WEB APPLICATION DEVELOPMENT

School Forum Project

GROUP MEMBER

Nguyễn Quốc Trạng - ITCSIU21239
Trần Công Bằng - BEBEIU21189

Overview

Key Features

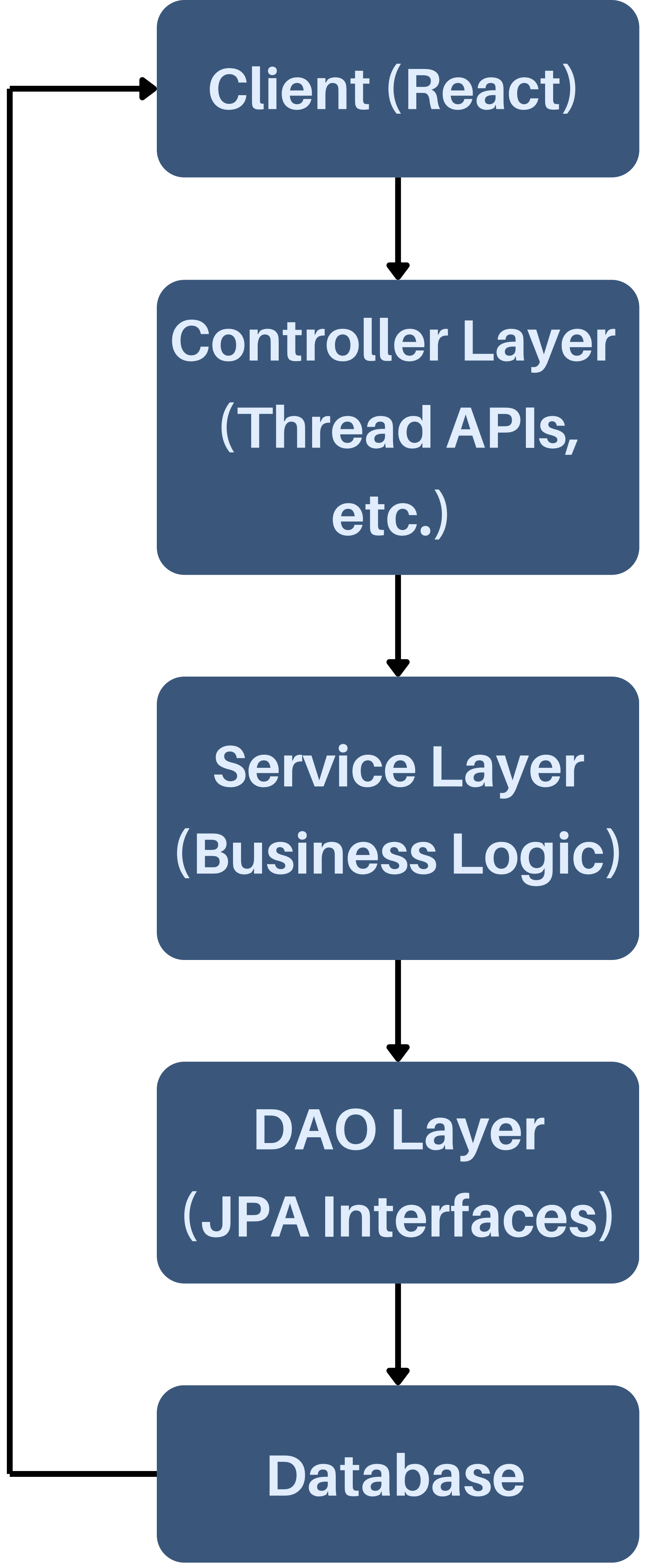
- Secure JWT-based Authentication.
- Role-based Permissions (Admin, Moderator, User).
- Thread and Reply Management.
- Robust RESTful API.

Tech stacks

- Backend: Java 21, Spring Boot, MySQL (MariaDB), Hibernate (JPA).
- Frontend: React, Tailwind CSS.
- Testing & Build Tools: Maven, Postman.

General Architechture

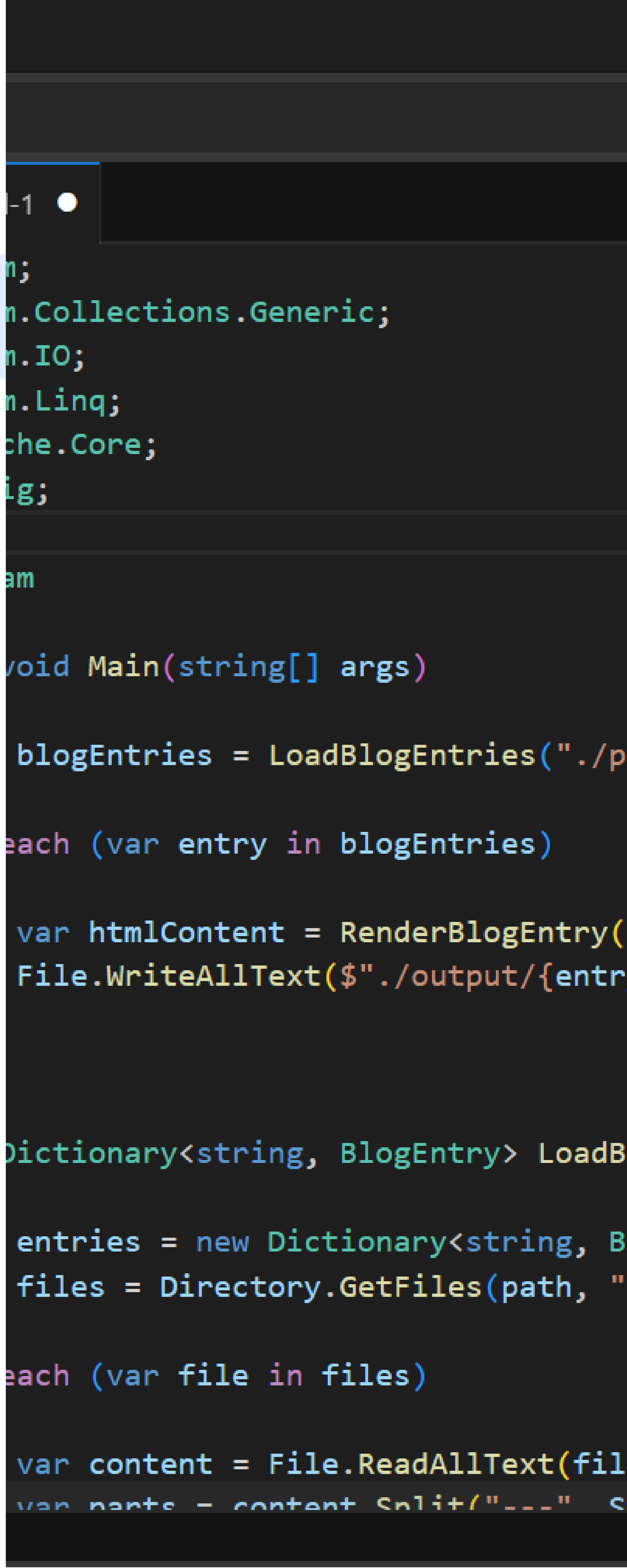
Controller Layer: Handles HTTP requests.
Service Layer: Contains business logic.
DAO Layer: Interacts with the database using JPA.
Model/Entity Layer: Represents database tables.


Client (React)
Controller Layer (Thread APIs, etc.)
Service Layer (Business Logic)
DAO Layer (JPA Interfaces)
Database

General flow:
User Action → Controller → Service → DAO → Database → Response.

Backend Directory Structure

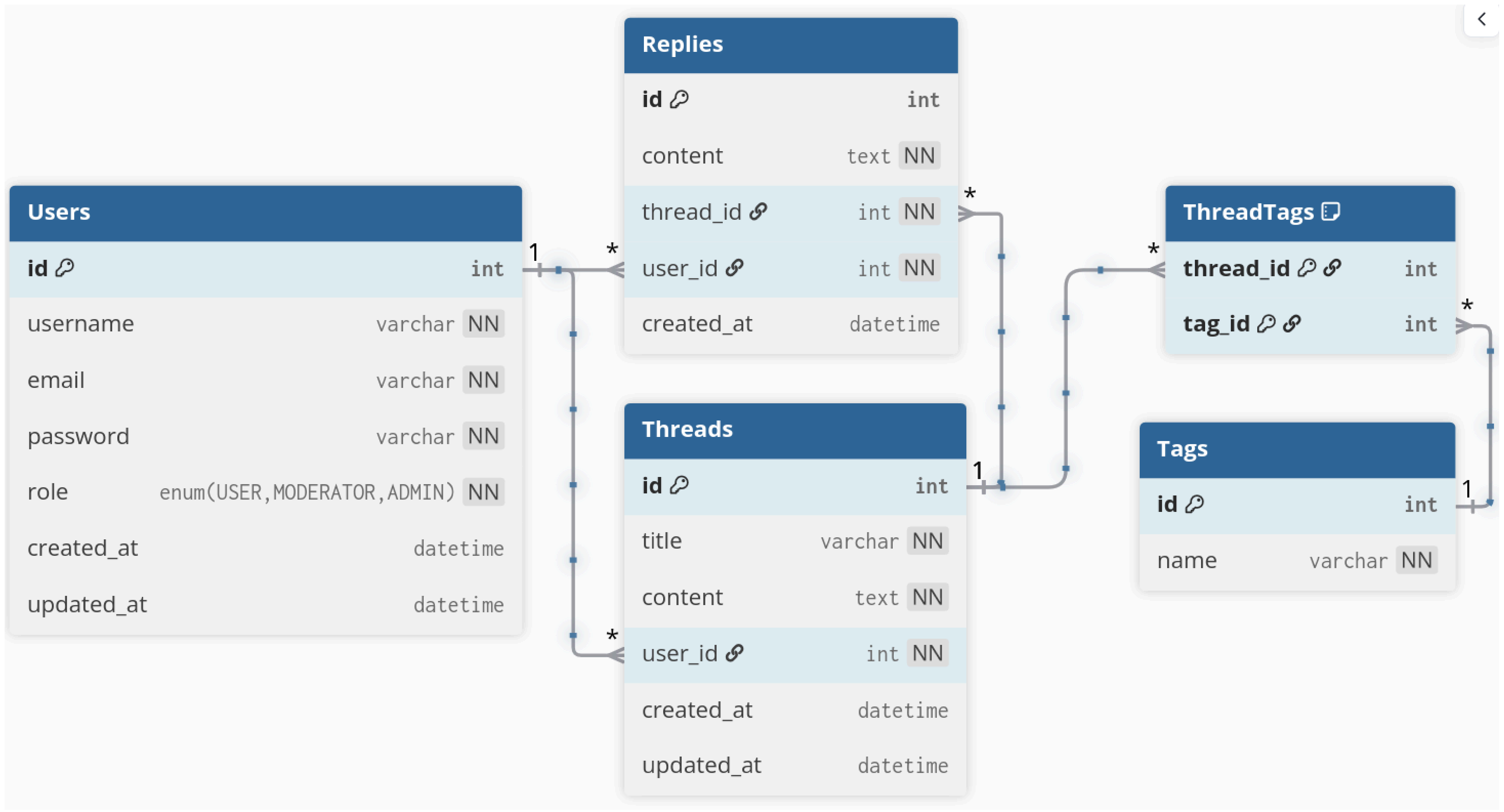
```
back/
  ├──── config/       # Configuration (CORS, JWT, Security)
  ├──── controller/   # Handles REST API requests
  ├──── service/      # Business Logic (Validation, Orchestration)
  ├──── dao/          # Data Access Objects (Database Operations)
  ├──── model/        # Defines Entities (User, Thread, Reply)
  ├──── resources/    # Application Settings (Properties)
  ├──── dto/          # Data Transfer Objects (Request/Response Models)
  ├──── security/     # Authentication and Authorization Logic
```



```
-1 ●
m;
m.Collections.Generic;
m.IO;
m.Linq;
che.Core;
ig;

am

void Main(string[] args)

 blogEntries = LoadBlogEntries("./p

each (var entry in blogEntries)

 var htmlContent = RenderBlogEntry(
 File.WriteAllText($"./output/{entr


Dictionary<string, BlogEntry> LoadB

 entries = new Dictionary<string, B
 files = Directory.GetFiles(path, "

each (var file in files)

 var content = File.ReadAllText(fil
 var parts = content.Split("---", S
```

Backend

Database



Replies		
id	int	
content	text	NN
thread_id	int	NN
user_id	int	NN
created_at	datetime	

Users		
id	int	
username	varchar	NN
email	varchar	NN
password	varchar	NN
role	enum(USER,MODERATOR,ADMIN)	NN
created_at	datetime	
updated_at	datetime	

Threads		
id	int	
title	varchar	NN
content	text	NN
user_id	int	NN
created_at	datetime	
updated_at	datetime	

ThreadTags	
thread_id	int
tag_id	int

Tags		
id	int	
name	varchar	NN

- **Users → Threads (One-to-Many)**
- **Users → Replies (One-to-Many)**
- **Threads → Replies (One-to-Many)**
- **Threads ↔ Tags (Many-to-Many)**

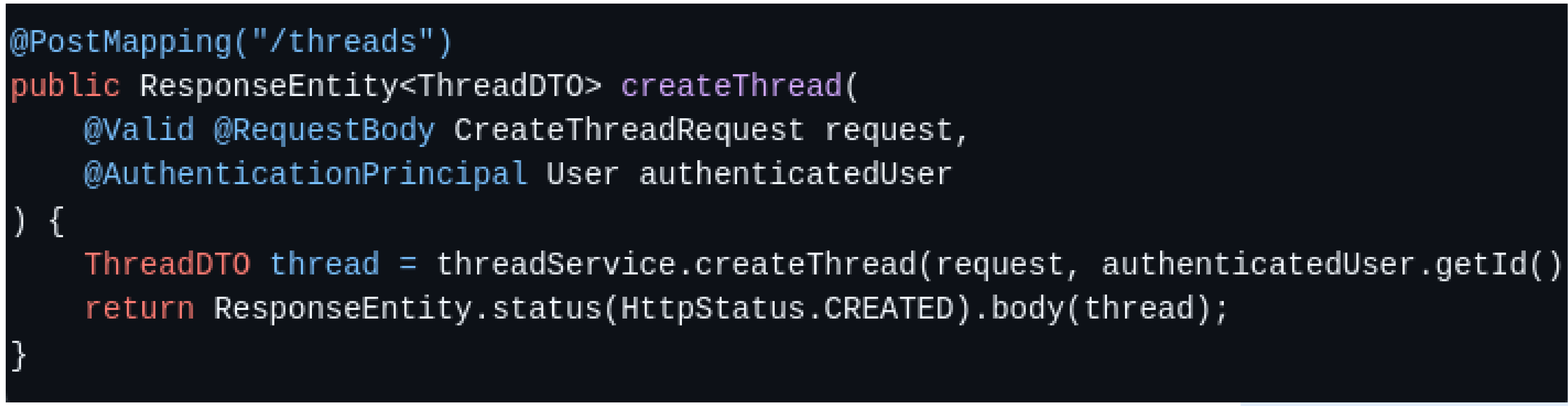
Backend

Use case: Create a thread

Flow: Steps for creating a thread.

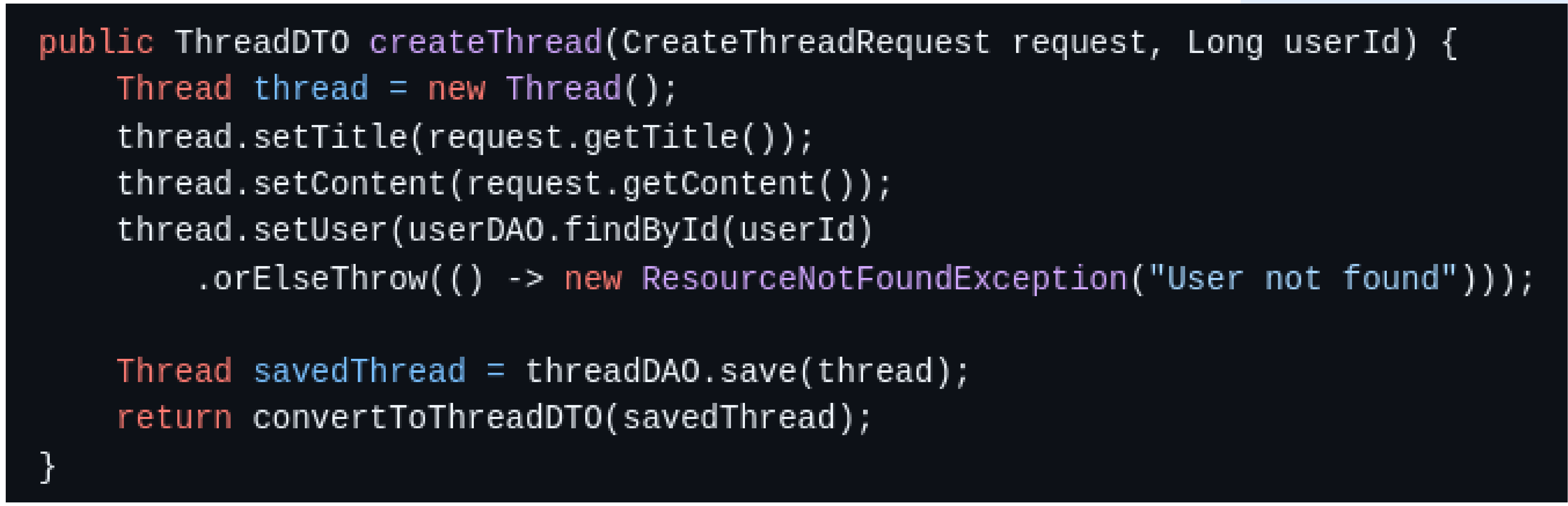
1. Client sends an HTTP POST request with thread details.
2. Controller captures the request, validates input, and sends it to the Service layer.
3. Service processes the data and calls the DAO for database operations.
4. DAO persists the thread in the database and returns the result.
5. Controller sends a JSON response back to the client with the success status and data.

Controller



```java
@PostMapping("/threads")
public ResponseEntity<ThreadDTO> createThread(
    @Valid @RequestBody CreateThreadRequest request,
    @AuthenticationPrincipal User authenticatedUser
) {
    ThreadDTO thread = threadService.createThread(request, authenticatedUser.getId()
    return ResponseEntity.status(HttpStatus.CREATED).body(thread);
}
```

Service



```java
public ThreadDTO createThread(CreateThreadRequest request, Long userId) {
    Thread thread = new Thread();
    thread.setTitle(request.getTitle());
    thread.setContent(request.getContent());
    thread.setUser(userDAO.findById(userId)
        .orElseThrow(() -> new ResourceNotFoundException("User not found")));

    Thread savedThread = threadDAO.save(thread);
    return convertToThreadDTO(savedThread);
}
```

Backend

Use case: Create a thread

Flow: Steps for creating a thread.

1. Client sends an HTTP POST request with thread details.
2. Controller captures the request, validates input, and sends it to the Service layer.
3. Service processes the data and calls the DAO for database operations.
4. DAO persists the thread in the database and returns the result.
5. Controller sends a JSON response back to the client with the success status and data.

DAO

```
@Repository
public interface UserDAO extends JpaRepository<User, Long> {
    // Inherits save() and findById() from JpaRepository
}
```

Response

```
HTTP Status: 201 Created
{
  "id": 1,
  "title": "How to set up Spring Boot",
  "content": "Having trouble setting up Spring Boot. Can someone help?",
  "userId": 101,
  "username": "JohnDoe",
  "createdAt": "2025-12-22T10:00:00Z"
}
```

Backend

Use case: Ban a user

Flow: Steps for banning a user.

1. Client sends an HTTP PUT request to /api/users/{id}/ban.
2. The **UserController** receives the request, extracts the id from the URL, and forwards it to the UserService.
3. The **UserService** fetches the user from the database via the **UserDAO**. Updates the banned status of the user to **true**.
4. The **UserDAO.save**() method persists the updated user entity in the database, setting the banned flag to true.
5. The **UserService** converts the updated User entity to a **UserDTO**.
6. The **UserController** sends the **UserDTO** as the response to the client, confirming the user's ban status.

Request

```
PUT /api/users/101/ban HTTP/1.1
Host: api.schoolforum.com
Authorization: Bearer <admin-token>
```

Controller

```
@PutMapping("/{id}/ban")
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<UserDTO> banUser(@PathVariable Long id) {
    // Step 1: Call the Service layer
    UserDTO user = userService.banUser(id);

    // Step 2: Send the response to the client
    return ResponseEntity.ok(user);
}
```

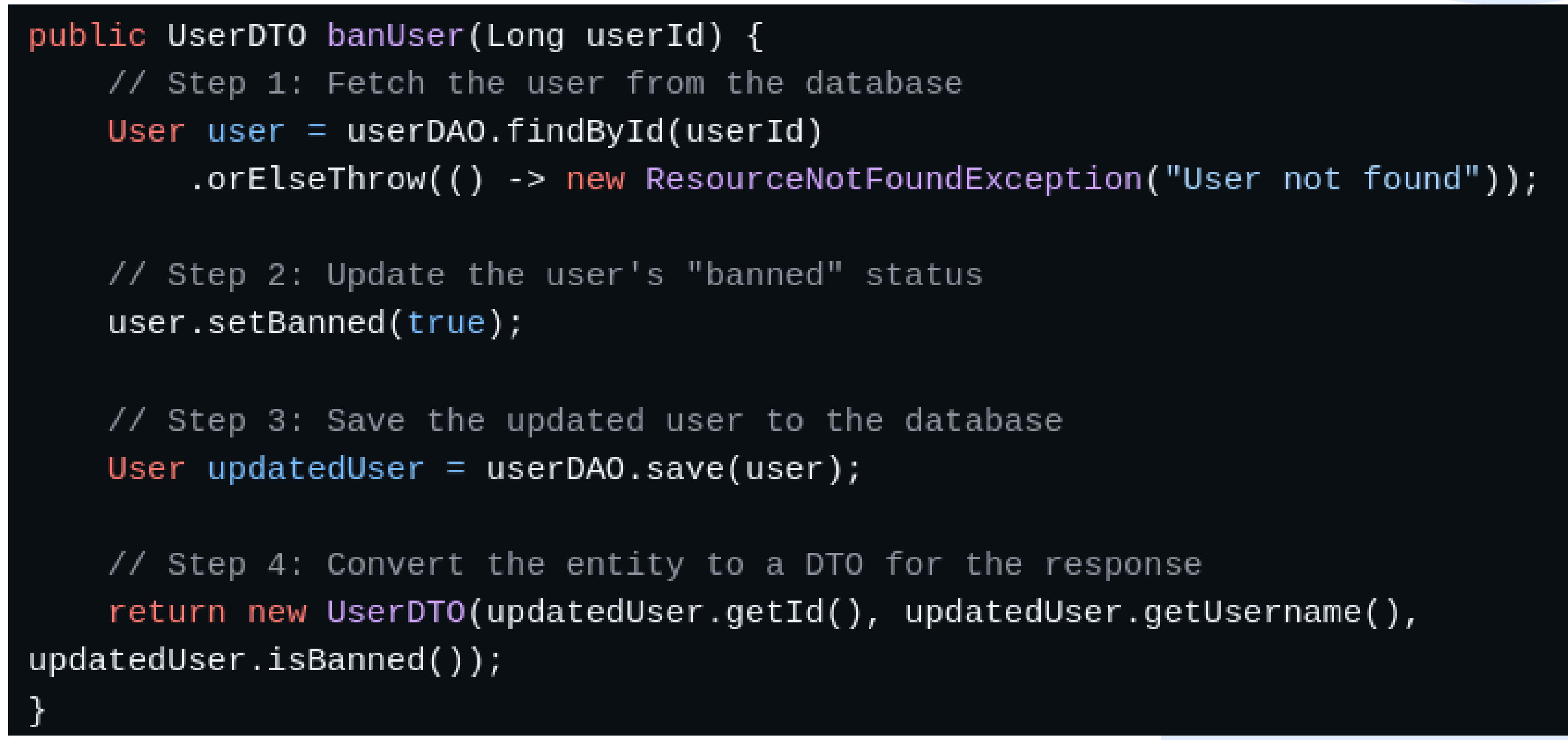
Backend

Use case: Ban a user

Flow: Steps for banning a user.

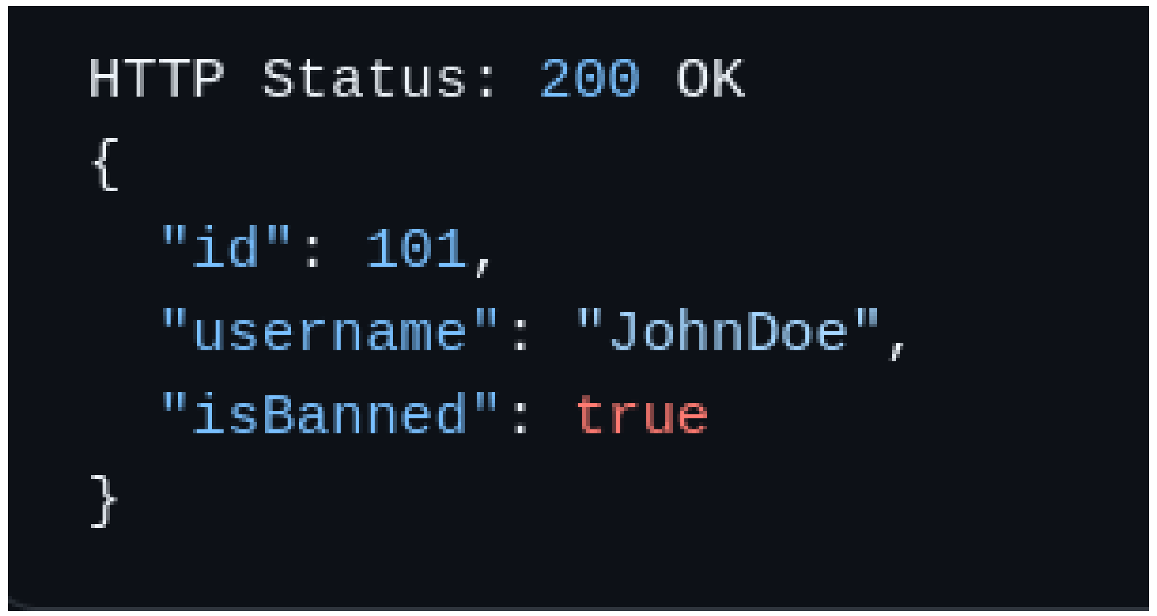
1. Client sends an HTTP PUT request to /api/users/{id}/ban.
2. The **UserController** receives the request, extracts the id from the URL, and forwards it to the UserService.
3. The **UserService** fetches the user from the database via the **UserDAO**. Updates the banned status of the user to **true**.
4. The **UserDAO.save**() method persists the updated user entity in the database, setting the banned flag to true.
5. The **UserService** converts the updated User entity to a **UserDTO**.
6. The **UserController** sends the **UserDTO** as the response to the client, confirming the user's ban status.

Service



```java
public UserDTO banUser(Long userId) {
    // Step 1: Fetch the user from the database
    User user = userDAO.findById(userId)
        .orElseThrow(() -> new ResourceNotFoundException("User not found"));

    // Step 2: Update the user's "banned" status
    user.setBanned(true);

    // Step 3: Save the updated user to the database
    User updatedUser = userDAO.save(user);

    // Step 4: Convert the entity to a DTO for the response
    return new UserDTO(updatedUser.getId(), updatedUser.getUsername(),
updatedUser.isBanned());
}
```

Response



```
HTTP Status: 200 OK
{
  "id": 101,
  "username": "JohnDoe",
  "isBanned": true
}
```

FRONTEND TECHNOLOGIES USED

Frontend Technologies Used

React 18.2.0
Component-based UI library

React Router 6.20.1
Client-side routing and navigation

Axios 1.6.2
HTTP client for API communication

Tailwind CSS 3.3.6
Utility-first CSS framework

Vite 5.0.8
Modern build tool and dev server

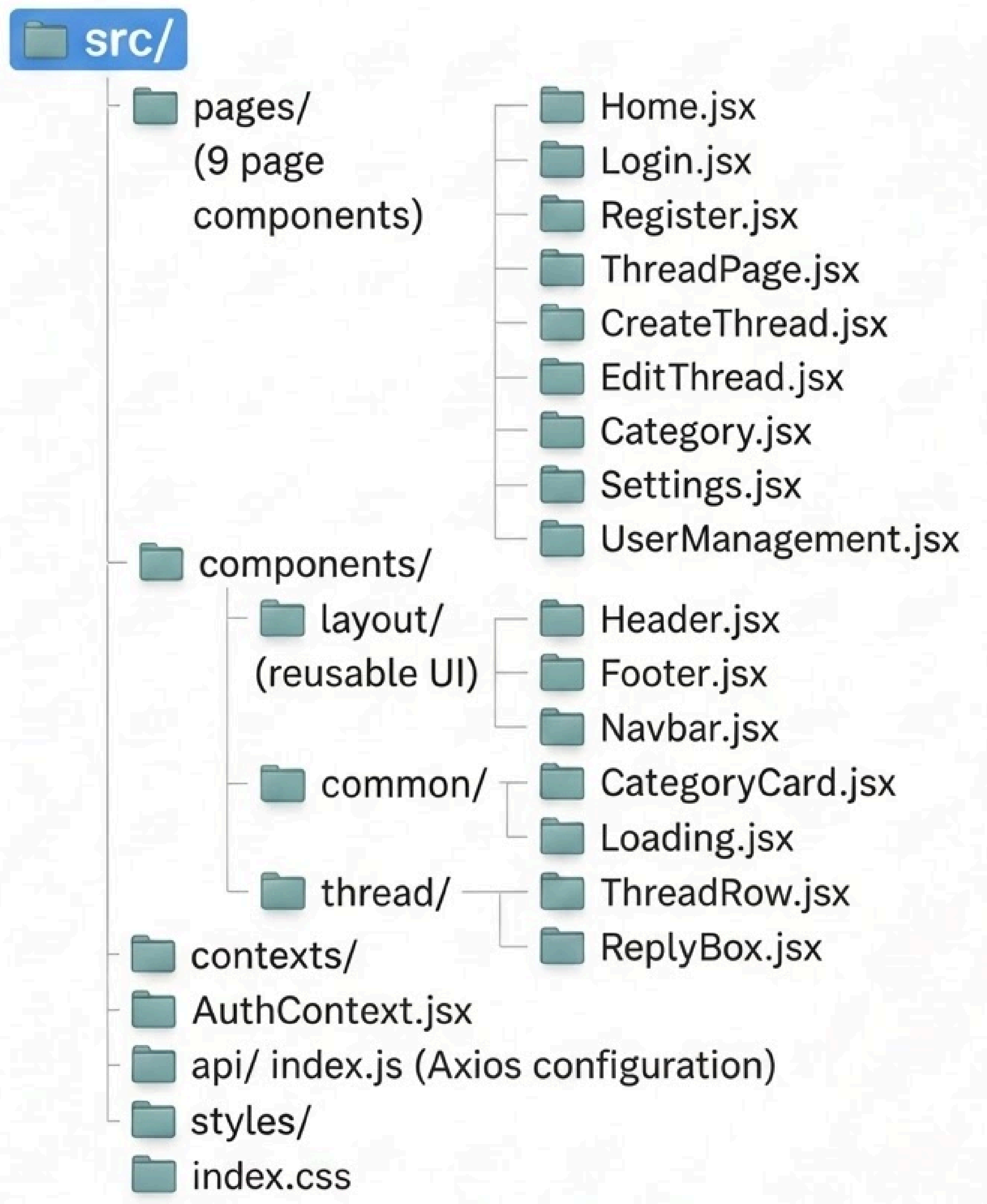
Context API
Global state management

Why These Technologies

- **React**: Industry standard, reusable components
- **Vite**: Super-fast development with HMR
- **Tailwind**: Rapid styling with utility classes
- **Axios**: JWT support via interceptors

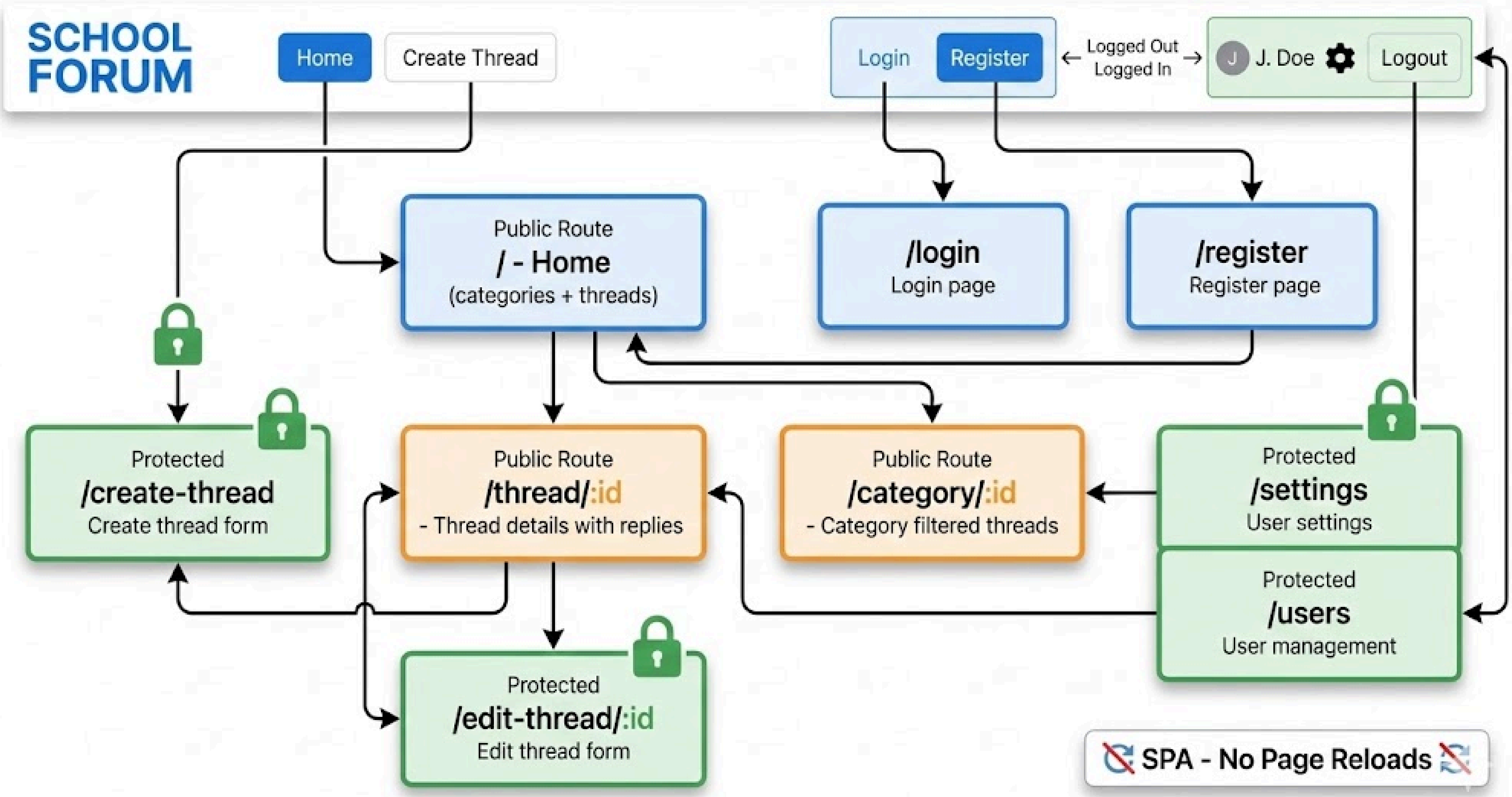
FRONTEND ARCHITECTURE & FOLDER STRUCTURE

React Frontrend Project Structure

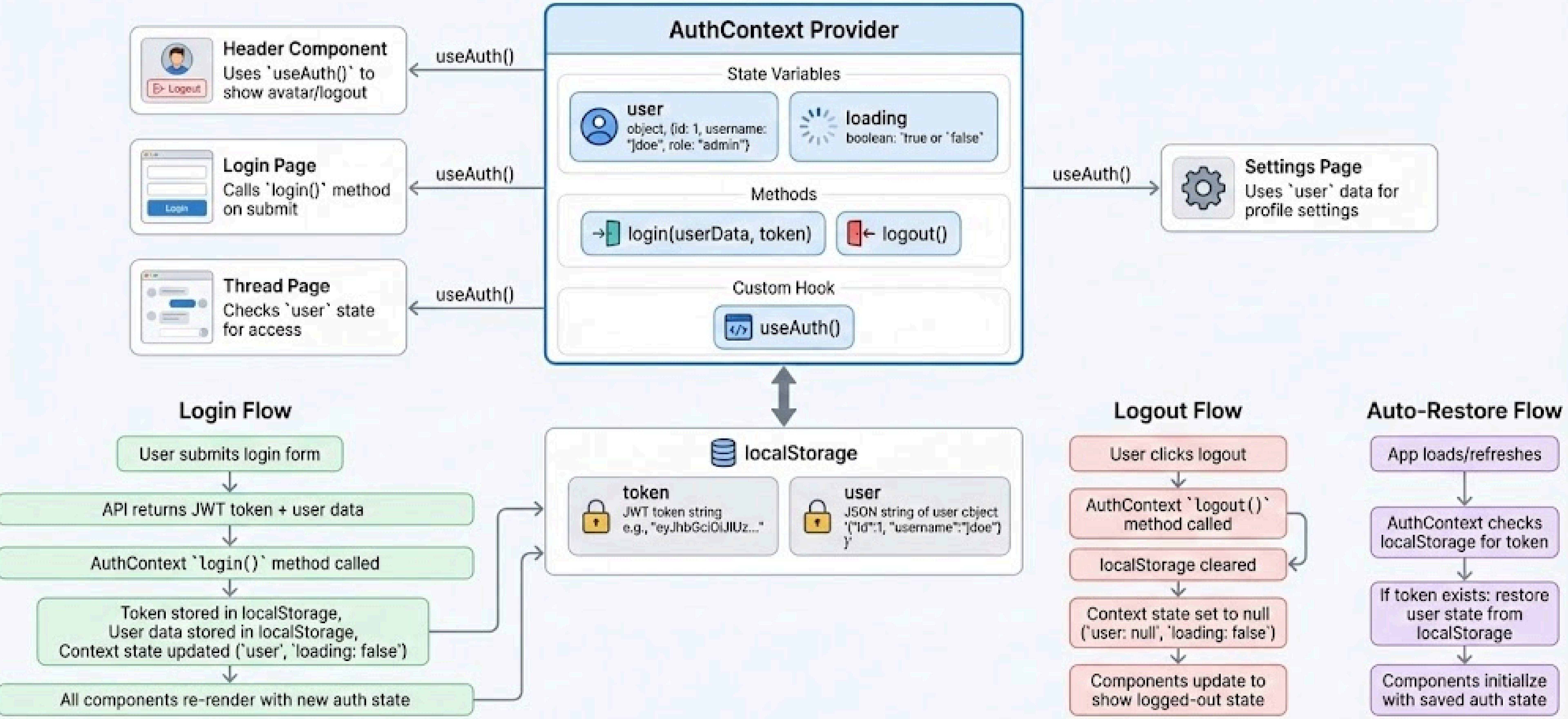


```
src/
├── pages/ (9 page components)
│   ├── Home.jsx
│   ├── Login.jsx
│   ├── Register.jsx
│   ├── ThreadPage.jsx
│   ├── CreateThread.jsx
│   ├── EditThread.jsx
│   ├── Category.jsx
│   ├── Settings.jsx
│   └── UserManagement.jsx
├── components/
│   ├── layout/ (reusable UI)
│   │   ├── Header.jsx
│   │   ├── Footer.jsx
│   │   └── Navbar.jsx
│   ├── common/
│   │   ├── CategoryCard.jsx
│   │   └── Loading.jsx
│   └── thread/
│       ├── ThreadRow.jsx
│       └── ReplyBox.jsx
├── contexts/
├── AuthContext.jsx
├── api/ index.js (Axios configuration)
├── styles/
└── index.css
```

ROUTING & NAVIGATION


SCHOOL FORUM
Home
Create Thread
Login
Register
Logged Out
Logged In
J
J. Doe
Logout
Public Route
/ - Home
(categories + threads)
/login
Login page
/register
Register page
Protected
/create-thread
Create thread form
Public Route
/thread/:id
- Thread details with replies
Public Route
/category/:id
- Category filtered threads
Protected
/settings
User settings
Protected
/users
User management
Protected
/edit-thread/:id
Edit thread form
SPA - No Page Reloads

AUTHENTICATION STATE MANAGEMENT


AuthContext Provider
State Variables
user
object, {id: 1, username: "jdoe", role: "admin"}
loading
boolean: `true` or `false`
Methods
login(userData, token)
logout()
Custom Hook
useAuth()
useAuth()
Header Component
Uses `useAuth()` to show avatar/logout
Logout
useAuth()
Login Page
Calls `login()` method on submit
Login
useAuth()
Thread Page
Checks `user` state for access
useAuth()
Settings Page
Uses `user` data for profile settings
localStorage
token
JWT token string e.g., "eyJhbGciOiJIUz..."
user
JSON string of user cbject '{"id":1, "username":"jdoe"}'
Login Flow
User submits login form
API returns JWT token + user data
AuthContext `login()` method called
Token stored in localStorage, User data stored in localStorage, Context state updated (`user`, `loading: false`)
All components re-render with new auth state
Logout Flow
User clicks logout
AuthContext `logout()` method called
localStorage cleared
Context state set to null (`user: null`, `loading: false`)
Components update to show logged-out state
Auto-Restore Flow
App loads/refreshes
AuthContext checks localStorage for token
If token exists: restore user state from localStorage
Components initialize with saved auth state

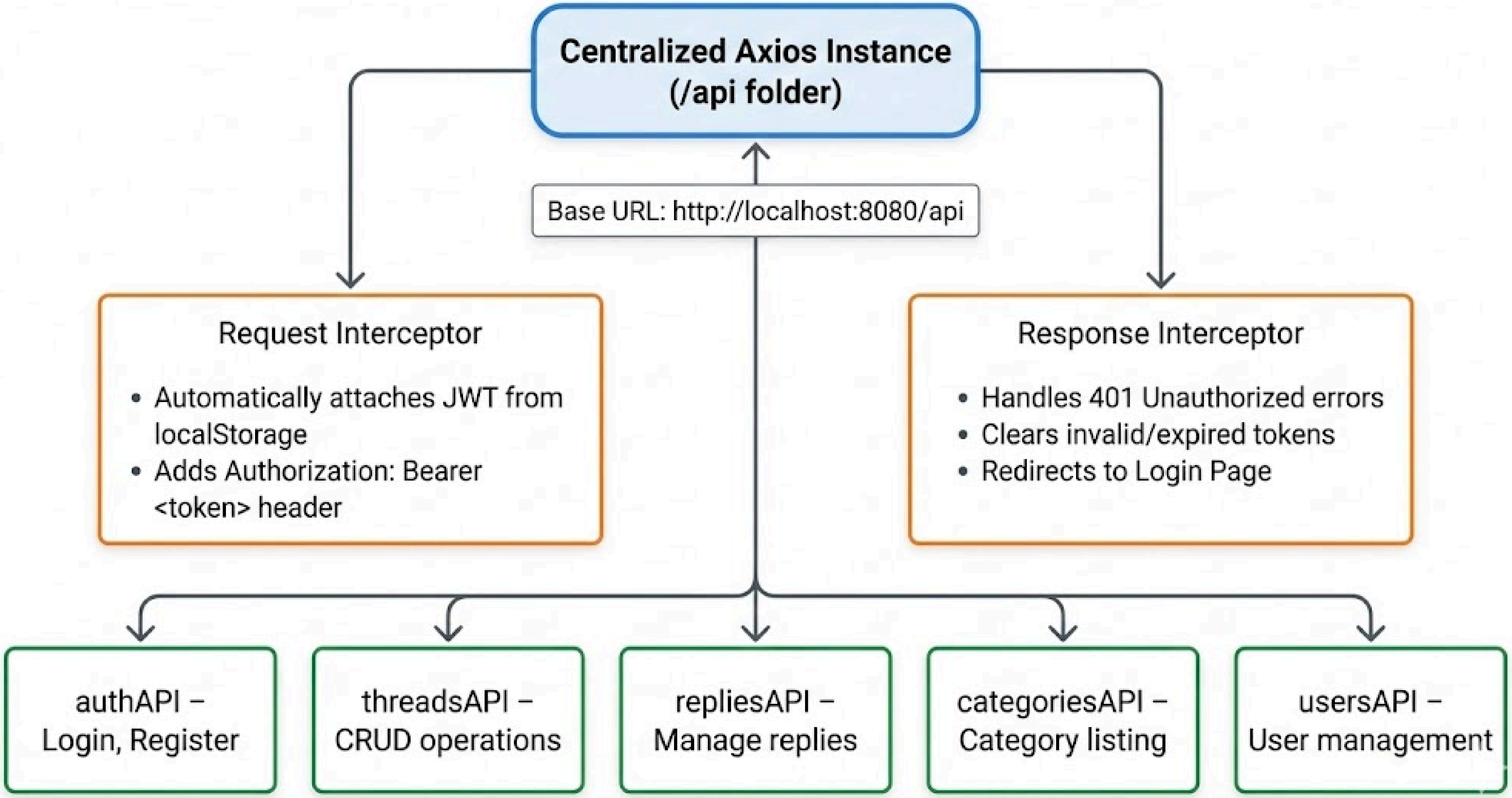
No Prop Drilling
Eliminates passing props through multiple levels, simplifying component hierarchy.

Global State
Accessible from any component in the application tree.

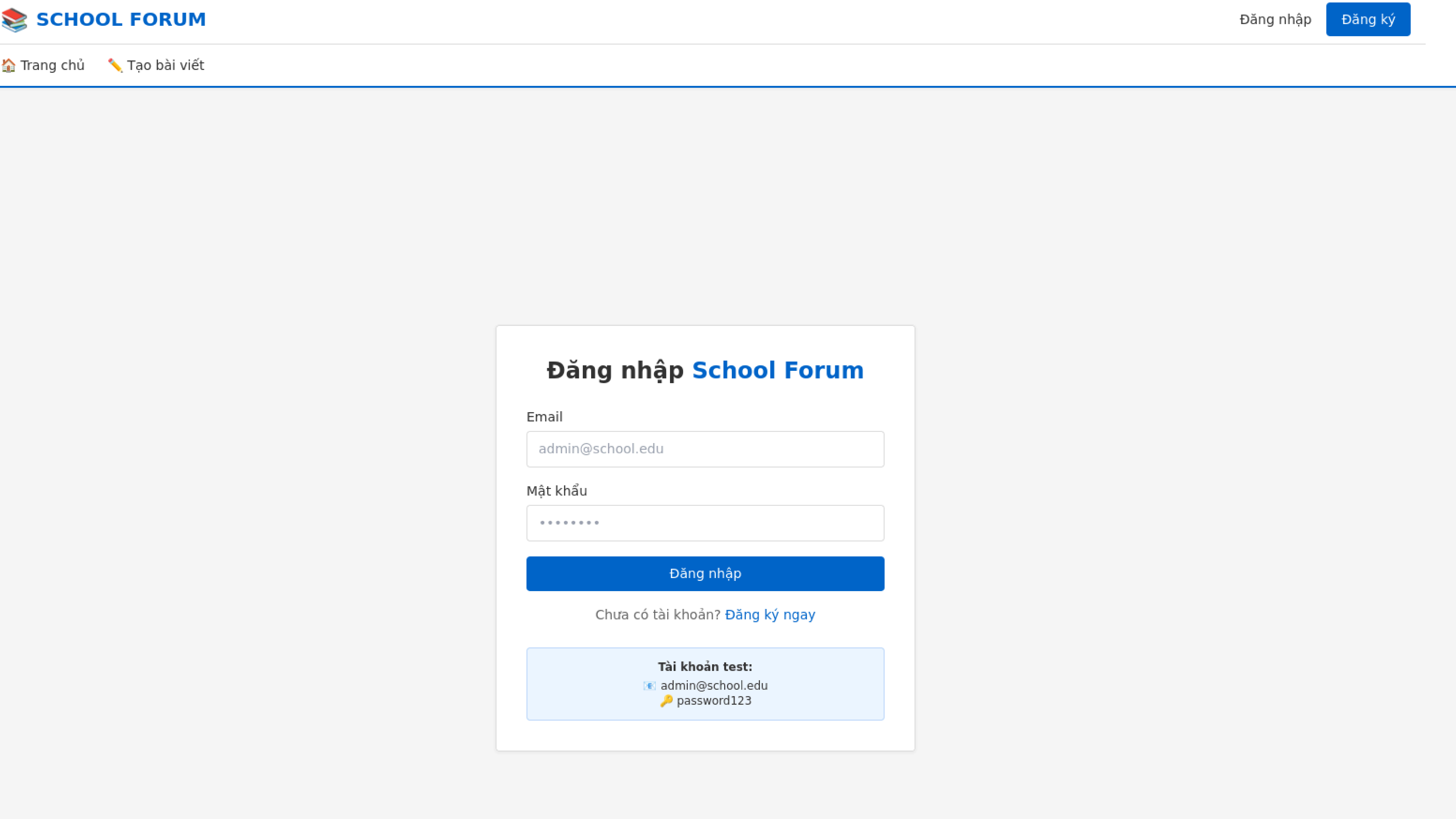
Persistence
localStorage maintains state across page refreshes and browser sessions.

Custom Hook
`useAuth()` provides an easy, abstract way to access auth state and methods.

API INTEGRATION WITH AXIOS


Centralized Axios Instance (/api folder)
Base URL: http://localhost:8080/api
Request Interceptor
• Automatically attaches JWT from localStorage
• Adds Authorization: Bearer <token> header
Response Interceptor
• Handles 401 Unauthorized errors
• Clears invalid/expired tokens
• Redirects to Login Page
authAPI – Login, Register
threadsAPI – CRUD operations
repliesAPI – Manage replies
categoriesAPI – Category listing
usersAPI – User management

USER LOGIN & REGISTRATION UI



📚 SCHOOL FORUM

Đăng nhập Đăng ký

🏠 Trang chủ ✏️ Tạo bài viết

Đăng nhập School Forum

Email

admin@school.edu

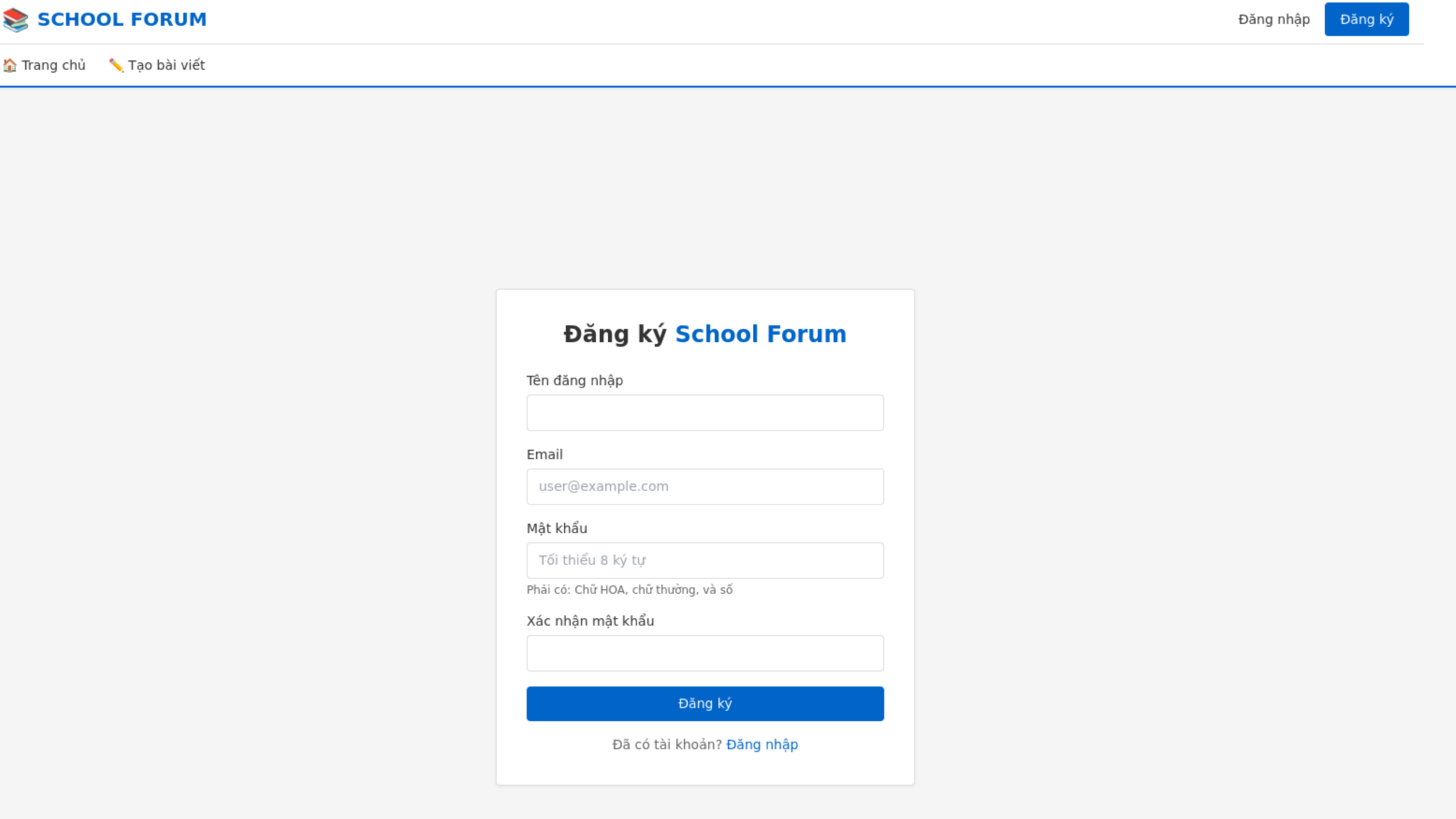
Mật khẩu

•••••••••

Đăng nhập

Chưa có tài khoản? Đăng ký ngay

Tài khoản test:
📧 admin@school.edu
🔑 password123



📚 SCHOOL FORUM

Đăng nhập | Đăng ký

🏠 Trang chủ ✏️ Tạo bài viết

Đăng ký School Forum

Tên đăng nhập

Email

user@example.com

Mật khẩu

Tối thiểu 8 ký tự

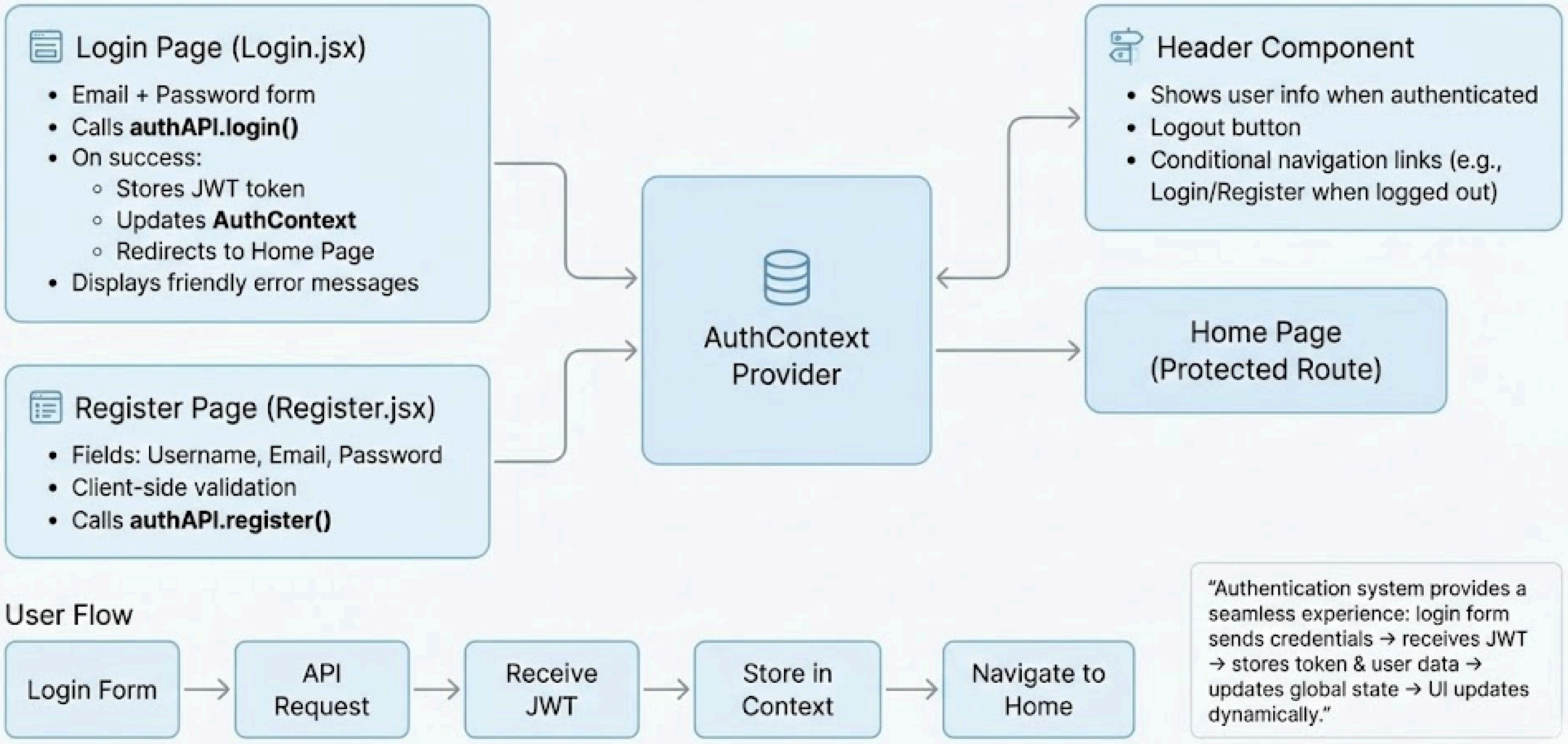
Phải có: Chữ HOA, chữ thường, và số

Xác nhận mật khẩu

Đăng ký

Đã có tài khoản? Đăng nhập

User Login & Registration UI Architecture



Login Page (Login.jsx)

- Email + Password form
- Calls **authAPI.login()**
- On success:
  - Stores JWT token
  - Updates **AuthContext**
  - Redirects to Home Page
- Displays friendly error messages

Register Page (Register.jsx)

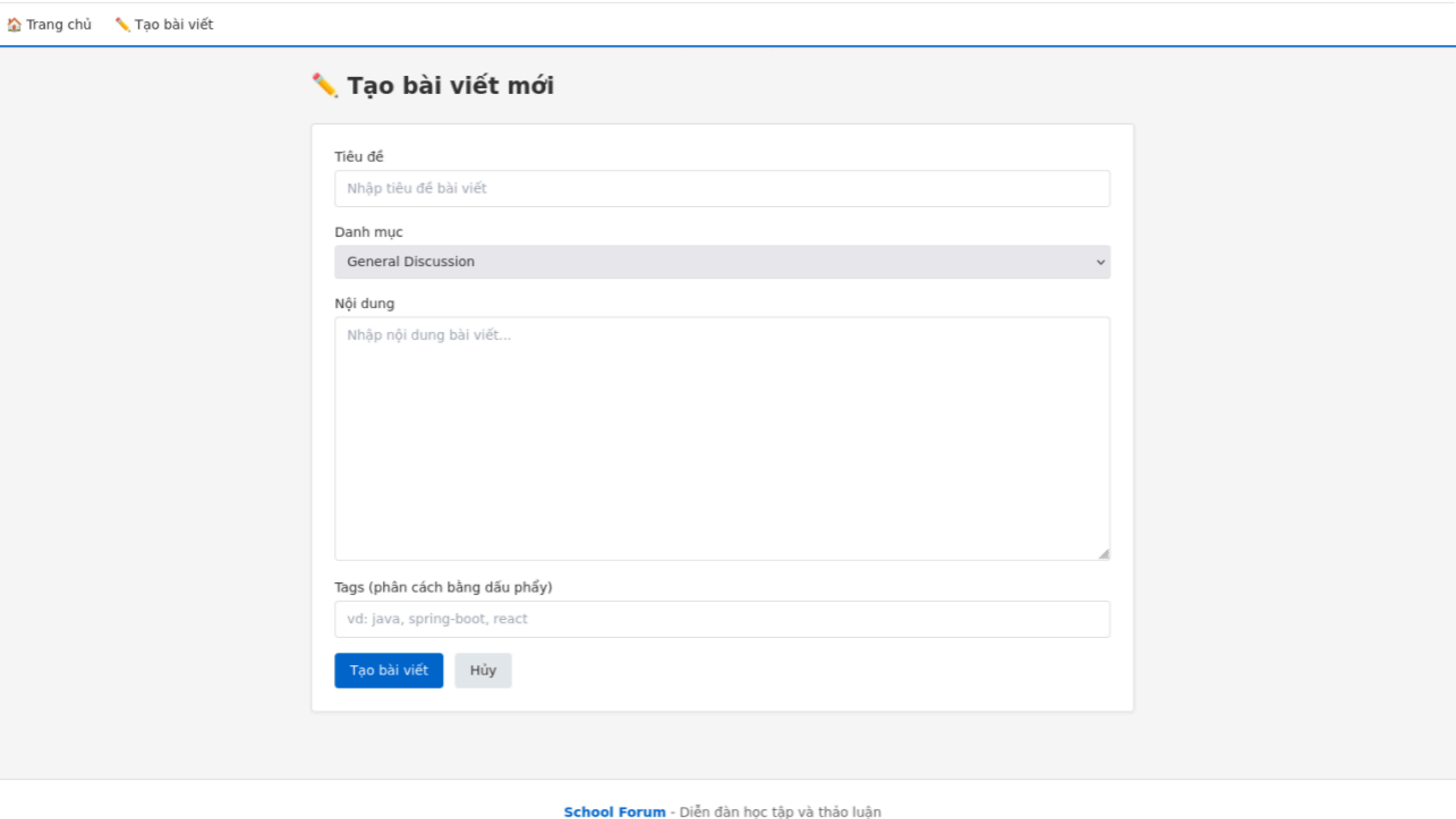
- Fields: Username, Email, Password
- Client-side validation
- Calls **authAPI.register()**

Header Component

- Shows user info when authenticated
- Logout button
- Conditional navigation links (e.g., Login/Register when logged out)

"Authentication system provides a seamless experience: login form sends credentials → receives JWT → stores token & user data → updates global state → UI updates dynamically."

THREAD DISPLAY & CREATION



🏠 Trang chủ ✏️ Tạo bài viết

✏️ Tạo bài viết mới

Tiêu đề

Nhập tiêu đề bài viết

Danh mục

General Discussion

Nội dung

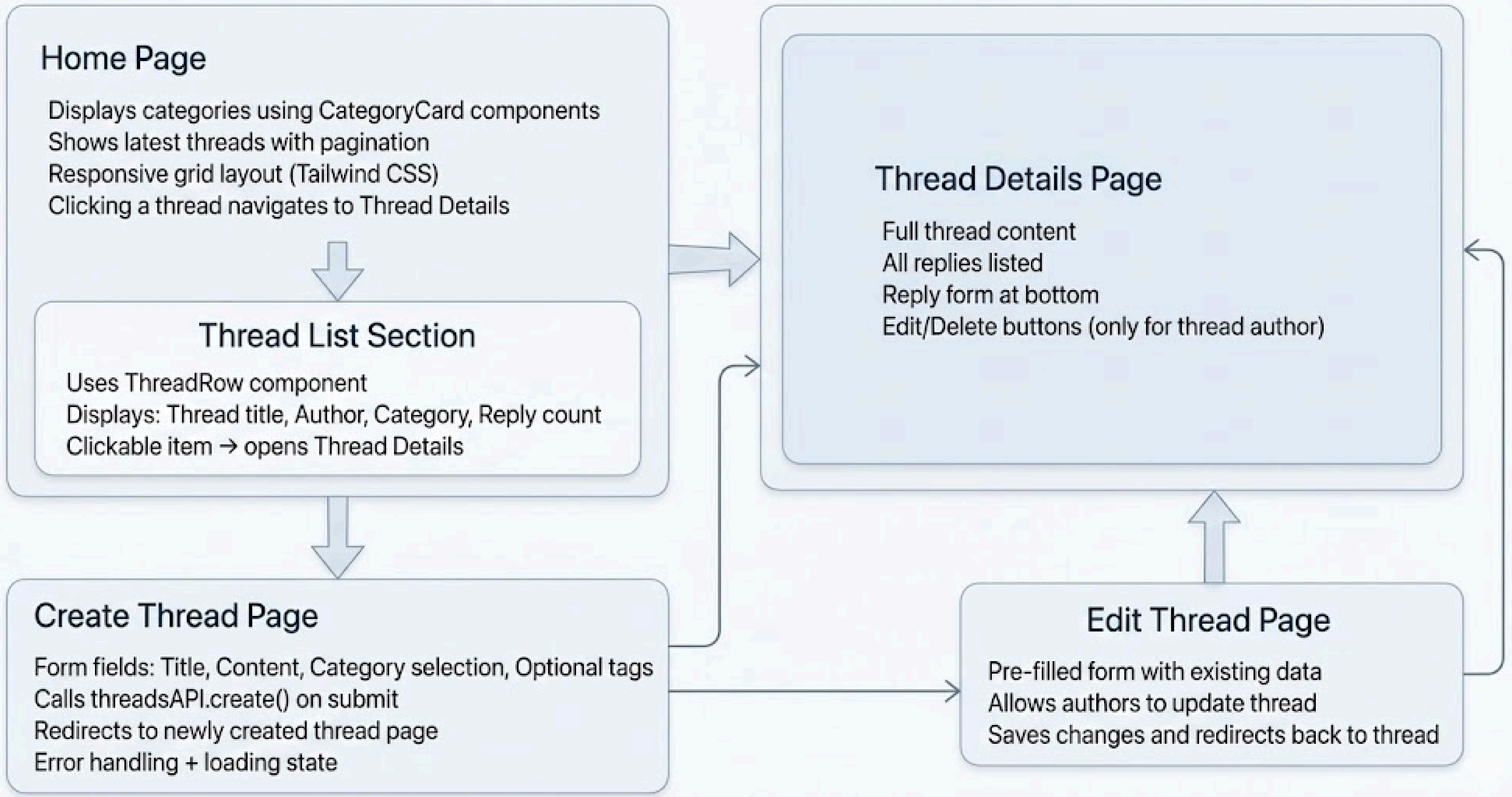
Nhập nội dung bài viết...

Tags (phân cách bằng dấu phẩy)

vd: java, spring-boot, react

Tạo bài viết Hủy

School Forum - Diễn đàn học tập và thảo luận


Home Page
Displays categories using CategoryCard components
Shows latest threads with pagination
Responsive grid layout (Tailwind CSS)
Clicking a thread navigates to Thread Details
Thread List Section
Uses ThreadRow component
Displays: Thread title, Author, Category, Reply count
Clickable item → opens Thread Details
Thread Details Page
Full thread content
All replies listed
Reply form at bottom
Edit/Delete buttons (only for thread author)
Create Thread Page
Form fields: Title, Content, Category selection, Optional tags
Calls threadsAPI.create() on submit
Redirects to newly created thread page
Error handling + loading state
Edit Thread Page
Pre-filled form with existing data
Allows authors to update thread
Saves changes and redirects back to thread

THANK
YOU!