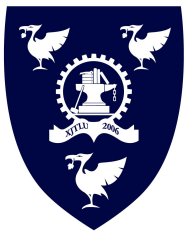

CPT202 Assignment 3

Individual Report for Software Engineering Project

An online meeting booking system

2024/2025 Semester 2

May 20th, 2025

URL to Project: http://114.55.137.200

Group number: G33

Edited by: Junhao.Huang2202, 2256792

Default Account Credentials for Testing:

Account Type	Username	Password
User	test	12345678
Admin	test_admin	12345678

Contents

1. Introduction

Our project created an **"online meeting booking system"** to address inefficiency and booking conflicts in university meeting room management. The product is a web application with a separate `Vue.js` front end and a `Spring Boot` back end [2], enabling real-time updates and improved campus meeting room management.

Our team provided a comprehensive booking platform for two primary user groups: **students** and **administrators**. Students, as normal users, can create accounts, maintain personal information, intelligently filter meeting rooms (by capacity, facilities, location), book time slots, and manage their bookings (viewing, modification, cancellation). Administrators have more extensive control, including managing meeting room resources, user account administration (like locking/unlocking accounts), centralized approval and status tracking of booking requests, and processing user feedback.

My key individual responsibilities were the construction of the **authentication mechanism** and **user profile modification**. The module I delivered incorporates core features for **security** and **user experience**. New user identity authenticity during registration is ensured through **`CAPTCHA` via email**. User passwords are **encrypted** using a "hashing and salting (Group33)" method to safeguard logins. I developed a **login authentication process** integrated with **session management** and added a **request time limit to `CAPTCHA` requests**, maintaining user login status and system flow control. The system also allows users to autonomously **update their login passwords and registered email addresses** in their profile page, providing essential self-management. A significant achievement was designing and implementing an **Aspect-Oriented Programming (AOP)-based authorization control scheme** [1]. This uses custom annotations (`@AuthCheck`) for **role-based access control (RBAC)**, ensuring system modules are accessible only to authenticated users with operational permissions.

Subsequent sections will review the **software development methodology**, focusing on my tasks within the **Scrum agile development framework**. An analysis of the software design for my **Product Backlog Items (`PBIs`)** will be presented, with a discussion on **change management**. The report will then examine **legal, social, ethical, and professional considerations** related to my contributions. The conclusion will summarize the **project experience**, **personal insights**, and suggest **future improvements**.

2. Software Development Process

Throughout the development lifecycle of the online meeting booking system, our team fully adopted the **Scrum framework**. As a mainstream agile development methodology, Scrum's core characteristics emphasized **iterative progress**, encouraging **close collaboration** and **continuous communication** among team members, and maintaining a flexible responsiveness to evolving requirements, which highly adhered to the nature of this project, and group mates can use these plans to explain and classify what project needs. Our motivation for choosing Scrum is the expectation that its systematic practice will enable us to more effectively navigate the various uncertainties in software development. Through fixed **Sprint cycles**, we were able to deliver operational software functional modules and integrate feedback after each iteration into subsequent development iterations. This ensured that the final product could meet users' core needs and successfully achieve the project's established strategic objectives.

2.1. Individual Work Planning within the Scrum Process

Throughout the team's Sprint Iteration framework, the **"Authentication"** and **"User Profile Modification"** modules that I was responsible for developing were also systematically planned and driven according to **Scrum's iterative principles**. My development work was mainly organized around the following phased objectives across the four Sprints, and the details could be seen in the Appendix.

2.2. Rationale Behind the Scrum Plan Design

This Scrum plan for my modules was designed with several considerations to ensure logical progression and alignment with the team's overall developing progress. Each sprint was structured to build incrementally on the previous one.

- **Sprint 1** focused on laying the groundwork. The priority was to establish the **core user entity** and a basic, albeit incomplete, registration pathway. This allowed foundational data structures to be agreed upon early.

- **Sprint 2** aimed to make the system usable from an authentication perspective by **implementing login and session management**, and crucially, **adding email verification** to bolster the security and validity of registrations. This provided an early testable authentication loop.
- **Sprint 3** shifted towards enhancing **user control over their accounts (password/email updates)** and introducing a critical security layer with **AOP-based authorization**. The decision to introduce AOP at this stage was based on having stable core authentication features and a growing number of APIs that would require protection.
- **Sprint 4** was dedicated to ensuring the completeness, security, and maintainability of my modules through **comprehensive testing**, **refinement**, and **documentation**. Checking AOP implementations also ensured that as the system functionalities stabilized as well as security.

PBIs were allocated to sprints based on their logical dependencies, a desire for early feedback on core functionalities, and the technical readiness to tackle more complex features like AOP once foundational elements were in place.

2.3. Scrum's Impact on Communication and Collaboration Efficiency

The application of Scrum principles and its structured cycle significantly enhanced our team's communication and collaborative efficiency:

- **Sprint Planning:**

  The **Sprint Planning Meeting** marks the start of each new Sprint, designed to define the **Sprint Goal** and the specific tasks to achieve it. At the beginning of each Sprint, our team prioritizes **incomplete features** based on current **project progress** and existing completed modules. These **highest-priority features** are then refined into smaller, more easily traceable **Sprint Backlog items**, which are subsequently assigned fairly to all team members. By the meeting's conclusion, we generate a clear **blueprint for action** to guide development for the entire Sprint cycle.

  For instance, in planning for Sprint 2, a key discussion involved the exact **request/response JSON format** for the Login API and the email verification API, covering details like parameters, response structures, and error codes. This required **direct communication** with frontend developers to ensure a unified transmission message. Such **proactive clarification** was crucial in preventing significant rework from misaligned API specifications, directly boosting overall efficiency.

- **Implementation & Daily Scrum:**

  In this phase, our team works from the **Sprint Backlog** to develop a **product increment**, engaging in **design, coding, and testing**. To maintain progress and address issues, we hold **Daily Scrums**, typically every Wednesday, where we **synchronize progress** and discuss impediments. This ensures we can quickly adapt and meet Sprint goals.

  For instance, while developing the user authentication feature in Sprint 3, a disagreement arose on the implementation. Some suggested simple 'if' statements for role checks. However, utilizing Spring's **AOP capabilities**, I proposed creating a custom `@Around` annotation for authentication and permission control. This approach offered better **re-usability** and **extensibility** for future interfaces. My proposal was ultimately chosen by the team. These **regular discussions** are crucial for identifying **cross-task issues** and collaboratively finding solutions, thereby maintaining the team's overall efficiency.

- **Sprint Review:**

  We conducted this section at the end of each cycle, provides a valuable platform for demonstrating our completed **product increment** to **stakeholders**, which includes course instructors, TAs, and group mates, and our target was to gather their **direct feedback**. This input is essential for me to assess if the features I developed meet expectations. Based on this, I can refine the **Product Backlog**, making necessary adjustments to guide future development.

  For instance, after Sprint 2, a TA's concern about preventing **fraudulent email API requests** was noted. I ensured this became a **PBIs for the next Sprint**, and its implementation enhanced system stability. Later,

after Sprint 3, I demonstrated the new **'user password modification'** feature and showcased how our custom `@AuthCheck` annotation effectively secured the admin-only **'delete user'** API, preventing unauthorized access. Through such demonstrations, we receive **immediate feedback**, enabling continuous, **collective wisdom-driven iteration** and optimization of the product's user experience.

- **Sprint Retrospective:**

  We schedule the **Sprint Retrospective** immediately after the Sprint Review and before the next Sprint Planning meeting. It serves as an **internal team activity** dedicated to **self-reflection** and **continuous improvement**. The primary objective is to collectively review the just-concluded Sprint, examining our **technology stack**, **team collaboration**, and **project progress**. We openly discuss what aspects were successful and should be maintained, identify any **problems or shortcomings**, collaboratively explore their **root causes**, and formulate specific, actionable improvement measures to apply in subsequent Sprints.

  For instance, after Sprint 2, our team identified a significant issue: the lack of easily accessible **API documentation** for backend services, including my authentication endpoints. This led to frequent questions from frontend developers and somewhat slowed down **integration efficiency**. To address this pain point, in the following Sprints, I allocating time to generate and update API documentation for my modules, utilizing tools like **Knife4j**. This decision, stemming from **collective team reflection**, reduced **communication barriers** and friction between backend and frontend developers.

This cyclical process of planning, executing, inspecting, and adapting, inherent in Scrum, fostered a highly communicative and efficient development environment. It ensured that my work on the critical authentication and user profile components was not only technically sound but also well-aligned with the team's progress and evolving needs. The detailed Sprint time arrangement could be seen in Appendix, in Figure. 7, 8, 9, 10

3. Software Design

This section details the design philosophy and specific implementation of the **Product Backlog Items (PBIs)** for which I was individually responsible. As mentioned earlier, these PBIs primarily revolve around the core functional areas of the **User Authentication Module** and **User Profile Management**, as well as include the implementation of a practical **Aspect-Oriented Programming (AOP)-based authorization mechanism**. Throughout the design and development process, **system security**, **usability**, and subsequent **maintainability** were the fields I mainly focus on.

3.1. General Design and Implementation of My PBIs

My individual contributions can be demarcated into several key functional areas, each comprising a series of interconnected PBIs aimed at collaboratively delivering a complete feature set. The main functions implemented include "Authentication" and "Profile modification", which is illustrated in the Figure. 1, and Table. 1

3.1.1. PBI Group 1: User On-boarding – Secure Registration & Identity Verification

Design & Implementation Overview:

- An API endpoint (`POST /user/register`) to handle new user sign-ups, accepting username, email, password, an optional avatar (`MultipartFile`), and a verification code.
- An email-based One-Time Password (OTP) system (`GET /user/ask_code`) was implemented to validate user email addresses before completing registration. This involved generating a random code, sending it via the `EmailSender` utility, and storing it temporarily in a `Caffeine` cache for verification (`UsersServiceImpl.askCode`, `UsersServiceImpl.verifyEmailCode`).
- To prevent abuse of the OTP service, a `RateLimiter` was implemented, restricting OTP requests from the same IP address to one per minute (`RateLimiter.java`).
- User passwords submitted during registration are securely stored using an MD5 hash combined with a static salt ("group33") (`UsersServiceImpl.getEncryptPassword`).

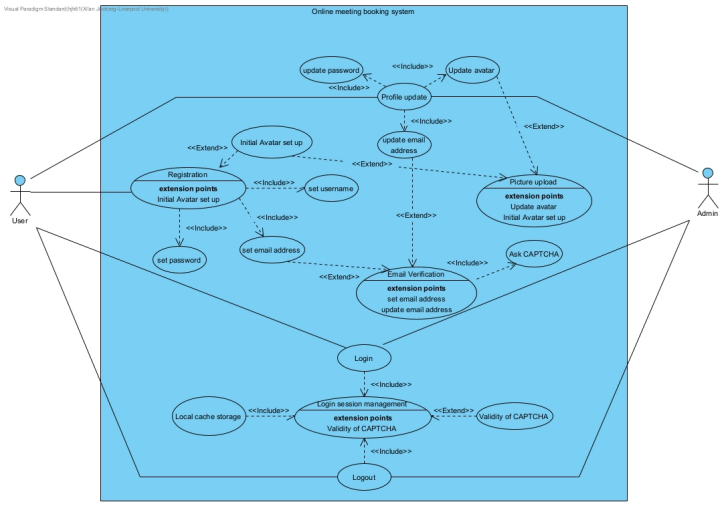

Figure 1: The Use Case Diagram of My Developed Functions

- Optional avatar uploads during registration are processed by `QiniuService` and the resulting URL/key is stored in the `users` table (`UsersController.userRegister`, `UsersServiceImpl.userRegister`, `QiniuService.uploadUserAvatar`).

Justification: Email OTP verification and request rate limiting ensure the authenticity of registrations and prevent abuse. Password hashing secures credentials. Cloud storage for avatars reduces server load and optimizes access speed.

3.1.2. PBI Group 2: User Access Control – Login, Session Management & Logout

Design & Implementation Overview:

- A login API endpoint (`POST /user/login`) was developed, accepting either a username or email along with a password (`UserLoginRequest`). The system verifies the provided password against the stored hash.
- Upon successful authentication, a server-side session is established using `HttpServletRequest` and `HttpSession`. The authenticated `Users` object is stored in the session under the key `UserConstant.USER_LOGIN_STATE`
- A dedicated endpoint (`GET /user/me`) allows authenticated users to retrieve their own sanitized profile information (`LoginUserVO`).
- A logout endpoint (`POST /user/logout`) invalidates the current user's session, securely terminating their access.

Justification: A secure login mechanism is fundamental to system access control. Session management maintains user context. A clear logout function ensures user account security.

3.1.3. PBI Group 3: User Account Maintenance – Profile Updates

Design & Implementation Overview:

- Functionality to change username (`POST /user/change-username`).
- Both functionality of updating email address (`POST /user/change-email`) and password reset/change mechanism (`POST /user/reset-password`), re-utilizes the email OTP verification process for the new email address to ensure its validity.
- Users can upload a new avatar or update their existing one via `POST /user/upload-avatar`, which interacts with `QiniuService`. An endpoint to delete an avatar (`POST /user/delete-avatar`, admin-only) was also provided.

Justification: User self-management of profile information and credentials is standard and essential for data accuracy and account security. Secondary CAPTCHA verification for critical changes enhances security.

3.1.4. PBI Group 4: System Security Enhancement – AOP-based Authorization

- A custom Java annotation, `@AuthCheck`, was created (`AuthCheck.java`). This annotation can be applied to controller methods and includes a `mustRole` parameter to specify the required user role (`"ADMIN"`, `"USER"`).
- An Aspect, `AuthInterceptor.java`, was developed using Spring AOP. This interceptor uses `@Around` advice to target methods annotated with `@AuthCheck`.
- Inside the interceptor, the `mustRole` value from the annotation is retrieved. The current logged-in user's details (including their role) are fetched from the HTTP session via `UsersService.getLoginUser(request)`.
- The interceptor then performs the authorization check on those methods with `@AuthCheck` annotations based on login user's role entity
- This mechanism is applied to various `UsersController` methods, such as `POST /user/update` and `POST /user/list`, which are restricted to admin users (`@AuthCheck(mustRole = UserConstant.ADMIN_ROLE)`).

Justification: AOP decouples authorization logic from business logic, improving code readability and maintainability. Declarative permission configuration simplifies security management.

3.2. Detailed Design and Implementation of 2 Core PBIs

From the PBIs I developed, two stand out due to their foundational importance and demonstration of key software engineering principles: **Secure User Registration with Email Verification** and **AOP-based Authorization for API Endpoints**.

3.2.1. Secure User Registration with Email Verification

- **User Story:** As a **new user**, I want to **register for an account using my email address and verify it with a CAPTCHA** so that I can securely create my account, confirm my email ownership, and gain access to the platform's services. Acceptance criteria could be seen in Table. 2
- **Logical Flow:** User registration starts with form submission to the `/user/register` endpoint. Before this, users request an email OTP via `/user/ask_code`, involving rate limiting, OTP dispatch, and temporary caching. The core registration logic then validates the OTP. Upon successful validation, it checks for username and email uniqueness and securely encrypts the password. A new user entity is created (default role `'USER'`), an optional avatar is processed (uploaded to cloud storage), and finally, the user data is persisted to the database. The OTP cache is then cleared, and a user object (`UserVO`) is returned. Validation failures trigger a `BusinessException`. Diagram could be referred in Figure. 19, 20
- **Database Support:**

  This PBI heavily relies on the `users` table. The `email` field's `NOT NULL UNIQUE` constraint is fundamental for email verification and user uniqueness. The `password` field stores the encrypted digest for security. The `username` is ensured unique at the business logic layer. `avatar_url` stores the avatar link, and `role` defaults to `'user'`. Timestamps like `created_at` and `updated_at` aid auditing. Figure. 6 [3] [4]
- **States of Objects Involved:**
  - **Users Entity Object:** Transitions from a transient in-memory state (during registration phase, such as setting username, hashed password, role to `'USER'`) to a persistent state (after successful database insertion, gaining a `user_id` and timestamps). Figure. 2
  - **CAPTCHA in LOCAL_CACHE:** Has two main states: `"Active & Valid"` (exists in cache, not expired) and `"Expired/Invalidated"` (removed due to TTL or usage). The `RateLimiter` also maintains cached states for IP request times. Figure. 3

- **UI Description:** Students could use registration page to create a account. The registration form includes fields for username, student ID, email, OTP, and password, with optional avatar upload. A button requests an email CAPTCHA from `/user/ask_code`, then briefly disables. Client-side validation checks email/password rules. Successful submission prompts an alert and redirects to login. Figure. 22, 23, 24

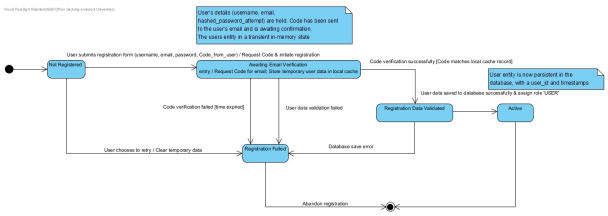

Figure 2: User Entity States During Registration

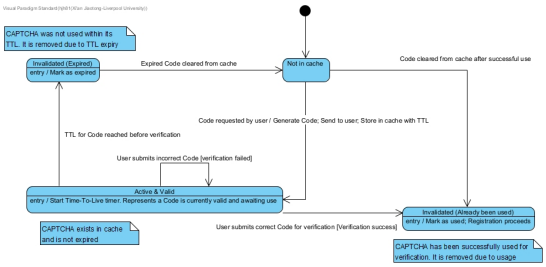

Figure 3: Email CAPTCHA States in Local Cache

3.2.2. Constructing an AOP-based Authorization Mechanism for API Endpoints

- **User Story:** As a **system administrator**, I want API endpoints to be **protected by a clear, role-based access control mechanism** so that only authorized users can access specific functionalities and data, thereby safeguarding system security. Acceptance criteria could be seen in Table. 2

- **Logical Flow:** When a request targets a controller method annotated with `@AuthCheck` (an admin-only endpoint), Spring AOP intercepts the call before method execution. An interceptor then retrieves the required role (`mustRole`) from the annotation and the current user's role from their session. Authorization logic is then applied: access is granted if `mustRole` isn't specified or if the user is an admin. If roles mismatch (a regular user attempts an admin action), a `BusinessException` is thrown, preventing unauthorized access. If authorization succeeds, the original controller method executes; otherwise, an appropriate error response is returned. Diagram could be referred in Figure. 4, 21

- **Database Support:** The core data for AOP authorization originates from the `role` field in the `users` table. The `AuthInterceptor` uses the logged-in user's role value (corresponding to constants in `UserConstant.java`) for its decisions. This role, defaulting to `"USER"` on registration, and its accuracy in the database, are crucial for the authorization logic. Figure. 6

- **States of Objects Involved:**

  - **`@AuthCheck` Annotation Instance:** Its `mustRole` attribute value is determined at compile/deployment time and read by the interceptor at runtime.
  - **Users Entity Object (Logged-in User):** Its `role` property (established at login and stored in the session) is the key state for authorization checks. Figure. 5

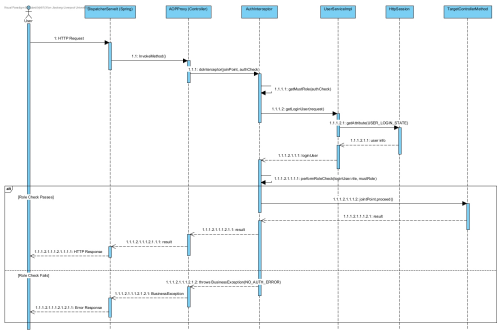

Figure 4: Sequence Diagram of AOP Authorization

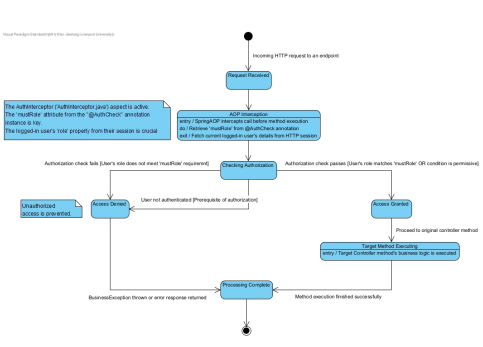

Figure 5: Access Request States During AOP Authorization

- **UI Description** For UI authorization, if a normal user attempts to directly access an admin-only URL, the backend `AuthInterceptor` blocks the request and redirects them to the homepage. On pages shared by different roles, the UI dynamically adapts by hiding features or buttons that the current user is not authorized to access based on their role. This ensures users only see and interact with functionalities appropriate for their permissions, providing a clear and secure experience. Figure. 25, 26

4. Change Management

4.1. Our Team's Approach to Adapting in Scrum

In our project, we knew changes were inevitable, so Scrum's adaptive approach was key for our team. We kept our **Product Backlog dynamic**, incorporating new `PBIs` based on ongoing discussions or feedback from our TA, who acted as our Product Owner. Our group leader then prioritize these revised tasks. If a significant change request emerged mid-Sprint that threatened our Sprint Goal, we'd **discuss it as a team to decide the best course of action**, to ensure we kept our Sprint on track and could still deliver value.

4.2. Case Studies in Managing Change

4.2.1. Case Study 1: Optimizing CAPTCHA Cache and User Session Management

- **Context & Original Approach (Sprint 2):**
  During Sprint 2, while designing management for CAPTCHA and user session IDs, we initially considered a potentially double caching strategy (`Redis` + `Caffeine`) for session information.
- **Reason for Change & Identification (Daily Scrum):**
  As we detailed implementing user login state, a Daily Scrum in Sprint 2 revealed a complex distributed cache for user sessions was unnecessary for our project's scale and will cause extra learning time. The priority became robust, simple OTP handling and effective `HttpSession` use.
- **Solution & Adaptation:**
  The team proposed simplifying. The `PBI` was adjusted to use a lightweight local cache (e.g., `Caffeine`) for OTPs (due to their ephemeral nature). For user login state, we committed to using standard `HttpSession` provided by the servlet container, storing user objects (like `UserVO`). This change, quickly adopted after the daily discussion, reduced session management overhead, letting us focus on core OTP security and session data handling.

4.2.2. Case Study 2: Streamlining Role-Based Login and Redirection

- **Original Requirement (Sprint 1):**
  In Sprint 1, the login interface initially required users to manually select their role (e.g., `'USER'` or `'ADMIN'`) at login. The backend would then validate credentials against this selected role.
- **Reason for Change & Identification (Sprint Retrospective):**
  During the Sprint Retrospective after Sprint 1, the team found this manual role selection cumbersome and unintuitive for users. It was an unnecessary step, as the system knew the user's role after authentication.
- **Solution & Adaptation:**
  It was decided to revise this `PBI` for Sprint 2. Manual role selection at login was removed. The backend would now auto-detect the user's role post-authentication and redirect them to the correct dashboard. This change greatly simplified the login UI and improved the user experience.

5. Legal, Social, Ethical, and Professional Issues

5.1. Legal Issues

Legally, our project prioritized protecting **user account privacy**. A key measure was **never storing plaintext passwords**; instead, all passwords are saved as **hashed and salted ciphertext** in the database, significantly enhancing security against **data breaches**. We also paid close attention to **software licensing**, ensuring all third-party libraries (e.g., `Spring Boot`, `Knife4j`) were used in **strict compliance** with their declared open-source terms, respecting **intellectual property rights**.

5.2. Social Issues

Socially, we aimed for an **accessible** and **intuitive interface**. We recognized the future value of **internationalization (I18N)** for broader **inclusivity** and considered providing user manuals or clear **onboarding** to help new users effectively utilize the system. This promotes **ease of use** and **equitable access** to functionalities for all students, enhancing the system's positive **social impact** within our campus community by making it welcoming to a **diverse user base**.

5.3. Ethical Issues

Ethically, our foremost concern was **responsible user data handling**, protecting it from **unauthorized access** or **misuse**. For instance, normal users are explicitly prevented from viewing other users' **private records** (like potential booking history) or altering their identity details, ensuring **data segregation** and **individual privacy**. Additionally, **transparency** is key; if an admin locks a user's account, the affected user receives a **clear notification** on their homepage, ensuring they are **promptly informed** of their account status. Such practices, ensuring **data integrity** and **user awareness**, are vital for maintaining **trust**.

5.4. Professional Issues

Professionally, our team embraced **Scrum principles**, fostering strong **collaboration** through daily stand-ups, sprint reviews, and retrospectives. We practiced clear **communication** and proactive **problem-solving**, evident when standardizing API documentation with `Knife4j` or deciding on the **AOP-based authorization strategy**. Our **adaptability** in managing changes, such as refining the login flow or OTP/session management, demonstrated our commitment to **continuous improvement** and delivering a **quality**, **secure product** within the academic project's scope and timeline.

6. Conclusion

Our team successfully delivered a **secure online meeting room booking platform** using the Scrum framework. My **primary contributions centered on developing the core user authentication system**, which included implementing **secure registration with email OTP verification** using `Caffeine` for caching, **managing user sessions effectively** with `HttpSession`, and designing the **AOP-based `@AuthCheck` annotation for API endpoint authorization**. I also took responsibility for ensuring these critical modules were **clearly documented** using `Knife4j`.

This project significantly sharpened my **technical toolkit**. I gained hands-on experience **configuring `HttpSession`** for user session management and **implementing local caching with `Caffeine`** for OTPs. A **major learning was designing AOP aspects for our `@AuthCheck` authorization**, deepening my understanding of `Spring Boot`'s capabilities. Furthermore, using `Knife4j` not just for API documentation but also for **testing my endpoints became a key practice**. Beyond these technical skills, I truly valued the **agile development experience with Scrum**, which **improved my team communication, problem-solving, and self-learning abilities** when tackling new challenges.

Looking ahead, this project has **strong potential**. Future enhancements could include **internationalization (I18N) support, comprehensive user manuals, or more advanced features** like refined user analytics or integration with other campus systems. I'm enthusiastic about the possibility of these additions further enhancing its **usability and practical value**.

7. Reference

References

[1] Francis Patrick Diallo and Cătălin Tudose. Optimizing the scheduling of teaching activities in a faculty. *Appl. Sci.*, 14:9554, 2024.

[2] Nurman Rasyid Panusunan Hutasuhut, Mochamad Gani Amri, and Rizal Fathoni Aji. Security gap in microservices: A systematic literature review. *International Journal of Advanced Computer Science and Applications*, 15(12), 2024.

[3] Jaydeep R. Tadhani, Vipul Vekariya, Vishal Sorathiya, Samah Alshathri, and Walid El-Shafai. Securing web applications against xss and sqli attacks using a novel deep learning approach. *Scientific Reports*, 14(1):1803.

[4] Toni Taipalus. Database management system performance comparisons: A systematic literature review. 2023.

8. Appendix

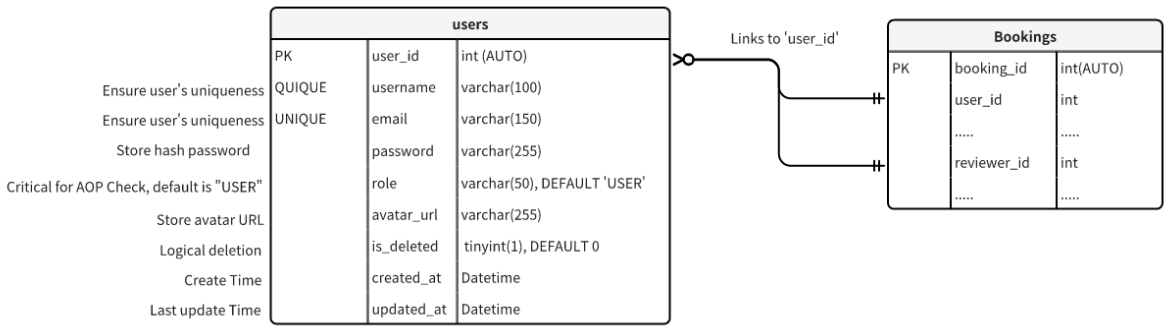

Figure 6: ERD of User Identity and Verification Module

Sprint I: Initial Design and Setup					
Sprint Duration	Start Date: 3rd March 2025 End Date:14th March 2025				
Sprint Theme	Establishing User Identity Foundations & Basic Registration				
Sprint Goal	1. Design "users" database schema and entity. 2. Set up project structure for Authentication module. 3. Implement initial User Registration API (core fields). 4. Implement password hashing (hash + salt).				
	Mon	**Tue**	**Wed**	**Thu**	**Fri**
Week 3	1st Sprint Planning	Group work: construct basement of the program framework and Database structure. Individual PBI 1: Design "User" database schema and entity.			Individual PBI 2: Set up project structure for Auth module
Week 4	Individual PBI 2: Set up project structure for Auth module	Individual PBI 3: Implement initial User Registration API (core field)	Daily Scrum	Individual PBI 3: Implement initial User Registration API (core field)	Individual PBI 4: Implement password hashing (hash + salt).
Week 5	1st Sprint Review	1st Sprint Retrospective			

Figure 7: Sprint 1 Time Planning

Sprint II: Foundational Backend Structures					
Sprint Duration	Start Date: 17th March 2025 End Date:1st April 2025				
Sprint Theme	Enabling Secure Login & Initial Profile Access with Email Verification				
Sprint Goal	1. Implement User Login API. 2. Implement basic session management. 3. Add email verification (OTP(Once Time Password) send & verify) to registration. 4. Design API for fetching user profile data.				
	Mon	**Tue**	**Wed**	**Thu**	**Fri**
Week 5			2nd Sprint Planning	Individual PBI 1: Implement User Login API.	
Week 6	Individual PBI 2: Implement basic session management.		Daily Scrum	Individual PBI 3: Add email verification to registration.	Individual PBI 4: Design API for fetching user profile data.
Week 7	2nd Sprint Review	2nd Sprint Retrospective			

Figure 8: Sprint 2 Time Planning

Sprint III: Core Logic Implementation & API Integration					
Sprint Duration	Start Date:2nd April 2025 End Date:22nd April 2025				
Sprint Theme	Enhancing Profile Management & Introducing API Authorization				
Sprint Goal	1. Implement Update Password functionality. 2. Add a time limit for requesting to send an Email. 3. Implement Update Email Address functionality. 4. Design and implement core AOP authorization aspect (@AuthCheck). 5. Apply AOP to critical user-specific APIs. 6. Apply AOP authentication to other team member's endpoints 7. Document the interfaces I have implemented.				
	Mon	**Tue**	**Wed**	**Thu**	**Fri**
Week 7			3rd Sprint Planning	Individual PBI 1: Implement Update Password functionality.	
Week 8	Individual PBI 2: Add a time limit for requesting to send an Email.		Daily Scrum	Individual PBI 3: Implement Update Email Address functionality.	
Week 9	Individual PBI 4: Design and implement core AOP authorization aspect.	Individual PBI 5: Apply AOP to critical user-specific APIs.	Daily Scrum	Individual PBI 6: Apply AOP authentication to other team member's endpoints	Individual PBI 7: Document the interfaces I have implemented.
Week 10	3rd Sprint Review	3rd Sprint Retrospective			

Figure 9: Sprint 3 Time Planning

Sprint IV: Deployment & Comprehensive Testing					
Sprint Duration	Start Date: 23rd April 2025 End Date:30th April 2025				
Sprint Theme	Module Hardening, Comprehensive Testing & Broader Authorization Integration				
Sprint Goal	1. Conduct unit and integration testing for all module features. 2. Refine error handling and API responses. 3. verify AOP authorization in team members' endpoints. 4. Complete the documentation on Authentication APIs.				
	Mon	**Tue**	**Wed**	**Thu**	**Fri**
Week 10			4th Sprint Planning	Group PBI 1: Conduct unit and integration testing for all module features.	Individual PBI 2: Refine error handling and API response.
Week 11	Individual PBI 3: Verify the AOP Auth function in other team member's endpoints.	4th Sprint Review 4th Sprint Retrospective	Individual PBI 4: Complete the documentation on Authentication APIs.		

Figure 10: Sprint 4 Time Planning

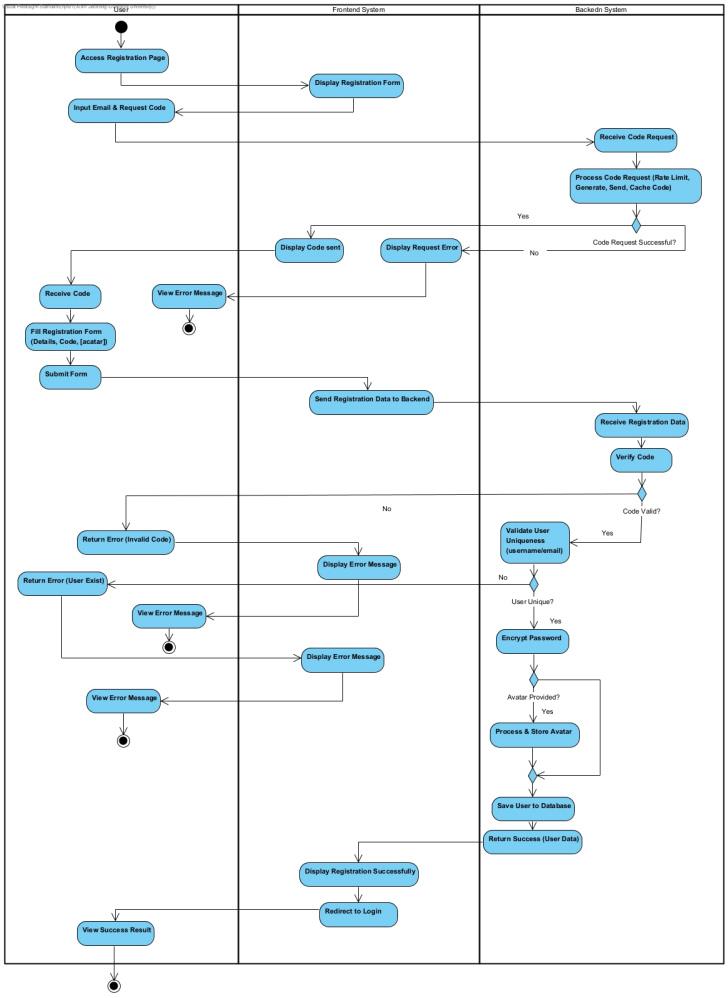

Figure 11: 3.1.1-Activity Diagram

Table 1: Sprint Plan Overview

Sprint Cycle	**My Key PBIs & Sprint Theme**	**Corresponding Group Sprint Phase**	
Sprint 1	1. Design users database schema and entity. 2. Set up project structure for Authentication module. 3. Implement initial User Registration API (core fields). 4. Implement password hashing (hash + salt).	Establishing User Identity Foundations & Basic Registration	Initial Design and Setup
Sprint 2	1. Implement User Login API. 2. Implement basic session management. 3. Add email verification (OTP – Once Time Password – send & verify) to registration. 4. Design API for fetching user profile data.	Enabling Secure Login & Initial Profile Access with Email Verification	Foundational Backend Structures
Sprint 3	1. Implement Update Password functionality. 2. Add a time limit for requesting to send an Email. 3. Implement Update Email Address functionality. 4. Design and implement core AOP authorization aspect (`@AuthCheck`). 5. Apply AOP to critical user-specific APIs. 6. Apply AOP authentication to other team member's endpoints. 7. Document the interfaces I have implemented.	Enhancing Profile Management & Introducing API Authorization	Core Logic Implementation & API Integration
Sprint 4	1. Conduct unit and integration testing for all module features. 2. Refine error handling and API responses. 3. Verify AOP authorization in team members' endpoints. 4. Complete the documentation on Authentication APIs.	"Module Hardening, Comprehensive Testing & Broader Authorization Integration"	Deployment & Comprehensive Testing

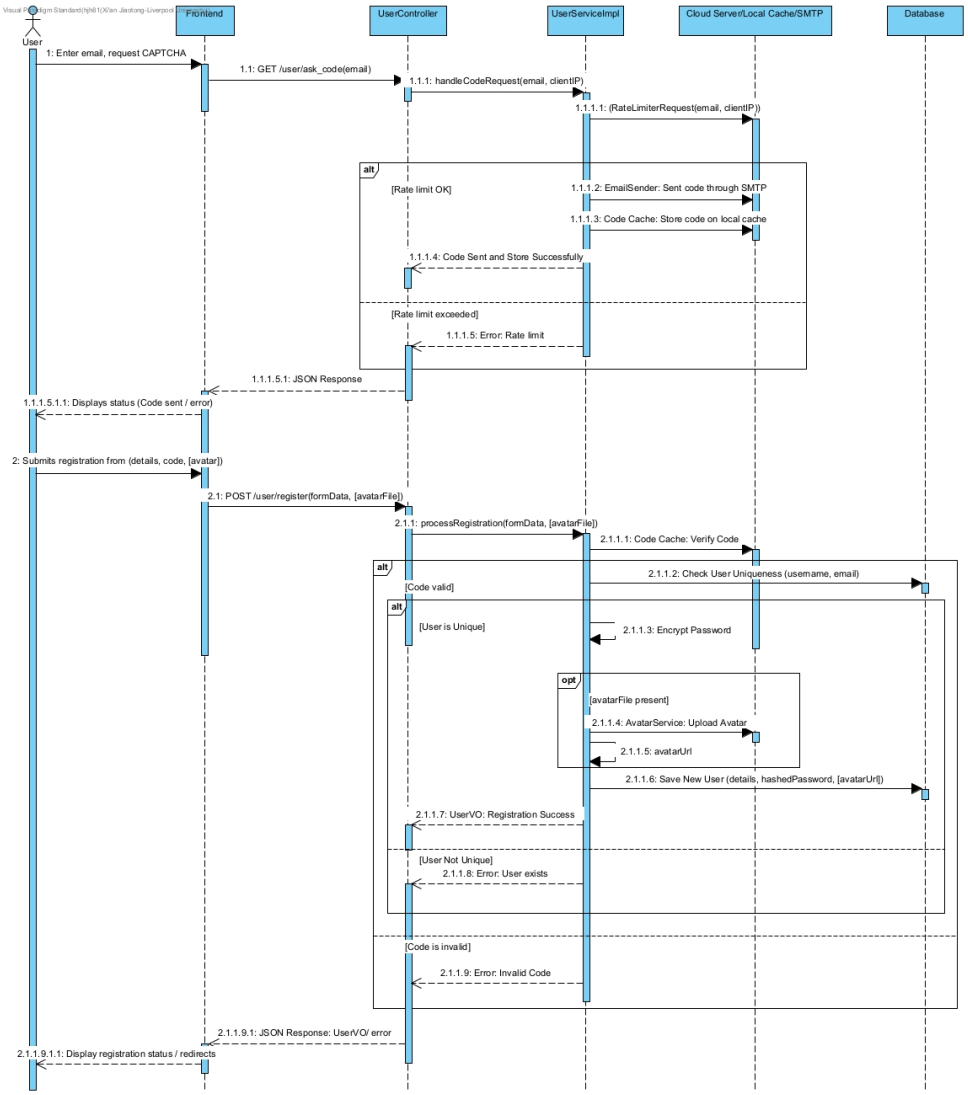

Figure 12: 3.1.1-Sequence Diagram

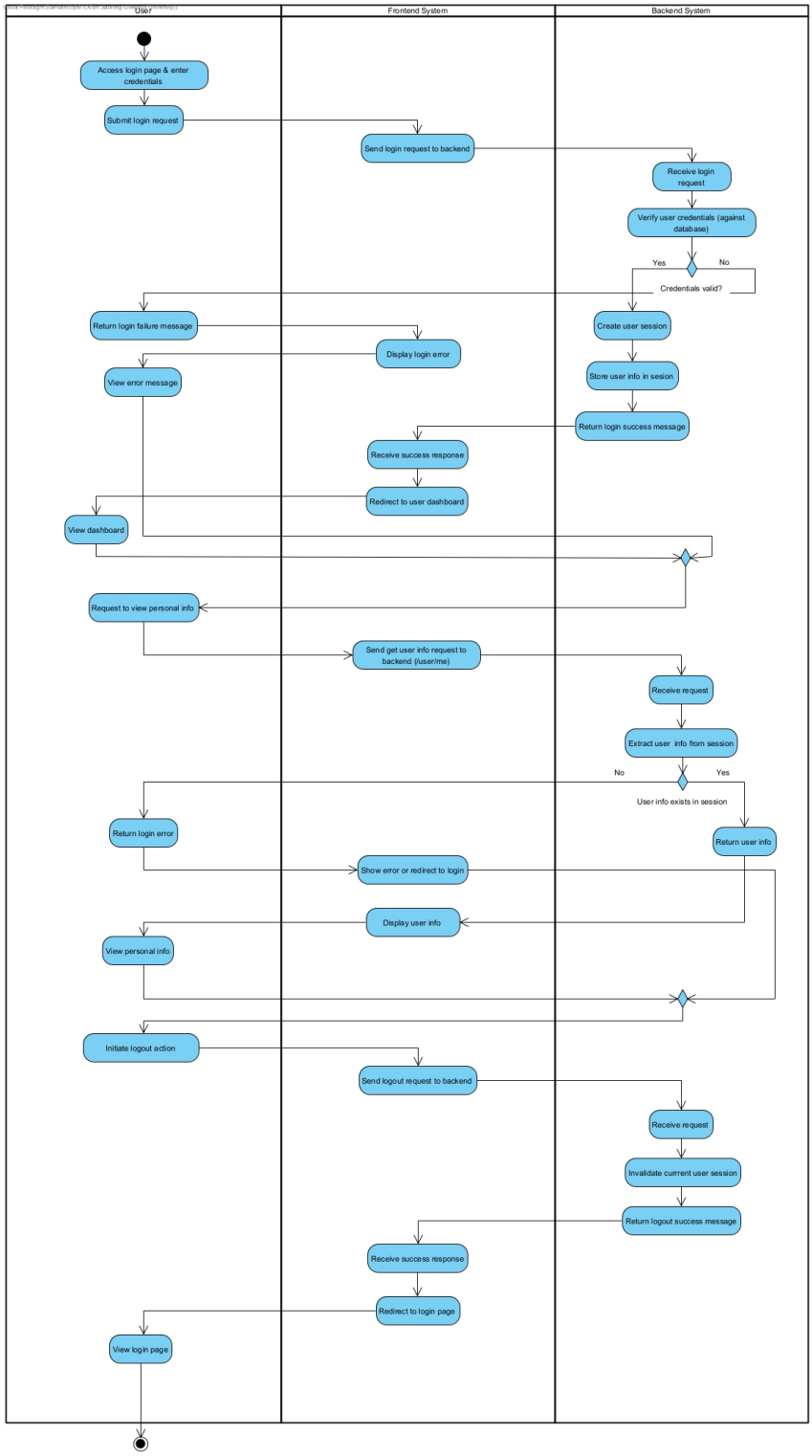

Figure 13: 3.1.2-Activity Diagram

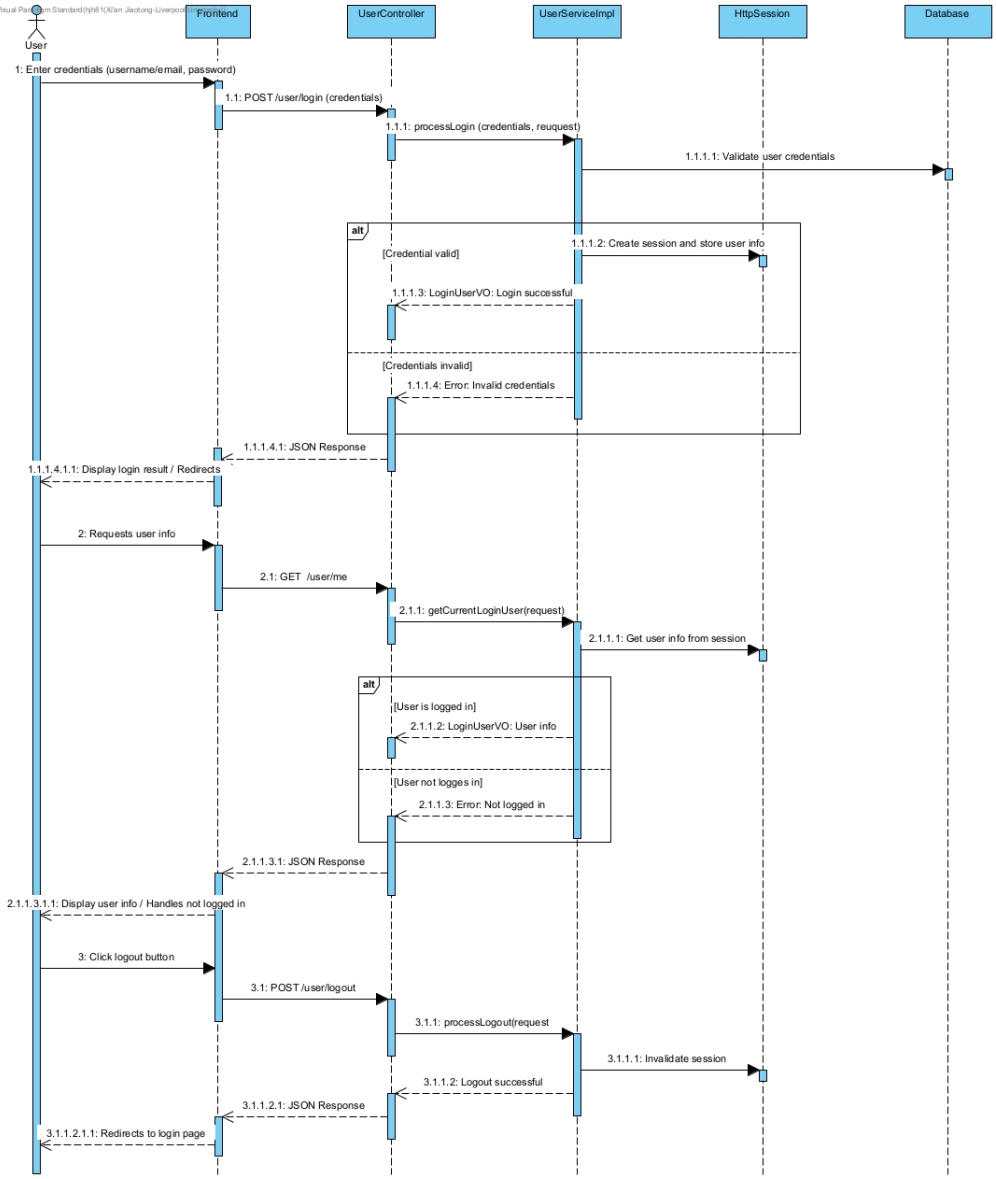

Figure 14: 3.1.2-Sequence Diagram

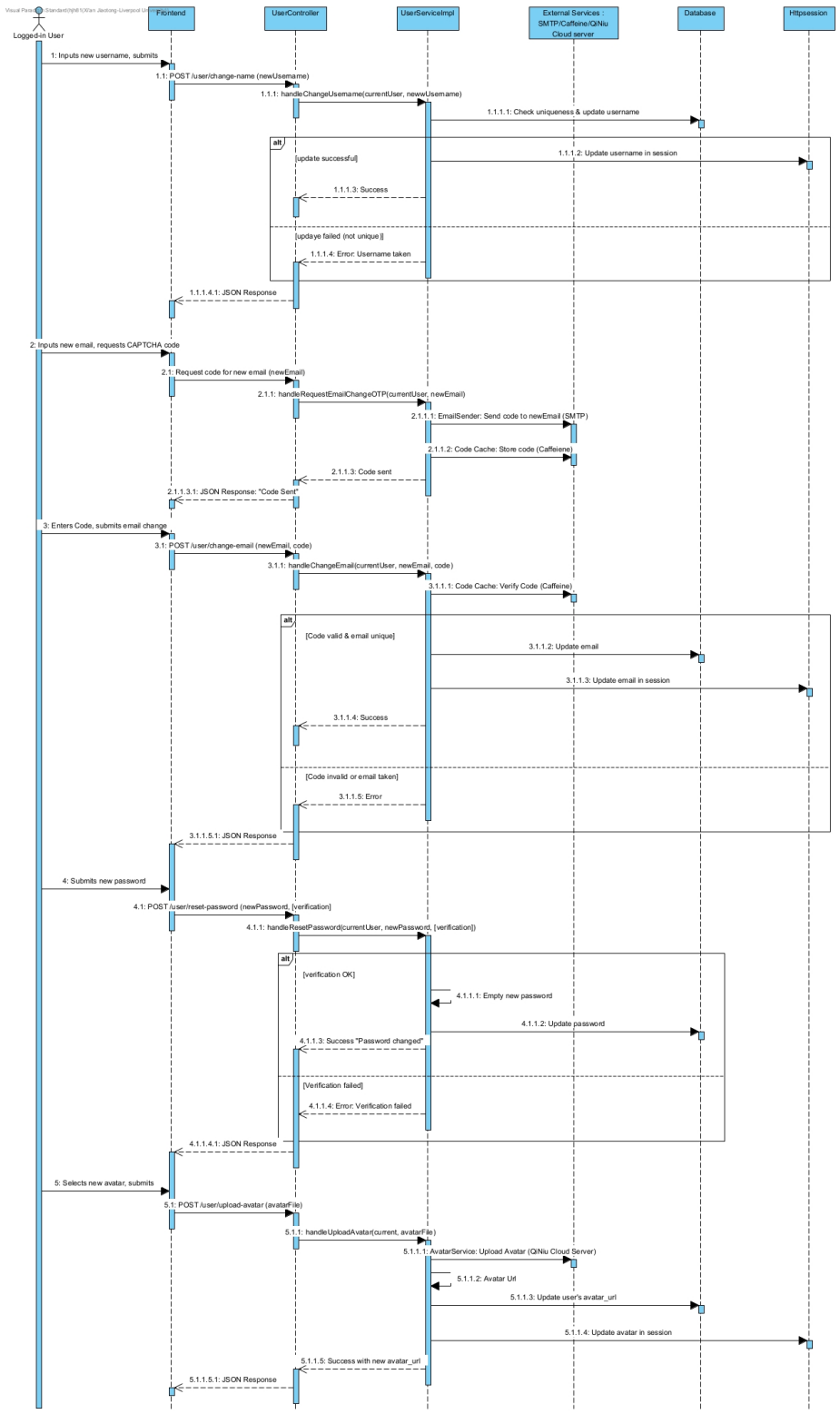

Figure 15: 3.1.3-Sequence Diagram

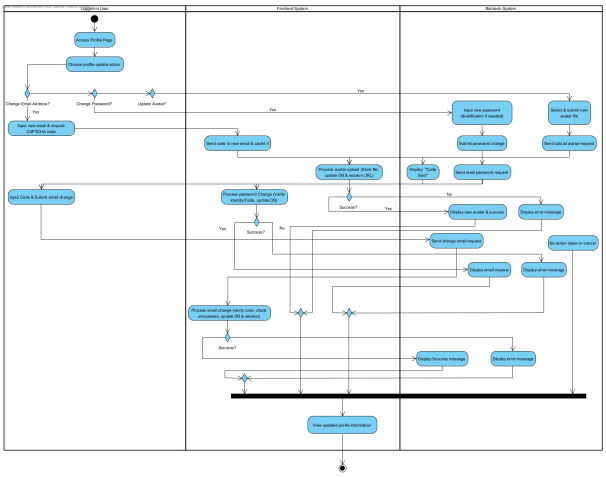

Figure 16: 3.1.3-Activity Diagram

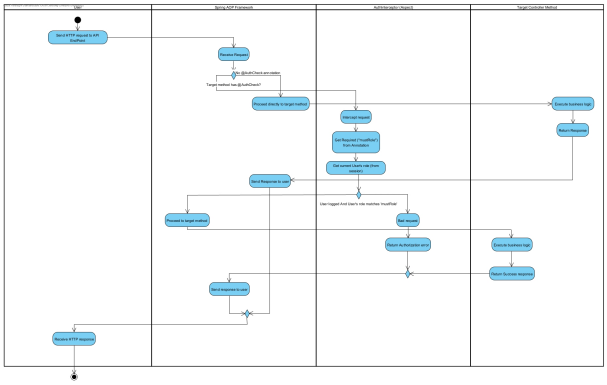

Figure 17: 3.1.4-Activity Diagram

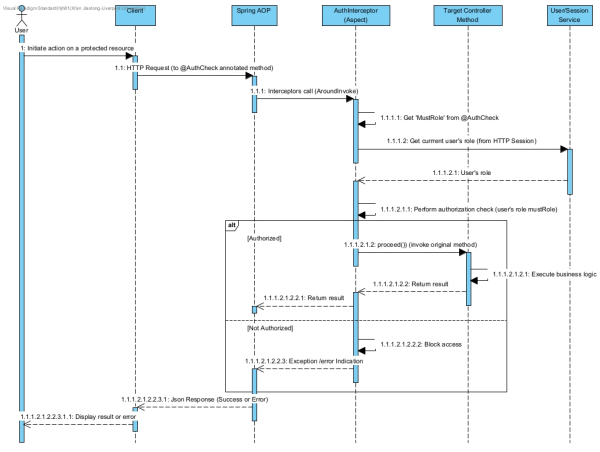

Figure 18: 3.1.4-Sequence Diagram

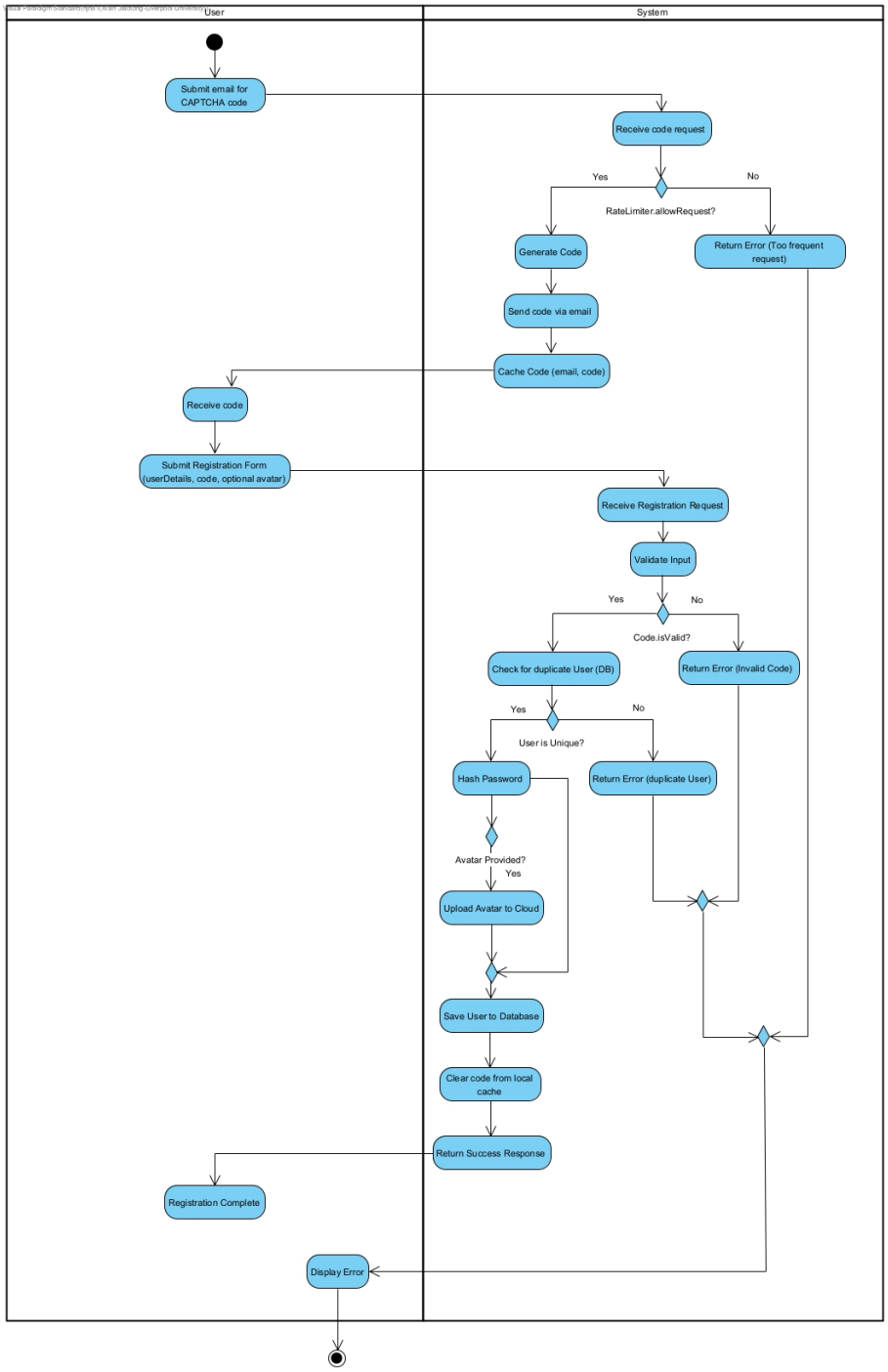

Figure 19: 3.2.1-Activity Diagram

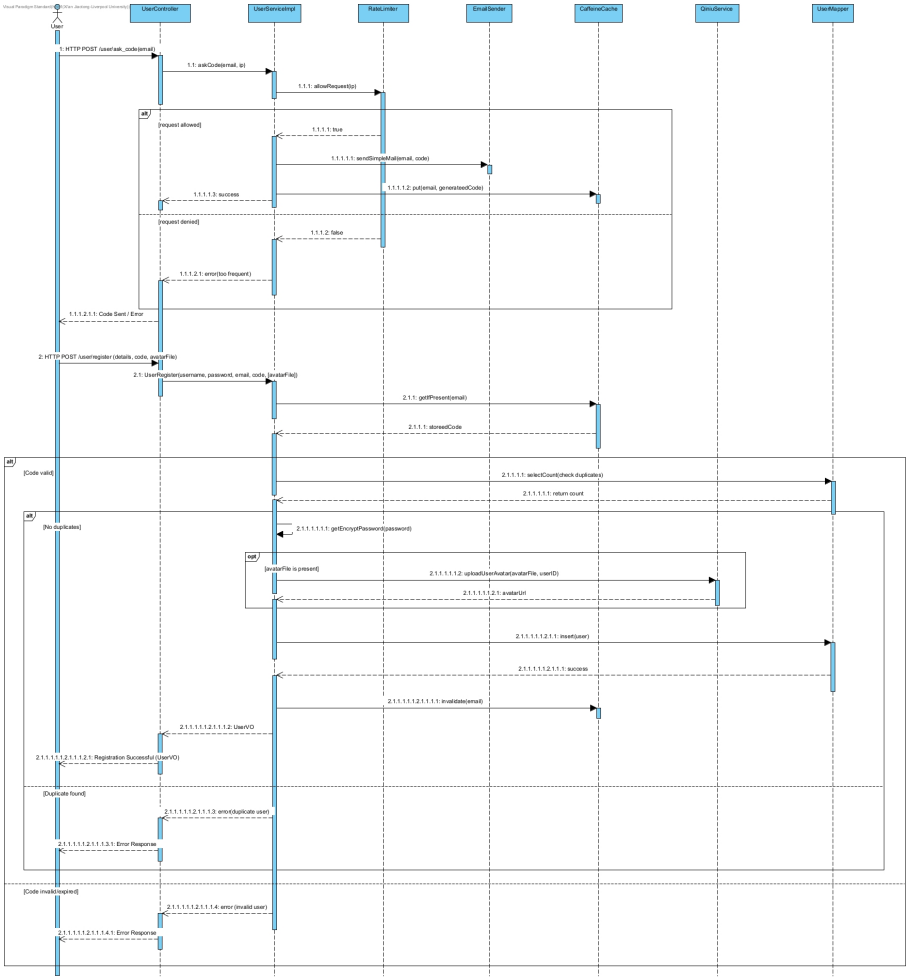

Figure 20: 3.2.1-Sequence Diagram

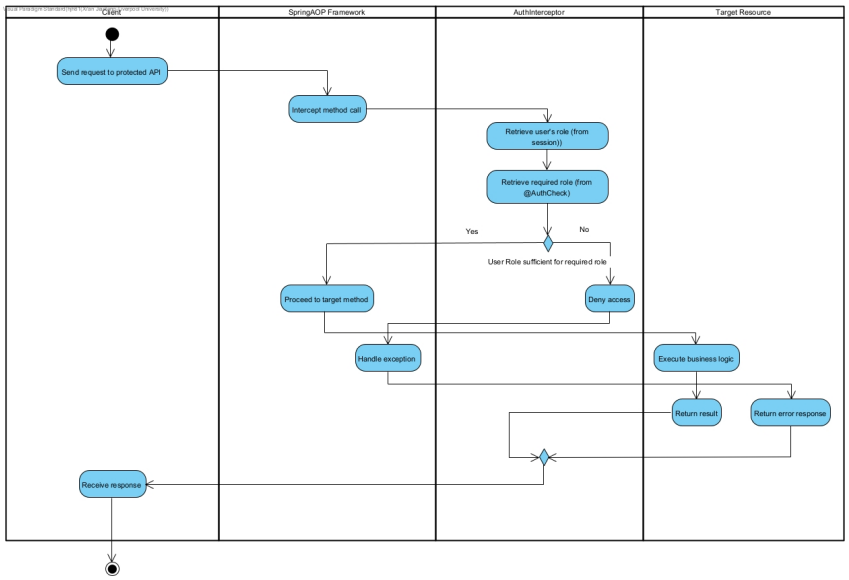

Figure 21: 3.2.2-Activity Diagram

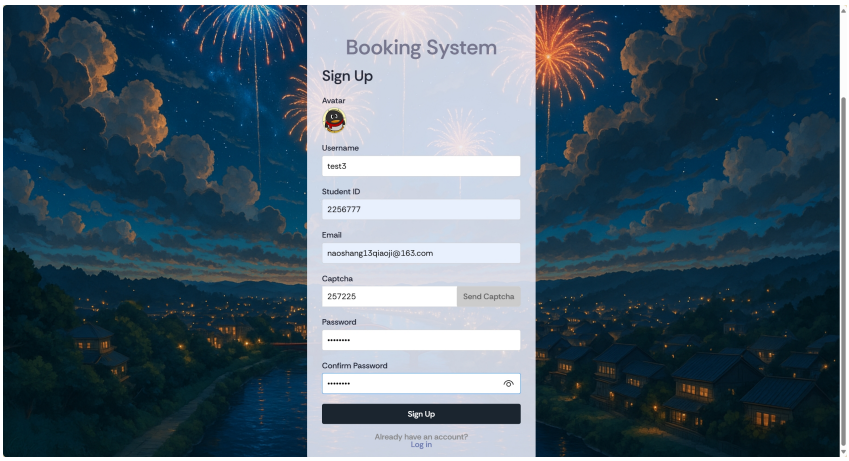

Figure 22: Registration Page

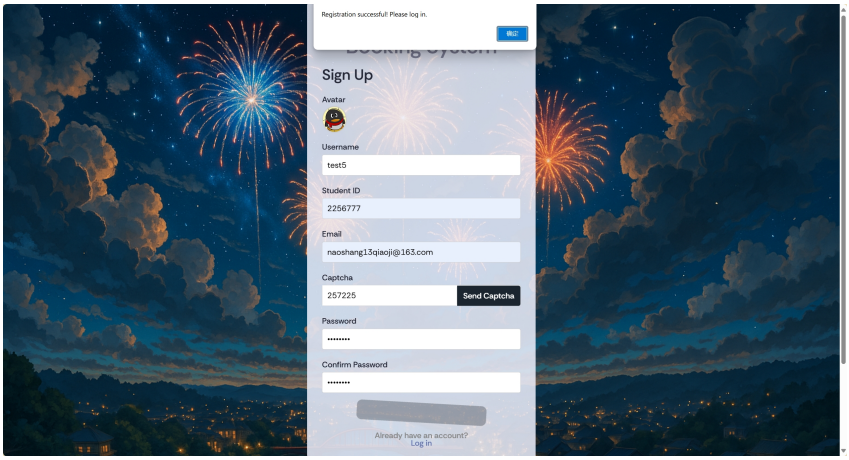

Figure 23: Registration Success

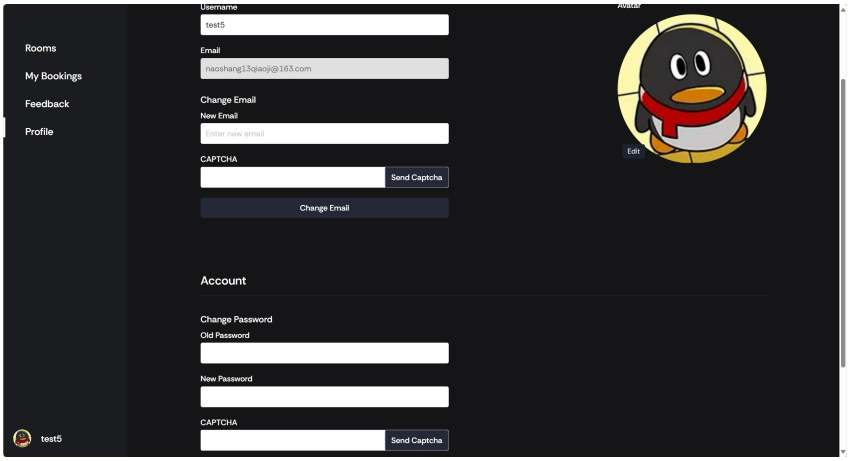

Figure 24: Profile Page

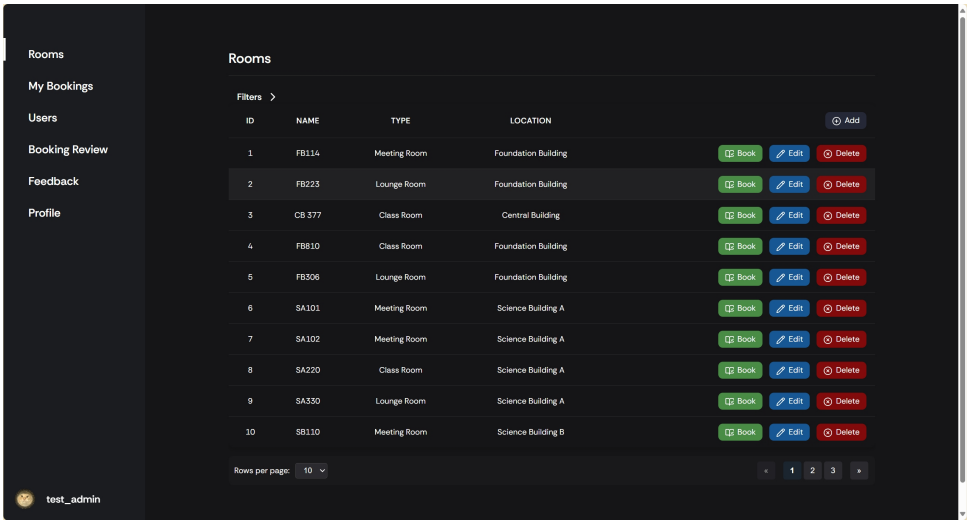

Figure 25: Admin Room Page

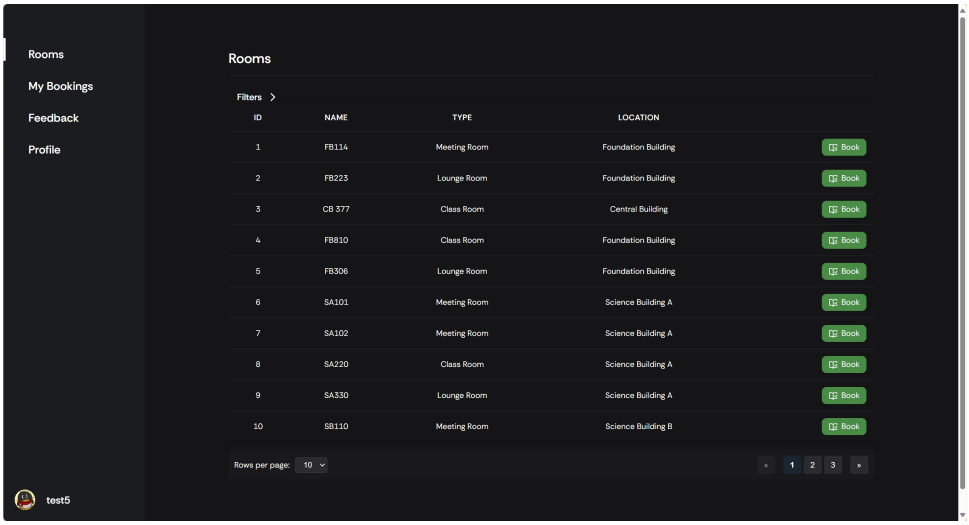

Figure 26: User Room Page

Table 2: Product Backlog Items and Acceptance Criteria of two Detailed PBIs

PBI Name	AC No.	Acceptance Criterion	Given	When	Then
PBI-REG-DETAIL: Secure User Registration with Email Verification	1.	**Successful Registration with Valid Inputs and CAPTCHA**	*Given:* I am an unregistered user on the system's registration page, and I have provided a username that is unique within the system, an email address that is also unique and valid, and a password that satisfies the system's complexity rule of being at least 8 characters long. I have also successfully requested an email CAPTCHA for the provided email address and have received it.	*When:* I accurately enter the received email CAPTCHA into the designated field, optionally upload a valid image file for my avatar, and subsequently click the "Sign Up" button to submit my registration details.	*Then:* My user account should be successfully created with the default role of `'USER'`. If an avatar image was provided during submission, I should be able to see this in my profile. Finally, I should be automatically redirected to the system's login page and presented with a clear success message "Successful registration."
	2.	**Attempted Registration with an Already Registered Email Address**	*Given:* I am an unregistered user on the system's registration page, filling out the required fields to create a new account.	*When:* I enter a email into the Email Address input field that is identical to a email already existing for another registered user in the system, and I subsequently attempt to ask CAPTCHA.	*Then:* The system should prevent the CAPTCHA request and must display a clear, user-friendly error message directly on the registration page, specifically indicating the nature of the problem, such as "Email already registered."
	3.	**Attempted Registration with an Already Registered Username**	*Given:* I am an unregistered user on the system's registration page, filling out the required fields to create a new account.	*When:* I enter a username into the username input field that is identical to a username already existing for another registered user in the system, and I subsequently attempt to submit the registration form.	*Then:* The system must reject my registration attempt and clearly display an appropriate error message on the page, such as "Username already registered" informing me of the conflict.

Continued on next page

Table 2 – continued from previous page

PBI Name	AC No.	Acceptance Criterion	Given	When	Then
	4.	**Attempted Registration with an Incorrect or Expired Email CAPTCHA**	*Given:* I am on the registration page, having filled in my desired username, email, and password, and I have previously received an email CAPTCHA.	*When:* I enter an email CAPTCHA into the verification code field that is either factually incorrect or has expired according to the system's validity period, and I then click the "Sign Up" button.	*Then:* An explicit error message, such as "Verification code error," must be displayed prominently on the registration page, and the system should not create a new user account based on this submission.
	5.	**Attempted Registration with a Password Shorter Than 8 Characters**	*Given:* I am a prospective new user on the registration page, intending to create an account.	*When:* I enter a desired password into the password input field, but the chosen password contains fewer than the system's required minimum of 8 characters.	*Then:* The system must prevent the registration from proceeding and should display a specific error message, such as "The password must contain at least 8 digits," directly associated with the password input field to guide me in correcting the input.
PBI-AOP-DETAIL: Constructing an AOP-based Authorization Mechanism for API Endpoints	1.	**Administrator Accessing an Endpoint Protected for Administrators Only**	*Given:* I am an authenticated user who is currently logged into the system, and my account possesses the designated `'ADMIN'` role.	*When:* I initiate an HTTP request targeting a specific API endpoint that is explicitly protected for `'ADMIN'` role access.	*Then:* The system must correctly identify my `'ADMIN'` role, determine that it satisfies the access requirement, and consequently permit my request to proceed to the target functionality.

Continued on next page

Table 2 – continued from previous page

PBI Name	AC No.	Acceptance Criterion	Given	When	Then
	2.	**Non-Administrator (User Role) Attempting to Access an Endpoint Protected for Administrators Only**	*Given:* I am an authenticated user currently logged into the system, but my account is assigned the standard `'USER'` role, not the `'ADMIN'` role.	*When:* I attempt to send an HTTP request to an API endpoint that is secured and designated for administrators only.	*Then:* The system must intercept my request, identify my `'USER'` role, recognize that it does not meet the `"ADMIN"` requirement, and consequently deny access. As a result, I should receive a clear error response from the system (e.g. "No permission").
	3.	**Unauthenticated User Attempting to Access any Protected Endpoint**	*Given:* I am a user who is not currently logged into the system.	*When:* I attempt to send an HTTP request to any API endpoint that requires authentication for access, regardless of the specific role needed.	*Then:* The system should correctly identify that I am not logged in and must deny my request. I should receive an error response from the system clearly indicating that login is required to access the requested resource.