# Building a Simple LLM-based Chatbot

This document explains step by step how to build a simple **LLM (Large Language Model) based chatbot** using LangChain and Groq. The chatbot will be able to:

* Have a conversation.
* Remember previous interactions (stateful).
* Use prompt templates for customization.
* Manage conversation history.

We’ll also discuss how this foundation helps when moving to more advanced concepts such as **Conversational RAG** and **Agents**.

## 1. Setup

First, install dependencies and set up your environment:

! pip install -r requirements.txt

Load environment variables (e.g., API key):

import os  
from dotenv import load\_dotenv  
  
load\_dotenv()  
groq\_api\_key = os.getenv("Groq\_key")

## 2. Initialize the Model

We use the **ChatGroq** model from LangChain:

from langchain\_openai import ChatOpenAI  
from langchain\_groq import ChatGroq  
  
model = ChatGroq(model="Gemma2-9b-It", groq\_api\_key=groq\_api\_key)

Test it:

from langchain\_core.messages import HumanMessage  
  
model.invoke([HumanMessage(content="Hi, my name is Krishna and I am chief AI Engineer")])

You can also simulate message history manually:

from langchain\_core.messages import AIMessage  
  
model.invoke([  
 HumanMessage(content="Hi, my name is Krishna and I am chief AI Engineer"),  
 AIMessage(content="Hi Krishna, it's nice to meet you! That's a very impressive title.\n\nWhat kind of AI projects are you working on?"),  
 HumanMessage(content="Hey, what is my name and what is my designation")  
])

## 3. Message History (Stateful Chatbot)

A chatbot must remember previous messages. We use ChatMessageHistory and wrap the model with RunnableWithMessageHistory.

from langchain\_community.chat\_message\_histories import ChatMessageHistory  
from langchain\_core.chat\_history import BaseChatMessageHistory  
from langchain\_core.runnables.history import RunnableWithMessageHistory  
  
store = {}  
  
# Function to retrieve session history  
def get\_session\_history(session\_id: str) -> BaseChatMessageHistory:  
 if session\_id not in store:  
 store[session\_id] = ChatMessageHistory()  
 return store[session\_id]  
  
with\_message\_history = RunnableWithMessageHistory(model, get\_session\_history)  
  
config = {"configurable": {"session\_id": "chat1"}}  
  
# First interaction  
response = with\_message\_history.invoke([  
 HumanMessage(content="Hi, my name is Krishna and I am a Python developer")  
], config=config)  
print(response.content)  
  
# Model remembers your name  
response = with\_message\_history.invoke([  
 HumanMessage(content="What is my name?")  
], config=config)  
print(response.content)

If you change session IDs, history won’t be remembered:

config1 = {"configurable": {"session\_id": "chat2"}}  
response = with\_message\_history.invoke([  
 HumanMessage(content="What is my name?")  
], config=config1)  
print(response.content) # Won't remember previous name

## 4. Prompt Templates

Prompt templates format user input before passing it to the LLM. This adds control and customization.

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder  
  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "You are a helpful assistant. Answer all questions to the best of your ability."),  
 MessagesPlaceholder(variable\_name="messages")  
])  
  
chain = prompt | model  
chain.invoke({"messages": [HumanMessage(content="Hi my name is Krishna")]})

With history:

with\_message\_history = RunnableWithMessageHistory(chain, get\_session\_history)  
config = {"configurable": {"session\_id": "chat3"}}  
  
response = with\_message\_history.invoke([  
 HumanMessage(content="Hi, my name is Krishna")  
], config=config)  
print(response.content)

## 5. Multi-language Support with Prompts

We can add a **language variable** to instruct the model:

prompt = ChatPromptTemplate.from\_messages([  
 ("system", "You are a helpful assistant. Answer all questions to the best of your ability in {language}"),  
 MessagesPlaceholder(variable\_name="messages")  
])  
  
chain = prompt | model  
  
response = chain.invoke({  
 "messages": [HumanMessage(content="Hi my name is Krishna")],  
 "language": "Hindi"  
})  
print(response.content)

## 6. Managing Conversation History

If history grows too large, it may exceed the LLM’s context window. We use trim\_messages to keep only the most relevant messages.

from langchain\_core.messages import SystemMessage, trim\_messages  
  
trimmer = trim\_messages(  
 max\_tokens=45,  
 strategy="last",  
 token\_counter=model,  
 include\_system=True,  
 allow\_partial=False,  
 start\_on="human"  
)  
  
messages = [  
 SystemMessage(content="You are a good assistant"),  
 HumanMessage(content="Hi, I am Krishna"),  
 AIMessage(content="Hi!"),  
 HumanMessage(content="I like vanilla ice cream"),  
 AIMessage(content="Nice"),  
 HumanMessage(content="What is 2+2"),  
 AIMessage(content="4"),  
 HumanMessage(content="Thanks"),  
 AIMessage(content="No problem!"),  
 HumanMessage(content="Having fun!"),  
 AIMessage(content="Yes!"),  
]  
  
trimmed = trimmer.invoke(messages)

Now integrate trimming into the pipeline:

from operator import itemgetter  
from langchain\_core.runnables import RunnablePassthrough  
  
chain = (  
 RunnablePassthrough.assign(messages=itemgetter("messages") | trimmer)  
 | prompt  
 | model  
)  
  
response = chain.invoke({  
 "messages": messages + [HumanMessage(content="What ice cream do I like?")],  
 "language": "English"  
})  
print(response.content)

With message history:

with\_message\_history = RunnableWithMessageHistory(  
 chain,  
 get\_session\_history,  
 input\_messages\_key="messages"  
)  
  
config = {"configurable": {"session\_id": "chat4"}}  
  
response = with\_message\_history.invoke({  
 "messages": messages + [HumanMessage(content="What math problem I asked?")],  
 "language": "English"  
}, config=config)  
  
print(response.content)

## 7. Key Takeaways

* **LLM Chatbot Basics:** Start with simple human/AI messages.
* **Stateful Chat:** Use ChatMessageHistory and RunnableWithMessageHistory.
* **Prompt Templates:** Add system instructions and custom variables.
* **Multi-language Responses:** Use placeholders in templates.
* **History Management:** Prevent context overflow with trim\_messages.

This chatbot forms the foundation for **Conversational RAG** (knowledge from external data) and **Agent-based chatbots** (performing actions).