

OpenAI Bots

Repositorio github [Repositorio GitHub - NegotiatorChatbotOpenAI](#)

Este documento se centra en brindar una visión exhaustiva sobre cómo aprovechar al máximo la API de OpenAI utilizando Python, una de las herramientas más potentes y versátiles en el ámbito del desarrollo y la inteligencia artificial. A lo largo de este documento, exploraremos detalladamente cómo configurar y utilizar las APIs de Assistants y Chat Completion de OpenAI para crear asistentes inteligentes capaces de responder preguntas, realizar tareas específicas mediante llamadas a funciones, y recuperar información de archivos mediante retrieval.

En los dos últimos apartados del índice, "Explicación detallada del código (backend)" y "Explicación detallada del código (frontend)", se proporcionan detalles meticulosos sobre el código implementado en el repositorio adjuntado. Estas secciones están meticulosamente construidas para ofrecer una comprensión profunda de cómo el código funciona, su estructura, y cómo se integra con la API de OpenAI para lograr los objetivos establecidos. Estas explicaciones detalladas son esenciales tanto para quienes están dando sus primeros pasos en la programación con OpenAI como para quienes buscan profundizar su comprensión y mejorar sus aplicaciones de inteligencia artificial.

Índice

- [Introducción](#)
 - [Retrieval \(no está activo para la API Chat Completions\)](#)
 - [API ASSISTANTS](#)
 - [Pasos para crear un asistente \(API Assistants\)](#)
 - [FUNCTION CALLS con la API de Assistants](#)
 - [Tips a tener en cuenta para los function calls](#)
 - [RETRIEVALS con la API de Assistants](#)
 - [Tips a tener en cuenta para retrievals:](#)
 - [Combinar function calls y recuperación de archivos](#)
 - [API DE CHAT COMPLETION](#)
 - [Function calls con la API de chat completion](#)
 - [Explicación detallada del código \(backend\)](#)
 - [Explicación detallada del código \(frontend\)](#)
-

Introducción

Para desarrollar un bot utilizando la inteligencia artificial de OpenAI, se tienen fundamentalmente dos enfoques: mediante el uso de la API de Assistants o a través de la API Chat Completion. La principal distinción entre ambas radica en sus capacidades y aplicaciones específicas.

A través de la API de Assistants, se otorga la posibilidad de integrar documentos en formato PDF como fuente de información, lo cual es particularmente útil cuando la pregunta del usuario se relaciona con contenidos

específicos dentro de estos documentos. Esta característica se habilita especificando en el system_message el momento adecuado para consultar el PDF. Además, esta API soporta la creación de un hilo conversacional donde se almacenan tanto los mensajes del usuario como las respuestas del asistente, permitiendo así una interacción contextual y continua.

Por otro lado, la API Chat Completion está orientada hacia una interacción más directa de pregunta y respuesta. Esta no permite la incorporación directa de documentos PDF para consulta, aunque es capaz de mantener un hilo de conversación manteniendo un registro de las preguntas del usuario y las respuestas del bot. Para esto, se requiere acumular los intercambios previos en cada nueva solicitud, situando el último mensaje del usuario como la pregunta actual a responder.

```
user_prompt = "When's the next flight from Amsterdam to New York?"  
  
ChatCompletionMessage(  
    content=None,  
    role='assistant',  
    function_call=FunctionCall(  
        arguments='{\n            "loc_origin": "AMS",\n            "loc_destination": "JFK"\n        }',  
        name='get_flight_info'  
    ),  
    tool_calls=None  
)
```

Como vemos en la salida, el bot no inicia ejecutando funciones de manera directa basándose en las solicitudes específicas. Primero, identifica la función más adecuada para la tarea basándose en el mensaje escrito por el usuario. A continuación, recopila los parámetros necesarios del contexto proporcionado por el usuario. Como desarrolladores, nuestra labor es ejecutar dicha función con los parámetros dados y preparar la respuesta para devolverla al bot. Luego, formateamos el resultado en un JSON estructurado que incluye el rol "function", el nombre de la función ejecutada y el contenido con el resultado. Por ejemplo:

```
{  
    "role": "function",  
    "name": "get_flight_info",  
    "content": "{\"loc_origin\": \"AMS\", \"loc_destination\": \"JFK\", \"datetime\": \"2024-04-10 11:59:06.823330\", \"airline\": \"KLM\", \"flight\": \"KL643\"}"  
}
```

Este paso asegura que el bot pueda interpretar correctamente la respuesta de la función y obtener la información necesaria para comunicarse de manera efectiva con el usuario.

Retrieval (no está activo para la API Chat Completions)

El retrieval aumenta el Asistente con conocimientos externos a su modelo, como información de productos patentados o documentos proporcionados por sus usuarios. Una vez que se carga un archivo y se pasa al Asistente, OpenAI fragmentará automáticamente sus documentos, indexará y almacenará las incrustaciones e

implementará la búsqueda vectorial para recuperar contenido relevante para responder las consultas de los usuarios.

Habilitar el retrieval

Pase el parámetro `retrieval` en la sección `tools` del Asistente para habilitar la recuperación:

```
assistant = client.beta.assistants.create(  
    instructions="You are a customer support chatbot. Use your knowledge base to  
best respond to customer queries.",  
    model="gpt-4-turbo-preview",  
    tools=[{"type": "retrieval"}]  
)
```

Subir archivos para recuperarlos

De manera similar a Code Interpreter, los archivos se pueden subir a nivel de Asistente o a nivel de Mensaje individual.

Ejemplo en Python:

```
# Subir un archivo con un propósito de "assistants"  
file = client.files.create(  
    file=open("knowledge.pdf", "rb"),  
    purpose='assistants'  
)  
  
# Agregar el archivo al asistente  
assistant = client.beta.assistants.create(  
    instructions="You are a customer support chatbot. Use your knowledge base to  
best respond to customer queries.",  
    model="gpt-4-turbo-preview",  
    tools=[{"type": "retrieval"}],  
    file_ids=[file.id]  
)
```

API ASSISTANTS

PASOS PARA CREAR UN ASISTENTE (API Assistants)

1. Importar la librería de OpenAI y configurar la clave de la API.

```
from openai import OpenAI  
import os  
from dotenv import load_dotenv
```

```
# Cargar las variables de entorno desde el archivo .env
load_dotenv()

# Configurar la clave de la API de OpenAI
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
```

2. Crear un asistente básico.

```
assistant_name = "Assistant"

assistant = client.beta.assistants.create(
    name=assistant_name,
    instructions=system_message,
    model=model_name
)
```

model_name: Debe ser uno de los modelos de GPT de OpenAI, por ejemplo, "gpt-4-0125-preview".

system_message: Describe el comportamiento del asistente, por ejemplo, "Eres un bot llamado Pedro que responde siempre alegre".

3. Crear el hilo de mensajes donde se iniciará la conversación.

```
thread = self.client.beta.threads.create()
```

4. Crear un mensaje y añadirlo al hilo.

```
client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content=message[ "content" ]
)
```

message["content"]: Contenido del mensaje, por ejemplo, "Hola, ¿qué tal?". thread_id: ID del hilo, que se obtiene con .id.

5. Ejecutar el asistente.

```
run = client.beta.threads.runs.create(
    thread_id=thread.id,
    assistant_id=assistant.id,
    instructions=system_message
)
```

6. Recuperamos el estado de la ejecución para ver cuándo ha terminado, en caso de que no haya terminado, en un tiempo especificado lo volvemos a comprobar hasta que el estado sea 'completed'.

```

while True:
    # Obtener el estado de la ejecución
    run_status = client.beta.threads.runs.retrieve(
        thread_id=thread.id,
        run_id=run.id
    )

    if run_status.status == 'completed':
        # Obtener los mensajes
        messages = client.beta.threads.messages.list(
            thread_id=thread.id
        )

        # Imprimir los mensajes
        for msg in messages.data:
            content = msg.content[0].text.value
            print(content)
            break
    else:
        time.sleep(3)

```

Con estos 6 pasos, hemos creado un asistente similar al chatbot de OpenAI.

FUNCTION CALLS con la API de Assistants

1. Para emplear FUNCTION CALLS con la API de Assistants de OpenAI, describimos las funciones disponibles en un JSON descriptivo:

```

function_descriptions = [
    {"type": "function", "function": {
        "name": "get_flight_info",
        "description": "Get flight information between two locations",
        "parameters": {
            "type": "object",
            "properties": {
                "loc_origin": {
                    "type": "string",
                    "description": "The departure airport, e.g. DUS",
                },
                "loc_destination": {
                    "type": "string",
                    "description": "The destination airport, e.g. HAM",
                },
            },
            "required": ["loc_origin", "loc_destination"]
        }
    }
]

```

- `name`: es el nombre de la función.
- `description`: es la descripción de la función.
- `parameters`: es donde se describen los argumentos que se le pueden pasar a la función para ejecutarse.
- `required`: sirve para indicar cuáles de los parámetros son obligatorios de obtener, ya que la función puede tener parámetros por defecto en caso de que no se especifiquen.

En caso de querer añadir otra función, se añade una coma al final del último corchete antes del cierre del paréntesis de la lista y se repite el mismo formato desde el principio para la otra función.

2. Definimos las funciones (en este caso se ha creado un ejemplo de respuesta de consulta de una api)

```
def get_flight_info(loc_origin, loc_destination):  
    """Get flight information between two locations."""  
    # Example output returned from an API or database  
    flight_info = {  
        "loc_origin": loc_origin,  
        "loc_destination": loc_destination,  
        "datetime": str(datetime.now() + timedelta(hours=2)),  
        "airline": "KLM",  
        "flight": "KL643",  
    }  
    return json.dumps(flight_info)
```

3. Luego, al crear el asistente, especificamos las herramientas que utilizará, incluyendo las funciones:

```
assistant = client.beta.assistants.create(  
    name=assistant_name,  
    instructions=system_message,  
    model=model_name,  
    tools=tool_list  
)
```

4. Crea un json con las funciones disponibles a las cuales el chatbot podrá llamar.

```
functions_available = {  
    "get_fliugh_info" : get_fliugh_info,  
}
```

5. El asistente detectará automáticamente cuándo debe llamar a una función. En esos casos, serás tú quien deberá llamar a la función correspondiente y devolverle al bot el resultado obtenido de las funciones detectadas. El resto de los pasos para crear un asistente no se modificarán.

```
elif run_status.status == 'requires_action':
    required_actions = run_status.required_action.submit_tool_outputs.model_dump()
    tool_outputs = []
    for action in required_actions["tool_calls"]:
        func_name = action['function']['name']
        arguments = json.loads(action['function']['arguments'])
        func_to_call = functions_available.get(func_name)
        if func_to_call:
            output = func_to_call(**arguments)
            tool_outputs.append({
                "tool_call_id": action['id'],
                "output": output
            })
        else:
            raise ValueError(f"Unknown function: {func_name}")

client.beta.threads.runs.submit_tool_outputs(
    thread_id=thread.id,
    run_id=run.id,
    tool_outputs=tool_outputs
)
```

Con estos pasos, podrás crear un asistente capaz de realizar llamadas a funciones utilizando la API de Assistants de OpenAI, ampliando así sus capacidades según tus necesidades específicas.

Tips a tener en cuenta para los function calls

"TOOL_LIST IMPORTANTE":

La descripción de la función en TOOL_LIST es crucial. Dedica tiempo suficiente para definir con precisión los parámetros de la función y cuándo y cómo se debe llamar a la función. Esto garantizará una comprensión clara y un uso efectivo de la herramienta por parte del bot.

"System_message IMPORTANTE":

La descripción del mensaje del sistema es igual de importante que la tool_list. Dedica tiempo en describir todas las acciones que el bot tiene disponible, y cuando debe usarla, esto ayudará a que no llame a funciones por error.

El sistema_message puede tener este formato:

```
system_message = """
    Hola, soy Pedro, especialista en ofrecer descuentos por pagos inmediatos y en
    negociar planes de pagos personalizados, utilizando el euro como moneda. Estoy
    aquí para asistirte con una variedad de servicios enfocados en la gestión de
    deudas.
```

Funcionalidades Disponibles:

- `set_debt_id`: Establece el ID de la deuda actual para la negociación, asegurando que todas las operaciones subsiguientes se realicen con respecto a la deuda correcta.
 - `get_all.debts`: Muestra todas las deudas asociadas al usuario, facilitando la selección para negociar.
 - `calculate_payment_plan`: Calcula un plan de pago personalizado basado en propuestas específicas del usuario.
 - `manage_negotiation`: Maneja el proceso de negociación de deudas, permitiendo al usuario solicitar una oferta inmediata de pago, responder con una contraoferta, o recibir una propuesta inicial.
 - `propose_payment_plan`: Formula un plan de pago adaptado sin necesidad de entrada adicional del usuario.
 - `propose_partial_immediate_payment`: Calcula un plan de pago ajustado para el saldo restante tras un pago parcial inmediato.
- """

Optimización de Funciones Complejas en la Gestión de Negociaciones:

En lugar de crear múltiples funciones pequeñas, priorizamos una función más extensa con argumentosopcionales. Esto simplifica la lógica del bot y evita confusiones al gestionar diferentes respuestas. Por ejemplo, en `manage_negotiation()`, utilizamos argumentos como `counteroffer` y `request_immediate_payment_offer` para adaptar el comportamiento del bot según la situación específica. Este enfoque ofrece flexibilidad y claridad en el código, facilitando su comprensión y mantenimiento. Ejemplo de función:

```
def manage_negotiation(self, counteroffer=None,
request_immediate_payment_offer=False):
    if self.debt_id is None:
        return json.dumps({"error": "Por favor, introduce un ID de deuda válido."})

    if self.current_price is None:
        debt = self.crud_service.get_debt_by_id(self.debt_id)
        if debt is None:
            return json.dumps({"error": "No tiene ninguna deuda con ese ID."})
        else:
            self.current_price = self.initial_price = debt.total_debt

    if request_immediate_payment_offer:
        self._aumentar_oferta()
        offer = self._get_discounted_price()
        return json.dumps({"message": f"¿Qué te parece si saldas la deuda hoy mismo y se te aplica un {self.current_discount} con un monto total de {offer}€? ¿Aceptas?"})

    if counteroffer is not None:
        if type(counteroffer) not in [int, float]:
            return json.dumps({"error": "Por favor, introduce un número válido."})
```

```
        self._aumentar_oferta()
        response = self._increase_attempt_or_maxed_out()
        if response:
            return response

        if counteroffer < self._get_min_price():
            return self._rechazar_oferta()
        elif counteroffer >= self.current_price:
            return self._aceptar_oferta()
        else:
            price = self._get_discounted_price()
            return json.dumps({"message": f"Mi oferta es de un descuento del {self.current_discount}%, se te quedaría en {price}€. ¿Aceptas?"})

    offer = self._get_discounted_price()
    return json.dumps({"message": f"¿Qué te parece si saldas la deuda hoy mismo y se te aplica un {self.current_discount} con un monto total de {offer}€? ¿Aceptas?"})
```

RETRIEVALS con la API de Assistants

Para que nuestro asistente de OpenAI tenga acceso a información contenida en archivos, primero necesitamos cargar estos archivos en el sistema. Esto se hace mediante el comando `client.files.create()`, que nos permite subir los archivos deseados y generar un identificador único (ID) para cada uno. Este ID es esencial porque es la referencia que utilizará nuestro asistente para acceder a la información contenida en los archivos.

Una vez tenemos los archivos cargados y sus respectivos IDs, el siguiente paso es asegurarnos de que nuestro asistente sepa que puede acceder a ellos. Para hacerlo, al momento de crear nuestro asistente, utilizaremos el parámetro `file_ids`. En este parámetro incluiremos los IDs de todos los archivos a los que el asistente debe tener acceso.

Finalmente, para que el asistente sea consciente de que puede hacer búsquedas dentro de estos archivos, debemos habilitar una herramienta específica conocida como "herramienta de recuperación de información". La forma de hacerlo es muy sencilla: al definir la lista de herramientas que el asistente puede usar (`tools`), simplemente añadimos la línea `{"type": "retrieval"}`. Con esto, el asistente estará listo para consultar los archivos siempre que necesite ampliar su conocimiento o responder preguntas más complejas.

Ejemplo de código:

```
assistant_name = "Assistant"

file_arenas = client.files.create(file=open("content/festivos-apertura-las-arenas-2022.pdf", "rb"), purpose="assistants")
file_comunidades = client.files.create(file=open("content/festivos-por-provincias-2023.pdf", "rb"), purpose="assistants")
```

```
file_linea105 =
client.files.create(file=open("content/Linea_105_Las_Palmas_Galdar_Horarios.pdf",
"rb"), purpose="assistants")

tools = {"type" : "retrieval"}

assistant = client.beta.assistants.create(
    name=assistant_name,
    instructions=system_message,
    model=model_name,
    tools=tools,
    file_ids=[file_arenas.id, file_comunidades.id, file_linea105.id]
)
```

Esto permitirá que nuestro asistente acceda a la información contenida en los archivos cargados y pueda utilizarla para responder preguntas o realizar tareas específicas.

Tips a tener en cuenta para retrievals

"System_message IMPORTANTE":

Para asegurarnos de que nuestro bot maneje las solicitudes de archivos de manera eficiente, es crucial prestar especial atención al mensaje del sistema que le enviaremos. Considera este mensaje como un resumen informativo que orienta al bot sobre el contenido de cada archivo, permitiéndole entender qué información puede encontrar en su interior para que no abra archivos cuando no sea necesario.

Adjuntar file_id al system_message:

Además, una estrategia efectiva para mejorar la precisión con la que nuestro bot identifica y accede a archivos específicos es incluir el identificador único, o ID, del archivo en el mensaje del sistema. De esta manera, el bot no solo recibe indicaciones sobre la naturaleza del archivo sino también una referencia directa a cuál archivo abrir. Es como darle una invitación personalizada a cada archivo, asegurándonos de que el bot sepa exactamente hacia dónde dirigirse para obtener la información requerida.

Ejemplo de ambos tips juntos:

```
system_message = """
Tengo acceso a información específica y valiosa que puede ser de interés:

Festivos de Apertura Comercial del CC Las Arenas (id del documento
{file_arenas_id}): Tengo información detallada sobre los horarios y los días
festivos en los que el Centro Comercial Las Arenas estará abierto. Puedes
consultarme si tienes dudas sobre la apertura en fechas particulares, como por
ejemplo, "¿El 25 de diciembre está abierto el CC Las Arenas?". Es importante
mencionar que si un día festivo no aparece en nuestra lista y coincide con un
festivo oficial en Canarias o es un día festivo nacional en España, el centro
comercial permanecerá cerrado.
```

Festivos de Apertura Comercial por Comunidades Autónomas (id del documento

{file_comunidades_id}): Dispongo de una base de datos actualizada con los días festivos de apertura comercial permitidos por comunidades autónomas en España, conforme a lo establecido por el BOE (Boletín Oficial del Estado). Esta información te permitirá saber en qué días festivos específicos los centros comerciales están autorizados a abrir en cada comunidad autónoma. Por ejemplo, puedes preguntar: "¿Los centros comerciales en Aragón abren el 25 de diciembre?"

Línea de Guagua 105 Las Palmas de Gran Canaria - Gáldar (id del documento {file_linea105_id}): Poseo los horarios completos de la línea de guaguas 105, tanto para días laborales como festivos, cubriendo las rutas de Las Palmas de Gran Canaria a Gáldar y viceversa. Esta información es crucial si necesitas planificar tus viajes y quieres saber, por ejemplo, "¿A qué horas puedo coger la guagua de Gáldar a Las Palmas un domingo?". Ten en cuenta que esta información se limita exclusivamente a la línea 105 entre Las Palmas y Gáldar.

"""

Combinar function calls y recuperación de archivos

Simplemente debemos combinar ambos aspectos aprendidos anteriormente:

```
<div style="text-align:center;">
    
</div>
```

Función para crear un asistente

La siguiente función es utilizada para crear un asistente en OpenAI. Esta función carga archivos necesarios y habilita la herramienta de recuperación de información para que el asistente pueda acceder a ellos.

```
```python
def create_assistant():
 """Función privada para crear un asistente."""
 assistant_name = "Assistant"

 # Cargar archivos necesarios
 file_arenas = self.client.files.create(file=open("content/festivos-apertura-las-arenas-2022.pdf", "rb"), purpose="assistants")
 file_comunidades = self.client.files.create(file=open("content/festivos-por-provincias-2023.pdf", "rb"), purpose="assistants")
 file_linea105 =
 self.client.files.create(file=open("content/Linea_105_Las_Palmas_Galdar_Horarios.pdf", "rb"), purpose="assistants")

 # Añadir la herramienta de recuperación de información a la lista de
 # herramientas
 tools = tools_list
 tools.append({"type": "retrieval"})
```

```

Crear el asistente
assistant = client.beta.assistants.create(
 name=assistant_name,
 instructions=system_message,
 model=model_name,
 tools=tools,
 file_ids=[file_arenas.id, file_comunidades.id, file_linea105.id]
)

```

Como vemos, en la `tool_list` donde se especifican las funciones, simplemente le añadimos la línea: `"tools.append({"type" : "retrieval"})"`, donde indicamos que, además de tener disponibles las llamadas a las funciones, nuestro asistente también será capaz de recuperar información de diferentes archivos.

Si queremos mezclar conceptos para un retrieval más potente, podemos implementar lo siguiente:

## Function calls que indiquen cuándo abrir un archivo

Cuando la creación de un mensaje del sistema, por más detallado que sea, no logra dirigir al bot hacia el archivo correcto justo cuando lo necesita, existe una solución alternativa: utilizar una llamada de función específica que indique al bot el ID del archivo a consultar. Para implementar esta solución de manera efectiva, resulta necesario combinar un `system_message` detallado con las llamadas a funciones.

La idea central es que, si tras analizar el contexto proporcionado el bot no logra identificar el archivo pertinente, una llamada de función activada bajo estas circunstancias puede retornar el ID del archivo en que se aloja la información relevante. De esta manera, el bot dispondrá de una referencia explícita y podrá acceder directamente al contenido necesario para responder a la consulta del usuario.

Se puede usar con diferentes enfoques, un ejemplo podría ser que tengamos un mismo documento con varias fechas, entonces recogeremos la fecha por el contexto en la pregunta del usuario y dependiendo de la fecha indicaremos al bot consultar en un archivo u otro.

### Ejemplo:

```

tools = {
 "type": "function",
 "function": {
 "name": "identify_arenas_holidays_query",
 "description": "Esta función se activa automáticamente cuando se detecta una pregunta relacionada con los festivos de apertura comercial del Centro Comercial Las Arenas de un año específico. La función está diseñada para identificar consultas relacionadas directamente con los festivos en el 'CC Las Arenas' y, una vez identificada como tal, le indica al bot que proceda a buscar la información relevante en el documento PDF proporcionado para 'CC Las Arenas', utilizando el año especificado como parámetro para encontrar la información correcta. En caso de que no se especifique año, se consultarán los festivos de apertura comercial del 'CC Las Arenas' del año 2024",
 "parameters": {
 "type": "object",
 "properties": {

```

```

 "year": {
 "type": "integer",
 "description": "El año para el cual se solicita la información sobre los festivos del Centro Comercial Las Arenas. Esto permite al bot buscar y proporcionar datos precisos correspondientes a ese año específico."
 }
 },
 {"type": "retrieval"
}

```

```

def identify_arenas_holidays_query(self, year=2024):
 if year == 2023:
 return json.dumps({"message": f"Los días festivos de apertura de Las Arenas en 2023 se encuentran en el archivo con id: {self.file_arenas_id_2022}. Si no aparece en la lista y el día solicitado es un festivo, el centro comercial estará cerrado."})

 if year == 2022:
 return json.dumps({"message": f"Los días festivos de apertura de Las Arenas en 2022 se encuentran en el archivo con id: {self.file_arenas_id_2023}. Si no aparece en la lista y el día solicitado es un festivo, el centro comercial estará cerrado."})

 return json.dumps({"message": f"Los días festivos de apertura de Las Arenas en 2024 se encuentran en el archivo con id: {self.file_arenas_id_2024}. Si no aparece en la lista y el día solicitado es un festivo, el centro comercial estará cerrado."})

```

Este código define la función `identify_arenas_holidays_query`, que recibe un año como argumento y devuelve un mensaje JSON indicando en qué archivo se pueden encontrar los días festivos de apertura del Centro Comercial Las Arenas para el año especificado. Si no se especifica un año, por defecto se busca en el archivo correspondiente al año 2024. Este enfoque combina las funcionalidades de llamadas a funciones y recuperación de información para mejorar la precisión y la eficacia del asistente.

```

chatbot.py
from openai import OpenAI
import time
import json
import dotenv
import os

dotenv.load_dotenv()
ASSISTANT_ID = os.getenv('ASSISTANT_ID')

class Chatbot:
 def __init__(self, functions_call):
 self.functions_call = functions_call

```

```
self.tools_list = self.functions_call.get_tools_list()
self.system_message = functions_call.get_system_message()
self.functions_available = functions_call.get_functions_available()
self.client = OpenAI()
self.run = None
self.thread_id = None
self.model_name = "gpt-4-0125-preview"

global ASSISTANT_ID
if ASSISTANT_ID is None:
 self._create_assistant()

if self.thread_id is None:
 self.create_thread()

def _create_assistant(self):
 """Función privada para crear un asistente."""
 assistant_name = "Negociator Assistant"

 file_arenas = self.client.files.create(file=open("content/festivos-apertura-las-arenas-2022.pdf", "rb"), purpose="assistants")
 file_comunidades = self.client.files.create(file=open("content/festivos-por-provincias-2023.pdf", "rb"), purpose="assistants")
 file_linea105 =
 self.client.files.create(file=open("content/Linea_105_Las_Palmas_Galdar_Horarios.pdf", "rb"), purpose="assistants")

 tools = self.tools_list
 tools.append({"type" : "retrieval"})

 assistant = self.client.beta.assistants.create(
 name=assistant_name,
 instructions=self.system_message,
 model=self.model_name,
 tools=tools,
 file_ids=[file_arenas.id, file_comunidades.id, file_linea105.id]
)

global ASSISTANT_ID
ASSISTANT_ID = assistant.id
dotenv.set_key('.env', 'ASSISTANT_ID', ASSISTANT_ID)
dotenv.set_key('.env', 'FILE_ID_ARENAS', file_arenas.id)
dotenv.set_key('.env', 'FILE_ID_COMUNIDADES', file_comunidades.id)
dotenv.set_key('.env', 'FILE_ID_LINEA105', file_linea105.id)

def create_thread(self):
 """Crea un nuevo hilo y devuelve su ID."""
 thread = self.client.beta.threads.create()

 self.thread_id = thread.id
```

```
def ask_assistant(self, message):

 global ASSISTANT_ID

 if self.thread_id is None:
 self.create_thread()

 self.client.beta.threads.messages.create(
 thread_id=self.thread_id,
 role="user",
 content=message
)

 # Step 4: Run the Assistant
 run = self.client.beta.threads.runs.create(
 thread_id=self.thread_id,
 assistant_id=ASSISTANT_ID,
 instructions=self.system_message
)

 while True:
 # If run is completed, get messages
 run_status = self.client.beta.threads.runs.retrieve(
 thread_id=self.thread_id,
 run_id=run.id
)

 if run_status.status == 'completed':
 messages = self.client.beta.threads.messages.list(
 thread_id=self.thread_id
)

 # Loop through messages and print content based on role
 for msg in messages.data:
 content = msg.content[0].text.value
 return content

 break

 elif run_status.status == 'requires_action':
 print("Function Calling")
 required_actions =
run_status.required_action.submit_tool_outputs.model_dump()
 print(required_actions)
 tool_outputs = []

 for action in required_actions["tool_calls"]:
 func_name = action['function']['name']
```

```
arguments = json.loads(action['function']['arguments'])

func_to_call = self.functions_available.get(func_name)

Verifica si la función existe.
if func_to_call:
 output = func_to_call(**arguments)
 tool_outputs.append({
 "tool_call_id": action['id'],
 "output": output
 })
else:
 raise ValueError(f"Unknown function: {func_name}")

print("Submitting outputs back to the Assistant...")
self.client.beta.threads.runs.submit_tool_outputs(
 thread_id=self.thread_id,
 run_id=run.id,
 tool_outputs=tool_outputs
)
else:
 time.sleep(3)

def delete_thread(self):
 self.client.beta.threads.delete(thread_id=self.thread_id)

def get_system_message(self):
 return self.system_message

def get_tools_list(self):
 return self.tools_list

def get_functions_available(self):
 return self.functions_available

def get_functions_call(self):
 return self.functions_call

def set_api_key(self, api_key):
 self.client.api_key = api_key
```

La clase function\_call, que se le pasa como parámetro, deberá contener las funciones desarrolladas en python que podrá llamar nuestro bot, la tool\_list de funciones en formato json y las funciones disponibles a las cuales el bot podrá llamar en un diccionario de python. Un ejemplo en el repositorio de github adjuntado. Además, assistant\_id y thread\_id se almacenan como variables de entorno, permitiendo que la aplicación acceda a estos identificadores tras un reinicio sin necesidad de recrearlos.

---

## API DE CHAT COMPLETION

---

## Function calls con la API de chat completion:

He incluido un tutorial detallado que te guía paso a paso sobre cómo realizar consultas a un bot utilizando la API de Chat Completion. Este tutorial está disponible en un repositorio de GitHub que contiene un archivo README.md, donde se describen los procedimientos necesarios. Además, hay un cuaderno de Jupyter incluido que ofrece un ejemplo práctico y ejecutable. Puedes encontrar las instrucciones específicas para ejecutar el ejemplo en el paso 8 del archivo README.md.

[Enlace al repositorio del tutorial \(OpenAIFunctionCallsChatCompletionTutorial\)](#)

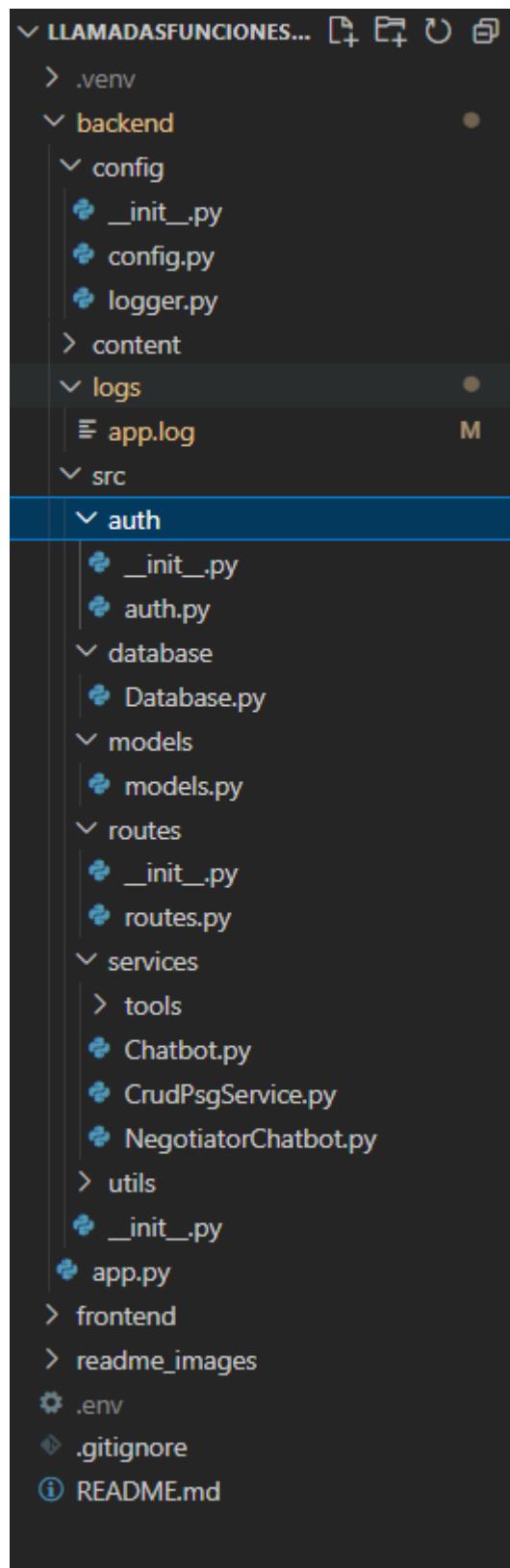
---

## Explicación detallada del código (backend)

---

Para dar inicio al desarrollo de nuestra aplicación con Flask, comenzemos por estructurar el proyecto de la manera siguiente:

Esta estructura organiza eficientemente los componentes y recursos de la aplicación para facilitar su desarrollo, mantenimiento y escalabilidad:



## 1. Configuración de Variables de Entorno con el Archivo .env (.env y config/config.py)

El manejo efectivo y seguro de las configuraciones y claves secretas es crucial en cualquier proyecto de software. Para nuestra aplicación Flask, utilizamos un archivo .env en la raíz del proyecto para definir y almacenar las variables de entorno. Este enfoque nos permite separar la configuración sensible y específica del entorno del código fuente, facilitando cambios sin necesidad de modificar el código y mejorando la seguridad al evitar exponer detalles críticos en el repositorio.

## 1. Estructura del Archivo .env

El archivo .env debe contener las siguientes variables claves que nuestra aplicación necesita para operar correctamente:

```

1 FLASK_ENV=development # o production o test
2 SQLALCHEMY_DATABASE_URI=tu_sqlalchemy_database_uri]
3
4 OPENAI_API_KEY=tu_api_key
5 FILE_ID_ARENAS=FILE_ID_ARENAS
6 FILE_ID_COMUNIDADES=FILE_ID_COMUNIDADES
7 FILE_ID_LINEA105=FILE_ID_LINEA105
8 ASSISTANT_ID=ASSISTANT_ID
9

```

## 2. Uso de Variables de Entorno en la Aplicación

Para acceder y utilizar estas variables de entorno en tu aplicación Flask, puedes usar el paquete python-dotenv, que facilita la carga de estas variables al entorno de ejecución de tu aplicación. Aquí tienes un ejemplo de cómo cargar y usar estas variables en app.py:

```

from dotenv import load_dotenv
import os

Cargar las variables de entorno
load_dotenv()

Acceder a una variable de entorno
openai_api_key = os.getenv('OPENAI_API_KEY')

```

## 3. Carga de Variables de Entorno mediante config.py

Para organizar aún más la configuración de nuestra aplicación y aprovechar las capacidades de Flask para manejar diferentes configuraciones según el entorno (desarrollo, producción, pruebas), definimos una clase Config en config.py que carga estas variables desde el archivo .env:

```

flask_app.md config.py
backend> config > config.py > TestingConfig
1 import os
2 from dotenv import load_dotenv
3
4 load_dotenv()
5
6 class Config:
7 FLASK_ENV = os.getenv('FLASK_ENV')
8 OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')
9 SQLALCHEMY_DATABASE_URI = os.getenv('SQLALCHEMY_DATABASE_URI')
10
11 class ProductionConfig(Config):
12 DEBUG = False
13
14 class DevelopmentConfig(Config):
15 DEBUG = True
16
17 class TestingConfig(Config):
18 TESTING = True

```

Este enfoque nos proporciona una manera estructurada de acceder a las configuraciones de la aplicación, dependiendo del entorno en el que se ejecute. La aplicación puede decidir qué clase de configuración cargar mediante el valor de FLASK\_ENV, centralizando y simplificando el manejo de la configuración.

## 4. Seguridad y Buenas Prácticas

Es crucial que el archivo .env no se suba al repositorio (es decir, debe estar listado en .gitignore) para mantener seguras las claves API, URLs de base de datos, y cualquier otra información sensible.

## 2. Inicialización de la Aplicación Flask (src/\_\_init\_\_.py)

El primer paso en el desarrollo de nuestro proyecto es establecer la forma en que nuestra aplicación Flask será inicializada. Este proceso es crucial para configurar correctamente el ambiente de trabajo, conectando todos los componentes necesarios para que la aplicación funcione de manera eficiente.

Para iniciar nuestra aplicación Flask, utilizamos el archivo src/\_\_init\_\_ que sera instanciado en el archivo app.py en la raíz de nuestro proyecto. El propósito principal de este archivo es instanciar y configurar la aplicación Flask, además de establecer las rutas y los endpoints que serán utilizados.

A continuación, te mostramos un ejemplo de cómo se vería el código dentro de \_\_init\_\_.py para inicializar la aplicación Flask:

```

EXPLORER LLAMADASFUNCIONESOPENAI
> .env
backend
 config
 __init__.py
 config.py
 logger.py
 content
 logs
 app.log
 src
 auth
 __init__.py
 auth.py
 database
 Database.py
 models
 __init__.py
 routes
 __init__.py
 routes.py
 services
 tools
 Chatbot.py
 CrudPgService.py
 NegotiatorChatbot.py
 utils
 __init__.py
 app.py
 frontend
 readme_images
 .env
 .gitignore
 READMEmd
 init.py

flask_app.md _init_.py
backend > src > __init__.py > ...
1 from flask import Flask
2 from config.config import Config, ProductionConfig, DevelopmentConfig, TestingConfig
3 from config.logging import conf_logging
4 from src.database.Database import Database
5
6 conf_logging()
7
8 def create_app(config_class=Config):
9 app = Flask(__name__)
10
11 if Config.FLASK_ENV == 'development':
12 app.config.from_object(DevelopmentConfig)
13
14 elif Config.FLASK_ENV == 'production':
15 app.config.from_object(ProductionConfig)
16
17 else:
18 app.config.from_object(TestingConfig) # it'd be testing
19
20 Database(app) # instanciamos la clase 'Database' pasándole la instancia de la app de Flask
21
22 return app
23
24
25
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\raauli\Desktop\GCID 3º\2º Cuatrimestre Externas\openai\LLAMADASFUNCIONESOPENAI> streamlit run frontend/streamlit_app.py
You can now view your Streamlit app in your browser.
Local URL: http://localhost:8501
Network URL: http://192.168.1.34:8501

```

### 3. Conexión de Rutas en la Aplicación (app.py)

Una vez inicializada la instancia de Flask en nuestro proyecto, el siguiente paso consiste en conectar las rutas definidas para que nuestra aplicación pueda responder adecuadamente a las solicitudes HTTP en los endpoints especificados.

Para integrar las rutas en nuestra aplicación Flask, seguimos un proceso de dos pasos en el archivo app.py:

#### 1. Importación del Blueprint de Rutas:

Primero, importamos el objeto Blueprint que representa el conjunto de nuestras rutas. Este objeto ha sido previamente definido en otro módulo (en este caso, src/routes/routes.py). El código de importación luce de la siguiente manera:

```
from src.routes.routes import api as api_blueprint
```

Aquí, api es el nombre del Blueprint que contiene todas nuestras rutas y lógica de endpoints. Al importarlo como api\_blueprint, lo hacemos disponible en app.py para su registro.

#### 2. Registro del Blueprint en la Aplicación:

Una vez importado el blueprint de rutas, procedemos a registrarla en nuestra instancia de Flask con el siguiente comando:

```
app.register_blueprint(api_blueprint)
```

Este método register\_blueprint es el encargado de conectar el Blueprint de rutas con nuestra aplicación Flask. De esta manera, Flask es informado sobre las rutas y vistas definidas en el Blueprint y puede direccionar adecuadamente las solicitudes HTTP a sus respectivos manejadores.

Mediante estos pasos, aseguramos que las rutas y los endpoints definidos en nuestro Blueprint sean reconocidos por la aplicación Flask, permitiendo una organización modular y clara de la lógica de rutas y vistas. La separación de las rutas en un Blueprint facilita también la escalabilidad y el mantenimiento de la aplicación al segmentar la lógica por funcionalidades o áreas específicas.

The screenshot shows a code editor interface with the following details:

- Left Sidebar (File Explorer):** Shows the project structure. Key files include `app.py`, `flask_app.md`, and several Python files like `backend.py`, `auth.py`, `models.py`, `routes.py`, and `services.py`.
- Top Bar:** Shows tabs for `flask_app.md` and `app.py`. The `app.py` tab is active.
- Code Editor (app.py):** Displays the following code:

```
backend > app.py > ...
1 from src import create_app
2 from src.routes.routes import api as api_blueprint
3
4 app = create_app()
5
6 app.register_blueprint(api_blueprint) # Registraremos el blueprint de la api que creamos en las rutas, o más si quieras organizarlo por tipo de servicio: login, chat, etc.
7
8 if __name__ == "__main__":
9 app.run(host="0.0.0.0") # Damos acceso a todas las IPs. Por defecto, Flask se ejecuta en el puerto 5000
```
- Bottom Terminal:** Shows the command `streamlit run frontend/streamlit app.py` being run in a terminal window. The output indicates the app is running locally at `http://localhost:8501` and network address `http://192.168.1.34:8501`.

## **4. Establecimiento de Rutas y Creación del Blueprint (src/routes/routes.py)**

El tercer paso en el desarrollo de nuestra aplicación Flask implica definir explícitamente las rutas y endpoints que manejarán las solicitudes de los usuarios. Esto se logra mediante la creación del archivo `src/routes/routes.py`, el cual aloja la definición del Blueprint nombrado anteriormente y el registro de todas las rutas disponibles en la aplicación.

## 1. Creación y Configuración del Blueprint

Comenzamos por importar las librerías y módulos necesarios:

```
from flask import request, jsonify, Blueprint, current_app
from src.database.Database import Database
from src.services.CrudPsgService import CRUDService
from src.services.NegotiatorChatbot import Negotiator
from src.auth.auth import Authentication
from src.services.tools.assistant_tools import Tooling
from src.services.Chatbot import Chatbot
```

Estas importaciones incluyen Blueprint de Flask para la creación de rutas, así como clases personalizadas para la gestión de la base de datos, servicios CRUD, autenticación, y nuestra lógica de chatbot, entre otros.

Con estas dependencias disponibles, procedemos a instanciar el Blueprint:

```
api = Blueprint('api', __name__)
```

Aquí, api representa el nombre de nuestro Blueprint, el cual agrupará todas las rutas relacionadas. La modularidad que aporta el uso de Blueprints facilita enormemente la gestión y mantenimiento de las rutas en aplicaciones Flask más grandes y complejas.

## 2. Registro de Rutas y Servicios

Luego, inicializamos los servicios y objetos necesarios que serán utilizados por las funciones de ruta:

```
db_session = Database.db.session
crud_service = CRUDService(db_session, current_app)
auth = Authentication()
negotiator = Negotiator(crud_service, auth)
assistant_tools = Tooling()
chatbot = Chatbot(negotiator)
```

La interacción entre diferentes componentes de la aplicación se establece aquí, evidenciando cómo las rutas pueden comunicarse con la base de datos, ejecutar operaciones CRUD, manejar autenticaciones, entre otras funcionalidades.

Finalmente, definimos la ruta base ('/') como un ejemplo inicial:

```
@api.route('/', methods=['GET'])
def index():
 return jsonify({'response': 'Welcome to my API!'}), 200
```

Esta ruta de ejemplo, accesible mediante un método GET, retorna una cálida bienvenida a los usuarios que acceden a la raíz de nuestra API. Esta función index demuestra la estructura básica de cómo se definirán las rutas en nuestro archivo routes.py.

The screenshot shows a code editor interface with a dark theme. The left sidebar displays a file tree for a project named 'LLAMADASFUNCIONES...'. The current file open is 'routes.py' under the 'src/routes' directory. The code in 'routes.py' is as follows:

```
from flask import request, jsonify, Blueprint, current_app
from src.database.Database import Database
from src.services.CrudPsgService import CRUDService
from src.services.NegotiatorChatbot import Negotiator
from src.auth.auth import Authentication
from src.services.tools.assistant_tools import Tooling
from src.services.Chatbot import Chatbot

api = Blueprint('api', __name__)

db_session = Database.db_session
crud_service = CRUDService(db_session, current_app)
auth = Authentication()
negotiator = Negotiator(crud_service, auth)
assistant_tools = Tooling()
chatbot = Chatbot(negotiator)

API routes
@api.route('/', methods=['GET'])
def index():
 return jsonify({'response': 'Welcome to my API!'}), 200

@api.route('/debts/', methods=['GET'])
def get_debts_by_user_email(user_email):
 return jsonify(crud_service.get_all_debts_by_user(user_email))

@api.route('/create_user', methods=['POST'])
def create_user():
 data = request.get_json()
 # Create user logic here

Streamlit configuration
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

At the bottom, there is a terminal window showing the command to run the Streamlit app and the local network URL.

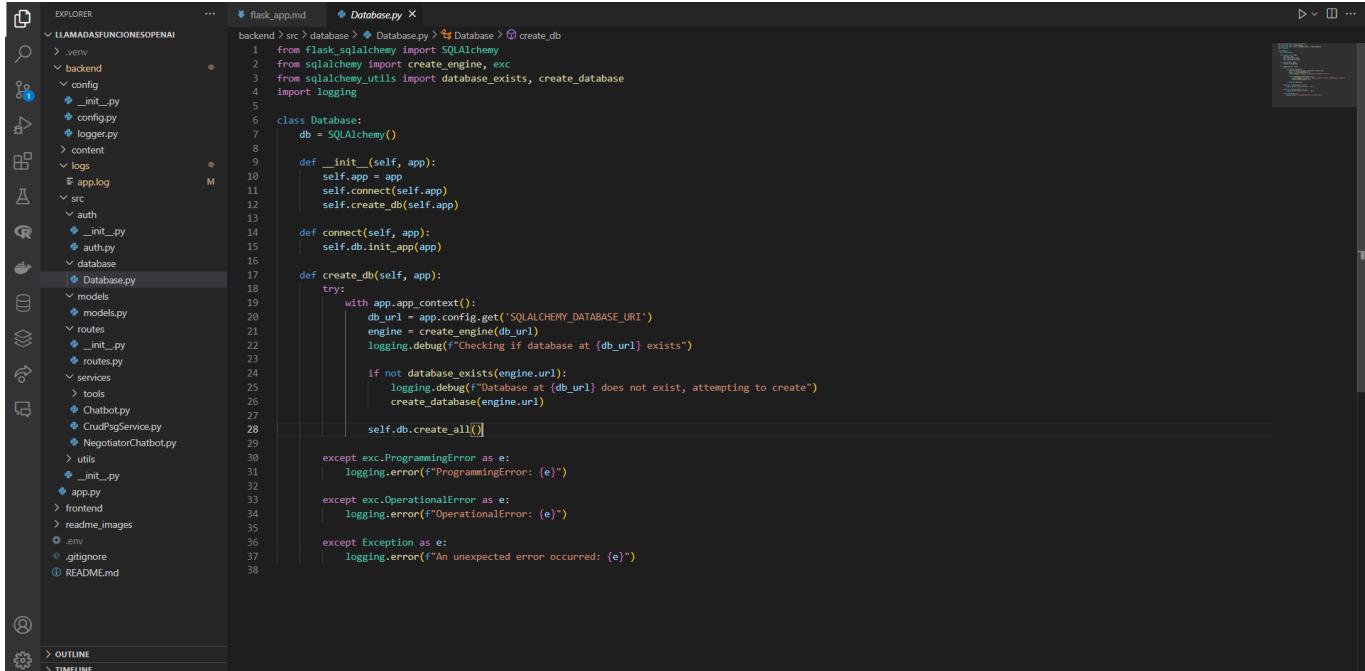
## 5. Inicialización de la Base de Datos con SQLAlchemy (src/models/models.py y src/database/Database.py)

El cuarto paso en nuestra jornada de desarrollo implica la preparación y configuración de nuestra base de datos, empleando SQLAlchemy como ORM (Mapeo Objeto-Relacional, por sus siglas en inglés). Esta poderosa herramienta facilita la interacción con la base de datos a partir de nuestro código Python, permitiéndonos definir modelos y realizar operaciones de base de datos de una manera eficiente y Pythonic.

Para comenzar, es necesario configurar SQLAlchemy y establecer un modelo básico de datos en el fichero Database.py. Este archivo será el encargado de inicializar y gestionar la conexión con nuestra base de datos, así como de verificar su existencia y crearla si es necesario.

## 1. Configuración de SQLAlchemy

El código inicial en Database.py se presenta a continuación:



```

flask_app.md Database.py

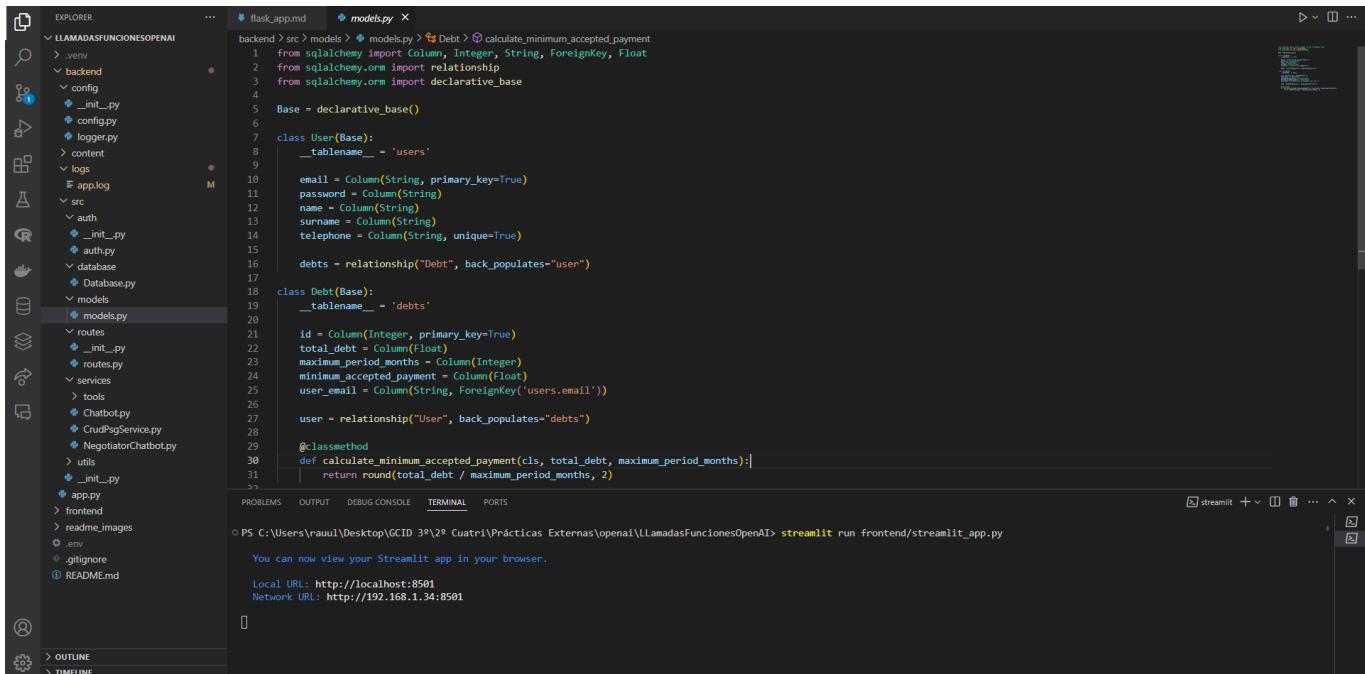
backend > src > database > Database.py > Database > create_db
1 from flask_sqlalchemy import SQLAlchemy
2 from sqlalchemy import create_engine, exc
3 from sqlalchemy_utils import database_exists, create_database
4 import logging
5
6 class Database:
7 db = SQLAlchemy()
8
9 def __init__(self, app):
10 self.app = app
11 self.connect(self.app)
12 self.create_db(self.app)
13
14 def connect(self, app):
15 self.db.init_app(app)
16
17 def create_db(self, app):
18 try:
19 with app.app_context():
20 db_url = app.config.get('SQLALCHEMY_DATABASE_URI')
21 engine = create_engine(db_url)
22 logging.debug("Checking if database at {} exists".format(db_url))
23
24 if not database_exists(engine.url):
25 logging.debug("Database at {} does not exist, attempting to create".format(db_url))
26 create_database(engine.url)
27
28 self.db.create_all()
29
30 except exc.ProgrammingError as e:
31 logging.error("ProgrammingError: ({})".format(e))
32
33 except exc.OperationalError as e:
34 logging.error("OperationalError: ({})".format(e))
35
36 except Exception as e:
37 logging.error("An unexpected error occurred: ({})".format(e))

```

A través de este código, inicializamos SQLAlchemy, creamos la conexión con la base de datos especificada en la configuración de la aplicación (SQLALCHEMY\_DATABASE\_URI), verificamos si la base de datos existe y, si no es así, procedemos a crearla.

## 2. Definición de Modelos Básicos

Con la conexión ya establecida, es el momento de definir nuestros modelos de datos, que se realizan en models.py:



```

flask_app.md models.py

backend > src > models > models.py > Debt > calculate_minimum_accepted_payment
1 from sqlalchemy import Column, Integer, String, ForeignKey, Float
2 from sqlalchemy.orm import relationship
3 from sqlalchemy.orm import declarative_base
4
5 Base = declarative_base()
6
7 class User(Base):
8 __tablename__ = 'users'
9
10 email = Column(String, primary_key=True)
11 password = Column(String)
12 name = Column(String)
13 surname = Column(String)
14 telephone = Column(String, unique=True)
15
16 debts = relationship("Debt", back_populates="user")
17
18 class Debt(Base):
19 __tablename__ = 'debts'
20
21 id = Column(Integer, primary_key=True)
22 total_debt = Column(Float)
23 maximum_period_months = Column(Integer)
24 minimum_accepted_payment = Column(Float)
25 user_email = Column(String, ForeignKey('users.email'))
26
27 user = relationship("User", back_populates="debts")
28
29 @classmethod
30 def calculate_minimum_accepted_payment(cls, total_debt, maximum_period_months):
31 return round(total_debt / maximum_period_months, 2)

```

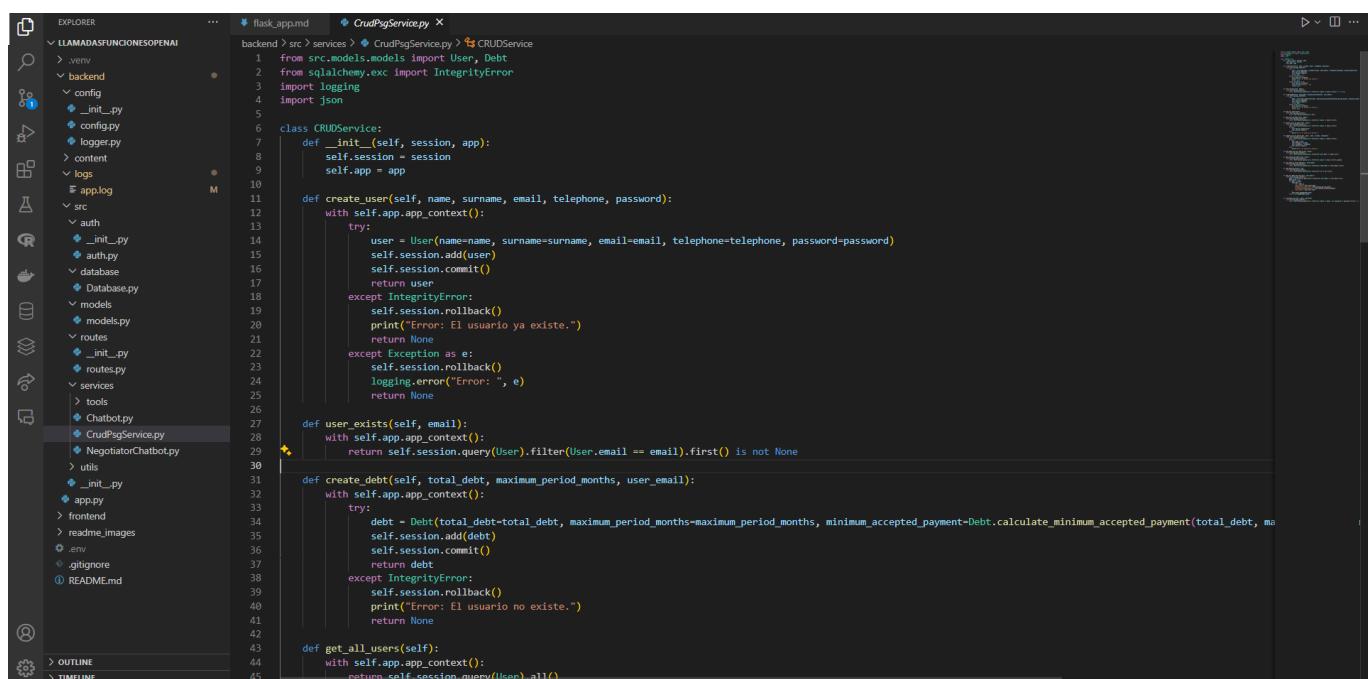
En este ejemplo, se definen dos modelos básicos: User y Debt, que representan usuarios y sus deudas, respectivamente. Estos modelos se basan en Base, que es una clase declarativa de SQLAlchemy para la definición de modelos. Gracias a SQLAlchemy y su integración con Flask, podemos construir y manipular nuestra base de datos directamente desde el código Python.

Con estos pasos completados, hemos sentado las bases para trabajar con datos de forma efectiva en nuestra aplicación Flask, utilizando el modelo ORM de SQLAlchemy.

**Creación de un Servicio CRUD** Tras establecer nuestra base de datos y definir los modelos en SQLAlchemy, el siguiente paso es implementar un servicio CRUD (Crear, Leer, Actualizar, Eliminar) que nos permitirá manipular la información almacenada en la base de datos de forma eficiente. Este servicio se encapsulará en el archivo `CrudService.py`, donde definiremos los métodos necesarios para interactuar con nuestra base de datos mediante SQLAlchemy.

## 6. Implementación del Servicio CRUD (/src/services/CrudPsgService.py)

El servicio CRUD nos brinda un conjunto de operaciones generales aplicables a cualquier modelo de datos definido. A continuación, se detalla una implementación básica del servicio CRUD:



```

EXPLORER ... flask_app.md CrudPsgService.py X
└── LLAMADASFUNCIONESOPENAI
 ├── .venv
 └── backend
 ├── config
 │ ├── __init__.py
 │ ├── config.py
 │ └── logger.py
 ├── content
 ├── logs
 │ └── app.log
 └── src
 ├── auth
 │ ├── __init__.py
 │ └── auth.py
 ├── database
 │ └── Database.py
 ├── models
 │ └── models.py
 ├── routes
 │ ├── __init__.py
 │ └── routes.py
 └── services
 ├── tools
 └── Chatbot.py
 ├── CrudPsgService.py
 └── NegotiatorChatbot.py
 ├── utils
 │ ├── __init__.py
 │ └── utils.py
 └── wsgi.py
 ├── frontend
 ├── README.md
 └── .gitignore

```

```

flask_app.md
backend > src > services > CrudPsgService.py > CRUDService
1 from src.models.models import User, Debt
2 from sqlalchemy.exc import IntegrityError
3 import logging
4 import json
5
6 class CRUDService:
7 def __init__(self, session, app):
8 self.session = session
9 self.app = app
10
11 def create_user(self, name, surname, email, telephone, password):
12 with self.app.app_context():
13 try:
14 user = User(name=name, surname=surname, email=email, telephone=telephone, password=password)
15 self.session.add(user)
16 self.session.commit()
17 return user
18 except IntegrityError:
19 self.session.rollback()
20 print("Error: El usuario ya existe.")
21 return None
22 except Exception as e:
23 self.session.rollback()
24 logging.error("Error: ", e)
25 return None
26
27 def user_exists(self, email):
28 with self.app.app_context():
29 return self.session.query(User).filter(User.email == email).first() is not None
30
31 def create_debt(self, total_debt, maximum_period_months, user_email):
32 with self.app.app_context():
33 try:
34 debt = Debt(total_debt=total_debt, maximum_period_months=maximum_period_months, minimum_accepted_payment=Debt.calculate_minimum_accepted_payment(total_debt, ma
35 self.session.add(debt)
36 self.session.commit()
37 return debt
38 except IntegrityError:
39 self.session.rollback()
40 print("Error: El usuario no existe.")
41 return None
42
43 def get_all_users(self):
44 with self.app.app_context():
45 return self.session.query(User).all()

```

En este snippet de código, `CRUDService` es una clase que encapsula los métodos para realizar operaciones CRUD:

`create`: Inserta una nueva instancia del modelo en la base de datos. `read`: Recupera un registro específico basado en su ID. `update`: Actualiza un registro existente en la base de datos. `delete`: Elimina un registro de la base de datos. Cada método de CRUD requiere que se pase una `model_instance` que representa una instancia del modelo sobre el cual se realizará la operación, o en el caso del método `read`, el `model` y la id del registro a recuperar.

## 1. Contexto de Aplicación y Sesión de Base de Datos

Importante destacar que, para garantizar que las operaciones se ejecuten dentro del contexto de la aplicación Flask y se manejen correctamente las sesiones de base de datos, se utilizan los métodos `with self.app.app_context()`. Esto asegura que cada operación tiene acceso a la configuración y recursos de la aplicación, como la conexión a la base de datos configurada.

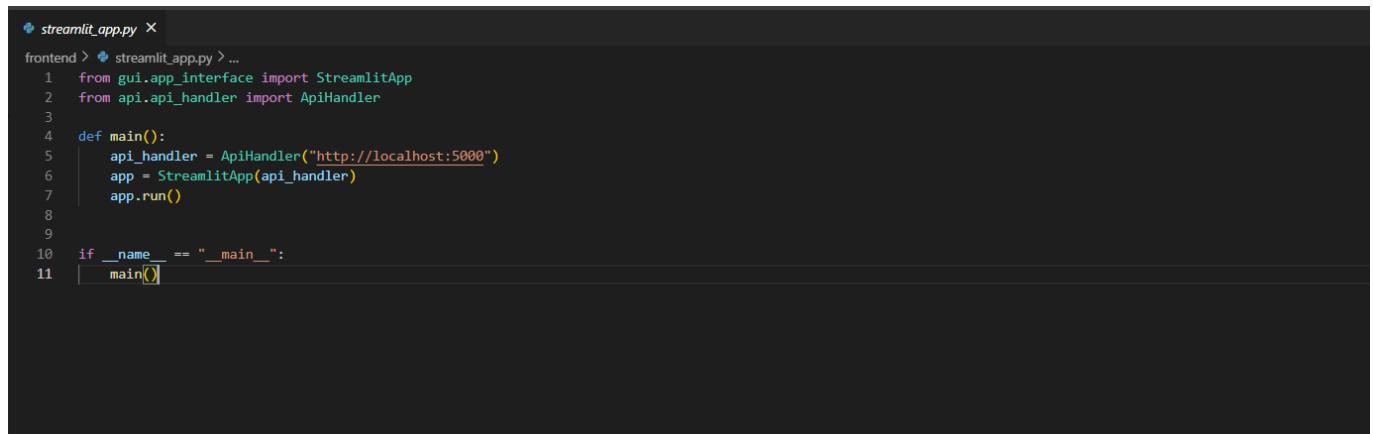
## 2. Uso del Servicio CRUD

El servicio CRUService puede ser instanciado y utilizado por otras partes de la aplicación para interactuar con diferentes modelos de datos, simplificando la lógica necesaria para manipular la información almacenada en la base de datos y promoviendo la reutilización del código.

## Explicación detallada del código (frontend)

### 1. Main de la app (streamlit\_app.py)

El script principal en Python utiliza la clase StreamlitApp, encargada de la interfaz de usuario, y la clase ApiHandler, destinada a manejar las peticiones a una API externa. Al ejecutarse, el script inicia una aplicación web interactiva con Streamlit que permite a los usuarios gestionar deudas y comunicarse con un bot de negociación. Al separar la lógica de la interfaz de usuario y las llamadas a la API, el código ofrece una estructura clara y modular, donde StreamlitApp interactúa con el backend a través del api\_handler creado con la dirección del servicio API, proporcionando así una experiencia de usuario rica e interactiva para la gestión de deudas y autenticación de usuarios.



```
streamlit_app.py
frontend > streamlit_app.py > ...
1 from gui.app_interface import StreamlitApp
2 from api.api_handler import ApiHandler
3
4 def main():
5 api_handler = ApiHandler("http://localhost:5000")
6 app = StreamlitApp(api_handler)
7 app.run()
8
9
10 if __name__ == "__main__":
11 main()
```

### 2. Código streamlit (app\_interface.py)

Este código facilita la interacción entre el usuario y un chatbot de negociación de deudas, manejando la autenticación de usuarios, gestión de API keys de OpenAI, creación de deudas, y limpieza de historial de chat. La interfaz dinámica proporcionada por Streamlit se complementa con llamadas a una API externa para ejecutar la lógica del backend de la aplicación.

#### 1. Configuración Inicial y Barra Lateral:

Se establece el título principal de la aplicación como ' SF OpenAI Chatbot'. Dentro de with st.sidebar:, comienza la configuración de la barra lateral, ofreciendo opciones de acceso y gestión dependiendo del estado de la sesión del usuario (st.session\_state).

```
def run(self):
 st.title('🤖💬 SF OpenAI Chatbot')
 with st.sidebar:
```

## 2. Autenticación y Registro:

Si el usuario no está autenticado ('user' not in st.session\_state), se muestra un radio botón con opciones para "Iniciar Sesión" o "Registrarse". Se invocan los métodos self.display\_login\_section() y self.display\_register\_section() dependiendo de la selección del usuario, permitiendo la autenticación o el registro, respectivamente.

```
with st.sidebar:

 if 'user' not in st.session_state:
 st.title('🔒 Acceso')
 login_option = st.radio('Selecciona una opción', ['Iniciar Sesión', 'Registrarse'])

 if login_option == "Iniciar Sesión":
 self.display_login_section()
 elif login_option == "Registrarse":
 self.display_register_section()
 else:
 self.display_user_section()
 st.empty()
```

## 3. Sección de Usuario y API Key:

Si el usuario está autenticado, se muestra la sección del usuario junto con la opción para ingresar o gestionar una API key de OpenAI, crucial para el funcionamiento del chatbot. Se permite al usuario ingresar una API key. Si la clave es validada con éxito por la API (status\_code == 200), se guarda en el estado de la sesión y se refresca la aplicación (st.rerun()). En caso de error, se informa al usuario.

```
self.display_user_section()
st.empty()

st.title('🔑 API Key')
if 'openai_api_key' not in st.session_state:
 api_key = st.text_input('Introduce tu API key de OpenAI',
 type='password')
 if api_key:
 api_response =
self.api_handler.post_request('set_api_key', {'api_key': api_key})
 st.session_state['openai_api_key'] = api_key

 if api_response.status_code == 200:
 st.success('API key guardada con éxito')
 st.rerun()
 if api_response.status_code == 400:
 st.error('Por favor, introduce una API key de OpenAI
válida')
 else:
```

```
 st.success('API key guardada con éxito, puedes proceder a
chatear con el bot.')
 st.button('Desvincular API key', on_click=self.unlink_api_key)

 st.empty()
```

```
def unlink_api_key(self):
 st.session_state.pop('openai_api_key')
```

#### 4. Creación de Deudas:

Se proporcionan entradas numéricas para que el usuario especifique el total de una deuda y un plazo máximo en meses. Al presionar el botón 'Crear deuda', se invoca self.create\_debt, que envía estos datos a la API para crear una nueva deuda asociada al usuario.

```
 st.title('📝 Crear deuda')
 total_debt = st.number_input('Total de la deuda (euros)')
 maximum_period_months = st.number_input('Plazo máximo (meses)',
step=1, min_value=1, value=1)

 debt_button = st.button('Crear deuda')

 if debt_button:
 debt_response = self.create_debt(total_debt,
maximum_period_months, st.session_state["user"]["email"])
 if debt_response.status_code == 201:
 st.success('Deuda creada exitosamente')
 else:
 st.error('Error al crear la deuda. Por favor, verifica los
datos ingresados.')
```

#### 5. Limpieza de Historial de Chat:

Se ofrece una opción para limpiar el historial del chat, lo que mejora la experiencia de usuario permitiendo iniciar nuevas conversaciones sin distracciones de chats anteriores.

```
 st.title('🗑 Limpiar historial')
 st.sidebar.button('🆕 Nuevo chat',
on_click=self.clear_chat_history)
```

```
def clear_chat_history(self):
 st.session_state.messages = []
 self.api_handler.post_request('clear_chat_history', {})
```

## 6. Interfaz de Chat:

Se envía un mensaje de bienvenida inicial a través de st.markdown(assistant\_message). Se iteran los mensajes en st.session\_state.messages, mostrándolos dentro de burbujas de chat correspondientes a su "rol" (usuario o asistente).

```
if "messages" not in st.session_state:
 st.session_state.messages = []

assistant_message = "¡Hola! Soy el DebtNegotiationBot, tu asistente personal para la negociación de deudas. ¿En qué puedo ayudarte hoy?"

st.markdown(assistant_message)

for message in st.session_state.messages:
 with st.chat_message(message["role"]):
 if message["role"] != "system":
 st.markdown(message["content"])
```

## 7. Interacción con el Chatbot:

Se provee un campo de entrada (st.chat\_input) para que el usuario escriba su mensaje. Este input se valida: Si el usuario no está autenticado o no ha proporcionado una API key de OpenAI válida, se muestra un mensaje de error correspondiente. En caso contrario, se envía el mensaje del usuario al backend a través de self.api\_handler.post\_request('ask\_assistant', {'message': prompt}). La respuesta se procesa y se muestra como un mensaje del asistente.

```
if prompt := st.chat_input("Escribe aquí..."):

 if not "user" in st.session_state:
 st.error('Por favor, inicia sesión para poder usar el chatbot')
 elif not "openai_api_key" in st.session_state:
 st.error('Por favor, introduce tu API key de OpenAI para poder usar el chatbot')
 else:
 st.session_state.messages.append({"role": "user", "content": prompt})
 with st.chat_message("user"):
 st.markdown(prompt)
```

```
with st.chat_message("assistant"):
 typing_message = st.empty()
 typing_message.text("Typing...")

 response = self.api_handler.post_request('ask_assistant',
{'message': prompt}).json()['response']

 typing_message.empty()

 st.markdown(response)
 st.session_state.messages.append({'role': "assistant",
"content": response})
```