

PROJECT SUMMARY AND BACKEND ENDPOINTS

The RAG-Powered Academic Assistance System is a tailored solution designed to help students at your college with their studies by providing contextually relevant and course-specific answers. Unlike traditional AI-powered Q&A systems that offer general information, this project leverages Retrieval-Augmented Generation (RAG) to deliver responses aligned with the academic structure, terminology, and complexity of your college courses.

Key Features & Functionality

- **User Authentication & Session Management**
 - Secure login and registration with JWT token-based authentication.
 - Endpoint-based access control to ensure data security and user privacy.
- **Chat Management**
 - Users can create, edit, and delete chatrooms.
 - Chat history is maintained, enabling contextual conversations.
 - Search functionality allows students to quickly retrieve past interactions.
- **RAG-Powered Q&A**
 - Students can ask academic questions through the system.
 - The RAG mode retrieves relevant course materials from a vector database (populated with professor-provided documents) to enrich the prompt before generating a response.
 - This ensures answers are precise, course-specific, and adhere to the academic language and complexity expected by students and professors.
- **Integration with OpenAI**
 - The system interacts with OpenAI's API to generate responses.
 - When RAG mode is enabled, the prompt is enriched with retrieved course-specific content, making the answer more accurate and contextually relevant.

- Without RAG mode, the system behaves like a standard AI chat.
- Chat History and Contextual Conversations
 - Students can view previous conversations, enabling continuity in discussions.
 - The system differentiates between user messages and AI-generated responses, making it easy to follow the conversation flow.

Impact & Benefits

- **Enhanced Learning Experience:** Provides students with **precise, course-aligned answers**, reducing the gap between generic AI responses and institution-specific knowledge.
- **Improved Efficiency:** Saves students' time by offering direct, accurate, and relevant answers from course materials.

API ENDPOINTS:

- <http://localhost:8080/register> (POST)

Input json structure:

```
{
  "username": "<string>",
  "password": "<string>"
}
```

Output json structures: for this api endpoint, the output is just a string

HTTP STATUS 409 : “username already exists” is returned

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200: “success” is returned

- <http://localhost:8080/login> (POST)

Input json structure:

```
{
```

```
    "username": "<string>",
    "password": "<string>"
  }
```

Output json structures: for this api endpoint, the output is a string

HTTP STATUS 401 : “failure” is returned (due to wrong password or non-existent username)

HTTP STATUS 200: jwt token in string form is returned.

NOTE: From this point onward, every endpoint requires authentication with a bearer token. If the bearer token is incorrect or missing or expired, a status code of 401 will be returned. In such cases, redirect the user to the login/register page.

- <http://localhost:8080/chat/create> (POST) {endpoint for creating a new chat}

Input json structure:

```
{
  "username": "<string>",
  "chatName": "<string>" (chat name which the user provides)
}
```

Output json structures: for this api endpoint, the output is a string

HTTP STATUS 404 : “no such user” is returned (due to non-existent username)

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200: “created” chatroom is created.

- <http://localhost:8080/chat/get/account/{username}> (GET) {endpoint for retrieving all the chats of a user.}

Input json structure: no input json structure, extracts string username from {username}

Output json structures: for this api endpoint, the output can be:

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200:

```
[
  {
    "id": <integer>,
    "username": "<string>",
    "chatName": "<string>"
  }
]
```

Basically, a list of chats, make sure to store id, since it is the unique identifier of a chat and will be used to retrieve message history or append new conversations to the chat. For all future references, the words id and chatId will be used interchangeably.

- <http://localhost:8080/chat/edit/{id}/{name}> (POST) {endpoint for editing the name of a chat}

Input json structure: no input json structure, extracts string name from {name} and integer chatId from {id}.

Output json structures: for this api endpoint, the output is a string.

HTTP STATUS 404: (chat with chatId {id} does not exist)

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200: "edited successfully" (the name of the chat is changed in the database).

- <http://localhost:8080/chat/search/{username}/{keyword}> (GET) {This endpoint is basically a search function... It gets all the chats of user where the chat name is similar or equal to the search keyword}

Input json structure: no input json structure, extracts string username from {username} and string keyword from {keyword}

Output json structures: for this api endpoint, the output can be:

HTTP STATUS 500: (no message only the internal server error status code is

returned)

HTTP STATUS 200:

```
[
  {
    "id": <integer>,
    "username": "<string>",
    "chatName": "<string>"
  }
]
```

Basically, a list of chats whose username matches {username} and chat names match or are similar to the {keyword}.

- <http://localhost:8080/chat/delete/{id}> (DELETE) {endpoint for deleting a chat}

Input json structure: no input json structure, extracts integer chatId from {id}.

Output json structures: for this api endpoint, the output is a string.

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200: "deleted" (the chat has been deleted from the DB along with it's history).

- <http://localhost:8080/question> (POST) {endpoint for sending a prompt to the OpenAI api in context of the chat history of a particular chat with or without processing and modifying the prompt through the RAG system}

Input json structure:

```
{
  "username": "<string>",
  "chatId": <integer>,
  "ragMode": <boolean>,
  "question": "<string>"
}
```

chatId: the id of the chat which the message should be part of.

ragMode: true if user enables the rag mode, false if user disables the rag mode.

question: the prompt which the user gives.

Advice -> just a suggestion but, you can keep a toggle switch in the top-right or

top-left corner for enabling or disabling the ragMode.

Output json structures: for this api endpoint, the output is a string

HTTP STATUS 404 : (chatId not present in DB, basically, chat does not exist)

HTTP STATUS 400: (question is an empty string)

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200: (OpenAI response is returned as a string to the user)

- <http://localhost:8080/fetch/{chatId}> (GET) {This endpoint retrieves the chat history of a particular chat}

Input json structure: no input json structure, extracts int chatId from {chatId}

Output json structures: for this api endpoint, the output can be:

HTTP STATUS 500: (no message only the internal server error status code is returned)

HTTP STATUS 200:

```
[
  {
    "message": "<string>",
    "user": "<boolean>"
  }
]
```


Basically, returns the chat history of a chat where,

message -> String which is part of the conversation history.

user -> true if the message is written by a user, false if the message is a response from AI.

Example:

```
[
  {
    "message": "My name is Om naidu",
    "user": true
  },
  {
    "message": "Hello Om Naidu, how can I assist you today?",
    "user": false
  },
  {
    "message": "What is my name?",
    "user": true
  },
  {
    "message": "Your name is Om Naidu.",
    "user": false
  }
]
```



EXTRA:

This was the response of the AI when it was asked “What is waterfall model?” through the RAG system:

The Waterfall Model is a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through various stages including planning, modeling, construction, and deployment.

This model is often referred to as the classic life cycle and is typically used when requirements are well understood at the beginning of the project.

The steps involved in the Waterfall Model are as follows:

1. **Requirements**: This is the initial phase where the project requirements are gathered and defined.
2. **Planning**: In this phase, plans are made regarding how the software will meet the requirements.
3. **Design**: The system architecture is structured during this phase.
4. **Implementation**: This involves writing the code that fulfills the design.
5. **Testing**: The software is verified to ensure that it works as intended and meets the quality standards.
6. **Deployment**: The software is released for use.

7. ****Maintenance****: This is an ongoing phase that involves support, bug fixes, and software updates after the deployment of the software.

However, it's important to note that the Waterfall Model has certain disadvantages such as difficulty in measuring progress within stages, inability to accommodate changing requirements, and high risk and uncertainty associated with the process model. The model also doesn't produce any working software until late in the life cycle. It is not suitable for complex or long-duration projects as requirements may change over time.