-Me encanta la iniciativa. Como profesor, te guiaré en la creación de un **Chatbot de Dominio Específico con Inteligencia Conversacional**. Este es un proyecto fantástico que abarca desde el procesamiento del lenguaje natural hasta la arquitectura de software.

**Visión General del Proyecto: "BizChat Assistant"**

Vamos a construir un asistente virtual que:

1. **Comprenda el lenguaje natural** (no solo comandos predefinidos).
2. **Mantenga un diálogo contextual**, recordando lo que se ha hablado en la conversación.
3. **Responda preguntas específicas** de un dominio o negocio que elijas (ej: una tienda online, un consultorio médico, soporte técnico de una app, etc.).
4. **Sea extensible**, para que puedas añadirle nuevas funcionalidades fácilmente en el futuro.

**Características Técnicas y Componentes**

1. **Backend (El Cerebro):** Una API REST en Python usando un framework moderno (FastAPI).

Una **API REST** es un conjunto de reglas y convenciones que permite que aplicaciones distintas se comuniquen de forma estandarizada, simple y eficiente a través de Internet usando el protocolo HTTP.

1. **Procesamiento de Lenguaje Natural (NLP):** Usaremos la biblioteca spaCy para tareas básicas de NLP (análisis léxico, similitud semántica) y scikit-learn para similitud de coseno. Más adelante, podemos integrar modelos de LLM (como GPT con OpenAI API) para hacerlo mucho más potente.
2. **Base de Conocimiento:** Un conjunto de **Preguntas y Respuestas (Q&A)** sobre tu negocio. Empezaremos con algo simple como un archivo JSON. Luego, lo podemos migrar a una base de datos.
3. **Memoria de Conversación:** Implementaremos un sistema simple para recordar el contexto de la conversación reciente.
4. **Frontend (La Cara):** Una interfaz web simple pero funcional usando HTML, CSS y JavaScript para chatear con nuestro backend. Usaremos Fetch API para la comunicación.

**Tecnologías que Aprenderás y Practicarás**

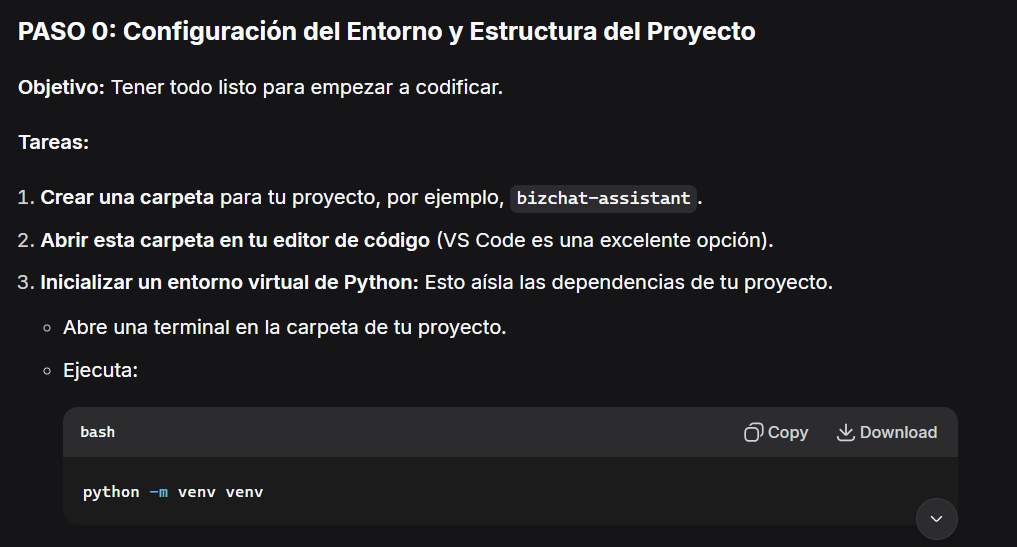
* **Python:** Lógica de programación, manejo de datos.
* **FastAPI:** Creación de APIs REST modernas y rápidas, autodocumentación con Swagger/OpenAPI.
* **POO (Programación Orientada a Objetos):** Diseñaremos clases para el chatbot y la conversación.
* **Procesamiento de Lenguaje Natural (NLP):** Tokenización, stemming, similitud de textos.
* **JavaScript (ES6+):** Manipulación del DOM, eventos, llamadas asíncronas (async/await) a una API.
* **Arquitectura Cliente-Servidor:** Cómo se comunican el frontend y el backend.
* **Manejo de Git:** Para el control de versiones de tu proyecto.

**Plan de Desarrollo por Pasos**

Comenzaremos con una base mínima viable (MVP) y la iremos mejorando iterativamente.

1. **Paso 0:** Configuración del entorno y estructura del proyecto.
2. **Paso 1:** Creación de la base de conocimiento (JSON) y su loader.
3. **Paso 2:** Implementación del núcleo del chatbot (lógica de matching de preguntas).
4. **Paso 3:** Construcción de la API REST con FastAPI.
5. **Paso 4:** Creación del frontend web simple.
6. **Paso 5:** Implementación de la memoria de conversación (contexto).
7. **Paso 6 (Opcional):** Integración con un modelo de LLM (OpenAI) para respuestas más generativas.

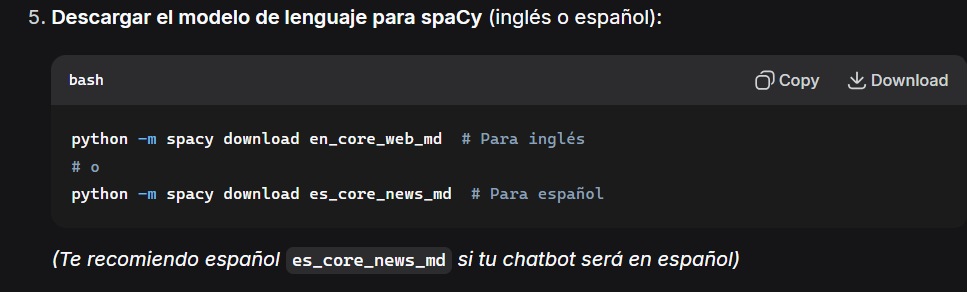
Siempre que completes un paso y lo tengas funcionando, me dices **"Listo para el siguiente paso"** y yo te daré las instrucciones detalladas para el siguiente.



python -m venv venv

1. Actívalo:
   * + **En Windows (PowerShell):** .\venv\Scripts\Activate.ps1
     + **En macOS/Linux:** source venv/bin/activate
   * Verás que el prompt de la terminal cambia a (venv), indicando que el entorno está activo.
2. **Instalar las dependencias iniciales:**

pip install fastapi uvicorn spacy scikit-learn



python -m spacy download en\_core\_web\_md # Para inglés

# o

python -m spacy download es\_core\_news\_md # Para español

**Estructura de archivos iniciales**

bizchat-assistant/

│

├── app/

│ ├── \_\_init\_\_.py

│ ├── main.py # Punto de entrada de FastAPI

│ ├── chatbot.py # Lógica principal del chatbot

│ └── data/

│ └── knowledge\_base.json # Nuestra base de conocimiento

│

├── static/

│ ├── style.css # Estilos CSS

│ └── script.js # Lógica del frontend

│

├── templates/

│ └── index.html # Página principal

│

├── requirements.txt # Dependencias del proyecto

└── venv/ # Tu entorno virtual (NO se sube a Git)

* El **entorno virtual (**venv**)** es crucial en desarrollo Python. Evita conflictos entre versiones de librerías de diferentes proyectos.
* **FastAPI** es el framework que elegimos por su alto rendimiento, facilidad de uso y generación automática de documentación.
* **spaCy** es una librería industrial para NLP. No es la más simple, pero es muy poderosa y te enseñará conceptos reales usados en la industria. El modelo \_md (medium) es un buen balance entre rendimiento y precisión.

PASO 1: La Base de Conocimiento.

Objetivo: Crear el archivo de datos que contendrá toda la información específica de tu negocio o tema, y escribir el código necesario para cargarlo y acceder a él desde Python.

Explicación Conceptual

La base de conocimiento (Knowledge Base) es el "libro de instrucciones" de tu chatbot. Define de lo que puede hablar y qué respuestas debe dar. Para un MVP, la forma más simple es usar un archivo JSON (JavaScript Object Notation), que es un formato ligero y fácil de leer tanto para humanos como para máquinas.

Vamos a estructurarlo como una lista de intenciones. Cada intención representa un tipo de pregunta o acción que el usuario podría quer realizar.

**Tareas a Realizar**

**1. Crear y poblar el archivo**knowledge\_base.json

Navega a la carpeta app/data/ y crea el archivo knowledge\_base.json. Abre el archivo y escribe el siguiente contenido de ejemplo para una tienda online (puedes adaptar los temas y respuestas a tu negocio):

**2. Crear la clase**Chatbot**y el método para cargar la base de conocimiento**

Ahora, ve al archivo app/chatbot.py (que debe estar vacío) y escribe el siguiente código:

**Explicación Docente del Código**

* **Clase**Chatbot**:** Usamos **Programación Orientada a Objetos (POO)**. Una clase es un "molde" que encapsula datos (atributos como intents) y comportamientos (métodos como load\_knowledge\_base). Esto hace el código más organizado, reutilizable y fácil de mantener.
* \_\_init\_\_**:** Este es el *constructor*. Se ejecuta automáticamente cuando creas una nueva instancia de la clase (bot = Chatbot(...)). Su trabajo es inicializar el objeto. Le pasamos la ruta del archivo JSON y inmediatamente llama al método para cargar los datos.
* **Manejo de Errores (**try-except**):** Es crucial manejar posibles errores, como que el archivo no exista o esté corrupto. try-except "intenta" ejecutar un código y "atrapa" (except) cualquier error que pueda ocurrir, permitiéndonos manejarlo elegantemente en lugar de que el programa se cierre de golpe.
* os.path.join**y**\_\_file\_\_**:** Usamos os.path.join para construir rutas de forma que funcionen en Windows, macOS y Linux. \_\_file\_\_ es una variable especial de Python que contiene la ruta del archivo actual (chatbot.py). os.path.dirname(\_\_file\_\_) nos da la carpeta que contiene chatbot.py, y desde ahí navegamos a data/knowledge\_base.json.
* if \_\_name\_\_ == "\_\_main\_\_":**:** Este es un patrón común. El código dentro de este bloque **solo se ejecutará si ejecutas este archivo directamente** (python chatbot.py). Es perfecto para poner código de prueba. Si importas este archivo desde otro (como main.py), este bloque no se ejecutará.

**Prueba del Paso 1**

1. Guarda todos los archivos.
2. En tu terminal, asegúrate de que tu entorno virtual esté activo ((venv) debería aparecer en el prompt).
3. Navega hasta la carpeta app/ (cd app).
4. Ejecuta el comando: python chatbot.py

**Salida esperada:**Base de conocimiento cargada exitosamente. 6 intenciones encontradas.

Tags de intenciones cargadas:

- saludo

- despedida

- horario\_atencion

- metodos\_pago

- agradecimiento

- fallback

**Si ves esta salida, ¡excelente! Has creado con éxito la base de datos de tu chatbot y la has cargado en memoria.**

**PASO 2: El Núcleo de la Inteligencia (Matching de Preguntas).**

**Objetivo: Implementar la lógica que permite al chatbot comprender la pregunta del usuario, encontrar la intención más similar en la base de conocimiento y seleccionar una respuesta.**

**Explicación Conceptual**

**El corazón de nuestro chatbot (por ahora) es un algoritmo de búsqueda de similitud.** La idea es:

1. Preprocesar tanto la pregunta del usuario como los patterns de nuestra base de conocimiento (convertir a minúsculas, eliminar signos de puntuación, etc.).
2. Convertir el texto preprocesado en una representación numérica (vector) que una computadora pueda entender y comparar. Usaremos la librería spaCy para esto.
3. Comparar la representación vectorial de la pregunta del usuario con la de cada pattern en nuestra base de conocimiento.
4. Encontrar la coincidencia más similar.
5. Si la similitud es lo suficientemente alta, elegir una respuesta aleatoria del grupo de respuestas de esa intención.
6. Si no se encuentra una coincidencia buena, usar la intención de fallback.

Actualización chatbot.py

**import json**

**import os**

**import random**

**from typing import List, Dict, Any**

**# Importamos spaCy y cargamos el modelo de lenguaje**

**import spacy**

**class Chatbot:**

**"""Clase principal que representa el chatbot y su lógica."""**

**def \_\_init\_\_(self, knowledge\_base\_path: str):**

**"""**

**Inicializa el chatbot cargando la base de conocimiento y el modelo de NLP.**

**Args:**

**knowledge\_base\_path (str): Ruta al archivo JSON de la base de conocimiento.**

**"""**

**self.knowledge\_base\_path = knowledge\_base\_path**

**self.intents = []**

**# Cargar el modelo de spaCy (asegúrate de haberlo descargado: 'python -m spacy download es\_core\_news\_md')**

**try:**

**self.nlp = spacy.load("es\_core\_news\_md") # Usa 'en\_core\_web\_md' para inglés**

**print("Modelo de lenguaje de spaCy cargado exitosamente.")**

**except OSError:**

**print("Error: El modelo de spaCy 'es\_core\_news\_md' no está instalado.")**

**print("Por favor, ejecuta en tu terminal: 'python -m spacy download es\_core\_news\_md'")**

**exit(1)**

**self.load\_knowledge\_base()**

**# Preprocesamos TODOS los patterns de la base de conocimiento una sola vez al iniciar**

**self.preprocess\_intents()**

**def load\_knowledge\_base(self) -> None:**

**"""Carga la base de conocimiento desde el archivo JSON."""**

**full\_path = os.path.join(os.path.dirname(\_\_file\_\_), self.knowledge\_base\_path)**

**try:**

**with open(full\_path, 'r', encoding='utf-8') as file:**

**data = json.load(file)**

**self.intents = data.get('intents', [])**

**print(f"Base de conocimiento cargada exitosamente. {len(self.intents)} intenciones encontradas.")**

**except FileNotFoundError:**

**print(f"Error: No se pudo encontrar el archivo en {full\_path}.")**

**self.intents = []**

**except json.JSONDecodeError:**

**print("Error: El archivo JSON está mal formado.")**

**self.intents = []**

**def preprocess\_intents(self) -> None:**

**"""**

**Preprocesa todos los 'patterns' de las intenciones.**

**Convierte cada pattern en un vector de spaCy (Doc) y lo guarda en una nueva clave 'patterns\_processed'.**

**Esto optimiza el proceso, ya que solo se hace una vez al inicio.**

**"""**

**for intent in self.intents:**

**intent['patterns\_processed'] = [self.nlp(pattern) for pattern in intent['patterns']]**

**def preprocess\_input(self, user\_input: str):**

**"""**

**Preprocesa la entrada del usuario usando el mismo pipeline de spaCy.**

**Args:**

**user\_input (str): La pregunta o texto escrito por el usuario.**

**Returns:**

**Doc: Un objeto Doc de spaCy que representa el texto procesado.**

**"""**

**return self.nlp(user\_input.lower().strip())**

**def find\_most\_similar\_intent(self, processed\_input) -> Dict[str, Any]:**

**"""**

**Encuentra la intención cuya lista de patrones es más similar a la entrada del usuario.**

**Args:**

**processed\_input (Doc): La entrada del usuario, procesada por spaCy.**

**Returns:**

**Dict[str, Any]: La intención más similar encontrada.**

**"""**

**best\_similarity = 0.0**

**best\_intent = None**

**# Itera sobre cada intención en la base de conocimiento**

**for intent in self.intents:**

**# Itera sobre cada pattern preprocesado de esta intención**

**for pattern\_doc in intent['patterns\_processed']:**

**# spaCy puede calcular la similitud entre dos objetos Doc**

**# similarity() devuelve un valor entre 0 (nada similar) y 1 (idéntico)**

**current\_similarity = processed\_input.similarity(pattern\_doc)**

**# print(f"Comparando '{processed\_input.text}' con '{pattern\_doc.text}': {current\_similarity}") # <-- Descomenta para debug**

**# Si encontramos una similitud mayor, la guardamos**

**if current\_similarity > best\_similarity:**

**best\_similarity = current\_similarity**

**best\_intent = intent**

**# Define un umbral de similitud. Ajusta este valor según tus pruebas (0.5-0.7 es un buen inicio).**

**similarity\_threshold = 0.6**

**print(f"Similitud más alta encontrada: {best\_similarity} (Umbral: {similarity\_threshold})")**

**# Si la similitud no supera el umbral, devolvemos la intención de fallback**

**if best\_similarity < similarity\_threshold:**

**print("La similitud es baja. Usando intención de fallback.")**

**for intent in self.intents:**

**if intent['tag'] == 'fallback':**

**return intent**

**# Si la supera, devolvemos la mejor intención encontrada**

**return best\_intent**

**def get\_response(self, user\_input: str) -> str:**

**"""**

**Función principal para obtener una respuesta del chatbot.**

**Orquesta todo el proceso: preprocesar, buscar intención, devolver respuesta.**

**Args:**

**user\_input (str): La pregunta escrita por el usuario.**

**Returns:**

**str: La respuesta generada por el chatbot.**

**"""**

**if not user\_input:**

**return "Por favor, escribe algo."**

**# 1. Preprocesar la entrada del usuario**

**processed\_input = self.preprocess\_input(user\_input)**

**print(f"Entrada procesada: '{processed\_input.text}'")**

**# 2. Encontrar la intención más similar**

**matched\_intent = self.find\_most\_similar\_intent(processed\_input)**

**# 3. Seleccionar una respuesta aleatoria del grupo de respuestas de esa intención**

**if matched\_intent and 'responses' in matched\_intent:**

**response = random.choice(matched\_intent['responses'])**

**return response**

**else:**

**# Esto no debería pasar si la intent 'fallback' está bien configurada, pero es un buen safeguard.**

**return "Lo siento, no estoy seguro de cómo responder a eso."**

**# Bloque de código para pruebas**

**if \_\_name\_\_ == "\_\_main\_\_":**

**# Instanciamos el chatbot**

**bot = Chatbot('data/knowledge\_base.json')**

**# Bucle simple de prueba en la terminal**

**print("\n--- Modo Prueba del Chatbot ---")**

**print('Escribe "salir" para terminar la prueba.')**

**while True:**

**user\_message = input("Tú: ")**

**if user\_message.lower() == 'salir':**

**break**

**bot\_response = bot.get\_response(user\_message)**

**print(f"Bot: {bot\_response}")**

**Explicación Docente del Código**

* **spaCy Doc Objects:** spaCy no solo divide el texto en palabras (tokens), sino que también calcula características lingüísticas. Un objeto Doc es un contenedor para acceder a estas annotations. Lo más importante para nosotros es que puede calcular la **similitud semántica** entre dos documentos.
* **Similitud por Defecto:** Por defecto, spaCy usa vectores de palabras (word vectors) para calcular similitud. Esto significa que puede entender que "¿A qué hora abren?" y "¿Cuál es su horario?" son preguntas similares, incluso si no comparten las mismas palabras exactas.
* **Preprocesamiento en Fase de Carga (**preprocess\_intents**):** Es una **optimización crucial**. En lugar de procesar cada pattern cada vez que un usuario escribe algo (lo que sería muy lento), procesamos todos los patterns **una sola vez** cuando el chatbot se inicia y guardamos los resultados. Esto hace que la búsqueda de respuestas sea muy rápida.
* **Umbral de Similitud (**similarity\_threshold**):** Este es un **hiperparámetro** que puedes ajustar. Un valor más bajo (ej: 0.5) hará que el chatbot sea más "respondón" pero a veces cometa errores. Un valor más alto (ej: 0.75) lo hará más preciso pero también más propenso a usar el fallback. Deberás ajustarlo basándote en tus pruebas.
* **Elección Aleatoria (**random.choice**):** Elegir una respuesta al azar de una lista de posibles respuestas para una intención hace que el chatbot se sienta menos robótico y repetitivo.

**Prueba del Paso 2**

1. Guarda el archivo.
2. En tu terminal, **asegúrate de estar en la carpeta**app/.
3. Ejecuta: python chatbot.py
4. **¡Interactúa con tu chatbot!** Prueba con diferentes saludos, preguntas sobre el horario y métodos de pago. También prueba con preguntas que no están en la base de conocimiento (ej: "¿Qué tiempo hace?") para ver cómo activa el fallback.
5. **Observa la terminal:** Verás mensajes de debug que te muestran la similitud calculada, lo cual es invaluable para ajustar el umbral.
6. **Posible Error & Solución:**  
   Si ves Error: El modelo de spaCy 'es\_core\_news\_md' no está instalado., **no cierres el programa**. Abre una **NUEVA** terminal, activa tu entorno virtual (venv) y ejecuta el comando que te indica: python -m spacy download es\_core\_news\_md. Luego, vuelve a ejecutar python chatbot.py.

**PASO 3: Creando la API con FastAPI**.

**Objetivo:** Exponer la funcionalidad de nuestro chatbot a través de una API web RESTful. Esto permitirá que un frontend (o cualquier otra aplicación) se comunique con nuestro backend enviando mensajes y recibiendo respuestas.

**Explicación Conceptual**

FastAPI es un framework moderno y ultrarrápido para construir APIs con Python. Su magia radica en:

1. **Autodocumentación:** Genera automáticamente documentación interactiva (Swagger UI y ReDoc) basada en tu código.
2. **Validación de Datos:** Usa Python type hints para validar automáticamente los datos que entran y salen de tu API.
3. **Asincronía:** Soporta operaciones asíncronas por defecto, lo que lo hace ideal para aplicaciones de alto rendimiento como un chatbot.

Vamos a crear un **endpoint** (un punto de acceso específico en nuestra API) que acepte una pregunta del usuario y devuelva la respuesta del chatbot

**Tareas a Realizar**

**1. Crear el archivo principal de FastAPI:**app/main.py

Este archivo será el punto de entrada de nuestra aplicación. Abre o crea el archivo app/main.py y escribe el siguiente código:

**from fastapi import FastAPI**

**from fastapi.middleware.cors import CORSMiddleware**

**from pydantic import BaseModel**

**from app.chatbot import Chatbot # Importamos nuestra clase Chatbot**

**# Define un modelo Pydantic para la estructura de la solicitud (request)**

**# Esto valida automáticamente que los datos entrantes tengan el formato correcto.**

**class ChatRequest(BaseModel):**

**message: str**

**# Define un modelo Pydantic para la estructura de la respuesta (response)**

**class ChatResponse(BaseModel):**

**response: str**

**# Instancia la aplicación FastAPI**

**app = FastAPI(**

**title="BizChat Assistant API",**

**description="Una API para un chatbot conversacional de dominio específico.",**

**version="1.0.0"**

**)**

**# Configura CORS (Crucial para que el frontend se comunique con el backend)**

**# Esto le dice al backend que acepte requests desde el origen (frontend) que especificaremos.**

**origins = [**

**"http://localhost", # Para producción, reemplaza con tu dominio real**

**"http://localhost:8000", # Dirección común para el servidor de desarrollo de FastAPI**

**"http://127.0.0.1:5500" # Dirección común para un servidor live de VSCode**

**]**

**app.add\_middleware(**

**CORSMiddleware,**

**allow\_origins=origins, # Lista de orígenes permitidos**

**allow\_credentials=True,**

**allow\_methods=["\*"], # Permite todos los métodos (GET, POST, etc.)**

**allow\_headers=["\*"], # Permite todos los headers**

**)**

**# Instancia global del chatbot**

**# Se crea al iniciar la aplicación y estará disponible para todos los requests.**

**chatbot = Chatbot('data/knowledge\_base.json')**

**# Define el endpoint principal de la API**

**@app.post("/chat", response\_model=ChatResponse)**

**async def chat\_endpoint(chat\_request: ChatRequest):**

**"""**

**Endpoint para enviar un mensaje al chatbot y recibir su respuesta.**

**- \*\*message\*\*: El mensaje/texto del usuario.**

**"""**

**print(f"[API] Received message: '{chat\_request.message}'")**

**# Usa la instancia del chatbot para obtener una respuesta**

**bot\_response = chatbot.get\_response(chat\_request.message)**

**print(f"[API] Sending response: '{bot\_response}'")**

**# Devuelve la respuesta empaquetada en el modelo de respuesta**

**return ChatResponse(response=bot\_response)**

**# Endpoint de prueba para verificar que la API está funcionando**

**@app.get("/")**

**async def root():**

**return {"message": "¡BizChat Assistant API está en línea!"}**

**# Endpoint para listar las intenciones cargadas (útil para debug)**

**@app.get("/intents")**

**async def list\_intents():**

**intents\_list = [{"tag": intent["tag"]} for intent in chatbot.get\_intents()]**

**return {"intents": intents\_list}**

**# Este bloque permite ejecutar la app con 'python main.py' para desarrollo**

**if \_\_name\_\_ == "\_\_main\_\_":**

**import uvicorn**

**uvicorn.run(app, host="0.0.0.0", port=8000)**

2. Crear el archivo requirements.txt en la raíz del proyecto

Este archivo lista todas las dependencias que necesita nuestro proyecto para funcionar. Es crucial para que otros desarrolladores (o un servidor de producción) puedan instalar exactamente lo que necesitas.

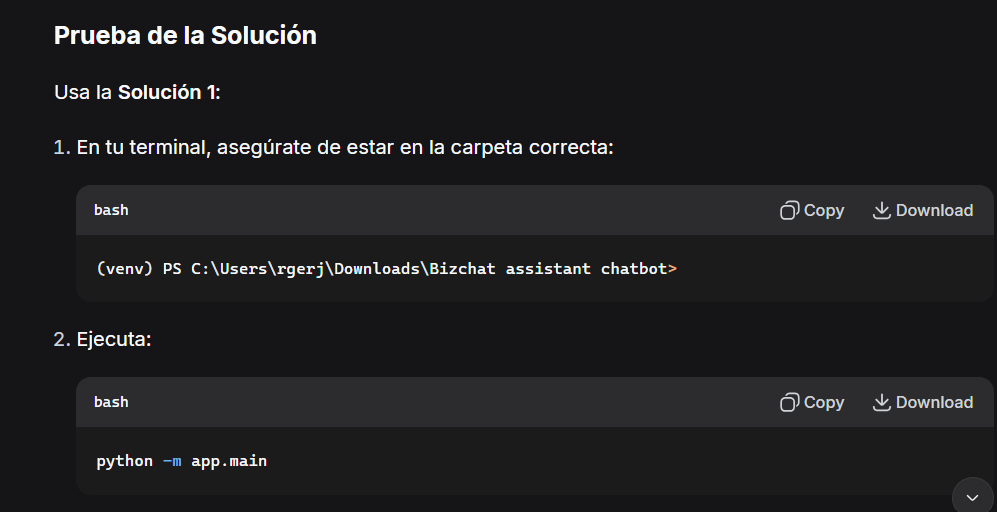
En la carpeta principal bizchat-assistant/ (junto a la carpeta app/), crea el archivo requirements.txt y añade:

**Explicación Docente del Código**

* **Pydantic BaseModel: Estas clases definen la "forma" de los datos. ChatRequest exige que cualquier POST a /chat debe contener un JSON con un campo message que sea un string. FastAPI valida esto automáticamente y devuelve un error si no se cumple. ChatResponse garantiza que siempre devolveremos un JSON con un campo response.**
* **CORS (Cross-Origin Resource Sharing): Es un mecanismo de seguridad del navegador. Nuestro frontend (en http://localhost:5500) y nuestro backend (en http://localhost:8000) son orígenes diferentes. Sin la configuración CORS, el navegador bloquearía las peticiones del frontend al backend. El middleware CORSMiddleware le dice al navegador "está bien, acepta requests desde este origen".**
* **@app.post("/chat") (Decorador): Esto define una ruta o endpoint. Le dice a FastAPI: "Cuando esta aplicación reciba una solicitud POST en la URL /chat, ejecuta la función chat\_endpoint".**
* **async def: Hace que la función sea asíncrona. Esto es excelente para operaciones de I/O (como esperar una respuesta de la base de conocimiento) porque permite que el servidor maneje otras solicitudes mientras espera.**
* **Instancia Global chatbot: Creamos una sola instancia del chatbot cuando se inicia la aplicación. Esto es eficiente porque la base de conocimiento y el modelo de spaCy se cargan solo una vez, en memoria, y se reutilizan para cada request.**

Prueba del Paso 3

1. Guarda todos los archivos.
2. En tu terminal, asegúrate de estar en la carpeta app/.
3. Ejecuta el servidor de desarrollo:

****

**¡Prueba la API! Tienes TRES formas increíbles de hacerlo:**

**A) Documentación Interactiva Automática (Swagger UI):  
Abre tu navegador y ve a**[**http://localhost:8000/docs**](http://localhost:8000/docs)**. Verás una página web generada automáticamente que documenta tu API. Haz clic en POST /chat, luego en "Try it out", escribe un mensaje (ej: "Hola") en el message del Request body y presiona "Execute". Verás la respuesta del servidor.**

**B) Documentación Alternativa (ReDoc):  
Ve a**[**http://localhost:8000/redoc**](http://localhost:8000/redoc)**. Otra forma visualmente diferente de ver la documentación.**

**C) Endpoint de Prueba:  
Ve a**[**http://localhost:8000**](http://localhost:8000/)**. Deberías ver {"message":"¡BizChat Assistant API está en línea!"}.  
Ve a**[**http://localhost:8000/intents**](http://localhost:8000/intents)**. Deberías ver un listado JSON de los "tags" de tus intenciones.**

**Detener el servidor**

Detener un servidor que se está ejecutando en la terminal es una operación fundamental. Hay dos formas principales, dependiendo de cómo lo hayas iniciado.

**Método 1: El Método Universal (Ctrl + C)**

Este es el método más común y funciona casi siempre que un programa se esté ejecutando en tu terminal.

1. Asegúrate de que la terminal donde se está ejecutando tu servidor de FastAPI (uvicorn) esté activa/visible.
2. Simplemente presiona la combinación de teclas:  
   Ctrl + C (Control + la tecla C)

¿Qué hace?  
Ctrl + C es una señal en sistemas Unix (Linux/macOS) y Windows que le dice al programa en primer plano que se interrumpa de manera controlada (SIGINT). FastAPI/UVicorn la captura y apaga el servidor de forma graceful, mostrando通常 un mensaje como:

Ahora que tu API está funcionando correctamente, es hora de crear la interfaz de usuario. Pasamos al**PASO 4: Creando el Frontend Web.**

Objetivo: Desarrollar una interfaz web simple pero funcional que permita a los usuarios interactuar con tu chatbot a través del navegador, utilizando HTML, CSS y JavaScript para comunicarse con la API de FastAPI.

Explicación Conceptual

Vamos a crear una SPA (Single Page Application) básica. La comunicación entre el frontend (navegador) y el backend (FastAPI) se realizará mediante llamadas asíncronas usando la Fetch API de JavaScript.

El flujo será:

1. El usuario escribe un mensaje y presiona "Enviar".
2. JavaScript captura el mensaje, lo envía via POST a nuestro endpoint /chat.
3. Espera la respuesta de la API.
4. Muestra la respuesta del chatbot en la interfaz, manteniendo el historial de la conversación.
5. **Tareas a Realizar**
6. **1. Crear el archivo HTML:**templates/index.html
7. Este será la estructura básica de nuestra aplicación web. Copia el siguiente código:

**<!DOCTYPE html>**

**<html lang="es">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>BizChat Assistant</title>**

**<link rel="stylesheet" href="../static/style.css">**

**</head>**

**<body>**

**<div class="chat-container">**

**<header class="chat-header">**

**<h1>BizChat Assistant</h1>**

**<p>Tu asistente virtual de confianza</p>**

**</header>**

**<div class="chat-messages" id="chat-messages">**

**<!-- Los mensajes se insertarán aquí dinámicamente con JavaScript -->**

**<div class="message bot-message">**

**<div class="message-content">**

**<p>¡Hola! Soy tu asistente virtual. ¿En qué puedo ayudarte hoy?</p>**

**</div>**

**</div>**

**</div>**

**<div class="chat-input-container">**

**<form id="chat-form">**

**<input**

**type="text"**

**id="user-input"**

**placeholder="Escribe tu mensaje aquí..."**

**autocomplete="off"**

**required**

**>**

**<button type="submit">Enviar</button>**

**</form>**

**</div>**

**</div>**

**<script src="../static/script.js"></script>**

**</body>**

**</html>**

2. Crear el archivo de estilos:**static/style.css**

Los estilos harán que nuestra interfaz sea más atractiva y usable. Copia el siguiente código:

**/\* Reset básico y estilos generales \*/**

**\* {**

**margin: 0;**

**padding: 0;**

**box-sizing: border-box;**

**}**

**body {**

**font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;**

**background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);**

**height: 100vh;**

**display: flex;**

**justify-content: center;**

**align-items: center;**

**}**

**.chat-container {**

**width: 90%;**

**max-width: 600px;**

**height: 80vh;**

**background: white;**

**border-radius: 15px;**

**box-shadow: 0 10px 30px rgba(0, 0, 0, 0.2);**

**display: flex;**

**flex-direction: column;**

**overflow: hidden;**

**}**

**.chat-header {**

**background: linear-gradient(135deg, #4facfe 0%, #00f2fe 100%);**

**color: white;**

**padding: 20px;**

**text-align: center;**

**}**

**.chat-header h1 {**

**font-size: 1.8rem;**

**margin-bottom: 5px;**

**}**

**.chat-header p {**

**font-size: 0.9rem;**

**opacity: 0.9;**

**}**

**.chat-messages {**

**flex: 1;**

**padding: 20px;**

**overflow-y: auto;**

**display: flex;**

**flex-direction: column;**

**gap: 15px;**

**}**

**.message {**

**display: flex;**

**max-width: 80%;**

**}**

**.user-message {**

**align-self: flex-end;**

**}**

**.bot-message {**

**align-self: flex-start;**

**}**

**.message-content {**

**padding: 12px 18px;**

**border-radius: 18px;**

**box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);**

**}**

**.user-message .message-content {**

**background: #4facfe;**

**color: white;**

**border-bottom-right-radius: 5px;**

**}**

**.bot-message .message-content {**

**background: #f1f3f6;**

**color: #333;**

**border-bottom-left-radius: 5px;**

**}**

**.message-content p {**

**margin: 0;**

**line-height: 1.4;**

**}**

**.chat-input-container {**

**padding: 20px;**

**border-top: 1px solid #eee;**

**}**

**#chat-form {**

**display: flex;**

**gap: 10px;**

**}**

**#user-input {**

**flex: 1;**

**padding: 12px 15px;**

**border: 2px solid #e1e5eb;**

**border-radius: 25px;**

**outline: none;**

**font-size: 1rem;**

**transition: border-color 0.3s ease;**

**}**

**#user-input:focus {**

**border-color: #4facfe;**

**}**

**button {**

**padding: 12px 25px;**

**background: linear-gradient(135deg, #4facfe 0%, #00f2fe 100%);**

**color: white;**

**border: none;**

**border-radius: 25px;**

**cursor: pointer;**

**font-weight: bold;**

**transition: transform 0.2s ease;**

**}**

**button:hover {**

**transform: translateY(-2px);**

**}**

**button:active {**

**transform: translateY(0);**

**}**

**/\* Scrollbar personalizada \*/**

**.chat-messages::-webkit-scrollbar {**

**width: 6px;**

**}**

**.chat-messages::-webkit-scrollbar-track {**

**background: #f1f1f1;**

**border-radius: 10px;**

**}**

**.chat-messages::-webkit-scrollbar-thumb {**

**background: #c1c1c1;**

**border-radius: 10px;**

**}**

**.chat-messages::-webkit-scrollbar-thumb:hover {**

**background: #a8a8a8;**

**}**

3. Crear la lógica JavaScript: **static/script.js**

Este es el cerebro del frontend que se comunicará con tu API. Copia el siguiente código:

**// URL base de nuestra API - ¡IMPORTANTE! Usa la misma que donde corre FastAPI**

**const API\_BASE\_URL = 'http://localhost:8000';**

**// Elementos del DOM**

**const chatForm = document.getElementById('chat-form');**

**const userInput = document.getElementById('user-input');**

**const chatMessages = document.getElementById('chat-messages');**

**// Función para añadir un mensaje al chat**

**function addMessage(message, isUser = false) {**

**const messageDiv = document.createElement('div');**

**messageDiv.className = `message ${isUser ? 'user-message' : 'bot-message'}`;**

**const messageContent = document.createElement('div');**

**messageContent.className = 'message-content';**

**const messageText = document.createElement('p');**

**messageText.textContent = message;**

**messageContent.appendChild(messageText);**

**messageDiv.appendChild(messageContent);**

**chatMessages.appendChild(messageDiv);**

**// Auto-scroll to the bottom**

**chatMessages.scrollTop = chatMessages.scrollHeight;**

**}**

**// Función para enviar mensaje al chatbot**

**async function sendMessageToChatbot(message) {**

**try {**

**// Mostrar un indicador de "escribiendo..." (puedes mejorarlo luego)**

**// addMessage("Escribiendo...", false);**

**const response = await fetch(`${API\_BASE\_URL}/chat`, {**

**method: 'POST',**

**headers: {**

**'Content-Type': 'application/json',**

**},**

**body: JSON.stringify({ message: message })**

**});**

**if (!response.ok) {**

**throw new Error(`Error HTTP: ${response.status}`);**

**}**

**const data = await response.json();**

**return data.response;**

**} catch (error) {**

**console.error('Error al comunicarse con la API:', error);**

**return 'Lo siento, hubo un error al procesar tu solicitud.';**

**}**

**}**

**// Manejar el envío del formulario**

**chatForm.addEventListener('submit', async function(event) {**

**event.preventDefault(); // Prevenir el comportamiento por defecto del formulario**

**const message = userInput.value.trim();**

**if (!message) return; // No enviar mensajes vacíos**

**// Añadir mensaje del usuario al chat y limpiar el input**

**addMessage(message, true);**

**userInput.value = '';**

**// Obtener y mostrar la respuesta del chatbot**

**const botResponse = await sendMessageToChatbot(message);**

**addMessage(botResponse, false);**

**});**

**// Focus en el input al cargar la página**

**userInput.focus();**

**// Opcional: Permitir enviar con Enter (ya lo hace el formulario, pero por si acaso)**

**userInput.addEventListener('keypress', function(event) {**

**if (event.key === 'Enter' && !event.shiftKey) {**

**event.preventDefault();**

**chatForm.dispatchEvent(new Event('submit'));**

**}**

**});**

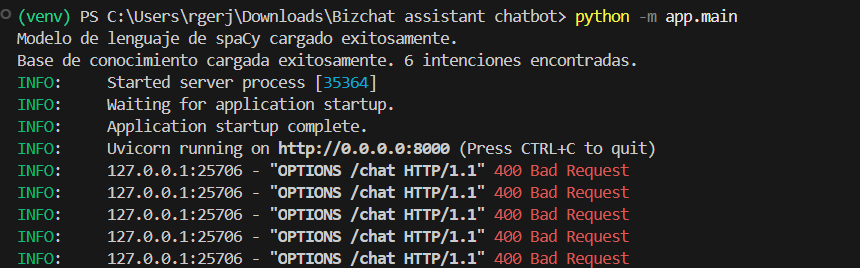
**Explicación Docente del Código**

* **Fetch API: Usamos fetch() para hacer peticiones HTTP desde JavaScript. Es moderno y reemplaza a XMLHttpRequest. La palabra clave await pausa la ejecución hasta que la promesa se resuelva, haciendo el código más legible.**
* **Async/Await: Las funciones async devuelven una Promesa. await solo puede usarse dentro de funciones async. Esto es crucial para operaciones que toman tiempo (como las llamadas de red).**
* **Manipulación del DOM: Usamos document.getElementById(), document.createElement(), y appendChild() para dinámicamente crear y añadir elementos HTML según la conversación avanza.**
* **Prevención del Comportamiento por Defecto: event.preventDefault() en el manejador del formulario evita que la página se recargue cuando se presiona "Enviar".**
* **CSS Flexbox: Los estilos usan Flexbox para crear un diseño responsive y bien organizado.**

**Prueba del Paso 4**

1. **Asegúrate de que tu servidor FastAPI esté corriendo (con python -m app.main desde la raíz del proyecto).**
2. **Sirve el archivo HTML: Tienes varias opciones:**
   * **Opción 1 (Recomendada): Usa la extensión "Live Server" de VS Code. Haz clic derecho en templates/index.html y selecciona "Open with Live Server".**
   * **Opción 2: Abre templates/index.html directamente en el navegador (puede haber problemas con CORS con esta opción).**
   * **Opción 3: Sirve la carpeta con Python. En otra terminal, navega a la carpeta templates/ y ejecuta: python -m http.server 5500**
3. **Abre tu navegador en la URL donde se está sirviendo el frontend (ej: http://127.0.0.1:5500/templates/index.html para Live Server, o http://localhost:5500 para http.server).**
4. **¡Testea tu chatbot completo! Escribe mensajes y deberías ver las respuestas de tu API aparecer en la interfaz.**

**Posible Error & Solución:  
Si ves errores de CORS en la consola del navegador (F12 > Console), verifica que en main.py la URL de tu frontend (ej: http://127.0.0.1:5500) esté incluida en la lista origins.**

**Error encontrado:  
**

**¡Excelente catch! Este error es muy común y es una lección importante sobre CORS (Cross-Origin Resource Sharing). El problema es que tu frontend y backend están en puertos diferentes, y el navegador está bloqueando la comunicación.**

**Explicación del Error**

El error OPTIONS /chat HTTP/1.1 400 Bad Request indica que el navegador está haciendo una petición Preflight antes de enviar el POST real. Esta petición OPTIONS verifica si el servidor permite las solicitudes desde el origen del frontend.

El error 400 (Bad Request) sugiere que FastAPI no está manejando correctamente esta petición OPTIONS porque falta la configuración CORS adecuada.

Solución: Configuración Correcta de CORS en FastAPI

Necesitamos modificar el archivo app/main.py para manejar correctamente las peticiones OPTIONS. Reemplaza la configuración CORS actual con este código mejorado:

**# Configura CORS (Crucial para que el frontend se comunique con el backend)**

**# Esto le dice al backend que acepte requests desde el origen (frontend) que especificaremos.**

**# REEMPLAZA ESTAS LÍNEAS EN TU main.py**

**from fastapi.middleware.cors import CORSMiddleware**

**# ... (el resto de tus imports)**

**# Configura CORS - ¡ACTUALIZA ESTO!**

**origins = [**

**"http://localhost:5500", # Puerto común para Live Server de VSCode**

**"http://127.0.0.1:5500", # Alternativa para Live Server**

**"http://localhost:8000", # Para probar directamente la API**

**"http://127.0.0.1:8000", # Alternativa para la API**

**]**

**app.add\_middleware(**

**CORSMiddleware,**

**allow\_origins=origins,**

**allow\_credentials=True,**

**allow\_methods=["\*"], # Permite todos los métodos: GET, POST, OPTIONS, etc.**

**allow\_headers=["\*"], # Permite todos los headers**

**expose\_headers=["\*"] # Expone todos los headers para el frontend**

**)**

**# ... (el resto de tu código)**

Paso Adicional Crucial: Actualiza la URL en script.js

También necesitamos asegurarnos de que el frontend esté apuntando al puerto correcto. En tu archivo static/script.js, verifica o cambia la URL base:

**// Asegúrate de que esta URL sea correcta**

**// Debe ser el mismo puerto donde corre FastAPI (8000)**

**const API\_BASE\_URL = 'http://localhost:8000';**

**// También puedes usar:**

**// const API\_BASE\_URL = 'http://127.0.0.1:8000';**

¡Tienes toda la razón! El problema es que el puerto del frontend está cambiando dinámicamente (59373 en este caso), y nuestra configuración CORS solo permite puertos específicos que listamos manualmente (5500, 8000).

Solución Definitiva: Permitir Todos los Puertos de Localhost

Necesitamos modificar la configuración CORS en app/main.py para permitir cualquier puerto de localhost. Reemplaza la lista origins por esta configuración más flexible:

**# Configura CORS - SOLUCIÓN DEFINITIVA para desarrollo local**

**# Permite cualquier puerto de localhost (127.0.0.1) y localhost**

**origins = [**

**"http://localhost", # Permite cualquier puerto de localhost**

**"http://localhost:\*", # Patrón comodín para cualquier puerto (algunos middleware lo soportan)**

**"http://127.0.0.1", # Permite cualquier puerto de 127.0.0.1**

**"http://127.0.0.1:\*", # Patrón comodín para cualquier puerto**

**]**

**# Si lo anterior no funciona, usa esta alternativa más agresiva pero efectiva:**

**# origins = ["\*"] # ¡CUIDADO! Esto permite todos los orígenes, solo para desarrollo local**

Pero hay un detalle: Algunas implementaciones de CORS no soportan el patrón :\*. La solución más robusta es usar una función personalizada. Reemplaza toda la configuración CORS con este código:

**# CONFIGURACIÓN CORS MEJORADA - Copia y pega esto**

**from fastapi.middleware.cors import CORSMiddleware**

**app.add\_middleware(**

**CORSMiddleware,**

**allow\_origin\_regex=r"http://(localhost|127\.0\.0\.1)(:\d+)?", # Regex para localhost en cualquier puerto**

**allow\_credentials=True,**

**allow\_methods=["\*"],**

**allow\_headers=["\*"],**

**expose\_headers=["\*"]**

**)**

1. Detén el servidor con Ctrl + C
2. Aplica los cambios en app/main.py y static/script.js
3. Reinicia el servidor:

**python -m app.main**

**el conflicto de CORS. Ese es uno de los problemas más comunes y frustrantes en el desarrollo web, así que es un gran logro haberlo superado.**

**PASO 5: Implementando Memoria de Conversación (Contexto).**

Objetivo: Hacer que el chatbot recuerde el contexto de la conversación reciente, para que pueda mantener diálogos más naturales y coherentes.

Explicación Conceptual

Hasta ahora, cada mensaje se procesa de forma aislada. Para darle "memoria", necesitamos:

1. Almacenar el historial de la conversación.
2. Incluir parte de este historial cuando procesemos una nueva pregunta.
3. Limitar la cantidad de historia que recordamos para no sobrecargar el sistema.

Vamos a implementar una memoria de ventana deslizante que recuerde las últimas N interacciones.