Contextos en GPT

Cómo dotar de contexto a GPT: un caso práctico en hotelería

Sinopsis:

Este documento describe con un caso práctico, cómo integrar inteligencia artificial basada en modelos de lenguaje (LLM) en un entorno hotelero real a través del uso de documentos en formato Markdown, embeddings semánticos, búsqueda con OpenSearch y el modelo GPT-4 de OpenAI, se muestra cómo construir un asistente virtual capaz de responder preguntas, realizar tareas, y diferenciar contenido según el rol del usuario (huésped, personal o procesos del sistema). Ideal para desarrolladores, arquitectos de soluciones y equipos de innovación que buscan aplicar IA en contextos reales.

Introducción:

Imaginemos un huésped del **Hotel Hilton Copacabana** usa un servicio de mensajería que tiene como contacto un asistente virtual y pregunta:

> *¿Hasta que hora está abierta la pileta de natación?*

O bien:

> *Quiero hacer una reserva de la piscina a las 15:30*

Ahora supongamos que personal de mantenimiento del hotel pregunta:

> *¿Cuál es el procedimiento para limpiar la pileta de natación?*

O bien:

> *Agregar el contacto del proveedor de filtros de agua a mis contactos*

Cuando uno quiere aplicar inteligencia artificial (AI), en este caso a un hotel y permitir que los huéspedes o personal del hotel dialogue con la AI, es imprescindible dotar al diálogo de un contexto. Sin contexto, ningún modelo podrá generar una respuesta relevante, excepto en casos en los que la pregunta pueda responderse gracias al conocimiento adquirido durante el entrenamiento del modelo.

Modelos como los LLM (Large Language Models), entrenados con billones de *tokens* tomados de fuentes públicas como internet, pueden responder con notable precisión en lenguaje natural, siempre y cuando tengan acceso al contexto necesario. Cuando la AI **no puede generar el contexto por sí misma**, debemos proveérselo explícitamente.

Para ello, se implementa a través de lo siguiente:

1. **Crear una base de conocimiento** en forma de documentos escritos en lenguaje natural. Por ejemplo, procedimientos internos, manuales, instructivos, políticas, etc.

2. **Dividir esos documentos en fragmentos** más pequeños, como secciones, capítulos o párrafos.

3. **Generar un *embedding* por cada fragmento**. Un *embedding* es una representación numérica (un vector de números reales) que codifica la información semántica de un fragmento de texto.

4. **Almacenar estos fragmentos** junto con sus embeddings en un motor de búsqueda semántica como por ejemplo OpenSearch. Esto permite búsquedas rápidas basadas en similitud semántica.

5. Cuando un usuario hace una pregunta, **se genera su embedding**, se compara con los embeddings almacenados, y se recuperan los fragmentos más relevantes.

6. Los fragmentos recuperados se usan como **contexto para responder la pregunta** utilizando el modelo GPT.

Vamos a desarrollar cada uno de estos puntos con un ejemplo práctico. Supongamos que queremos una AI que funcione como asistente virtual del **Hotel Hilton Copacabana** para huéspedes pero también para personal del hotel.

Podríamos estructurar nuestra base de documentos dividiéndola por áreas: piscina, restaurantes, spa, bares, servicio de cuarto, limpieza, etc.

En este ejemplo, vamos a crear un documento para el área de **piscina**, que contendrá información tanto para huéspedes como para el personal del hotel.

Esto nos plantea un problema: **en un mismo documento puede haber información destinada a diferentes tipos de audiencia**, y debemos garantizar que la AI sepa qué puede y qué no puede revelar a cada usuario.

Necesitamos contar con un **repositorio de documentos**. En esta etapa no nos preocupamos por la estructura interna del documento, solo por las características del sistema de gestión de documentos. Este repositorio debe cumplir los siguientes requisitos:

- Permitir almacenar cientos o miles de documentos.

- Permitir la colaboración: que varias personas puedan publicar o editar documentos.

- Contar con control de versiones por documento.

- Permitir revisiones antes de la publicación, incluyendo visualización de cambios.
- Notificar a un sistema externo (mediante *webhook*) cuando un documento es agregado, modificado o eliminado.

GitHub permite crear repositorios públicos o privados y cumple con todo lo mencionado anteriormente.

Ahora sí, debemos dotar a cada documento de una estructura que nos permita dividirlo posteriormente en fragmentos. Existen muchos formatos posibles para estructurar un documento, como por ejemplo XML o JSON. En nuestro caso, vamos a utilizar el formato **Markdown**, una sintaxis ampliamente adoptada por plataformas como GitHub, que permite redactar y visualizar documentos de forma clara, sin sobrecargarlos con marcas complejas.

Para poder descomponer el documento en fragmentos, necesitamos una forma de indicar el inicio de cada uno. Markdown utiliza el símbolo **#** para señalar títulos, por lo que adoptaremos esa convención para marcar el comienzo de cada fragmento.

Veamos un ejemplo de cómo se vería un fragmento en el archivo **`pool.md`**:

```
# Piscina del cuarto piso

## Ubicación

Piscina al aire libre con vista 360 grados panorámica de Río ubicada en el
cuarto piso.

## Horarios

La piscina está disponible de 8:00 a 22:00 todos los días.
Por razones de mantenimiento, puede haber cierres ocasionales los miércoles
por la tarde.

## Capacidad y Zonas

La piscina tiene capacidad máxima para 30 personas.
Dispone de zonas diferenciadas para adultos y niños, con señalización
visible.
```

En este ejemplo se utilizan títulos, subtítulos y párrafos. Al visualizar el documento en GitHub, los títulos aparecen con un tamaño de letra mayor que los subtítulos, y estos a su vez se diferencian de los párrafos, lo que facilita la identificación visual de cada fragmento.

Como mencionamos antes, un mismo documento puede contener fragmentos dirigidos a diferentes audiencias, como por ejemplo los huéspedes o el personal del hotel. Para poder identificar a qué destinatario corresponde cada fragmento, agregaremos un bloque de

propiedades. Para indicarle al parser que se trata de metadatos, utilizaremos un marcador especial con propiedades escritas en formato **YAML**.

A continuación, veamos cómo agregar estas propiedades al documento:

````
# Piscina del cuarto piso

```yaml
properties:
  audience: guest
```
## Ubicación

Piscina al aire libre con vista
[...]
````

Proceso de escritura, publicación y procesamiento de documentos

Repasemos el flujo completo desde la creación del documento hasta su procesamiento final:

1. Se crea un **repositorio privado en GitHub**, por ejemplo: **`documents/`**.

2. Un colaborador autorizado redacta un documento en su entorno local. El documento está compuesto por uno o más **fragmentos**, cada uno con su título, propiedades (en YAML), subtítulos y párrafos.

3. El documento se publica en el repositorio GitHub mediante un comando **push** de Git y a continuación un **pull request** (revisión y merge).

4. El **supervisor del repositorio** recibe una notificación automática indicando que se ha creado o modificado un documento. Revisa los cambios utilizando el control de versiones de GitHub y, si está conforme, aprueba el pull request.

5. Una vez aprobado, GitHub ejecuta el **webhook** configurado en el repositorio. Este webhook realiza una solicitud HTTP(S) a un **endpoint** externo.

6. El **endpoint** recibe la notificación en formato JSON. Esta incluye información como el nombre del repositorio, los archivos afectados (agregados, modificados o eliminados) y detalles relevantes.

7. Cada documento recibido se procesa automáticamente para **extraer los fragmentos** definidos por los encabezados (**#**). Cada fragmento se convierte en una estructura JSON compatible con el almacenamiento en **OpenSearch**, incluyendo tanto el contenido textual como las propiedades asociadas.

Cada fragmento genera una estructura JSON similar a la siguiente:

```
{
   _id: [Identificador único en OpenSearch],
   document: [Nombre del documento],
   prompt: [Título + Subtítulos + Párrafos que figuran en el documento],
   last_update: [Hora universal],
   version: [Versión del documento],
   properties: [Propiedades que figuran en el documento],
   functions: [Funciones que figuran en el documento],
   embedding: [Vector semántico generado por openai],
   sign: [Firma del prompt SHA256 para ese fragmento]
 }
```

Este JSON se utiliza para alimentar el motor de búsqueda semántica basado en **AI**, lo que permite que los huéspedes (o el personal del hotel) realicen preguntas en lenguaje natural y obtengan respuestas precisas, extraídas directamente de los documentos publicados.

Consideraciones sobre procesamiento y uso de AI

Algunas consideraciones clave para el flujo de procesamiento y uso del sistema:

- El **embedding**, que es el vector que permite realizar la búsqueda semántica, se obtiene llamando a la API de **OpenAI** utilizando como entrada el contenido del fragmento (prompt).

- Cuando recibimos desde el **webhook de GitHub** el payload en formato JSON, este indica qué documentos han sido agregados, modificados o eliminados. Dado que puede haber múltiples documentos involucrados, se los procesa de la siguiente manera:

  Documentos eliminados: se eliminan de OpenSearch todos los fragmentos cuyo campo `"document"` coincida con el nombre del documento eliminado.

  Documentos agregados: se parsea el documento, se generan los embeddings para cada fragmento y se indexan en OpenSearch.

  Documentos modificados: se obtiene tanto la versión actual como la anterior del documento. Se calculan firmas (hashes) de cada fragmento para detectar diferencias. Esto permite actualizar OpenSearch agregando, modificando o eliminando únicamente los fragmentos afectados, optimizando el proceso.

- El uso del asistente de **AI** se realiza a través de un sistema de mensajería del hotel. Uno de los contactos disponibles se llama "AI". Cuando un huésped o miembro del personal envía un mensaje por texto o voz a "AI", el sistema identifica automáticamente al emisor y su rol (por ejemplo: huésped, encargado de limpieza, mantenimiento, supervisor, etc.).

- Al realizar la búsqueda en OpenSearch, se utiliza el embedding de la pregunta como clave de búsqueda semántica. Sin embargo, se aplica un filtro adicional sobre la

propiedad "`audience`" del fragmento, tal como está especificado en cada fragmento del documento. Esto garantiza que, por ejemplo, un huésped no reciba como contexto información interna reservada para el personal del hotel.

Supongamos que un huésped realiza la siguiente consulta:

¿Hasta qué hora está abierta la piscina?

El sistema consulta OpenSearch usando embeddings y obtiene como resultado el siguiente fragmento:

```
Piscina del cuarto piso
Horarios
La piscina está disponible de 8:00 a 22:00 todos los días.
Por razones de mantenimiento, puede haber cierres ocasionales los miércoles
por la tarde.
```

La estructura del mensaje a enviar a la API de OpenAI, incluyendo el contexto recuperado y la pregunta del huésped, sería como sigue:

```
const messages = [
  {
    role: 'system',
    content: `Sos un asistente de hotel. Respondé en forma clara y cortés.
  },
  {
    role: 'assistant',
    content: `Contexto:\n${context}`, //fragmento(s) obtenidos de OpenSearch
  },
  {
    role: 'user',
    content: `${question}`, // la pregunta del huésped
  }
];
```

La respuesta generada por el modelo podría ser, por ejemplo:

“La piscina está abierta hasta las 22:00 todos los días.”

Hasta aquí se ha utilizado solo el o los fragmentos obtenidos de OpenSearch como contexto. Sin embargo, para casos en los que exista un intercambio más prolongado, **la estructura debería incluir también todo el hilo del diálogo anterior**, respetando el orden cronológico de los mensajes (recordemos que OpenAI es stateless).

Esto implica agregar todos los mensajes previos del usuario (`role: 'user'`) y las respuestas de la AI (`role: 'assistant'`), de manera que el modelo pueda interpretar correctamente la conversación completa y mantener coherencia y contexto en sus respuestas.

Idiomas y funciones dinámicas

Debemos considerar lo siguiente: en un hotel, los huéspedes hablan distintos idiomas. Cuando se genera un embedding utilizando la API de OpenAI, el sistema detecta automáticamente el idioma del mensaje y genera el vector semántico correspondiente. Esto permite realizar búsquedas por similitud en OpenSearch independientemente del idioma original del huésped.

Sin embargo, los documentos almacenados en OpenSearch en este caso están escritos en castellano, por tanto, cuando el huésped escribe en otro idioma, el sistema debe:

1. Detectar el idioma de la consulta del huésped.
2. Generar el embedding con OpenAI.
3. Realizar la búsqueda en OpenSearch usando ese embedding.
4. Traducir el fragmento recuperado (en castellano) al idioma del huésped.
5. Incluir el fragmento traducido como parte del contexto (`role: 'assistant'`) antes de invocar a OpenAI para generar la respuesta.

De este modo, el huésped puede interactuar en su propio idioma, mientras que la documentación interna se mantiene en un solo idioma base.

En general, los documentos contienen información estática o que se actualiza con poca frecuencia. En algunos casos, es necesario obtener información en tiempo real o ejecutar una acción a partir del mensaje. Por ejemplo, reservar un servicio o agregar un contacto como vimos al comienzo. En estos casos, la respuesta no proviene directamente del contenido estático de los documentos, sino de una **función definida dentro del contexto**.

Las funciones permiten extender la capacidad del asistente para interactuar con sistemas propios del hotel. En el contexto que se le pasa a la API de OpenAI, puede incluirse una o varias funciones disponibles. Cada función especifica:

- Un nombre identificador.
- Una breve descripción.
- Los parámetros requeridos.
- El tipo de cada parámetro.
- Un endpoint HTTP para realizar la llamada.

Entonces, agreguemos al documento un ejemplo de función, también en formato YAML:

```
```yaml
properties:
  audience: guest

functions:
  - name: reserve
    description: Reservar piscina
    parameters:
      - name: hour
        type: string
        required: true
      - name: guests
        type: string
        required: true
    endpoint: https://api.hotelclass.com/pool/reserve
```


Reserva

No es obligatorio realizar una reserva, pero hacerlo nos permite tener
listas las reposeras y toallones para cuando llegue.
Los huéspedes que reserven con anticipación recibirán una copa sin cargo
como cortesía.
```

El permitir que un huésped reserve la piscina no solo mejora su experiencia, sino que también le brinda al hotel métricas útiles sobre la ocupación por franja horaria.

Supongamos que un huésped dice:

“*Quiero reservar la piscina para dos personas a las 15:30”*

En este caso, se ejecuta el siguiente proceso:

1. Se genera el embedding a partir del mensaje del huésped.

2. Se realiza la búsqueda en OpenSearch usando ese embedding.

3. OpenSearch devuelve un fragmento que incluye la definición de una función utilizable.

4. El sistema detecta que el mensaje puede dar lugar a una ejecución de función.

5. Se arma la estructura del requerimiento para la API de OpenAI incluyendo el mensaje, el contexto y la definición de la función.

6. El modelo puede decidir automáticamente si invocar la función, y con qué parámetros.

La estructura para invocar a OpenAI es:

```
const completion = await openai.chat.completions.create({
  model: 'gpt-4',
  messages: [...], // contexto como vimos antes, incluyendo el hilo del diálogo
  functions: functions, // array con las funciones disponibles en el fragmento
  function_call: 'auto' // permite al modelo decidir si debe llamar o no a la
función
});
```

La función definida en el documento `pool.md` para este caso se declara así:

```
const functions = [
  {
    name: 'reserve',
    description: 'Reservar piscina',
    parameters: {
      type: 'object',
      properties: {
        hour: { type: 'string' },
        guests: { type: 'string' }
      },
      required: ['hour', 'guests']
    }
  }
];
```

Cuando el modelo decide invocar la función, genera los parámetros a partir del mensaje del usuario:

```
{
  "name": "reserve",
  "arguments": {
    "hour": "15:30",
    "guests": "2"
  }
}
```

Finalmente, el sistema realiza la llamada HTTP al endpoint indicado en la función, por ejemplo::

```
POST https://api.hotelclass.com/pool/reserve
Content-Type: application/json
{
  "hour": "15:30",
  "guests": "2"
}
```

La respuesta del endpoint (por ejemplo, “Confirmado”) es devuelta al huésped en lenguaje natural, como:

“La piscina ha sido reservada para dos personas a las 15:30. Como cortesía, recibirán una copa sin cargo. ¡Gracias por anticiparse!”

Ejemplos de interacción y comportamiento del modelo

Veamos cómo el modelo interpreta frases naturales y responde de forma contextualizada, considerando las funciones disponibles y las restricciones del documento.

[Huésped]
“Reservar la pileta de natación a las cinco de la tarde”

Aunque se utilice la expresión *pileta de natación* en lugar de *piscina*, o se diga *cinco de la tarde* en lugar de *17 horas*, el modelo comprende perfectamente la intención gracias al procesamiento semántico del embedding.

Sin embargo, en este caso no se especifica la cantidad de personas, y ese parámetro fue declarado como obligatorio en la definición de la función dentro del fragmento. Por lo tanto, la respuesta del modelo será:

[AI]
“Claro, puedo ayudarte con la reserva. ¿Podrías indicarme cuántas personas serán?”

[Huésped]
“Somos dos”

Entonces el modelo ya dispone de todos los parámetros requeridos y realiza la invocación a la función. La respuesta generada podría ser:

[AI]
“¡Perfecto! La reserva para la piscina a las 17:00 horas para 2 personas fue confirmada. Como cortesía, tendrán una copa sin cargo. ¡Que lo disfruten!”

Ahora veamos otro ejemplo. El huésped dice:

[Huésped]
"Quiero reservar la piscina a las 11 de la noche."

En este caso, el fragmento de contexto incluye explícitamente el horario de funcionamiento de la piscina: de 8:00 a 22:00. El modelo lo tiene en cuenta antes de intentar invocar la función, y responde:

[AI]
La piscina está disponible hasta las 22:00, por lo tanto no es posible reservarla a las 23:00. ¿Le gustaría reservarla a las 22:00 o en otro horario disponible?

Veamos ahora otro fragmento del documento, esta vez dirigido al personal de mantenimiento de la piscina:

````
# Piscina del cuarto piso
```yaml
properties:
    audience: hotel
    role:pool-maintenance
```
## Mantenimiento diario

El proceso de limpieza comienza a las 5:00 y debe finalizar antes de las 7:30.
Utilizar únicamente productos aprobados por la administración.
Registrar las tareas realizadas en el sistema de mantenimiento.

## Mantenimiento paso a paso

1. Verificar que la piscina esté vacía de personas y elementos flotantes.

2. Apagar el sistema de recirculación de agua.

3. Vaciar parcialmente el agua si es necesario (nunca completamente salvo
mantenimiento profundo).
[...]
## Protocolos especiales
En caso de lluvia intensa, suspender uso y evaluar limpieza adicional.

Informar a recepción si el mantenimiento impide la apertura a horario.

Supervisar que todos los productos estén correctamente almacenados.

## Auditoría y control
El supervisor de mantenimiento debe realizar una revisión semanal y verificar que:

Se sigan los pasos del procedimiento.

Los productos utilizados estén registrados.
````

Si un miembro del personal de mantenimiento pregunta:

“¿Cuáles son los pasos para limpiar la piscina?”

La AI responde en función del procedimiento detallado bajo el subtítulo Mantenimiento paso a paso.

Ahora imaginemos otro escenario: quien formula la consulta no es un humano, sino un proceso automatizado del sistema del hotel. Por ejemplo, un backend que necesita generar una página con los pasos del procedimiento. La petición podría ser:

“"Generame una tabla en formato HTML de dos columnas, paso y descripción con el procedimiento completo para limpiar la piscina. "

La AI, accediendo al contexto del fragmento, responde con una estructura HTML como la siguiente:

```
<table border="1" cellpadding="5" cellspacing="0">
  <thead>
    <tr>
      <th>Paso</th>
      <th>Descripción</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>1</td>
      <td>Verificar que la piscina esté vacía de personas y elementos
flotantes.</td>
    </tr>
    <tr>
      <td>2</td>
      <td>Apagar el sistema de recirculación de agua.</td>
    </tr>
    <tr>
      <td>3</td>
      <td>Vaciar parcialmente el agua si es necesario (nunca completamente salvo
mantenimiento profundo).</td>
    </tr>
  </tbody>
</table>
```

La respuesta es un texto estructurado que puede insertarse directamente en una página web y ser interpretado por un navegador.

Esto demuestra que ya no solo las personas interactúan con la AI, **¡también lo hacen los procesos!**
La AI se convierte así en un punto de integración entre humanos y sistemas, entregando respuestas útiles tanto para el lenguaje natural como para la automatización.

Conclusiones Finales:

La integración de inteligencia artificial en entornos como el hotelero no solo es posible, sino que resulta altamente efectiva cuando se construye sobre una base sólida de contexto documental y semántica.

Algunos puntos clave que emergen de los expuesto:

- **El contexto es esencial.** La AI no puede inventar lo que no sabe. Dotarla de conocimiento mediante fragmentos bien estructurados permite respuestas

relevantes, específicas y seguras.

- **El uso de Markdown como formato base** permite simplicidad y claridad, a la vez que facilita el versionado, colaboración y automatización del flujo de trabajo.

- **El filtrado por audiencia (por ejemplo, huéspedes vs. personal)** asegura que la información entregada sea adecuada y acorde a cada rol, preservando confidencialidad y pertinencia.

- **La AI puede realizar tareas, no solo responder.** Mediante funciones definidas en el contexto, es capaz de ejecutar acciones como hacer una reserva, incluso completando parámetros faltantes mediante diálogo dinámico.

- **Los procesos automatizados también pueden interactuar.** No solo los humanos hacen preguntas; sistemas internos pueden solicitar respuestas en formatos estructurados (como HTML, JSON, XML u otro), mostrando la versatilidad del enfoque.

Todo lo anterior es aplicable a cualquier entorno donde existan procedimientos, roles y necesidad de acceso eficiente al conocimiento. El resultado es un sistema inteligente, flexible y contextualizado, que dialoga, responde y actúa de forma efectiva.

Puede verse el documento usado en este ejemplo en el repositorio público de GitHub:

https://github.com/hotelclass/documents

#InteligenciaArtificial #GPT4 #LLM #OpenAI #ContextoSemántico #Embeddings #BúsquedaSemántica #OpenSearch #AIAplicada #ProcesamientoDeLenguajeNatural #Markdown #HotelTech #DesarrolloAI #Automatización #AIconContexto #ChatbotInteligente #AWS #Hospitality

Marcelo Rocha
AWS Solution Architect Professional
AWS Developer Engineer Professional