Approach and Explanation for the RAG-based QA Bot Project

1. Introduction

The goal of this project is to develop a Retrieval-Augmented Generation (RAG)-based Question Answering (QA) bot. The bot takes a PDF as input, extracts its contents, and answers user queries based on the text in the document. Streamlit is used for the web interface, where users can upload PDFs and ask questions. Embeddings are generated from the document's content to retrieve relevant chunks of information using a vector store, which is then used to provide answers to the user's questions.

2. Project Structure

The project follows a modular structure, breaking down tasks into several components:
- \`app.py\`: The main Streamlit application where the user interacts by uploading a PDF and asking questions.
- \`pdf_loader.py\`: This file handles loading the PDF, extracting text from it, and splitting it into chunks for better processing.
- \`embeddings.py\`: This module generates embeddings from the chunks of text for use in retrieval.
- \`vector_store.py\`: Sets up the vector store, storing embeddings for efficient retrieval.
- \`qa_pipeline.py\`: The core pipeline that takes user questions, queries the vector store, and returns relevant answers.

3. Approach and Design Decisions

Below is the step-by-step explanation of the approach taken and the design decisions made:

3.1 Loading PDFs and Splitting Text

For loading PDFs, I initially used the PyPDF2 library, which allowed reading and extracting text from the uploaded PDF. However, I later shifted to using the \`langchain_community.document_loaders\` library, specifically \`PyPDFLoader\`, to leverage its ability to handle both local files and file-like objects. One challenge was handling files uploaded as \`BytesIO\` objects via Streamlit. To resolve this, I created a temporary file to write the \`BytesIO\` content and passed the path to \`PyPDFLoader\` for extraction.

3.2 Chunking Text and Generating Embeddings

Once the PDF is loaded and text is extracted, the next challenge was splitting the text into manageable chunks. This was done using simple newline delimiters, but it could be further refined to use smarter text-splitting techniques for better coherence. Embeddings were generated using Google's Generative AI API, where each chunk of text was embedded. This allowed us to later compare the user's query to relevant document chunks efficiently.

3.3 Setting Up the Vector Store

The vector store setup was critical for retrieval. I used an in-memory vector store for simplicity during development. This can be replaced by more scalable solutions like PineconeDB, FAISS, or other vector databases for production.

4. Challenges Faced and Solutions

4.1 Challenge: Handling File Uploads in Streamlit

Streamlit uploads files as `BytesIO` objects, which presented an issue for loading the PDF with `PyPDFLoader`, which requires a file path or file-like object. This caused errors during the PDF loading phase.

Solution:

To solve this, I wrote the `BytesIO` content into a temporary file using Python’s `tempfile` library. The temporary file’s path was then passed to `PyPDFLoader`, which successfully processed the PDF content.

4.2 Challenge: Efficient Text Retrieval

After generating embeddings, an efficient method of retrieving relevant chunks of text based on user questions was needed. Simply comparing user queries with the document’s text without embeddings would be slow and inaccurate.

Solution:

I used a vector store for storing and retrieving embeddings. The user’s query is embedded and compared against the pre-computed embeddings of the document chunks, and the closest match is retrieved. This allows for fast and accurate retrieval of relevant information.

4.3 Challenge: Answering User Queries

Once relevant text chunks were retrieved, the next challenge was formulating an answer. Simply displaying the raw text was insufficient.

Solution:

For this, I used Google’s Generative AI to take the user’s question and the relevant retrieved text chunks to generate a cohesive response. This enhanced the QA bot’s ability to give meaningful answers beyond just returning the closest text.

5. Dockerization

To ensure the project can be easily deployed, I chose to containerize it using Docker. The following steps were taken for Dockerization:

- **Base Image**: I used the official `python:3.9-slim` image to minimize the size.
- **Dependencies**: A `requirements.txt` was created to list all the necessary Python packages. These are installed during the Docker build process.
- **Environment Variables**: Sensitive API keys (such as the Google API key) were kept out of the code and passed as environment variables either via a `.env` file or directly during container runtime.
- **Streamlit Server**: The Dockerfile exposes port 8501 for Streamlit and sets the `CMD` to run the app inside the container.

6. Conclusion

This project demonstrated how to build a RAG-based QA bot by combining document embeddings, a vector store for retrieval, and a language model for generating answers. The challenges encountered, such as handling PDF uploads, efficient retrieval of information, and containerizing the application, were solved using a combination of Python libraries, Google Generative AI, and Docker. The result is a deployable application that can be further expanded with scalable infrastructure and advanced features for more complex use cases.