Project Guide: Building Your Local RAG System with FastAPI

This guide provides a project directory structure and a step-by-step plan to build your RAG application. It assumes you will write the code, using the architecture from our previous discussion as a reference.

1. Project Directory Structure

Organizing your project from the start is key. Here is a standard structure for a FastAPI application that separates concerns cleanly.

```
rag-fastapi-project/
├── app/
│   ├── __init__.py
│   ├── main.py           # Your main FastAPI app definition and endpoints
│   ├── core/
│   │   ├── __init__.py
│   │   └── rag.py        # The core RAG logic (embedding, search, LLM call)
│   └── models/
│       ├── __init__.py
│       └── schemas.py    # Pydantic models for API requests/responses
│
├── data/
│   ├── knowledge_base/   # Place your source documents (.txt, etc.) here
│   │   └── example.txt
│   └── vector_store/     # To store the generated FAISS index and mapping
│
├── scripts/
│   ├── __init__.py
│   └── ingest.py         # The one-time script to process data and create the index
│
├── .env                  # To store your secret API keys
├── .gitignore            # To exclude files from Git version control
├── requirements.txt      # Lists all Python package dependencies
└── README.md             # A description of your project
```

2. Step-by-Step Instructions

Follow these steps to build and run your project.

Step 1: Set Up Your Environment

1. **Create Project Folder**: Create the rag-fastapi-project directory and the sub-folders outlined above.
2. **Virtual Environment**: It's highly recommended to use a virtual environment.
   ```
   python -m venv venv
   source venv/bin/activate  # On Windows: venv\Scripts\activate
   ```

3. **Create requirements.txt**: Create this file in your root directory and add the following dependencies:
   ```
   fastapi
   uvicorn[standard]
   python-dotenv
   sentence-transformers
   faiss-cpu
   langchain
   openai
   ```

4. **Install Dependencies**:
   ```
   pip install -r requirements.txt
   ```

Step 2: Get Your LLM API Key

Since the LLM is not running locally, you'll need an API key from a provider. We'll use OpenAI as an example.

1. Go to the OpenAI Platform and create an API key.
2. Create a file named .env in your project's root directory.
3. Add your key to the .env file. **Never commit this file to Git.**
   ```
   OPENAI_API_KEY="sk-YourSecretKeyGoesHere"
   ```

Step 3: Write the Ingestion Script (scripts/ingest.py)

This script will read your documents, chunk them, create embeddings, and save the FAISS index.

- **Goal**: Implement the logic from "Phase 1" of the architecture guide.
- **Input**: Files in data/knowledge_base/.
- **Output**: Two files saved to data/vector_store/: faiss.index and chunk_mapping.pkl.
- **Key Libraries**: os, langchain.text_splitter, sentence_transformers, faiss, pickle.
- **Action**: Write the Python code for this script. You'll need functions to:

- Load all .txt files from the data/knowledge_base directory.
- Use RecursiveCharacterTextSplitter from LangChain to chunk the loaded text.
- Use SentenceTransformer('all-MiniLM-L6-v2') to create embeddings.
- Build and save the FAISS index and the chunk mapping file.

Step 4: Build the FastAPI Application

Now, build the API that will use your indexed data.

1. **app/models/schemas.py**: Define your Pydantic models for type checking and API documentation.
   ```
   from pydantic import BaseModel
   from typing import List

   class QueryRequest(BaseModel):
       question: str
       top_k: int = 5

   class QueryResponse(BaseModel):
       answer: str
       context: List[str]
   ```

2. **app/core/rag.py**: This is the heart of your application.
   - **Goal**: Create a function, let's call it get_rag_response, that takes a question and top_k as input.
   - **Logic**:
     1. Embed the user's question using the same SentenceTransformer model.
     2. Search the FAISS index to get the IDs of the most relevant chunks.
     3. Use the chunk_mapping.pkl file to retrieve the actual text of those chunks.
     4. Construct the augmented prompt using the retrieved context and the question.
     5. **Make the API call to OpenAI.** Use the openai library to send the prompt to a model like gpt-3.5-turbo.
     6. Return the final answer and the context that was used.
3. **app/main.py**: This file ties everything together.
   - **Goal**: Set up the FastAPI app, load models on startup, and create the API endpoint.
   - **On Startup**: Use the @app.on_event("startup") decorator to load your SentenceTransformer model, the FAISS index, and the chunk mapping into global variables. This prevents reloading them on every request. You'll also want to load your OpenAI API key from the .env file here.

- **Endpoint**: Create a POST /query endpoint that:
  - Accepts a QueryRequest object.
  - Calls the get_rag_response function from app/core/rag.py.
  - Returns a QueryResponse object.

Step 5: Run Your Application

1. **First, run the ingestion script** to create your vector database:
   python scripts/ingest.py

   Verify that faiss.index and chunk_mapping.pkl are created in data/vector_store/.
2. **Second, start the FastAPI server** from the root directory:
   uvicorn app.main:app --reload

   The --reload flag is great for development, as it automatically restarts the server when you save a file.

Step 6: Test Your Endpoint

Once the server is running, you can send requests to it. You can use tools like Postman/Insomnia, or a simple curl command in your terminal:

```
curl -X POST "http://127.0.0.1:8000/query" \
-H "Content-Type: application/json" \
-d '{"question": "What is the main topic of the document?"}'
```

You have a complete roadmap now. Happy coding!