how to build an LLM-powered chatbot using LangChain, Groq API, and message history management. This code is designed to handle conversational context for different sessions, enabling the chatbot to remember prior interactions based on unique session IDs.

**Explanation of the Code Structure**

1. **Environment Setup and API Key Loading**:

python

Copy code

import os

from dotenv import load\_dotenv

load\_dotenv()

The code loads environment variables (like the Groq API key) from a .env file, which securely stores sensitive data without hardcoding it in the script.

1. **Setting Up the Language Model**:

python

Copy code

from langchain\_groq import ChatGroq

groq\_api\_key = "YOUR\_GROQ\_API\_KEY" # Replace with your actual key

model = ChatGroq(model="Gemma2-9b-It", groq\_api\_key=groq\_api\_key)

Here, the ChatGroq model from Groq is initialized with a specific API key and model version (Gemma2-9b-It). This allows calling the language model (LLM) for conversational tasks.

1. **Simple Chat with Memory**:

python

Copy code

from langchain\_core.messages import HumanMessage, AIMessage

The script imports HumanMessage and AIMessage to facilitate a structured chat. It invokes the model with both a user message and an AI response, emulating a conversation.

1. **Defining Message History with Session IDs**:

python

Copy code

from langchain\_community.chat\_message\_histories import ChatMessageHistory

from langchain\_core.chat\_history import BaseChatMessageHistory

from langchain\_core.runnables.history import RunnableWithMessageHistory

store = {}

def get\_session\_history(session\_id: str) -> BaseChatMessageHistory:

if session\_id not in store:

store[session\_id] = ChatMessageHistory()

return store[session\_id]

* + **Session-based Message History**: This code initializes a dictionary (store) to keep track of chat histories across sessions.
  + **get\_session\_history Function**: Checks if a session ID exists in the store. If it does, it returns the session’s message history. If not, it creates a new session history and saves it in store.

1. **Maintaining Message Context with Runnable History**:

python

Copy code

with\_message\_history = RunnableWithMessageHistory(model, get\_session\_history)

config = {"configurable": {"session\_id": "chat1"}}

* + **RunnableWithMessageHistory**: This component wraps the model and session history, allowing the chatbot to track conversations.
  + **Session Configuration**: config is defined with a session\_id ("chat1") to keep track of which session's history to use.

1. **Invoking the Model with Session-Aware Configuration**:

python

Copy code

response = with\_message\_history.invoke(

[HumanMessage(content="Hi, My name is Krish and I am a Chief AI Engineer")],

config=config

)

print(response.content)

The model is invoked with a HumanMessage, and it responds based on the ongoing session, remembering details from prior messages (like the user's name and role).

1. **Testing Memory Recall**:

python

Copy code

with\_message\_history.invoke(

[HumanMessage(content="What's my name?")],

config=config,

)

By repeating an interaction within the same session\_id, the chatbot recalls that the user’s name is "Krish." Changing the session ID would prevent this recall, as it starts a new conversation without prior context.

**Key Features of the Code**

* **Session Memory**: The chatbot remembers details within a conversation by using unique session IDs.
* **Session Switching**: Changing the session ID resets the conversation, as there’s no shared history.
* **Stateful Chat History**: RunnableWithMessageHistory wraps the model, enabling stateful conversations.

This script illustrates how to create a chatbot that can remember conversational context across user sessions, making it ideal for use cases where continuity is important.

Here’s a step-by-step explanation with some additional detail to clarify how it works:

Code Breakdown

store Initialization:

python

Copy code

store = {}

store is a dictionary that acts as a container for session-based chat histories.

Each entry in store will use a session\_id as the key, and the corresponding value will be an instance of ChatMessageHistory.

get\_session\_history Function:

python

Copy code

def get\_session\_history(session\_id: str) -> BaseChatMessageHistory:

This function takes one parameter, session\_id, which uniquely identifies each chat session.

The function returns a BaseChatMessageHistory object, allowing it to manage the messages associated with the session ID.

Checking and Storing Session History:

python

Copy code

if session\_id not in store:

store[session\_id] = ChatMessageHistory()

Purpose of the Check: if session\_id not in store checks if there’s already a chat history associated with the session\_id.

Creating New History: If there’s no existing chat history for this session\_id, it creates a new ChatMessageHistory() instance.

Storing in store: This new instance is then stored in store with the session\_id as the key.

Returning the Chat History:

python

Copy code

return store[session\_id]

The function then returns the ChatMessageHistory instance (whether newly created or existing) associated with that session.

This allows your chatbot to access or update the specific session's chat history whenever the function is called with a session\_id.

Summary

So, in essence:

store keeps track of all chat histories, each identified by a session\_id.

get\_session\_history retrieves an existing chat history or creates a new one if it doesn’t exist.

ChatMessageHistory() (or its superclass BaseChatMessageHistory) is responsible for managing the actual conversation messages, making it easy to maintain session-specific memory.

This setup allows each user session to have its own history, and it’s stored in store so that the chatbot can recall previous interactions in future conversations.