VIETNAM INTERNATIONAL UNIVERSITY- HO CHI MINH CITY
INTERNATIONAL UNIVERSITY

WEB APPLICATION DEVELOPMENT PROJECT
SCHOOL FORUM PROJECT

BY
Nguyễn Quốc Trạng - ITCSIU21239
Trần Công Bằng - BEBEIU21189

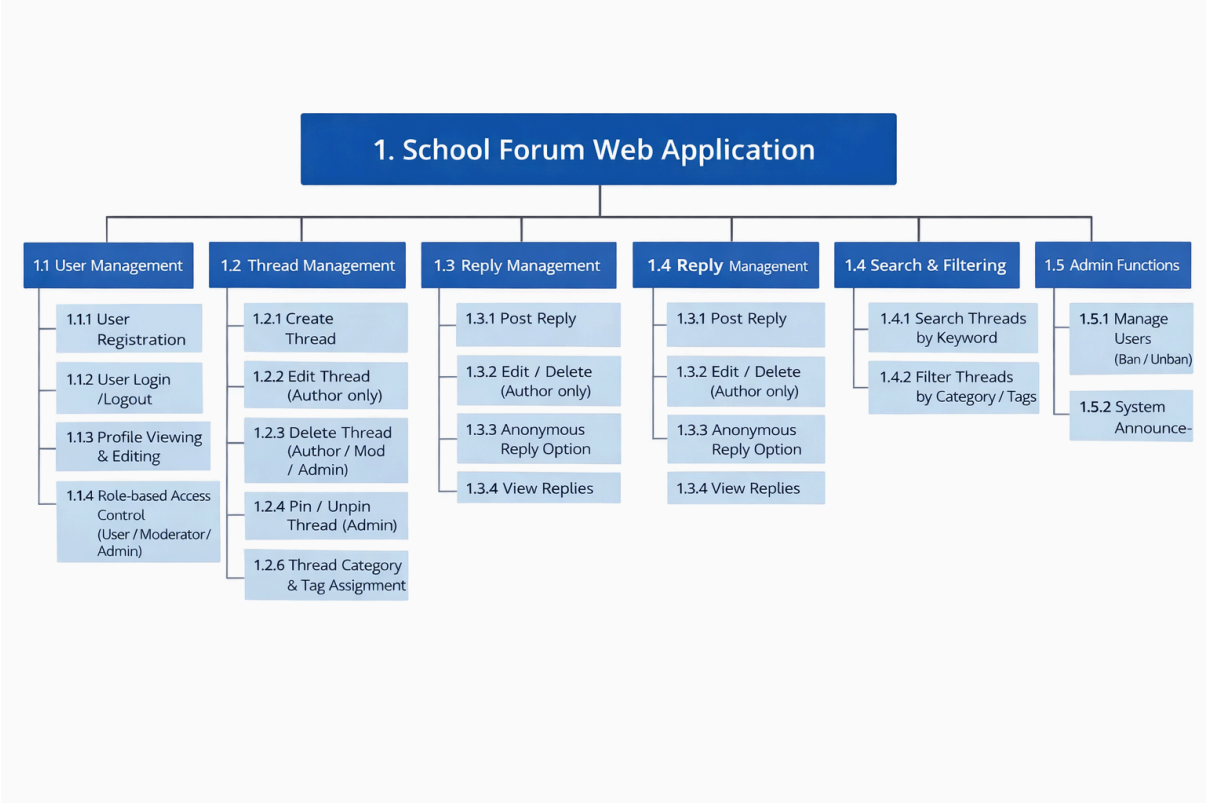
Table of Contents

I. Project Overview

1. Introduction

The School Forum Web Application is a comprehensive platform developed to support academic collaboration within educational institutions. It enables students and lecturers to engage in meaningful discussions, create and reply to threads, and share knowledge efficiently. The system incorporates role-based access control, thread organization and a responsive interface for both desktop and mobile devices. Built using Java Spring Boot for the backend and React with Vite for the frontend, the application offers a secure, interactive, and user-friendly environment that enhances communication and learning among its users.

2. Project Overview



3. Development Process

Our team has chosen to follow the Waterfall method for our project. The development process is organized into sequential phases, starting with requirement analysis, followed by system design, implementation, testing, deployment, and maintenance. Each phase is completed before moving on to the next, ensuring that the system is carefully planned and documented. This approach allows us to have a clear roadmap from start to finish and guarantees that each module meets the defined specifications before integration. The final product will provide users with a stable, fully functional forum system to interact with efficiently.

4. Development Environment

Programming Languages
The School Forum Web Application is built using a modern full-stack technology stack. The backend is developed in Java 21, leveraging the latest language features and performance improvements. The frontend is implemented using JavaScript with React 18.2.0, providing a dynamic and responsive user interface. The application follows a clear separation of concerns between the client-side and server-side components.

Frameworks & Libraries

Backend Technologies

- Spring Data JPA: Database abstraction and ORM capabilities with Hibernate as the persistence provider.
- Spring Security: Handles authentication and authorization with JWT (JSON Web Token) using JJWT
- Lombok : Reduces boilerplate code with annotations.
- Spring Boot Validation: Ensures input validation.
- Spring Boot DevTools: Enables hot reload during development.

Frontend Technologies
The frontend is built with React and React DOM for interactive user interfaces.

- React Router DOM: Client-side routing and navigation.
- Axios: Handles HTTP requests to the backend API.
- Tailwind CSS: Utility-first styling for responsive design.
- Vite: Fast development server and optimized production builds.
- PostCSS & Autoprefixer: CSS processing and compatibility.

Architecture
The application follows a client-server architecture with a clear separation between frontend and backend.
Backend: RESTful API and MVC (Model-View-Controller) layered architecture:

- Model Layer: Entity classes (User, Thread, Reply, Category, etc.) representing domain models using JPA annotations.
- DAO Layer: Repository interfaces extending Spring Data JPA for database access.

- Service Layer: Business logic, transactions, and data transformation.
- Controller Layer: REST API endpoints handling HTTP requests/responses using DTOs.

Frontend: Component-based architecture organized into pages, components, and contexts for state management. Uses JWT-based stateless authentication.

Database
The application uses MySQL as the relational database management system.

- Database connectivity via MySQL Connector/J
- JPA with Hibernate as the ORM framework.
- Database schema is managed automatically using Hibernate DDL auto-update.
- Spring Data JPA repositories provide pagination, sorting, and custom queries.

Development Tools

- Backend: Maven for build automation and dependency management.
- Frontend: npm for JavaScript dependencies and development scripts.
- IDE: IntelliJ IDEA, Eclipse, or VS Code.
- Version Control: Git for collaborative development and code management.
- Maven compiler plugin is configured for Java 21.
- Spring Boot Maven plugin for application packaging and execution.

II. Requirement Analysis and Design

1. Requirement Analysis

Purpose
The purpose of this document is to specify the software requirements for the School Forum Web Application. This SRS is intended for lecturers, students, developers, and evaluators involved in the design, development, and assessment of the system.
Scope
The School Forum Web Application is an online discussion platform designed for educational environments. It enables students to create discussion threads, post replies and search content, . Moderators and administrators manage content, users, and categories to ensure a safe and organized forum.

Definitions, Acronyms, and Abbreviations

- SRS: Software Requirements Specification
- UC: Use Case
- JWT: JSON Web Token
- User: Registered student or forum participant
- Moderator: User with content management privileges
- Admin: System administrator

2. Overall Description

Product Perspective

The system is a web-based application consisting of a frontend client and a backend RESTful API connected to a relational database.

Product Functions

- User authentication and profile management
- Forum category and thread management
- Reply system
- Moderation and administration features

User Classes and Characteristics

- User: Can browse, post threads, reply, and manage own profile
- Moderator: Can manage threads and users
- Admin: Full system control including categories and roles

Operating Environment

- Web browser (Chrome, Firefox, Edge)
- Backend server (Java Spring Boot)
- Database (MySQL)

Design and Implementation Constraints

- Role-based access control
- Secure authentication using JWT
- Compliance with academic use policies

---

3. Specific Requirements

Functional Requirements

The system shall support all use cases defined in Section 3.3.

Non-Functional Requirements

- The system shall respond to user actions within 2 seconds under normal load.
- The system shall ensure secure access control.
- The system shall support concurrent users.

SRS Use Case Tables

This section defines all system use cases in formal SRS format, covering UC-01 to UC-18.

Field	Description

UC-01 – Register	
Primary Actor	User
Description	Create a new user account
Preconditions	User not authenticated
Trigger	Submit registration form
Main Flow	Enter info → Validate → Save user → Login
Postcondition	Account created
UC-02 – Login	
Primary Actor	User
Description	Authenticate user
Preconditions	Account exists
Trigger	Submit login form
Main Flow	Validate credentials → Issue token
Postcondition	User logged in
UC-03 – View Categories	
Primary Actor	User
Description	View forum categories
Preconditions	None
Trigger	Open homepage
Main Flow	Load categories
Postcondition	Categories displayed
UC-04 – View Threads	
Primary Actor	User
Description	Browse threads
Preconditions	Category exists
Trigger	Open category
Main Flow	Load threads
Postcondition	Threads displayed
UC-05 – View Thread Details	
Primary Actor	User
Description	View thread and replies
Preconditions	Thread exists

Trigger	Select thread
Main Flow	Load thread & replies
Postcondition	Content displayed
UC-06 – Search Threads	
Primary Actor	User
Description	Search threads
Preconditions	Forum accessible
Trigger	Submit search
Main Flow	Query → Show results
Postcondition	Results shown
UC-07 – Create Thread	
Primary Actor	User
Description	Create new thread
Preconditions	User authenticated
Trigger	Submit thread form
Main Flow	Validate → Save thread
Postcondition	Thread created
UC-08 – Edit Thread	
Primary Actor	User
Description	Edit thread content
Preconditions	Ownership or permission
Trigger	Click edit
Main Flow	Update thread
Postcondition	Thread updated
UC-09 – Delete Thread	
Primary Actor	User
Description	Delete thread
Preconditions	Permission granted
Trigger	Confirm delete
Main Flow	Remove thread
Postcondition	Thread deleted
UC-10 – View Replies	
Primary Actor	User

Description	View replies
Preconditions	Thread exists
Trigger	Open thread
Main Flow	Load replies
Postcondition	Replies shown
UC-11 – Create Reply	
Primary Actor	User
Description	Reply to thread
Preconditions	Authenticated, thread unlocked
Trigger	Submit reply
Main Flow	Save reply
Postcondition	Reply added
UC-12 – Ban User	
Primary Actor	Moderator
Description	Ban a user
Preconditions	Target not admin
Trigger	Click ban
Main Flow	Update user status
Postcondition	User banned
UC-13 – Edit Own Profile	
Primary Actor	User
Description	Edit profile
Preconditions	Authenticated
Trigger	Submit changes
Main Flow	Update profile
Postcondition	Profile updated

4. System Design

Architectural Overview

The School Forum Web Application follows a three-tier architecture to ensure separation of concerns, scalability, and maintainability. The system is designed as a web-based client–server application where the presentation layer communicates with the application layer through RESTful APIs, and the application layer interacts with the data layer for persistent storage.

The architecture supports multiple user roles, including User, Moderator, and Administrator, with role-based access control enforced at the application layer using JWT authentication.
Presentation Layer (Frontend)
The Presentation Layer provides the user interface for interacting with the forum system. It is responsible for displaying data, collecting user input, and sending requests to the backend server.
Responsibilities:

- Display forum pages (categories, threads, replies)
- Provide authentication interfaces (login, register)
- Handle user actions (create thread, reply, edit profile)
- Store and attach JWT tokens to authenticated requests

Technologies (example):

- HTML, CSS, JavaScript
- Frontend framework (e.g., React)

Application Layer (Backend / API Layer)

The Application Layer handles all business logic and acts as an intermediary between the frontend and the database. It exposes RESTful APIs for all system functionalities defined in the Requirement Analysis phase.
Responsibilities:

- Process client requests
- Validate input data
- Enforce authorization and role-based access control
- Execute business rules (locking threads, banning users)
- Generate and validate JWT tokens
- Coordinate communication with the database

Technologies (example):

- Spring Boot (Java)
- REST API
- JWT Authentication

Data Layer (Database Layer)

The Data Layer is responsible for persistent data storage. It stores all forum-related data, including users, threads, replies, categories and tags.
Responsibilities:

- Store and retrieve application data
- Maintain relationships between entities
- Ensure data integrity and consistency

Technologies (example):

- MySQL (Relational Database)
- JPA / Hibernate for ORM

System Components
The major components of the system include:

- Authentication Component
    - Handles user registration and login
    - Issues and validates JWT tokens
- User Management Component
    - Manages user profiles
    - Handles banning and role management
- Forum Management Component
    - Manages categories, threads, and replies
    - Supports thread locking and moderation

Each component communicates through well-defined service interfaces to maintain modularity.

Security Architecture
Security is enforced primarily at the Application Layer using JWT-based authentication and role-based authorization.
Security mechanisms include:

- Password hashing using BCrypt
- JWT tokens for stateless authentication
- Role validation (USER, MODERATOR, ADMIN)

- Access restrictions for protected endpoints
- Input validation to prevent invalid or malicious requests

Deployment View
The system is deployed as a web application with the following configuration:

- Client browser accesses the frontend application
- Frontend communicates with backend via HTTPS
- Backend server communicates with the database
- Database server stores all persistent data

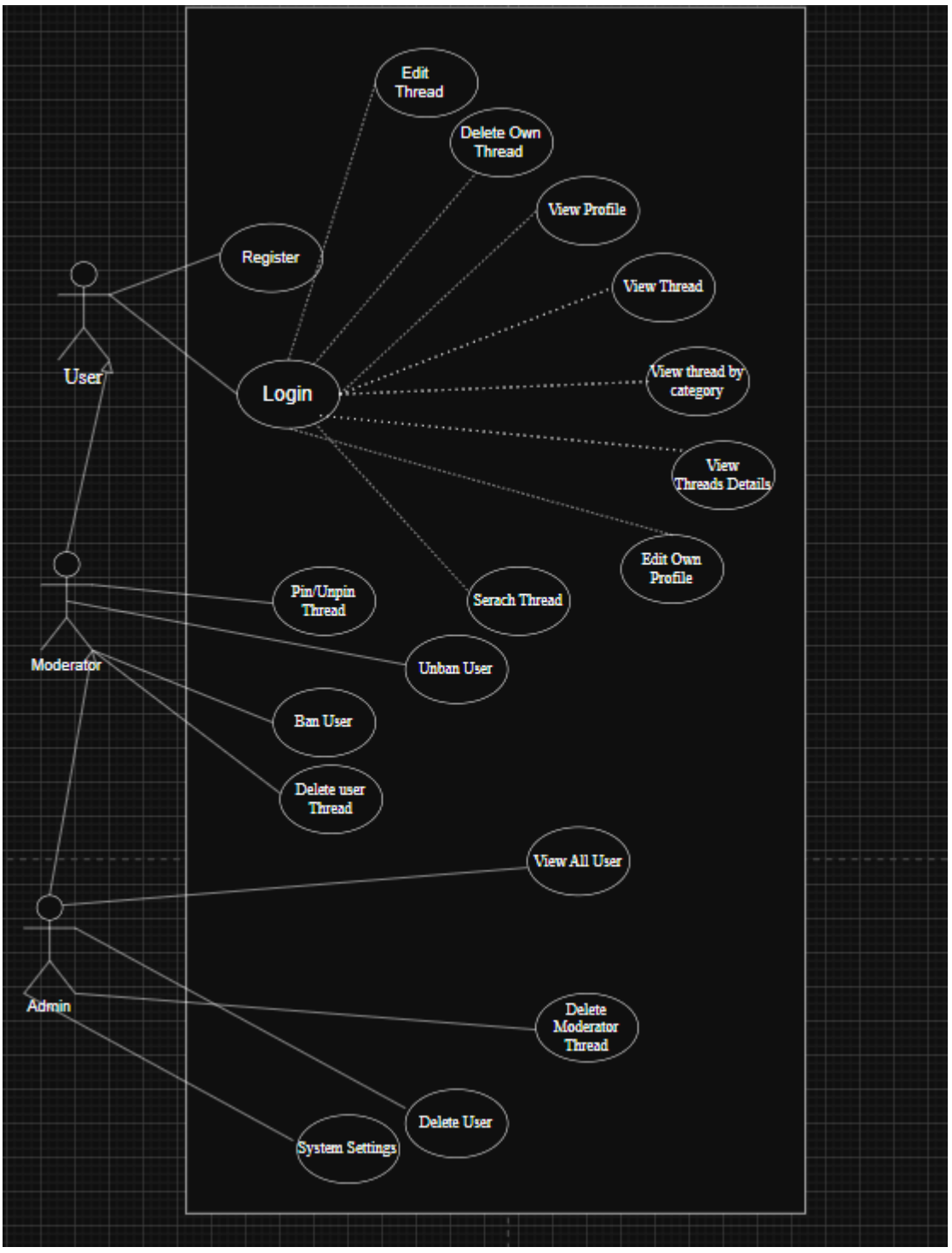
This deployment model allows the system to scale independently across layers and supports future expansion.

Architectural Benefits
The chosen architecture provides several advantages:

- Scalability: Each layer can be scaled independently
- Maintainability: Clear separation of concerns
- Security: Centralized authentication and authorization
- Extensibility: New features can be added with minimal impact
- Reusability: APIs can be reused by other clients (e.g., mobile apps)

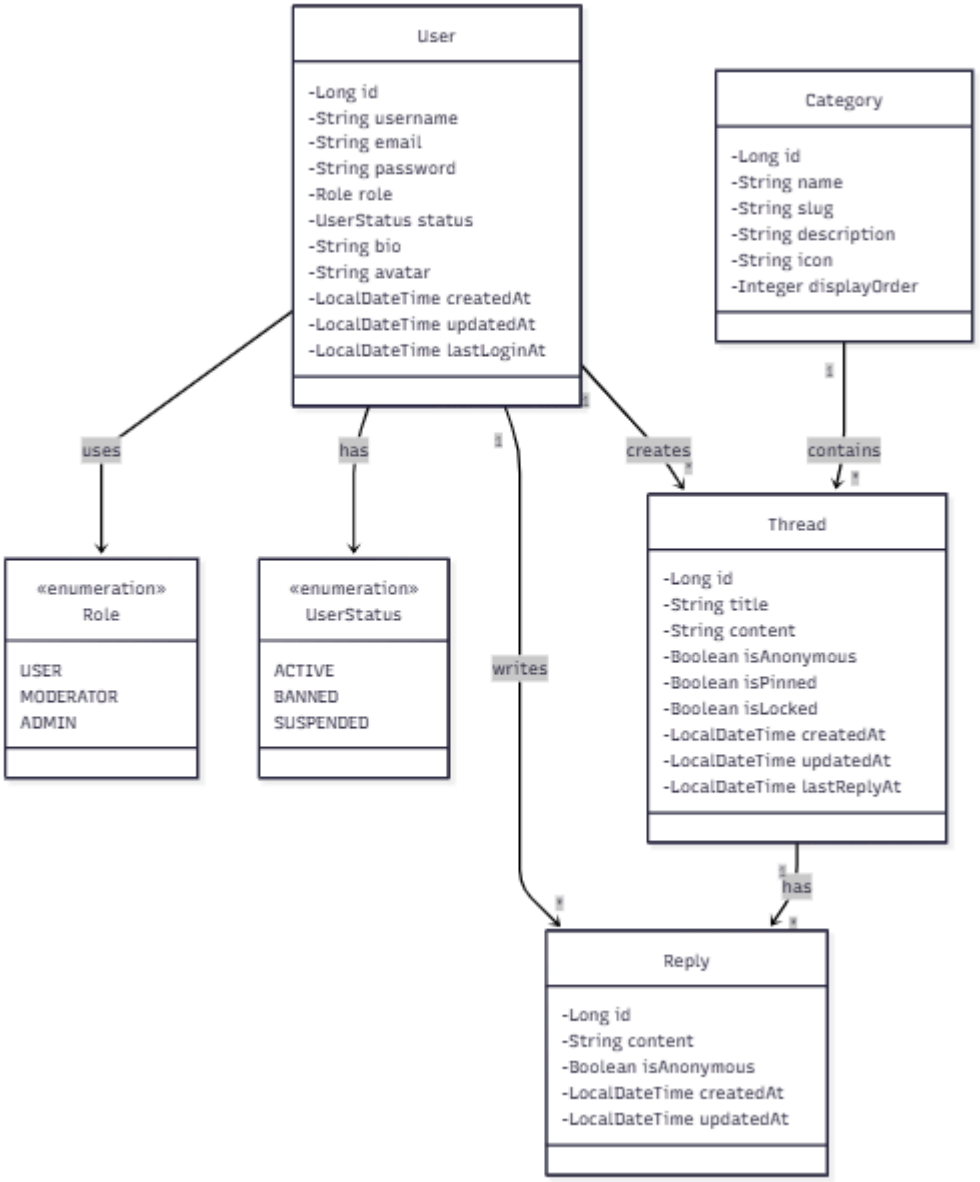
5. High-Level Use Case Diagram

This diagram provides a high-level overview of the School Forum Web Application, illustrating the main actors (User, Moderator, Administrator) and their interactions with the system. It summarizes all core functionalities identified during the Requirement Analysis phase.


Edit Thread
Delete Own Thread
View Profile
Register
View Thread
User
Login
View thread by category
View Threads Details
Edit Own Profile
Pin/Unpin Thread
Serach Thread
Moderator
Unban User
Ban User
Delete user Thread
View All User
Admin
Delete Moderator Thread
Delete User
System Settings

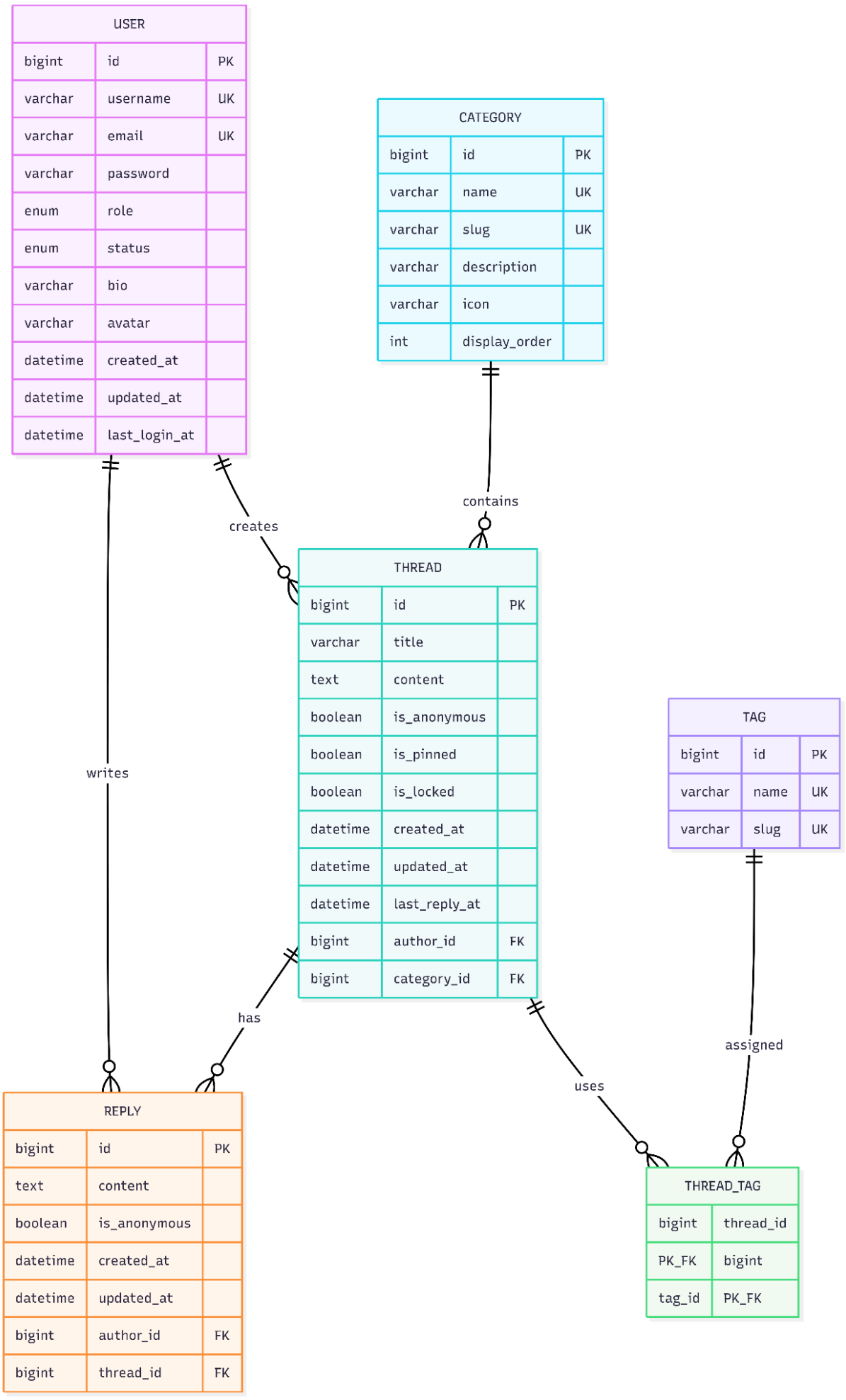
6. Class Diagram

The class diagram illustrates the object-oriented structure of the School Forum Web Application. It presents the main domain classes, their key attributes, and the relationships between them. The diagram focuses on core entities such as users, threads, replies and categories, and demonstrates how these classes collaborate to support the system’s functional requirements.


User
-Long id
-String username
-String email
-String password
-Role role
-UserStatus status
-String bio
-String avatar
-LocalDateTime createdAt
-LocalDateTime updatedAt
-LocalDateTime lastLoginAt
Category
-Long id
-String name
-String slug
-String description
-String icon
-Integer displayOrder
uses
has
writes
creates
contains
«enumeration»
Role
USER
MODERATOR
ADMIN
«enumeration»
UserStatus
ACTIVE
BANNED
SUSPENDED
Thread
-Long id
-String title
-String content
-Boolean isAnonymous
-Boolean isPinned
-Boolean isLocked
-LocalDateTime createdAt
-LocalDateTime updatedAt
-LocalDateTime lastReplyAt
has
Reply
-Long id
-String content
-Boolean isAnonymous
-LocalDateTime createdAt
-LocalDateTime updatedAt

7. ER Diagram

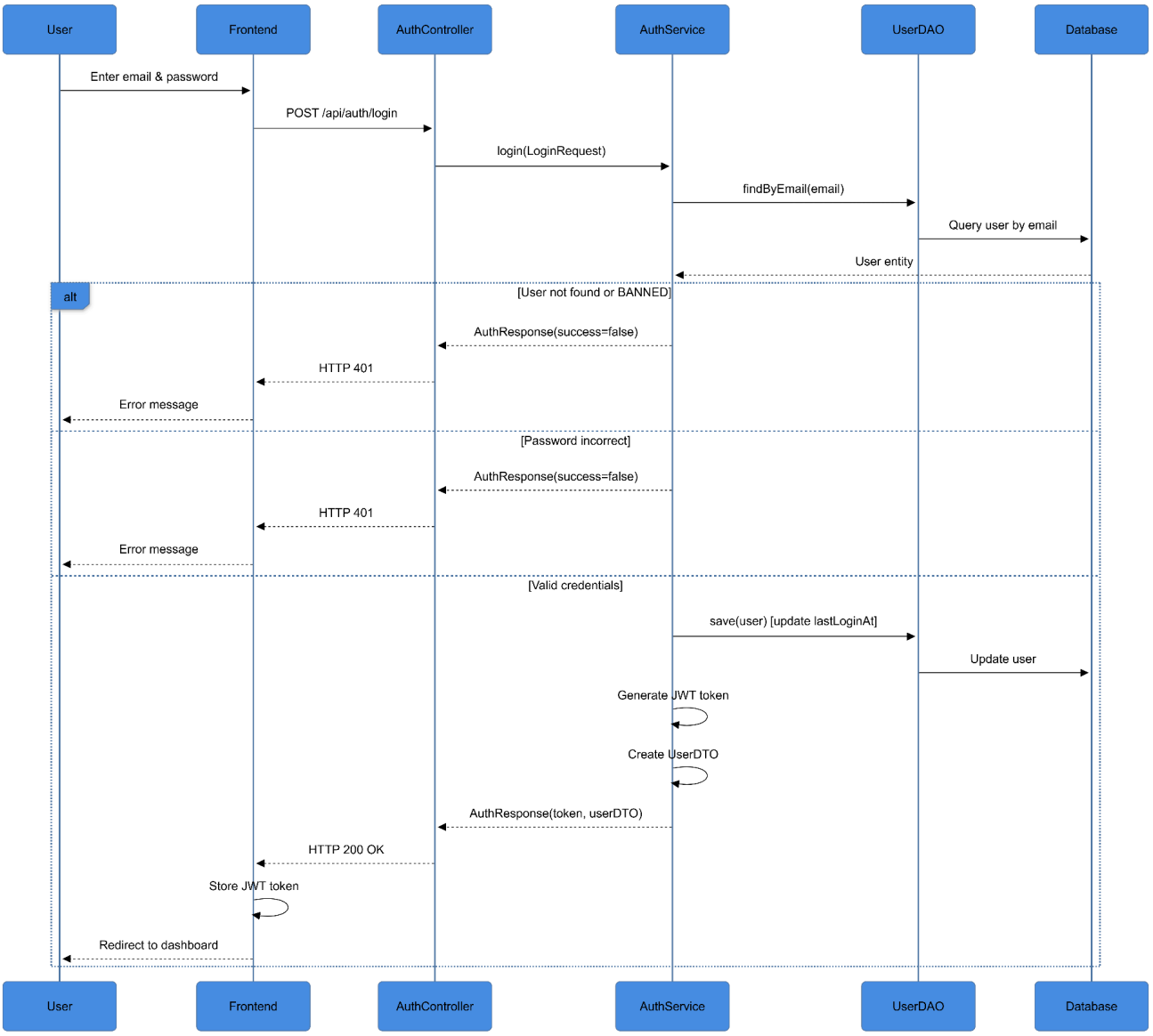
The database schema is modeled through an Entity-Relationship Diagram consisting of eight entities that capture the core domain concepts: user management, content creation (threads and replies), categorization, tagging. The diagram specifies cardinality constraints including one-to-many relationships (User-Thread, Category-Thread, Thread-Reply) and a many-to-many relationship (Thread-Tag) resolved through a junction table, ensuring data integrity and supporting the application's functional requirements.


USER
bigint id PK
varchar username UK
varchar email UK
varchar password
enum role
enum status
varchar bio
varchar avatar
datetime created_at
datetime updated_at
datetime last_login_at
CATEGORY
bigint id PK
varchar name UK
varchar slug UK
varchar description
varchar icon
int display_order
THREAD
bigint id PK
varchar title
text content
boolean is_anonymous
boolean is_pinned
boolean is_locked
datetime created_at
datetime updated_at
datetime last_reply_at
bigint author_id FK
bigint category_id FK
TAG
bigint id PK
varchar name UK
varchar slug UK
REPLY
bigint id PK
text content
boolean is_anonymous
datetime created_at
datetime updated_at
bigint author_id FK
bigint thread_id FK
THREAD_TAG
bigint thread_id
PK_FK bigint
tag_id PK_FK
creates
contains
writes
has
uses
assigned

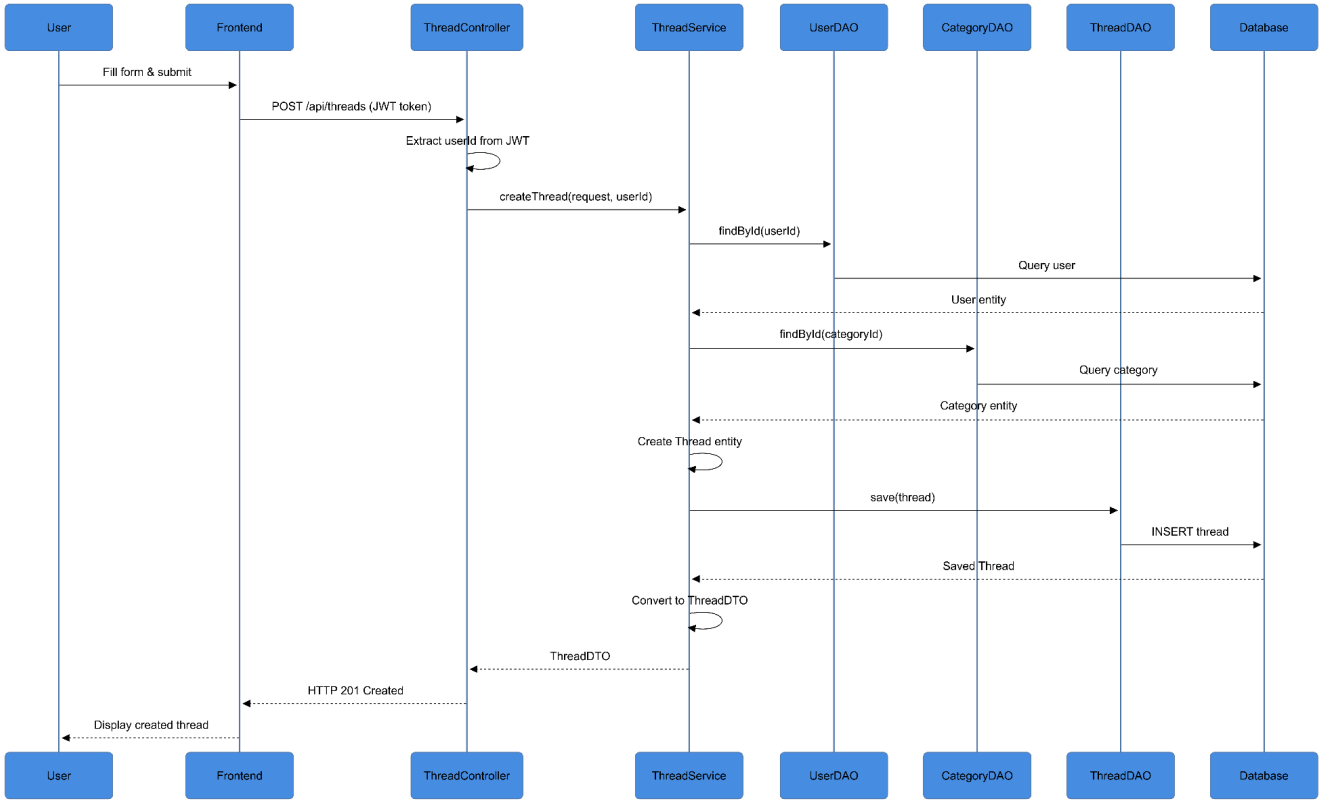
8. Sequence Diagram

Sequence diagrams are provided to illustrate the dynamic interactions and message flow between system components during critical operations. Three key sequence diagrams are presented: User Login, Create Thread and Search Threads, representing the most important workflows in the forum application. These diagrams demonstrate how requests flow through the layered architecture from Controllers to Services, then to DAOs, and finally to the database, with responses propagating back through the same layers. The selected scenarios provide comprehensive coverage of the system's behavior patterns while maintaining documentation clarity and avoiding unnecessary overhead. Each diagram shows the temporal sequence of operations, error handling paths, and the collaboration between different architectural layers.

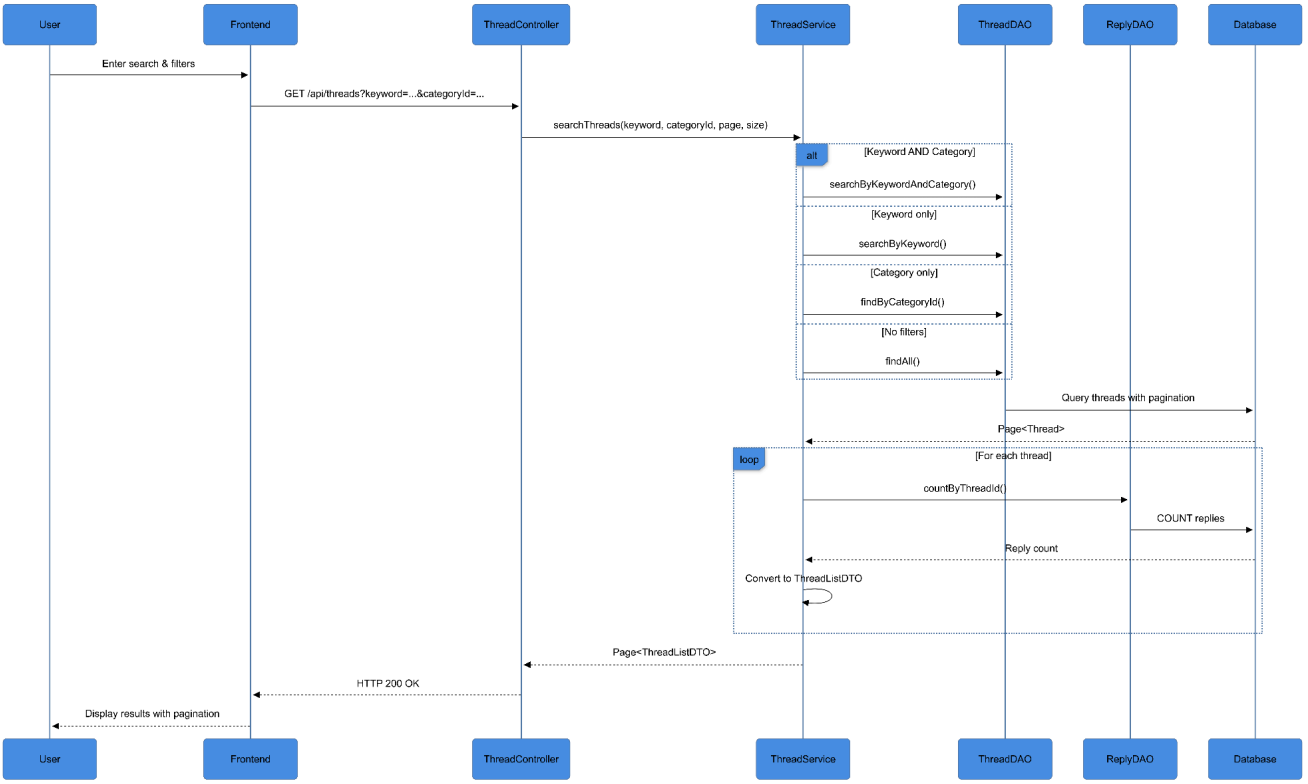
Sequence Diagram 1: User Login Flow



Sequence Diagram 2: Create Thread Flow



Sequence Digram 3: Search Thread Flow



III. Implementation

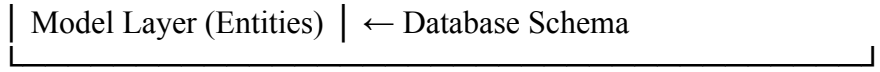
1. System Architecture Overview

The School Forum Web Application follows a **layered architecture pattern** (also known as the N-tier architecture), which separates concerns into distinct layers to promote maintainability, scalability, and testability. The system is divided into two main components: a backend REST API server and a frontend single-page application.

1.1 Backend Architecture

The backend implements a four-layer architecture pattern:

```
┌─────────────────────────────────────────┐
│  Controller Layer (REST API)            │ ← HTTP Request/Response
├─────────────────────────────────────────┤
│  Service Layer (Business Logic)         │ ← Business Rules & Validation
├─────────────────────────────────────────┤
│  DAO Layer (Data Access)                │ ← Database Operations
└─────────────────────────────────────────┘
```



Controller Layer (com.schoolforum.controller): This layer handles HTTP requests and responses. Controllers receive incoming REST API requests, validate input using DTOs (Data Transfer Objects), delegate business logic to service classes, and return appropriate HTTP responses. The main controllers include: - AuthController: Handles authentication endpoints (/api/auth/*) - ThreadController: Manages thread-related operations (/api/threads/*) - ReplyController: Handles reply operations (/api/threads/{id}/replies and /api/replies) - CategoryController: Manages forum categories (/api/categories/*) - UserController: Handles user profile and management (/api/users/*) - AdminController: Provides administrative functions (/api/admin/*)

Service Layer (com.schoolforum.service): This layer contains business logic and orchestrates data operations. Services are responsible for: - Validating business rules - Coordinating between multiple DAOs - Converting between entity models and DTOs - Implementing transaction management - Handling authorization checks

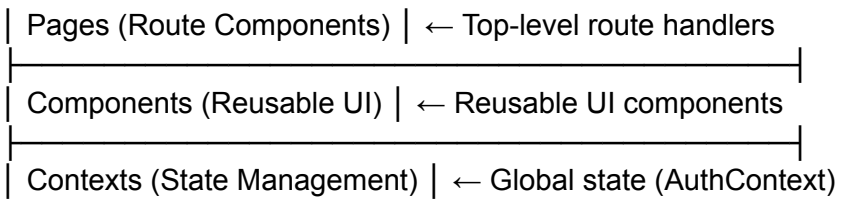
The key service classes are: - AuthService: Manages user registration, login, and authentication - ThreadService: Handles thread creation, updates, deletion, and retrieval with pagination - ReplyService: Manages reply creation and deletion

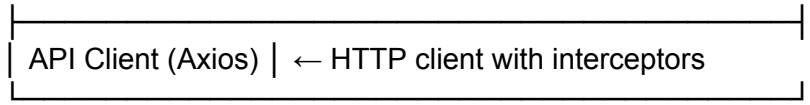
DAO Layer (com.schoolforum.dao): The Data Access Object layer uses Spring Data JPA repositories to interact with the database. These interfaces extend JpaRepository and provide CRUD operations plus custom query methods. The DAOs include: - UserDAO: User entity operations - ThreadDAO: Thread entity operations with custom queries for filtering - ReplyDAO: Reply entity operations - CategoryDAO: Category entity operations - TagDAO: Tag entity operations

Model Layer (com.schoolforum.model): This layer contains JPA entity classes that map to database tables. Entities use JPA annotations to define relationships, constraints, and table mappings. The main entities are: - User: Represents forum users with roles and status - Thread: Represents discussion threads - Reply: Represents replies to threads - Category: Represents forum categories - Tag: Represents thread tags

1.2 Frontend Architecture

The frontend follows a component-based architecture using React. The application structure is organized as follows:





Pages (src/pages/): Each page component corresponds to a route in the React Router configuration. Pages include: - Home.jsx: Displays categories and recent threads - Login.jsx and Register.jsx: Authentication pages - ThreadPage.jsx: Displays thread details and replies - CreateThread.jsx and EditThread.jsx: Thread creation and editing - Category.jsx: Displays threads in a category - UserManagement.jsx: Admin panel for user management - Settings.jsx: User settings page

Components (src/components/): Reusable UI components organized by purpose: - layout/: Header, Footer, Navbar - thread/: ThreadRow, ReplyBox - common/: CategoryCard, Loading

State Management: The application uses React Context API for global state management. AuthContext (src/contexts/AuthContext.jsx) manages authentication state, storing user information and JWT tokens in localStorage.

API Integration: All API calls are centralized in src/api/index.js using Axios. The Axios instance is configured with: - Base URL: http://localhost:8080/api - Request interceptor: Automatically attaches JWT token from localStorage - Response interceptor: Handles 401 errors by redirecting to login

1.3 Frontend-Backend Communication

Communication between frontend and backend follows the REST architectural style:

1. Request Flow: Frontend makes HTTP requests to backend REST endpoints using Axios. The JWT token is automatically included in the Authorization header via Axios interceptors.

2. Response Flow: Backend processes requests, executes business logic, and returns JSON responses. The frontend receives responses and updates the UI accordingly.
3. Authentication: JWT tokens are stored in browser localStorage after successful login. Every authenticated request includes the token in the Authorization: Bearer <token> header.

4. CORS Configuration: Cross-Origin Resource Sharing is configured in SecurityConfig.java to allow requests from http://localhost:3000 and http://localhost:5173 (Vite dev server).

2. Technologies and Tools Used

2.1 Backend Technologies

Programming Language: Java 17 (as specified in pom.xml)

Framework: Spring Boot 3.2.0 - spring-boot-starter-web: REST API and MVC support - spring-boot-starter-data-jpa: JPA/Hibernate ORM - spring-boot-starter-security: Authentication and authorization - spring-boot-starter-validation: Input validation

Database: MariaDB/MySQL 8.0 - JDBC Driver: mariadb-java-client - ORM: Hibernate (via Spring Data JPA) - Database configuration: application.properties

Security: - JWT (JSON Web Tokens) using jjwt library (version 0.12.3) - Spring Security for authentication and authorization - BCrypt for password hashing

Build Tool: Maven 3.9.x (with Maven Wrapper)

Development Tools: - Lombok: Reduces boilerplate code (getters, setters, constructors) - Spring Boot DevTools: Hot reload during development

2.2 Frontend Technologies

Framework: React 18.2.0 - Component-based UI library - React Router 6.20.1 for client-side routing

Build Tool: Vite 5.0.8 - Fast development server - Optimized production builds - Hot Module Replacement (HMR)

HTTP Client: Axios 1.6.2 - Promise-based HTTP client - Request/response interceptors - Automatic JSON parsing

Styling: Tailwind CSS 3.3.6 - Utility-first CSS framework - Responsive design utilities - PostCSS for processing

Language: JavaScript (ES6+)

2.3 Development Tools

- Version Control: Git
- Package Management: npm (frontend), Maven (backend)
- Code Quality: Lombok annotations (backend)

3. Backend Implementation

3.1 REST API Endpoints

The backend exposes RESTful API endpoints under the /api base path. All endpoints return JSON responses.

3.1.1 Authentication Endpoints (/api/auth)

POST /api/auth/register: Registers a new user - Request Body: RegisterRequest (username, email, password) - Validation: Email format, password strength (minimum 8 characters, uppercase, lowercase, digit) - Response: AuthResponse with success status and message - Implementation: AuthController.register() → AuthService.register() - Error messages in Vietnamese: “Email không hợp lệ”, “Mật khẩu phải có ít nhất 8 ký tự”, etc.

POST /api/auth/login: Authenticates a user - Request Body: LoginRequest (email, password) - Response: AuthResponse with JWT token and user DTO - Implementation: AuthController.login() → AuthService.login() → JwtUtil.generateToken() - Checks if user is banned before allowing login

GET /api/auth/me: Retrieves current authenticated user - Requires: JWT token in Authorization header - Response: UserDTO - Implementation: Extracts user ID from JWT token and retrieves user data via AuthService.getCurrentUser()

POST /api/auth/logout: Logs out user (frontend handles token removal) - Returns success message - Note: With JWT, logout is handled on frontend by removing token

3.1.2 Thread Endpoints (/api/threads)

DELETE /api/threads/{id}: Deletes a thread - Requires: Authentication, must be author, moderator, or admin (with role-based rules) - Response: 204 No Content - Implementation: ThreadController.deleteThread() → ThreadService.deleteThread() (complex authorization logic)

POST /api/threads/{id}/pin: Toggles thread pin status - Requires: ADMIN role (@PreAuthorize("hasRole('ADMIN')")) - Response: 200 OK - Implementation: ThreadController.togglePin() → ThreadService.togglePin()

3.1.3 Reply Endpoints

GET /api/threads/{threadId}/replies: Retrieves all replies for a thread - Response: List<ReplyDTO> - Implementation: ReplyController.getRepliesByThread() → ReplyService.getRepliesByThread()

POST /api/threads/{threadId}/replies: Creates a new reply - Requires: Authentication - Request Body: CreateReplyRequest (content, isAnonymous) - Response: ReplyDTO - Implementation: ReplyController.createReply() → ReplyService.createReply()

POST /api/replies: Alternative endpoint for creating replies - Requires: Authentication - Request Body: CreateReplyRequest (must include threadId) - Response: ReplyDTO - Implementation: ReplyController.createReplySimple() → ReplyService.createReply()

DELETE /api/replies/{id}: Deletes a reply - Requires: Authentication, must be author, moderator, or admin - Response: 204 No Content - Implementation: ReplyController.deleteReply() → ReplyService.deleteReply()

3.1.4 Category Endpoints (/api/categories)

GET /api/categories: Retrieves all categories - Response: List<CategoryDTO> - Public endpoint (no authentication required)
GET /api/categories/{id}: Retrieves category by ID - Response: CategoryDTO - Public endpoint
POST /api/categories: Creates a new category - Requires: ADMIN role (@PreAuthorize("hasRole('ADMIN')")) - Request Body: CategoryDTO - Response: CategoryDTO
PUT /api/categories/{id}: Updates a category - Requires: ADMIN role - Request Body: CategoryDTO - Response: CategoryDTO
DELETE /api/categories/{id}: Deletes a category - Requires: ADMIN role - Response: 204 No Content

3.1.5 User Endpoints (/api/users)

GET /api/users/{id}: Retrieves user by ID - Public endpoint - Response: UserDTO

GET /api/users/all: Retrieves all users - Requires: MODERATOR or ADMIN role (manual role check in controller) - Response: List<UserDTO> - Implementation: Manual role check in UserController.getAllUsers()

GET /api/threads: Retrieves paginated list of threads - Query Parameters: page (default: 0), size (default: 20), sort (default: “createdAt”), categoryId (optional) - Response: Page<ThreadListDTO> - Implementation: ThreadController.getAllThreads() → ThreadService.getAllThreads() or getThreadsByCategoryPaged()

GET /api/threads/category/{categoryId}: Retrieves threads by category (non-paginated) - Response: List<ThreadListDTO> - Implementation: ThreadController.getThreadsByCategory() → ThreadService.getThreadsByCategory()

GET /api/threads/{id}: Retrieves thread by ID with full details - Response: ThreadDTO (includes replies, tags, author, category) - Implementation: ThreadController.getThreadById() → ThreadService.getThreadById()

POST /api/threads: Creates a new thread - Requires: Authentication (@PreAuthorize("isAuthenticated()")) - Request Body: CreateThreadRequest (title, content, categoryId, tags, isAnonymous) - Response: ThreadDTO - Implementation: ThreadController.createThread() → ThreadService.createThread()

PUT /api/threads/{id}: Updates a thread - Requires: Authentication, must be thread author - Request Body: CreateThreadRequest - Response: ThreadDTO - Implementation: ThreadController.updateThread() → ThreadService.updateThread() (authorization check)

PUT /api/users/{id}: Updates user profile - Requires: Must be the user or ADMIN - Request Body: Map<String, String> with fields to update (username, avatar, bio) - Response: Success message

PUT /api/users/{id}/ban: Bans a user - Requires: MODERATOR or ADMIN role (manual check) - Cannot ban ADMIN users - Response: Success message

PUT /api/users/{id}/unban: Unbans a user - Requires: MODERATOR or ADMIN role (manual check) - Response: Success message

PUT /api/users/{id}/role: Changes user role - Requires: ADMIN role only (manual check) - Request Body: {"role": "USER"|"MODERATOR"|"ADMIN"} - Cannot demote ADMIN users - Response: Success message

DELETE /api/users/{id}: Deletes a user - Requires: ADMIN role only (manual check) - Cannot delete ADMIN users - Response: Success message

3.1.6 Admin Endpoints (/api/admin)

GET /api/admin/settings: Retrieves forum settings - Requires: ADMIN role (@PreAuthorize("hasRole('ADMIN')")) - Response: Settings including autoDeleteDays and autoDeleteEnabled

PUT /api/admin/settings/auto-delete-days: Updates auto-delete days setting - Requires: ADMIN role - Request Body: {"days": 90} (0 to disable) - Response: Success message with updated settings

3.2 Controllers Implementation

Controllers are annotated with @RestController and @RequestMapping to define the base path. They use dependency injection (@Autowired) to access service classes.

Example from ThreadController.java:

```
@RestController
 @RequestMapping("/api/threads")
 @CrossOrigin(origins = {"http://localhost:5173", "http://localhost:3000"})
 public class ThreadController {
 @Autowired
 private ThreadService threadService;

 @Autowired
 private JwtUtil jwtUtil;

 @GetMapping
 public ResponseEntity<Page<ThreadListDTO>> getAllThreads(
 @RequestParam(required = false) Long categoryId,
 @RequestParam(defaultValue = "0") int page,
 @RequestParam(defaultValue = "20") int size,
 @RequestParam(defaultValue = "createdAt") String sort) {

// Implementation
 }
 }
```

Controllers extract user information from JWT tokens using helper methods: - extractUserId(HttpServletRequest): Extracts user ID from Authorization header - extractRole(HttpServletRequest): Extracts user role from JWT token

3.3 Service Layer Implementation

Services contain business logic and coordinate between multiple DAOs. They are annotated with @Service and use @Transactional for database operations.
Example from ThreadService.java:

```
@Service
 public class ThreadService {
 @Autowired
 private ThreadDAO threadDAO;

 @Autowired
 private UserDAO userDAO;

 @Autowired
 private CategoryDAO categoryDAO;

 @Autowired
 private TagDAO tagDAO;

 @Transactional
 public ThreadDTO createThread(CreateThreadRequest request, Long userId) {
 User author = userDAO.findById(userId)
 .orElseThrow(() -> new RuntimeException("User not found"));

 Category category = categoryDAO.findById(request.getCategoryId())
 .orElseThrow(() -> new RuntimeException("Category not found"));

 Thread thread = new Thread();
 thread.setTitle(request.getTitle());
 thread.setContent(request.getContent());
 thread.setAuthor(author);
 thread.setCategory(category);
 // ... set other fields

 Thread saved = threadDAO.save(thread);
 return convertToDetailDTO(saved);
 }
 }
```

Key Service Methods:

1. **AuthService.register():**
    - Validates email format using regex: ^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$

◦ Validates password strength (length, uppercase, lowercase, digit)

◦ Checks for duplicate username/email using UserDAO.existsByUsername() and existsByEmail()

◦ Encrypts password using BCrypt

◦ Creates new user with USER role and ACTIVE status

◦ Returns Vietnamese error messages

2. AuthService.login():

◦ Finds user by email using UserDAO.findByEmail()

◦ Checks if user is banned (status == BANNED)

◦ Verifies password using BCrypt PasswordEncoder.matches()

◦ Updates last login timestamp

◦ Generates JWT token via JwtUtil.generateToken()

◦ Returns token and user DTO

3. ThreadService.createThread():

◦ Validates author and category exist

◦ Creates thread entity

◦ Handles tag creation: for each tag name, finds existing tag or creates new one

◦ Saves thread and converts to DTO

4. ThreadService.deleteThread():

◦ Implements complex authorization:

▪ Users can delete their own threads

▪ Moderators can delete USER threads only

▪ Admins can delete USER and MODERATOR threads, but not other ADMIN threads

5. ReplyService.createReply():

◦ Validates thread and author exist

◦ Creates reply entity

◦ Updates thread’s lastReplyAt timestamp

6. ReplyService.deleteReply():

◦ Checks permission: author, moderator, or admin can delete

◦ Throws exception if unauthorized

3.4 DTOs (Data Transfer Objects)

DTOs are used to transfer data between layers and to define API contracts. They separate the internal entity structure from the API interface.

Key DTOs:

• RegisterRequest: Contains username, email, password with validation annotations
• LoginRequest: Contains email and password
• AuthResponse: Contains success flag, message, token (optional), and user DTO (optional)
• CreateThreadRequest: Contains title, content, categoryId, tags (list of strings), isAnonymous with @NotBlank and @Size validations
• ThreadDTO: Complete thread representation with author, category, tags, replies
• ThreadListDTO: Lightweight thread representation for list views (includes reply count)
• ReplyDTO: Reply representation with author information
• CreateReplyRequest: Contains content, isAnonymous, and optional threadId
• UserDTO: User representation without password
• AuthorDTO: Simplified author information (id, username, avatar, role)
• CategoryDTO: Category representation (id, name, description)
• TagDTO: Tag representation (id, name)

DTOs use Lombok @Data annotation to generate getters, setters, and constructors automatically.

3.5 Database Models (Entities)

Entities are JPA-annotated classes that map to database tables. They use Hibernate annotations for relationships and constraints.

User Entity (User.java): - Fields: id, username (unique), email (unique), password, role (enum: USER, MODERATOR, ADMIN), status (enum: ACTIVE, BANNED, SUSPENDED), bio, avatar, timestamps - Relationships: @OneToMany with Thread and Reply (lazy fetch) - Uses @CreationTimestamp and @UpdateTimestamp for automatic timestamp management - Default role: USER, default status: ACTIVE

Thread Entity (Thread.java): - Fields: id, title (max 200), content (TEXT), views (default 0), isAnonymous (default false), isPinned (default false), timestamps, lastReplyAt - Relationships: - @ManyToOne with User (author) and Category (lazy fetch) - @OneToMany with Reply (lazy fetch, cascade ALL) - @ManyToMany with Tag (via thread_tags join table, lazy fetch)

Reply Entity (Reply.java): - Fields: id, content (TEXT), isAnonymous (default false), timestamps - Relationships: @ManyToOne with Thread and User (author) (lazy fetch)

Category Entity (Category.java): - Fields: id, name (unique), slug (unique), description, icon, displayOrder - Relationships: @OneToMany with Thread (lazy fetch)

Tag Entity (Tag.java): - Fields: id, name (unique), slug (unique) - Relationships: @ManyToMany with Thread (inverse side, lazy fetch)

3.6 Key Logic Flows

3.6.1 User Registration Flow

1. Client sends POST request to /api/auth/register with username, email, password
2. AuthController.register() receives request and validates DTO using @Valid
3. AuthService.register() executes:
   - Validates email format (regex)
   - Validates password strength (length >= 8, uppercase, lowercase, digit)
   - Checks username/email uniqueness via UserDAO.existsByUsername() and existsByEmail()
   - Creates new User entity
   - Encrypts password using PasswordEncoder.encode() (BCrypt)
   - Sets default role (USER) and status (ACTIVE)
   - Saves user via UserDAO.save()
4. Returns AuthResponse with success message (Vietnamese: "Đăng ký thành công")

3.6.2 User Login Flow

1. Client sends POST request to /api/auth/login with email and password
2. AuthController.login() receives request
3. AuthService.login() executes:
   - Finds user by email via UserDAO.findByEmail()
   - Checks if user exists and is not banned (status != BANNED)
   - Verifies password using PasswordEncoder.matches()
   - Updates lastLoginAt timestamp
   - Generates JWT token via JwtUtil.generateToken() (includes username, userId, role)
   - Creates UserDTO (without password)
   - Returns AuthResponse with token and user DTO
4. Frontend stores token in localStorage

3.6.3 Thread Creation Flow

1. Authenticated client sends POST request to /api/threads with thread data
2. JwtAuthenticationFilter validates JWT token and sets authentication in SecurityContext
3. ThreadController.createThread() extracts user ID from JWT token
4. ThreadService.createThread() executes:
   - Validates author exists via UserDAO.findById()
   - Validates category exists via CategoryDAO.findById()
   - Creates new Thread entity
   - Processes tags: for each tag name in request, finds existing tag via TagDAO.findByName() or creates new one
   - Saves thread via ThreadDAO.save() (persists thread and tag relationships)
   - Converts entity to ThreadDTO (includes author, category, tags)
5. Returns ThreadDTO with 201 Created status

3.6.4 Thread Deletion with Authorization Flow

1. Authenticated client sends DELETE request to /api/threads/{id}
2. ThreadController.deleteThread() extracts user ID and role from JWT

3. ThreadService.deleteThread() executes authorization logic:
   ◦ Retrieves thread and author
   ◦ If user is thread author: allows deletion
   ◦ If user is MODERATOR: allows deletion only if author is USER
   ◦ If user is ADMIN: allows deletion if author is USER or MODERATOR (not ADMIN)
   ◦ Throws exception if unauthorized
4. Deletes thread via ThreadDAO.delete() (cascades to replies via JPA)
5. Returns 204 No Content

3.6.5 Reply Creation Flow

1. Authenticated client sends POST request to /api/threads/{threadId}/replies with content
2. ReplyController.createReply() extracts user ID from JWT token
3. ReplyService.createReply() executes:
   ◦ Validates thread exists via ThreadDAO.findById()
   ◦ Validates author exists via UserDAO.findById()
   ◦ Creates Reply entity
   ◦ Sets timestamps
   ◦ Saves reply via ReplyDAO.save()
   ◦ Updates thread's lastReplyAt timestamp
   ◦ Converts to ReplyDTO
4. Returns ReplyDTO with 201 Created status

4. Key Logic Flow

4.1 User Registration Flow

1. Client sends POST request to /api/auth/register with username, email, password
2. AuthController.register() receives request and validates DTO using @Valid
3. AuthService.register() executes:
   ◦ Validates email format using regex: ^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$
   ◦ Validates password strength:
      ▪ Minimum 8 characters
      ▪ At least one uppercase letter (.*[A-Z].*)

▪ At least one lowercase letter (.*[a-z].*)

▪ At least one digit (.*[0-9].*)

◦ Checks username uniqueness via UserDAO.existsByUsername()

◦ Checks email uniqueness via UserDAO.existsByEmail()

◦ Creates new User entity

◦ Encrypts password using PasswordEncoder.encode() (BCrypt)

◦ Sets default role (User.Role.USER) and status (User.UserStatus.ACTIVE)

◦ Saves user via UserDAO.save()

4. Returns AuthResponse with success message (in Vietnamese: “Đăng ký thành công”)

5. If validation fails, returns AuthResponse with success=false and error message

4.2 User Login Flow

1. Client sends POST request to /api/auth/login with email and password

2. AuthController.login() receives request

3. AuthService.login() executes:

◦ Finds user by email via UserDAO.findByEmail()

◦ Checks if user exists (returns error if not found)

◦ Checks if user is banned (status == BANNED) - rejects if banned

◦ Verifies password using PasswordEncoder.matches(plainPassword, hashedPassword)

◦ Updates lastLoginAt timestamp to current time

◦ Generates JWT token via JwtUtil.generateToken(username, userId, role):

▪ Includes username as subject

▪ Includes userId and role in claims

▪ Sets expiration time (604800000 ms = 7 days from application.properties)

▪ Signs with HMAC-SHA256 using secret key

◦ Creates UserDTO (without password)

◦ Returns AuthResponse with success=true, token, and user DTO

4. Frontend stores token in localStorage via AuthContext.login()

JWT Token Structure: - Header: Algorithm (HS256), Type (JWT) - Payload: Subject (username), Claims (userId, role), Issued At, Expiration - Signature: HMAC-SHA256(header + payload, secret)

4.3 Thread Creation Flow

1. Authenticated client sends POST request to /api/threads with thread data
2. JwtAuthenticationFilter validates JWT token and sets authentication in SecurityContext
3. ThreadController.createThread() extracts user ID from JWT token using extractUserId()
4. ThreadService.createThread() executes:
   - Validates author exists via UserDAO.findById(userId)
   - Validates category exists via CategoryDAO.findById(request.getCategoryId())
   - Creates new Thread entity
   - Sets title, content, author, category
   - Initializes views to 0, isPinned to false
   - Sets createdAt and updatedAt to current time
   - Processes tags: for each tag name in request.getTags():
     - Queries TagDAO.findByName(tagName) to check if tag exists
     - If exists: uses existing tag
     - If not exists: creates new Tag entity and saves it
   - Associates tags with thread via @ManyToMany relationship
   - Saves thread via ThreadDAO.save() (persists thread and tag relationships)
   - Converts entity to ThreadDTO (includes author, category, tags)
5. Returns ThreadDTO with 201 Created status

Transaction Management: Method is annotated with @Transactional to ensure atomicity of thread and tag creation.

4.4 Thread Deletion with Authorization Flow

1. Authenticated client sends DELETE request to /api/threads/{id}
2. ThreadController.deleteThread() extracts user ID and role from JWT

3. ThreadService.deleteThread() executes authorization logic:
    - Retrieves thread via ThreadDAO.findById(threadId)
    - Gets thread author and author's role
    - If user is thread author: allows deletion (any role can delete own threads)
    - If user is MODERATOR: allows deletion only if author is USER (not ADMIN/MODERATOR)
    - If user is ADMIN: allows deletion if author is USER or MODERATOR (not other ADMIN)
    - Throws RuntimeException if unauthorized
4. Deletes thread via ThreadDAO.delete() (cascades to replies due to CascadeType.ALL)
5. Returns 204 No Content

4.5 Reply Creation Flow

1. Authenticated client sends POST request to /api/threads/{threadId}/replies with content
2. ReplyController.createReply() extracts user ID from JWT
3. ReplyService.createReply() executes:
    - Validates thread exists via ThreadDAO.findById(threadId)
    - Validates author exists via UserDAO.findById(userId)
    - Creates Reply entity:
        - Sets content, thread, author
        - Sets createdAt and updatedAt to current time
    - Saves reply via ReplyDAO.save()
    - Updates thread's lastReplyAt timestamp to current time
    - Saves thread via ThreadDAO.save()
    - Converts to ReplyDTO
4. Returns ReplyDTO with 201 Created status

4.6 Tag System Flow

Implementation Location: ThreadService.createThread() (tag processing)

Process: 1. Receives list of tag names in CreateThreadRequest.getTags() (List) 2. For each tag name: - Queries TagDAO.findByName(tagName) to check if tag exists - If exists: uses existing tag entity - If not exists: creates new Tag entity with name (slug is auto-generated or set separately) 3. Associates tags with thread via @ManyToMany relationship 4. Hibernate automatically manages thread_tags join table on save

Tag Uniqueness: Tag names are unique (enforced by database constraint @Column(unique = true)).

4.7 Thread Pinning Flow

Implementation Location: ThreadService.togglePin()

Process: 1. Endpoint requires ADMIN role (@PreAuthorize("hasRole('ADMIN')")) 2. ThreadController.togglePin() receives thread ID 3. ThreadService.togglePin() executes: - Retrieves thread via ThreadDAO.findById(threadId) - Toggles isPinned boolean field: thread.setPinned(!thread.isPinned()) - Saves thread 4. Returns 200 OK

Usage: Pinned threads should appear at the top of thread lists (frontend sorting logic).

4.8 Moderator Permissions Flow

Implementation: Role-based authorization is implemented in service methods and controller annotations.

Moderator Capabilities:

1. Delete Threads: ThreadService.deleteThread() allows moderators to delete USER threads only

• if (userRole.equals("MODERATOR")) {

if (authorRole.equals("USER")) {

threadDAO.delete(thread);

return;

}

throw new RuntimeException("Moderators cannot delete Admin/Moderator threads");

}

2. Delete Replies: ReplyService.deleteReply() allows moderators to delete any reply

• if (!reply.getAuthor().getId().equals(userId) &&

!userRole.equals("MODERATOR") &&

!userRole.equals("ADMIN")) {

throw new RuntimeException("Unauthorized to delete this reply");

}

3. Ban/Unban Users: UserController.banUser() and unbanUser() allow moderators to ban/unban users (except admins)

◦ Manual role check: currentUser.getRole() != User.Role.ADMIN && currentUser.getRole() != User.Role.MODERATOR

◦ Cannot ban ADMIN users

4. View All Users: UserController.getAllUsers() allows moderators to view user list

◦ Manual role check in controller

Role Hierarchy: ADMIN > MODERATOR > USER

5. Frontend Implementation

5.1 Page Components

Home Page (Home.jsx): - Displays all categories using CategoryCard components - Fetches categories via categoriesAPI.getAll() - Provides navigation to category pages and thread creation - Shows recent/popular threads

Login Page (Login.jsx): - Form with email and password fields - Calls authAPI.login(email, password) on submit - On success: stores token and user in localStorage via AuthContext.login() - Redirects to home page - Displays error messages on failure

Register Page (Register.jsx): - Form with username, email, and password fields - Client-side validation for password strength - Calls authAPI.register(username, email, password) on submit - Displays success/error messages - Redirects to login or home on success

Thread Page (ThreadPage.jsx): - Fetches thread details via threadsAPI.getById(id) - Displays thread content, author, category, tags - Fetches replies via repliesAPI.getByThread(threadId) - Renders ReplyBox component for posting replies - Shows edit/delete buttons for thread author - Displays all replies in chronological order

Create Thread Page (CreateThread.jsx): - Form with title, content, category selection, tag input - Fetches categories for dropdown via categoriesAPI.getAll() - Calls threadsAPI.create(data) on submit - Redirects to created thread page on success - Handles validation errors

Edit Thread Page (EditThread.jsx): - Pre-fills form with current thread data - Fetches thread via threadsAPI.getById(id) - Calls threadsAPI.update(id, data) on submit - Only accessible to thread author

Category Page (Category.jsx): - Fetches threads by category via threadsAPI.getAll({ categoryId }) - Implements pagination - Displays threads using ThreadRow components - Shows thread count and sorting options

User Management Page (UserManagement.jsx): - Admin/Moderator only page - Fetches all users via usersAPI.getAll() - Displays user list with roles and status - Provides ban/unban functionality via usersAPI.ban(id) and usersAPI.unban(id) - Provides role change functionality (admin only) - Shows user details and activity

Settings Page (Settings.jsx): - User profile settings - Allows updating username, avatar, bio - Calls usersAPI.update(id, data) on submit

5.2 API Integration

All API calls are centralized in src/api/index.js:

```
const api = axios.create({
  baseURL: 'http://localhost:8080/api',
  headers: { 'Content-Type': 'application/json' }
});

// Request interceptor: Attach JWT token
api.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Response interceptor: Handle 401 errors
api.interceptors.response.use(
  response => response,
  error => {
```

```
    if (error.response?.status === 401) {
      localStorage.removeItem('token');
      localStorage.removeItem('user');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);
```

API Methods: - authAPI.login(email, password): POST /api/auth/login - authAPI.register(username, email, password): POST /api/auth/register - threadsAPI.getAll(params): GET /api/threads with query parameters - threadsAPI.getById(id): GET /api/threads/{id} - threadsAPI.create(data): POST /api/threads - threadsAPI.update(id, data): PUT /api/threads/{id} - threadsAPI.delete(id): DELETE /api/threads/{id} - repliesAPI.getByThread(threadId): GET /api/threads/{threadId}/replies - repliesAPI.create(threadId, content): POST /api/threads/{threadId}/replies - categoriesAPI.getAll(): GET /api/categories - usersAPI.getAll(): GET /api/users/all - usersAPI.ban(id): PUT /api/users/{id}/ban - usersAPI.unban(id): PUT /api/users/{id}/unban

5.3 State Management

AuthContext (src/contexts/AuthContext.jsx): - Manages global authentication state - Provides user, login(), logout(), and loading state - Persists user data in localStorage - Loads user from localStorage on application mount

```
export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const token = localStorage.getItem('token');
    const storedUser = localStorage.getItem('user');
    if (token && storedUser) {
      setUser(JSON.parse(storedUser));
    }
```

```
    setLoading(false);
  }, []);

  const login = (userData, token) => {
    localStorage.setItem('token', token);
    localStorage.setItem('user', JSON.stringify(userData));
    setUser(userData);
  };

  const logout = () => {
    setUser(null);
    localStorage.removeItem('user');
    localStorage.removeItem('token');
  };

  return (
    <AuthContext.Provider value={{ user, login, logout, loading }}>
      {children}
    </AuthContext.Provider>
  );
}
```

Components access authentication state using the useAuth() hook:

```
const { user, login, logout } = useAuth();
```

5.4 Routing

React Router is configured in App.jsx:

```
<BrowserRouter>
```

```
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/login" element={<Login />} />
    <Route path="/register" element={<Register />} />
    <Route path="/thread/:id" element={<ThreadPage />} />
    <Route path="/create-thread" element={<CreateThread />} />
    <Route path="/edit-thread/:id" element={<EditThread />} />
    <Route path="/category/:id" element={<Category />} />
    <Route path="/settings" element={<Settings />} />
    <Route path="/users" element={<UserManagement />} />
  </Routes>
</BrowserRouter>
```

Note: Route protection is handled at the component level by checking user from AuthContext, not via route guards.

5.5 UI Interactions

Navigation: The Header component displays navigation links and user menu. It uses useAuth() to conditionally show login/register or user profile links.

Form Handling: Forms use controlled components with React state. On submit, they call API methods and handle responses with success/error messages.

Loading States: Components display loading indicators while fetching data. The Loading component provides a consistent spinner UI.

Error Handling: API errors are caught and displayed to users. 401 errors automatically redirect to login page via Axios interceptor.

6. Database Implementation

6.1 Database Schema Overview

The database schema consists of five main tables with relationships defined through foreign keys and join tables.

6.1.1 Users Table

```
CREATE TABLE users (
```

```
    id BIGINT PRIMARY KEY AUTO_INCREMENT,

    username VARCHAR(50) UNIQUE NOT NULL,

    email VARCHAR(100) UNIQUE NOT NULL,

    password VARCHAR(255) NOT NULL,

    role ENUM('USER', 'MODERATOR', 'ADMIN') NOT NULL DEFAULT 'USER',

    status ENUM('ACTIVE', 'BANNED', 'SUSPENDED') NOT NULL DEFAULT 'ACTIVE',

    bio VARCHAR(500),

    avatar VARCHAR(255),

    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,

    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

    last_login_at TIMESTAMP

);
```

Constraints: - username and email are unique - role defaults to ‘USER’ - status defaults to ‘ACTIVE’ - Timestamps are automatically managed by Hibernate (@CreationTimestamp, @UpdateTimestamp)

6.1.2 Categories Table

```
CREATE TABLE categories (

    id BIGINT PRIMARY KEY AUTO_INCREMENT,

    name VARCHAR(100) UNIQUE NOT NULL,

    slug VARCHAR(100) UNIQUE NOT NULL,

    description VARCHAR(500),

    icon VARCHAR(50),

    display_order INT NOT NULL DEFAULT 0

);
```

Constraints: - name and slug are unique - display_order determines category ordering

6.1.3 Threads Table

```
CREATE TABLE threads (
```

```
    id BIGINT PRIMARY KEY AUTO_INCREMENT,

    title VARCHAR(200) NOT NULL,

    content TEXT NOT NULL,

    views INT NOT NULL DEFAULT 0,

    is_anonymous BOOLEAN NOT NULL DEFAULT FALSE,

    is_pinned BOOLEAN NOT NULL DEFAULT FALSE,

    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,

    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

    last_reply_at TIMESTAMP,

    author_id BIGINT NOT NULL,

    category_id BIGINT NOT NULL,

    FOREIGN KEY (author_id) REFERENCES users(id),

    FOREIGN KEY (category_id) REFERENCES categories(id)

);
```

Constraints: - author_id references users(id) - category_id references categories(id) - Boolean fields default to FALSE

6.1.4 Replies Table

```
CREATE TABLE replies (

    id BIGINT PRIMARY KEY AUTO_INCREMENT,

    content TEXT NOT NULL,

    is_anonymous BOOLEAN NOT NULL DEFAULT FALSE,

    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,

    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,

    thread_id BIGINT NOT NULL,

    author_id BIGINT NOT NULL,

    FOREIGN KEY (thread_id) REFERENCES threads(id) ON DELETE CASCADE,
```

```
    FOREIGN KEY (author_id) REFERENCES users(id)
);
```

Constraints: - thread_id references threads(id) with CASCADE delete - author_id references users(id)

6.1.5 Tags Table

```
CREATE TABLE tags (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) UNIQUE NOT NULL,
    slug VARCHAR(50) UNIQUE NOT NULL
);
```

6.1.6 Thread-Tags Join Table

```
CREATE TABLE thread_tags (
    thread_id BIGINT NOT NULL,
    tag_id BIGINT NOT NULL,
    PRIMARY KEY (thread_id, tag_id),
    FOREIGN KEY (thread_id) REFERENCES threads(id) ON DELETE CASCADE,
    FOREIGN KEY (tag_id) REFERENCES tags(id) ON DELETE CASCADE
);
```

Constraints: - Composite primary key on (thread_id, tag_id) - CASCADE delete on both foreign keys

6.2 Entity Relationships

The database implements the following relationships:

1. User → Threads: One-to-Many (one user can create many threads)
    - Foreign key: threads.author_id → users.id
    - JPA: @OneToMany(mappedBy = "author") in User, @ManyToOne in Thread
2. User → Replies: One-to-Many (one user can post many replies)
    - Foreign key: replies.author_id → users.id

◦ JPA: @OneToMany(mappedBy = "author") in User, @ManyToOne in Reply

3. Category → Threads: One-to-Many (one category contains many threads)

◦ Foreign key: threads.category_id → categories.id

◦ JPA: @OneToMany(mappedBy = "category") in Category, @ManyToOne in Thread

4. Thread → Replies: One-to-Many (one thread has many replies)

◦ Foreign key: replies.thread_id → threads.id

◦ CASCADE delete: deleting a thread deletes all its replies

◦ JPA: @OneToMany(mappedBy = "thread", cascade = CascadeType.ALL) in Thread

5. Thread ↔ Tags: Many-to-Many (threads can have multiple tags, tags can be on multiple threads)

◦ Join table: thread_tags

◦ CASCADE delete on both sides

◦ JPA: @ManyToMany with @JoinTable in Thread, @ManyToMany(mappedBy = "tags") in Tag

6.3 Database Configuration

Database configuration is specified in application.properties:

spring.datasource.url=jdbc:mariadb://localhost:3306/forum_db

spring.datasource.username=root

spring.datasource.password=123123

spring.datasource.driver-class-name=org.mariadb.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format_sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDBDialect

Key Settings: - ddl-auto=update: Automatically creates/updates tables based on entity definitions - show-sql=true: Logs all SQL queries for debugging - format_sql=true: Formats SQL queries in logs for readability

7. Security Implementation

7.1 Authentication

JWT-Based Authentication: The system uses JSON Web Tokens for stateless authentication.

JWT Filter (JwtAuthenticationFilter.java): - Extends OncePerRequestFilter to intercept all HTTP requests - Extracts token from Authorization: Bearer <token> header - Validates token using JwtUtil.validateToken() - Extracts user information (username, userId, role) from token - Creates UsernamePasswordAuthenticationToken with role authority (ROLE_USER, ROLE_MODERATOR, ROLE_ADMIN) - Sets authentication in SecurityContextHolder

Token Generation (JwtUtil.java): - Uses io.jsonwebtoken library (version 0.12.3) - Signs tokens with HMAC-SHA256 algorithm - Secret key stored in application.properties (jwt.secret) - Token expiration: 7 days (604800000 ms) - Token payload includes: username (subject), userId, role

Token Validation: - Checks token signature using Jwts.parser().verifyWith(getSigningKey()) - Verifies expiration date - Returns false on any JWT exception (JwtException, IllegalArgumentException)

7.2 Authorization

Spring Security Configuration (SecurityConfig.java): - Disables CSRF (not needed for stateless JWT) - Configures CORS for frontend origins (http://localhost:3000, http://localhost:5173) - Sets session creation policy to STATELESS - Configures public endpoints: - /api/auth/** - Authentication endpoints - /api/test/** - Test endpoints - /api/categories/** - Category endpoints (public read access) - /api/threads/** - Thread endpoints (public read access) - /api/users/** - User endpoints (public read access for some) - Requires authentication for all other endpoints - Adds JwtAuthenticationFilter before UsernamePasswordAuthenticationFilter

Method-Level Security: - Uses @PreAuthorize annotations for role-based access control - Examples: - @PreAuthorize("isAuthenticated()"): Requires any authenticated user - @PreAuthorize("hasRole('ADMIN')"): Requires ADMIN role - @PreAuthorize("hasRole('MODERATOR')"): Requires MODERATOR role

Programmatic Authorization: - Service methods check user roles and ownership programmatically - Example from ThreadService.deleteThread(): checks if user is author, moderator, or admin with specific rules - UserController uses manual role checks instead of @PreAuthorize for some endpoints

7.3 Password Security

BCrypt Hashing: - Passwords are hashed using BCrypt algorithm - Configured via PasswordEncoder bean in SecurityConfig (BCryptPasswordEncoder) - BCrypt automatically includes salt in hash - Password verification uses PasswordEncoder.matches(plainPassword, hashedPassword)

Password Validation: - Minimum 8 characters - Requires uppercase, lowercase, and digit - Validation occurs in AuthService.register() (service layer, not DTO validation)

7.4 CORS Configuration

CORS Setup (SecurityConfig.java): - Allows origins: http://localhost:3000 and http://localhost:5173 - Allows methods: GET, POST, PUT, DELETE, OPTIONS - Allows all headers (*) - Allows credentials (cookies, authorization headers)

7.5 Input Validation

Jakarta Validation: - DTOs use validation annotations: - @NotBlank: Field cannot be null or empty - @NotNull: Field cannot be null - @Size(max = 200): String length constraint - Controllers use @Valid annotation to trigger validation - Invalid requests return 400 Bad Request with validation errors

Service Layer Validation: - AuthService.register(): Validates email format, password strength, uniqueness - ThreadService.createThread(): Validates author and category exist - ReplyService.createReply(): Validates thread and author exist

Database Constraints: - Unique constraints on username, email, category name/slug, tag name/slug - Foreign key constraints ensure referential integrity - NOT NULL constraints on required fields

8. Error Handling & Validation

8.1 Global Exception Handler

Implementation (GlobalExceptionHandler.java): - Annotated with @RestControllerAdvice to catch exceptions from all controllers - Provides centralized exception handling

Exception Handlers:

1. ResourceNotFoundException: Returns 404 Not Found

• @ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Map<String, Object>> handleResourceNotFoundException(...) {

// Returns 404 with error message

}

2. RuntimeException: Returns 400 Bad Request (for business logic errors)

• @ExceptionHandler(RuntimeException.class)

public ResponseEntity<Map<String, Object>> handleRuntimeException(...) {

// Returns 400 with error message

}

3. AccessDeniedException: Returns 403 Forbidden

• @ExceptionHandler(AccessDeniedException.class)

public ResponseEntity<Map<String, Object>> handleAccessDeniedException(...) {

// Returns 403 with "Access denied" message

}

4. BadCredentialsException: Returns 401 Unauthorized

• @ExceptionHandler(BadCredentialsException.class)

public ResponseEntity<Map<String, Object>> handleBadCredentialsException(...) {

// Returns 401 with "Invalid email or password" message

}

5. IllegalArgumentException: Returns 400 Bad Request (validation errors)

6. NullPointerException: Returns 500 Internal Server Error (with logging)

7. Generic Exception: Returns 500 Internal Server Error (catch-all)

Error Response Format:

{

"success": false,

"message": "Error message",

"status": 404,

"timestamp": "2025-12-22T10:00:00"

}

8.2 Input Validation

DTO Validation: - CreateThreadRequest: @NotBlank on title and content, @Size(max=200) on title, @NotNull on categoryId - RegisterRequest: Basic presence validation (detailed validation in service layer) - LoginRequest: Basic presence validation

Service Layer Validation: - AuthService.register(): Validates email format (regex), password strength, uniqueness - ThreadService.createThread(): Validates author and category exist - ReplyService.createReply(): Validates thread and author exist

Database Constraints: - Unique constraints on username, email, category name/slug, tag name/slug - Foreign key constraints ensure referential integrity - NOT NULL constraints on required fields

8.3 Frontend Error Handling

Axios Interceptor (src/api/index.js): - Response interceptor catches HTTP errors - 401 errors: Clears localStorage and redirects to login - Other errors: Logs error and rejects promise

Component Error Handling: - Components use try-catch blocks for API calls - Error messages displayed to users via UI components - Loading states prevent duplicate requests

9. Implementation Challenges

9.1 Entity Relationships

Challenge: Managing complex relationships between User, Thread, Reply, Category, and Tag entities, especially the many-to-many relationship between Thread and Tag.

Solution: - Used JPA annotations (@ManyToOne, @OneToMany, @ManyToMany) to define relationships - Created join table thread_tags for many-to-many relationship - Used CascadeType.ALL for automatic deletion of related entities - Implemented lazy loading (FetchType.LAZY) to avoid N+1 query problems

9.2 JWT Authentication Integration

Challenge: Integrating JWT authentication with Spring Security in a stateless manner, extracting user information from tokens in controllers.

Solution: - Created JwtAuthenticationFilter to intercept requests and validate tokens - Extracted user information (userId, role) from JWT claims - Stored authentication in SecurityContextHolder for Spring Security integration - Created helper methods in controllers to extract userId and role from JWT tokens - Used JwtUtil utility class for token generation and validation

9.3 Role-Based Authorization

Challenge: Implementing complex authorization rules (e.g., moderators can delete USER threads but not ADMIN threads, users can only edit their own threads).

Solution: - Combined @PreAuthorize annotations for method-level security - Implemented programmatic authorization checks in service methods - Created role hierarchy: ADMIN > MODERATOR > USER - Added ownership checks (users can only modify their own content) - Used manual role checks in UserController for some endpoints

9.4 Pagination Implementation

Challenge: Implementing efficient pagination for thread lists with filtering by category.

Solution: - Used Spring Data JPA Pageable and Page interfaces - Created custom query methods in ThreadDAO.findByCategoryId() with pagination - Implemented DTO conversion for paginated results - Frontend handles pagination state and requests

9.5 DTO Conversion

Challenge: Converting between entity models and DTOs while maintaining clean separation of concerns.

Solution: - Created separate DTO classes for API requests and responses - Implemented conversion methods in service classes (convertToDTO(), convertToDetailDTO(), convertToListDTO()) - Used Lombok @Data to reduce boilerplate in DTOs - Maintained separate DTOs for list views (ThreadListDTO) and detail views (ThreadDTO)

9.6 Frontend State Management

Challenge: Managing authentication state across the application without prop drilling.

Solution: - Implemented React Context API (AuthContext) - Stored user data and token in localStorage for persistence - Created useAuth() hook for easy access to authentication state - Implemented loading state to prevent race conditions

9.7 CORS Configuration

Challenge: Configuring CORS to allow frontend (running on different port) to access backend API.

Solution: - Configured CORS in SecurityConfig.java using CorsConfigurationSource - Allowed specific origins (http://localhost:3000, http://localhost:5173) - Enabled credentials for JWT token transmission - Added CORS configuration to Axios requests

9.8 Transaction Management

Challenge: Ensuring data consistency when creating threads with tags (creating new tags if they don’t exist).

Solution: - Used @Transactional annotation on service methods - Ensured atomicity of thread creation and tag creation - Handled tag lookup and creation within the same transaction

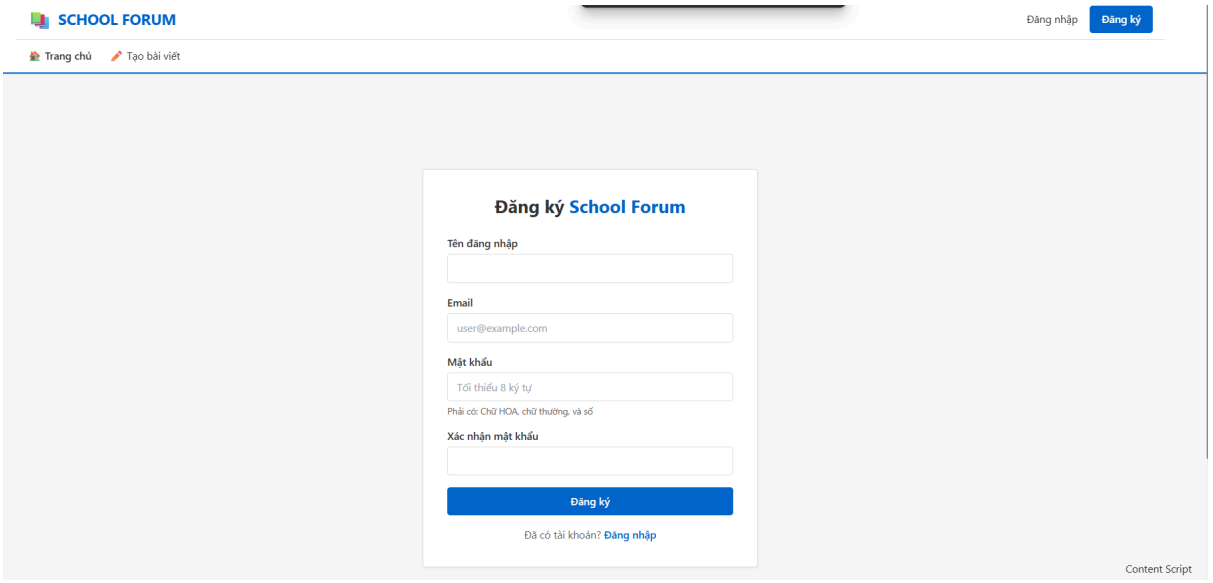
9.9 Error Response Consistency

Challenge: Providing consistent error responses across all endpoints.

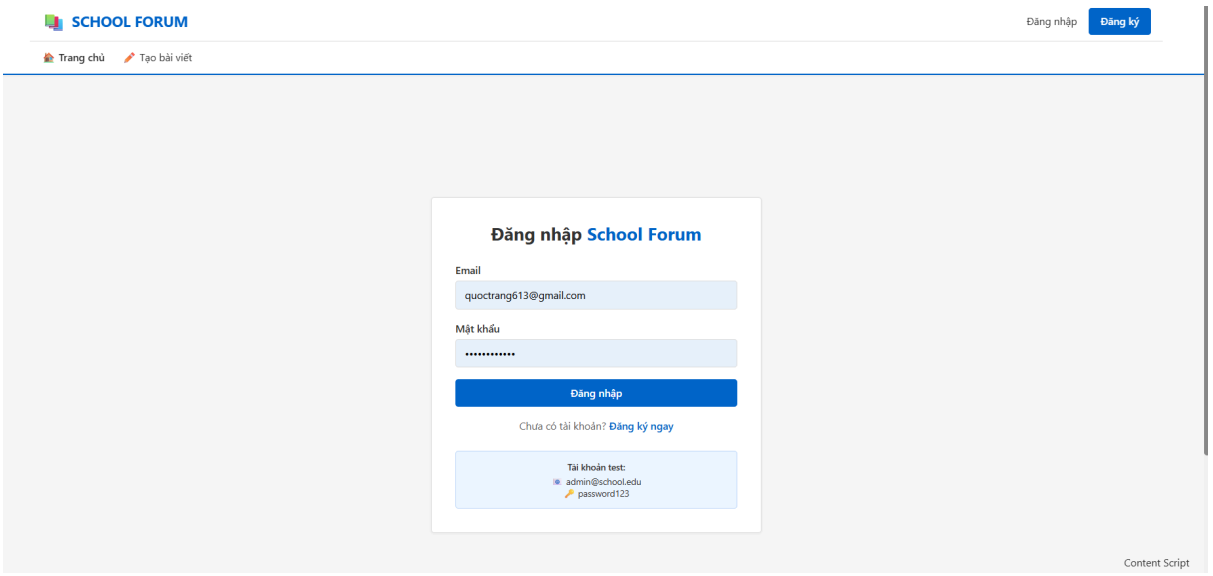
Solution: - Implemented GlobalExceptionHandler with @RestControllerAdvice - Standardized error response format (success, message, status, timestamp) - Handled different exception types with appropriate HTTP status codes - Logged errors for debugging while returning user-friendly messages

10. Screenshot (Demo for each function)

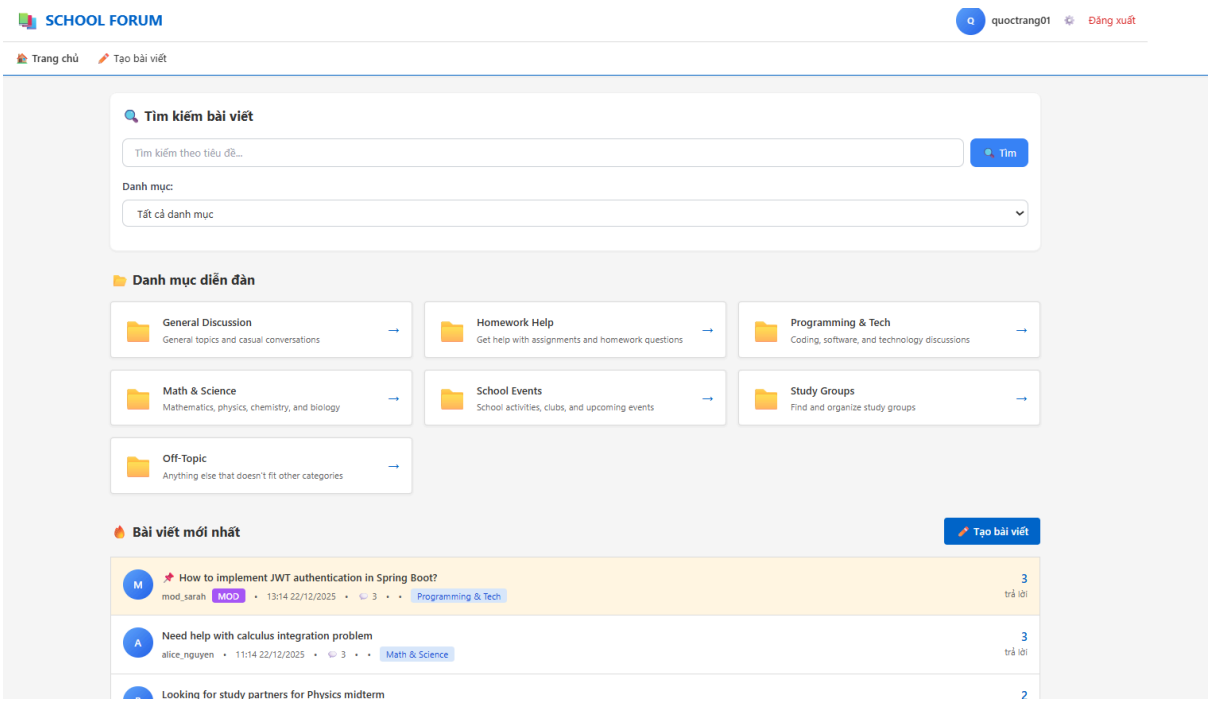
Register Page



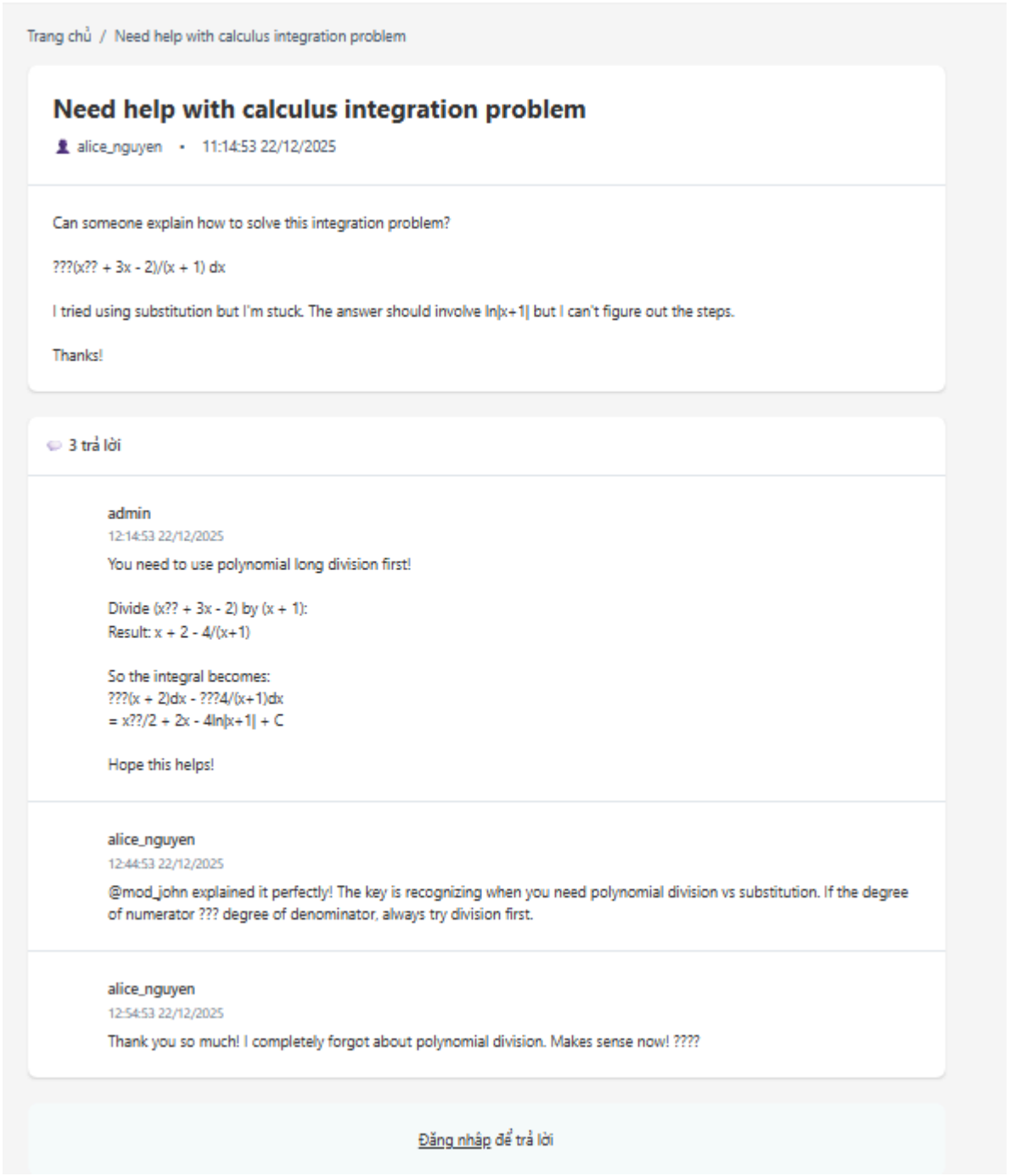
Login Page



HomePage



View Thread and Replies (Create Reply only available if you already login)



Edit your own thread

Sheek

✏️ Sửa 🗑️ Xóa 📌 Ghim

👤 admin • 11:41:21 20/12/2025

HW HELPaaaaa

✏️ Chỉnh sửa bài viết

Chủ đề

Homework Help

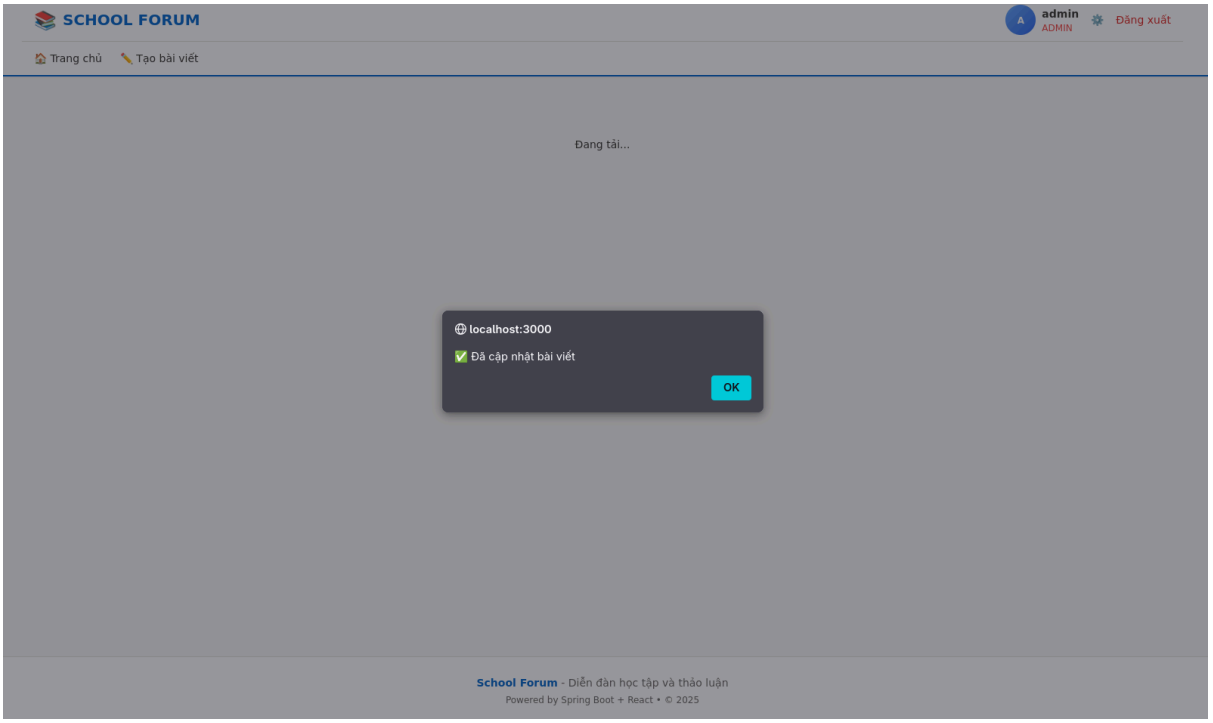
Tiêu đề

Sheek

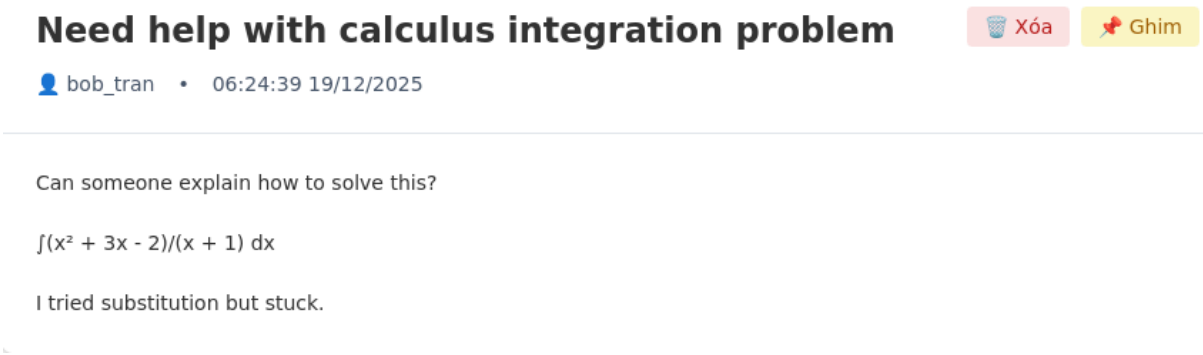
Nội dung

HW HELPaaaaa

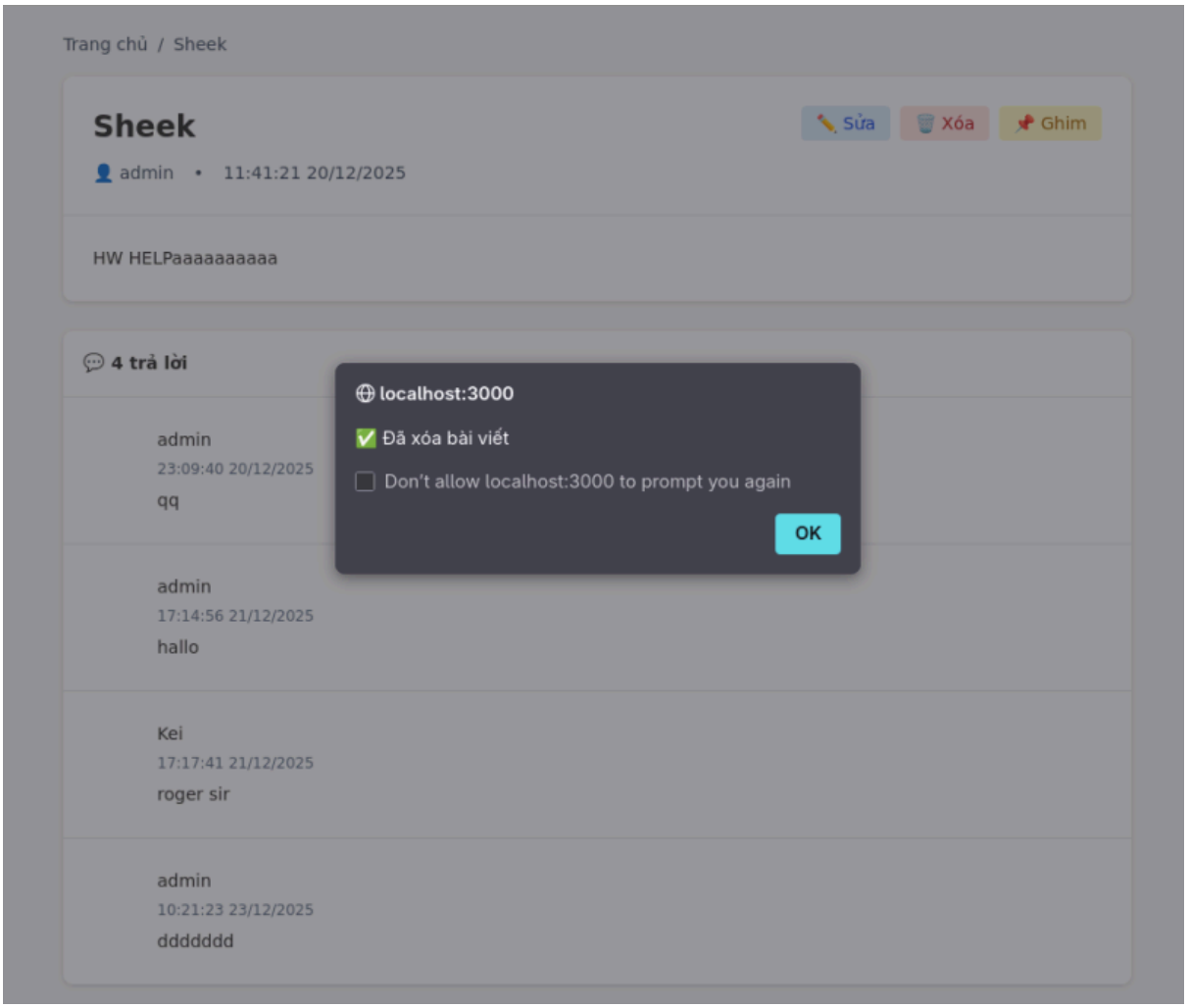
💾 Lưu thay đổi ❌ Hủy

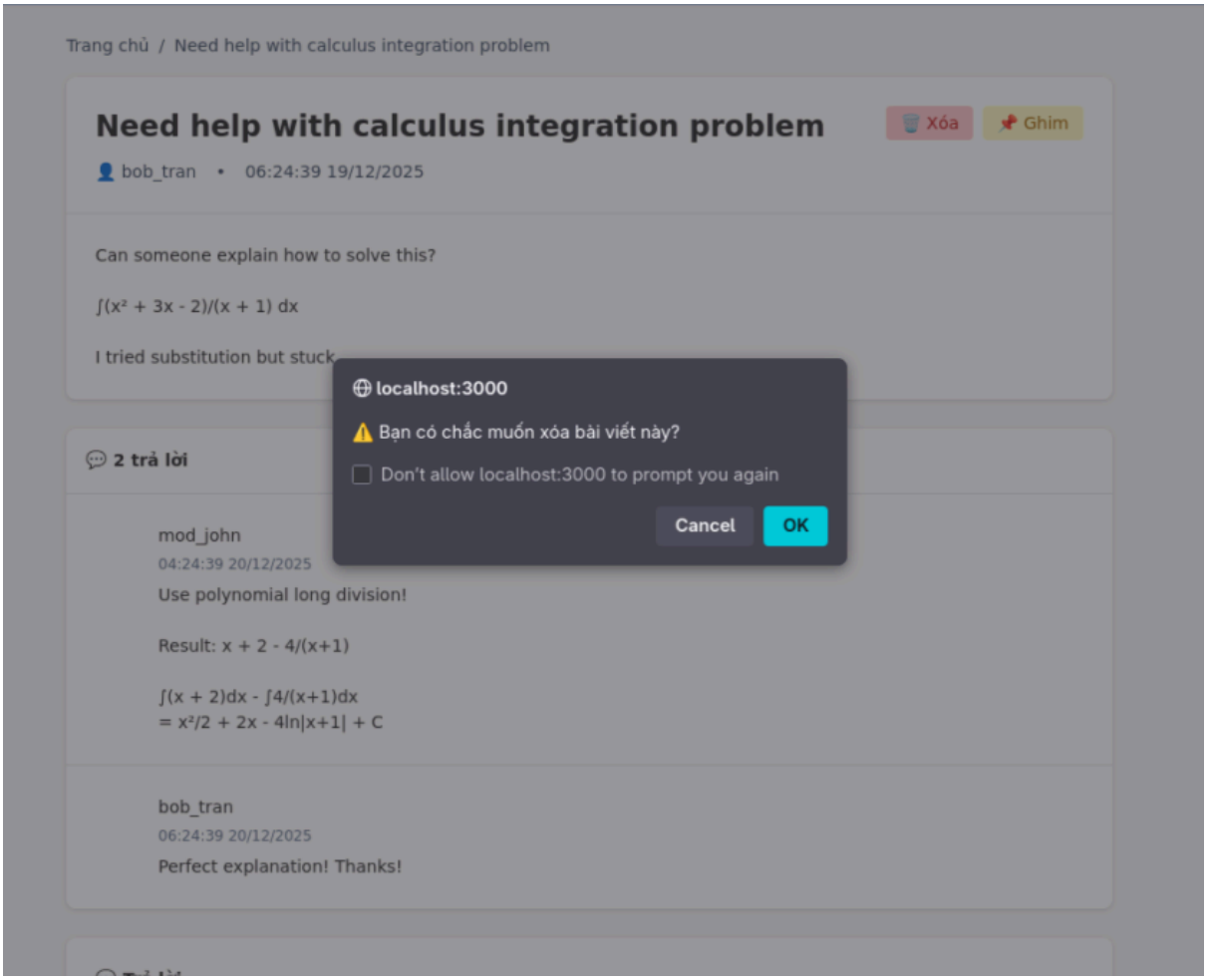


Delete Thread (admin/mod)

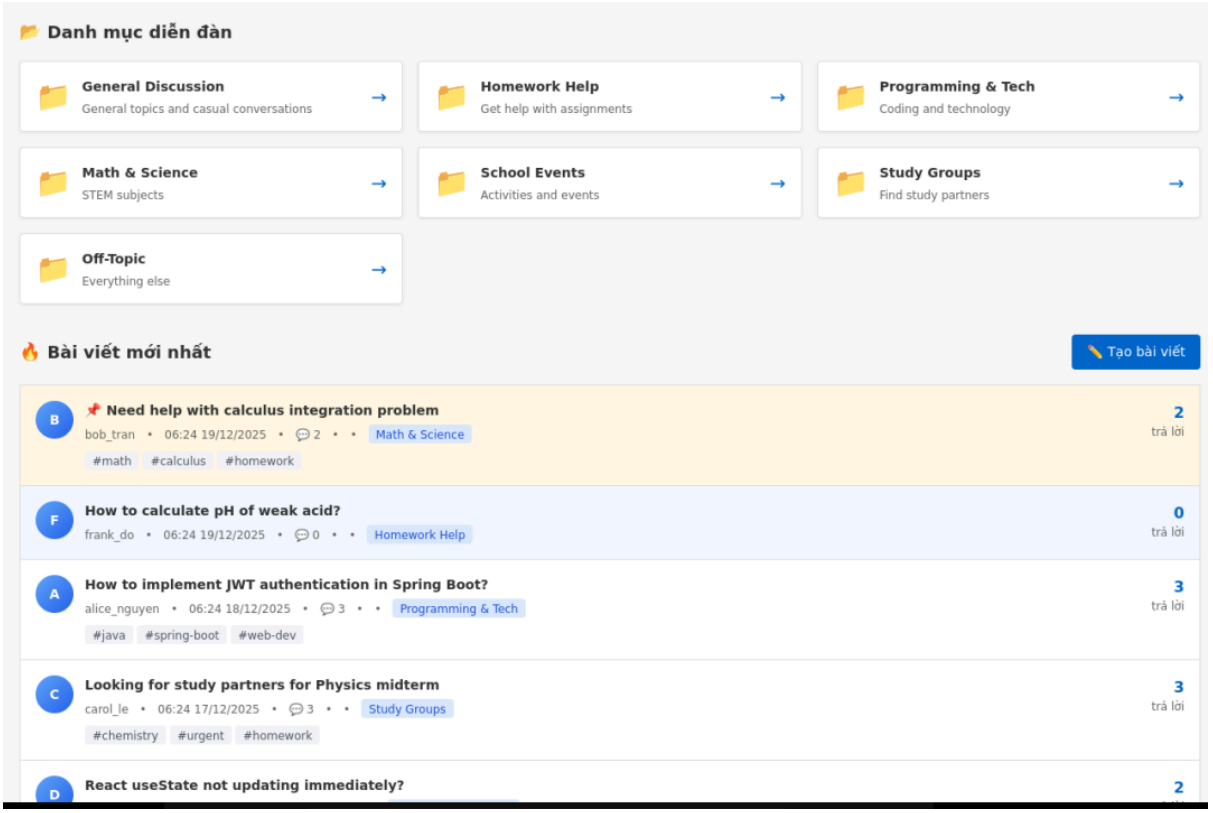


Thread Delete successful

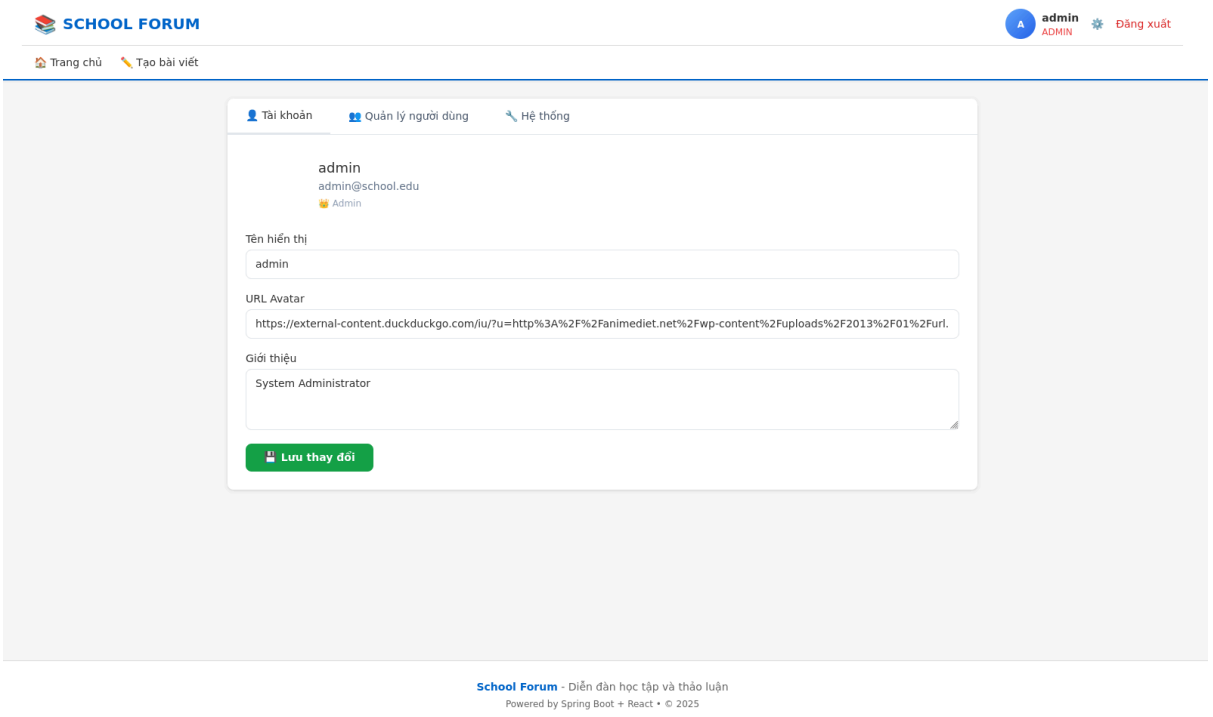



Trang chủ / Need help with calculus integration problem
Need help with calculus integration problem
Xóa
Ghim
bob_tran • 06:24:39 19/12/2025
Can someone explain how to solve this?
∫(x² + 3x - 2)/(x + 1) dx
I tried substitution but stuck
localhost:3000
Bạn có chắc muốn xóa bài viết này?
Don't allow localhost:3000 to prompt you again
Cancel
OK
2 trả lời
mod_john
04:24:39 20/12/2025
Use polynomial long division!
Result: x + 2 - 4/(x+1)
∫(x + 2)dx - ∫4/(x+1)dx
= x²/2 + 2x - 4ln|x+1| + C
bob_tran
06:24:39 20/12/2025
Perfect explanation! Thanks!

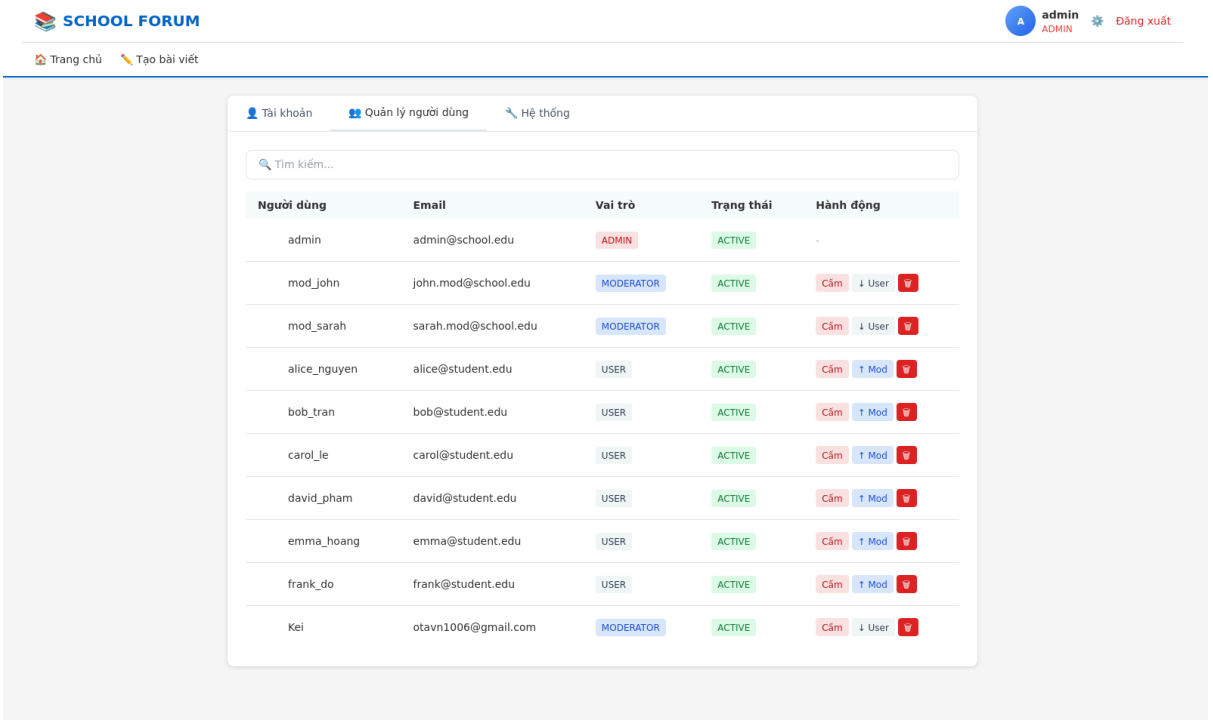
Thread which being pin will be highlight



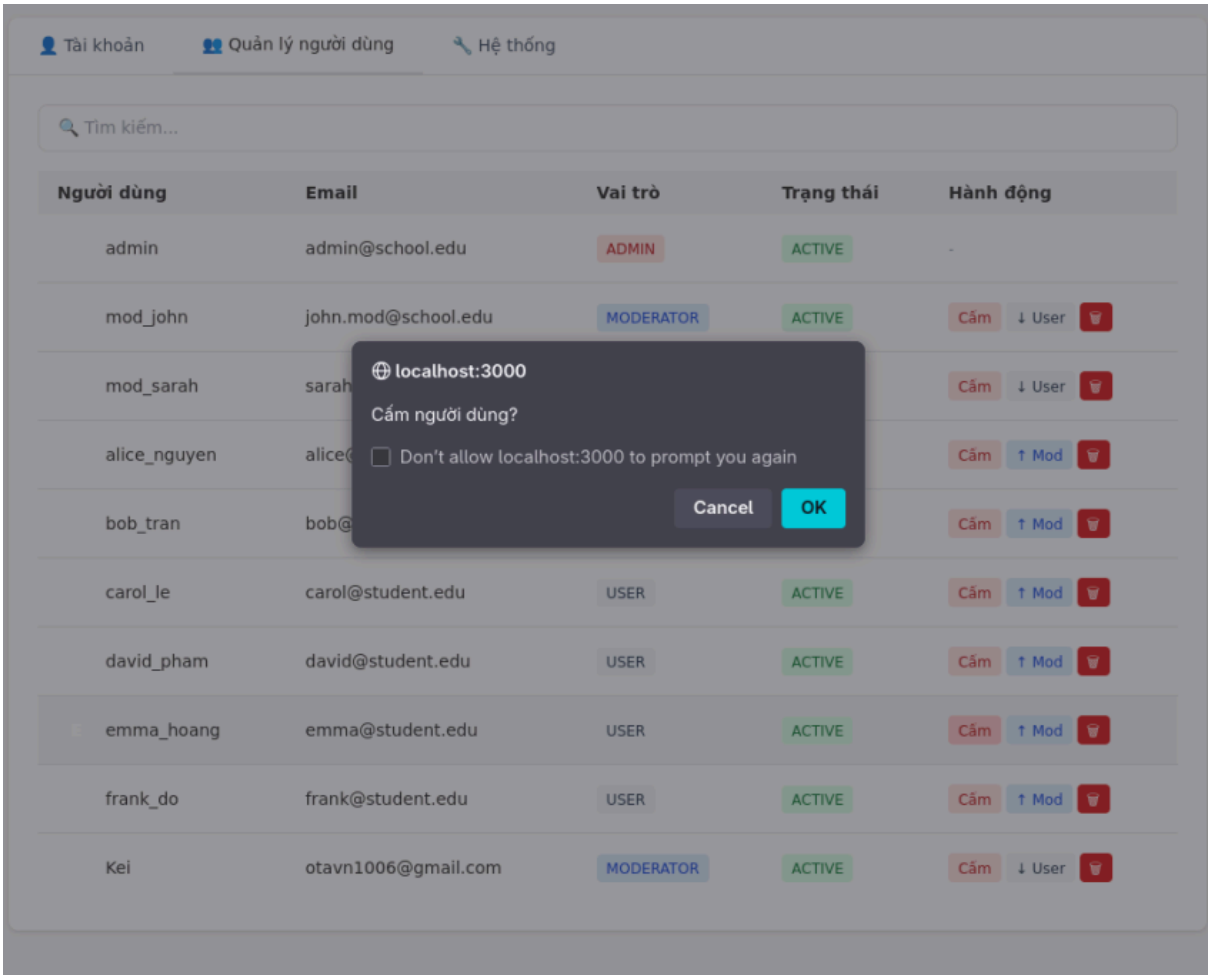
Settings page for admin



This tab only available for admin/mod, admin have full functions but mod can’t delete user

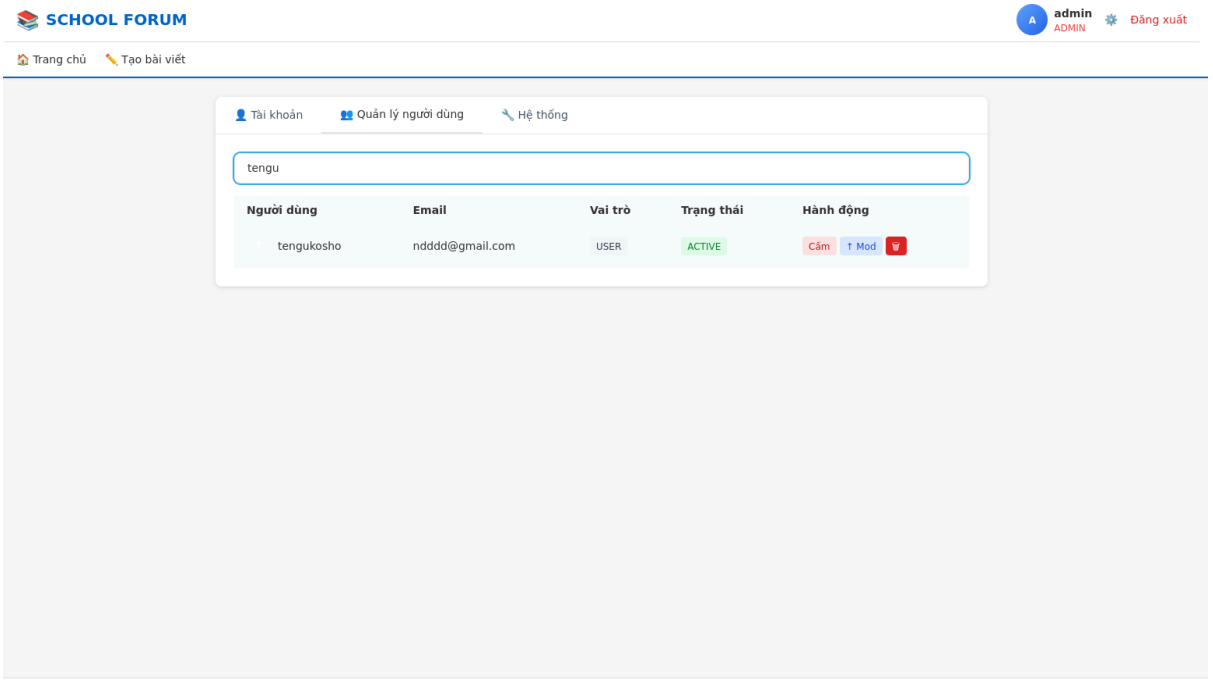


Ban user

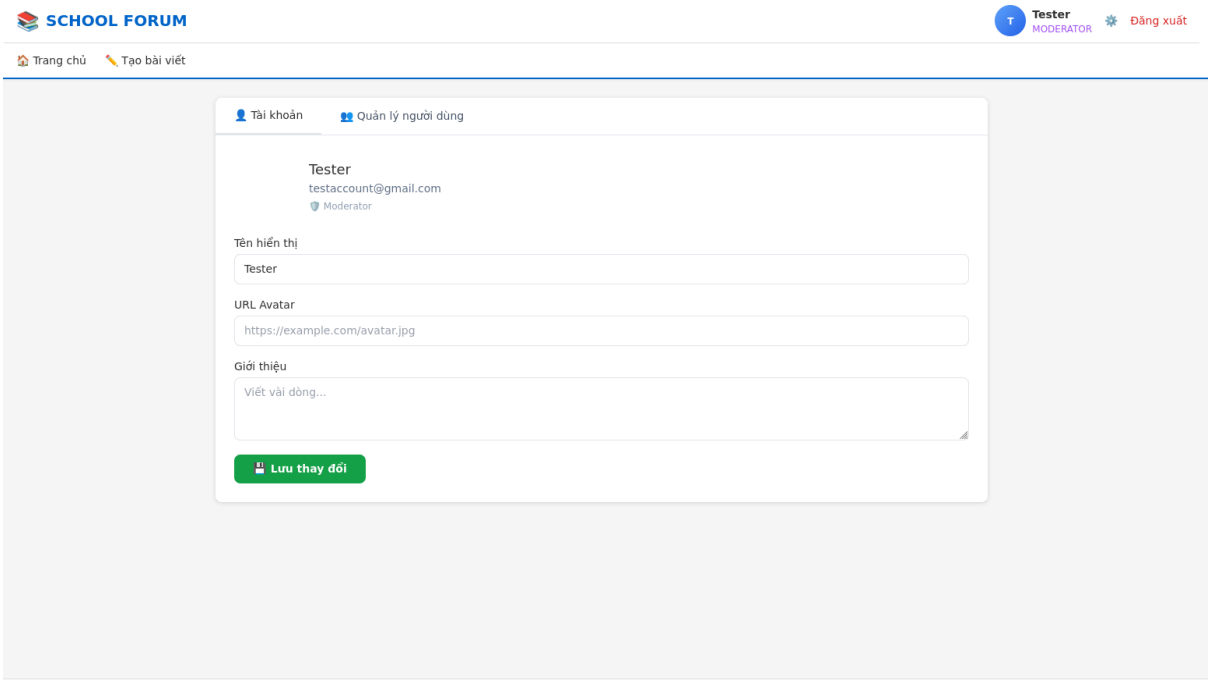


Ban successful

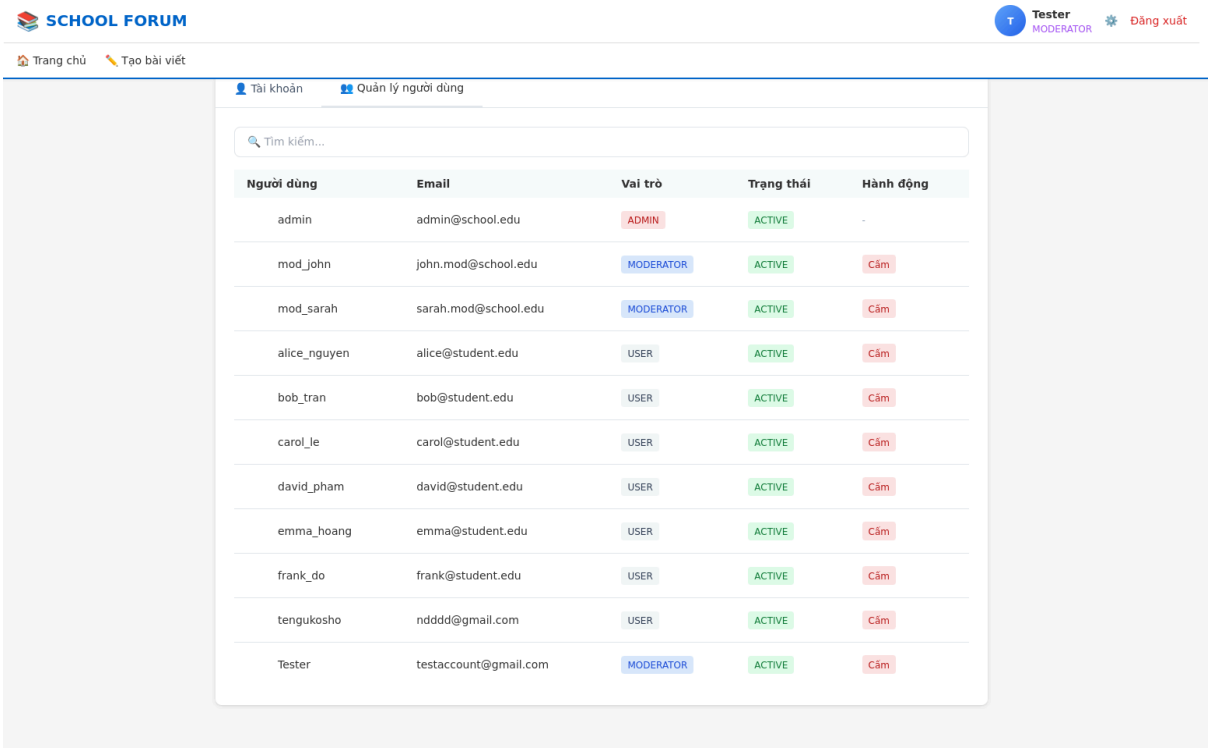
Search function in “ Quản lý người dùng “ with keyword “tengu”



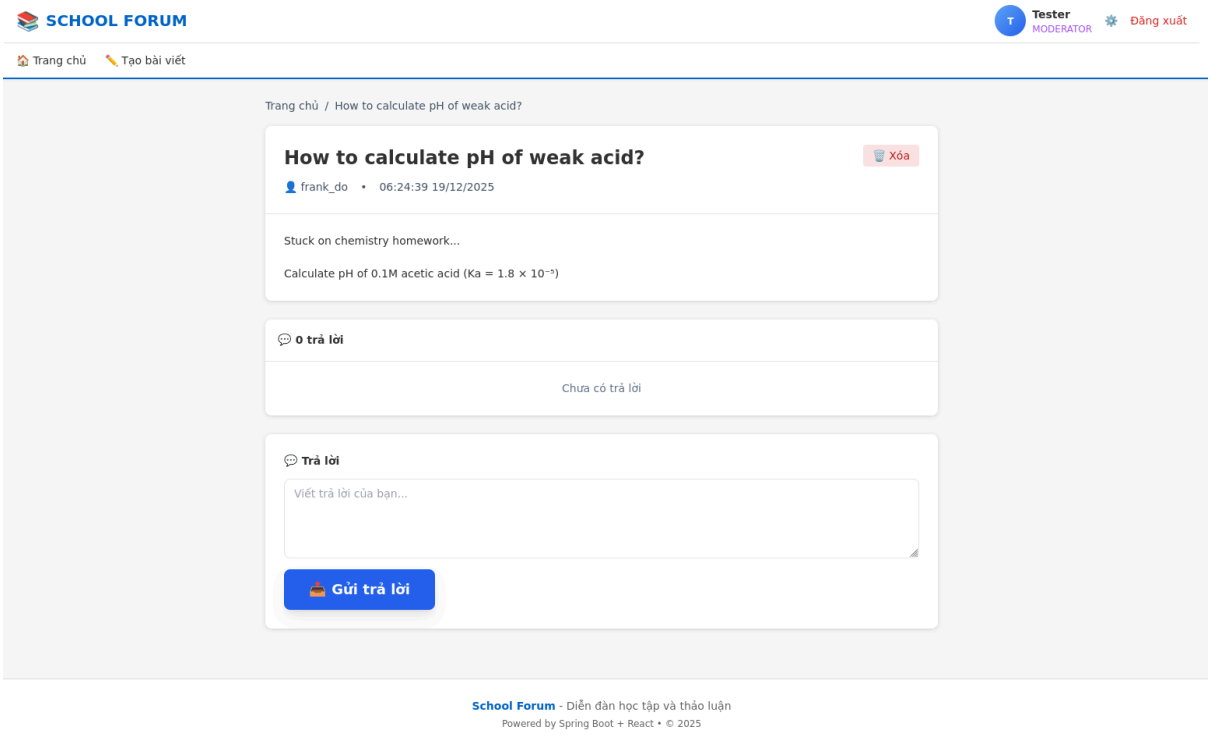
Settings page for mod (2 tab)

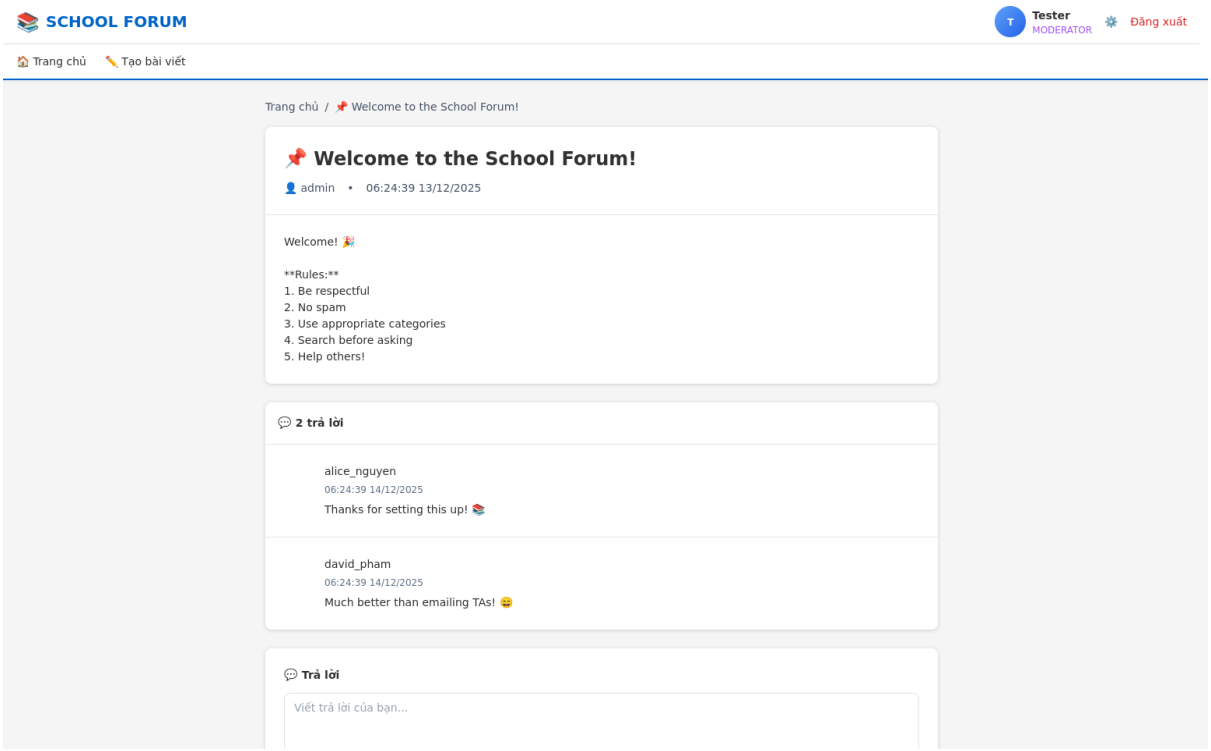


Mod only can ban/unban user

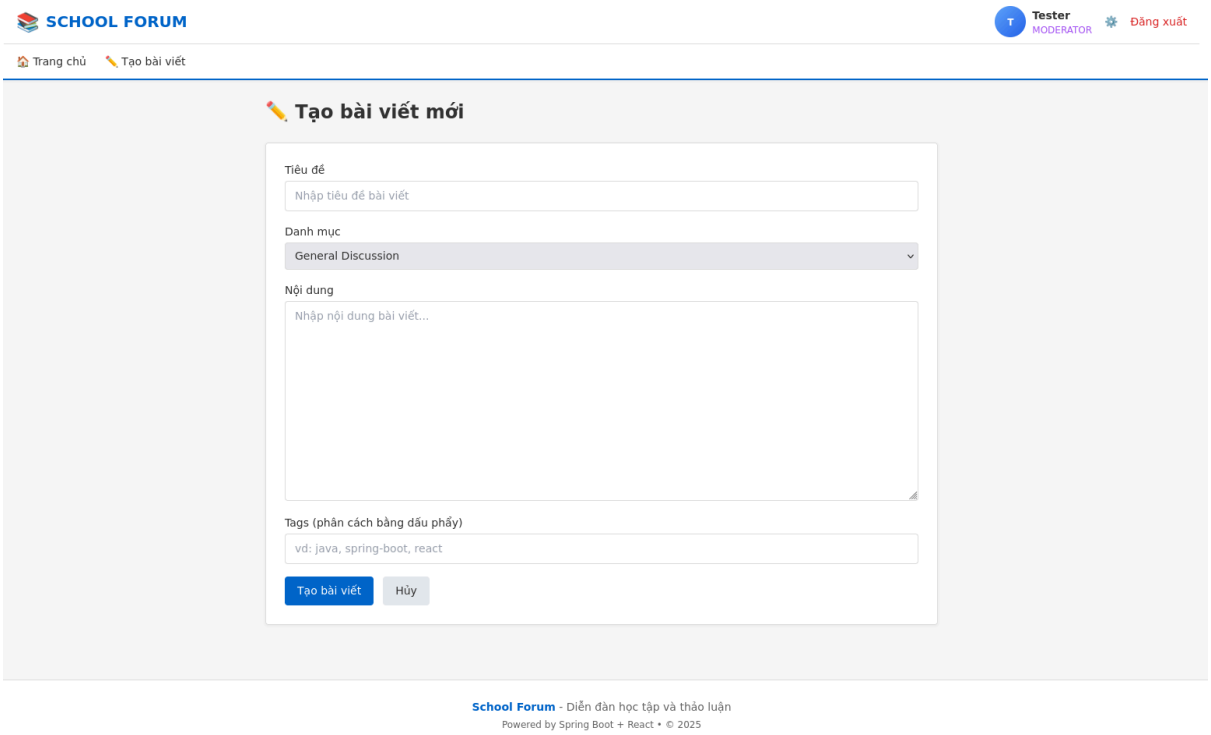


Mod can delete user’s thread but can’t delete admin’s thread





Create new thread (For all roles)



IV. Reference

Oracle. (2024). Java Platform, Standard Edition 21 Documentation. https://docs.oracle.com/en/java/
Spring Project. (2024). Spring Boot Reference Documentation. https://spring.io/projects/spring-boot
ReactJS. (2024). React Documentation. https://react.dev/
MySQL. (2024). MySQL Reference Manual. https://dev.mysql.com/doc/

Vite. (2024). Vite Documentation. https://vitejs.dev/
Tailwind CSS. (2024). Documentation. https://tailwindcss.com/docs
RFC 7519. (2015). JSON Web Token (JWT). https://datatracker.ietf.org/doc/html/rfc7519

V. Future Improvement

Future enhancements of the School Forum Web Application can focus on expanding interactivity, scalability, and system intelligence. A major improvement direction is the implementation of real-time communication features using WebSockets, enabling live updates for new threads, replies, and moderator actions. The platform can also benefit from a more powerful and efficient advanced search system by integrating indexing technologies such as Elasticsearch, allowing users to quickly locate threads, categories, or content across large datasets.

In terms of moderation and content quality, the system may introduce AI-assisted content moderation, helping automatically detect inappropriate posts, spam, or harmful behavior while reducing the manual workload for moderators. Database performance can be further optimized through query tuning, caching layers, and read–write separation, ensuring smooth operation even under high traffic and large-scale deployments.

To improve scalability, maintainability, and deployment flexibility, the application can transition toward a containerized architecture using Docker and Kubernetes, allowing automated scaling, rolling updates, and efficient environment management.

Additional feature improvements include enhancing moderation tools with a thread locking mechanism, allowing moderators to freeze discussions when necessary. The thread subscription system can be expanded to give users fine-grained control over notifications, including options such as mute (no notifications) or receiving only critical updates. Together, these features contribute to a more robust, user-friendly, and future-proof forum platform.