Document Intelligence Platform

Core Backend System

Document Handling

1. Create four API endpoints: **POST /document GET /document** and **GET /document/:id DELETE /document/:id**
2. **POST /document:** Receives two parameters, ***file*** with the file object and ***name*** with the file name.
3. This endpoint will do three things, first is putting the document in a directory, second is splitting it in chunks, vectorizing it and storing it in Chroma DB and third is storing the meta information in MySQL (more on these later).
4. **GET /document:** This will return the list of all documents in the database as an array.
5. **GET /document/:id:** Returns a specific document with the ID provided in the URL param.
6. **DELETE /document/:id:** Deletes the specific document and its respective chunks from the database with the ID provided in the URL.

Chunk Creation

1. Declare two constants at the top of the file: ***CHUNK_LENGTH*** and ***CHUNK_OVERLAP*** and let the initial values be ***25*** and ***5*** respectively. Note that these are the "tokens" not "characters". Generative AI models work in tokens. A token is roughly 4 characters of english language. So a chunk length of 25 tokens means 25 x 4 = 100 characters. Now, we will use a proper tokenizer to calculate tokens as 4 characters is just an approximation. Since we will be using OpenAI API, let's use tiktoken as our tokenizer.
2. Create a function called ***split_in_chunks(document_text: str) -> str[]***
3. Implement the function as follows:
    a. Create an empty array called ***chunks***
    b. Use the tiktoken library to convert the text into individual tokens
    c. You will get an array of tokens
    d. On this array, run a loop and create sub arrays of lengths ***CHUNK_LENGTH*** taking into consideration the ***CHUNK_OVERLAP***
    e. Each sub array is an individual chunk
    f. Join the tokens of each array to form a proper text and push it in the chunks array
    g. Now your text is splitted into chunks
    h. Return this array

Vectorization

When a document is uploaded, it will be stored in a folder. Once the document is stored, process the document as follows:

1. Determine the file type (txt, pdf or docx)
2. Read accordingly as these three will need different reading logics
3. Read the content of the document and store that in a variable
4. Call the previous ***split_in_chunks*** function and pass the document content as input
5. This will return an array of chunks for the document content
6. Setup Chroma DB (more on it later). By this point, it should already be setup
7. There should be two collections in Chroma DB
    a. **Document**
    b. **DocumentChunk**
8. Create an object in **Document** collection with the document name (file name), the total pages. This will return the **ID** of the created object. Store that in a variable.
9. Now, run a loop on the chunks array
    a. For each chunk, use the **SentenceTransformer** python library to create an embedding for that particular chunk
    b. Create an object in **DocumentChunk** collection with the **ID** of the document stored previously, the chunk number (keep it the current index in the chunk array +1), and the embedding
10. Once all chunks are processed, return success response from the API
11. Also, create a record in a **Documents** table in MySQL with the document name, the Chroma DB object ID for the document (previous **ID** variable) and an auto incrementing ID field

Chroma DB Setup

Chroma DB is a lightweight vector database. Configuring it is very simple. Follow the official documentation to set it up for python.

1. Everytime the project starts, create an instance of the DB
2. Make sure that there are two collections:
    a. **Document:** With fields for document name and the total pages
    b. **DocumentChunk:** With fields for document ID, chunk number, chunk content (which is the embedding)
3. If they don't exist, create them

AI Chat

All the chat messages from users will come to an API endpoint **POST /ask**. Once the messages reaches this API:

1. The API accepts the entire chat history from the user
2. The chat history will be passed to a function called ***make_search_query*** (more on that later)
3. This function will return a search query. Store that in a variable.
4. Pass this query in a function called ***find_similar_chunks*** (more on that later)

5. This will return similar chunks to the search query
6. We will use OpenAI's GPT-4o fpr this project
7. Prompt the model with the prompt available at the end of the document
8. Return the model's response along with the similar chunks as citations in JSON from the API

Similarity Search

Similarity search should use Chroma DB to find similar chunks to the provided query. Here is how:

1. Create a function called ***make_search_query(chat_history) -> str***
2. Prompt GPT-4o with the query creation prompt at the end of this document
3. Return the query created by this prompt
4. Create another function called ***find_similar_chunks(query: str) -> str[]***
5. Convert the query to embedding using **SentanceTransformer**
6. Use this to find top 5 matching chunks and their content
7. Return this from this function

Prompts

Query Creation

You will see your chat history with a user who is asking questions about a document. The document is stored in a database in chunks. Create a search query (plain english) to find similar chunks of the document which will later help you to answer the user's query.

Here is the chat history:

{chat history}

Only output the search query without any special formatting or surrounding text or anything else. Just provide the search query.

Response Generation

You are a helpful assistant helping a user with questions regarding a document. You will be provided your chat history with the user, along with relevant pieces of chunks from the document which will help you answer the user's query. Rely entirely on the relevant chunks provided to you and your chat history with the user to answer the user's query. Do not use your own knowledge to answer the query.

Here is your chat history with the user:

{chat history}

Here are relevant chunks from the document to help you answer the user's query. Rely only on them and the chat history for information:

{similar chunks}