

Proyecto – Cambiando el Rumbo - Asistente de viajes

Rafael Rodríguez Vázquez

Bloque 1 – Instalaciones y configuración del modelo (prueba de uso)

Para el siguiente proyecto hemos instalado las siguientes librerías:

```
# Por si es necesario instalar alguna dependencia, mostramos las usadas:
"""!pip install langchain openai torch python-dotenv"""
"""!pip install -U langchain-openai""" # actualizar langchain para poder utilizar chat openai
"""!pip install langdetect""" # Dependencia para detectar el texto de entrada
"""!pip install translatepy""" # dependencia para traduccion.
"""!pip install geopy""" # dependencia para establecer ubicaciones y mapas.
"""!pip install langgraph""" # dependencia para utilizar la memoria en el agente
"""!pip install --upgrade langgraph""" # actualizamos Langraph para usar la funcion de MessagesState
```

- **LangChain** – uso de las chains y la lógica en el flujo de la conversación con el chat.
- **LangGraph** – uso y gestión del flujo para la memoria y resumen del asistente
- **Openai** – Uso de api para utilizar el modelo gpt-3.5 turbo
- **Geopy y translatepy/langdetect** - para las funciones de rutas y ubicación y que la aplicación responda en varios idiomas.
- **Python-dotenv** – Uso para localizar variables de entorno y cargar la contraseña del api de Openai.

Una vez instalada las dependencias, realizamos las importaciones necesarias:

```
from openai import OpenAI # libreria de openAI para usar el modelo
import os # Libreria de os propia de python para navegar a las carpetas locales, la usaremos para identificar la variable de entorno de openAI
from dotenv import load_dotenv # libreria para cargar la variable de entorno del equipo local
```

- Importamos OpenAI, para el uso del api y el modelo gpt.
- Importamos os para poder navegar por nuestro entorno local.
- Importamos load_dotenv desde la librería dotenv, para navegar a las credenciales del api de openai, desde nuestras variables de entorno local.

Procedemos a realizar la prueba para ver si la api y el modelo funcionan correctamente:

```
# cargamos la variable de entorno para obtener la api de openAI
load_dotenv()

# Asignamos la clave api de openAI
client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

# Prueba de conexión con el modelo GPT-3.5 usando un prompt de ejemplo (dejamos el entrecorillado para evitar hacer más solicitudes al cargar el código)
"""chat_completion = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        ("role": "user", "content": "Dí algo interesante en una línea, sobre viajar a Japón")
    ]
)

print(chat_completion.choices[0].message.content)

# frase del modelo, ratifica que este código funciona:

Japón es un país donde lo tradicional se fusiona con lo moderno, ofreciendo una experiencia única llena de cultura, tecnología y gastronomía.
"""

Japón es un país donde lo tradicional se fusiona con lo moderno, ofreciendo una experiencia única llena de cultura, tecnología y gastronomía.
```

Cargamos las variables de entorno con `load_dotenv`, y luego asignamos la clave de OpenAI en una variable llamada `client`, para instanciar el cliente de OpenAI para su uso.

Luego en la siguiente línea de código, encontramos diferentes conceptos:

```
chat_completion = client.chat.completions.create(  
    model="gpt-3.5-turbo",  
    messages=[  
        {"role": "user", "content": "Dí algo interesante en una línea, sobre viajar a  
        Japón"}  
    ]
```

- **Chat_completion**, hace referencia a la variable, que utilizamos para hacer la instancia al llm, donde luego imprimiremos la respuesta de del modelo según los inputs.
- **Client.chat.completions.create**, esta línea crea la solicitud para generar una respuesta en formato de chat, donde:
 - **Model**, es el modelo a utilizar
 - **Messages**, es el input que recibe el modelo, puede ser un mensaje o una lista de mensajes definidos para establecer el flujo de conversación con el modelo. Dentro de este encontramos:
 - **role**, especifica el tipo de mensaje y quien lo hace, en este caso user, indica el mensaje que genera un usuario; pero tambien puede ser: assistant, que en ese caso sería un mensaje enviado por el modelo, o system, que son instrucciones que se le dan al modelo para que tenga un contexto predeterminado o definido.
 - **Content**, se refiere al tipo de contenido que recibirá como input el modelo.

Con esto, tendríamos la estructura básica para que el modelo genere una repuesta.

Para mostrar el contenido, tendremos que poner la siguiente línea de código:

```
print(chat_completion.choices[0].message.content)
```

En esta línea tenemos: **chat_completion**, hace referencia a la instancia de la estructura que recibe el chat y mostrará una respuesta; **.choices[0]**, las respuestas tienen multiples elecciones de respuesta y en este elegimos la primera (**[0]**); **.message.content**, esta parte accede al lugar donde se encuentra el contenido del mensaje de respuesta generado por el modelo.

Bloque 2 – Instalaciones y configuración del llm y langchain (prueba de uso)

Para comprobar el uso del modelo y langchain, importamos:

```
from langchain import PromptTemplate
from langchain_openai import ChatOpenAI
```

- PromptTemplate – para la creación de la plantilla de prompt que instruccionalará al modelo.
- ChatOpenAI – para realizar la configuración del llm. (Se mostrará más adelante la configuración.)

Vamos a realizar la prueba de uso:

```
# Creamos una template (plantilla) de prompt para tratar con langchain y hacer una prueba de chain básica
'''template = """
    Eres un asistente de viajes experto en la sección asiática, destacando los territorios de: Japón, China y Corea.
    Tienes conocimientos detallados sobre estos territorios.
    Pregunta: {consulta}

    Respuesta:
    """'''

# Definimos el prompt templates para langchain
prompt = PromptTemplate(input_variables=["consulta"], template=template)

# Configuramos el modelo LLM para integrar nuestro modelo gpt-3.5 turbo de openAI
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.8, api_key=os.environ.get("OPENAI_API_KEY"))

# Probamos la chain creada, estableciendo el prompt y el modelo en una secuencia runnable
consulta = "¿Cuál es la mejor época para viajar a Japón?"
respuesta = (prompt | llm).invoke({"consulta": consulta})

print("Respuesta del modelo, para una chain básica:\n", respuesta)'''

# print("Respuesta del modelo, para una chain básica:\n", respuesta["content"]) esta línea devuelve la misma línea de código
# pero solo el contenido de la respuesta, no el análisis de tokens.

# Respuesta a esta interacción:
'''content='La mejor época para viajar a Japón depende de tus preferencias personales.
Sin embargo, muchos viajeros consideran que la primavera (marzo a mayo) es la mejor época para visitar Japón,
ya que es cuando los cerezos están en flor y el clima es agradable. Otoño (septiembre a noviembre)
también es una excelente época para visitar Japón, ya que los colores del otoño son impresionantes.
Sin embargo, Japón es un destino maravilloso durante todo el año, así que realmente puedes visitarlo en cualquier momento
que se adapte a tu agenda y preferencias.''''
```

Para la prueba hemos generado una plantilla con las instrucciones que seguirá el modelo; configuramos el modelo (se mostrará en el siguiente punto) y probamos con una consulta sencilla. Donde podemos ver que la respuesta es acorde a la plantilla de prompt y a la consulta realizada.

2.1 – Generamos las chains y las probamos.

Ahora si mostramos la configuración del llm:

```
# Activamos el llm:
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0.6, api_key=os.environ.get("OPENAI_API_KEY"))
```

Para hacer las invocaciones a nuestro modelo, la configuración, la guardamos en la variable **llm** que de ahora en adelante será el uso de nuestro modelo; para esto

utilizamos dentro de llm:

- **ChatOpenAI** – desde la librería de langchain-openai, es la manera que tiene langchain de configurar un modelo de lenguaje que interactúe como chat. Esto recibe tres parámetros:
 - **Model** – hace referencia al modelo que usaremos, en este caso *gpt 3.5 turbo*.
 - **Temperature** – este valor puede oscilar entre 0 y 1, muestra la creatividad y “libre albedrío” que genera el modelo en sus respuestas, teniendo en cuenta que cuando es más cercano a 0 es menos creativo y se centra más en dar una respuesta constructiva y cuando es más cercano a 1 es más “libre” a la hora de generar las respuestas; en nuestro caso 0.6.
 - **Api_key** – Clave api proporcionada en nuestro caso por OpenAI para acceder a su servidor y poder enviar y devolver las interacciones del modelo. El apartado `[os.environ.get("OPENAI_API_KEY")]`, hace referencia a que la clave segura que me ha dado openai, la tengo guardada en la variable de entorno local como OPENAI_API_KEY, por lo que a través del módulo os la localiza para acceder a ella.

Con esto tenemos nuestro llm activo y funcional, como vimos en la sección anterior.

Continuamos con la generación de las chains, que viene a ser las “cadenas” que utilizaremos para elaborar las respuestas.

Para nuestro asistente hemos generado 11 chains diferentes: historia, costumbres, destinos, gastronomía, actividades, logística, medioambiente, souvenirs, eventos, legalidad e itinerarios. De las cuales para no hacer muy extensa la documentación mostraremos un ejemplo de cómo creamos la chain.

En este caso elegimos la chain de gastronomía:

```
# Chain para gastronomía
template_gastronomia = """
Has sido durante 20 años un experto chef, durante este tiempo has recorrido cada rincón del planeta y te has especializado en todo tipo de comidas.
En los últimos 10 años también has participado como crítico gastronómico en diversos países, lo que te ha permitido visitar sus rincones más profundos.
La respuesta que debes ofrecer ha de ser en un tono claro y conciso, destacando tanto los lugares o restaurantes más icónicos como emergentes y tradicionales.
Dependiendo del perfil del viajero:
- Mochileros con poco dinero.
- Familias con niños.
- Parejas de luna de miel.
- Personas adineradas.
- Etc.
Ofrecerás una respuesta ajustada a sus necesidades económicas.

Los detalles que ofrezcas no tienen que ser muy técnicos, centra el foco en explicar con claridad los lugares más convenientes para cada tipo de persona.
Acompaña las recomendaciones con las horas de apertura y cierre estimadas para los locales que recomiendes.
Considera ofrecer tanto restaurantes de categoría, lugares con eventos especiales como música o iluminación determinada y también sitios de comida rápida.

Pregunta:{consulta}

Respuesta:
"""

prompt_gastronomia = PromptTemplate(
    input_variables=["consulta"],
    template=template_gastronomia
)

chain_gastronomia = prompt_gastronomia | llm
```

Para la chain de gastronomía, encontramos:

- **Template_gastronomia** - la plantilla con la instrucción de como el modelo debe generar una respuesta. Esto es algo que con técnicas de prompting y estructura, entre otras se puede generar que el modelo realice una buena respuesta.
- **Prompt_gastronomia** – en esta sección encontramos el siguiente código:
 - `prompt_gastronomia = PromptTemplate(
input_variables=["consulta"],
template=template_gastronomia)`

utilizamos PromptTemplate para generar una estructura del mensaje que recibirá el modelo y así poder elaborar una respuesta. En esta estructura el modelo recibe:

- **input_variable** – Es la variable que define la consulta, es decir, el mensaje que introduce el usuario para saber información sobre gastronomía en este caso.
 - **Template_gastronomia** – es la instrucción que tiene el modelo para generar una respuesta a la consulta.
- **Chain_gastronomia** – es como integramos la chain creada a nuestro llm; tenemos el código:
 - **chain_gastronomia = prompt_gastronomia | llm** – Utilizamos el operador [|], para unir nuestro prompt al modelo de lenguaje, al utilizar esta sintaxis facilitamos el trabajo entre ambos.

Con esto tendríamos nuestra cadena de gastronomía preparada para generar respuestas en este tema, como vemos a continuación:

```
# Prueba para chain gastronomia

tema = "gastronomia"
consulta = """Me gustaría probar comida en Taiwan pero no se bien a donde ir.
Me gustaría probar comida callejera por el día en algun puesto y por la noche visitar un gran restaurante"""

respuesta = obtener_respuesta(
    tema=tema,
    consulta=consulta,
)

print(respuesta)
```

En Taiwan, para probar comida callejera durante el día te recomendaría visitar el Mercado Nocturno de Shilin en Taipei. Este mercado es conocido por su gran variedad de puestos de comida callejera donde podrás probar platos típicos taiwaneses a precios muy accesibles. Es importante tener en cuenta las medidas de higiene y salud al comer en puestos callejeros, por lo que te recomendaría elegir aquellos que se vean limpios y concurridos.

Para la noche, te sugiero reservar en el restaurante Din Tai Fung, famoso por sus deliciosos dumplings y otros platos tradicionales taiwaneses. Este restaurante tiene varias sucursales en Taipei, por lo que podrás elegir la que te quede más cómoda. Es importante destacar que, como persona adinerada, vas a probar algunos de los platos más exclusivos y extravagantes que ofrecen, como el cangrejo de caparazón blando o el pato Peking.

Horario estimado de Mercado Nocturno de Shilin: 17:00 - 00:00 hrs.
Horario estimado de restaurante Din Tai Fung: 11:00 - 21:00 hrs.

De esta manera podemos decir que nuestra chain para gastronomía esta finalizada.

Y siguiendo las instrucciones de este ejemplo, aplicadas a las demás chains, ofrecen la misma generación de respuestas válidas, para cada chain establecida.

Por último, generamos un diccionario, para tener un acceso más inmediato a cada chain:

```
# Generamos un diccionario con las chains

chains_dic = {
    "historia_cultura" : chain_historia_cultura,
    "destinos" : chain_destinos,
    "costumbres" : chain_costumbres,
    "gastronomia" : chain_gastronomia,
    "actividades" : chain_actividades,
    "logistica" : chain_logistica,
    "medioambiente" : chain_medioambiente,
    "souvenir" : chain_souvenir,
    "eventos" : chain_evento,
    "legalidad" : chain_legalidad,
    "itinerario" : chain_itinerario
}
```

Bloque 3 – Configuración y prueba de Agente.

Con nuestras chains creadas correctamente, lo que continua es establecer un agente para que tome la acción de que chain elegir en la interacción del usuario con el asistente.

El agente que hemos elegido será, ***create react agent***:

```
from langgraph.prebuilt import create_react_agent # importación para utilizar el agente

# definimos el modelo (llm) y las herramientas (tools), junto con el prompt y la memoria pre-establecidas.
agent_viajes = create_react_agent(
    model=llm,
    tools=tools_viajes,
    state_modifier=system_prompt
)
```

Para utilizar nuestro agente, primero desde langgraph.prebuilt (funciones internas de langgraph), importamos ***create_react_agent***.

Y para acceder a nuestro agente le nombramos una variable llamada ***agent_viajes***, donde encontramos:

- **model = llm** – nuestro modelo que definimos previamente como llm.
- **Tools = tolos_viajes** – nuestras tools o herramientas, son las chains introducidas al contexto del agente para que las utilice mediante su interacción en el chat.
- **State_modifier = system_prompt** – *state_modifier* es la manera que tiene ***create_react_agent*** para introducir un prompt con instrucciones para el agente; en nuestro caso *system_prompt* contiene esta información.

3.1 – Creación de tools que utilizará el agente

Ya que el agente toma la acción de que chain elegir mediante se produce el flujo conversacional, las tools o herramientas, son las encargadas de conectar las chains al agente.

```
# Para configurar las herrmaientas que utilizara el agente realizamos las siguiente importacion
from langchain.agents import tool, Tool
```

Para generar la construcción de las mismas tools desde la librería de langchain invocamos la función `[.agents]` e importamos tool.

Ya que cada herramienta está asociada a una chain, y tenemos 11, como en el ejemplo anterior mostraremos una para ver su funcionamiento, sirviendo como muestra, ya que todas se construyen de manera parecida.

```
# tool para itinerario
@tool
def itinerario(consulta: str) -> str:
    """
    Te encargas de generar un itinerario de viaje para el lugar o lugares donde se desee viajar. "
    "Debes organizar un itinerario para todos los días que el usuario vaya a viajar, ajustando las preferencias al presupuesto del usuario y a sus
    """
    itinerario_info = chains_dic["itinerario"].invoke(consulta)
    return itinerario_info
```

Según las instrucciones de la página oficial de langchain, para crear las herramientas en las versiones más recientes de langchain, se debe poner el decorador `[@tool]` justo antes de la función que definirá la misma herramienta. Este decorador indica que la función que le continua se registra como una herramienta en el flujo de trabajo de chat/agente.

La función tiene el parámetro consulta seguido de str, `[(consulta: str) -> str]`, donde:

- **Consulta: str** – es la consulta planteada por el usuario, que se recibe en formato texto (str)
- **-> str** – hace referencia a que la respuesta que genere la herramienta se enviara en formato de texto, al igual que la consulta.

Dentro de la función se le añade una especie de prompt, donde se indica lo que debe hacer la herramienta, en este ejemplo generar un .

Luego tenemos, `itinerario_info=chains_dic["itinerario"].invoke(consulta)`, esta línea de código guarda en una variable que retornará la función (`return itierario_info`); en la variable tenemos la selección de la chain itinerario dentro de nuestro diccionario de chains que generamos anteriormente y a traves del método `[.invoke]`, método propio que tiene langchain en sus últimas versiones, con este método conseguimos que se procese la información que hay en consulta, en relación a la chain seleccionada, en este caso itinerario.

Este proceso lo aplicaríamos a las 11 tools establecidas, para generar la correcta interacción del agente. Finalmente unimos todas las herramientas en una lista:

```
# Inicializamos una lista donde incorporaremos las tools creadas con [.append], por si más adelante necesitamos añadir alguna más
tools_viajes = []

# Añadimos las tools
tools_viajes.extend([
    historia_cultura, destinos, costumbres, gastronomia, actividades,
    logistica, medioambiente, souvenir, eventos, legalidad, itinerario
])
```

y la lista [***tools_viajes***] es la que enviamos al agente que gestionará las herramientas.

3.2 – *Funcionalidades extra: traducción y geolocalización.

Para que nuestro asistente sea más versátil, le añadimos dos funcionalidades:

1. Detección de idiomas y traducción.

```
# Extra 1 - Deteccion y traduccion.
from langdetect import detect # importacion para detectar el idioma de la consulta.
from translatepy import Translator # importacion para la traduccion al idioma de la consulta

# Funcion que detecta el idioma del input del usuario, para devolver el output en el mismo idioma
def detectar_idioma(texto):

    try:
        return detect(texto)

    except Exception as e:
        return 'es' # contemplamos que si falla al detectar idioma, el predeterminado sera el español.

# Funcion para traducir la respuesta del agente al idioma de la consulta
def traducir_respuesta(respuesta, idioma):

    if idioma != 'es':
        translator = Translator()
        traduccion = translator.translate(respuesta, idioma)
        return traduccion.result
    else:
        return respuesta
```

Para realizar este bloque, tenemos que hacer las instalaciones de las librerías langdetect y translatepy, e importamos de ellas detect, para detectar el idioma de la consulta y translator para realizar la traducción.

En la función detectar idioma *def detectar_idioma(texto)*, simplemente detectamos el idioma de la consulta, y devolvemos el idioma detectado.

En la función *def traducir_idioma (respuesta, idioma)*, recibimos el idioma que ha sido detectado y la respuesta generada por el modelo, de modo que, con un bucle, en caso de que el idioma sea diferente al español, realizamos la traducción de la respuesta (por defecto siempre es en español) al idioma que ha sido detectado; finalmente se devuelve la respuesta, pero traducida.

2. Geolocalización.

```
# Extra 2 - Geolocalización
from geopy.geocoders import Nominatim # importacion para utilizar el geolocalizador (openstreetmap)
from geopy.distance import geodesic # importacion para calcular la distancia entre dos puntos.

# Inicializacion del servicio de Nominatim
geoloc = Nominatim(user_agent='proyecto_viajes')

# funcion que toma una ubicacion en coordenadas (latitud y longitud) y devuelve una url con el mapa (google maps)
def generar_mapa(direccion: str) -> str:
    ubicacion = geoloc.geocode(direccion)

    if ubicacion:
        lat, lon = ubicacion.latitude, ubicacion.longitude
        enlace_google = f"https://www.google.com/maps?q={lat},{lon}" # generamos un enlace de google maps, con las coordenadas de la ubicacion.
        return f"La direccion solicitada: {direccion}, tiene las siguientes coordenadas:\nLatitud: {lat}\nLongitud: {lon}\nPuedes visitarla en mapa: {enlace_google}"
    else:
        return f"La direccion {direccion}, no ha podido registrarse correctamente. Porfavor vuelva a intentarlo o pruebe con otra direccion"
```

Para las funciones de ubicación y ruta debemos primero instalar la librería geopy, de ella importamos Nominatim (nos generara un servicio de geolocalizador), y geodesic (nos permite calcular distancias).

Iniciamos el servicio de nominatim y luego establecemos una función *def generar_mapa(dirección: str) -> str* con esta función recibimos un parámetro en formato string(texto), que será una dirección de un lugar; con esto utilizamos geoloc.geocode desde geopy, para capturar la ubicación señalada. Luego realizamos un bucle para que la función devuelva un enlace de google maps, con la latitud, la longitud y la ubicación solicitada.

```
# funcion que calculara la distancia entre dos puntos de la ubicacion
def ruta_destino(direccion1, direccion2):
    ubicacion1 = geoloc.geocode(direccion1)
    ubicacion2 = geoloc.geocode(direccion2)

    if ubicacion1 and ubicacion2:
        coordenadas1 = (ubicacion1.latitude, ubicacion1.longitude)
        coordenadas2 = (ubicacion2.latitude, ubicacion2.longitude)

        distancia = geodesic(coordenadas1, coordenadas2).kilometers # calculamos la distancia entre dos puntos

        ruta_direcciones = f"https://www.google.com/maps/dir/{coordenadas1[0]},{coordenadas1[1]}/{coordenadas2[0]},{coordenadas2[1]}"

        return f"La distancia entre {direccion1} y {direccion2} es de {distancia: .2f}\nConsulta la ruta establecida en el siguiente enlace: {ruta_direcciones}"
    else:
        return f"Una ubicación o ambas no se han podido detectar. Porfavor, revise las entradas"
```

Para establecer una ruta, tenemos otra función *def ruta_destino(dirección1, dirección2)*, donde sacamos ambas ubicaciones mencionadas con geoloc.geocode. Luego generamos un condicional donde sacamos las latitudes y longitudes de las direcciones, calculamos la distancia en kilómetros entre ellas y enviamos un enlace de google maps, con ambas rutas y la distancia que existe entre ellas.

Finalmente añadimos las funciones de geolocalización al agente, mediante la creación de tools específicas para estas, de manera que el agente también gestione los servicios de geolocalización en la interacción con el usuario.

```
# Tool para obtener enlace de google maps con la ubicacion
@tool
def generar_mapa(direccion: str) -> str:
    """
    Esta herramienta te permite obtener las coordenadas de un lugar y devolverla en un enlace de Google maps con su ubicación.
    """
    ubicacion = geoloc.geocode(direccion)

    if ubicacion:
        lat, lon = ubicacion.latitude, ubicacion.longitude
        enlace_google = f"https://www.google.com/maps?q={lat},{lon}" # generamos un enlace de google maps, con las coordenadas de la ubicacion.
        return f"La direccion solicitada: {direccion}, tiene las siguientes coordenadas:\nLatitud: {lat}\nLongitud:{lon}\nPuedes visitarla en mapa: {enlace_google}"
    else:
        return f"La direccion{direccion}, no ha podido registrarse correctamente. Porfavor vuelva a intentarlo o pruebe con otra direccion"
```

```
# Tool para generar una ruta en google maps
@tool
def ruta_destino(direccion1: str, direccion2: str) -> str:
    """
    Esta herramienta proporciona la distancia entre dos lugares establecidos y devuelve una ruta que las conecta en un enlace de Google maps.
    """
    ubicacion1 = geoloc.geocode(direccion1)
    ubicacion2 = geoloc.geocode(direccion2)

    if ubicacion1 and ubicacion2:
        coordenadas1 = (ubicacion1.latitude, ubicacion1.longitude)
        coordenadas2 = (ubicacion2.latitude, ubicacion2.longitude)

        distancia = geodesic(coordenadas1, coordenadas2).kilometers # calculamos la distancia entre dos puntos

        ruta_direcciones = f"https://www.google.com/maps/dir/{coordenadas1[0]},{coordenadas1[1]}/{coordenadas2[0]},{coordenadas2[1]}"

        return f"La distancia entre {direccion1} y {direccion2} es de {distancia:.2f}\nConsulta la ruta establecida en el siguiente enlace: {ruta_direcciones}"
    else:
        return f"Una ubicación o ambas no se han podido detectar. Porfavor, revise las entradas"

# Añadimos las herramientas a nuestra lista de herramientas
tools_viajes.extend([generar_mapa, ruta_destino])
```

Para finalizar este bloque, añadimos las dos nuevas herramientas a la lista de tools_viajes, para que la gestione el agente.

Bloque 4 – Gestión de la memoria.

Con nuestras herramientas definidas y nuestro agente completo, ya podemos realizar consultas y que el modelo genere una respuesta. Pero tenemos un inconveniente principal y es que para poder tener una conversación con el chat fluida y con contexto, necesitamos que aunque sea solo en esa interacción se almacene la información establecida en los mensajes del chat.

```
# Configuramos la memoria
memory = MemorySaver()
```

Con memory saver(), podremos guardar toda la interacción para que nuestro chat pueda recuperar información previa.

A su vez esto es un problema genérico porque memory saver,

Para esto una rama de langchaing, langgraph, ofrece dos posibilidades:

- Hacer un trimado por tokens o mensajes.
- Generar un resumen con la información relevante.

En este caso optamos por realizar un resumen de la conversación y su contexto y eliminar mensajes antiguos.

```
from langchain_core.messages import SystemMessage, HumanMessage, RemoveMessage, AIMessage
from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import MessagesState, StateGraph, START, END

import uuid # para generar id de la config en memoria

# Configuramos la memoria
memory = MemorySaver()

# definimos la configuracion que guiara a la memoria
config = {"configurable": {"thread_id": str(uuid.uuid4())}}
```

Desde langgraph importamos los mensajes que procesa internamente el modelo, la memoria y los grafos para la gestión de la memoria.

Establecemos la memoria como hemos visto hace un momento; y langgraph para gestionar la memoria necesita un identificador de la conversación, en este caso utiliza *thread_id* para identificar un id único y poder enlazar la memoria y el grafo a el chat. Para dar siempre un id único dependiendo de la sesión iniciada utilizamos la librería *uuid*.

Para establecer el grafo y cómo funcionará, deben crearse funciones que gestionen las acciones del grafo.

```
def summary(state: MessagesState):
    # guardamos en una variable los mensajes que tenemos en la memoria, a excepcion de el ultimo enviado
    chat_history = state["messages"][:-1]
    last_human_message = state["messages"][-1] # guardamos el ultimo mensaje del usuario

    '''print("\n[DEBUG - summary] Mensajes antes de generar el resumen:")
    for mensaje in chat_history:
        print(f"{{type(mensaje).__name__}} {{mensaje.content}}")
    ...'''

    # Generamos un resumen de la conversacion cuando el historial del chat supere los 3 mensajes
    if len(chat_history) > 3:

        summary_prompt = (
            "Resume la conversacion anterior incluyendo las preferencias del usuario, preguntas especificas, y las recomendaciones o respuestas dadas. "
            "El resumen debe ser suficientemente detallado para que alguien que no haya leído la conversacion entienda el contexto, y pueda ofrecer una r
            "incluyen todos los detalles de la ocnversacion anterior que sean clave, como: preferencias del usuario, preguntas especificas, respuestas de
            "nombres de lugares, actividades, itinerarios, fechas y cualquier informacion relevante. Asegurate de que el resumen sea completo y conserve
        )

        summary_message = llm.invoke(chat_history + [HumanMessage(content=summary_prompt)])

        '''print("\n[DEBUG - summary] Resumen generado:")
        print(f"{{type(summary_message).__name__}} {{summary_message.content}}")
        ...'''

        # Reemplazar los mensajes en el estado con solo el resumen y el último mensaje
        state["messages"] = [summary_message, last_human_message]

        '''print("\n[DEBUG - summary] Mensajes después de actualizar el state:")
        for mensaje in state["messages"]:
            print(f"{{type(mensaje).__name__}} {{mensaje.content}}")'''

    return {"messages": state["messages"]}
```

Empezamos con la función que hará el resumen del contexto y eliminará los mensajes antiguos una vez se haya generado el resumen.

En `def summary (state: MessagesState)`, encontramos:

- **State** – que hace referencia al estado, es decir el estado de la memoria, que unimos a los mensajes en memoria.
- **Chat_history = state["messages"][:-1]** – Una variable donde almacenamos todos los mensajes del chat, a excepción del ultimo, que viene a ser la siguiente consulta del usuario.
- **Last_human_message = state["messages"][-1]** – una variable donde guardamos el último mensaje del usuario, es decir, la última consulta realizada.
- **Condiciona que gestiona la generación del resumen** – en `if len(chat_history) > 3`, aplicamos que si existen más de 3 mensajes en la memoria, se genera un resumen, donde:
 - **Summary_prompt** – es el prompt que tiene el modelo para generar un resumen consecuente con la conversación
 - **Summary_message** – es la acción que le hacemos al modelo para que genere el resumen teniendo en cuenta el prompt y el historial del chat.
- **State["messages"] = [summary_message + last_human_message]** – en esta línea reemplazamos los mensajes del estado solo por el resumen generado y por la última consulta del usuario.

Por últimos devolvemos al formato al state, nuestro state modificado con el resumen.

```
# funcion que generara la respuesta en base al resumen por nuestro agente
def call_model(state: MessagesState):

    '''
    print("\n[DEBUG - call_model] Mensajes antes de la limpieza:")
    for mensaje in state["messages"]:
        print(f"[{type(mensaje).__name__}] {mensaje.content}")'''

    # Si hay más de 7 mensajes, quedarnos solo con los últimos 2 (el resumen y la consulta más reciente)
    if len(state["messages"]) > 7:
        state["messages"] = state["messages"][-4:]

    '''print("\n[DEBUG - call_model] Mensajes después de la limpieza:")
    for mensaje in state["messages"]:
        print(f"[{type(mensaje).__name__}] {mensaje.content}")'''

    response = agent_viajes.invoke({"messages": state["messages"]})

    '''print("\n[DEBUG - call_model] Respuesta generada por el agente:")
    print(f"[{type(response['messages'][-1]).__name__}] {response['messages'][-1].content}")'''

    return {"messages": response["messages"][-1]}
```

Continuamos con la otra función del grafo; la función `def call_model(state: MessagesState)`.

En esta función tenemos por un lado a nuestro agente configurado, que generará la respuesta basada en la consulta del usuario y el estado de la memoria:

- **Tenemos un condicional `if len(state["messages"]) > 7`** – este condicional nos gestiona la cantidad de mensajes que se almacenan en la memoria, de modo

que cuando el número de mensajes en memoria llega a más de siete, alteramos el state de la memoria para dejar solo los 4 últimos, donde siempre contemplamos el resumen generado en la función anterior y los dos últimos mensajes donde uno es la consulta más reciente del usuario.

- **response = agent_viajes.invoke({"messages": state["messages"]})** – aquí generamos la respuesta del agente a la consulta, en base a la modificación del estado de la memoria del paso anterior, devolviendo solo el contenido del mensaje de respuesta generado por el agente.

Con esto tendríamos las funciones que gestionan nuestra memoria.

El siguiente paso que necesita langgraph es crear el flujo del grafo:

```
# Generar el flujo para gestionar el resumen
workflow = StateGraph(state_schema=MessagesState)

# Añadimos el nodo de resumen y llamada al agente
workflow.add_node("agent", call_model)
workflow.add_node("summary", summary)

# Configuramos el flujo para que llame a nuestras funciones
workflow.add_edge(START, "summary")
workflow.add_edge("summary", "agent")
workflow.add_edge("agent", END)

# Compilamos el flujo
app = workflow.compile(checkpointer=memory)
```

Luego inicializamos el estado del grafo, donde añadimos dos nodos que generaran las acciones del grafo, estos nodos `workflow.add_node("agent", call_model)` y `workflow.add_node("summary", summary)`, son los nodos de nuestras funciones anteriores.

ahora estructuramos el flujo del grafo, donde tenemos:

```
workflow.add_edge(START, "summary")
workflow.add_edge("summary", "agent")
workflow.add_edge("agent", END)
```

Ahora cuando enviamos una consulta el modelo pasa por este flujo, donde primero inicia (Start) y va directo a la función de realizar el resumen, una vez generado o no el resumen, pasamos a la función donde el modelo genera la respuesta, cuando el modelo ha generado la respuesta finaliza el flujo y recibiríamos la respuesta del

modelo en el chat, gestionando resumen y mensajes en la memoria, si fuese necesario, como hemos configurado anteriormente.

```
# Compilamos el flujo
app = workflow.compile(checkpointer=memory)
```

Por ultimo compilamos el flujo del grafo a la memoria mediante *checkpointer*, que es como langgraph necesita establecer que todo este proceso se lleva a cabo en la memoria de nuestro asistente.

Llegados a este punto nuestro asistente está listo para funcionar.

Bloque 5 – Prueba de agente.

```
# Importamos OutputParserException para manejar errores del parser de langchain
from langchain.schema import OutputParserException

def conversacion1():
    print("¡Bienvenido!, soy un asistente de viajes, especializado en el este asiático. ¿En qué puedo ayudarte?")

    while True:
        user_input = input("\nConsulta: ")

        if user_input.lower() == 'listo':
            print("\nEspero haberte ayudado con tu viaje. ¡Hasta la próxima!")
            break

        try:
            # Detectar el idioma
            idioma_entrada = detectar_idioma(user_input)
            print(f"Idioma detectado: {idioma_entrada}")

            # Invocar el flujo
            response = app.invoke({"messages": [HumanMessage(content=user_input)]}, config=config)

            # Obtener el resultado del mensaje del agente
            resultado = response["messages"][-1].content

            # Manejo del idioma
            if idioma_entrada == "es":
                print(f"\nRespuesta agente:\n(resultado)")
            else:
                # Traducir la respuesta al idioma detectado
                resultado_traducido = traducir_respuesta(resultado, idioma_entrada)
                print(f"\nRespuesta agente:\n(resultado_traducido)")

        except OutputParserException:
            print("No he podido procesar la consulta adecuadamente. ¿Podrías intentar una entrada diferente?")
            continue

        except Exception as e:
            print(f"Ha sucedido un error inesperado: {str(e)}")
            continue
```

Creamos una función que nos permitirá tener una conversación fluida mediante las celdas de jupyter notebook.

En esta función establecemos un mensaje de bienvenida, seguido de un código para introducir una entrada, luego detectamos el idioma de la consulta, después generamos la respuesta a la consulta pasando por todo nuestro flujo de agente-> herramientas -> agente -> flujo de gestión de memoria -> agente y este envía una

respuesta; en caso de que la respuesta haya sido detectado en otro idioma diferente al español, traduce la respuesta a el idioma detectado.

Como punto extra importamos las excepciones de outputparser, que son los errores relacionados con los parser de langchain, que viene a ser errores producidos internamente por la validación de las respuestas de langchain.

y ejecutamos el bloque:

```
conversacion1()

¡Bienvenido!, soy un asistente de viajes, especializado en el este asiático. ¿En qué puedo ayudarte?

Consulta: Hablame brevemente de la historia de osaka
Idioma detectado: es

Respuesta agente:
Osaka es una ciudad con una rica historia que se remonta siglos atrás. Durante el periodo feudal japonés, Osaka fue un importante centro comercial y político. Uno de los monumentos más emblemáticos de la ciudad es el Castillo de Osaka, una imponente fortaleza que data del siglo XVI y que ha sido escenario de importantes batallas a lo largo de la historia.

Además, en Osaka se pueden encontrar reliquias como la Espada de Heshikiri Hasebe, una espada samurái de gran valor histórico y cultural. Esta espada perteneció a un destacado guerrero japonés y es considerada una verdadera obra de arte.

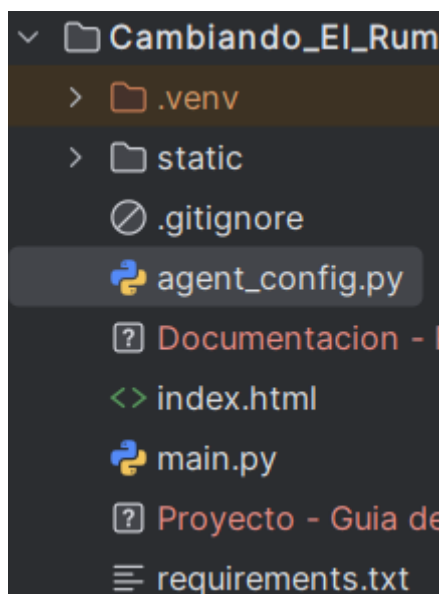
En cuanto a aspectos socio-culturales, Osaka es conocida por su tradicional vestimenta, como el kimono, y por sus antiguos utensilios de cerámica, como las tazas de té. La ciudad también es famosa por su gastronomía, especialmente por el takoyaki, una delicia local que no puedes dejar de probar.

Un dato interesante sobre Osaka es que fue la sede de la Exposición Universal de 1970, un evento que atrajo a millones de visitantes de todo el mundo y que dejó un legado arquitectónico impresionante en la ciudad, como la Torre del Sol.
```

Nuestro asistente ya funciona perfectamente.

Bloque 6 – Generacion de app en FastApi y despliegue en Render.

Este bloque es un bloque final, donde pondremos en despliegue nuestra aplicación, no será tan extenso como la creación del asistente, ya que la idea principal era tratar la creación y uso de la tecnología de langchain/langgraph y la api de OpenAI, pero daremos un vistazo al proceso.



Esta etapa del proyecto se desarrolla en Pycharm, donde en la imagen superior vemos los archivos más importantes.

- **agent_config** – Es el código donde está todo el proceso que hemos visto en los bloques anteriores, la creación del asistente total.
- **Main.py** – El archivo principal donde ejecutamos la aplicación.
- **Index.html** – código javascript, html y estilo css para la visión en la web del proyecto.

Main.py:

```
# Importaciones para ejecutar FastAPI y uvicorn
import uvicorn
from fastapi import FastAPI, HTTPException, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from pydantic import BaseModel

# Importaciones para el uso del chat con langchain
from langchain_core.messages import HumanMessage
from agent_config import app_flujo, config, detectar_idioma, traducir_respuesta
from langchain.schema import OutputParserException

app = FastAPI(title='Asistente de viajes (Cambiando el Rumbo)', version='1.0.0')

# Montamos la carpeta static para que sea accesible en la ruta /static
app.mount("/static", StaticFiles(directory="static"), name="static")

class ChatRequest(BaseModel):
    message: str

# Ruta para servir directamente el archivo index.html
@app.get("/", response_class=HTMLResponse)
async def root():
    with open("index.html", "r", encoding="utf-8") as file:
        return HTMLResponse(content=file.read())
```



```

class ChatRequest(BaseModel): 1 usage  ⚡ Kocxi
    message: str

# Ruta para servir directamente el archivo index.html
@app.get("/", response_class=HTMLResponse)  ⚡ Kocxi
async def root():
    with open("index.html", "r", encoding="utf-8") as file:
        return HTMLResponse(content=file.read())

# Endpoint para el chatbot
@app.post("/chat")  ⚡ Kocxi
def chat_endpoint(request: ChatRequest):
    user_input = request.message

```

```

# Endpoint para el chatbot
@app.post("/chat")  ⚡ Kocxi
def chat_endpoint(request: ChatRequest):
    user_input = request.message

    try:
        # detectamos el idioma de entrada del usuario
        idioma_entrada=detectar_idioma(user_input)

        # generamos la respuesta con nuestro flujo que gestiona el agente y la memoria.
        response = app_flujo.invoke({"messages" : [HumanMessage(content=user_input)]}, config=config)

        # guardamos la respuesta de nuestro agente, en una variable para acceder a ella facilmente
        resultado = response["messages"][-1].content

        if idioma_entrada == 'es':
            return {"response": resultado}

        else:
            resultado_traducido = traducir_respuesta(resultado, idioma_entrada)
            return {"response": resultado_traducido}

    except OutputParserException:
        raise HTTPException(status_code=400, detail="No he podido procesar la consulta adecuadamente. ¿Podrias intentar una entrada diferente?")

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

```

```

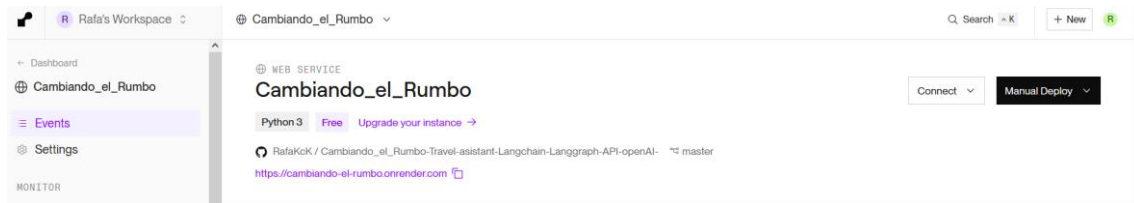
# Punto de entrada para arrancar el servidor con Python
if __name__ == "__main__":
    uvicorn.run(
        app,
        host="127.0.0.1",
        port=8000
    )

```

Todo el código del main, ejecutaría mi agente de viajes sin problemas como veremos más adelante.

Con los archivos completos, subimos el proyecto a github, y desde github conectamos a render mediante web service. Render lee el archivo requirements.txt, que instalará las dependencias necesarias, para desplegar correctamente la función y nos ofrece el siguiente enlace:

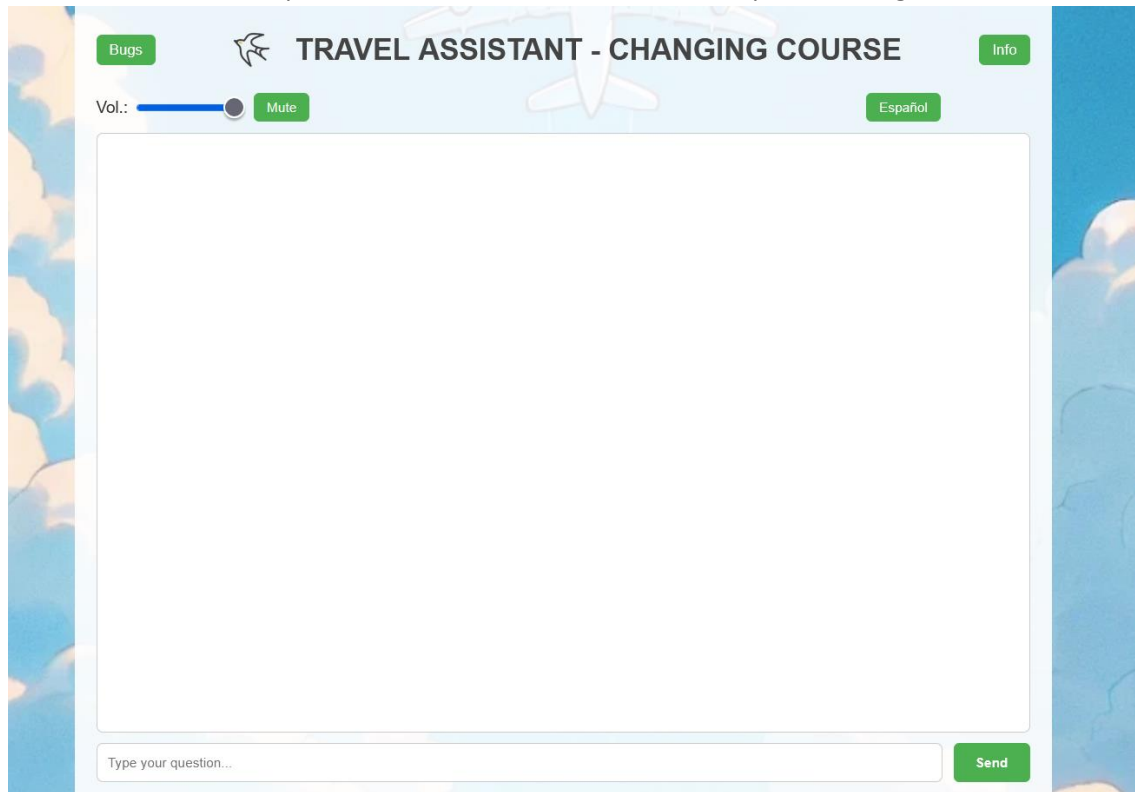
<https://cambiando-el-rumbo.onrender.com>



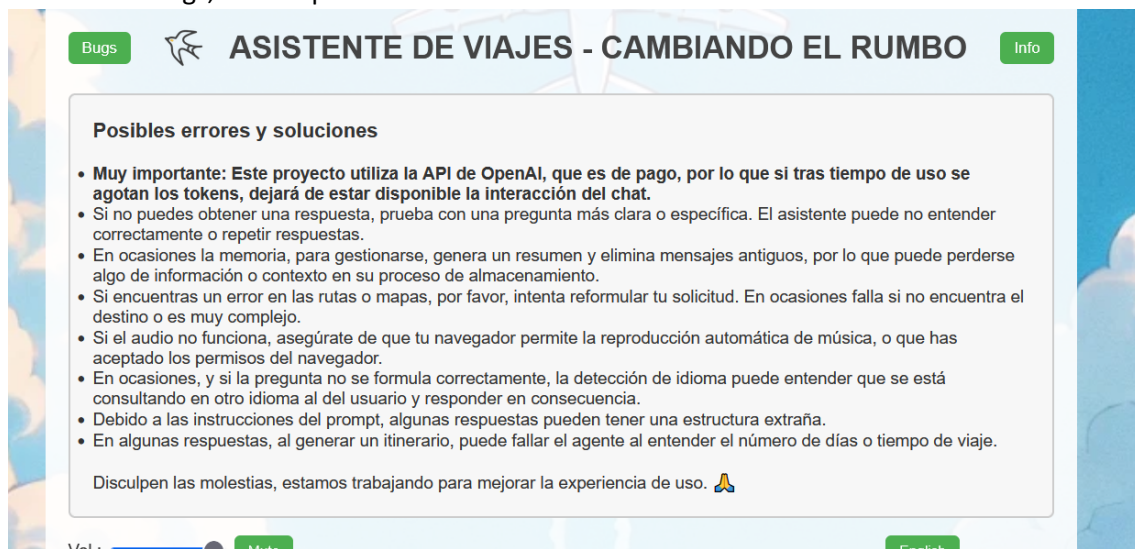
Al clicar en el enlace nos lleva a nuestro proyecto desplegado en la web y de acceso publico:



Donde tenemos varios paneles entre ellos cambio de idioma Español <-> Inglés:




El botón de bugs, con los posibles errores:



El botón de info, con instrucciones:

Bugs


 ASISTENTE DE VIAJES - CAMBIANDO EL RUMBO

Info

Instrucciones y Consejos - Este asistente puede ayudarte a planificar tu viaje. Algunas sugerencias:

- Si se actualiza la página se borra el chat, no guarda información en una base de datos, solo interactúa mientras se opere en esta sesión de chat.
- El Agente de viajes está especializado en 11 temas: Historia, costumbres, destinos, gastronomía, actividades, logística, medioambiente, souvenirs, eventos, legalidad e itinerarios.
- Puedes preguntar: "Recomiéndame actividades en Tokio", "Quiero un itinerario de 5 días", etc.
- Si necesitas rutas o mapas, menciona "ubicación" o "ruta".
- El asistente responde en varios idiomas, ya que tiene funciones de traducción según el idioma con el que se interactúa.
- En caso de error, prueba reformular tu pregunta.
- Es **muy importante** formular la consulta correctamente, para una contestación fluida del asistente.
- Escribe **preguntas claras** relacionadas con viajes y logística. Cualquier otra no será contestada.
- **Importante:** Según del tipo de consulta, el modelo puede tardar más o menos en realizar una respuesta, dependiendo de las consultas internas que deba de realizar y como se le estructure la información. ¡Paciencia! :)

¡Disfruta usando el asistente!


Vol.: 

Mute


English

Y por último hacemos una prueba de chat:

Bugs

 ASISTENTE DE VIAJES - CAMBIANDO EL RUMBO

Info


Vol.: 

Mute


English

Hola me gustaría viajar a japon, en qué epoca me recomendarías ir?

Bugs

 ASISTENTE DE VIAJES - CAMBIANDO EL RUMBO

Info

Vol.: 

Mute

English

Hola me gustaría viajar a japon, en qué epoca me recomendarías ir?

Para viajar a Japón, te recomendaría dos épocas diferentes según tus preferencias:

1. ****Primavera:**** Si te encantan los cerezos en flor y quieres experimentar el famoso hanami (observación de flores de cerezo), te recomendaría viajar a Japón entre finales de marzo y principios de abril. Durante esta época, podrás disfrutar de la belleza de los sakura en lugares como el Parque Ueno en Tokio o el Castillo de Himeji en la región de Kansai.
2. ****Invierno:**** Si prefieres evitar las multitudes turísticas y disfrutar de paisajes nevados, te recomendaría viajar a Japón entre diciembre y febrero. Durante esta temporada, podrás practicar deportes de invierno en lugares como la región de Hokkaido o visitar los famosos baños termales de Nagano.

En resumen, la mejor época para viajar a Japón dependerá de tus preferencias personales. Ya sea que prefieras la primavera para disfrutar de los cerezos en flor o el invierno para disfrutar de paisajes nevados, Japón tiene algo especial que ofrecer en cada temporada. ¡Espero que disfrutes al máximo de tu viaje a este fascinante país!

sea que prefieras la primavera para disfrutar de los cerezos en flor o el invierno para disfrutar de paisajes nevados, Japón tiene algo especial que ofrecer en cada temporada. ¡Espero que disfrutes al máximo de tu viaje a este fascinante país!

Prefiero el clima calido, que ciudad me recomiendas visitar?

¡Qué interesante pregunta! Si estás buscando una ciudad con clima cálido en Japón, te recomendaría visitar Okinawa en primavera u otoño. Esta preciosa isla al sur de Japón es conocida por sus playas de aguas cristalinas, su rica cultura y su deliciosa gastronomía.

En primavera, podrás disfrutar de temperaturas suaves y flores en plena floración, como los famosos cerezos en flor. En otoño, el clima sigue siendo cálido y podrás disfrutar de festivales tradicionales y deliciosa comida local.

Además, en Okinawa podrás escuchar viejas historias y leyendas populares sobre la isla y su gente, así como visitar lugares preciosos como el castillo de Shuri o la playa de Kabira. No te pierdas la oportunidad de probar los mariscos frescos en los mejores restaurantes de la isla.

En resumen, Okinawa es una excelente opción para aquellos que buscan un clima cálido en Japón, y tanto la primavera como el otoño son épocas ideales para visitar esta hermosa isla. ¡Espero que disfrutes tu viaje!

si decido ir para alla, podrias hacerme un itinerario para 4 dias?

¡Aquí tienes un itinerario de 4 días en Okinawa, Japón!

Día 1 en Okinawa, Japón:

- Visitar el Castillo de Shuri.
- Recorrer el Mercado de Makishi.
- Cena en un Izakaya tradicional.
- Paseo por la calle Kokusai.

Día 2 en Okinawa, Japón:

- Visitar el Parque Nacional de Okinawa.
- Hacer snorkel en la playa.
- Cena en un restaurante con vista al mar.
- Disfrutar de un espectáculo de música tradicional.

Día 3 en Okinawa, Japón:

- Excursión a la isla de Iriomote.
- Visitar la cascada de Pinaisara.
- Cena en un restaurante de mariscos.
- Observar las estrellas en la playa.

Día 4 en Okinawa, Japón:

- Visitar el Acuario de Churaumi.
- Paseo en barco por la costa.
- Cena en un restaurante de sushi.
- Caminar por la playa de noche.

¡Espero que disfrutes de tu viaje a Okinawa, Japón! Si deseas realizar alguna modificación en el itinerario o necesitas más recomendaciones, ¡no dudes en decírmelo!

Día 4 en Okinawa, Japón:

- Visitar el Acuario de Churaumi.
- Paseo en barco por la costa.
- Cena en un restaurante de sushi.
- Caminar por la playa de noche.

¡Espero que disfrutes de tu viaje a Okinawa, Japón! Si deseas realizar alguna modificación en el itinerario o necesitas más recomendaciones, ¡no dudes en decírmelo!

puedes enviarme la ubicación del castillo de shuri?

La dirección del Castillo de Shuri en Okinawa, Japón, es la siguiente:

- Latitud: 26.2170014
- Longitud: 127.7193727

¡Puedes visitar el Castillo de Shuri en el siguiente enlace de Google Maps para conocer su ubicación exacta y planificar tu visita: [Ver Ubicación en Google Maps](<https://www.google.com/maps?q=26.2170014,127.7193727>)! Si necesitas más información o ayuda para planificar tu viaje, no dudes en consultarme. ¡Disfruta tu visita al Castillo de Shuri en Okinawa!

Y ese sería un vistazo a como funcionaria nuestra aplicación!

Conclusiones y posibles mejoras.

1. Hemos conseguido desarrollar un asistente de viajes funcional, desde el inicio configurando los entresijos y funcionamientos del mismo y teniendo el resultado de que nos ofrece respuestas coherentes y de utilidad.
2. Hemos logrado establecer una gestión de memoria eficaz, capaz de realizar un resumen que mantiene el contexto de la conversación, permitiéndonos así eliminar mensajes para no saturar la memoria.
3. Hemos generado un interfaz intuitivo, sencillo y dinámico para interactuar con el chat utilizando FastApi, logrando desplegar nuestra aplicación en el servicio gratuito de Render, integrando las funciones del api de OpenAI con éxito, teniendo de este modo una aplicación final desarrollada.
4. Los prompts de cada cadena/herramienta, todavía pueden ser mejorados para dar una tasa de exactitud mayor.
5. La gestión de la memoria, aunque eficiente, puede mejorarse siguiendo las instrucciones de la página de langgraph, gestionando mejor la eliminación de mensajes con los comandos internos de memory saver, o utilizando otra técnica.
6. Las funciones extra de Geolocalización y traducción, pueden optimizarse para ser más precisas y evitar fallos.