Documentación RAG Chatbot
Promtior

1. **Introducción y contexto del proyecto**

- **Introducción**
  Se pidió desarrollar y desplegar un **chatbot asistente** basado en la arquitectura **RAG (Retrieval Augmented Generation)**, utilizando la librería **LangChain**, capaz de responder preguntas sobre el contenido del sitio web de Promtior y documentos adicionales.
- **Desafíos**
  - Lenguaje python
  - Subir el proyecto a un entorno cloud
  - Tipo de proyecto con IA

- **Decisiones iniciales**
  - Primera elección: **Ollama**
    Para el desarrollo local, se optó por usar **Ollama** como motor de lenguaje. Esta decisión permitió correr modelos como LLaMA directamente en la máquina de desarrollo.
    - Ventaja: el modelo funcionaba muy bien en local, con respuestas rápidas respondiendo preguntas con certeza y contenido prolijo con LangChain.
    - Resultado: el chatbot quedó operativo y con buen rendimiento en el entorno local.

- **Problemas encontrados**
  - Limitaciones de **Ollama** en **Railway**
    Al intentar desplegar el proyecto en Railway, surgió un problema: **Ollama** requiere **mucha memoria**.
    - **Railway no soporta** la instalación de **Ollama**.
    - Los modelos son pesados para los límites de CPU/memoria del servicio.
    - Esto hacía inviable el despliegue.

- **Decisiones finales**
  - **Migración a OpenAI**
    Para poder desplegar el chatbot en Railway, se decidió **reemplazar Ollama por OpenAI** como motor de embeddings y LLM, lo que llevó a sustituir código y un refator.
    - Se configuró el uso de OpenAIEmbeddings y ChatOpenAI con modelos como **text-embedding-3-large** y **gpt-4o-mini**.
    - Se gestionó la clave de acceso mediante la variable de entorno **OPENAI_API_KEY** la cual es paga.
    - Se mantuvo la arquitectura RAG con FAISS como vectorstore y LangServe para exponer el endpoint `/chat`.

- **Resultado**

El chatbot quedó desplegado en Railway, accesible públicamente por la siguiente url, (https://promtior-rag-chatbot-production-f5b8.up.railway.app/chat/playground/), capaz de responder preguntas sobre el sitio de Promtior teniendo como fuente el pdf enviado y la url de la empresa. La decisión de migrar a OpenAI garantizó compatibilidad con el entorno cloud, aunque implicó depender de créditos de API.

2. Estructura del proyecto

- **app/loaders.py**

  Responsable de cargar el conocimiento desde distintas fuentes:
  - WebBaseLoader + Playwright:
    - _load_web_playwright(urls) → usa Playwright para renderizar páginas con JavaScript levantando la pagina con Chrome/Chromium y dejamos la información un poco mas manejable con BeautifulSoup
    - _load_web_fallback(urls) → fallback con WebBaseLoader si Playwright no está disponible (ej. Railway).
  - PDFs:
    - Usa DirectoryLoader + PyPDFLoader para cargar PDFs como documentos LangChain (texto plano) desde data/.
- **app/main.py**

  Punto de entrada de la aplicación:

  - Importamos el RAG
    from app.rag_chain import create_rag_chain

  - Inicializamos FASTAPI
    app = FastAPI(
      title="Promtior RAG Chatbot",
      version="1.0",
      description="Chatbot using RAG + LangChain"
)
  - Se instancia el RAG chain definido en **rag_chain.py**
    qa_chain = create_rag_chain()

  - Exponemos el chain como endpoint
    add_routes(
      app,
      qa_chain,
      path="/chat"
)

- **app/rag_chain.py**

  - Variables de entorno (.env)
    OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
    OPENAI_CHAT_MODEL = os.getenv("OPENAI_CHAT_MODEL", "gpt-4o-mini")

OPENAI_EMBEDDING_MODEL =
os.getenv("OPENAI_EMBEDDING_MODEL", "text-embedding-3-large")

- Función **create_rag_chain**
  - **Carga de documentos**
    docs = load_documents()

  - **Divide los documentos en fragmentos de 500 caracteres con solapamiento de 50**
    splits = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50).split_documents(docs).

  - **Crear embeddings (vector numérico) y LLM**
    embeddings = OpenAIEmbeddings(model=OPENAI_EMBEDDING_MODEL, api_key=OPENAI_API_KEY)
    llm = ChatOpenAI(model=OPENAI_CHAT_MODEL, temperature=0, api_key=OPENAI_API_KEY)

  - **Vectorstore FAISS**
    vectorstore = FAISS.from_documents(splits, embeddings)
    retriever = vectorstore.as_retriever(k=10)

  - **Prompt**
    Definimos las reglas del asistente
    prompt = ChatPromptTemplate.from_template("""
    You are a question-answering assistant.
    RULES:
    - Use ONLY the information in the context below.
    - If the answer is not in the context, reply: "I don't have information about that in the provided documents."
    ...
    """)

El flujo del RAG comienza cargando los documentos desde web o PDF. Luego se dividen en **chunks** pequeños para que el modelo los procese mejor. Cada chunk se convierte en un **embedding**, un vector numérico que lo representa, se guarda en **FAISS**, la base vectorial. El **Retriever** se encarga de consultar FAISS y devolver los fragmentos más relevantes como contexto para responder la pregunta.

3. **Diagrama**
   Se presenta el diagrama en la carpeta doc

4. **Conclusiones**

- Decisión clave: Migrar de Ollama a OpenAI para poder desplegar en Railway.
- Arquitectura: FastAPI + LangServe exponen el RAG chain como servicio REST.
- Carga de datos: Web + PDFs, con fallback para Railway.
- Archivos .env para credenciales y requirements.txt con versiones fijadas.