Contents

I Introduction

A Background and Requirements

With the rising importance of collaborative learning, university library discussion rooms are highly sought after. Current campus room booking systems are limited, offering only basic functionalities like login, registration, and manual reservations without advanced features such as room filtering or detailed information. These shortcomings result in inefficient resource usage, lower user satisfaction, and inadequate administrative oversight. Thus, there is a need for a more flexible, transparent, and maintainable discussion room booking system.

B Aims of The Project

This project aims to develop a secure, scalable, and user-friendly online booking system for academic discussion rooms, our development team named it "***WeMeet***". Key features include email-based registration, room searching with dynamic filtering by capacity, equipment, and location, and comprehensive reservation management. Administrators can manage room data, monitor user activities, and control bookings. Additionally, the system offers logs and visual dashboards for real-time usage analysis, enhancing both user experience and administrative efficiency.

C User Characteristics

- **Regular Users**: Students, faculty, and staff who need an easy-to-use platform for reserving rooms. They require features like availability displays, filtering options, and access to booking history.
- **Administrators**: University staff managing room resources and overseeing bookings. They need advanced tools for room management, booking reviews, and resolving scheduling conflicts via a centralized dashboard.

D Assumptions and Dependencies

1 Assumptions

- **Stable Access**: Users have reliable internet access and use modern desktop browsers.
- **Institutional Email**: Registration and verification require a valid university-issued email address.
- **Administrator Maintenance**: Administrators keep room information current, including availability and equipment.
- **Policy Compliance**: Users adhere to the institution's room booking policies.

2 Dependencies

- **Email Service**: Reliable university email services are essential for account verification and password resets.
- **Containerization**: ***Docker*** ensures consistent deployment across environments.
- **Cloud Hosting**: ***Alibaba Cloud*** is used for production deployment.
- **Technology Stack**: Utilizes ***Spring Boot***, ***Thymeleaf***, and ***MySQL*** for backend and database development.
- **Development and Testing Tools**: ***GitHub*** for version control and collaboration, and ***Postman*** for API testing.

E Project Scope

The system encompasses user registration, secure login, email verification, profile management, room filtering and search, detailed room information, and full booking capabilities (create, edit, cancel). It supports two roles: regular users and administrators. Administrators can manage rooms, view and control bookings, and handle user accounts. The backend is built with ***Spring Boot***, the frontend with ***Thymeleaf***, and data is stored in ***MySQL***. Deployment uses ***Docker*** on ***Alibaba Cloud***. Currently, the system is optimized for desktop browsers and utilizes demonstration data. The specific user functionalities are illustrated in the use case diagram (see Figure 1 in Appendix A).

F Project Risks

- **Module Integration Challenges**: Integrating modules developed separately led to occasional system incompatibilities due to implementation variations and insufficient early coordination.
- **Deployment Configuration Issues**: Challenges with ***Docker*** and ***Alibaba Cloud*** deployment included environment inconsistencies and email service configuration failures.
- **Team Coordination Delays**: Division of work sometimes caused delays in aligning development progress, impacting task scheduling and milestone tracking.
- **Version Conflicts**: Working on separate branches without timely synchronization resulted in merge conflicts and inconsistent UI components, managed through structured version control and consistent code reviews on ***GitHub***.

- **Risk Mitigation**: Addressed through regular team meetings, clear responsibility assignments, sprint-based development, and collaborative use of ***GitHub*** for source control and issue tracking.

II Architectural Design

A System Architecture

Our system adopts a **Layered Architecture**, clearly separating responsibilities into Presentation, Business Logic, and Data Access Layers, enhancing maintainability and extensibility. The Presentation Layer internally utilizes the classic **MVC Pattern**, effectively isolating user interface, business logic, and data management. Additionally, the system employs **Feature-Based Modularization**, organizing code into independent modules based on business functionality, further improving maintainability. This modularization approach will be detailed in the **Software Design** section.

Moreover, the system follows a **Browser/Server (B/S) Architecture**, using ***AJAX*** for asynchronous frontend-backend communication, thus enhancing user experience and response efficiency. A **Multi-Level Caching Strategy** with ***Redis*** optimizes data access performance and reduces database load. This design ensures structural clarity, interface interactivity, and overall system performance.

Finally, the system is deployed on ***Alibaba Cloud ECS*** instances, with services containerized via ***Docker*** and managed by ***Docker Compose***. The database utilizes ***Alibaba Cloud RDS (MySQL)*** for reliability and performance. User-uploaded files (avatars, meeting photos, etc.) are stored in ***Alibaba Cloud OSS***. Additionally, ***Alibaba Cloud SLB*** distributes traffic, ensuring stable service operation under high concurrency.

The overall architecture is illustrated in Figure 2, with detailed descriptions provided below:

1 Layered Architecture

To ensure clear separation of responsibilities and facilitate development, maintenance, and scalability, we adopted the traditional **Layered Architecture Design**. Our system's layered architecture includes:

- **Presentation Layer**: Handles user interactions, displays data, and captures input. The frontend communicates with the backend via ***AJAX*** asynchronous requests, enhancing user experience by avoiding full page reloads.
- **Business Logic Layer**: Manages core operations like meeting room reservations, user management, and log auditing. Each functional module (e.g., user, meeting room) has a dedicated service layer, minimizing coupling between features.

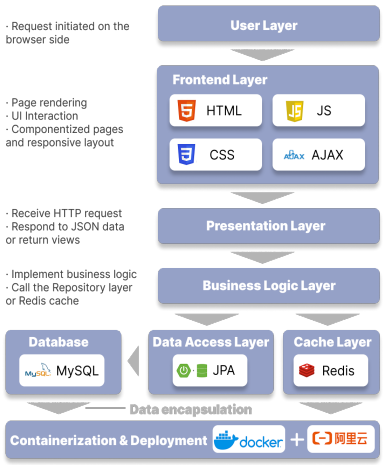


Figure 2: Architecture Design for WeMeet System

- **Data Access Layer**: Interfaces with the database using ***Spring Data JPA*** repositories to streamline data operations. It encapsulates ***MySQL*** CRUD actions, ensuring efficient data retrieval.

2 MVC Architecture

To clarify system logic, we implemented the **MVC Pattern** in the presentation layer, separating data representation, business logic, and user interactions:

- **Model**: Utilizes `dto` and `entity` classes (e.g., `BookingDTO`, `UserProfileUpdateRequest`) for data transmission and mapping, simplifying transfers and minimizing data exposure risks.
- **View**: Employs ***Thymeleaf*** as the template engine to dynamically render ***HTML*** templates, integrating ***CSS*** and ***JavaScript*** for the user interface. Dynamic data is populated via `model.addAttribute()` and ***AJAX*** enables partial updates without full page reloads.
- **Controller**: Handles user requests and coordinates with backend services, directing operations to the appropriate business logic layer. Dedicated controller classes manage different modules (e.g., reservation, user management) using annotations like `@GetMapping` and `@PostMapping` to handle various request types.

Figure 3 illustrates the MVC architecture pattern in the presentation layer, highlighting the separation of Model, View, and Controller components.

3 B/S Architecture

We adopted the **B/S Architecture** model, enabling the frontend to communicate directly with the server via a

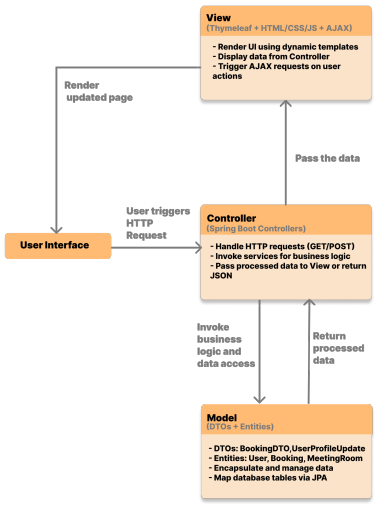


Figure 3: MVC Architecture for WeMeet System

browser. The reasons for choosing the **B/S Architecture** include:

- **No Client Installation**: Users access the system through a browser, reducing deployment and maintenance costs.
- **High Availability**: Supports access from various client devices.
- **Easy to Extend and Update**: Only server-side code requires updates, eliminating the need for client-side modifications.

4 AJAX Asynchronous Communication

We utilized ***AJAX*** to enable the frontend to communicate asynchronously with the backend without page reloads, enhancing user experience. ***AJAX*** is employed in our project as follows:

- **Form Submission**: For actions such as user login and reservation requests, the frontend sends **POST** requests via ***AJAX*** to the backend, which responds with ***JSON*** data. The frontend then updates the page based on the response.
- **Data Loading**: hen users check meeting room status or reservation records, data is fetched from the backend via ***AJAX***, avoiding full page reloads and enhancing interactivity.

5 Multi-Level Caching Strategy

We implemented a **Multi-level Caching Strategy** to optimize data access performance and reduce database load. The system integrates ***Redis*** caching within a multi-level approach, significantly improving data retrieval speed, especially under high concurrency. The strategy includes:

- **Session Management**: User login information and verification codes are cached using ***Redis***, minimizing frequent database queries and enhancing response times.
- **Hot Data Caching**: Frequently accessed data, such as meeting room availability and user reservation time slots, are cached in ***Redis***, reducing database load.
- **Global Caching Strategy**: Cache updates are managed through scheduled tasks or event triggers to ensure data timeliness and consistency.

6 Docker and Aliyun Deployment

We employed ***Docker*** for containerized deployment, allowing each service (including backend services, ***Redis*** caching, and ***MySQL*** database) to run independently within ***Docker Containers***. This facilitates flexible scaling of the system's services on ***Alibaba Cloud ECS***, enabling rapid deployment of new instances during high load periods to manage traffic fluctuations. ***Docker Compose*** is utilized for container orchestration, ensuring efficient and maintainable connections and dependency management between services.

B Frontend System

The frontend serves as the primary interface for users interacting with the ***WeMeet*** platform. It leverages ***HTML*** rendering and ***AJAX*** to deliver a seamless user experience. The key components include:

1 HTML + CSS: Structure and Styling

We utilize **HTML5** and **CSS3** to ensure compatibility and maintainability across devices. ***Bootstrap 5*** is employed for responsive layouts, with a main layout template `layout/main.html` ensuring a consistent appearance. Modular design separates the navigation bar, footer, and alerts into reusable components. Pages are categorized into functional areas such as authentication (`login.html`, `register.html`), administration (`admin/rooms/list.html`), and user interfaces (`user/portal.html`, `bookings/list.html`). Global styles in `style.css` and Font Awesome enhance visual consistency, while inline ***CSS*** addresses specific styling needs.

2 JavaScript: Interactive Logic

JavaScript powers dynamic interactions and rich user experiences through DOM manipulation and event handling. Core functionalities include form validation, dynamic UI management, date/time pickers, modal operations, and data visualization. Scripts are organized into functional modules (e.g., `js/auth/login.js`,

`js/auth/register.js`) to maintain feature isolation. The use of ***jQuery*** simplifies DOM operations and ***AJAX*** requests, while ***Bootstrap JS*** provides interactive components like modals and dropdowns.

3 AJAX: Asynchronous Backend Communication

We implement ***AJAX*** to facilitate asynchronous interactions with the backend, enhancing responsiveness and reducing load times. All data exchanges use ***JSON*** format, with the frontend utilizing the `fetch()` API to send HTTP requests. Key applications of ***AJAX*** include:

- **User Authentication**: Handling login, registration, and password recovery without page reloads.
- **Data Queries**: Fetching meeting room availability and booking statuses dynamically.
- **Form Submissions**: Creating and updating reservations seamlessly.

Additionally, ***AJAX*** supports real-time verification and error handling, providing immediate feedback and enhancing system reliability.

4 Thymeleaf: Server-Side Templating with Spring Boot

We selected ***Thymeleaf*** for its seamless integration with ***Spring Boot***, facilitating efficient frontend development while preserving ***HTML*** semantics. ***Thymeleaf*** enables reusable layouts and fragments, reducing redundant code through features like the main layout template (`layout/main.html`) and common sections (navigation, footer). It supports conditional rendering based on user roles, integrated with ***Spring Security*** using `sec:authorize`, ensuring secure and role-specific UI presentation. The natural templating concept allows developers to edit templates directly without additional tools or compilation steps.

C Backend System

The backend of our system is built with ***Java Spring Boot*** and adopts a **Layered Architecture** comprising *Controller*, *Service*, *Repository*, and *Entity* layers. A **Feature-Based Modularization** strategy organizes each functional domain, enhancing scalability, maintainability, and team efficiency.

1 Technology Stack and Justification

- **Spring Boot**: Streamlines application setup with embedded servers and auto-configuration.
- **Spring MVC**: Provides an extensible **MVC** framework for building ***RESTful APIs***.
- **Spring Security + JWT**: Enforces stateless authentication. Tokens are issued upon login and verified by a custom `JwtAuthenticationFilter`. Role-based authorization is supported (e.g., `ROLE_USER`, `ROLE_ADMIN`).
- **Spring Data JPA + Hibernate**: Facilitates ORM through repository interfaces, abstracting database interactions.
- **MySQL (Aliyun RDS)**: Offers reliable and consistent relational storage for user, room, and booking data.
- **Redis**: Implements multi-level caching with `Redis Template` for session management, verification codes, and hot data caching.
- **Thymeleaf**: Renders dynamic ***HTML*** pages using server-side templates.
- **Maven**: Manages project lifecycle and dependencies efficiently.

2 Component Structure

- **Controllers**: Organize ***RESTful APIs*** by feature, such as `AuthController`, `UserBookingApiController`, and `MeetingRoomApiController`, using annotations like `@GetMapping` and `@PostMapping` for request handling.
- **Services**: Contain core business logic, including user management, booking validation, and notifications, following an interface-implementation pattern.
- **Repositories**: Extend ***Spring Data JPA*** interfaces (e.g., `UserRepository`, `BookingRepository`) to support CRUD operations without explicit ***SQL***.
- **Entities**: Define domain models with ***JPA*** annotations (e.g., `@Entity`, `@Id`) for persistence, such as `User`, `Booking`, and `MeetingRoom`.
- **DTOs**: Encapsulate data transfer between layers with objects like `BookingDTO` and `LoginRequest`.
- **Security**: Incorporate components like `JwtAuthenticationFilter` and `CustomUserDetailsService` to manage authentication and authorization processes.
- **Utilities**: Provide helper classes (e.g., `JwtUtils`, `CaptchaUtils`) to support verification and token management.

D High-level Database Design

The ER diagram of our project database is presented in Figure 4.

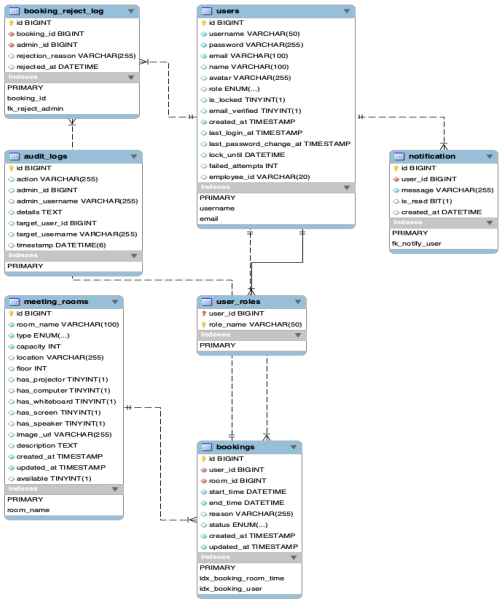


Figure 4: ER Diagram for Database of WeMeet System

1 Entity Perspective

Our system's database is structured around key entities that represent the core components of a university room reservation service. These entities include:

- `User`: Represents students and staff, managing their bookings and roles.
- `MeetingRoom`: Defines the attributes and availability of reservable rooms.
- `Booking`: Captures reservation details such as time, purpose, and status.
- `UserRole`: Facilitates role-based access control for flexible permissions.
- `Notification`: Manages system messages and updates to users.
- `BookingRejectLog`: Logs details of rejected bookings for accountability.
- `AuditLog`: Records administrative actions for security and compliance.

This modular design supports scalability, role-based access, and advanced features like conflict resolution and real-time availability.

2 Attribute Perspective

Each entity includes a set of attributes tailored to their functionalities:

- **User**: `id`, `username`, `password`, `email`, `name`, `role`, `is_locked`, and timestamps for account activity.
- **MeetingRoom**: `room_name`, `type`, `capacity`, `location`, `has_projector`, `has_screen`, etc.
- **Booking**: `start_time`, `end_time`, `status`, `created_at`, `updated_at`, `reason`.
- **Notification**: `message`, `is_read`, `created_at`.
- **AuditLog**: `action`, `details`, and relevant timestamps.
- **BookingRejectLog**: `id`, `booking_id`, `admin_id`, `rejection_reason`, `rejected_at`.

Primary keys are auto-incremented, with foreign keys ensuring referential integrity. Default timestamps and enumerated types support consistent data entry and reduce redundancy.

3 Relationship Perspective

The database schema ensures data integrity and coherence through clearly defined relationships:

- `Booking` has many-to-one relationships with both `User` and `MeetingRoom`, allowing multiple bookings per user and room.
- `UserRole` establishes a many-to-many relationship with `User`, enabling flexible role assignments.
- `BookingRejectLog` links rejected bookings to the respective `Booking` and the administrator responsible.
- `Notification` associates with `User` through a many-to-one relationship, allowing multiple notifications per user.
- `AuditLog` connects administrative actions to both the acting and affected users for comprehensive tracking.

Foreign keys are defined with appropriate `ON DELETE` and `ON UPDATE` constraints (e.g., `CASCADE`, `RESTRICT`) to maintain consistency. Indexes on key fields like `user_id`, `room_id`, and time intervals optimize performance for frequent queries and ensure efficient handling of high-concurrency scenarios.

This normalized, relational structure enhances modularity, minimizes redundancy, and ensures the system can efficiently manage complex queries and high user loads.

III Software Design

A Software Modules

The system employs a modularization strategy that enhances scalability, maintainability, and security by dividing the system into distinct functional modules. This approach allows for independent development, reduces interdependencies, and simplifies the integration of new features.

Our modularization strategy is **Feature-Based Modularization**, while the system is structured into seven primary modules, each responsible for specific features of the meeting room booking process:

- **User Management Module (UMM)**: Handles user registration, authentication, profile updates, and password management.
- **User Management Module (UMM)**: Handles user registration, authentication, profile updates, and password management.
- **Meeting Room Management Module (MRMM)**: Manages the creation, modification, deletion, and querying of meeting rooms.
- **Booking Management Module (BMM)**: Oversees the reservation, modification, cancellation, and history tracking of bookings.
- **Notification System Module (NSM)**: Ensures timely communication through email and in-app notifications regarding bookings and system alerts.
- **Audit Logging Module (ALM)**: Records system activities and administrative actions to maintain transparency and accountability.
- **Security Module (SM)**: Implements authentication and authorization mechanisms to protect against unauthorized access and threats.
- **Reporting and Statistics Module (RSM)**: Generates analytical reports on system usage, booking trends, and user activities.

Figure 5 depicts the interaction and data flow between the system's primary modules. User actions initiate processes within the **UMM**, which communicates with the **SM** for authentication and the **BMM** for handling reservations. The **BMM** verifies room availability through the **MRMM**, logs actions via the **ALM**, and sends notifications through the **NSM**. Administrators interact with the **ALM**, **MRMM**, and **RSM** to oversee operations and generate reports.

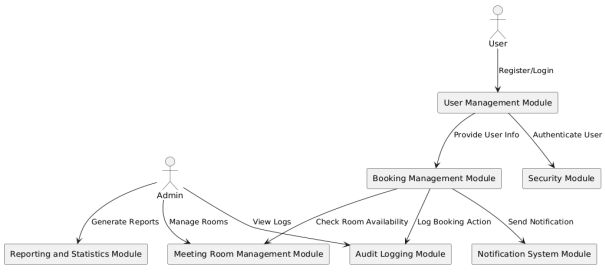


Figure 5: Software Modules Interaction Flow

An alternative approach is **Layered Modularization**, which organizes modules into distinct layers such as presentation, business logic, and data access. While this method provides clear separation and simplifies dependency management, it can restrict flexibility for adding new features.

Instead, the **Feature-Based Modularization** was chosen for our system. This strategy groups modules by specific functionalities, enhancing cohesion and decoupling. It facilitates easier maintenance and scalability, supports parallel development, and allows targeted security measures within individual modules. This approach better aligns with the project's scope and resource constraints, ensuring a robust and adaptable system.

B High-level Process Flow

Based on the software modules defined, our system's functionalities are divided into **user-facing** and **administrator-facing** categories, ensuring efficient workflows for both end-users and administrators in managing meeting room reservations.

1 User Functionality

- **User Registration and Login**: Users access the system by creating accounts and authenticating securely.

  New users provide usernames, passwords, and email addresses. The **UMM** verifies password strength and email format, while the **SM** generates email verification codes. These codes are sent via the **NSM** and stored in **SM**'s Redis cache with a 15-minute expiration to prevent misuse. Upon successful verification, user information is stored in the database.

  Registered users log in with their credentials. The **UMM** validates the credentials, and the **SM** tracks failed attempts, locking accounts for 5 minutes after 5 unsuccessful tries to deter brute-force attacks. On successful login, the **SM** generates a JWT token for session management, which is stored in Redis.

  Key mechanisms include BCrypt password encryption, verification code validation, login failure limits, and JWT-based session management, all managed by the **UMM**, **SM**, and **NSM**. The sequence diagram (see Figure 6 in Appendix A) illustrates this workflow.

- **Personal Profile Management**: Users can manage their personal information securely.

  For password resets, users provide their registered email. The **UMM** validates the email format, and the **SM** generates a verification code sent via the **NSM**, stored in Redis for 15 minutes. Users submit the code and a new password, which the **SM** validates before updating the encrypted password in the database.

  Users can also update personal details like avatars and passwords through the Profile page. The **UMM**

validates inputs, and the **SM** ensures security compliance. Successful updates are saved in the database, and users receive confirmation.

Key mechanisms include BCrypt encryption, verification code validation, and input validation, supported by the **UMM**, **SM**, and **NSM**. The sequence diagram (see Figure 7 in Appendix A) illustrates this process.

- **Room Search and Reservation**: Users can search for and manage meeting room bookings.

  Users apply filters such as floor, capacity, and equipment to search for rooms. The **MRMM** retrieves matching rooms and available time slots from the database, displaying the results. For reservations, users select a room, date, and time slot. The **BMM** checks availability, saves the booking if the room is free, and the **NSM** confirms the reservation.

  Users can modify or cancel reservations. The **BMM** ensures no conflicts before updating or deleting bookings in the database. Successful changes trigger confirmation messages via the **NSM**, keeping users informed.

  Key mechanisms include real-time availability checks, conflict resolution, and user notifications, managed by the **MRMM**, **BMM**, and **NSM**. The sequence diagram (see Figure 8 in Appendix A) illustrates these operations.

2 Administrator Functionality

- **Meeting Room Management**: Administrators manage meeting room resources and operations.

  Administrators can review all meeting rooms, accessing details such as room names, floors, capacities, and equipment. The **MRMM** retrieves this data from the database for accurate decision-making. To add a room, administrators provide details like room name, floor, and capacity. The **MRMM** validates uniqueness and stores the information upon confirmation.

  For deletions, administrators specify the room ID. The **MRMM** verifies existence and works with the **BMM** to handle associated bookings. If no bookings exist, the room is deleted directly. If bookings are present, the **BMM** removes them, and the **NSM** notifies affected users about cancellations, allowing them to make alternative arrangements. Finally, the **MRMM** deletes the room record, ensuring all related data is properly handled.

  Key mechanisms include input validation, booking checks, and user notifications, supported by the **MRMM**, **BMM**, and **NSM**. The sequence diagram (see Figure 9 in Appendix A) illustrates this workflow.

- **User Account Management**: Administrators oversee user accounts to prevent resource misuse.

  Administrators review booking records to identify abnormal behaviors like excessive bookings. The **BMM** provides comprehensive booking data for analysis. Upon detecting misuse, administrators can lock a user's account via the **UMM**, which updates the account status to "locked" and triggers a notification through the **NSM**.

  To reinstate access, administrators manually unlock the account, with the **UMM** updating the status to "active" and the **NSM** informing the user. This ensures fair resource allocation and system integrity.

  Key mechanisms include booking data analysis, account status management, and user notifications, facilitated by the **BMM**, **UMM**, and **NSM**. The sequence diagram (see Figure 10 in Appendix A) illustrates this process.

- **Booking Management**: Administrators oversee and manage meeting room bookings efficiently.

  Administrators can view all bookings, accessing details such as User IDs, Room IDs, booking times, and statuses. The **BMM** retrieves this data from the database for accuracy. Administrators can cancel bookings by specifying the booking ID. The **BMM** verifies the booking's existence and collaborates with the **NSM** to notify affected users about the cancellation. Once notified, the booking record is deleted from the database.

  Key mechanisms include booking record retrieval, validation, and user notification, supported by the **BMM** and **NSM**. The sequence diagram (see Figure 11 in Appendix A) illustrates this workflow.

Additionally, the **ALM** records system activities to maintain transparency, and the **RSM** generates statistical reports to provide data support, implicitly participating in all functionalities.

C Software Support Services

1 Database Services

- **Relational Database Integration**: We use ***MySQL 8.0*** to store user information, meeting room details, and booking records. Configuration files specify connection parameters, driver classes, and ***SQL*** dialect settings. ***UTF-8 encoding*** ensures multilingual support and scalability. ***Spring Data JPA*** provides an ORM layer with annotated entity classes (e.g., `User`, `MeetingRoom`) and repository interfaces (e.g., `UserRepository`), enabling complex queries without explicit ***SQL***.

- **Connection Pool Management**: ***HikariCP*** manages the database connection pool, configured for optimal performance through connection timeout and pool size settings. ***Spring Boot***'s auto-configuration ensures efficient resource utilization under high concurrency, enhancing performance during heavy loads.
- **Caching Infrastructure**: ***Redis*** is employed as a distributed cache for storing captchas and temporary data with expiration times. In case ***Redis*** is unavailable, the system defaults to an in-memory `ConcurrentHashMap`, ensuring continued operation and data consistency across deployments.

2 Security Services

- **Authentication and Authorization Framework**: ***Spring Security*** manages authentication and authorization. Security configurations define access policies, restricting administrative pages to users with `ROLE_Admin`. Passwords are hashed using ***BCryptPasswordEncoder***, and account lockout mechanisms prevent brute-force attacks. Email verification ensures unique usernames and email addresses during registration.
- **JWT-Based Authentication**: We implement stateless authentication using ***JSON Web Tokens (JWT)***. A utility class handles token generation and validation, while a custom JWT filter processes tokens from HTTP headers or cookies, establishing the security context. This approach enhances scalability and fault tolerance by eliminating server-side session storage.
- **Security Auditing and Protection Mechanisms**: Audit logs track critical user actions for security monitoring. ***Thymeleaf***'s automatic ***HTML*** escaping prevents cross-site scripting (**XSS**) attacks, and ***Bean Validation*** enforces input constraints. File operations, such as avatar uploads, include validations for file type and size. Method-level authorization is enforced using `@PreAuthorize` annotations to ensure secure access control.

3 File Storage Services

Aliyun Object Storage Service (OSS) is utilized for storing user avatars. The `FileService` interface abstracts file operations, saving files in the `avatars/` directory with ***UUID-based*** filenames. Validations for file type and size are performed before uploads, and old files are deleted upon avatar updates to manage the file lifecycle effectively.

4 Notification and Messaging Services

- **Email Notification Service**: ***Spring Mail*** sends verification codes and notifications using configured ***JavaMailSender*** beans and ***SMTP*** settings. Emails are formatted in HTML for better readability, and ***Redis*** handles captcha management to ensure quick response times and data consistency.
- **Internal Notification System**: An internal system notifies users of booking status changes, storing user ID, message content, read status, and timestamps. Notifications are styled with distinct ***CSS*** classes to differentiate between read and unread messages, enhancing user awareness.

5 Development Tools and Services

- **Version Control**: ***Git*** is used for version control, supporting collaboration through a branch management strategy and standardized commit messages. This facilitates parallel development and smooth integration via pull requests.
- **Build and Dependency Management**: ***Maven*** standardizes the build process and manages dependencies through a shared `pom.xml`, ensuring consistency and ease of onboarding for new developers.
- **Containerization Platform**: ***Docker*** enables consistent development and deployment environments defined by `Dockerfiles` and `docker-compose.yml`. This approach accelerates environment setup and ensures reproducibility across different machines.
- **Developer Productivity Tools**: ***Lombok*** reduces boilerplate code with annotations, while ***Spring Boot DevTools*** provides hot reloading for automatic application restarts. These tools enhance development efficiency and streamline the testing cycle.

D Coding Structure and Convention

1 Coding Structure

The system utilizes the traditional ***Spring Boot*** and ***Thymeleaf Model-View-Controller (MVC)*** architecture, promoting clear separation of concerns into data management, user interface, and control logic. This enhances modularity, maintainability, and scalability. The code structure is shown in Figure 12.

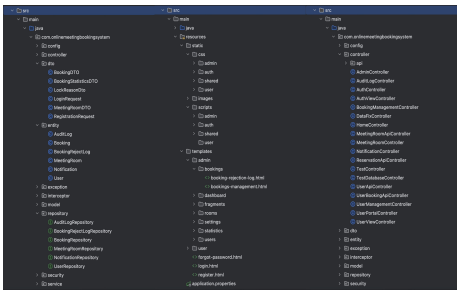


Figure 12: Code Structure of WeMeet Project

- **Model**: Contains packages with Java classes such as `entity` (e.g., `Booking`, `MeetingRoom`, `User`) aligned with database entries. The `dto` package (Data Transfer Object) secures data and ensures compatibility with the frontend. The `model` package handles non-database objects and encapsulates service data. The `repository` package includes interfaces for database manipulation.

- **View**: Frontend files (***CSS***, ***JavaScript***) reside in `resources/static`, while ***HTML*** files are in `resources/templates`. These files create user interfaces and functionalities like booking time slots, rejecting invalid bookings, and managing dashboards for users and admins. ***AJAX*** enhances user interactions and smooth designs. Two separate directories clearly distinguish admin and user-oriented ***HTML*** files, while shared files (`login.html`, `register.html`) provide common interfaces.

- **Controller**: Includes classes like `UserBookingApi Controller` and `UserBookingController`. The `API controller` is a ***RESTful*** controller returning ***JSON*** data for frontend ***JavaScript*** queries. The view controllers navigate and render specific ***HTML*** pages. Controllers manage requests such as booking meetings or rejecting reservations. User functions are handled by controllers like `UserBookingController` and `UserViewController`, while `AdminController` and `AuthViewController` serve administrators, ensuring clear separation of user and admin functionalities.

2 Coding Convention

In our project, we apply consolidated coding style for readability and maintainability. The coding scheme is illustrated in Figure 13 and Figure 14.

Figure 13: Coding Scheme of WeMeet Project (a)

- **Code Layout**: We use four spaces for indentations, improving the clarity of the code structure for the overall project.

Figure 14: Coding Scheme of WeMeet Project (b)

- **Identifier Naming**: We use ***camelCase*** for variables and functions, and ***PascalCase*** for classes to make the code readable and consistent.

- **OOP Principle**: We use object-oriented-principle to create every class to enhance the maintainability of the codes.

- Comments: We use block comments to give overall descriptions for modules and functions, while the inline comments illustrate some sophisticated code segments. Using different types of comments could guarantee group members to understand quickly when cooperating together.

- **Error Handling**: We use ***try-catch*** blocks within some codes where the errors would appear potentially to enhance the stability and resilience.

E Software Configuration and Production Environment

1 Software Configuration

The software configuration for ***WeMeet*** web application is designed after we reach a consensus on how to build an efficient system.

- **Front-end**: The interface applies ***HTML***, ***CSS***, and ***JavaScript*** to create user-oriented and responsive web pages. The interfaces are also showing different contents and realizing distinct functions for users and admins. Aligned with ***AJAX***, ***JavaScript*** is utilized to promote the dynamic updates of pages.

- **Back-end**: The whole system is a ***Spring Boot*** based project, along with the ***MVC Pattern*** to deliver a clear and maintainable environment.

- **Database**: ***MySQL*** is served as a database management system. The tables and attributes are tailored carefully to ensure it could accommodate diverse functionalities within the system like booking a room, rejecting a booking, and downloading logs.

Redis is served as a key-value database to guarantee the mail verification function could be operated safely and efficiently. Both of the two databases could ensure the data integrity and quick access.

2 Production Environment

The production environment is constructed on ***JDK 17***, ***MySQL 8.0.41***, ***Redis 7.0.15***, and ***Spring Boot Version 3.4.4***.

Our project is deployed on ***Alibaba Cloud*** server to ensure the accessibility and responsiveness. The configuration of the server is shown in Table 1.

Table 1: Server Configuration

Operating System	Ubuntu 22.04 (64-bit)
CPU & Memory	4 core (vCPU) & 8 GiB
Public Network Bandwidth	5 Mbps (Max)

To make the deployment safer and faster, we use ***Docker*** to build an isolate application mirror. Moreover, ***Docker*** could make the application consistent and prevent complex configurations. We installed and utilized ***Docker*** to deploy our application on server. The detailed ***Docker Container*** is illustrated in Figure 15.

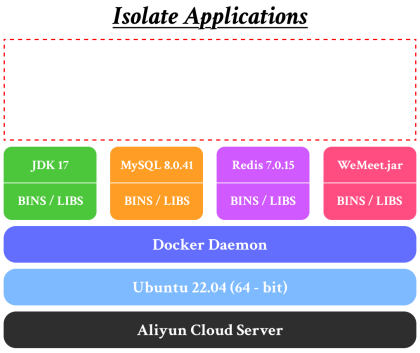


Figure 15: Docker Container of WeMeet Project

IV Software Testing

A Unit Testing

Unit testing is crucial in our strategy to ensure individual components, especially service layer classes, function correctly in isolation. We employed ***JUnit 5*** and ***Mockito*** to create mock objects, isolating dependencies like database mappers and other services. This enables verification of business logic within each unit without relying on external systems.

Our unit tests follow a clear structure: setting up test conditions with input data and mocks, executing the method under test, and verifying outcomes by checking return values and mock interactions against expectations.

We have an extensive suite of unit tests covering critical functionalities. For instance, in the user management module—a core system service—we developed over **81 unit tests** using ***JUnit*** and ***Mockito***. Below are examples of our approach to testing administrator functions for locking and unlocking user accounts.

1 Successful User Lock Test

This test case verifies the `lockUser()` method in `AdminServiceImpl`. It ensures that when an administrator locks an existing, currently unlocked user, the user's status is updated correctly, and the necessary persistence method is invoked exactly once with the correct parameters. ***Mockito*** is used to mock the `UserMapper` dependency.

```
@Test
void testLockUser_Success() {
    // Arrange
    String username = "testuser";
    User user = new User();
    user.setUsername(username);
    user.setAccountNonLocked(true); // Initially
    unlocked

    // Mock dependencies using Mockito
    when(userMapper.findByUsername(username)).
    thenReturn(user);
    when(userMapper.updateUserLockStatus(username
    , false)).thenReturn(1);

    // Act: Call the method under test
    adminService.lockUser(username);

    // Assert: Verify outcomes and interactions
    assertFalse(user.isAccountNonLocked()); //
    Check user object state
    verify(userMapper, times(1)).findByUsername(
    username);
    verify(userMapper, times(1)).
    updateUserLockStatus(username, false);
}
```

Listing 1: Unit Test for Successful User Account Lock

2 Successful User Unlock Test

Similarly, this test case validates the `unlockUser()` method. It checks that when an administrator unlocks an existing, currently locked user, the user's status is updated to `true` (account non-locked), and the `userMapper.updateUserLockStatus()` method is called correctly.

```
@Test
void testUnlockUser_Success() {
    // Arrange
    String username = "testuser";
    User user = new User();
    user.setUsername(username);
    user.setAccountNonLocked(false); // Initially
     locked

    // Mock dependencies
```

```
    when(userMapper.findByUsername(username)).
    thenReturn(user);
    when(userMapper.updateUserLockStatus(username
    , true)).thenReturn(1);

    // Act
    adminService.unlockUser(username);

    // Assert
    assertTrue(user.isAccountNonLocked()); //
    Check user object state
    verify(userMapper, times(1)).findByUsername(
    username);
    verify(userMapper, times(1)).
    updateUserLockStatus(username, true);
}
```

Listing 2: Unit Test for Successful User Account Unlock

These examples illustrate our methodology for ensuring the correctness of individual service methods. We conducted comprehensive unit tests across various modules to enhance code reliability.

Using ***JUnit 5*** and ***Mockito***, we developed and executed **542 unit test cases**, covering **8 major functional areas** of the system, including foundational services like user management. This extensive testing scope provides strong confidence in the functional correctness of individual components and significantly improves the overall robustness and reliability of the meeting room booking system.

B Integration Testing

Integration testing focused on verifying interactions between components, ensuring **API** endpoints behaved correctly during client-server communication. Our strategy was guided by **Product Backlog Items** (PBIs) from the requirements phase, providing a framework for defining key integration points across the system's eight major functional areas.

We used ***Postman*** for ***API-level*** integration testing. Requests were manually crafted with the method, URL, headers (including authentication), and body. These requests were sent to backend endpoints, and responses were verified by checking HTTP status codes and response content against expected results. Authenticated tests included credentials from prior login requests.

Examples below demonstrate tests on core authentication and data retrieval endpoints.

1 User Authentication Test

This test case verifies the basic login functionality for a standard user via the designated `/login` API endpoint using form data.

A **POST** request containing valid user credentials was sent to the `/login` endpoint. The system returned the expected `200 OK` status and success message, confirming successful authentication, as shown in Figure 16 in Appendix B.

2 Administrator Authentication Test

This test validates the authentication process for administrator accounts via the `/login` endpoint.

A **POST** request using valid administrator credentials was sent. Figure 17 in Appendix B shows the successful `200 OK` response, confirming admin login and allowing subsequent admin actions.

3 Admin Data Retrieval - User List

This test checks if an authenticated administrator can retrieve the user list via the `/admin/users` endpoint.

Using the admin's authentication cookie obtained from login, a **GET** request was sent. The necessary ***Cookie*** was included in the headers. Figure 18 in Appendix B demonstrates the successful `200 OK` response containing the expected ***JSON*** array of user data, verifying the endpoint and authorization.

These selected tests demonstrate the verification of crucial integration points related to user authentication, role-based access, and basic data retrieval, forming part of our broader integration testing effort guided by the project's **PBIs** across its main functional modules. Similar Postman tests were conducted for other key **APIs**, including those related to meeting room booking and management, ensuring components interact correctly across the system.

C Acceptance Testing

Acceptance testing is the final validation phase, assessing the complete system from the end-user's perspective to ensure it meets business requirements and user expectations outlined in project specifications and **PBIs** or **User Stories**. This ensures the system delivers intended functionality and value to both users and administrators.

1 Testing Process

We design test cases based on requirements and PBIs, simulating realistic user scenarios with defined steps, input data, and expected outcomes from acceptance criteria. Team members manually execute these tests via the application's UI in a test environment resembling production. Results are compared to expected outcomes to determine pass or fail, with defects documented and tracked through resolution and re-testing.

2 Scope of Testing

Acceptance testing covers all major functional and key non-functional aspects defined in the requirements, including:

- **User Lifecycle**: Registration with email verification, role-based login/logout, and profile updates.
- **Room Discovery**: Listing, viewing details, searching, and filtering rooms.

- **User Booking Management**: Creating, viewing, canceling, and updating bookings.
- **Administrator Capabilities**: Managing rooms and users (including locking/unlocking accounts) and handling all bookings.
- **System Feedback and Usability**: Verifying notifications, navigation, and user feedback.
- **Security Basics**: Ensuring role-based access control.

3 Testing Result Example

Using the user registration scenario as an example, the **PBI** for **User Registration** outlined key requirements and acceptance criteria:

(1) Users must provide account name, avatar, email, password, and confirm password.

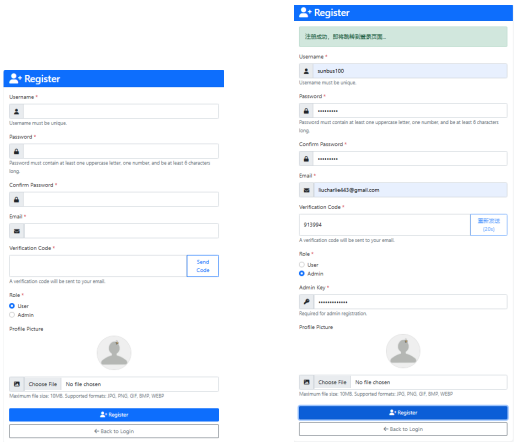
(2) The page should have clear field labels and placeholders.

(3) Each field must meet specific format requirements.

(4) Feedback messages should appear for empty or incorrectly formatted fields.

(5) A "Registration" button must submit the form and trigger validation.

As shown in Figure 19(a), 19(b), all acceptance criteria are fully met.



(a) Registration (b) Validation

Figure 19: User Registration and Validation Interface

V Conclusion and Future Work

A Achievement

Our team successfully developed and deployed a modular, scalable, and user-centric online meeting booking system tailored for academic environments. The platform allows students, faculty, and staff to register, search for available rooms, and manage bookings based on criteria such as capacity, location, and IT equipment.

Utilizing the ***MVC*** architecture with ***Spring Boot***, ***Thymeleaf***, and ***MySQL***, we achieved a clear separation of concerns, enhancing maintainability. The system includes administrative features like user management, real-time logs, and visual dashboards for tracking reservations. Comprehensive unit and integration testing ensured the system's reliability.

Adopting a sprint-based workflow with weekly meetings improved coordination and efficiency, transforming a basic backend prototype into a cohesive and user-friendly application. Initial user testing provided valuable feedback, leading to improvements in UI design and booking flow logic.

B Limitations

- **Mobile Compatibility**: Currently, the system is optimized for desktop browsers and lacks responsiveness for mobile devices, limiting accessibility for mobile users.
- **Real Data Integration**: The platform operates with demonstration data and has not yet integrated real-time institutional room information.
- **Accessibility Features**: The absence of support for assistive technologies, such as screen readers, restricts usability for users with disabilities.
- **Deployment Complexity**: Configuring ***Docker*** containers and setting up production-level email verification proved challenging during deployment.

C Future Work

- **Mobile and Accessibility Enhancements**: Future developments will focus on responsive design for mobile access and the incorporation of accessibility features to accommodate a broader user base.
- **Data Integration and Analytics**: We plan to integrate real-time university data and implement analytical tools to help administrators monitor usage trends and optimize room allocation.
- **Feature Expansion**: Additional functionalities such as calendar synchronization, notification systems, and automated conflict detection will be introduced to enhance the user experience.
- **Formal User Testing**: Structured usability testing with target users will be conducted to gather insights for further refining the interface and workflow.

Appendix

Appendix A: UML Diagrams

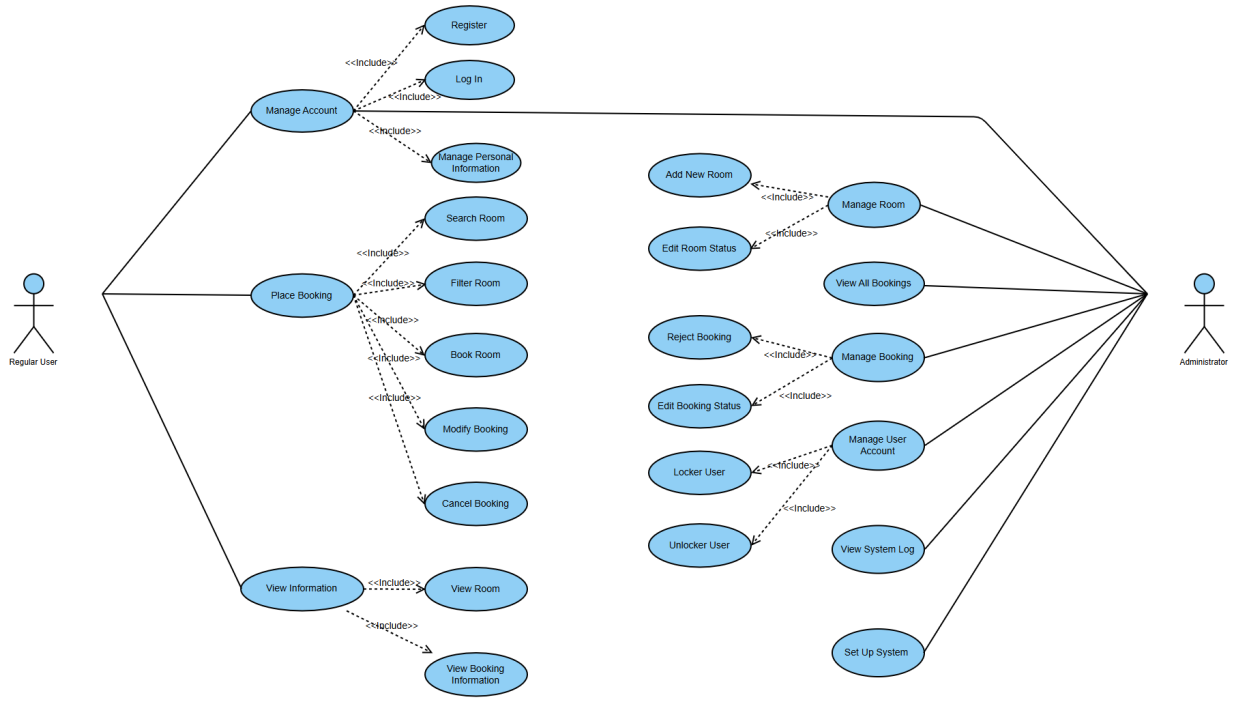


Figure 1: Use Case Diagram for WeMeet System

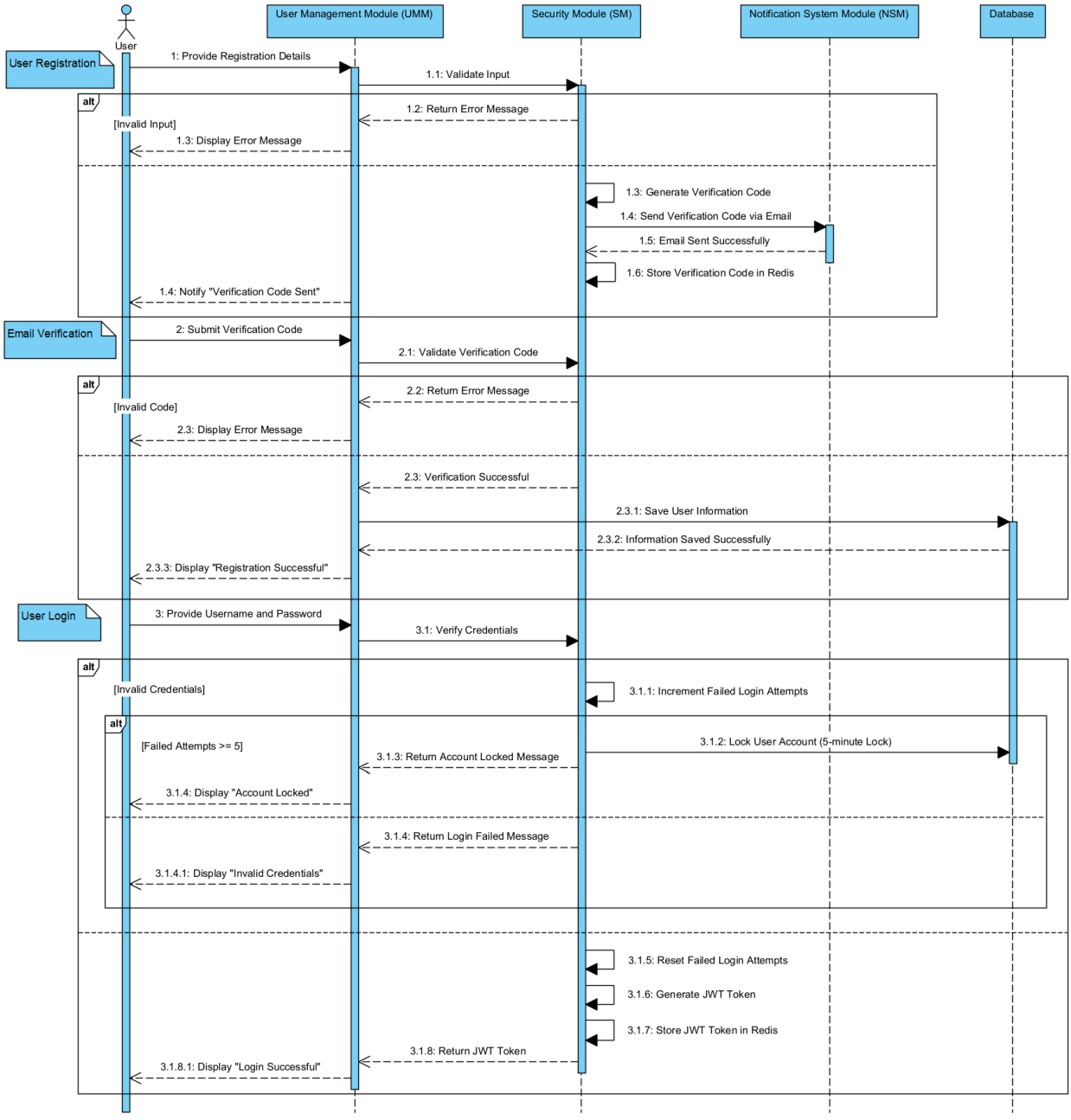


Figure 6: Sequence Diagram for User Registration and Login

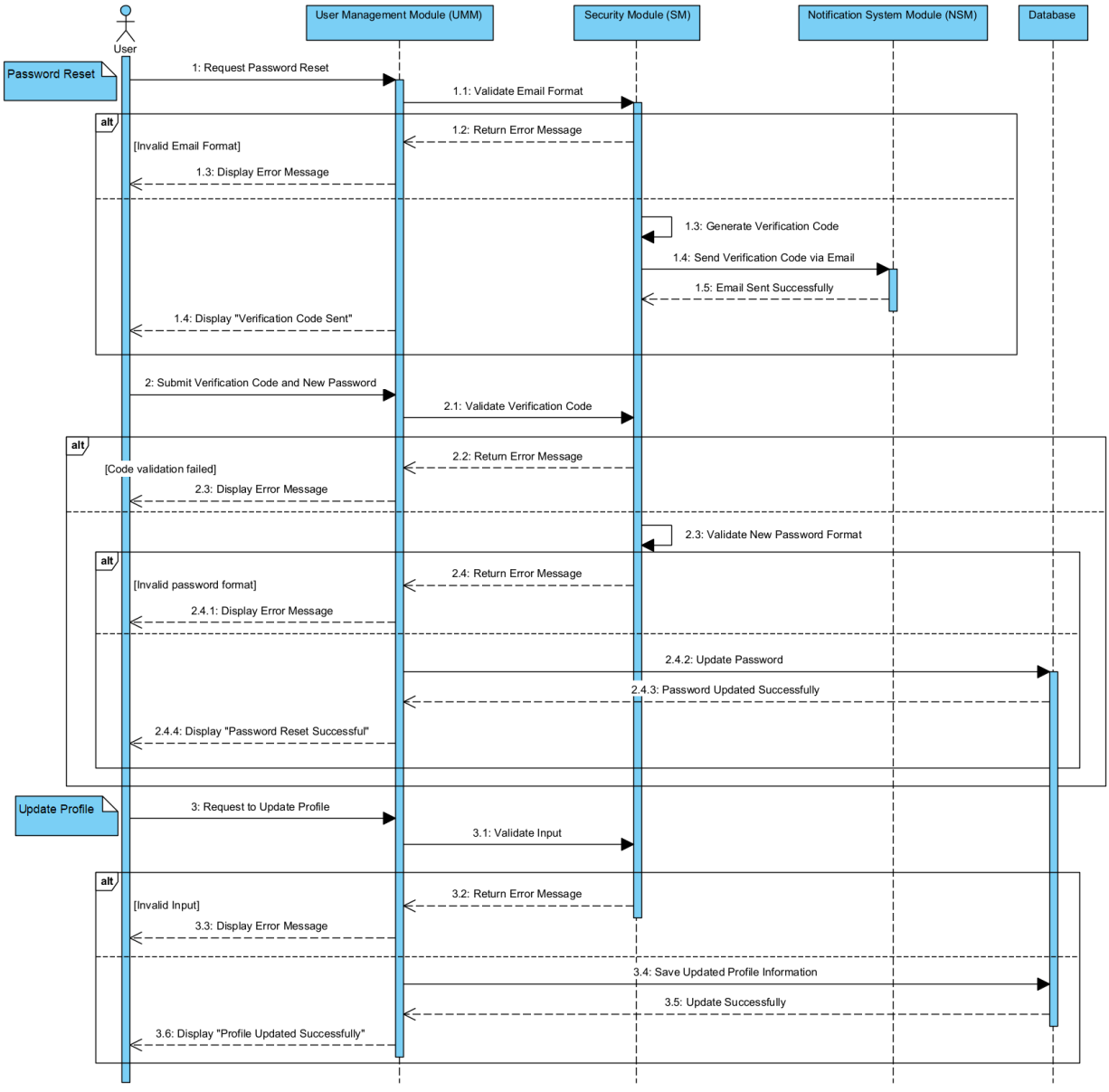


Figure 7: Sequence Diagram for Personal Profile Management

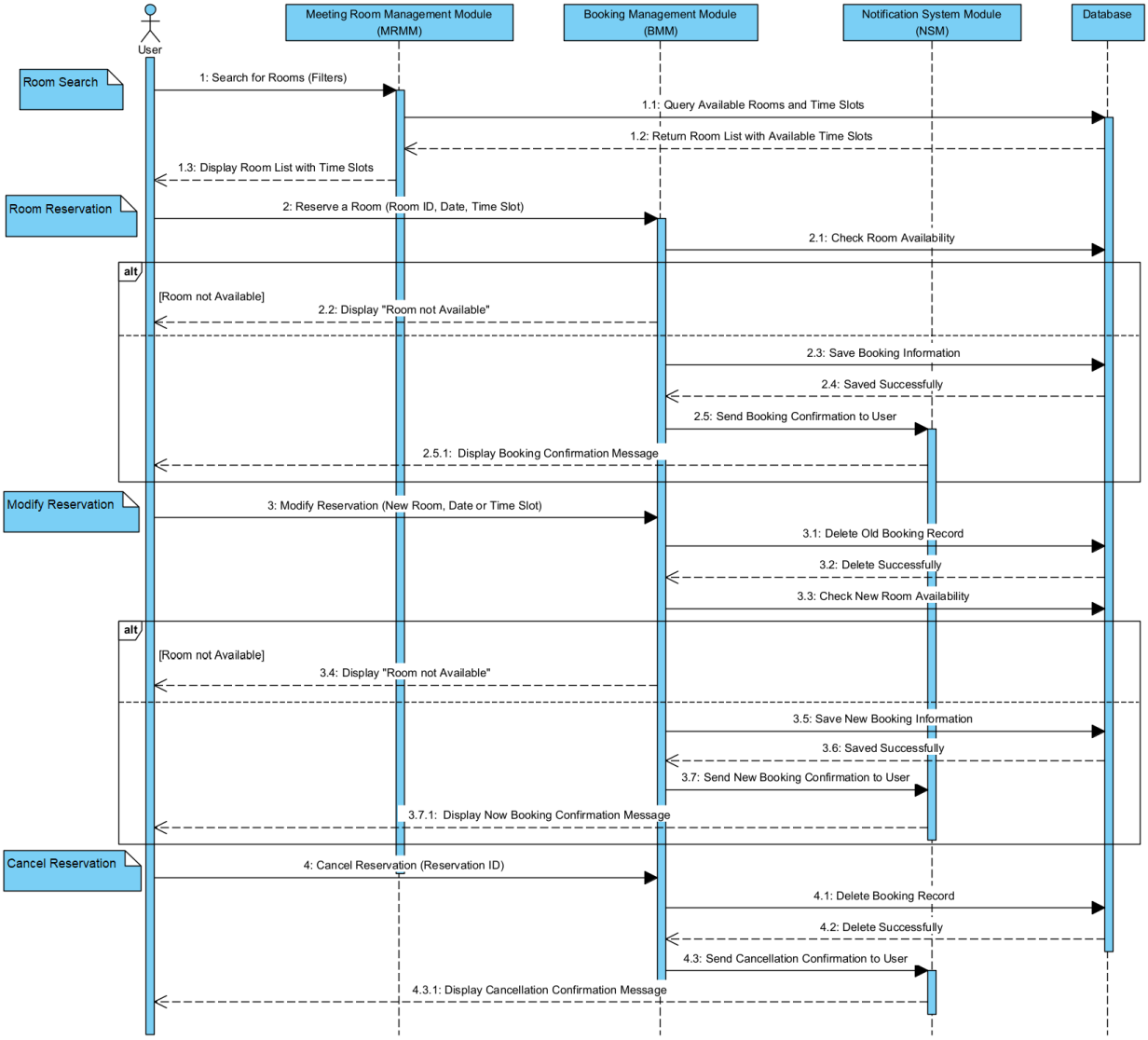


Figure 8: Sequence Diagram for Room Search and Reservation

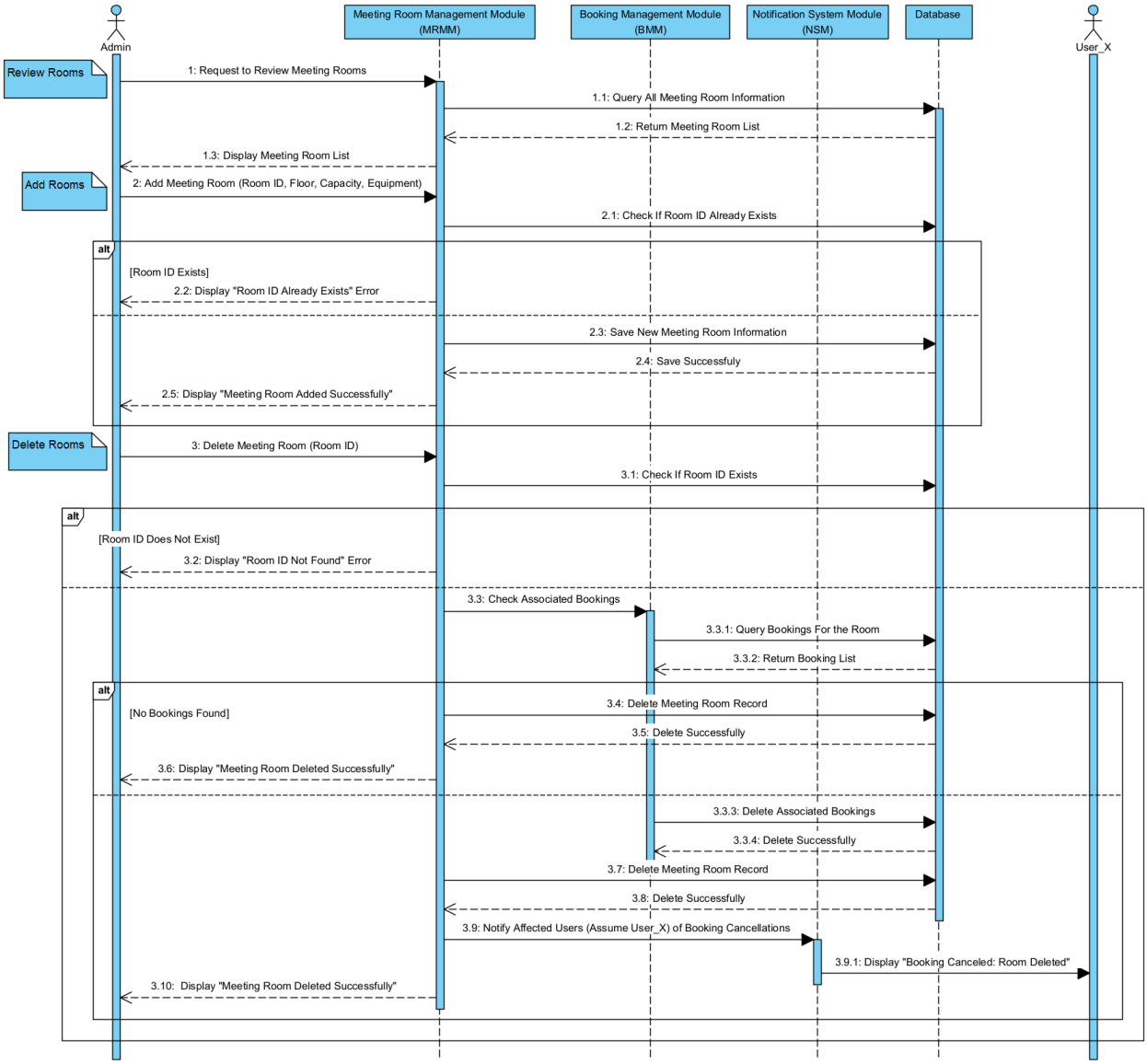


Figure 9: Sequence Diagram for Meeting Room Management

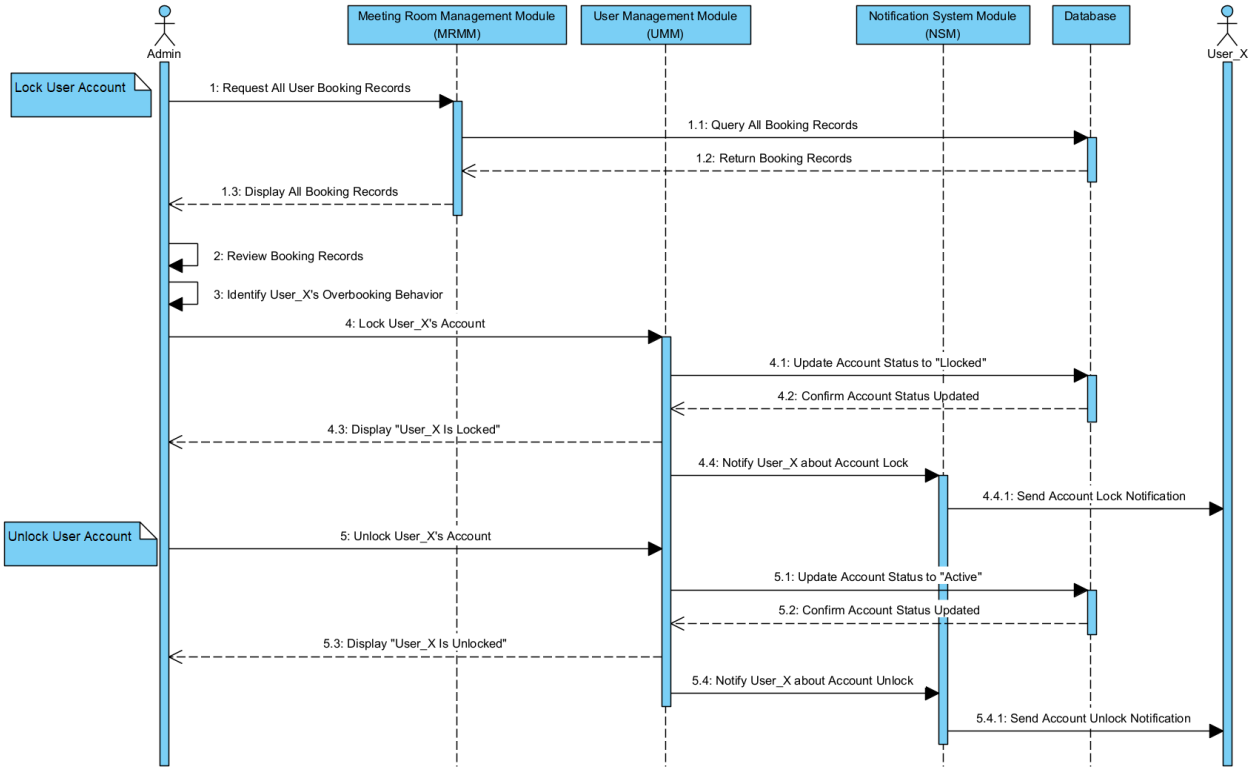


Figure 10: Sequence Diagram for User Account Management

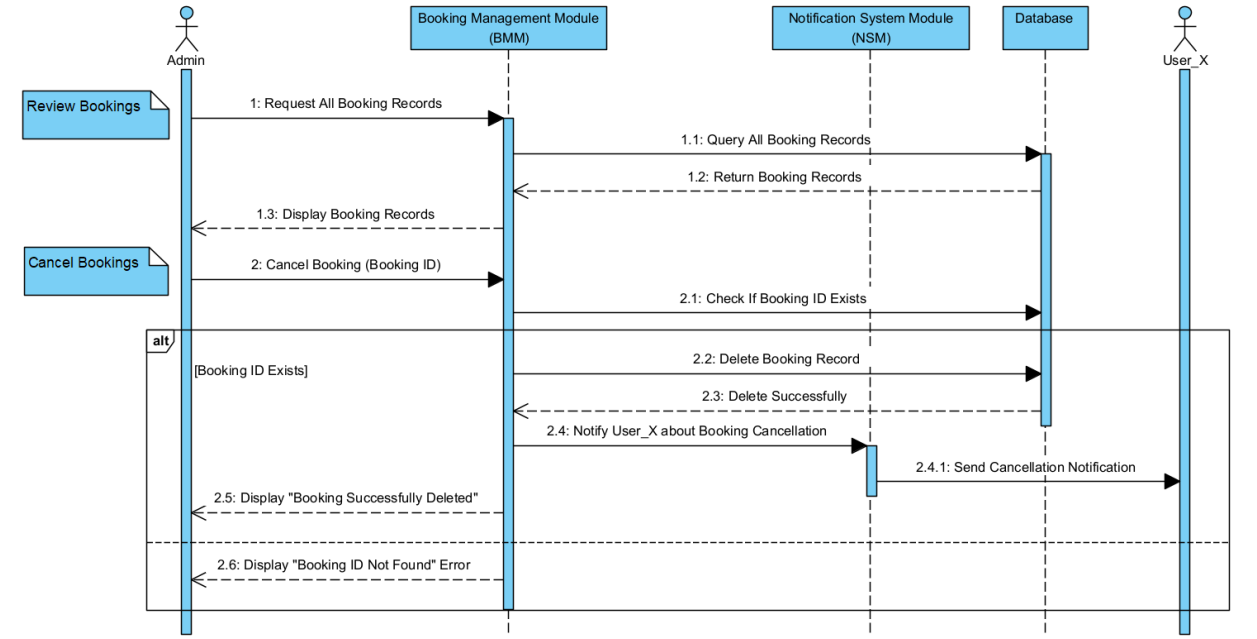


Figure 11: Sequence Diagram for Booking Management

Appendix B: Postman Testing Results

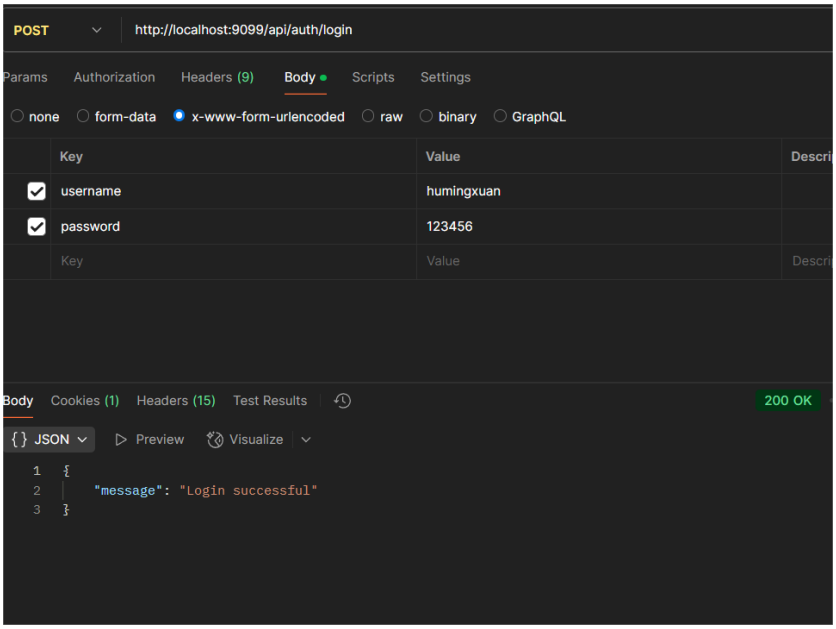


Figure 16: Postman Test for Successful User Login via /login Endpoint

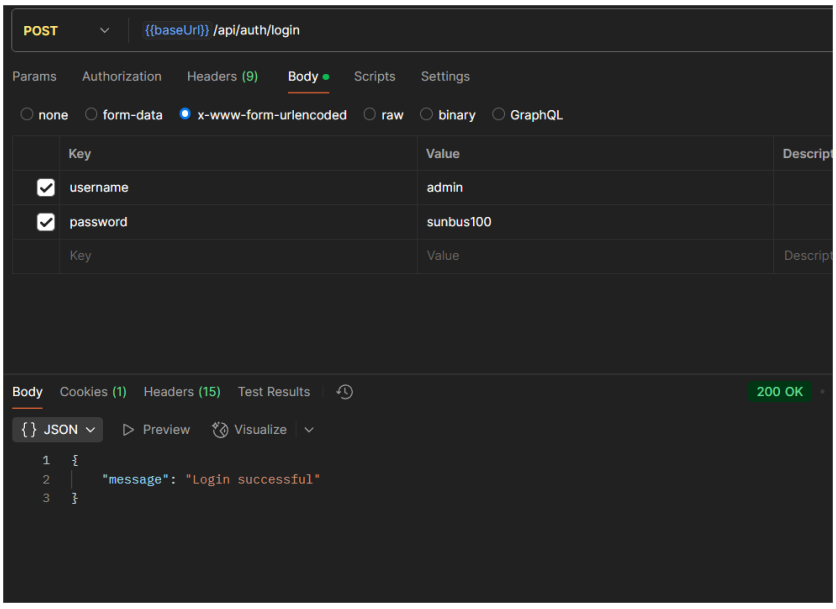


Figure 17: Postman Test for Successful Administrator Login via /login Endpoint.

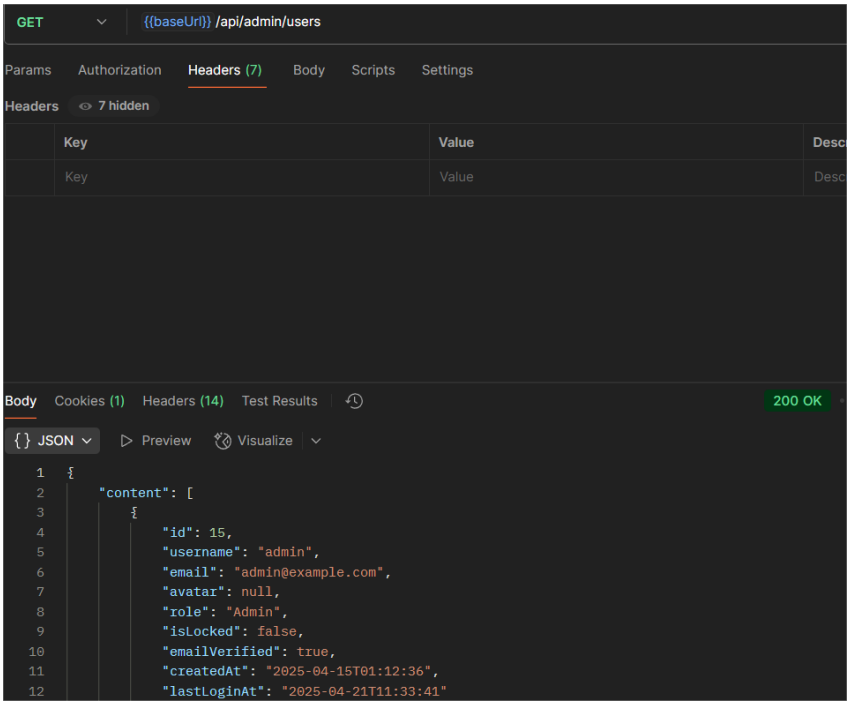


Figure 18: Postman Test for Successful Admin User List Retrieval via /admin/users Endpoint

Appendix C: Sprint Backlogs

Sprint 1		
Duration	start date: 1th March	end date: 14th March
Purpose	- Smooth out the **basic functional processes** - Realize the functions of meeting room browsing and filtering - Ensure that the basic registration and authentication process for users is correct	
Result	- Users can successfully register and log in ➜ They can browse, search and filter meeting room information - Distinction between administrator and regular user permissions ➜ Administrators complete the addition, deletion, modification, and query of the meeting room	

Sprint 2		
Duration	start date: 18th March	end date: 2th April
Purpose	- Complete **core business processes**: meeting room reservation, reservation approval, and account information management - Improve the entire process operation of meeting room reservation - Support users to view and manage personal appointments	
Result	- Users can successfully create, modify and cancel reservations ➜ Administrators can reject reservation requests ➜ Users can view and manage their appointment records - Realize basic account management functions such as account password modification	

Sprint 3		
Duration	start date: 5th March	end date: 18th April
Purpose	- The system functions are complete and security is optimized - Enhance user experience - Carry out data encryption and protection to enhance system security	
Result	- Provide a reminder function for appointment status updates - The overall security protection of the system is in place (encryption, anti-SQL injection, etc.) - Support administrators in batch exporting user data	

Figure 20: Sprint Overview

A Sprint					
	Mon	Tue	Wed	Thu	Fri
Week1	PBI Grooming; Daily scrum	Daily scrum	PBI Grooming, Daily scrum	Daily scrum	Daily scrum
Week2	PBI Grooming; Daily scrum	Daily scrum	PBI Grooming, Daily scrum	Daily scrum	Sprint Review & Retrospective

Figure 21: Sprint Schedule

Name	Function modules	Functional sub-item	Sprint Allocation
Liu ziqi	6. Account Management	6.1 Enhanced User Profile Update	Sprint 2
		6.2 Change Password	Sprint 2
		6.3 Admin Lock User Account	Sprint 3
		6.4 Admin Unlock User Account	Sprint 3
		6.5 Admin Bulk Export User Data	Sprint 3
Tu peiling	7. Search and Filtering	7.1 Basic Meeting Room Search Function	Sprint 1
		7.2 Filter Meeting Rooms by Capacity	Sprint 1
		7.3 Filter Meeting Rooms by Location	Sprint 1
		7.4 Filter Meeting Rooms by IT Equipment	Sprint 1
		7.5 Combined Filtering and Results Sorting	Sprint 2
Li yilin	8.Information display	8.1 View available meeting rooms with filtering and sorting	Sprint 1
		8.2 View detailed room information	Sprint 1
		8.3 View and manage personal bookings	Sprint 2
		8.4 Search bookings by date or room name	Sprint 2
		8.5 Set reminders for upcoming bookings	Sprint 3
Wu fan	9. Security optimization	9.1 Encrypt Personal Information	Sprint 3
		9.2 Block Repeated Failed Login Attempts	Sprint 3
		9.3 Enforce Session Timeout and Inactivity Handling	Sprint 3
		9.4 Provide Immediate Feedback	Sprint 3
		9.5 Provide Pre-fill Defaults	Sprint 3

Figure 22: Sprint Task Allocation (Team Members 1–4)

Name	Function modules	Functional sub-item	Sprint Allocation
Hu mingxuan	1. User registration and authentication	1.1 User Registration Page	Sprint 1
		1.2 Email Verification	Sprint 1
		1.3 Avoid Duplicate Registration	Sprint 1
		1.4 User Password Strength Verification	Sprint 1
		1.5 Registration Success Page	Sprint 1
Zhou zhenghao	2. Role permission management	2.1 Role-Based Access for Regular Users	Sprint 1
		2.2 Role-Based Access for Administrator	Sprint 1
		2.3 Secure Login for Regular Users	Sprint 1
		2.4 Forgot Password and Account Recovery	Sprint 1
		2.5 Login Attempt Limitation and Password Expiry Management	Sprint 1
Sang rui	3. Meeting room management	3.1 Display a List of All Meeting Rooms	Sprint 1
		3.2 View Meeting Room Details	Sprint 1
		3.3 Add a New Meeting Room	Sprint 1
		3.4 Edit Meeting Room Information	Sprint 1
		3.5 Delete a Meeting Room	Sprint 1
Zhou ziyang	4. Meeting room booking	4.1 User Books Meeting Room Creation	Sprint 2
		4.2 User Modifies Meeting Room Booking	Sprint 2
		4.3 User Cancels Meeting Room Creation	Sprint 2
		4.4 Booking Reminder Notifications	Sprint 3
		4.5 Inviting Mate Option	Sprint 3
Jiang jinhong	5. Appointment approval	5.1 View a List of Booking Information	Sprint 2
		5.2 Accept the Booking	Sprint 2
		5.3 Reject the Booking	Sprint 2
		5.4 View a List of User Information	Sprint 3
		5.5 Support a List of Log Information for Checking	Sprint 3

Figure 23: Sprint Task Allocation (Team Members 5–9)

Appendix D: System UI Interfaces

Our system's front-end interface is designed with ***HTML***, ***JavaScript***, ***CSS***, ***Bootstrap***, and ***jQuery***, providing a smooth, interactive, and visually appealing user experience. Utilizing ***AJAX*** technology, content dynamically updates without page refreshes, greatly enhancing responsiveness. The **interactive design** guides users seamlessly through tasks, while the **responsive layout** ensures optimal viewing across various devices. Additionally, **day and night color modes** offer personalized comfort, and the clean aesthetics combined with efficient code structure ensure performance and maintainability.

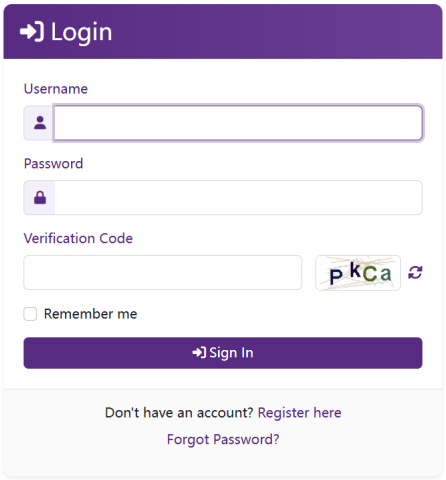


Figure 24: User Login Page

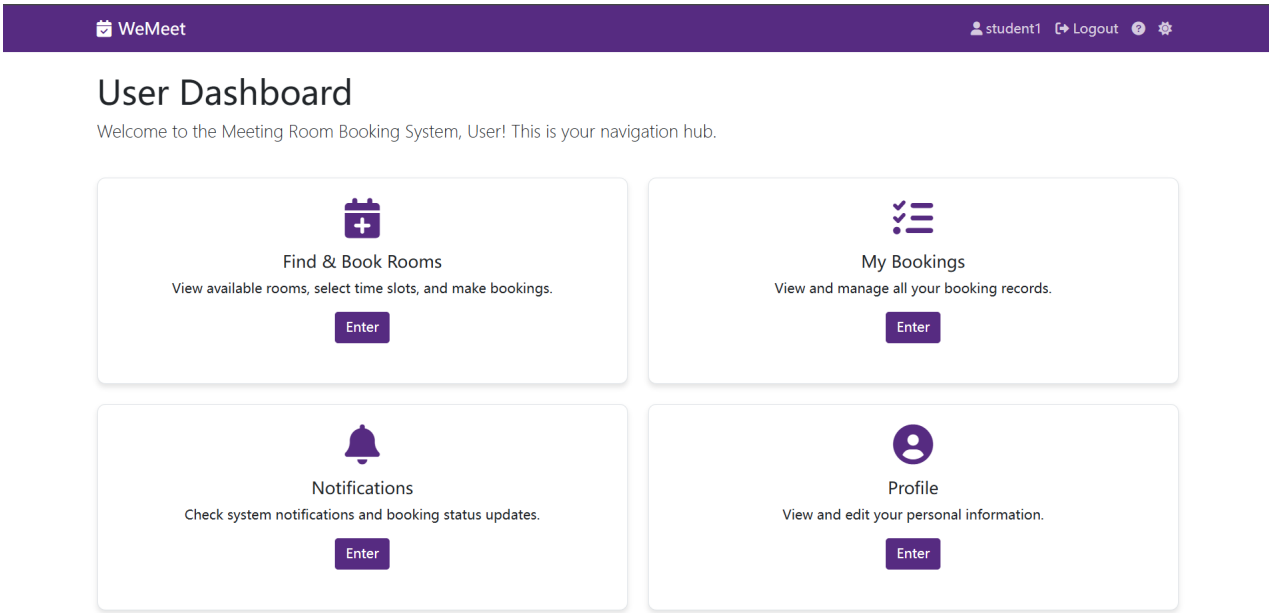


Figure 25: User Dashboard Page

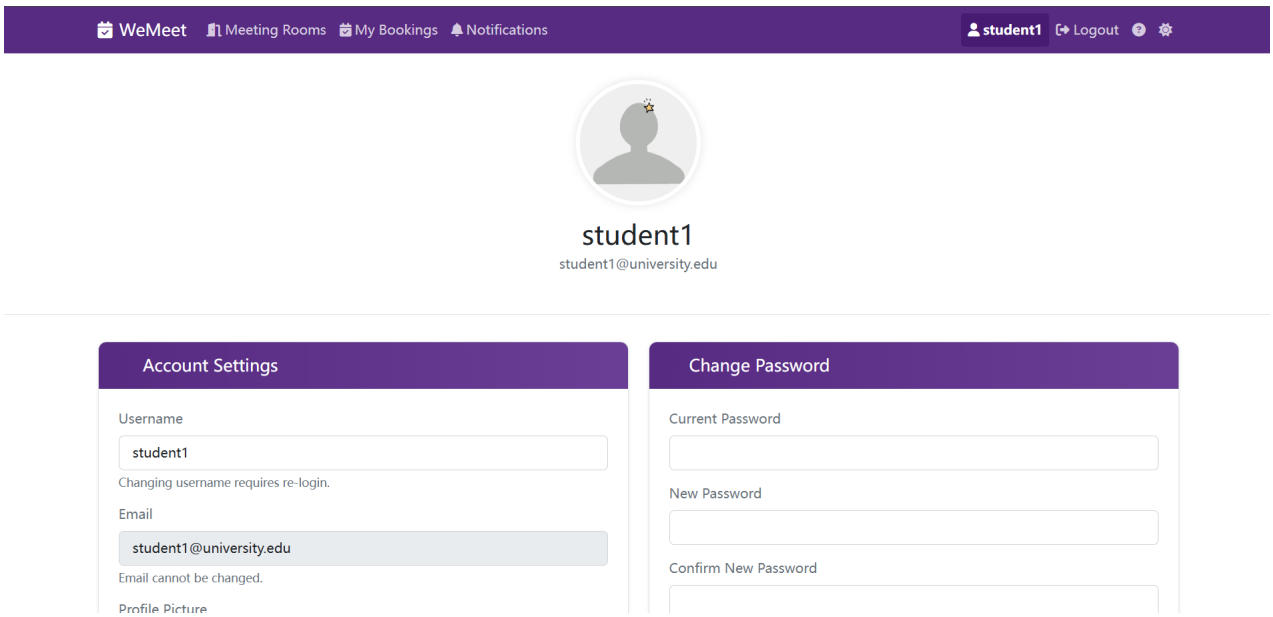


Figure 26: User Profile Page

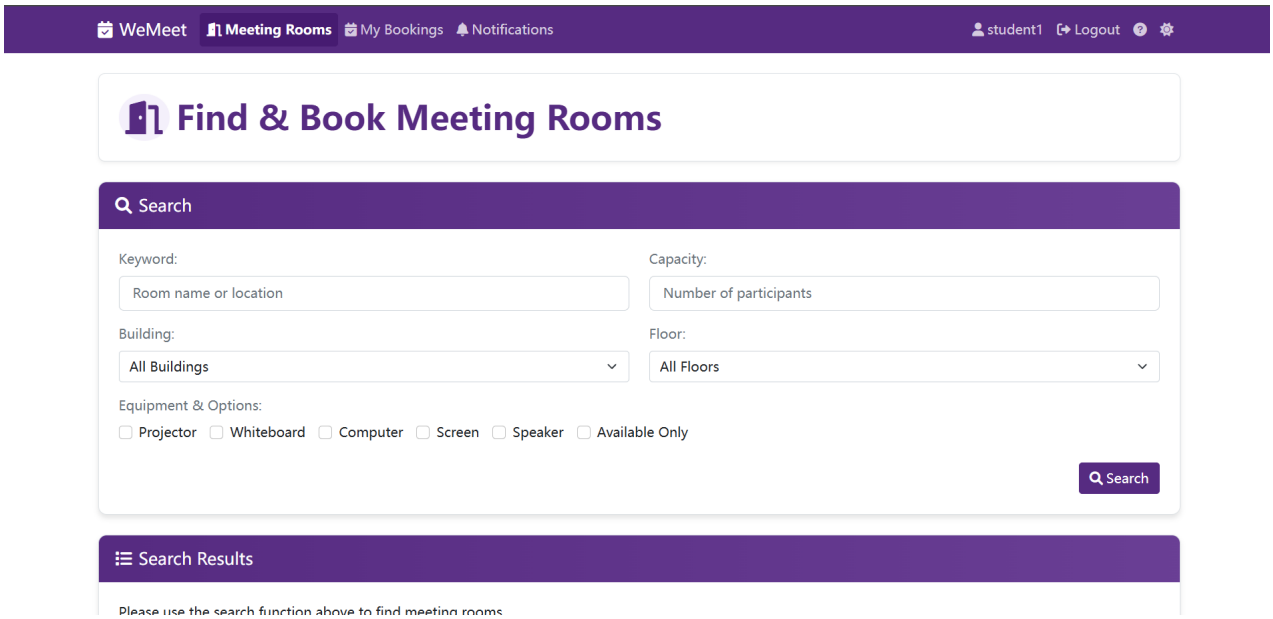


Figure 27: Room Search Page

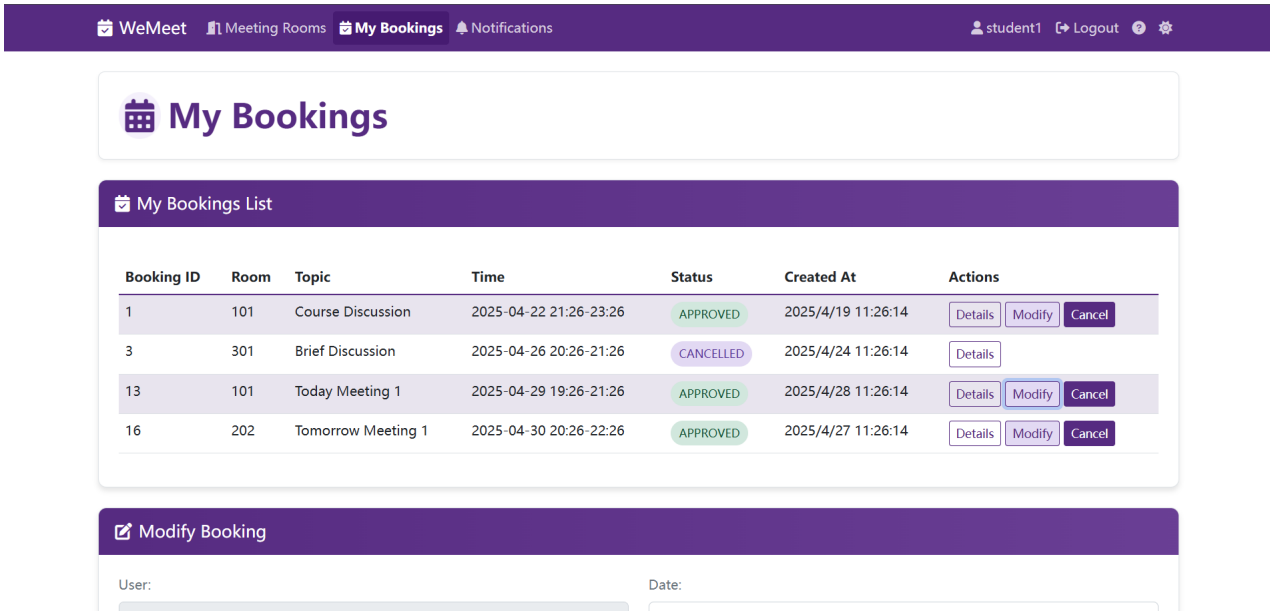


Figure 28: My Bookings Page

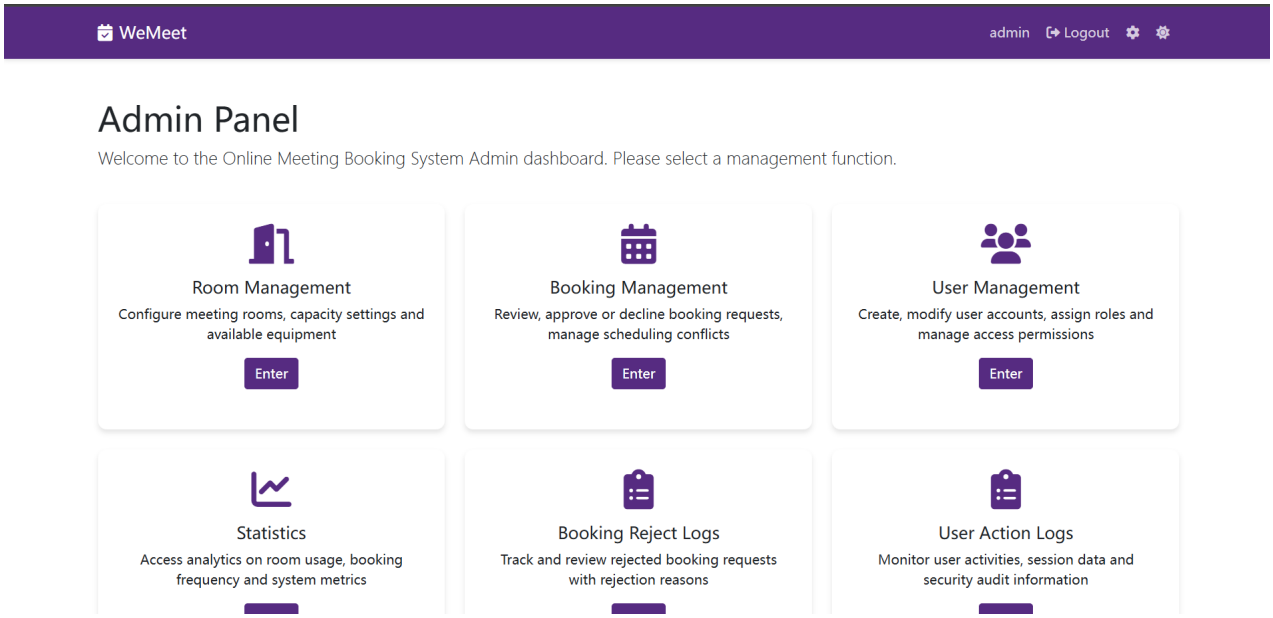


Figure 29: Admin Panel Page

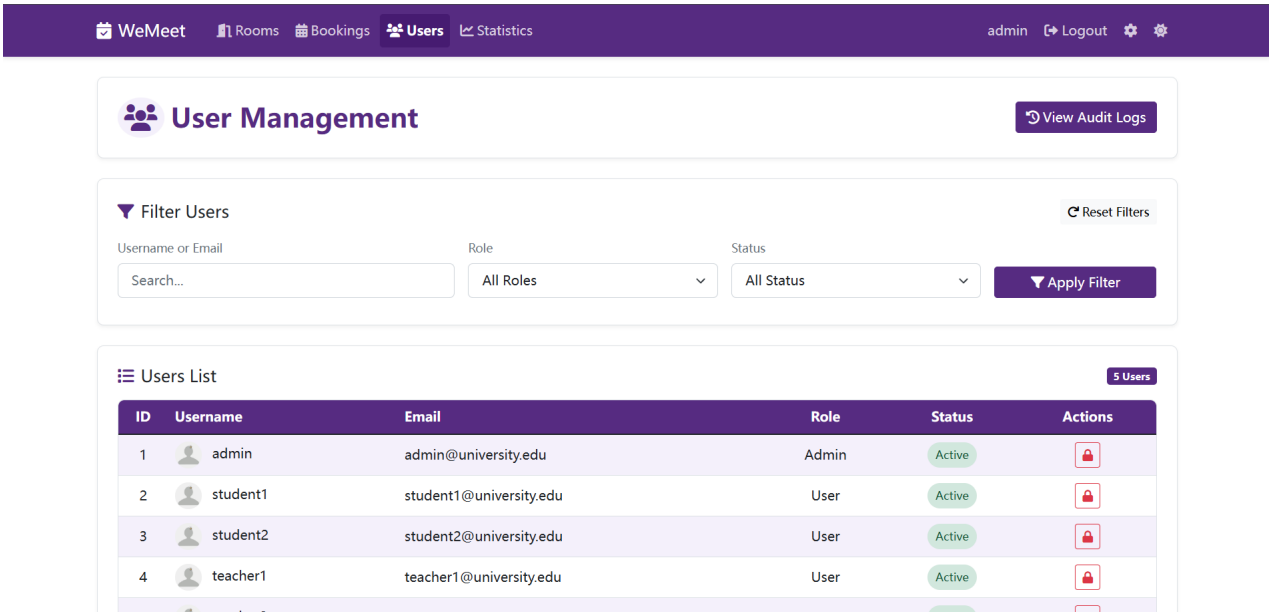


Figure 30: User Management Page

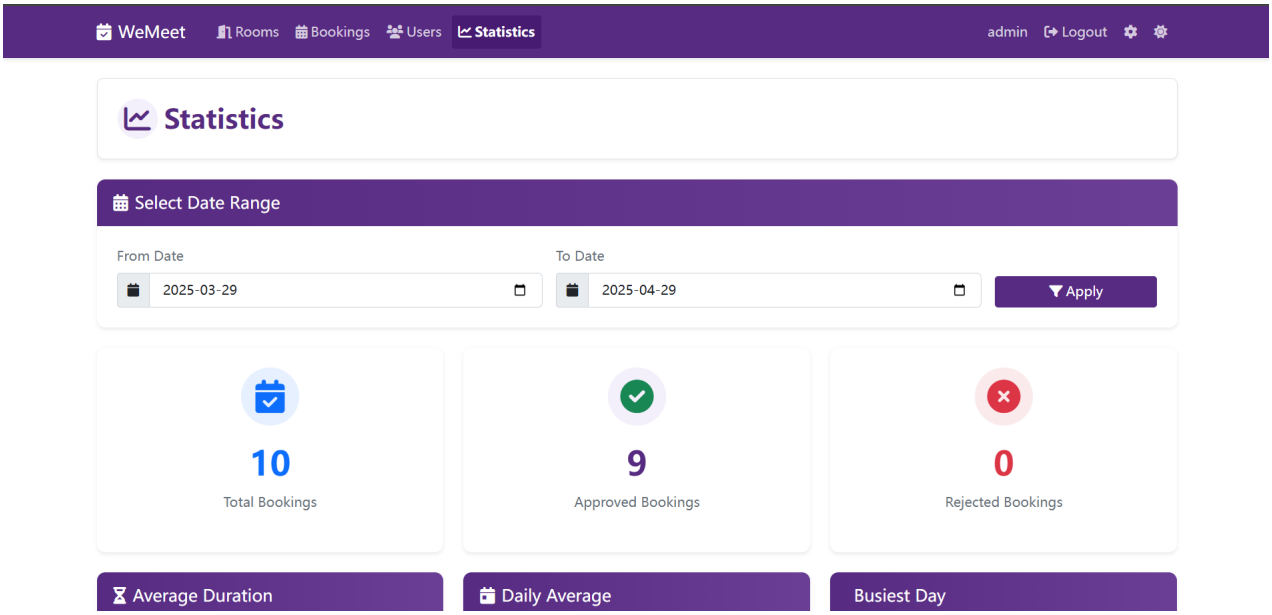


Figure 31: Statistics Overview Page

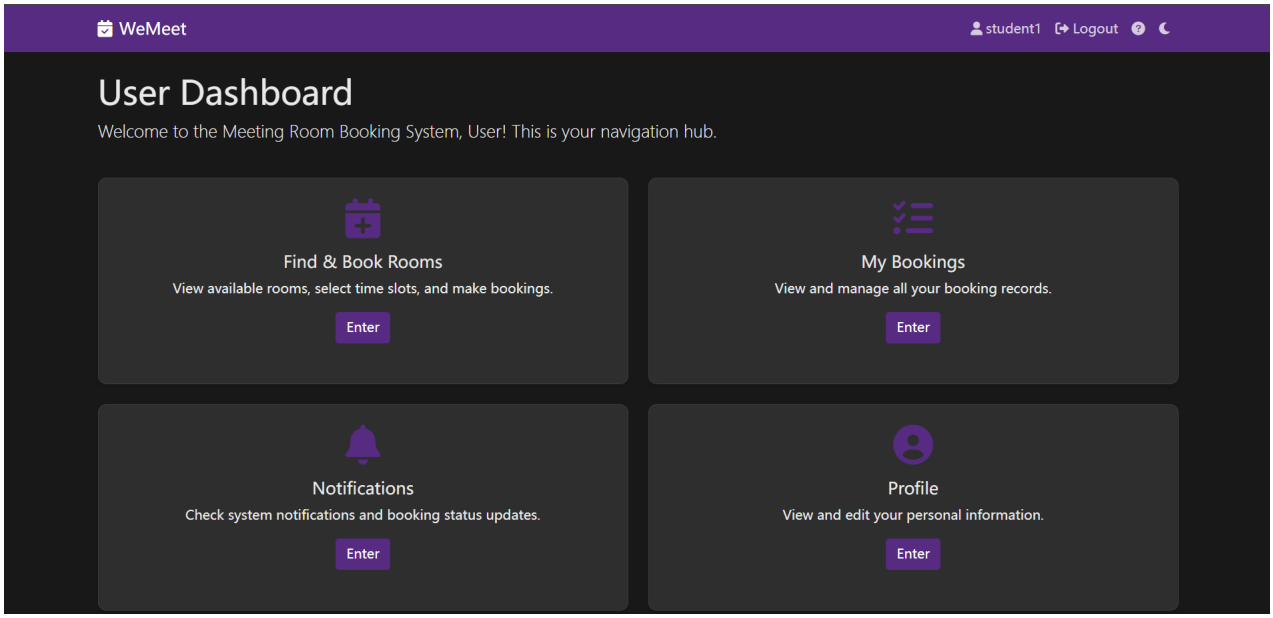


Figure 32: User Dashboard Page (Night Mode)

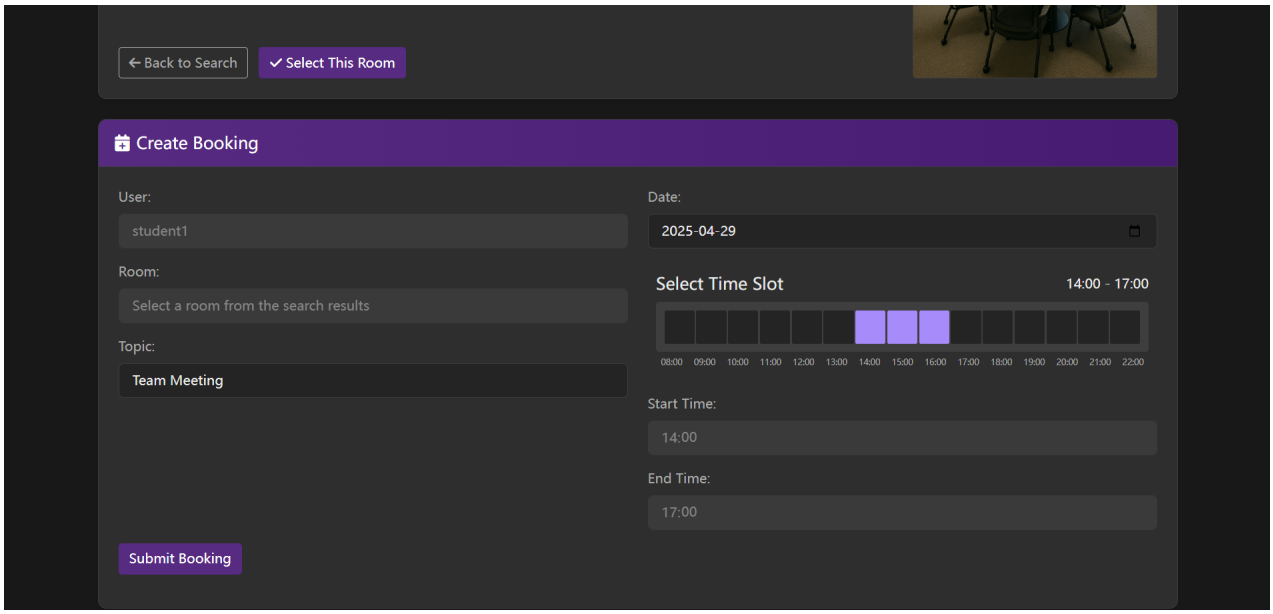


Figure 33: Room Booking Page (Night Mode)

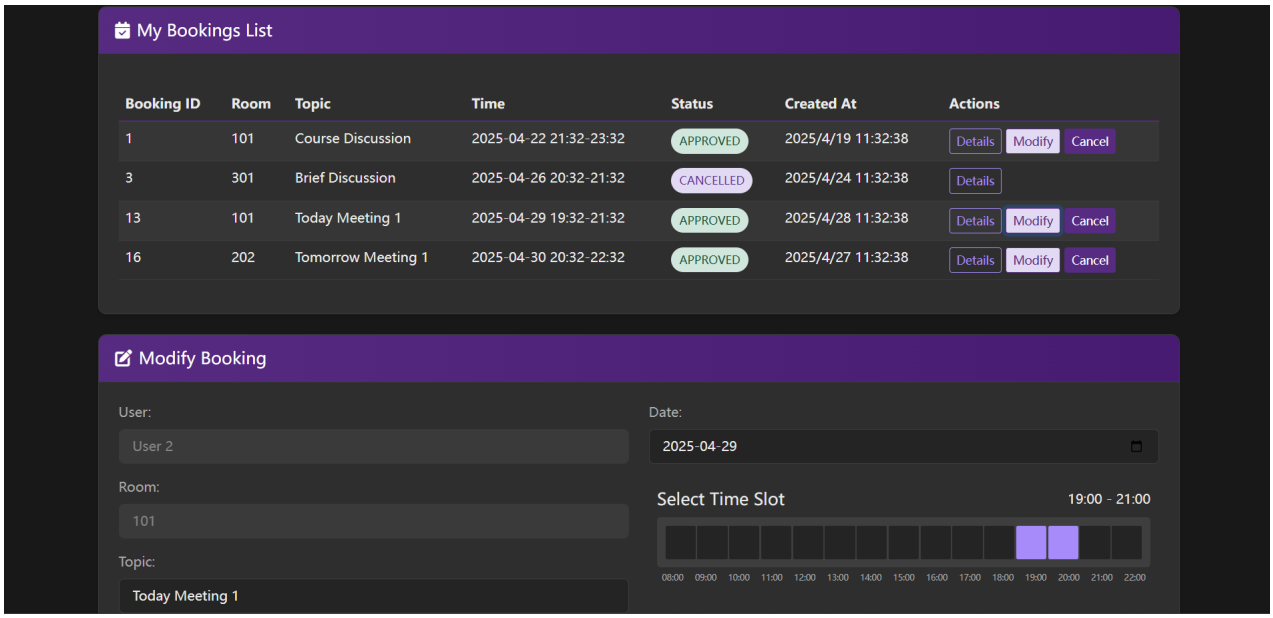


Figure 34: Room Modifying Page (Night Mode)

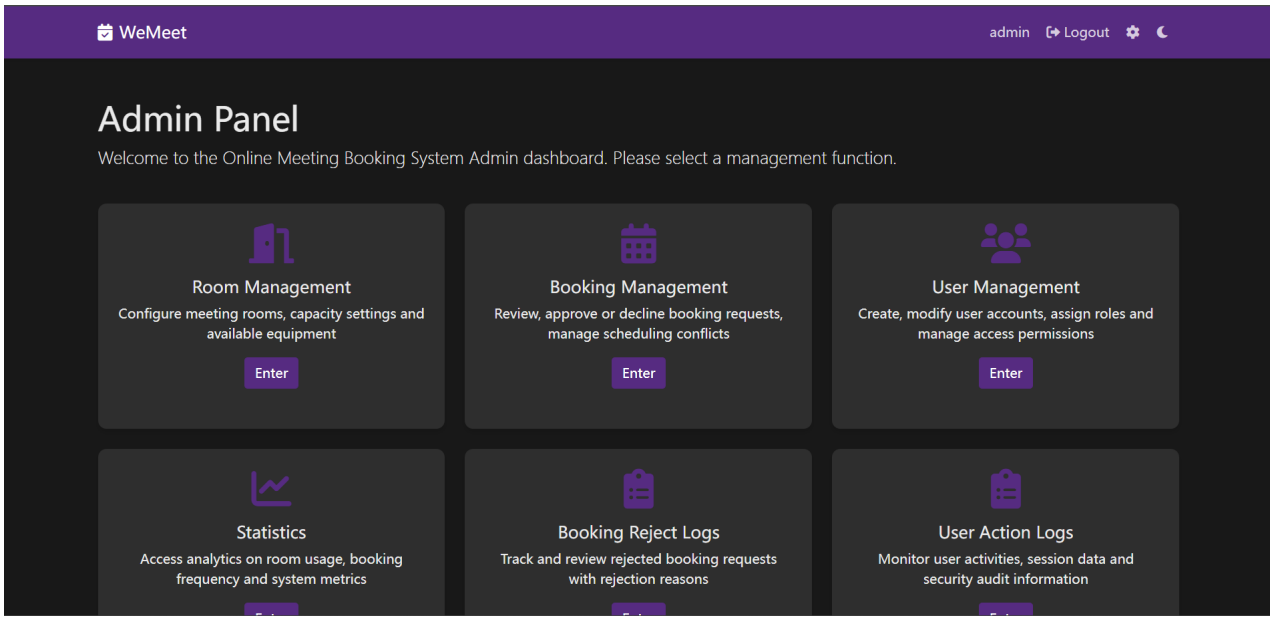


Figure 35: Admin Panel Page (Night Mode)

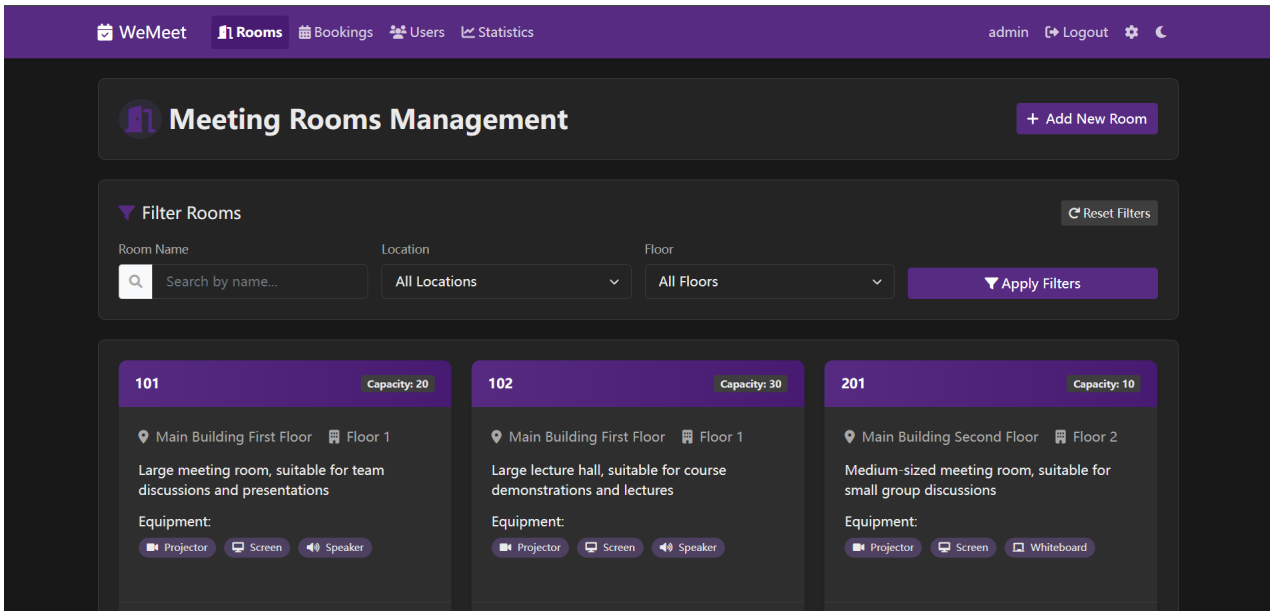


Figure 36: Meeting Room Management Page (Night Mode)

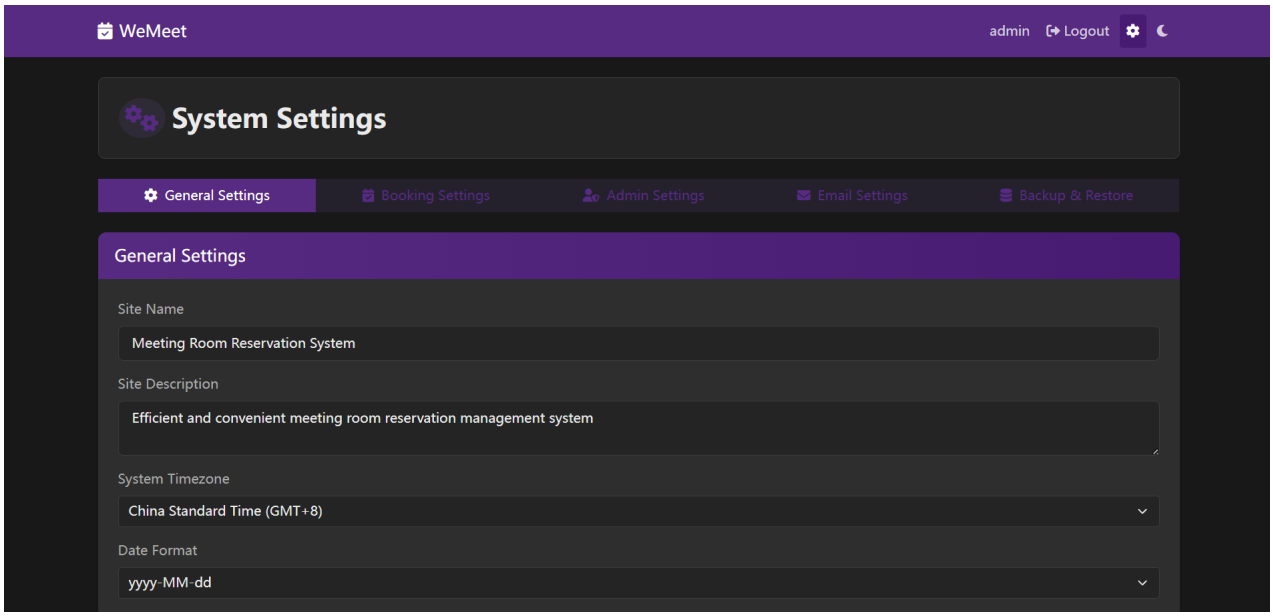


Figure 37: System Settings Page (Night Mode)