from fastapi import FastAPI

Comment fonctionne FastAPI

Question: Qu'est ce qu'une API ?

- Le backend est une application API, et le frontend se connecte au backend via des requêtes **HTTP.**
- Une **API** (Application Programming Interface) est une application qui permet à d'autres logiciels de se connecter et de communiquer avec elle via des requêtes HTTP (GET, POST, PUT, DELETE.

1) HTTP request

→ HTTP (HyperText Transfer Protocol) est un protocole de transfert hypertexte, utilisé pour la communication entre un client et un serveur sur Internet ou d'autres réseaux

→ Request HTTP

GET : Utilisé pour lire des données depuis le serveur, sans rien modifier sur le serveur.

Exemple : GET /products → Récupère la liste des produits.

POST : Utilisé pour envoyer des données du client vers le serveur, souvent pour créer quelque chose de nouveau.

Exemple : POST /users avec les données { "name": "Minh", "email": "minh@gmail.com" → Crée un nouvel utilisateur.

POST est généralement accompagné d'un formulaire ou de JSON dans le corps de la requête (request body).

PUT : Utilisé pour mettre à jour un élément existant, en le remplaçant entièrement.

Exemple : PUT /users/5 avec { "name": "Minh", "email": "minh@gmail.com" } → Modifie toutes les informations de l'utilisateur 5.

1) App API

Composant	Rôle
Endpoint (Route)	Les chemins appelés par le client (ex. /login, /products).
Logique métier	Travailler avec la base de données, vérifier les données, traiter la logique
Response **JSON**	Le résultat est toujours en JSON (ou XML, mais principalement JSON).

Sécurité Auth	Le client doit s'authentifier avec un **token,** une clé API, **JWT**, etc.
Documentation API	**Swagger**, Postman ou OpenAPI pour documenter l'utilisation de l'API.

→ router est un groupe d'endpoints. C'est une partie d'une application web qui gère les requêtes du client. Il dirige les requêtes vers le bon endpoint.

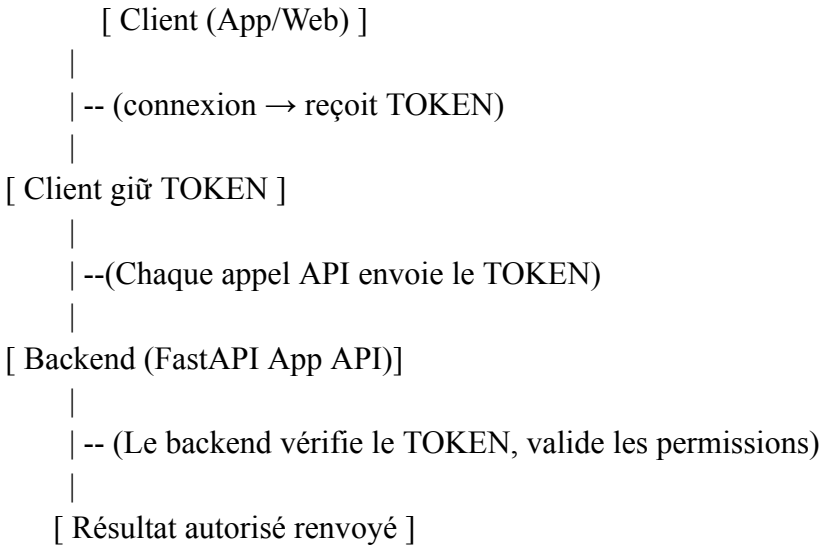
→ endpoint:est une URL spécifique vers laquelle le router redirige. Chaque endpoint correspond à une action précise (ex : lire des données, modifier, supprimer...), et l'application exécutera cette action, puis renverra une réponse.

→ JSON(JavaScript Object Notation) :

- Le JSON n'est pas limité à JavaScript. C'est un format universel permettant aux applications de s'échanger des données (via API, Internet...).
- JSON = Une façon de représenter des données en texte, facile à lire par les humains et les machines.
- Un endpoint FastAPI reçoit et retourne souvent du JSON

→ (Auth)Token : une chaîne de caractères (comme un mot de passe temporaire) utilisée pour : Vérifier l'identité de l'utilisateur et autoriser ce que l'utilisateur peut faire sur le serveur.

- JWT (JSON Web Token) : un token qui contient lui-même les infos, très rapide à valider, et très utilisé aujourd'hui
- token : est envoyé avec chaque requête API. Il est inclus dans l'en-tête HTTP pour que le backend puisse vérifier :
  Est-ce que ce token est valide ?
  Est-ce que l'utilisateur peut accéder à cette API ?
-



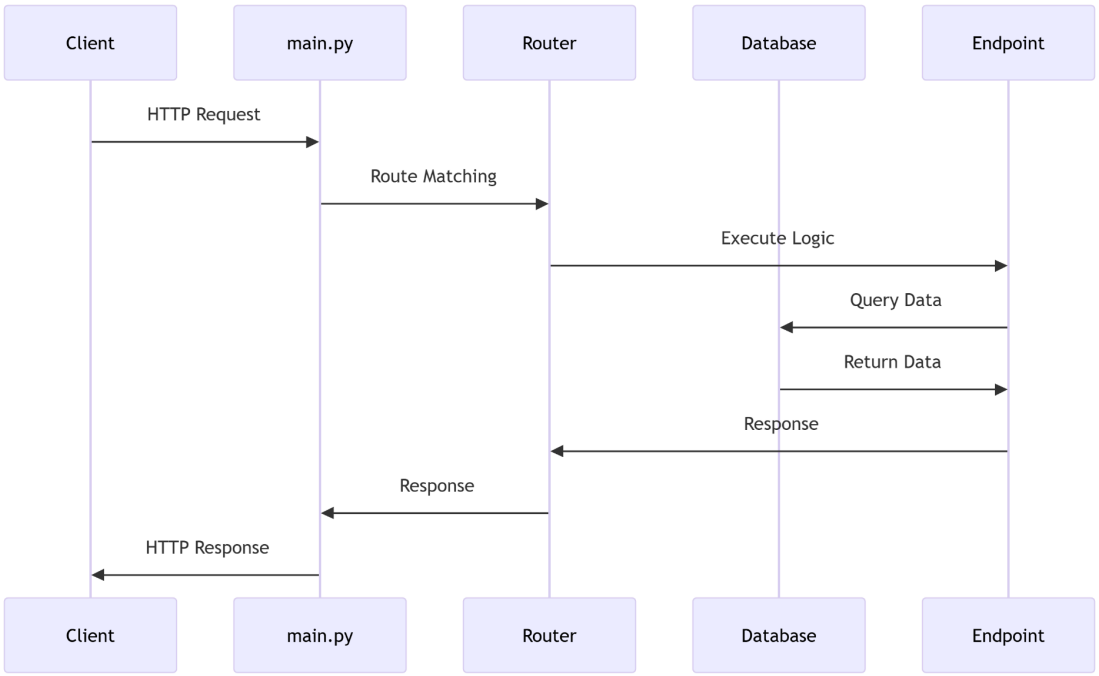
→ Pourquoi avons-nous besoin de documentation d'API?

- Ce n'est pas l'utilisateur final qui appelle directement les API
- Ce sont des applications comme frontend web (React, VueJS) ou mobile app qui appellent les API pour récupérer et envoyer des données.
- On a créé une application API (backend) avec beaucoup d'API différentes :
    - nom de l'API,
    - réponse attendu,
    - logique métier,
    - besoin d'un token ou non..
- Si le développeur frontend n'a pas la documentation → il ne saura pas comment appeler correctement ces API→ Il faut donc générer automatiquement une documentation API via Swagger, Postman, ou OpenAPI
- Swagger (Swagger (fourni automatiquement par FastAPI)):
    - Quand vous lancez l'app FastAPI, vous avez un Swagger UI.
    - Swagger lit automatiquement vos routes FastAPI depuis le code, et crée une interface visuelle.
    - Le développeur frontend peut tester directement l'API depuis Swagger.

      Exemple :

        - API POST /login
        - À envoyer : username, password
        - Retourne : un token

2) Les étapes de fonctionnement de FastAPI



from fastapi.middleware.cors import CORSMiddleware

Utilisé pour configurer CORS (Cross-Origin Resource Sharing) – permet au frontend d'accéder à l'API depuis des domaines différents.

from app.api import chat, auth

Importer les routeurs chat et auth déclarés dans app/api/.

from app.core.config import settings

- SSettings dans un projet logiciel : c'est un module défini dans config (qui est aussi un module) contenant toutes les configurations ou paramètres nécessaires à l'application.
- Configuration : c'est un ensemble de paramètres que nous pouvons ajuster pour modifier le comportement de l'application sans modifier le code source.

Question 1: **Qu'est-ce que le "comportement de l'application"** ?

Réponse : C'est la manière dont l'application exécute ses fonctions et traite les tâches, par exemple : la façon dont elle se connecte à la base de données, traite les requêtes utilisateur, etc.

Question 2: **Que signifie "modifier le comportement sans modifier le code source" ?**

Réponse : Cela signifie changer la manière dont l'application fonctionne (comportement) en modifiant ses configurations, au lieu de modifier les lignes de code directement.

Exemple :

Une application web peut avoir la configuration suivante pour la connexion à une base de données, dans le code source :

```
# config.py
class Config:
    DB_URL = "postgresql://user:password@localhost/mydatabase"
```

- Si nous souhaitons changer la base de données (par exemple, passer de localhost à un serveur distant, nous pouvons simplement modifier l'adresse dans DB_URL sans toucher au reste du code (app.py).

app = FastAPI(title="Chatbot API", version="1.0.0")

Initialisation de l'application FastAPI appelée app.

app.add_middleware(
CORSMiddleware,

(CORS dans cet exemple)Cross-Origin Resource Sharing est un mécanisme de sécurité utilisé pour gérer le partage de ressources entre différents domaines.

allow_origins=["*"], →Autorise toutes les origines (domaines) à envoyer des requêtes
allow_credentials=True,
allow_methods=["*"], →Autorise toutes les méthodes HTTP (GET, POST, PUT, DELETE, etc.)
allow_headers=["*"], →Autorise tous les en-têtes
)

Exemple pour mieux comprendre : Lorsque vous voulez permettre aux clients (comme des applications web ou mobiles) d'envoyer des requêtes à votre API depuis divers domaines.

app.include_router(auth.router, prefix="/auth", tags=["Auth"])

Ajouter le routeur depuis le module auth à l'application FastAPI.

auth.router : un routeur défini dans le module auth (contenant les endpoints liés à l'authentification, comme login, register, etc.).

prefix="/auth" : tous les endpoints dans auth.router auront le préfixe /auth.

Ex. : /login devient /auth/login dans l'API.

tags=["Auth"] : permet de regrouper les routes dans la documentation Swagger UI de FastAPI.

app.include_router(chat.router, prefix="/chat", tags=["Chat"])

Même principe que auth.router, mais pour les routes de discussion (chat).

@app.on_event("startup")

C'est un décorateur FastAPI qui indique que la fonction startup() sera appelée au démarrage de l'applicatio

async def startup():
await init_db()

Appelle une fonction asynchrone pour initialiser la connexion à la base de données (création de tables, pool de connexions, etc.).

Question 1 : **Qu'est-ce que *async* ?**

Réponse : C'est l'une des parties les plus importantes de Python moderne, en particulier lorsqu'on développe des applications web avec FastAPI.

async (abréviation de *asynchronous*) signifie asynchrone, c'est-à-dire que le programme n'a

pas besoin d'attendre qu'une tâche soit terminée avant de continuer à exécuter les étapes suivantes.

Question 2 : **Pourquoi a-t-on besoin de *async* ?**
Réponse : Dans une application web, certaines opérations prennent du temps :

- Appels d'API vers un autre serveur
- Requêtes vers une base de données
- Lecture/écriture de fichiers

Sans async, le programme devrait attendre la fin de chaque tâche, l'une après l'autre.
Avec async, le programme peut continuer à exécuter d'autres traitements pendant l'attente, ce qui permet de :

- Gagner en rapidité
- Économiser des ressources
- Servir plus d'utilisateurs en même temp

if __name__ == "__main__":
Assurer que ce bloc de code ne s'exécute que si le fichier main.py est exécuté directement (et non importé ailleurs).
Nếu import file đó từ file khác, thì __name__ sẽ là tên của file/module đó (ví dụ "main")
import uvicorn
Question : **On pensais que FastAPI est déjà un serveur. Pourquoi avons-nous besoin de Uvicorn ?**
Réponse : FastAPI n'est pas un serveur web – c'est un framework pour construire des applications web (API).

En d'autres termes :

- FastAPI permet de définir les routes, gérer les requêtes, retourner les réponses.
- Mais il ne gère pas les connexions HTTP entrantes.
- Uvicorn est le serveur qui écoute les connexions HTTP et les transmet à FastAPI pour traitement.

uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
Ce code permet de lancer un serveur FastAPI (ou une application ASGI), en utilisant Uvicorn comme serveur web.
📁 app/api/chat.py
from fastapi import APIRouter
Importer APIRouter pour créer un groupe de routes distinctes (routes modulaires), permettant de structurer les routes par fonctionnalité.

from app.services.chat import get_chat_response

Importer la fonction de traitement de chat depuis services/chat.py.

router = APIRouter()

Créer un nouveau routeur pour définir les API liées au chat.

@router.post("/")

async def chat(message: dict):

Définit une route POST / (quand on appelle /chat/) qui reçoit un message sous forme de dictionnaire JSON.

response = await get_chat_response(message["message"])

Appelle la fonction de traitement du message et attend la réponse du chatbot.

return {"response": response}

Retourne la réponse au format JSON à l'utilisateur.

from fastapi import APIRouter

Import router như trên.

router = APIRouter()

Importation du routeur comme précédemment.

@router.post("/login")

def login():

Crée une route POST /login pour simuler une connexion.

return {"msg": "Login success (fake)"}

Retourne un message indiquant une connexion réussie (simulée, sans vérification).

📁 app/core/config.py

class Settings:

Créer une classe contenant les paramètres principaux de l'application → cela permet de centraliser la configuration, et lors du changement d'environnement (dev → test → production), il suffit de modifier les paramètres sans toucher au code

Question : **Qu'est-ce que SQLite et aiosqlite ?**

sqlite : une base de données légère, sans serveur, stockée dans un fichier .db.

aiosqlite : un pilote qui permet une connexion asynchrone à SQLite.

DB_URL: str = "sqlite+aiosqlite:///./chatbot.db"

URL de connexion à la base de données SQLite, utilisant aiosqlite pour le support async.

SECRET_KEY: str = "super-secret-key"

Clé secrète utilisée pour encoder les tokens JWT.

ALGORITHM: str = "HS256"

Algorithme utilisé pour l'encodage des tokens JWT.

RASA_URL: str = "http://localhost:5005/webhooks/rest/webhook" # URL de server RASA

URL du serveur RASA à qui envoyer les messages du chatbot.

settings = Settings()

Initialisation d'un objet settings à utiliser dans toute l'application.

📁 app/core/security.py

```
from datetime import datetime, timedelta
from jose import JWTError, jwt
from app.core.config import settings
```

Importer les bibliothèques nécessaires pour créer et vérifier les tokens JWT.

```
# Fonction pour créer un jeton d'accès
def create_access_token(data: dict, expires_delta: timedelta | None = None):
    to_encode = data.copy()
```

Copie des données utilisateur pour éviter de modifier les données d'origine.

```
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=15))
```

Ajoute une date d'expiration (par défaut 15 minutes).

```
    to_encode.update({"exp": expire})
 encoded_jwt=jwt.encode(to_encode,settings.SECRET_KEY,algorithm=settings.ALGORIT)
```

Encode le token avec la clé et l'algorithme définis.

```
    return encoded_jwt
```

Retourne le token encodé.

```
# Fonction pour vérifier un jeton
def verify_token(token: str):
    try:
        payload = jwt.decode(token, settings.SECRET_KEY,
algorithms=[settings.ALGORITHM])
        return payload
```

Décode le token. Si la vérification réussit, retourne le contenu (payload).

```
    except JWTError:
        return None
```

En cas d'échec (JWT invalide), retourne None.

📁 app/db/database.py

```
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from app.core.config import settings
from app.db import models
```

Importer les composants nécessaires pour la connexion à la base de données avec SQLAlchemy en mode asyn.

```
engine = create_async_engine(settings.DB_URL, echo=True)
```

Créer un moteur async pour la connexion à la base de données.

Question **: Qu'est-ce qu'un "engine" ?**

→ C'est l'objet responsable de la connexion à la base de données

```
SessionLocal = sessionmaker(engine, class_=AsyncSession, expire_on_commit=False)
```

Créer une fabrique pour produire des sessions asynchrones de base de données

```
async def init_db():
```
Fonction pour initialiser la base de données.
```
    async with engine.begin() as conn:
        await conn.run_sync(models.Base.metadata.create_all)
```
Ouvre une connexion et synchronise le schéma défini dans models.py vers la base.
Question : **Qu'est-ce qu'un "model" ?** C'est une classe représentant une table dans la base de données.

📁 app/db/models.py
```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
```
Import các thành phần của SQLAlchemy.

```
Base = declarative_base()
```
Initialisation de la base pour les modèles ORM.

Question : ORM, qu'est que c'est ?

→ ORM (Object Relational Mapping) est une méthode pour représenter les tables d'une base de données avec des classes Python.
```
class ChatHistory(Base):
```
Définir le modèle ORM ChatHistory.
```
    __tablename__ = "chat_history"
```
Attribuer un nom de table.
```
    id = Column(Integer, primary_key=True, index=True)
    question = Column(String)
    answer = Column(String)
```
Définir les colonnes

📁 app/services/chat.py
```
from app.services.rasa import send_to_rasa
async def get_chat_response(message: str) -> str:
```
Fonction pour obtenir une réponse du chatbot.
```
    response = await send_to_rasa(message)
    return response
```
Envoie un message à RASA et retourne la réponse.

📁 app/services/rasa.py
```
import httpx
from app.core.config import settings
```
Utilise httpx pour envoyer une requête asynchrone vers RASA.

```
async def send_to_rasa(message: str) -> str:
    async with httpx.AsyncClient() as client:
        try:
            response = await client.post(
                settings.RASA_URL,
```

```
            json={"sender": "user", "message": message},
            timeout=5.0
        )
```
Envoie une requête POST contenant le message.
```
        response.raise_for_status()
        data = response.json()
```
Vérifie les erreurs et lit les résultats au format JSON.
```
        if data and isinstance(data, list):
            return " ".join([d.get("text", "") for d in data])
```
Si la réponse est une liste, concatène tous les textes.
```
        return "RASA did not return a response."
    except httpx.RequestError as e:
        return f"Request to RASA failed: {e}"
```
Si une erreur survient, retourne un message d'erreur.

📁 app/utils/logging.py
```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```
Configure le système global de journalisation (logging) pour l'application.

📁 app/utils/helpers.py
```
# Fonction utilitaire par exemple : vérifier les données d'entrée
def is_valid_message(message: str) -> bool:
    return bool(message and isinstance(message, str) and message.strip())
```
Fonction utilitaire pour vérifier si un message est valide (non vide, de type string).

📁 requirements.txt
```
fastapi
uvicorn
```
serveur web
```
sqlalchemy
aiosqlite
```
gestion BDD async
```
httpx
```
requêtes async
```
python-jose
```
gestion des JWT

📁 tests/test_api.py

def test_dummy(): Petit test de logique.
    assert 1 + 1 == 2

📁 tests/test_services.py
def test_service():
    from app.services.chat import get_chat_response
    import asyncio
    result = asyncio.run(get_chat_response("Hello"))
    assert isinstance(result, str)
        Test get_chat_response("Hello") et vérifie si le résultat est de type string.