1st i create the structure backbone (or atleast i hope soo)

- backend/ (folder): This is the root directory for the backend application.
- main.py (.py file): The App entry point. This file typically initializes the FastAPI application, includes the necessary routes, and might contain application-level middleware or event handlers.
- config.py (.py file): Configuration. This file holds settings, environment variables, or other configurable constants for the application (e.g., database credentials, API keys, etc.).
- api/ (folder): Dedicated to API endpoints and routing.
- routes.py (.py file): Defines the specific HTTP routes (paths) for the API, mapping them to functions that handle incoming requests.
- services/ (folder): Contains the business logic and interactions with external resources or data manipulation. This follows the Service Layer pattern.
- vector_service.py (.py file): Implements the logic for a Vector store (e.g., handling vector database operations, embedding generation, or similarity search).
- chat_service.py (.py file): Contains the core Chat logic or business rules related to the application's chat functionality.
- models/ (folder): Used for defining the data structures.
- schemas.py (.py file): Defines Data models using tools like Pydantic (common with FastAPI) for request/response validation and database schema definition.
- utils/ (folder): Holds general-purpose, reusable code that doesn't fit into the other categories.
- helpers.py (.py file): Contains various Utilities or helper functions that can be imported and used across different parts of the application.

2nd in config py

- **Purpose / goal:** Manage environment variables and configuration using Pydantic and dotenv for clean and secure parameter handling.
- **Dependencies / libs:**
  - pathlib – handle filesystem paths cleanly.
  - pydantic-settings – structured, validated environment configs.
  - typing.Optional – type hint for optional fields.
  - os – access environment variables.
  - python-dotenv – load variables from `.env` file.

2.5 Settings Class Structure (config.py)

- Purpose / goal: Create a unified configuration model for the RAG app, defining all major constants, API keys, and runtime parameters.
- Main sections:
  - API Configuration: Basic app metadata and host settings.
  - CORS: Define allowed origins for cross-origin requests.
  - API Keys: Securely load sensitive tokens like GROQ_API_KEY and HF_TOKEN.
  - Model Config: Set LLM name, temperature, max tokens, and embedding model, OLLAMA_BASE.
  - RAG Config: Chunk size, overlap, and retrieval parameters.
  - File Storage: Temporary directories, file size limits, and allowed file types , VECTORS_DIR

3RD Main Application Setup (`main.py`)

- Import FastAPI, logging, middleware, static files, etc.
- Configure logging format and level
- Create `FastAPI` app using values from `settings`
- Enable **CORS** (Cross-Origin Resource Sharing)
  - Allows requests from front-end or external domains
- Include API router from `routes.py`
- Mount frontend static files (HTML, CSS, JS)

- Create `/` route to serve main HTML interface
- Define `startup` and `shutdown` events:
  - Log startup info
  - Test Ollama connection
  - Warn if embedding model missing
- Add Uvicorn server launcher for direct runs

4TH (schemas.py)

- Purpose / goal: Define request/response models for the API endpoints, enforce validation, and power automatic OpenAPI docs.
- Steps to implement in this file:
  - Import `BaseModel`, `Field`, `validator` from Pydantic and typing helpers.
  - Define request models (e.g., `ChatRequest`) with strict validators for required formats, prompt
  - Define response models
    - `ChatResponse (sources,answer)`
    - `UploadResponse` (message, filename, pages, chunks , sessionID)
    - `SessionInfo (,session_id,pdf_info,message_count,has_vectorstore )`
    - `HealthResponse (ollama_status,embedding_model,llm_model)`
    - `ErrorResponse(error_type,  detail ).`
  - Attach field-level metadata for OpenAPI (descriptions, min/max lengths).
  - Implement `@validator` functions for non-trivial checks (e.g., session id format, no blank prompts).
  - Add unit tests for schema validation edge cases (empty prompt, malformed session id, negative counts).

5th Routes.py

Purpose:

Defines and manages all API routes for the RAG Chatbot backend — connecting the frontend to the underlying services like `chat_service` and `vector_service`.
Basically, this file:

- Handles incoming HTTP requests.
- Validates data (PDFs, session IDs, etc.).
- Calls the right service to do the work.
- Sends structured responses back to the client.

Main Steps / Implementation Points

1. **Import Dependencies**
   - Bring in FastAPI modules (`APIRouter`, `UploadFile`, `HTTPException`).
   - Import services (`chat_service`, `vector_service`), schemas, helpers, and settings.
   - Set up logging and define the router.
2. **Initialize Router** (logger , router)
3. **Health Check Endpoint**
   - `GET /health`
   - Confirms the API and Ollama connection are working.
   - Returns embedding and model info, plus active sessions.

4. PDF Upload Endpoint

- `POST /upload`
- Accepts a PDF + `session_id` form data.
- Validates session and file.
- Saves, processes (via `vector_service`), and cleans up temporary files.
- Returns pages, chunks, and confirmation message.

5. Chat Endpoint

- `POST /chat`
- Accepts user prompts tied to a session.
- Validates that a PDF vectorstore exists.
- Calls `chat_service.chat()` to generate responses.
- Returns the AI answer + source context.

6. Session Info Endpoint

- `GET /sessions/{session_id}/info`
- Returns metadata about a specific session (PDF info, message count, etc.).

7. Delete Session Endpoint

- `DELETE /sessions/{session_id}`
- Removes all data related to a session from both chat and vector stores.

8. Clear Session History Endpoint

- `POST /sessions/{session_id}/clear-history`
- Clears the chat log but **keeps** the vectorstore (so the uploaded PDF stays).

9. List Sessions Endpoint

- `GET /sessions`
- Lists all active sessions stored in the system.

6TH (vector_service.py)

Purpose

This file's entire reason to exist:
Take PDFs → split text → embed → store → retrieve.
It's the memory layer of your RAG system.

Steps to implement in this file:

1. Imports & Setup

- You bring in LangChain tools for loading PDFs, creating embeddings, and saving FAISS vector stores.
- Logging and typing keep things clean.
- `settings` comes from `config.py`, so you'll need that working first.

  Start your rebuild by declaring imports and initializing logging.
  If you mess up this section, nothing else will even import cleanly.

2. Class Definition → `VectorStoreService`

This class is your *engine room*. It stores:

- **Embeddings model** (Ollama or any other)
- **Cache** (for in-memory vectorstores)
- **Metadata** (like how many pages or chunks were processed)
- When you rebuild, start small—just create the constructor and get the embeddings working.
  - **Constructor:**

    ```
    def __init__(self):
        self.embeddings = OllamaEmbeddings(
            model=settings.EMBEDDING_MODEL,
            base_url=settings.OLLAMA_BASE_URL
        )
        self.vectorstore_cache = {}
        self.session_metadata = {}
    ```

    You can add logging later once it's alive.

3. Core Function → `process_pdf()`

- This is the heavy lifter. It:
  1. Loads the PDF
  2. Splits it into chunks
  3. Creates embeddings and FAISS index
  4. Saves it to disk and cache

If you're rebuilding logically:

- Get **PyPDFLoader** working first.
- Then try **splitting text** with `RecursiveCharacterTextSplitter`.
- Once that's fine, create embeddings → store them in FAISS.

  This is where 90% of the bugs hide (missing text, empty splits, embedding model errors).

Structure of this function:

```
Load PDF → Split → Embed → Save → Cache → Return stats
```

4. Retrieval Function → `get_vectorstore()`

When someone queries your chatbot, this function ensures you've got the right memory loaded

Flow:

1. Check memory cache (fast).
2. If not, load from disk.
3. Cache it again for next time.

This is your "lazy loader." Build it after `process_pdf()` works, because it depends on saved vectorstores.

5. Helper Functions

All of these are support players:

- **`has_vectorstore()`** → Boolean check
- **`delete_vectorstore()`** → Clean up session (from cache + disk)
- **`get_metadata()`** → Returns stored info like page/chunk count
- **`get_active_sessions()`** → Returns number of loaded sessions

You can safely implement these *after* your main pipeline (`process_pdf` + `get_vectorstore`) is functional.

6. Global Instance

At the end:

```
vector_service = VectorStoreService()
```

That's just a singleton-style shortcut—so anywhere else (like `chat_service` or `routes`) can import and use it immediately.
Rebuild it last.

Rebuild Order (smartest workflow)

1. Imports + logging
2. `__init__()` → get embeddings running
3. `process_pdf()` → the main transformation logic
4. `get_vectorstore()` → retrieval mechanism
5. Helper functions (delete/check/metadata)
6. Global instance

7th (vector_service.py)

Main Purpose

This `VectorStoreService` handles **PDF ingestion, text chunking, embedding, and retrieval** for a retrieval-augmented generation (RAG) pipeline.
Basically, it:

- Reads PDF files
- Splits them into smaller text chunks
- Converts those chunks into **vector embeddings**
- Stores the vectors in a **FAISS database** for later retrieval in chatbot or search queries
- Manages these vector stores per **session ID**

So it's the memory and search engine behind your RAG app.

Steps to Implement It

1. **Setup Dependencies & Config**
   - Install and import `langchain_community`, `FAISS`, `PyPDFLoader`, `OllamaEmbeddings`, and config your model, base URL, and paths in `settings`.

2. **Initialize the Service**
   - Create an instance of `VectorStoreService()`.
   - Load the embedding model from Ollama (`settings.EMBEDDING_MODEL`).
   - Prepare in-memory caches for `vectorstore_cache` and `session_metadata`.

3. **Process a PDF**

   - Load the PDF using `PyPDFLoader`.
   - Split the content into chunks using `RecursiveCharacterTextSplitter`.
   - Create embeddings for those chunks.
   - Build a FAISS vector store and save it locally (in `VECTORS_DIR/session_id`).
   - Store metadata (pages, chunks) in memory.

4. **Retrieve a Vector Store**

   - Check if the store exists in memory cache.
   - If not, load it from disk with `FAISS.load_local()`.
   - Raise an error if missing or corrupted.

5. **Check / Manage Sessions**

   - `has_vectorstore(session_id)` → check existence
   - `get_metadata(session_id)` → get stored info (pages, chunks)
   - `get_active_sessions()` → count cached sessions
   - `delete_vectorstore(session_id)` → clean from cache + disk
6. **Global Instance**

   - Create a single shared `vector_service` object to be reused across the app.

8th chat_service.py

Main Purpose

This `ChatService` runs the **RAG (Retrieval-Augmented Generation)** logic for chat sessions.
In plain terms:

- It takes user input (`prompt`)
- Pulls the most relevant info from your previously uploaded PDFs (via FAISS vector store)
- Feeds that info + chat history into an LLM (Groq API)
- Returns a contextual, memory-aware answer

It's basically the **conversation engine** that sits on top of your PDF knowledge base.

Steps to Implement It

1. **Setup the LLM**
   - Initialize `ChatGroq` using your `GROQ_API_KEY`, model name, temperature, and token limits from `settings`.
   - This acts as the reasoning brain for your RAG pipeline.

2. **Session History Management**
   - Each `session_id` has its own `ChatMessageHistory` instance to keep track of previous messages.
   - `get_session_history()` retrieves or creates a history object for that session.
   - It allows the model to answer based on conversation context (memory).

3. **RAG Chat Pipeline**
   - **Retrieve the vector store:**
     Load the FAISS store for that session using `vector_service.get_vectorstore()`.
     (If none exists, the session can't chat — no knowledge base, no brains.)
   - **Build the retriever:**
     Convert the vector store into a retriever that fetches top-k relevant chunks based on similarity.
   - **Rephrase user queries with context:**
     Uses `create_history_aware_retriever()` + a prompt that rewrites user questions into standalone, self-contained ones (so the model doesn't get lost in pronouns like "that" or "he said earlier").
   - **Answer generation chain:**
     Builds a `qa_prompt` instructing the model on how to use the retrieved context (be concise, structured, and cite pages).
   - **Combine into a retrieval chain:**
     The retriever fetches context → the document chain generates the response.
   - **Attach memory to it:**
     Wraps the entire RAG pipeline with `RunnableWithMessageHistory`, binding it to the session's chat history so the system remembers past turns.

4. **Generate and Return Output**
    - Invoke the pipeline with the user’s prompt.
    - The result contains:
        - `answer`: the model’s reply
        - `context`: the documents retrieved
    - Extract and format document sources (e.g., “Page 4, Page 12”).

5. **Session Controls**
    - `clear_history(session_id)`: reset a conversation’s memory.
    - `delete_session(session_id)`: remove the entire chat session.
    - `get_message_count(session_id)`: count how many messages have been exchanged.
    - `session_exists(session_id)`: check if a session is active.

6. **Global Instance**
    - `chat_service = ChatService()` gives the whole backend a shared access point to this functionality.