**1st Project 🡪 LANGGRAPH**

**🧠 Purpose**

This notebook creates a chatbot using:

* **LangGraph** to define a stateful conversational flow.
* **LangChain + Groq's LLM (Gemma2-9b-It)** for the actual language model.
* A simple REPL loop for user interaction.

**📦 1. Imports & Setup**

from langchain\_groq import ChatGroq

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

* Instantiates an LLM from Groq using the Gemma2-9b-It model.

from typing import Annotated

from typing\_extensions import TypedDict

from langgraph.graph import StateGraph, START, END

from langgraph.graph.message import add\_messages

* These are required to define the state structure and build the flow using LangGraph.

**🧱 2. Define State and Node Logic**

class State(TypedDict):

messages: Annotated[list, add\_messages]

* Defines the state with a messages field that accumulates chat messages using add\_messages.

def chatbot(state: State):

return {"messages": llm.invoke(state['messages'])}

* A node function: sends the message history to the LLM and gets the assistant's response.

**🔁 3. Build the Graph**

graph\_builder = StateGraph(State)

graph\_builder.add\_node("chatbot", chatbot)

graph\_builder.add\_edge(START, "chatbot")

graph\_builder.add\_edge("chatbot", END)

graph = graph\_builder.compile()

* Creates a LangGraph with a single node: chatbot.
* Connects the flow: START → chatbot → END.

**🖼 4. Visualize the Graph**

display(Image(graph.get\_graph().draw\_mermaid\_png()))

* Tries to display the graph structure as an image (via Mermaid).

**💬 5. Run the Chatbot Loop**

while True:

user\_input = input("User: ")

if user\_input.lower() in ["quit", "q"]:

print("Good Bye")

break

for event in graph.stream({'messages': ("user", user\_input)}):

for value in event.values():

print("Assistant:", value["messages"].content)

* Simple command-line interface:
  + Takes user input.
  + Sends it to the chatbot graph.
  + Streams and prints the assistant's response.

**✅ Summary**

This notebook is a minimal, functioning LangGraph chatbot pipeline that:

1. Sends user messages to a Groq-powered LLM.
2. Manages the stateful flow using LangGraph.
3. Allows interactive chatting via a terminal interface.

**2nd Project 🡪 LANGGRAPH Multiple Nodes**

This code defines a LangGraph application that processes user messages through a sequence of interconnected nodes.

**1. Imports and Setup**

* ChatGroq, TypedDict, Annotated, StateGraph, START, END, add\_messages, Image, display are imported to set up the LangGraph, define the state, and potentially display the graph.
* groq\_api\_key and llm: An API key for Groq is set, and a ChatGroq instance is initialized using the "Gemma2-9b-It" model. This llm object will be used by the chatbot node to generate responses.

**2. State Definition (**State**)**

* class State(TypedDict): This defines the structure of the data that flows through the graph.
  + messages: An Annotated list used to store the conversation history. add\_messages is a special LangGraph helper that ensures new messages are correctly appended to this list.
  + sentiment: A string field to store the sentiment detected from the user's message.

**3. Node Definitions (Functions)**

* Each function represents a "node" in the graph, performing a specific task:
  + preprocess(state: State) -> State: This node takes the current State. It accesses the content of the latest message (state["messages"][-1].content), removes leading/trailing whitespace using .strip(), and updates the message content in the state.
  + analyze\_sentiment(state: State) -> State: This node examines the content of the last message. It sets the sentiment in the state to "positive" if the word "good" is found in the message; otherwise, it sets it to "neutral".
  + chatbot(state: State) -> State: This is where the Large Language Model (LLM) is used. It invokes the llm with the messages from the current state. The LLM generates a response, which is then added to the messages list in the state.
  + logger(state: State) -> State: This node is for logging. It prints the content of the last message (which would be the chatbot's response at this stage) and the detected sentiment to the console.

**4. Graph Construction (**StateGraph**)**

* builder = StateGraph(State): An instance of StateGraph is created, indicating that this graph will manage and modify a state conforming to the State TypedDict.
* builder.add\_node(...): Each of the defined functions (preprocess, analyze\_sentiment, chatbot, logger) is added to the graph as a named node.

**5. Flow Definition (**add\_edge**)**

* This section defines the sequential path through which the data (the State object) will flow:
  + builder.add\_edge(START, "preprocess"): The graph execution begins by sending the initial input to the "preprocess" node.
  + builder.add\_edge("preprocess", "analyze\_sentiment"): After preprocessing, the state moves to the "analyze\_sentiment" node.
  + builder.add\_edge("analyze\_sentiment", "chatbot"): After sentiment analysis, the state proceeds to the "chatbot" node to generate a response.
  + builder.add\_edge("chatbot", "logger"): After the chatbot's response, the state goes to the "logger" node for printing information.
  + builder.add\_edge("logger", END): After logging, the graph execution concludes.

**6. Compilation and Visualization**

* graph = builder.compile(): This step compiles the defined nodes and edges into an executable graph.
* display(Image(graph.get\_graph().draw\_mermaid\_png())): This line attempts to generate and display a visual representation of the graph using Mermaid syntax, helping to illustrate the workflow visually.

**7. Graph Invocation**

* The code then shows how to run the graph:
  + graph.invoke({"messages": ("user", input\_message)}): This initiates the graph's execution with an initial user message. The graph processes this message through the defined nodes (preprocess -> analyze\_sentiment -> chatbot -> logger).
  + The final\_state returned by invoke() contains the complete updated state, including the original message, the chatbot's response, and the detected sentiment.
  + The print statements display the final state and extracted information (chatbot's response and sentiment) for two different input messages.

This code demonstrates a **sequential data processing pipeline** using LangGraph. It is designed for a single flow of control and data, rather than independent agents interacting with each other. Each node performs a specific, predefined operation on the shared state, and the execution proceeds in a fixed order.