

Guía de Arquitectura y Funcionamiento Interno del Bot Normativo.

I. GLOSARIO

- **API (Interfaz de Programación de Aplicaciones):** Conjunto de reglas y definiciones que permiten que las aplicaciones se comuniquen entre sí, facilitando la integración de servicios externos en un sistema.
- **Backend:** Parte del sistema que maneja la lógica, base de datos y la interacción con servicios externos. Es el “lado del servidor” de una aplicación.
- **Chatbot:** Aplicación que utiliza inteligencia artificial (IA) y procesamiento de lenguaje natural (PLN) para interactuar con usuarios de forma automatizada a través de mensajes escritos.
- **Flask:** Un micro-framework para Python, utilizado para desarrollar aplicaciones web. En este proyecto, se utiliza para gestionar las rutas, recibir solicitudes y devolver respuestas.
- **Frontend:** Parte de la aplicación que interactúa directamente con el usuario. Es la interfaz visual (HTML, CSS, JavaScript) que se muestra en el navegador.
- **Pinecone:** Plataforma que proporciona bases de datos vectoriales para realizar búsquedas rápidas basadas en semántica. En este proyecto, se usa para almacenar y consultar datos normativos.
- **Embedding:** Representación numérica (vector) de un texto o documento, utilizado en procesamiento de lenguaje natural (PLN) para comparar y buscar patrones semánticos.
- **GPT (Generative Pretrained Transformer):** Modelo de lenguaje basado en inteligencia artificial creado por OpenAI que puede generar texto coherente en respuesta a preguntas o indicaciones. Es el modelo utilizado en este proyecto para generar respuestas normativas.
- **JavaScript (JS):** Lenguaje de programación utilizado en el frontend para crear interactividad en las páginas web, como la manipulación del DOM y la gestión de eventos.
- **LangChain:** Un marco de trabajo en Python utilizado para construir aplicaciones basadas en modelos de lenguaje como GPT-3 o GPT-4, permitiendo el uso de herramientas como bases de datos vectoriales y otras funcionalidades avanzadas.
- **LLM (Large Language Model):** Modelo de lenguaje de gran tamaño, como GPT, que ha sido entrenado en grandes cantidades de datos para comprender y generar lenguaje humano.

- **OpenAI:**
- **PineconeVectorStore:** Implementación específica de Pinecone para almacenar y recuperar vectores (embeddings) asociados con los datos normativos para realizar consultas rápidas y efectivas.
- **Prompt:** Instrucción o pregunta formulada a un modelo de lenguaje como GPT para generar una respuesta. En este proyecto, se utiliza un prompt con el contexto normativo para obtener respuestas sobre regulaciones.
- **REST API:** Arquitectura de comunicación entre sistemas que usa solicitudes HTTP para obtener o modificar datos. Este sistema utiliza una API REST para que el frontend y el backend se comuniquen.
- **JSON:** Formato de intercambio de datos ligero y fácil de leer/escribir. Se utiliza para enviar y recibir datos entre el cliente y el servidor en este sistema.

II. INTRODUCCIÓN

El presente sistema, denominado Bot Normativo, es una aplicación web desarrollada con Python que permite realizar consultas automáticas sobre normativas, regulaciones o información estructurada a través de una interfaz de chatbot. Su propósito principal es facilitar el acceso a contenidos normativos mediante un lenguaje natural, aprovechando tecnologías de procesamiento de lenguaje y bases de datos vectoriales.

La arquitectura del sistema integra distintos componentes:

- Un servidor backend en Flask, responsable del manejo de rutas, procesamiento de mensajes del usuario, conexión con modelos de lenguaje y gestión de consultas.
- Un frontend web interactivo, construido con HTML, CSS y JavaScript, que permite a los usuarios realizar preguntas y recibir respuestas en tiempo real.
- Un módulo de integración con Pinecone, utilizado como base de datos vectorial para realizar búsquedas semánticas eficientes sobre documentos previamente procesados.

Los archivos principales de la carpeta llamada BOTNORMATICO_CON_ENV son los siguientes:

- app.py: Archivo principal del backend que contiene la lógica del servidor web desarrollado con Flask. Gestiona las rutas HTTP, la conexión con servicios externos y el manejo de mensajes del chatbot.
- pinecone_embed.py: Módulo del backend encargado de la integración con Pinecone para búsquedas vectoriales, utiliza embeddings de OpenAI para realizar consultas semánticas al índice y generar respuestas mediante

modelos GPT. También contiene la definición del prompt y la lógica para el procesamiento de las preguntas.

- package.json: Archivo de configuración del frontend que define las dependencias JavaScript necesarias, scripts para automatización y otras configuraciones relacionadas con el entorno de desarrollo web.
- script.js: Archivo JavaScript que controla la interacción dinámica en el frontend, manejando el envío de mensajes, la recepción de respuestas y la actualización de la interfaz de usuario
- style.css: Hoja de estilos CSS que define la apariencia visual y el diseño responsivo de la aplicación web, garantizando una experiencia de usuario coherente y atractiva.
- chat.html: Plantilla HTML principal que conforma la interfaz del chatbot, donde el usuario puede ingresar preguntas y visualizar las respuestas en tiempo real.
- admin.html: Página administrativa que permite supervisar y gestionar las conversaciones registradas, facilitando la revisión y eliminación de preguntas y respuestas almacenadas.

III. IMPLEMENTACIÓN DE CÓDIGO

1) pinecone_embed.py

```
!pip install langchain-core -q
!pip install langchain-openai -q
!pip install langchain-community -q
!pip install -qU langchain-pinecone -q
!pip install langchain-unstructured -q
!pip install pinecone
```

Figura 1. Librerías utilizadas.

Primero, se instalan las librerías necesarias para la ejecución del código. La Figura 1 muestra cuáles fueron instalados.

Langchain-core, esta es la librería base ya que proporciona las clases y funciones fundamentales para construir modelos de lenguaje, incluyendo manejo de cadenas de textos, prompt, lógica de ejecución y control de flujo entre componentes.

Langchain-openai, esta librería permite integrar modelos de OpenAI dentro de una aplicación en langchain en donde puedo tener las funciones principales tales como enviar prompts a los modelos de OpenAI, obtener y manejar respuestas, configurar parámetros como temperatura, número de tokens, etc.

Langchain-community, Contiene conectores y herramientas creadas por la comunidad LangChain. Incluye integraciones no oficiales o en desarrollo con servicios de terceros, como por ejemplo conectores para bases de datos, APIs, herramientas de scraping, etc.

Langchain-pinecone, Ofrece integración con **Pinecone**, un servicio de vectorización y búsqueda semántica, sirve para almacenar y buscar documentos vectorizados, realizar búsquedas basadas en similitud de significado (no solo palabra clave).

Langchain-unstructured, integra langchain con la librería de unstructured, lo que permite extraer texto estructurado desde archivos no estructurados como pdfs, imágenes, Word, etc. Su función clave en este proyecto es leer y fragmentar texto/documentos complejos y muy extensos.

Instalar la librería de cliente oficial de pinecone “!pip install pinecone”, es la que permite conectarse directamente al servicio Pinecone desde Python, y esta me permite crear y administrar índices de vectores, instalar, actualizar y eliminar vectores y a su vez realizar búsquedas de similitud.

NOTA: Aunque langchain-pinecone permite la integración con Pinecone, es necesario instalar también pinecone, ya que contiene el cliente oficial que permite conectarse directamente al servicio desde Python.

```
import os
os.environ["OPENAI_API_KEY"]="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
os.environ["PINECONE_API_KEY"]="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Figura 2. Establecimiento de la llave de uso de la API de OpenAI y de Pinecone.

En la Figura 2 se configuran las claves de API de OpenAI y Pinecone. La clave de OpenAI es imprescindible para autenticar y facturar las solicitudes a los modelos de lenguaje, garantizando así un acceso continuo y preciso al servicio. Por su parte, la clave de Pinecone permite crear y gestionar índices vectoriales: añadir, actualizar o eliminar vectores según sea necesario para optimizar la recuperación semántica. Mantener ambas claves correctamente actualizadas es fundamental para el correcto funcionamiento del sistema.

```
import os
from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatOpenAI
from langchain.chains import create_retrieval_chain
from langchain_pinecone import PineconeVectorStore
from langchain.chains import RetrievalQA
from pinecone import Pinecone
from langchain_openai import OpenAIEmbeddings
```

Figura 3. Importación de librerías al espacio de trabajo.

También se lleva a cabo la importación de componentes específicos de las bibliotecas previamente instaladas, incorporándose al espacio de trabajo como se ve en la [Figura 3](#). Este proceso es esencial para acceder a las funcionalidades particulares de estas bibliotecas durante la ejecución del código.

```
# Inicializar Pinecone
pc = Pinecone(api_key=os.environ["PINECONE_API_KEY"]) → 1

# Nombre del índice en Pinecone
index_name = "integrational" → 2

# Modelo de embeddings de OpenAI
model_name = "text-embedding-ada-002" → 3

embed = OpenAIEmbeddings(
    model=model_name,
    openai_api_key=os.environ["OPENAI_API_KEY"]
) → 4
```

Figura 4. Inicialización de Pinecone y OpenAI Embeddings.

La primera línea (1) de código, inicializa la conexión con nuestra base de datos utilizada "Pinecone". Se utiliza la API obtenida desde las variables de entorno del sistema, permitiendo así al código autenticar y acceder a los servicios de Pinecone de manera segura.

La segunda línea (2) de código, únicamente se define el nombre del índice que se utilizará en Pinecone donde se almacenaran los vectores que representan la información procesada por los modelos de OpenAI.

La tercera línea (3) de código se especifica el modelo de embedding de OpenAI que se utilizará para convertir el texto en vectores.

Finalmente se crea un objeto (4) de la clase OpenAIEmbeddings de la librería correspondiente, en donde este objeto fue configurado con el tipo de modelo escogido y la API lo cual es esencial para permitirle al código interactuar con los modelos de OpenAI.

```
vector_store = PineconeVectorStore.from_existing_index(index_name=index_name, embedding=embed)
```

Figura 5. Conexión al índice que se utilizará en Pinecone.

Con la instrucción de código especificada en la [Figura 5](#) se inicializa un cliente conectado al índice existente en Pinecone, empleando el modelo de embeddings configurado. Esto permite realizar consultas por búsquedas de similitud de forma rápida y sencilla.

```
# Definir la plantilla de pregunta para el chatbot
def get_answer(query):
    question = query
    template = """INSTRUCCIONES PARA EL PROMPT
{context}
Question: {question}
Helpful Answer:"""
    QA_CHAIN_PROMPT = PromptTemplate.from_template(template)

    llm = ChatOpenAI(model_name="gpt-4", temperature=0.1)
    qa_chain = RetrievalQA.from_chain_type(
        llm,
        retriever=vector_store.as_retriever(),
        chain_type_kwargs={"prompt": QA_CHAIN_PROMPT}
    )
    result = qa_chain({"query": question})
    return result["result"]
```

Figura 6. Función para procesar preguntas y generar respuestas basadas en contexto.

Finalmente se define la función que se muestra en la [Figura 6](#) la cual es la encargada de recibir una pregunta, la procesa para buscar información relevante en un almacén vectorial, construye un prompt con contexto, y envía la consulta a un modelo GPT-4 configurado para responder de manera precisa y útil. Luego, retorna la respuesta generada. Esta función está estructurada de la siguiente manera:

- a. **Pregunta del usuario:** La pregunta planteada por el usuario.
- b. **Template:** Es una cadena de texto que define el prompt que se enviará a la API de OpenAI. Aunque se podría enviar únicamente la pregunta del usuario, utilizar un prompt estructurado permite incorporar contexto adicional, lo cual mejora significativamente la calidad y precisión de las respuestas del modelo. En esta función, el template cumple varios propósitos: contextualiza al chatbot, establece directrices que delimitan su comportamiento, e integra espacios específicos para incluir el contexto recuperado desde la base de datos, la consulta del usuario y la respuesta generada por el modelo.
- c. **Configuraciones del modelo:** Se especifica el modelo a utilizar (en este caso, GPT-4) y el parámetro de temperatura, que controla el grado de aleatoriedad en la generación de texto. La temperatura influye en el equilibrio entre precisión y creatividad: valores bajos (como 0.1) hacen que el modelo

produzca respuestas más conservadoras y predecibles, mientras que valores altos (como 1.0) favorecen respuestas más variadas y creativas, aunque con mayor riesgo de incoherencia o inexactitud. Dado que el objetivo del chatbot es proporcionar información factual y confiable, se optó por una temperatura baja.

2) app.py

El siguiente código es un script en Python que utiliza el framework web Flask para implementar un sistema básico de chat con un chatbot:

a. Importación de módulos

```
from flask import Flask, render_template, request, jsonify, redirect
from pinecone_embed import get_answer
```

Figura 7. Importación de funciones y clases al archivo app.py

En la Figura 7 se muestra cómo se importa la función `get_answer` desde el módulo `pinecone_embed.py`, además se observa la importación de las clases y funciones principales del framework. Flask Esta función es la encargada de manejar la lógica principal del chatbot, incluyendo la interacción con Pinecone para realizar búsquedas vectoriales y la generación de respuestas mediante el modelo GPT.

b. Inicialización de Flask, ruta principal y almacenamiento temporal de conversaciones

```
app = Flask(__name__)

@app.route('/')
def home():
    return render_template('chat.html')

conversations = []
```

Figura 8. Creación de la instancia principal de la clase flask en app.py

Como se logra ver en la Figura 8, en primer lugar, se crea una instancia para la aplicación Flask la cual actúa como un servidor web encargado de manejar solicitudes y respuestas HTTP. Seguido de esto se define la ruta que se encarga de devolver la plantilla HTML que es la interfaz grafica principal donde el usuario puede interactuar con el chatbot. Y finalmente se declara una lista vacía en donde se estaría almacenando

el historial temporal de mensajes intercambiados entre el usuario y el chatbot durante la sesión activa.

c. Envío de mensajes

```
@app.route('/send_message', methods=['POST'])
def send_message():
    user_message = request.form['user_message']

    # Llama a la función get_answer para obtener la respuesta del chatbot
    bot_response = get_answer(user_message)
    #registra la conversación
    conversations.append({"user_message": user_message, "bot_response": bot_response})

    return jsonify({'bot_response': bot_response})
```

Figura 9. Procesamiento de solicitudes POST y respuesta del chatbot

En la [Figura 9](#) Se continúa definiendo una ruta para recibir mensajes enviados desde el frontend mediante un método POST. La función asociada procesa el mensaje del usuario, llama a la función `get_answer` para generar la respuesta del chatbot, y registra la interacción en la lista de conversaciones. Finalmente, envía la respuesta del bot al cliente en formato JSON para que sea mostrada en la interfaz. Esta estructura es importante ya que permite gestionar la comunicación entre el usuario y el chatbot de manera dinámica y asíncrona, facilitando la interacción en tiempo real.

d. Visualización y gestión del historial de conversaciones

```
@app.route("/conversations", methods=["GET"])
def get_answered_conversations():
    # Devuelve las conversaciones como JSON
    return render_template("admin.html", answered=conversations)
```

Figura 10. Definición de ruta para mostrar conversaciones registradas en la interfaz administrativa.

Esta ruta define un endpoint que, al ser accedido mediante una solicitud GET, devuelve la página administrativa “admin.html” mostrando el historial de conversaciones almacenadas. Su función principal es permitir la visualización y supervisión de las interacciones entre los usuarios y el chatbot.

e. Ruta para eliminar preguntas del historial de conversaciones

```
@app.route("/delete_question", methods=["POST"])
def delete_question():
    question_to_delete = request.form.get("question")

    for conversation in conversations:
        if conversation["user_message"] == question_to_delete:
            conversations.remove(conversation)
            break

    return redirect("/conversations")

if __name__ == '__main__':
    app.run()
```

Figura 11. Gestión y limpieza del historial de conversaciones mediante POST

Esta ruta recibe solicitudes POST para eliminar una pregunta específica del historial de conversaciones almacenado en el servidor. Al recibir el texto de la pregunta a eliminar, busca esa entrada en la lista `conversations` y, si la encuentra, la remueve. Finalmente, redirige al usuario nuevamente a la página de visualización de conversaciones.

f. Bloque principal de ejecución

```
✓ if __name__ == '__main__':
    app.run()
```

Figura 12. Definición del bloque principal de ejecución en `app.py`

En la [Figura 12](#) Se utiliza para asegurar que el servidor Flask se ejecute únicamente cuando el archivo `app.py` se ejecuta directamente, y no cuando es importado como un módulo en otro script. La llamada a `app.run()` inicia el servidor web local de Flask, poniendo la aplicación en funcionamiento para recibir y atender las solicitudes HTTP.

3) Implementación de la página web

Cuando ingresamos la dirección de una página web en un navegador (por ejemplo, Chrome o Firefox), este envía una solicitud al servidor donde se encuentra alojada la página, utilizando protocolos estándar como TCP/IP y HTTP. Si la solicitud es exitosa, el servidor responde enviando un conjunto de archivos que el navegador procesa para mostrar el contenido al usuario.

Entre estos archivos, el más importante es el código HTML, que define la estructura y el contenido de la página web. Sobre esta estructura se aplican estilos visuales mediante CSS, los cuales determinan la apariencia, colores, tipografías y disposición de los elementos. Finalmente, el comportamiento dinámico e interactivo de la página es gestionado por JavaScript, que permite manejar eventos, actualizaciones en tiempo real y la comunicación con el servidor (Soriano, 2022).

ES importante mencionar que para la implementación de este proyecto, el servidor web se despliega en la plataforma en la nube PythonAnywhere, seleccionada por su facilidad para ejecutar aplicaciones desarrolladas con Flask, un framework liviano y flexible de Python orientado a la creación rápida de aplicaciones web.

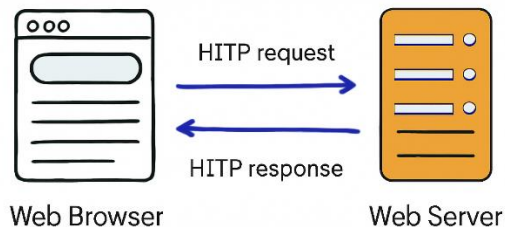


Figura 13. Ciclo de petición y respuesta HTTP entre navegador y servidor

La Figura 13 ilustra el proceso fundamental de comunicación en la web, donde un navegador (cliente) envía una solicitud HTTP (HTTP request) al servidor web, y este responde enviando una respuesta HTTP (HTTP response) con los archivos necesarios para mostrar la página solicitada.

El servidor procesa la solicitud y entrega principalmente código HTML, CSS y JavaScript, que el navegador interpreta para construir la página web visible y funcional para el usuario.

