Conversational RAG Q&A with PDF Upload and Chat History

1. Project Description

Build a **Conversational RAG Q&A** web app in **Streamlit** where users upload one or more PDFs, chat about their content, and receive **context-aware** answers grounded in those documents. The app uses a **vector store** for retrieval, **LangChain** for orchestration, and **chat history** to reformulate follow-up questions.

Key features:

- **PDF Upload & Ingestion:** Users upload PDFs; the app parses, chunks, embeds, and stores text in a vector DB (e.g., Chroma).
- **Session & History:** Users provide a **Session ID**; each session maintains a **chat history** and uses it to refine queries.
- **Conversational RAG:** Questions are **contextualized** using history, then answered using retrieved chunks.
- **API Key Handling:** User provides LLM provider key (e.g., Groq) securely at runtime (no plaintext persistence).
- **UI:** Streamlit interface for uploading PDFs, setting session, entering questions, and viewing both answers and visible chat history.

---

2. Tasks to Complete the Project

1. **Environment & Dependencies**
    - Create a virtual environment and install Streamlit, LangChain, Chroma, PyPDFLoader, HuggingFaceEmbeddings, and your chosen LLM provider (e.g., `langchain_groq`).
2. **PDF Ingestion Pipeline**
    - Load PDFs → split into chunks (RecursiveCharacterTextSplitter) → embed (HuggingFaceEmbeddings) → store in Chroma.
3. **Vector Store Setup**
    - Initialize/persist a Chroma collection per project (e.g., `persist_directory=".chroma/student-rag"`).
4. **RAG Core (LangChain)**
    - Build a **contextualize-question** step that uses chat history to rewrite follow-ups.
    - Build a retriever (from Chroma) and an answer chain that grounds responses in retrieved text.
5. **Chat History & Session Memory**

- Maintain per-session memory keyed by **Session ID** (LangChain `ChatMessageHistory` + RunnableWithMessageHistory or manual state in Streamlit).

6. **Streamlit UI**

   - Inputs: API Key (secure), Session ID, PDF uploader, chat input.
   - Views: answer stream, retrieved sources (optional), visible chat transcript.

7. **Security & Robustness**

   - Hold API keys in `st.session_state`, never write to disk.
   - Limit file size/type, handle ingestion failures, sanitize text.

8. **Testing & Debugging**

   - Unit test: chunking, embedding, retrieval, and history-aware reformulation.
   - Manual test: multi-PDF sessions, cold start vs. follow-ups, empty store behavior.

9. **(BONUS) Enhancements**

   - Citations with source pages, MMR tuning, streaming tokens, evaluation with RAGAS, reranking.

---

3. Guidance to Make it OOP (and swap the trivial chatbot with RAG)

You already encapsulated logic in **Project 1** (e.g., `Chatbot`, `Database`, `Student`). Reuse that structure and **swap the engine** via the **Strategy pattern**:

1) Define a stable interface for chat engines

```
from typing import Protocol, List, Tuple

class ChatEngine(Protocol):
    def answer(self, session_id: str, question: str) -> str: ...
    def get_history(self, session_id: str) -> List[Tuple[str, str]]: ...
```

2) Keep your existing Chatbot façade, depend on the interface

```
class Chatbot:
    def __init__(self, engine: ChatEngine):
        self.engine = engine

    def ask(self, session_id: str, question: str) -> str:
        return self.engine.answer(session_id, question)

    def history(self, session_id: str):
        return self.engine.get_history(session_id)
```

3) Your old trivial bot becomes one implementation

```
class SimpleFAQEngine(ChatEngine):
    def __init__(self, faq_map: dict[str, str]):
        self.faq_map = faq_map
        self.histories = {}

    def answer(self, session_id: str, question: str) -> str:
        ans = self.faq_map.get(question.lower(), "Sorry, I don't know.")
        self.histories.setdefault(session_id, []).append(("user", question))
        self.histories[session_id].append(("assistant", ans))
        return ans

    def get_history(self, session_id: str):
        return self.histories.get(session_id, [])
```

4) Add a RAG engine that uses your LangChain pipeline

```
class RagEngine(ChatEngine):
    def __init__(self, rag_chain_with_history, history_store):
        self.chain = rag_chain_with_history
        self.history_store = history_store  # maps session_id -> ChatMessageHistory

    def answer(self, session_id: str, question: str) -> str:
        resp = self.chain.invoke(
            {"question": question, "history": []},
            config={"configurable": {"session_id": session_id}},
        )
        return resp

    def get_history(self, session_id: str):
        h = self.history_store.get(session_id)
        if not h: return []
        return [("user", m.content) if m.type == "human" else ("assistant",
m.content) for m in h.messages]
```

5) Wire it in your Streamlit app

- Where you previously did `Chatbot(SimpleFAQEngine(...))`, now do:
- `chatbot = Chatbot(RagEngine(with_history, store))`
- UI remains unchanged—only the **engine** changed. This cleanly upgrades the app from "trivial replies" to **RAG** without touching the UI layer.

Tips

- Keep **config** (model name, chunk sizes, k) in a small `Config` class or `.yaml` to switch providers easily.
- Use **dependency inversion**: UI imports the `Chatbot` façade, not LangChain internals.
- Add an **adapter** for different vector stores (e.g., Chroma now, Pinecone later) behind a `VectorStoreClient` interface.

---

4. Deliverables

1. **Recorded Demo of the Running Project**

   - Show uploading multiple PDFs.
   - Show entering **API key** and **Session ID**.
   - Demonstrate first question vs. follow-up question that resolves via **history-aware reformulation**.
   - Display answers and the visible chat transcript.
   - (Optional) Show retrieved snippets/citations.

2. **ZIP Folder with Source Code**

   - `app.py` (Streamlit UI + wiring)
   - `engines/`
     - `base.py` (ChatEngine protocol)
     - `simple_faq.py` (legacy engine from Project 1)
     - `rag_engine.py` (new engine)
   - `rag/`
     - `ingest.py` (PDF → chunks → embeddings → Chroma)
     - `pipeline.py` (prompts, LCEL chain, history wiring)
   - `requirements.txt`
   - `README.md` with setup, env vars, and run instructions
   - (Optional) `config.yaml` to tweak chunking/LLM/k

Final Guideline (bridging Project 1 → Project 2)

- **Keep your UI and Chatbot façade** from Project 1 intact.
- **Replace only the engine**:
  - Implement `RagEngine` to satisfy the same `ChatEngine` interface.
  - Inject `RagEngine` into `Chatbot` instead of `SimpleFAQEngine`.
- **Reuse your auth/session scaffolding**:
  - Session handling from Project 1 maps directly to **Session ID** for RAG history.
- **Extend safely**:
  - Add `VectorStoreClient` and `LLMClient` adapters to swap providers without UI changes.
  - Promote tunables (chunk size, overlap, `k`, model) to config for easy experimentation.

Thank You
Edges For Training Team