

ChatBot BILL v2

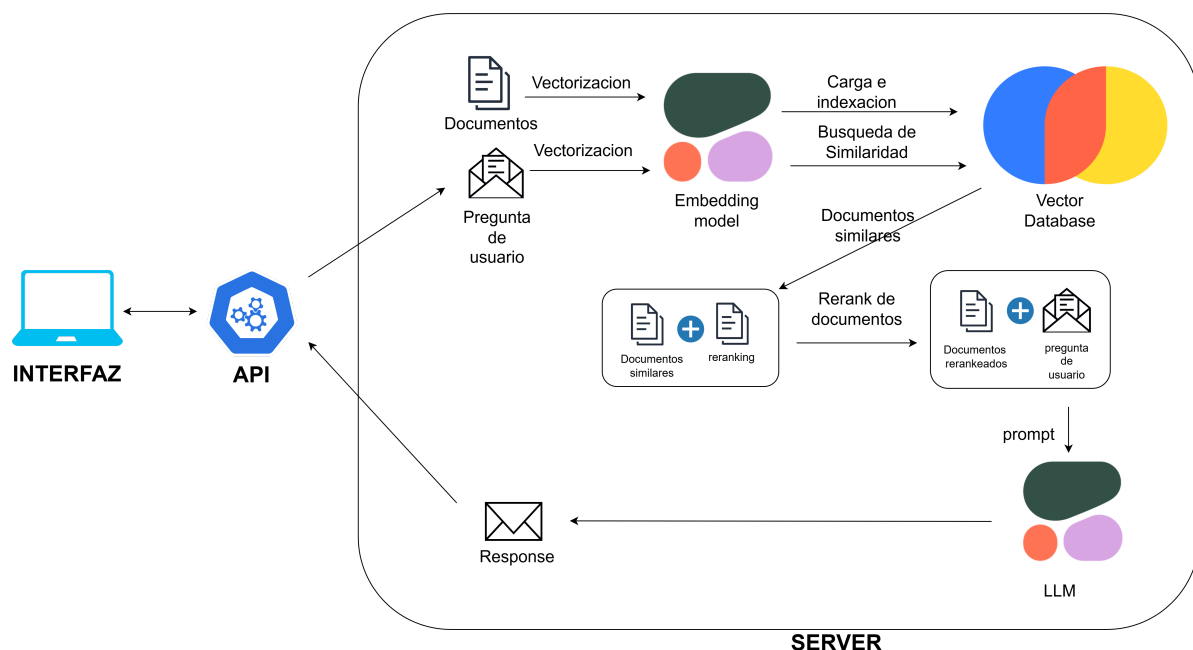
Bill es un asistente virtual diseñado para ayudar a los estudiantes en su preparación para el examen **Microsoft Certified: Azure AI Engineer Associate (AI-102)**.

Objetivo

El objetivo de **Bill** es ofrecer soporte en tiempo real para consultas sobre la guía de estudio y repasar temas para los estudiantes que se están preparando para el examen AI-102, mejorando la eficiencia y efectividad en su preparación.

Bill asistirá con la guía de estudio, repasando conceptos clave y respondiendo preguntas, facilitando así un apoyo integral durante el proceso de estudio.

Arquitectura



La arquitectura trabajada consta en 3 grandes bloques de funcionamiento que serán definidos a continuación:

- Interfaz
- Api

- Server

Interfaz

La interfaz de usuario está construida con Streamlit. Las funciones se localizan en el archivo `app.py` y se ejecuta con el comando:

```
streamlit run app.py
```

La interfaz de **Streamlit** proporciona un entorno interactivo y fácil de usar para la interacción con el chatbot.

API

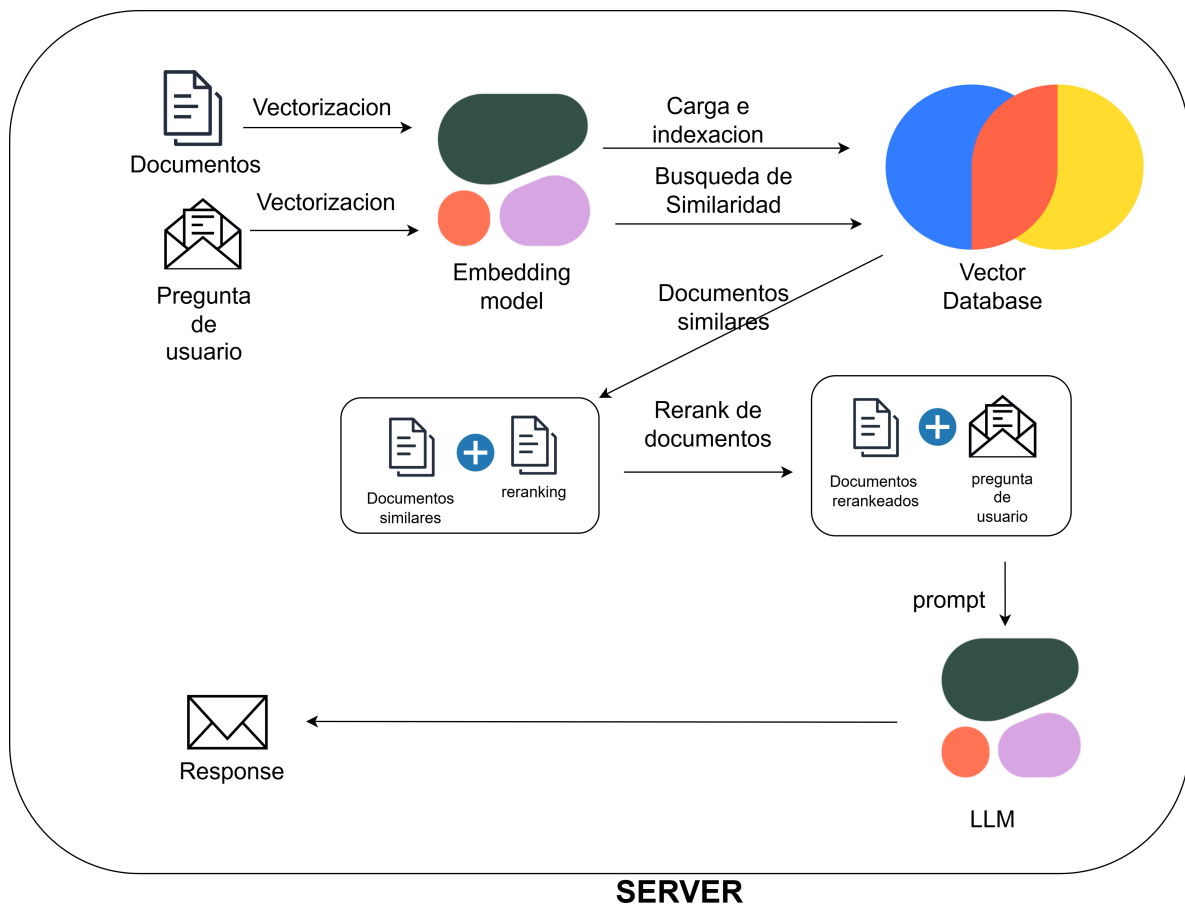
La API está creada con FastAPI. Se encarga de manejar las consultas de los usuarios y generar las respuestas. El código se organiza en diferentes capas para mantener la modularidad y la claridad:

- **main:** Contiene la entrada a la API.
- **routers:** Define el endpoint POST que recibe una consulta del usuario (en formato de texto) y devuelve la respuesta.
- **models:** Define las clases de **Pydantic** que se utilizan para validar la entrada y salida de datos.
- **database:** Contiene las funciones para interactuar con la base de datos **ChromaDB** y **Cohere**.
- **app:** Contiene el código que maneja la interfaz de usuario en **Streamlit**.

se ejecuta con el comando:

```
uvicorn main:app --reload
```

Server y Arquitectura RAG



1. Preparación y Carga de Datos

En el archivo `load_database.py` se definieron las funciones:

- **Conexiones:**
 - `connect_cohere()` : Realiza la conexión a Cohere.
 - `connect_database()` : Realiza la conexión con la base de datos vectorial ChromaDB.
- **Crear Colección:**
 - `get_embeddings()` : Genera los embeddings de los textos con el modelo `embed-multilingual-v3.0` de Cohere, utilizando el `input_type` `search_document` que es el adecuado para vectorizar documentos de texto.
 - `MyEmbeddingFunction` : Implementación personalizada de la clase `EmbeddingFunction` de ChromaDB que utiliza la función `get_embeddings()` para convertir los textos en embeddings.

- Se crea una instancia de cliente persistente pasándole como path la constante `PATH_CONEXION`.
- Se crea una colección, se le pasa el nombre a través de una constante `COLLECTION_NAME`, se le asigna la función de embedding personalizada, y se especifica que el algoritmo a usar es el de similitud de coseno porque es el recomendado para realizar búsquedas de similaridad en textos.
- **Preparación de Datos:**
 - La información utilizada por **Bill** se extrajo de la guía de estudio oficial disponible en la página de Microsoft. Los módulos de la guía se cargaron en documentos PDF.
 - `read_pdf()` : Lee un archivo PDF y extrae todo el texto contenido en él.
 - `preparar_fragmentos_metadatos()` : Toma el texto extraído de un PDF y lo divide en fragmentos (chunks) de tamaño determinado. Además, agrega metadatos a cada fragmento para identificar el módulo o título del contenido. Los fragmentos se almacenan como una lista de diccionarios que contiene tanto el texto como los metadatos asociados.
 - Utiliza el **RecursiveCharacterTextSplitter** de **LangChain** para dividir el texto en fragmentos. Se eligieron los valores `chunk_size=2000` y `chunk_overlap=259` porque estos eran los valores adecuados para dividir el texto sin que se pierda el significado, asegurando que el contenido mantuviera la consistencia y no se interrumpieran ideas clave entre fragmentos.
- **Carga de Datos:**
 - **Definición de `lista_pdfs`** : Se crea un diccionario que asocia los títulos de los módulos con las rutas de los archivos PDF correspondientes.
 - Se inicializa la lista `all_chunks` vacía para almacenar los fragmentos de texto de todos los documentos. Luego, se itera sobre los elementos de `lista_pdfs` utilizando la función `read_pdf()`, que extrae el texto del archivo.
 - `preparar_fragmentos_metadatos()` : Divide el texto en fragmentos y añade el título del módulo como metadatos.
 - Se establece la conexión con la base de datos utilizando `connect_database()`, y luego se cargan todos los fragmentos de texto y sus metadatos a la colección de la base de datos vectorial.

2. Retrieve

- **get_query_embeddings()**: Convierte la entrada del usuario (consulta) en embeddings utilizando el modelo de embedding `embed-multilingual-v3.0` de **Cohere** con el parámetro `input_type="search_document"`, ya que este tipo de entrada está diseñado específicamente para búsquedas por similitud.
- **get_documents()**: Toma la entrada del usuario, genera los embeddings correspondientes y realiza una búsqueda en la colección de **ChromaDB** para encontrar los documentos más relevantes basados en la similitud semántica.

3. Reader

- **realizar_reranking()**: Realiza un reordenamiento de los documentos recuperados, usa el modelo de **Cohere** para realizar un reordenamiento (reranking) de los documentos según la relevancia.
- **rag_answer()**: Genera respuestas utilizando el enfoque **Retrieval-Augmented Generation (RAG)**. Esta función toma como entrada una pregunta del usuario, realiza un reordenamiento de los documentos relevantes y luego genera una respuesta personalizada utilizando el modelo `command-r-plus-08-2024` de **Cohere**.
- **chatbot()**: Maneja las interacciones básicas con el usuario, como saludos y despedidas, antes de delegar las consultas más complejas a la función `rag_answer()`. Esta función responde con un saludo adecuado si la entrada es una de las frases predefinidas, y si no lo es, redirige la consulta a la función **RAG** para obtener una respuesta más compleja.