Some important things to know :

Backend

1. The frontend and backend both are running on the same port, i.e http://localhost:8080, to access the backend use http://localhost:8080/api/v1
2. To add a new service you need to create a new folder inside the backend/apps/ directory, and then create a main.py file inside this file you can write the logic for this file. For. example, we created a folder called legalsearch and wrote a main.py in backend/apps directory.
3. If required we need to make changes in the config.py file in the backend directory when adding a new service
4. To write a new api for a particular service, dive directly in backend/apps/web/routers/ and choose the service you want to add a new api.
5. to access the running docker container use the command `docker exec -it <container-name> bash` and now you can see the files in running docker container
6. to check the running processes you can use, sudo lsof -i :<port-number>, if you want to kill this process simple type `kill <PID>`
7. Most important when you add new service/new route do not forget to add the api to fetch it in the frontend, src/lib/apis/ if you are creating a new service then you need to add a new folder and create a new file index.ts then inside it you can write the app. If there is an existing service in which you added a new route then choose that folder and write a api inside the index.ts file of that folder
8. to see all the running docker containers use docker ps -a
9. Use this forward your port from a server to local machine localhost, do this when there is a when in test-server. Also this command needs to be run from local machine cli not from inside the server. `ssh -i /home/ronny/Downloads/techpeek_1.pem -L localhost:8080:localhost:8080 ubuntu@ec2-13-237-148-39.ap-southeast-2.compute.amazonaws.com`
10. If you are adding a new service like for e.g legalsearch, you are creating a folder in backend/apps, you need to import this inside the backend/main.py folder, as `from apps.legalsearch.main import app as legal_app`,
11. After completing the previous step you need to mount this service on the backend using `app.mount("/legalsearch/api/v1", legal_app)`
12. be sure to update your python, node and pip version to the latest.
13. To update the whole chat.techpeek.in, you need to build a new docker image, `docker build -t yourimagename:tag . ` and then you can test this docker image using `docker run -p 8080:8080 yourimagename:tag` , dont forget to push this image so that it can be accessed even if deleted ever from the system.
14. if you want to completely clean the current service running on chat.techpeek.in then you need to first delete the docker containers, then delete the docker image using `docker rmi <image name>` and also delete the docker volume related to it `docker volumes ls`, `docker volumes rm <volume name>`
15. to stop and start the docker container use, `docker start <container id>`, `<docker stop <container id>`, to see only running docker containers use `docker ps` to see all use `docker ps -a`.

16. if you want to make changes to the ui on techpeek.in , from the root directory goto /var/www/html , there you will different folders where UI is being fetched from. the current ui is fetched from the build folder.
17. to make changes on the routes on the server from root directory goto /etc/nginx/sites-enabled, there you will find 3 services the default is for techpeek.in, other one is for chat.techpeek and third one has our ollama running on it.
18. a new db file is created in backend/data/ called webui.db which contains all the info in it from auth to chat history everything
19. To install the open-webui locally ```git clone https://github.com/open-webui/open-webui.git cd open-webui/ - Copying required .env file cp -RPp .env.example .env - Building Frontend Using Node npm i npm run build - Serving Frontend with the Backend cd ./backend pip install -r requirements.txt -U bash start.sh```

Some important things to know:

Frontend

1. If the user is not authenticated and tries to access a route that requires authentication, then the user will be redirected to auth page which can be found at src/routes/auth/+page.svelte
2. When authenticated, then it will show you the page in src/routes/(app)/+page.svelte, the files with '+' are svelte reserved files.
3. If you to add a page, then you might need to create a folder inside the src/routes/(app)/ directory and add a +page.svelte file inside which you can write the js logic and develop the ui.
4. you can write the apis for different services as required in the src/lib/apis directory, as discussed in the backend documentation.
5. you can create a separate api base url for your service like export const LEGAL_API_BASE_URL = `${WEBUI_BASE_URL}/legalsearch/api/v1`; use this while writing the api

Milvus db installation

1. For installing milvus db you have to run the Docker Compose file first
2. After you have executed the docker compose files, 3 new containers of milvus db will get started, these 3 containers are very important do not delete them until and unless very important
3. After this to create the embeddings you need to run the ‘script.py’ file, it takes some time for the embeddings to get created, these embeddings and all the data of milvus db is stored in a ‘volumes’ folder.
4. If you want to update the milvus db with new embeddings then you can do so by replacing the volumes folder with a updated volumes folder without deleting the existing milvus db containers

5. If you want to cross check whether the milvus db is properly loaded you can run the ‘query.py’ file.
6. Do not run the script.py once the milvus db is populated, if you re-run the script.py then the existing collection will get deleted and you will have to wait again until the script.py execution is finished to get a repopulated db again

Everything about LegalSearch Service

Overview

The LegalSearch service is designed to help users search for legal cases and receive concise summaries and relevant documents. This service involves a backend built with FastAPI and a frontend using Svelte, with integration of large language models (LLMs) for generating summaries and handling queries.

Backend Architecture

FastAPI

FastAPI is chosen for the backend due to its speed and ease of use. It provides an asynchronous framework which makes it efficient for handling multiple requests simultaneously.

1. **CORS Middleware**: CORS (Cross-Origin Resource Sharing) is enabled to allow the frontend, which might be served from a different domain, to interact with the backend.
2. **Data Models**: Pydantic is used to define data models such as `LegalQuery`, `DocumentDescription`, and `AnswerResult`. These models ensure data validation and structure.
3. **Sentence Transformers**: The `SentenceTransformer` model is used to encode queries into embeddings, which are vector representations of the text.
4. **LLM Integration**: The HuggingFace Hub is used to access an LLM (e.g., Nous Hermes) for generating summaries of legal documents and answering user queries.
5. **Chroma CLI**: This is a tool for managing and querying the document collection. It allows efficient retrieval of relevant legal documents based on the encoded query embeddings.

Process Flow

1. **Query Handling**: When a query is received, it's encoded into an embedding using the SentenceTransformer model.
2. **Document Retrieval**: The encoded query is used to search the document collection via Chroma CLI. The collection contains legal documents with metadata such as date of judgment and case title.

3. **Summary Generation**: For each retrieved document, a short summary is generated using the LLM.
4. **Response Construction**: The service constructs a response with the generated summaries and sends it back to the frontend.

Frontend Architecture

Svelte

Svelte is a modern front-end framework that offers a reactive, component-based approach to building user interfaces.

1. **User Interface**: The UI consists of input fields for entering the legal query and specifying the number of results. It also includes components for displaying messages (queries and results) and a loading indicator.
2. **APIs**: The frontend communicates with the backend using the `legalSearch` function, which makes POST requests to the backend's `/legalsearch` endpoint.
3. **Error Handling**: Errors during the fetch operation are caught and handled gracefully, with appropriate messages displayed to the user.
4. **Dynamic Updates**: The UI updates dynamically based on user interactions and the state of the request (loading, results received, etc.).

Process Flow

1. **Input Handling**: Users enter their query and the desired number of results. This input is bound to variables in Svelte.
2. **API Request**: When the search is initiated, an API request is made to the backend. A loading indicator is shown, and the progress is updated dynamically.
3. **Result Display**: The received results, including document summaries, are displayed in a user-friendly format with HTML for better readability.
4. **Cancellation**: Users can cancel an ongoing search, which aborts the API request and updates the UI accordingly.

Connecting Backend and Frontend

1. **API Endpoint**: The backend exposes an endpoint `/legalsearch` which the frontend calls to perform the search.
2. **CORS Middleware**: This ensures that the frontend can make requests to the backend without being blocked by the browser's same-origin policy.
3. **Authorization**: The frontend includes a token in the request headers for authentication and authorization.

Use of LLMs

1. **Sentence Transformer**: This model is used for encoding the legal query into an embedding that can be compared with document embeddings for similarity.
2. **HuggingFace Hub LLM**: This is used to generate human-readable summaries of the legal documents and to provide an answer based on the retrieved documents.

Detailed Explanation of the Document QnA Service

Overview

The Document QnA Service is a web application designed to let users query documents using advanced machine learning models. It extracts relevant information from the documents and provides accurate responses based on user queries. The system comprises a backend built with FastAPI and a frontend built with Svelte, integrated with machine learning models and a vector database for efficient document retrieval.

Backend

1. FastAPI Setup

- **FastAPI Application:** The backend application is created using FastAPI, a modern web framework for building APIs with Python. The app is configured to handle various API endpoints, manage the application state, and provide CORS (Cross-Origin Resource Sharing) support to allow requests from different origins.
- **State Management:** The application state is used to store configuration settings such as chunk size, chunk overlap, embedding model details, and RAG template. These settings are essential for processing documents and handling queries.

2. Model and Vector Database Integration

- **SentenceTransformer:** The SentenceTransformer model from the `sentence_transformers` library is used to generate embeddings for both the documents and user queries. These embeddings help in finding the relevance between the query and document content.
- **ChromaDB:** ChromaDB is used as the vector database. It stores document embeddings and allows efficient retrieval of the most relevant document chunks based on query embeddings. Collections in ChromaDB represent different sets of documents, each identified by a unique collection name.

3. Document Handling

- **Document Loaders:** Various document loaders are integrated to handle different file formats like PDF, CSV, Word, and more. These loaders read the content of the documents and prepare them for processing.
- **Text Splitter:** Documents are often too large to process as a whole. The RecursiveCharacterTextSplitter splits documents into smaller, manageable chunks. This ensures efficient processing and retrieval.
- **Storing Documents:** Uploaded documents are processed and stored in the vector database. The embeddings for the document chunks are created and added to ChromaDB collections.

4. Query Handling

- **Query Processing:** When a user submits a query, the system generates an embedding for the query using the SentenceTransformer model. This embedding is used to search the vector database for the most relevant document chunks.
- **Merging Results:** If the query spans multiple document collections, the results from each collection are merged and sorted based on relevance to provide a comprehensive response.

5. Endpoints

- **Status and Configuration Endpoints:** These endpoints provide information about the service status, chunk parameters, embedding models, and RAG templates.
- **Document Management Endpoints:** These endpoints handle document uploading, embedding model updates, directory scanning for documents, and resetting the vector database.
- **Query Endpoints:** These endpoints process user queries and return the most relevant document chunks.

Frontend

1. Svelte Setup

- **Stores:** Svelte stores are used to manage the application state, including documents, user settings, and current chat sessions.
- **Components:** Various UI components are used to create a responsive and interactive user interface. These include forms for uploading documents, modals for settings, and components for displaying query results.

2. Document Management

- **Document Upload:** Users can upload documents through the frontend interface. The uploaded documents are sent to the backend for processing and storage.

- **Document Listing:** A list of uploaded documents is displayed, allowing users to search, filter, and manage their documents. Users can delete or edit document metadata as needed.

3. Query Interface

- **Query Input:** Users can input their queries in a search bar. These queries are sent to the backend, which processes them and retrieves the most relevant document chunks.
- **Result Display:** The frontend displays the retrieved results, highlighting the relevant parts of the documents. This helps users quickly find the information they are looking for.

4. Interactivity

- **Real-time Updates:** The frontend interacts with the backend in real-time using asynchronous API calls. This provides immediate feedback to users on their queries and document uploads.
- **Drag and Drop:** The interface supports drag-and-drop functionality for uploading files, making it user-friendly and efficient.

Connection Between Backend and Frontend

- **APIs:** The frontend communicates with the backend using RESTful APIs. These APIs handle document uploads, querying, updating settings, and more.
- **Authentication:** Secure endpoints use token-based authentication to ensure that only authorized users can access them.
- **Real-time Feedback:** The frontend provides real-time feedback by making asynchronous API calls and updating the UI based on the responses from the backend.

Use of LLMs

- **Embedding Generation:** Large Language Models (LLMs), like SentenceTransformer, are used to generate embeddings for documents and queries. These embeddings help in finding the relevance between user queries and document content.
- **Query Processing:** The embeddings are compared to find the most relevant document chunks, which are then returned to the user as the query result.

Important Files

1. backend/apps/rag/main.py

- **Purpose:** This file contains the main setup for the FastAPI application and defines various API endpoints.
- **Key Responsibilities:**

    - **Application Setup:** Initializes the FastAPI application, sets up middleware for CORS, and manages application state.
    - **Embedding and Vector Database:** Integrates SentenceTransformer for generating embeddings and sets up ChromaDB for storing and querying document embeddings.
    - **Endpoints:**
        - **Status Endpoints:** Provides status and configuration details.
        - **Document Handling Endpoints:** Allows uploading, processing, and managing documents. This includes endpoints for uploading files, web URLs, and scanning directories.
        - **Query Endpoints:** Handles user queries, retrieves relevant document chunks, and merges results from multiple collections.

2. src/lib/apis/documents/index.ts

- **Purpose:** Contains the frontend API functions for interacting with document management endpoints in the backend.
- **Key Functions:**
    - **createNewDoc:** Creates a new document entry in the backend.
    - **getDocs:** Retrieves a list of all documents.
    - **getDocByName:** Fetches a document by its name.
    - **updateDocByName:** Updates a document's details using its name.
    - **tagDocByName:** Adds tags to a document by its name.
    - **deleteDocByName:** Deletes a document using its name.
- **Usage:** These functions are used by the frontend components to manage documents, including creating, retrieving, updating, tagging, and deleting documents.

3. src/lib/apis/rag/index.ts

- **Purpose:** Contains the frontend API functions for interacting with Retrieval-Augmented Generation (RAG) specific endpoints.
- **Key Functions:**
    - **getChunkParams:** Fetches the chunk size and overlap parameters.
    - **updateChunkParams:** Updates the chunk size and overlap parameters.
    - **getRAGTemplate:** Retrieves the RAG template.
    - **getQuerySettings:** Fetches the current query settings.
    - **updateQuerySettings:** Updates the query settings.
    - **uploadDocToVectorDB:** Uploads a document file to the vector database.
    - **uploadWebToVectorDB:** Uploads a web URL's content to the vector database.
    - **queryDoc:** Queries a single document collection.
    - **queryCollection:** Queries multiple document collections.
    - **scanDocs:** Initiates a scan of the document directory.
    - **resetVectorDB:** Resets the vector database.

- **Usage:** These functions are used by the frontend components to interact with the backend for RAG operations, including managing chunk parameters, templates, and querying the vector database.

4. src/lib/stores/index.ts

- **Purpose:** Contains Svelte stores to manage the application state.
- **Key Stores:**
  - **WEBUI_NAME:** Stores the name of the web UI.
  - **config:** Holds the configuration settings.
  - **user:** Stores user information.
  - **theme:** Manages the theme (e.g., dark mode).
  - **chatId:** Stores the current chat ID.
  - **chats:** Holds the list of chats.
  - **tags:** Manages tags for documents and chats.
  - **models:** Stores information about available models.
  - **modelfiles:** Manages the list of model files.
  - **prompts:** Stores the list of prompts.
  - **documents:** Holds the list of documents, each with details like collection name, filename, name, and title.
  - **settings:** Manages user settings.
  - **showSettings:** Controls the visibility of the settings modal.
  - **showChangelog:** Controls the visibility of the changelog modal.
- **Usage:** These stores are used across the Svelte components to manage and share state, ensuring reactivity and state consistency throughout the application.

5. src/routes/(app)/documents/+page.svelte

- **Purpose:** Svelte component for the document management page.
- **Key Responsibilities:**
  - **Document Upload:** Handles the upload of documents through file input and drag-and-drop functionality.
  - **Document Listing:** Displays the list of documents, allows filtering by tags, and provides search functionality.
  - **Document Management:** Provides options to delete, edit, and tag documents. Users can also import and export document mappings.
  - **UI Components:** Utilizes various UI components like checkboxes, modals, and placeholders to enhance user interaction and experience.
- **Usage:** This component is used for managing documents within the application, allowing users to upload, view, and interact with their documents.

6. src/routes/(app)/+page.svelte

- **Purpose:** Svelte component for the main application interface.
- **Key Responsibilities:**

    - **Chat Interface:** Manages the chat interface, allowing users to input queries and view responses.
    - **Model Selection:** Provides a UI for selecting models to use for generating responses.
    - **Message Handling:** Manages the sending and receiving of messages, including handling user inputs and displaying assistant responses.
    - **Interactivity:** Implements features like auto-scrolling, real-time updates, and response management (e.g., stopping and regenerating responses).
    - **UI Components:** Utilizes components like Navbar, Messages, ModelSelector, and MessageInput to create a cohesive and interactive user experience.
- **Usage:** This component serves as the main interface for users to interact with the document QnA service, facilitating query input, response display, and overall user interaction.
    -

Summary

The Document QnA Service efficiently combines backend processing capabilities with a responsive frontend interface. It leverages modern NLP models for embedding generation and retrieval, ensuring accurate and contextually relevant answers to user queries. The integration of FastAPI, Svelte, and ChromaDB provides a robust and scalable solution for document querying and management.