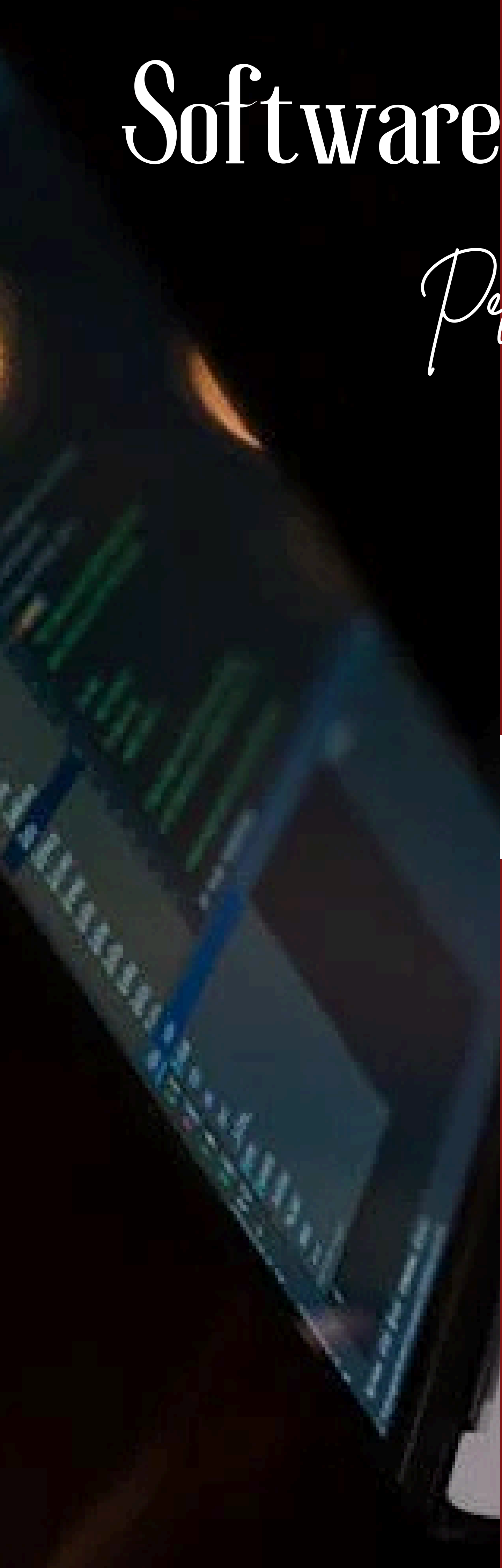
Software Engineering Project



TEAM 16

Prepared By:

Anish Maity (21f3000417)
Abhishek Darji (22f1000678)
Achin Aggarwal (22f1001390)
Sarath Sasidharan Pillai (22f1000152)
Atharva Sarbhukan (22f1000533)
Rituparno Sen (21f1004275)
Dr. Ambrish Naik (22f2000424)

API ENDPOINTS

API ENDPOINTS

Detailed Descriptions

1. Authentication

- **/signup :**

The Signup handles user registration by accepting details such as name, username, email, password, about, and role (either "Student" or "Instructor"). It validates that all required fields are provided, checks for duplicate usernames or emails, and ensures the specified role exists in the database. The password is securely hashed before being stored, and the user is added to the database with a timestamp for account creation. If the user is an instructor, a pending approval request is created. The method returns a success message based on the role, or an error message if any validation fails, such as missing fields or invalid roles.

- **/login :**

The Login handles user login by accepting the username (or email) and password. It first checks if the provided credentials match any user in the database. If no matching user is found or the password is incorrect, it returns an "Invalid credentials" message. For users with the "Instructor" role, the system checks if their approval is pending by looking up their status in the InstructorRequest table. If the instructor is not approved, the login is blocked with a message stating "Instructor approval pending." If the credentials are valid and the instructor is approved (if applicable), the method updates the user's last_login timestamp and generates both an access_token and a refresh_token for the user. The response includes these tokens along with the username and a success message indicating successful login.

- **/token_refresh :**

The RefreshToken allows users to obtain a new access_token using a valid refresh_token. The @jwt_required(refresh=True) decorator ensures that only requests with a valid refresh token can access this endpoint. Upon receiving the refresh_token, the method decodes it to extract the user's user_id, role, and username from the payload. Then, a new access_token is generated using these details. The response includes the newly generated access_token, enabling the user to continue their session without needing to log in again.

- **/logout :**

The Logout handles user logout functionality. It uses the @jwt_required() decorator to ensure that only authenticated users (those with a valid JWT token) can access this endpoint. When a user sends a POST request to log out, the method returns a success message: "Logout successful". If an exception occurs during the process, the method catches the error and returns an error message with the exception details. However, since the logout process generally involves invalidating the user's session or token on the client side (which is not explicitly handled in this code), the response mainly indicates a successful logout action.

2. Admin

- **/topquery :**

The TopSupportQueries retrieves the top 5 most asked queries in the past 7 days from the ChatbotHistory table, requiring user authentication with the @jwt_required() decorator. It calculates the date for 7 days ago, queries the database to count how often each query was asked, groups and orders the queries by count in descending order, and limits the result to the top 5. The queries are formatted into a list of dictionaries with query text and count, and the response is returned as a JSON object containing the top 5 queries.

- **/query_detail/<int:query_id> :**

The QueryDetail retrieves detailed information about a specific query by its query_id. The @jwt_required() decorator ensures the user is authenticated before accessing this resource. The method fetches the query from the IssueQuery table using the query_id. If the query is not found, it returns a 404 error with a "Query not found" message. If the query exists, it retrieves the associated student's name from the User table (or "Unknown Student" if no student is found) and returns a JSON response containing the query details such as the query text, student name, and timestamp.

- **/solve_query/<int:query_id> :**

The SolveQuery allows an admin to solve a specific query by providing an answer. The @jwt_required() decorator ensures the user is authenticated. It first checks if the logged-in user is an admin by verifying their role. If the user is not an admin, it returns a 403 Unauthorized message. Then, the method fetches the query using the provided query_id. If the query is not found, it returns a 404 error. The user must provide an answer for the query; if not, it returns a 400 error. Upon receiving a valid answer, it creates a new SolveIssue entry to store the solution, commits it to the database, and returns a success message with details of the solved query. If an error occurs during the database operation, it rolls back the transaction and returns a 500 error with the exception details.

- **/add_course :**

The AddCourse allows an admin to add a new course. It checks if the user is an admin, returning a 403 error if not. The method then validates that both the course name and description are provided. If any are missing, a 400 error is returned. Upon successful validation, a new course is created, added to the database, and committed. A success message is returned, or a 500 error is returned if an issue occurs during the process.

- **/edit_course/<int:course_id> :**

The EditCourse allows an admin to update a course. It first checks if the user is an admin, returning a 403 error if not. Then, it verifies the course exists, returning a 404 error if not found. The method updates the course's name and description based on the provided data, leaving existing values unchanged if not provided. After committing the changes to the database, it returns a success message, or a 500 error if an issue occurs during the process.

- **/add_course/<int:course_id>/material :**

The AddCourseMaterial allows an admin to add new materials to a course. It checks if the user is an admin, returning a 403 error if not. Then, it verifies that the course exists, returning a 404 error if not found. The method validates that both the title and material link are provided; otherwise, it returns a 400 error. Upon successful validation, a new course material is added to the database. A success message is returned, or a 500 error is returned if an issue occurs during the process.

- **/edit_course/material/<int:material_id> :**

The EditCourseMaterial allows an admin to update course materials. It first checks if the user is an admin, returning a 403 error if not. Then, it verifies the material exists, returning a 404 error if not found. The method updates the material's title and link based on the provided data, leaving existing values unchanged if not provided. After committing the changes to the database, a success message is returned, or a 500 error is returned if there is an issue during the update process.

3. Student

- **/student_profile :**

The StudentProfile retrieves a student's profile information. It first authenticates the user with jwt_required(), then fetches the student's details from the database using their ID. If the student is not found, it returns a 404 error. Otherwise, it returns a JSON response containing the student's ID, name, username, email, role, and last login timestamp.

- **/student_dashboard :**

The StudentDashboard retrieves a student's enrolled courses, assignments, course materials, chatbot history, and issue queries using jwt_required() for authentication. It fetches student details from User, retrieves enrolled courses from UserCourse, and gathers related assignments and materials. Additionally, it collects chatbot interactions from ChatbotHistory and issue queries from IssueQuery. The code could be optimized by reducing individual database queries using join(), applying lazy='joined' in SQLAlchemy relationships, and limiting chatbot history and queries to recent entries for better performance.

4. Instructor

- **/add_suplementary :**

The LessonResource allows authenticated users to add supplementary materials to a course using jwt_required(). It validates the presence of course_id, material_type, and content before creating a new SupplementaryMaterial entry in the database. The lesson is then added and committed to the database. Optimization can be achieved by handling bulk inserts efficiently, implementing validation for material types, and indexing frequently queried fields like course_id to enhance performance.

- **/add_assigments :**

The AssignmentResource enables authenticated users to add assignments to a course using jwt_required(). It validates the presence of course_id, week_number, assignment_link, and description before creating a new Assignment entry in the database. Once validated, the assignment is added and committed. Optimization can be achieved by ensuring unique constraints on week_number per course, implementing validation for assignment links, and indexing course_id for efficient querying.

- **/add_livesession :**

The LiveSessionResource class allows authenticated users to add live sessions to a course using jwt_required(). It checks for required fields (course_id, yt_link, description) before creating a LiveSession entry in the database. The session is then added and committed. Optimization can include validating yt_link format, enforcing unique constraints on sessions per course, and indexing course_id for faster lookups.

5. Professor

- **/pending_instructor :**

The PendingInstructors retrieves all pending instructor requests by first verifying the identity of the professor via jwt_required(). It then checks if the professor exists in the database and fetches pending instructor requests from InstructorRequest based on their status being PENDING. The response is formatted to return the request details in JSON format, including the instructor ID, status, and creation timestamp. For optimization, you could consider limiting the number of results returned (pagination), indexing status for better query performance, and adding error handling for potential database issues.

- **/approve_instructor :**

The ApproveInstructor allows a professor to approve or reject an instructor request by changing its status. It first validates the professor's identity via JWT authentication. Then, it retrieves the status from the request payload and checks if it's valid (either "Approved" or "Rejected"). If the status is valid, it fetches the instructor request from the InstructorRequest model. If the request exists, it updates the status and commits the change to the database. A success message is returned upon completion. For optimization, consider adding error handling for database failures, validating the professor's role to ensure they are authorized to approve/reject, and checking if the instructor request has already been processed.

- **/solved_issues :**

The SolvedIssues retrieves solved issues by joining the IssueQuery and SolveIssue tables on the issue ID, returning both the issue details and its solution. The retrieved data is processed into a list containing the issue ID, details, creation timestamp, solver ID, solution answer, and the solved time. The class then returns a JSON response with the count of solved issues and the corresponding list. Optimizations could include using with_entities() to select only necessary columns, adding pagination for large data sets, and improving error handling when no solved issues are present.

- **/pending_issues :**

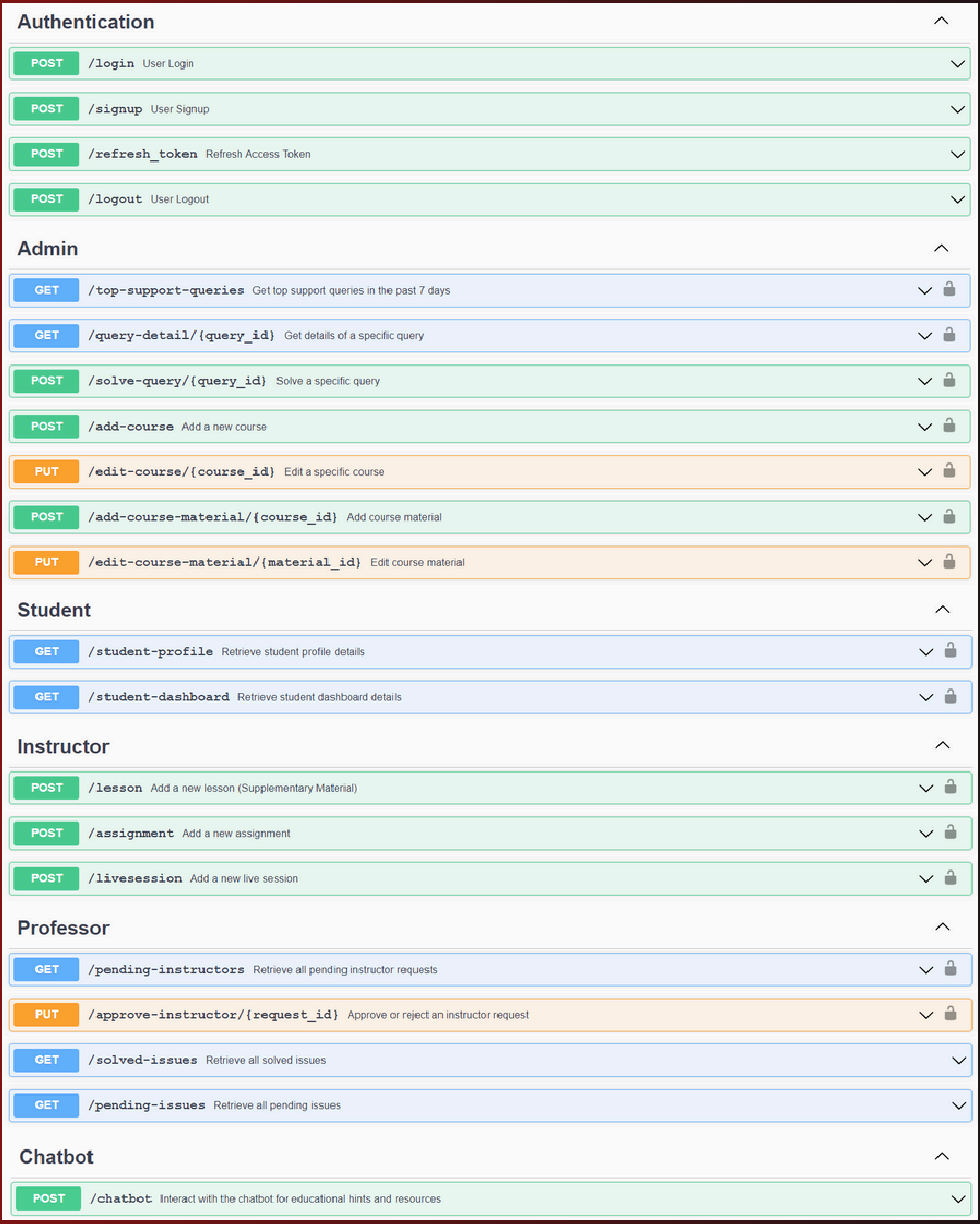
The PendingIssues retrieves issues that are not yet solved by performing a subquery to get all issue IDs from the SolveIssue table and then filtering the IssueQuery table to exclude those IDs. The resulting pending issues are processed into a list containing the issue ID, details, and creation timestamp. The class returns a JSON response with the count of pending issues and the corresponding list. To improve performance, consider using pagination for large result sets and optimizing query execution by selecting only necessary fields using with_entities().

6. ChatBot

- **/chat :**

The ChatbotAPI class handles incoming queries by first verifying if the query is empty. It then retrieves relevant documents from the database using a similarity search, generates context from the documents, and prepares messages for the chatbot by providing the context and the user's query. The chatbot response is generated and returned as part of the JSON response. Additionally, document references (subject, week, document type) are included to guide the user with useful resources. The class has a provision to store chat history in the database, though that part is currently commented out. Error handling is in place to return an error message if any issue occurs during the process. To optimize, consider caching results for frequently asked queries and refining document retrieval to improve response time.

SWAGGER



Authentication

POST `/login` User Login

POST `/signup` User Signup

POST `/refresh_token` Refresh Access Token

POST `/logout` User Logout

Admin

GET `/top-support-queries` Get top support queries in the past 7 days

GET `/query-detail/{query_id}` Get details of a specific query

POST `/solve-query/{query_id}` Solve a specific query

POST `/add-course` Add a new course

PUT `/edit-course/{course_id}` Edit a specific course

POST `/add-course-material/{course_id}` Add course material

PUT `/edit-course-material/{material_id}` Edit course material

Student

GET `/student-profile` Retrieve student profile details

GET `/student-dashboard` Retrieve student dashboard details

Instructor

POST `/lesson` Add a new lesson (Supplementary Material)

POST `/assignment` Add a new assignment

POST `/livesession` Add a new live session

Professor

GET `/pending-instructors` Retrieve all pending instructor requests

PUT `/approve-instructor/{request_id}` Approve or reject an instructor request

GET `/solved-issues` Retrieve all solved issues

GET `/pending-issues` Retrieve all pending issues

Chatbot

POST `/chatbot` Interact with the chatbot for educational hints and resources