**Visión General del Backend**

1. **Flask + Python**:  
   El backend está construido en Python utilizando Flask. Se exponen diferentes endpoints —por ejemplo, "/chat\_stream"— que reciben las peticiones del front-end (o del usuario).
2. **Arquitectura por “chain”**:  
   El núcleo de la lógica vive en un “pipeline” definido en el archivo assistant\_chain\_lcel.py. Esto utiliza la **LangChain Expression Language (LCEL)**, lo que permite orquestar varios pasos (Runnables) de forma secuencial o paralela.
3. **Objetivo**:  
   Responder preguntas del usuario, opcionalmente utilizando RAG (Retrieval-Augmented Generation) para buscar fragmentos de texto relevantes en una base documental (via GroundX). Si la pregunta no requiere RAG, se hace una respuesta “genérica”. Finalmente se construye un mensaje final y se transmite por streaming al front-end.
4. **conditional\_rag\_retriever**:
   * Verifica si la pregunta amerita buscar información en GroundX (chequea “should\_call\_groundx” en rag\_service.py).
   * Si NO hace falta RAG, pone context = "No documents...".
   * Si SÍ, ejecuta una búsqueda paralela:
     + **Spanish**: Llama a la base con el query original.
     + **English**: Primero traduce la pregunta al inglés, luego llama a la base.
   * Se fusionan ambos resultados en un solo texto “context”.
5. **summarizer\_lcel**:
   * Opcionalmente, si el historial de conversación es muy grande, resume la parte antigua.
   * Lo importante es que en cada paso se copian los keys como "domain", "company", etc., para no perderlos.
6. **final\_prompt\_generator**:
   * Toma la variable "system\_prompt", "company", "domain", el context resultante y el query.
   * Construye el mensaje final para el LLM (mezclando prompt base, historial, y contexto RAG).
7. **chat\_model**:
   * Es la llamada final al modelo (p. ej. gpt-4o-mini) en modo streaming. Emite tokens en chunks hacia Flask, que los reenvía al cliente en tiempo real.

**Archivo assistant\_chain\_lcel.py**

* **Define** la *chain principal*, usando:
  + **RunnableParallel** para correr en **paralelo** la búsqueda en español vs. la traducción+inglés.
  + **Módulos**:
    - summarizer\_lcel (para recortar historial),
    - final\_prompt\_generator\_fn (para armar el prompt final),
    - chat\_model (para el streaming de la respuesta).
* **Exposición** de la chain:
  + Una función run\_chain\_stream(chain\_input, run\_id, session\_id) que se invoca desde Flask para procesar la pregunta y **stream** de la respuesta.
* **Memory** (conversación):
  + Se utiliza RunnableWithMessageHistory para guardar y recuperar el historial de la conversación a lo largo de varios turnos.

**Archivo rag\_service.py**

* **Encapsula** la lógica de RAG:
  + should\_call\_groundx(query) define si hace falta RAG.
  + groundx\_search\_spanish\_only(...) y groundx\_search\_english\_only(...) hacen las consultas a GroundX.
  + translate\_spanish\_to\_english(...) utiliza un modelo (OpenAI) para traducción cuando se requiere la vía en inglés.
* **Fusión**:
  + Ya no ocurre dentro de rag\_service. En la nueva versión, se hace en merge\_rag\_fn dentro del chain, para permitir la ejecución **paralela** de las búsquedas.

**Resumen**

La arquitectura se basa en:

* **Un backend Flask** que expone endpoints.
* **Una “chain”** en assistant\_chain\_lcel.py que define paso a paso la orquestación (conditional RAG, summarización, prompt final, LLM).
* **Runnables en paralelo** para español e inglés, combinados en un “merge” final.
* **Streaming** de la respuesta parcial del LLM hasta el front-end.

Así se logra un pipeline flexible para la búsqueda de contexto (RAG), la construcción de prompts y la emisión de la respuesta final, **sin perder** las variables importantes (company, domain, etc.) a lo largo del camino.