Lesson 2: Langchain, RAG & Vector databases

Introduction

- **Problem:** LLMs (like GPT) forget context, can't handle huge documents, and sometimes make up answers (hallucination).
- **Solution:**
    - **LangChain** → helps us connect LLMs with external data/tools.
    - **Vector Databases** → store and search large text/data efficiently.
    - **RAG (Retrieval-Augmented Generation)** → fetch the right info before generating answers.

👉 Think of it like **Google Search + ChatGPT combined**.

What is LangChain?

- **Definition:** A framework that helps developers build apps with LLMs by chaining together different components (data, tools, prompts, memory).
- **Analogy:** Like **Express.js** in Node → it doesn't do everything itself, but helps you connect routes, middleware, and databases.
- **Use cases:** Chatbots, document Q&A, agents, workflows.

Example (simple Q&A app with LangChain + OpenAI)

```
import { ChatOpenAI } from "@langchain/openai";
import { PromptTemplate } from "@langchain/core/prompts";
import { LLMChain } from "langchain/chains";
import dotenv from "dotenv";

dotenv.config();

const llm = new ChatOpenAI({
  model: "gpt-3.5-turbo",
  apiKey: process.env.OPENAI_API_KEY,
});
```

```
const prompt = PromptTemplate.fromTemplate(
  "Explain {topic} in simple words."
);

const chain = new LLMChain({ llm, prompt });

const result = await chain.run({ topic: "JavaScript closures" });
console.log(result);
```

👉 Output: A closure is like a backpack where a function keeps variables it needs, even after it's moved somewhere else.

What is a Vector Database?

- **Problem:** Normal databases (SQL/Mongo) are good for structured data, but not for searching meaning in text/images.
- **Vector DBs:** Store text as **embeddings** (mathematical vectors).
- **How it works:**
    1. Convert text → vector (e.g., [0.12, 0.98, ...]) using an embedding model.
    2. Store vectors in a **vector DB** (like Pinecone, Weaviate, Milvus, or even FAISS locally).
    3. When user asks something, convert query → vector → find "closest" vectors (semantic similarity).
- **Analogy:** Like Spotify "Find similar songs" → instead of title match, it matches by *sound similarity*.

Example (storing and searching)

```
import { OpenAIEmbeddings } from "@langchain/openai";
import { FaissStore } from "@langchain/community/vectorstores/faiss";
import { Document } from "langchain/document";
import dotenv from "dotenv";
```

```
dotenv.config();

const embeddings = new OpenAIEmbeddings({ apiKey: process.env.OPENAI_API_KEY });

const docs = [
  new Document({ pageContent: "React is a frontend library" }),
  new Document({ pageContent: "Node.js runs on server" }),
  new Document({ pageContent: "MongoDB stores JSON" }),
];

// Create FAISS in-memory DB
const db = await FaissStore.fromDocuments(docs, embeddings);

// Query
const query = "Which tech stores data?";
const results = await db.similaritySearch(query, 1);

console.log("Answer:", results[0].pageContent);
```

👉 Output: "MongoDB stores JSON"

Application Implementation: ChainRAG BOT

1. Project Setup

```
# Backend
mkdir chatbot-backend && cd chatbot-backend
npm init -y
npm install express cors dotenv langchain @langchain/google-genai

# Frontend
npx create-react-app chatbot-frontend
```

```
cd chatbot-frontend
npm install
```

2. Backend Code (Node + Express + LangChain)

```
//chatbot-backend/server.js
import express from "express";
import cors from "cors";
import dotenv from "dotenv";

import { ChatGoogleGenerativeAI } from "@langchain/google-genai";
import { ChatPromptTemplate } from "@langchain/core/prompts";
import { RunnableWithMessageHistory } from "@langchain/core/runnables";
import { InMemoryChatMessageHistory } from "@langchain/core/chat_history";

dotenv.config();

const app = express();
app.use(cors());
app.use(express.json());

const PORT = 5000;

//routes/genai.js
const messageHistories = new Map();

// Create Gemini Chat model
const model = new ChatGoogleGenerativeAI({
  model: "gemini-1.5-pro",
  apiKey: process.env.GOOGLE_API_KEY,
```

```
});

// System + Human prompt
const prompt = ChatPromptTemplate.fromMessages([
  ["system", "You are a helpful chatbot that remembers past conversations."],
  ["placeholder", "{history}"],
  ["human", "{input}"],
]);

// Chain with model + prompt
const chain = prompt.pipe(model);

// Wrap chain with memory
const chainWithHistory = new RunnableWithMessageHistory({
  runnable: chain,
  getMessageHistory: async (sessionId) => {
    if (!messageHistories.has(sessionId)) {
      messageHistories.set(sessionId, new InMemoryChatMessageHistory());
    }
    return messageHistories.get(sessionId);
  },
  inputMessagesKey: "input",
  historyMessagesKey: "history",
});

app.post("/api/chat", async (req, res) => {
  const { message, sessionId } = req.body;
```

```
 try {
   const response = await chainWithHistory.invoke(
     { input: message },
     { configurable: { sessionId } }
   );

   res.json({ reply: response.content });
 } catch (err) {
   console.error(err);
   res.status(500).json({ error: "Something went wrong" });
 }
});
//chatbot-backend/server.js
import express from "express";
import cors from "cors";
import dotenv from "dotenv";

import { ChatGoogleGenerativeAI } from "@langchain/google-genai";
import { ChatPromptTemplate } from "@langchain/core/prompts";
import { RunnableWithMessageHistory } from "@langchain/core/runnables";
import { InMemoryChatMessageHistory } from "@langchain/core/chat_history";

dotenv.config();

const app = express();
app.use(cors());
app.use(express.json());
```

```
const PORT = 5000;

app.listen(PORT, () => console.log(`🚀 Server running on port ${PORT}`));
```

3. Frontend Code (ReactJS)

```
//chatbot-frontend/src/App.js
import React, { useState } from "react";
import { v4 as uuidv4 } from "uuid";

function App() {
  const [messages, setMessages] = useState([]);
  const [input, setInput] = useState("");
  const [sessionId] = useState(uuidv4()); // unique per chat session

  const sendMessage = async () => {
    if (!input.trim()) return;

    // Add user message
    setMessages([...messages, { sender: "user", text: input }]);

    const res = await fetch("<http://localhost:5000/api/chat>", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ message: input, sessionId }),
    });

    const data = await res.json();
    setMessages((prev) => [...prev, { sender: "bot", text: data.reply }]);
    setInput("");
```

```
 };

 return (
   <div style={{ padding: "20px", fontFamily: "Arial" }}>
     <h2>🤖 Gemini Chatbot with Memory</h2>

     <div style={{
       border: "1px solid #ccc", padding: "10px", height: "400px", overflowY: "scroll", marginBottom: "10px"
     }}>
       {messages.map((msg, idx) => (
         <div key={idx} style={{ margin: "5px 0" }}>
           <b>{msg.sender === "user" ? "□ You" : "🤖 Bot"}:</b> {msg.text}
         </div>
       ))}
     </div>

     <inputvalue={input}
       onChange={(e) => setInput(e.target.value)}
       style={{ width: "70%", padding: "10px" }}
       placeholder="Type a message..."
     />
     <button onClick={sendMessage} style={{ padding: "10px 20px", marginLeft: "10px" }}>
       Send
     </button>
   </div>
 );
}
```

export default App;

What is RAG (Retrieval-Augmented Generation)?

- **Definition:** A technique where the LLM **retrieves data first** from a vector DB (or external source) before generating an answer.
- **Analogy:** Imagine an **open-book exam**: Instead of memorizing everything, you quickly find the right page and then explain.
- **Why:** Prevents hallucination, keeps answers **grounded in real data**.

Before we dive into code, let's understand the **why**. Large Language Models (LLMs) like ChatGPT are powerful, but they have limitations

- They are trained on fixed data (knowledge cutoff).
- They may hallucinate facts.
- They don't know company-specific/private data.

RAG (Retrieval-Augmented Generation) solves this by combining an LLM with an external knowledge base. It retrieves relevant documents and augments the LLM's input with that context, producing grounded, accurate responses."

Flow of RAG

1. User asks a question → "What are MERN stack components?"
2. Convert question → embedding.
3. Search in Vector DB (retrieves relevant docs about Mongo, Express, React, Node).
4. Pass both **question + retrieved docs** into LLM.
5. LLM gives an accurate, data-backed answer.

Real-World Use Cases

- **Chatbot for company docs** → Employees ask "What's our leave policy?" → Bot searches HR docs in vector DB → Gives accurate answer.
- **Customer support** → Bot searches FAQs before replying.
- **Search engine + chat** → Like Perplexity.ai.
- **Code assistants** → Search repo codebase → answer dev questions.

Summary

- **LangChain** = Framework to connect LLM with tools/data.
- **Vector DB** = Stores embeddings for semantic search.
- **RAG** = Retrieves info + generates accurate answer.

👉 Together, they help us build **smart, data-aware AI apps**.