: application/json"
-d "{\"text\": \"je veux apprendre Python\",
\"token\": \"ici le token généré\"}"
```

# API Docs

## Introduction

Ce guide de référence est votre point d'entrée pour comprendre l'API du Chatbot [Data.gov.ma](#) .

**!** Vous avez besoin d'un jeton d'intégration pour interagir avec l'API. Vous pouvez trouver un jeton d'intégration après avoir créé une intégration sur le fichier `config.env` .

## Conventions

L'URL de base pour envoyer toutes les requêtes API est `chatbot.data.gov.ma/api` .

L'API suit les conventions RESTful lorsque cela est possible, la plupart des opérations étant effectuées via des requêtes `GET` et `POST` sur les ressources de page. Les corps de requête et de réponse sont encodés au format JSON.

## POINTS DE TERMINAISON PRIS EN CHARGE

Méthode HTTP	Point de terminaison
GET	Vérifier l'état de l'API
POST	Demander l'information

## RÉPONSES

Champ	Type	Description
<code>output</code>	<code>chaîne</code>	La réponse retournée
<code>language</code>	<code>chaîne</code>	Langue utilisé : <code>"fr"</code> ou <code>"ar"</code> .
<code>text</code>	<code>chaîne</code>	L'input
<code>executed_function</code>	<code>chaîne</code>	La fonction exécutée lors de la classification de la requête

## Codes de statut

Les codes de réponse HTTP sont utilisés pour indiquer les classes générales de réussite et d'erreur.

### Code de réussite

Citation de statut HTTP	Description
200	Requête traitée avec succès.

### Codes d'erreur

Les réponses d'erreur contiennent plus de détails sur l'erreur dans le corps de la réponse, dans les propriétés `"detail"` .







Citation de statut HTTP	<code>detail</code>	Explication
400	Invalid request data	Le corps de la requête n'a pas pu être décodé en JSON
	Invalid input data	Les données de la requête ne sont pas valides
500	Internal Server Error	Problème de connexion ou bien l'API key non valide
403	Invalid token	Le token du dataset <code>open_data</code> est invalide.



Citation de statut HTTP	detail	Explication
	Could not authenticate token	Seul pour l'endpoint <code>/general_qst</code> le token non valide.
	Token not found	Le token du dataset <code>open_data</code> n'existe pas.

# Points de terminaison (Endpoints)

## Points de terminaison

Aa Nom	Type	URL	Payloads	Headers	Description	Catégorie
 <a href="#">Requête de données</a>	POST	<a href="#">/req_data_v2</a>	- <code>text</code> : La donnée que vous voulez chercher. - <code>lang</code> : " fr " ou bien " ar " . - <code>token</code> : Token d' <code>open_data</code>	Nécessite une clé d'API dans les entêtes <code>X-API-Key</code>	Répond sur une requête de données d'après les mots et titres figurés dans la dataset <code>TAGS</code> . (consultez svp <code>config.env</code> )	Open Data
 <a href="#">Question générale</a>	POST	<a href="#">/gener_v1</a>	- <code>text</code> : La question que vous voulez sa réponse. - <code>lang</code> : " fr " ou bien " ar " . - <code>token</code> : Token d' <code>open_data</code>	Nécessite une clé d'API dans les entêtes <code>X-API-Key</code>	Répond sur une question générale d'après les dataset <code>ANSWERS_FR</code> et <code>ANSWERS_AR</code> (consultez svp <code>config.env</code> )	Open Data
 <a href="#">Classification et réponse sur une question quelconque</a>	POST	<a href="#">/classify_intent_v4</a>	- <code>text</code> : Une question quelconque. - <code>lang</code> : " fr " ou bien " ar " . - <code>token</code> : Token d' <code>open_data</code>	Nécessite une clé d'API dans les entêtes <code>X-API-Key</code>	Classifie une question si elle est une question générale ou bien une requête sur les données, puis répond.	Open Data
 <a href="#">Réponse sur dataset</a>	POST	<a href="#">/general_qst</a>	- <code>text</code> : Ce que vous cherchez. - <code>token</code> : Token de la dataset.	Nécessite une clé d'API dans les entêtes <code>X-API-Key</code>	Retourne une réponse d'une dataset autre que celles d' <code>Open Data</code>	Autre
 <a href="#">Home</a>	GET	<a href="#">/</a>			Simple route	Health
 <a href="#">Vérification d'état</a>	GET	<a href="#">/health</a>			Vérifie si l'API est active	Health
<a href="#">Sans titre</a>						

## Exemples de code

Ces exemples de code donnent une intuition sur comment utiliser les `API Endpoints` .

Pour le portail [data.gov.ma](#) vous pouvez utiliser les trois `endpoints` comme ce qui suit:

### Endpoints:

▼ `classify_intent_v4`

```
import requests

headers = { 'X-API-Key' : "Ta clé d'Api",
            'Content-Type': 'application/json' }

payloads = { 'text' : 'Ta question',
              'lang' : "La langue souhaitée 'fr' ou 'ar' ",
```

```

        'token' : "Le token généré d'open_data " }

link = "https://chatbot.data.gov.ma/api/classify_intent_v4"

request = requests.post(link, headers=headers, json=payloads)

print(request.json())

#Exemple
# {
    'output': "La réponse",
    'language': "fr",
    'executed_function': "la fonction exécutée",
    'input_text': "Ta question"
}

```

#### ▼ general\_v1

```

import requests

headers = { 'X-API-Key' : "Ta clé d'Api",
            'Content-Type': 'application/json'}

payloads = { 'text' : 'Ta question générale',
              'lang' : "La langue souhaitée 'fr' ou 'ar' ",
              'token' : "Le token généré d'open_data " }

link = "https://chatbot.data.gov.ma/api/general_v1"

request = requests.post(link, headers=headers, json=payloads)

print(request.json())

#Exemple
# {
    'output': "La réponse"
}

```

#### ▼ req\_data\_v2

```

import requests

headers = { 'X-API-Key' : "Ta clé d'Api",
            'Content-Type': 'application/json'}

payloads = { 'text' : 'Ta question générale',
              'lang' : "La langue souhaitée 'fr' ou 'ar' ",
              'token' : "Le token généré d'open_data " }

link = "https://chatbot.data.gov.ma/api/req_data_v2"

request = requests.post(link, headers=headers, json=payloads)

print(request.json())

#Exemple
# {

```

```
        'output': "La réponse"
    }
```

Pour d'autre dataset ou portails vous pouvez utiliser cet **endpoint** :

Endpoint:

▼ **general\_qst**

```
import requests

headers = { 'X-API-Key' : "Ta clé d'Api",
            'Content-Type': 'application/json' }

payloads = { 'text' : 'Ta question générale',
             'token' : "Le token généré pour la dataset correspondante" }

link = "https://chatbot.data.gov.ma/api/general_v1"

request = requests.post(link, headers=headers, json=payloads)

print(request.json())

#Exemple
# {
    'output': "La réponse"
}
```