📕 Chatbot using LangGraph

1. Playlist Context

- This video is part of **Agentic AI using LangGraph** playlist.
- So far, covered:
  - Fundamentals of LangGraph & Agentic AI basics
  - Types of workflows:
    - Sequential workflows
    - Parallel workflows
    - Conditional workflows
    - Looping workflows
- With these basics, we are now ready to build real-world applications.
- Today's task: **Build a Chatbot using LangGraph**.

---

2. Features of the Chatbot (Planned for the Series)

- **Basic chatting**: LLM-based chatbot that can answer user queries.
- **RAG (Retrieval-Augmented Generation)**: If needed, bot will fetch answers from documents.
- **Tools Integration**: Allow chatbot to perform actions using tools.
- **UI integration**: Add a user interface.
- **LangSmith integration**: For monitoring & debugging.
- **Advanced Concepts** included in later videos:
  - Memory
  - Persistence (state saving)
  - Checkpointers
  - Human-in-the-loop (HITL)
  - Retry logic
  - Fault tolerance

👉 The chatbot is used as a **single project** to cover all advanced LangGraph concepts.

---

3. Today's Focus (Part 1)

- Build a **simple chatbot**:
  - Can chat with the user
  - Remembers **conversation history**
- Later, complexity will be added step by step.

---

4. Chatbot Design

- Chatbot = **workflow** with an LLM.
- Here, workflow is **very simple**:
  - **Sequential flow** with just **one node**.
  - Flow:
    - Start → Chat Node (LLM) → End
- Example:
  - User: "What is the capital of India?"
  - LLM: "New Delhi"
  - Flow ends → result sent back.

---

5. State Definition

- In LangGraph, every workflow needs a **State**.
- For chatbot:

    - State = **Conversation history**.
- Implementation:
    - Messages stored in a **list** inside the state.
    - Types of messages:
        - **HumanMessage** (user input)
        - **AIMessage** (LLM response)
        - **SystemMessage** (instructions to LLM)
        - **ToolMessage** (tool outputs)
    - All inherit from **BaseMessage**.
    - State = List[BaseMessage].

---

6. Reducer Function

- Problem:
    - By default, LangGraph **replaces old state with new state**.
    - Example:
        - User: "What is the capital of India?" → stored
        - AI: "New Delhi" → replaces previous
        - So history is lost.
- Solution:
    - Use a **reducer function** to append messages.
    - Instead of operator.add, LangGraph provides **add_messages**.
    - Optimized for working with BaseMessage.

---

7. Building the Graph

1. Create a **StateGraph** using ChatState.
2. Add one node → chat_node.
3. Define function chat_node(state):
    - Extracts messages from state
    - Sends them to LLM
    - Gets AI response
    - Stores back in state as AIMessage
4. Add edges:
    - START → chat_node
    - chat_node → END
5. Compile graph → becomes chatbot workflow.

---

8. First Test (Single Message)

- Initial state contains:
    - One HumanMessage: "What is the capital of India?"
- Invoke chatbot.
- Returns:
    - HumanMessage
    - AIMessage: "The capital of India is New Delhi."
- Shows **basic working chatbot**.

---

9. Adding Loop for Continuous Chat

- Problem: Current version ends after one question.
- Solution: Wrap in a **while loop**:

    - Loop keeps asking user input.
    - If user types exit, quit, or bye, loop breaks.
    - Otherwise:
        - HumanMessage added
        - Workflow invoked
        - AI response shown
- This gives **real chatbot feel** (though console-based).

---

10. Major Problem: No Memory

- Issue:
    - Bot forgets previous conversation.
    - Example:
        - User: “Hi, my name is Nitesh”
        - AI: “Hello Nitesh!”
        - User: “What is my name?”
        - AI: “Sorry, I don’t know.”
- Why?
    - Each invoke() call **resets the state**.
    - Previous messages are erased once flow ends.

---

11. Solution: Persistence

- Fix using **LangGraph Persistence**:
    - Store state in **RAM or database** after execution.
    - Next invocation → fetch old state + add new messages.
- Implementation:
    - Import MemorySaver from langgraph.checkpoint.memory.
    - Define checkpointer = MemorySaver().
    - Pass checkpointer when compiling graph.
- Thread IDs:
    - Each user conversation = one thread.
    - Thread ID uniquely identifies conversation.
    - Allows multiple users (Nitesh, Rahul, etc.) to chat simultaneously.
- While invoking chatbot:
    - Provide both messages + config (with thread_id).

---

12. Persistence in Action

- Example:
    - User: “Hi, my name is Nitesh.”
    - AI: “Hello Nitesh!”
    - User: “What is my name?”
    - AI: “Your name is Nitesh.”
- Now memory works because:
    - Old state was fetched from RAM.
    - New message appended using add_messages.
- Limitation:
    - If program restarts, RAM-based state is lost.
    - In production → store state in **database** for durability.

---

13. Key Takeaways

- ✅ Chatbot = Sequential workflow with one node (LLM).
- ✅ State stores **conversation history** as messages.
- ✅ Reducer function ensures **history is not overwritten**.
- ✅ Problem solved using **Persistence**:
  - MemorySaver keeps state in RAM.
  - Database persistence for real-world production.
- ✅ Thread IDs allow **multi-user conversations**.

🔑 Code Explanation (Without Code)

1. **State Definition**:
   - ChatState with attribute messages: List[BaseMessage].
   - Uses add_messages reducer for appending.
2. **Graph Creation**:
   - One node: chat_node.
   - Extracts messages → sends to LLM → returns response.
3. **Workflow**:
   - Sequential: START → chat_node → END.
4. **Invocation**:
   - Initial state: HumanMessage.
   - Output: AIMessage.
5. **Looping**:
   - While loop runs until user types exit/quit/bye.
   - Each input → added as HumanMessage.
   - Workflow invoked → AI reply printed.
6. **Persistence**:
   - Add MemorySaver checkpointer.
   - Use thread_id in config for each user.
   - Ensures previous conversation is preserved.