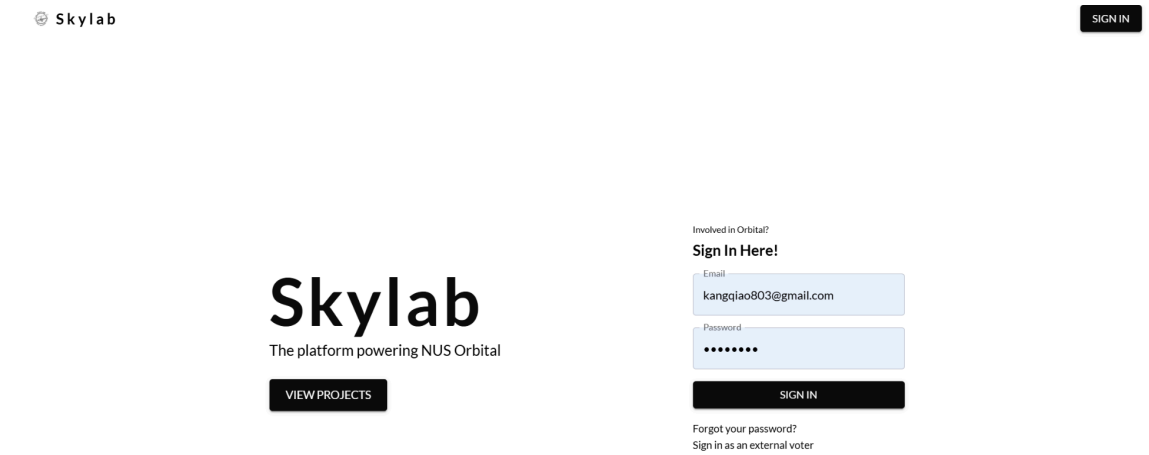
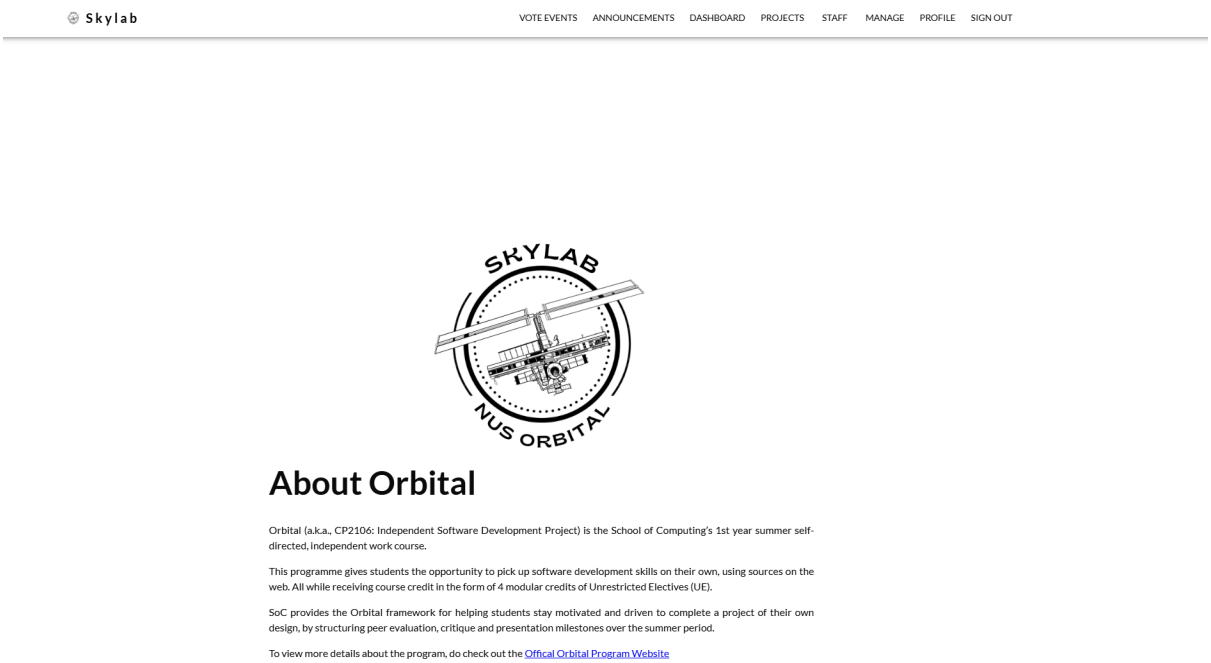
Screenshots of locally hosted Skylab





Name: Kang Qiao
Task selected: Add a forum

1. Understanding Process

From my understanding, the forum is an additional feature which can be added to the front page of the Skylab menu. This feature would require both frontend and backend implementation. Since Skylab uses the PERN stack, the frontend can be made using React and Next.JS. I would first create a frontend page while using as many of the functional components already created for consistency before creating any of my own. Here are the features I believe the forum should have:

Role-Based Access

- **Students**: Can create threads, reply to discussions, and search for mentors. They can post anonymously if they wish to as well.
- **Mentors**: Can participate in discussions, answer queries, and manage their profiles.
- **Advisors**: Can participate in discussions, answer queries, and manage their profiles.
- **Admins**: Can create threads, moderate content and manage users.

Forum Structure

- **Categories**:
    - **Project Discussions**: General project-related topics.
    - **Q&A**: Ask and answer questions about specific technical challenges.
    - **Mentor Matching**: Seek mentors based on specific skills or project needs.
- **Threads**: Individual discussions initiated within a category.
- **Posts**: Replies to threads for continued discussion.

Thread and Post Features

- Rich text editor for creating threads and posts.
- Markdown or WYSIWYG support for formatting.
- Ability to add attachments (e.g., code snippets, screenshots).

Search and Filter

- Full-text search for threads and posts.
- Filters based on categories, tags, or keywords.

Mentor Matching

- Keyword-based search to find mentors with relevant expertise.
- Display mentor profiles with information like skills, availability, and past contributions.

Moderation Tools

- Admins can edit or delete inappropriate content.
- Reporting system for flagging threads/posts.

For the backend, firstly I believe the database should be something like this:
1. Database Design for Postgresql

Tables/Collections:

- **Users**:
  - id, username, email, role, password_hash, avatar, bio, skills, availability, is_anonymous
- **Categories**:
  - id, name, description
- **Threads**:
  - id, title, content, category_id, created_by, created_at, is_anonymous, is_flagged
- **Posts**:
  - id, thread_id, content, created_by, created_at, is_flagged
- **Flags**:
  - id, flagged_item_id, item_type (thread/post), reason, flagged_by, status

2. Role-Based Access Control

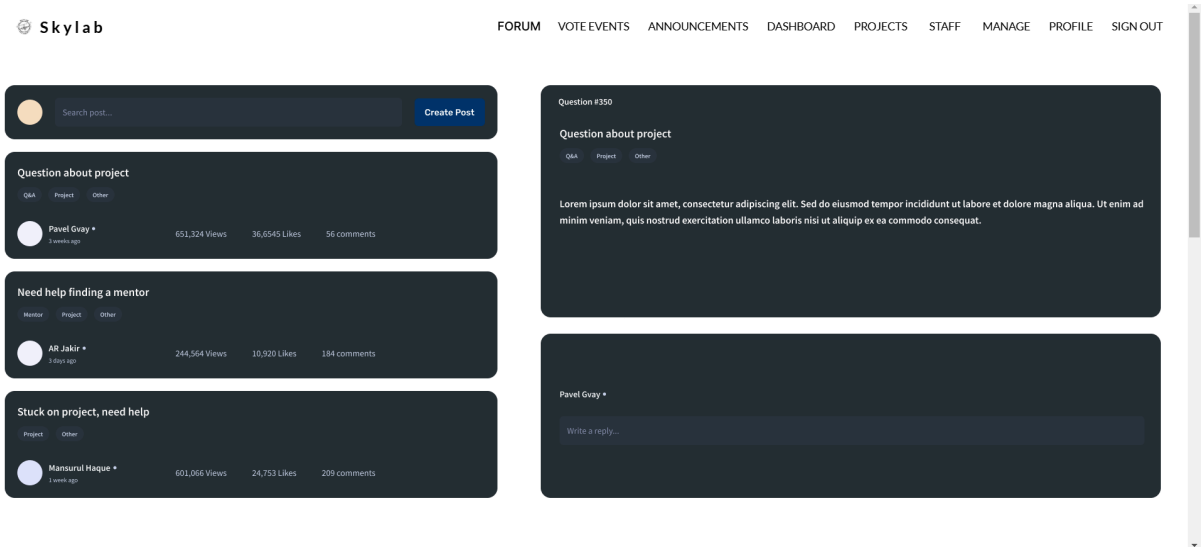
Define permissions for each role:

- **Students**: Can create, read threads/posts, search for mentors, and post anonymously.
- **Mentors**: Can read, reply, manage_profile, but cannot create categories or moderate.
- **Admins**: Can create, edit, delete any thread/post and manage users and flags.
- Advisors: Can read, reply any thread/post and review, approve projects and project milestones

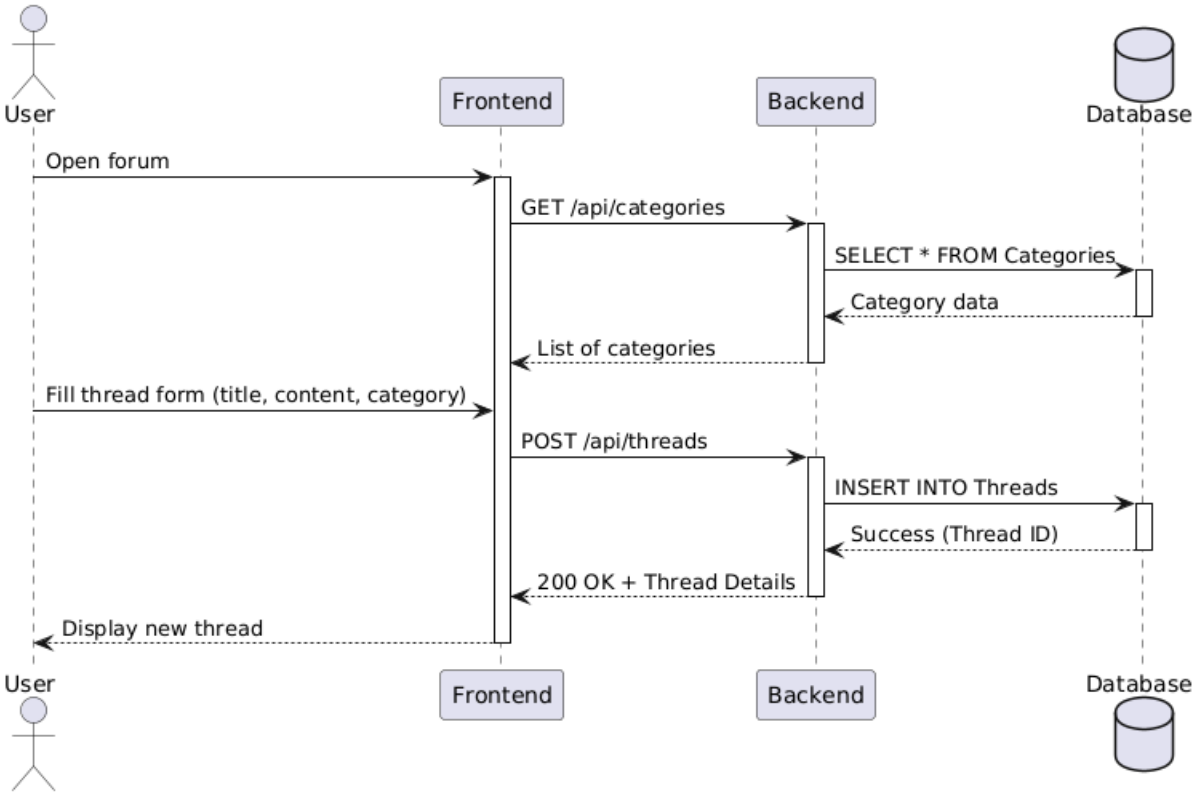
These permissions will be enforced in the middleware utilising Express and Node. Express and Node will also be used to facilitate the transfer of data from the frontend to the Postgresql database via API calls and SQL queries. Frontend sends API requests to the backend, the backend receives the request and interacts with SQL database via queries and lastly sends the response back to the frontend.

2. Design Diagrams

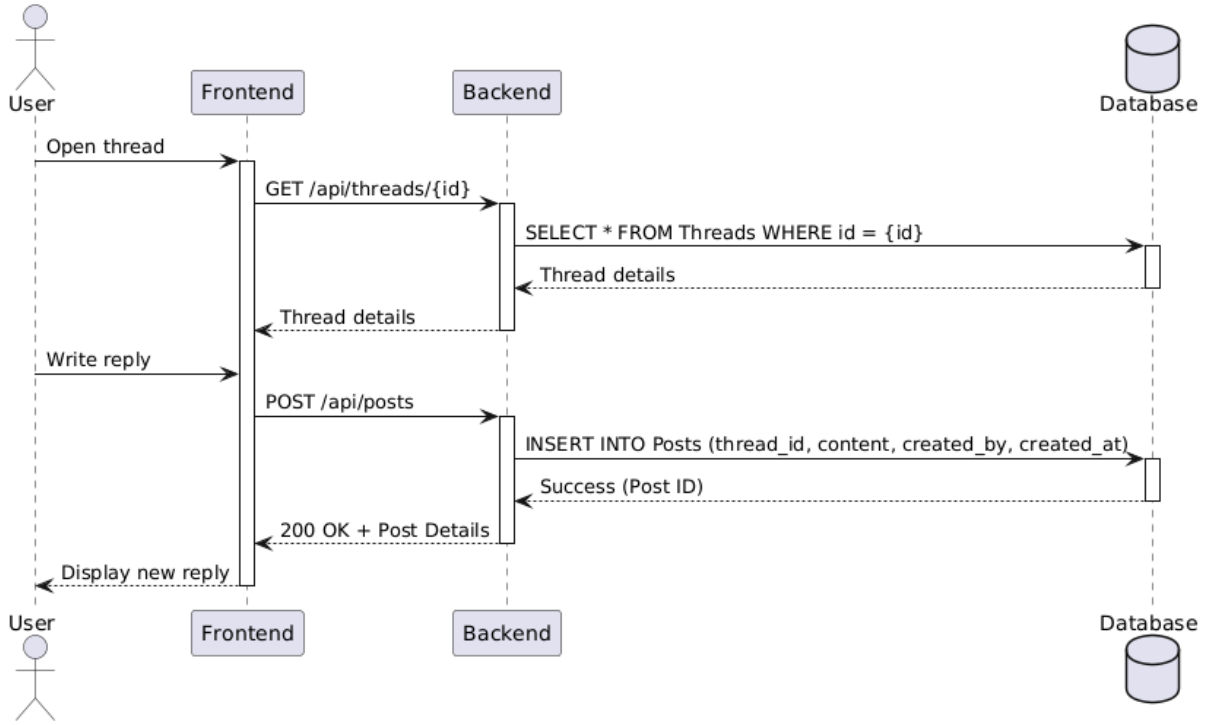
Basic UI mockup for the forum:



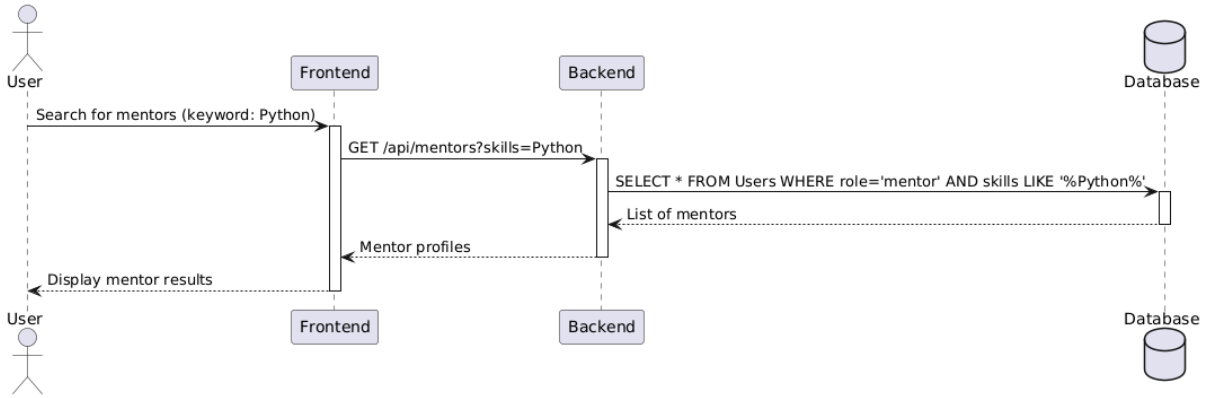
Sequence Diagram for creating a thread:



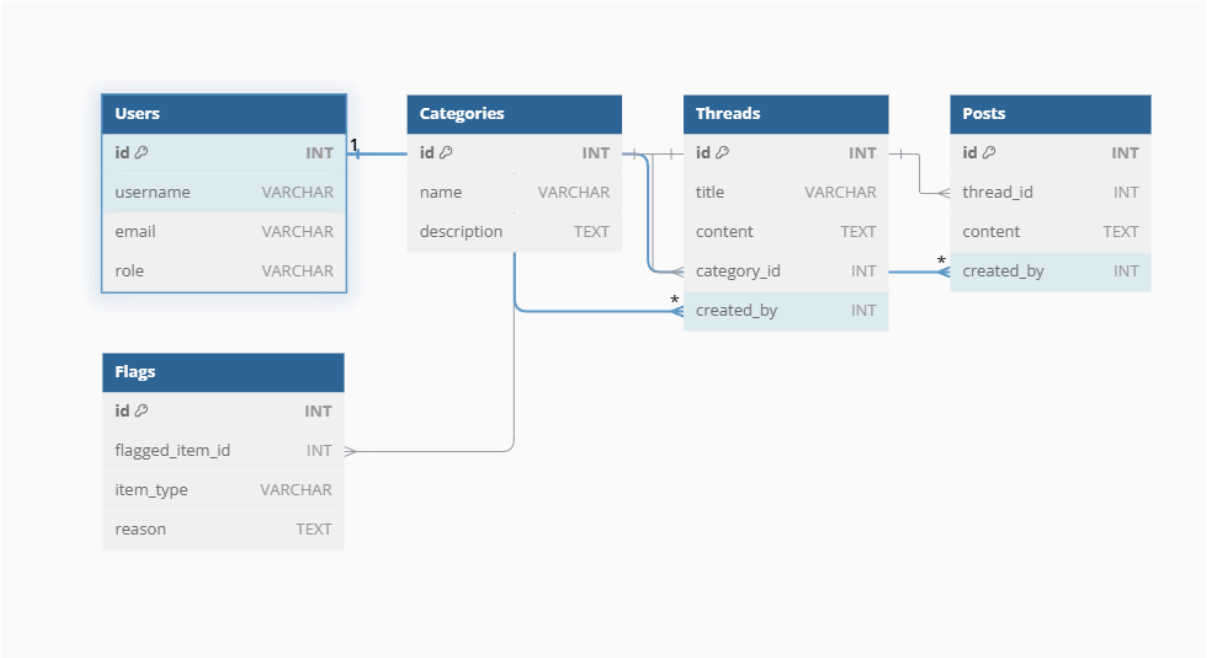
Sequence Diagram for replying to a thread(i.e creating a post):



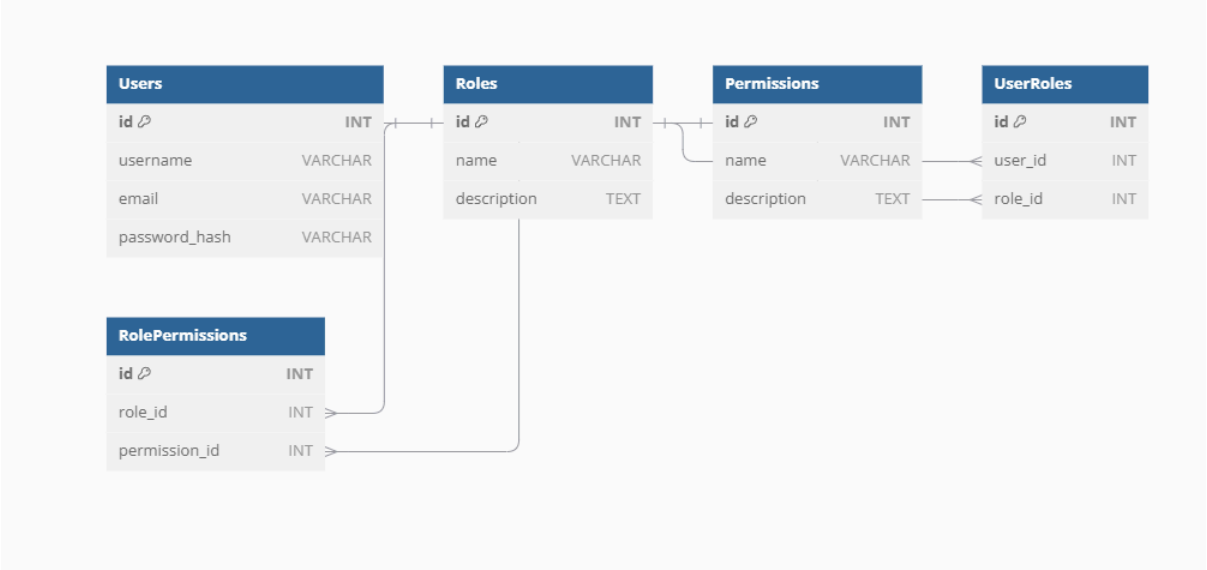
Sequence Diagram for searching mentors:



ER diagram from database



ER diagram for Role based Access Control



3. Draft Plan for Testing Features

a. Testing Goals

- Ensure all features (e.g., role-based access, CRUD operations, search, mentor matching, and moderation) work as intended.
- Identify and fix bugs or performance bottlenecks.
- Validate usability and security.

b. Testing Types

1. **Unit Testing**:
    - Test individual components (e.g., API endpoints, database queries).
    - Tools: Jest (backend), React Testing Library (frontend).
2. **Integration Testing**:
    - Ensure seamless interaction between components (frontend ↔ backend ↔ database).
    - Example: Test if a POST request to create a thread successfully updates the database and returns the expected response.
3. **End-to-End (E2E) Testing**:
    - Test workflows from the user's perspective (e.g., creating a thread, replying, searching for mentors).
    - Tools: Cypress
4. **Performance Testing**:
    - Evaluate system behavior under load (e.g., 100 concurrent users).
    - Tools: Cypress

c. Testing Workflow

1. **Backend Testing**:
    - Test database queries for edge cases (e.g., empty inputs, invalid data).
    - Mock database interactions to test API endpoints independently.
2. **Frontend Testing**:
    - Validate UI components and user interactions (e.g., input fields, buttons).
    - Test if error messages are displayed for invalid inputs.
3. **Integration Testing**:
    - Use tools like Postman to simulate API requests and verify responses.
    - Ensure data consistency between frontend, backend, and database.
4. **End-to-End Testing**:
    - Simulate real-world scenarios:
        - Login as different roles (student, mentor, admin).
        - Create a thread, reply, flag content, search for mentors.

4. Development Plan

- **Phase 1: Planning**
    - Finalize requirements and system design (UI mockups, ER diagrams, sequence diagrams).
- **Phase 3: Frontend Development**
    - Build React components for UI (main forum page, thread view, post editor, mentor matching).

- **Phase 2: Backend Development**
    - Set up PostgreSQL database schema.
    - Develop API endpoints for CRUD operations (threads, posts, users, categories).
    - Implement role-based access control.
    - Implement real-time notifications (e.g., for new replies).
    - Add search and filtering functionality.
    - Integrate to the frontend with backend API endpoints.
- **Phase 4: Integration**
    - Connect frontend and backend.
    - Test full workflows (e.g., posting a thread, searching mentors).
- **Phase 5: Testing**
    - Conduct unit, integration, and E2E tests.
    - Fix bugs and optimize performance.
- **Phase 6: Deployment**

5. Relevant Technologies / Libraries

Backend

- **Node.js**: JavaScript runtime for building server-side logic.
- **Express.js**: Framework for routing, middleware, and RESTful APIs.
- **pg (node-postgres)**: PostgreSQL client for Node.js to interact with the database.
- **JWT (jsonwebtoken)**: Authentication and role-based access control.
- **bcrypt.js**: Secure password hashing.

Frontend

- **React.js**: Component-based library for building UIs.
- **React Router**: For frontend routing.

Database

- **PostgreSQL**: Relational database for storing users, threads, posts, and categories.

Testing

- **Jest**: Unit testing for backend and frontend components.
- **React Testing Library**: Testing React components and interactions.
- **Cypress**: End-to-end testing for real-world workflows.
- **Postman**: Manual API testing.

Other

- **Socket.IO**: Real-time communication for notifications.
- **Docker**: For containerizing and deploying the application.
- **Nginx**: As a reverse proxy for production.

6. Justifications for Good SE Principles

Modularity

- The system is divided into independent components:
  - **Backend** handles API logic and database interaction.
  - **Frontend** focuses on user interface and interactions.
- Encourages reuse and maintainability.

Scalability

- Using **PostgreSQL** ensures scalable data storage with advanced features like indexing and full-text search.
- Real-time updates (e.g., notifications) are handled efficiently with **Socket.IO**.

Security

- **JWT** for secure authentication and role-based access control.
- Input sanitization using libraries like sanitize-html prevents XSS attacks.
- Parameterized queries using pg prevent SQL injection.

Performance

- Connection pooling with pg optimizes database interactions.

Testability

- Comprehensive testing plan (unit, integration, E2E) ensures robust code.
- Mocking tools like Jest allow independent testing of backend and database.

Usability

- Clean and intuitive UI mockups improve user experience.
- Real-time updates and notifications enhance responsiveness.

Maintainability

- Follows RESTful API conventions, making endpoints predictable and easier to extend.
- Consistent use of environment variables for configuration.