# Cómo crear un chatbot con memoria y prompts usando Go, Ollama y LangchainGo

En este post quiero mostrarte cómo crear un chatbot inteligente que recuerde lo que has dicho antes (memoria), entienda lo que le preguntas (prompt) y responda de forma natural gracias a un modelo de lenguaje (LLM). Lo mejor de todo: ¡todo esto lo hacemos usando Go!

**Tecnologías que vamos a usar**

Go (Golang)

Go es un lenguaje de programación compilado, concurrente, y muy eficiente. Su sintaxis es sencilla y permite crear programas rápidos y robustos. Perfecto para aplicaciones backend como chatbots.

LLMs (Large Language Models)

Los LLMs son modelos entrenados con enormes cantidades de texto para generar lenguaje natural. En este caso usamos llama3.1:8b, que es un modelo de código abierto y muy capaz.

LangchainGo

Es la versión en Go de la famosa librería Langchain, que permite trabajar de forma modular con LLMs. Nos facilita tareas como: crear prompts personalizados, manejar memoria conversacional, conectar fácilmente con proveedores de modelos como Ollama.

Ollama

Ollama es una herramienta que nos permite correr modelos LLM localmente. Así podemos tener un chatbot inteligente sin depender de APIs externas ni la nube.

**Estructura de nuestro chatbot**

En el código hay cuatro partes principales:

1. Crear el modelo de lenguaje

```
llm, err := ollama.New(ollama.WithModel("llama3.1:8b"))
```

Este es el motor que genera las respuestas. En este caso usamos Ollama para conectarnos a un modelo local llamado “llama3.1:8b”.

También crearemos nuestro contexto raíz que utilizaremos para pasar información entre funciones:

```
ctx := context.Background()
```

2. Definir el prompt

Usamos ChatPromptTemplate para estructurar la conversación:

```
systemMessage := "Eres un profesor de secundaria y reponderás de forma fácil a las preguntas que te haga el usuario."
```

Esto le dice al modelo cómo comportarse: en este caso, como un profe, claro y paciente. También incluimos el historial de la conversación y la nueva pregunta.

```
promptTemplate := prompts.NewChatPromptTemplate([]prompts.MessageFormatter{
    prompts.NewSystemMessagePromptTemplate(
        systemMessage,
        nil,
    ),
    prompts.NewGenericMessagePromptTemplate(
        "history",
        "{{range .historyMessages}}{{.GetContent}}\n{{end}}",
        []string{"historyMessages"},
    ),
    prompts.NewHumanMessagePromptTemplate(
        `[Brief] Responde mi pregunta. Esta es mi pregunta: {{.question}}`,
        []string{"question"},
    ),
})
```

3. Memoria de la conversación

```
history := memory.NewChatMessageHistory()
```

Cada vez que el usuario hace una pregunta y el modelo responde, guardamos ambos mensajes para que el modelo tenga contexto en futuras respuestas.

```
historyMessages, err := history.Messages(ctx)
```

```
promptText, err := promptTemplate.Format(map[string]any{
    "historyMessages": historyMessages,
    "question":        question,
})
```

¿Por qué usar memoria?

Sin memoria, el modelo no sabe qué dijiste antes. Con memoria, puede dar respuestas coherentes en una conversación larga, recordar tus dudas previas y hasta corregirse.

```
history.AddUserMessage(ctx, question)
history.AddAIMessage(ctx, respuesta)
```

4. Generación de la respuesta con streaming

```
fmt.Println("Generando respuesta...")

var respuesta string
_, err = llms.GenerateFromSinglePrompt(ctx, llm, promptText,
    llms.WithStreamingFunc(func(ctx context.Context, chunk []byte) error {
        fmt.Print(string(chunk))
        respuesta += string(chunk)
        return nil
    }),
)
```

Esta función hace que el modelo de lenguaje (LLM) genere la respuesta completa a partir del prompt, pero en lugar de esperar a que termine para mostrar todo junto, va recibiendo la respuesta por partes (chunk) y las muestra en tiempo real a medida que el modelo las produce.
Esto permite que el texto se muestre en consola a medida que se genera, lo que da una sensación más natural e interactiva — como si estuvieras viendo al asistente pensar y hablar en tiempo real.

Además, vamos acumulando cada chunk en la variable respuesta, para después guardar esa respuesta completa en el historial del chat, manteniendo el contexto de la conversación.

**Flujo del chatbot:**

- Pides al usuario que escriba una pregunta.

- Se construye un prompt con el historial y la nueva pregunta.

- El LLM genera una respuesta.

- Se muestra la respuesta en consola.

- Se guarda la pregunta y la respuesta en el historial.

**Conclusión**

Crear un chatbot en Go con memoria y prompts es más fácil de lo que parece gracias a LangchainGo y Ollama. Puedes tener un asistente conversacional, ejecutándose localmente, sin preocuparte por latencias o costos de APIs.

Si les interesa ver el código en detalle o charlar un poco más sobre cómo aplicar IA en Go, estoy a disposición. ¡No duden en escribirme!

Saludos.

#Go #AI #GenAI