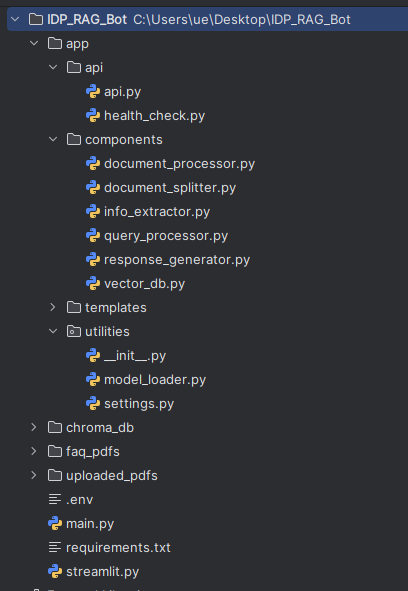
**Detailed Architecture and System Overview**

#### **1. System Architecture Overview**

The system is designed as a modular and scalable document processing and query handling service. Below is a detailed breakdown of each component and how they interact within the architecture:

**A. Components Overview:**

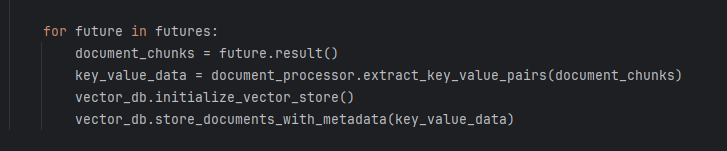
Folder Structure-



1. **Document Processor**:
   * **Role**: This component is responsible for loading PDF documents, splitting them into manageable chunks, and extracting key-value pairs using a large language model (LLM).
   * **Core Classes**: DocumentProcessor, CharacterTextSplitter
   * **Key Functionality**:
     + Loads documents from a directory.
     + Splits documents based on paragraphs or pages.
     + Utilizes a language model to extract key-value pairs from document chunks.
2. **Document Splitter**:
   * **Role**: Splits documents into chunks to make them manageable for processing and embedding.
   * **Core Classes**: DocumentSplitter
   * **Key Functionality**:
     + Splits documents by paragraphs or pages.
     + Generates smaller, more manageable document chunks for further processing.
3. **Info Extractor**:
   * **Role**: Embeds documents into vector representations that can be stored and searched.
   * **Core Classes**: InfoExtractor
   * **Key Functionality**:
     + Utilizes a pre-trained model to convert document content into vector embeddings.
     + Prepares documents for storage in the vector database.
4. **Vector Database (VectorDB)**:
   * **Role**: Stores document chunks with metadata in a vector database and performs similarity searches.
   * **Core Classes**: VectorDB
   * **Key Functionality**:
     + Stores document vectors and metadata in a persistent database (ChromaDB).
     + Executes similarity searches to find documents related to a given query.
5. **Response Generator**:
   * **Role**: Generates responses to user queries based on the stored document embeddings and associated metadata.
   * **Core Classes**: ResponseGenerator
   * **Key Functionality**:
     + Retrieves relevant document embeddings from the vector database.
     + Utilizes an LLM to generate accurate and contextually relevant responses.
     + Handles token limits and ensures responses are grounded in the provided document content.
6. **API Layer**:
   * **Role**: Exposes endpoints for document processing and query handling.
   * **Core Classes**: FastAPI Router
   * **Key Functionality**:
     + Provides a RESTful interface for interacting with the system.
     + Allows users to upload documents for processing and submit queries.

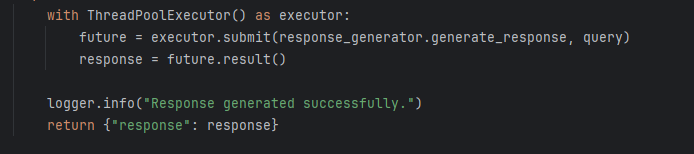
**B. Interaction Flow**:

1. **Document Processing**:
   * Users upload PDF documents via the /process\_documents/ endpoint.
   * The DocumentProcessor loads the documents and splits them into chunks.
   * The InfoExtractor embeds the document chunks into vectors.
   * These vectors, along with their metadata, are stored in the VectorDB.



2. **Query Handling**:

* Users submit a query via the /query/ endpoint.
* The ResponseGenerator retrieves relevant document vectors based on the query.
* The system checks whether to perform a similarity search or directly use the document content based on token limits.
* A context-aware response is generated and returned to the user.



**C. Data Flow and Storage**:

* **Data Flow**:
  + Documents are ingested, split, embedded, and stored as vectors.
  + Queries are processed by searching for similar vectors, and responses are generated based on the retrieved content.
* **Storage**:
  + Documents and embeddings are stored in ChromaDB, ensuring persistent storage and fast retrieval.
  + Metadata, such as key-value pairs, are stored alongside document embeddings to enhance query responses.

### **2. Chosen Technologies**

* **FastAPI**: A fast web framework for building APIs with Python, chosen for its performance and ease of integration with other components.
* **LangChain**: Provides the framework for managing language model operations, including text splitting and chain management.
* **Hugging Face Transformers**: Used for pre-trained models that handle embeddings and natural language processing tasks.
* **ChromaDB**: A vector database used to store and retrieve document embeddings efficiently, supporting high-performance similarity search operations.
* **Python**: The primary programming language for implementing the system, chosen for its rich ecosystem and libraries for NLP and AI.

### **3. How to Run the System**

#### **A. Prerequisites**

1. **Python 3.10**: Ensure Python is installed on your system.
2. **Environment Setup**:



**B. Install Dependencies**:

Use pip to install all required Python packages:



#### **C. Running the System**

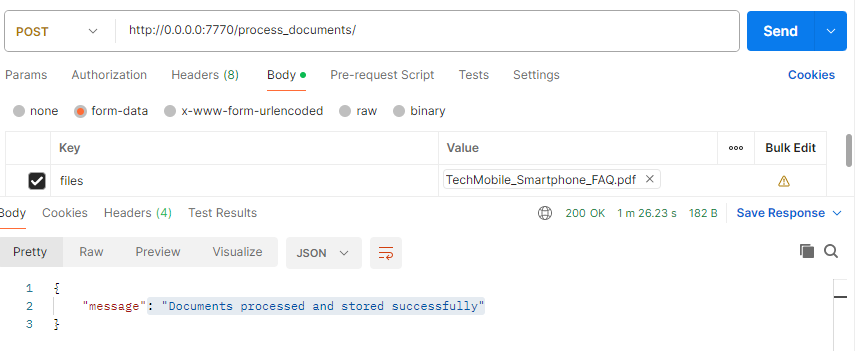
1. **Start the FastAPI Server**:



**uvicorn app.main:app --host 0.0.0.0 --port 7770**

**2.1 Access the API via Postman:**

* **Process Documents: Use the /process\_documents/ endpoint to upload and process documents.**

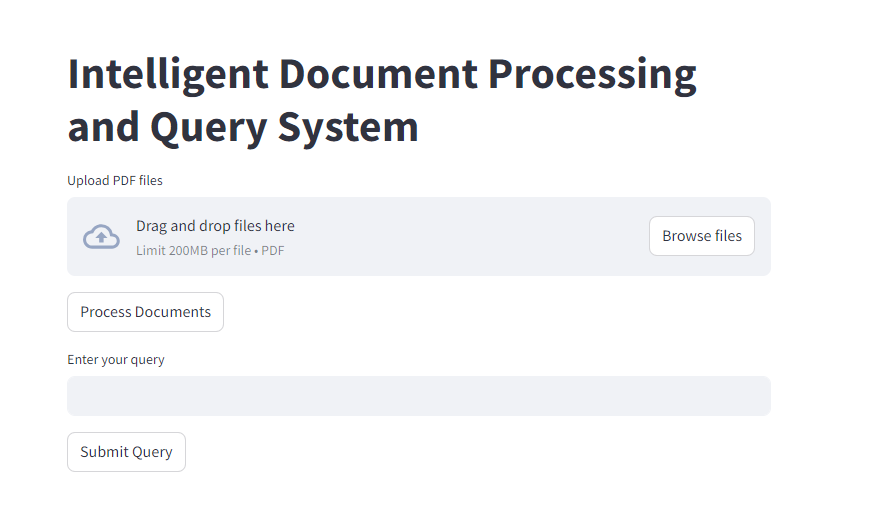
****

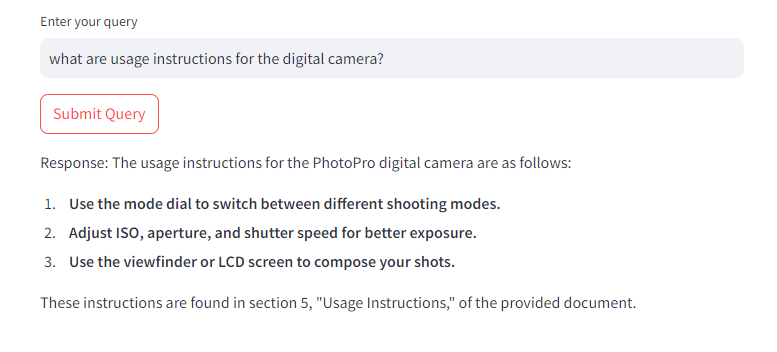
* **Submit Queries: Use the /query/ endpoint to submit queries and retrieve answers.**

****

**2.2 Access APIs via Streamlit UI**

**Run the command- streamlit run streamlit.py**

****

****

**4. Performance Report**

#### **A. System Performance**

**1. Response Times:**

* **Document Processing:**
  + Loading and Splitting: Time taken is proportional to the document size. For a 2,3-page document, it typically takes about 10 seconds.
  + Key-Value Pair Extraction: Extracting key-value pairs using the LLM takes around 1-3 seconds per document chunk.
* **Query Handling:**
  + Similarity Search: Typically, returns results within 5-7 seconds, depending on the size of the vector database and the complexity of the query.
  + Response Generation: Response times vary between 4-8 seconds, depending on whether the system uses document embeddings or directly utilizes content from the documents**.**

**2. Accuracy:**

* The system maintains high accuracy by ensuring that all responses are directly supported by the document content and metadata.
* No Out-of-Context Responses: The system is designed to prevent the generation of out-of-context or fabricated answers by following strict content adherence rules.

#### **B. Resource Utilization**

* **Memory Usage:**
  + **Vector Embeddings:** The system requires significant memory to store and retrieve document embeddings, particularly as the document corpus grows.
  + **Model Inference:** Running inference on large language models can be resource-intensive, particularly for generating complex responses.
* **CPU/GPU Utilization:**
  + **Document Processing:** CPU usage is moderate during document processing but can spike during embedding operations if using CPU inference.
  + **Query Handling:** Utilization is efficient, with most operations being I/O bound during retrieval and embedding search.

#### 

#### **C. Optimization Opportunities**

* **Asynchronous Processing:** Further improvements can be made by introducing more granular asynchronous processing, especially in document loading and query handling.
* **Model Selection:** Experimenting with lighter or more efficient models can reduce response times while maintaining accuracy.
* **Addition of queuing and parallel processing:** Although a basic futures-multi-processing pool has been added, proper implementation of a queueing system like rabbitMQ and some parallel processing like mechanism “celery” can enhance performance and scalability.
* **Database integration:** Along with addition of queues, a database must be added to store request ids for celery to perform parallel processing.