Chatbot — Explanation and Reasoning

1. Project overview

This project implements a small document ingestion and chat pipeline built with FastAPI, LangChain helper utilities and Qdrant as the vector store. Main goals:

- Accept files (PDFs), extract text and tables.
- Turn extracted text (and table content) into documents, chunk them, and store embeddings in Qdrant.
- Provide endpoints for creating/deleting collections and for asking queries (chat endpoint) that retrieve from the vector store.

Main Features:

- router (FastAPI endpoints)
- DocumentTextExtractor (PDF text + table extraction using PyPDFLoader and camelot)
- QdrantVectorStoreDB (wrapper around LangChain's Qdrant integration)
- upload_document_* helpers (wrapping QdrantVectorStore.from_documents)
- AnswerQuery service (to perform retrieval + answer generation)

2. High-level request flows (what happens when a client calls an endpoint)

2.1 POST /documents/extract (files -> extracted text)

- Client uploads one or more files via multipart/form-data (field name files).
- FastAPI handler creates a temporary directory using tempfile.TemporaryDirectory() to store each uploaded file, generating a unique filename using uuid to avoid collisions.
- Each file is written to disk, then handed to DocumentTextExtractor.extract_text().
- The extractor returns a dict containing text (raw concatenated pages) and tables (list of table structures, currently providing Markdown strings). Those are added to the results list returned to the client.

Why this flow?

- Writing uploaded bytes to a temporary file simplifies compatibility with many PDF libraries that expect a filesystem path.

- Using a temp dir ensures uploaded files are cleaned up automatically.

2.2 POST /upload_documents?collection_name=... (ingestion into Qdrant)

1. The endpoint expects a JSON body (UploadDocumentSchema) that contains a results list where each item represents a file's extraction result (file_name, text, tables).
2. For every drs result the endpoint:
    a. Adds a base document with file_name and text.
    b. Adds each detected table as a separate document with file_name_table_{i} and the table text (chooses markdown or csv or json if present).
3. The list of documents (each: {"file_name": ..., "text": ...}) is passed to vectorstore.upload_documents(...).
4. QdrantVectorStoreDB.upload_documents:
    a. Converts each dict into a LangChain Document with page_content and metadata.source set to the file name.
    b. Splits documents with RecursiveCharacterTextSplitter(chunk_size=1500, chunk_overlap=150).
    c. Calls upload_document_existing_collection(...), which in turn uses QdrantVectorStore.from_documents(...) to push embeddings to Qdrant.

2.3 POST /create_collection & DELETE /delete_collection/{collection_name}

- create_collection currently calls upload_document_new_collection which invokes QdrantVectorStore.from_documents(documents=[], ... force_recreate=True) to create an empty collection.
- delete_collection calls qdrant_client.delete_collection(...) to remove a collection.

Note: Creating an empty collection by calling from_documents([]) is a pragmatic approach but there are better, explicit APIs to create an empty collection or to initialize schema. See suggested improvements.

2.4 POST /chat (querying)

- Calls AnswerQuery.in_memory_answer_query(query, session_id, collection_name) which is expected to create/obtain a QdrantVectorStore from the collection and perform retrieval and LLM-based answer generation.

3. Implementation decisions & reasoning

3.1 Temporary files

Using tempfile.TemporaryDirectory() keeps lifecycle automatic.

Unique filenames via uuid prevents collisions and helps debugging.

3.2 Table extraction with Camelot (lattice + stream)

Two flavors are tried because PDFs vary: lattice works when tables have explicit borders; stream works when tables rely on whitespace.

Extracted pandas.DataFrame is converted to Markdown using df.to_markdown() to preserve the 2D structure and make it suitable for llm.

3.3 Text splitting strategy

- RecursiveCharacterTextSplitter with chunk_size=1500 and chunk_overlap=150 is chosen to balance providing enough context per chunk while keeping chunks small enough for embedding/LLM token limits.
- Overlap keeps some context between chunks which helps with continuity in retrieval and answering.

3.4 Metadata

Each split Document stores metadata.source with file_name (or file_name_table_#), enabling trace-back to the original file/table for provenance and debugging.

3.5 Embeddings & Vector DB

- GoogleGenerativeAIEmbeddings used with models/text-embedding-004. API key is loaded from settings (env var recommended).
- Qdrant acts as the vector store since it provides fast similarity search and is well-integrated via LangChain.