Report

Last edited by **Duy Dinh Thai** 8 hours ago

ONLINE LIBRARY MANAGEMENT - TEAM 1

Table of Contents

I. Introduction

The online library management system project aims to streamline library operations, improve the user experience, and enhance access to library resources. It automates tasks such as book cataloging, tracking, and borrowing while offering user-friendly interfaces for staff and patrons. The system increases operational efficiency, improves accessibility, and provides a centralized platform for managing library resources.

I.1. Project Description

The current challenges faced by the library include manual book tracking, limited accessibility, and outdated systems. Manual book tracking processes are time-consuming and prone to errors, leading to difficulties in managing the library's collection.What is more, limited accessibility refers to the challenges faced by users in accessing library resources remotely or outside of library operating hours. Outdated systems hinder efficient operations and may lack modern features that enhance the user experience.

The online library management system will address these challenges by:

- Automating book tracking processes. It will provide a centralized database where staff can easily catalog books, update information, and track borrowing and returning activities. This automation minimizes errors, saves time, and streamlines the overall management of the library's collection.
- In terms of limited accessibility, the online system will offer web interfaces that allow users to search for books, view availability, and place online reservations from anywhere, at any time. It eliminates the need for physical visits to the library and provides convenient access to resources remotely. Additionally, the system can include a digital repository for e-books, journals, and other digital materials, further expanding accessibility to a wider range of resources.
- Regarding outdated systems, the online library management system will provide a modern and efficient solution. It will leverage current technologies, programming languages, and frameworks to offer a user-friendly and intuitive interface for both staff and users. The system will incorporate features such as advanced search capabilities, personalized recommendations, and automated processes for book checkout and return. This modernization enhances the user experience, improves efficiency, and keeps the library up-to-date with technological advancements.

I.2. Goal and Objective

The goals and objectives of the online library management system project are as follows:

- **Automating Book Cataloging:** The system aims to automate the process of book cataloging, including adding new books to the library's collection, updating book information, and maintaining an accurate and up-to-date catalog.
- **Enabling Online Book Reservations:** The system will provide patrons with the ability to place online reservations for books, allowing them to conveniently reserve desired items and manage their borrowing activities.
- **Facilitating User-Friendly Search and Checkout Processes:** The system will offer user-friendly interfaces and functionalities for patrons to search for books using various criteria, such as title, author, or genre. Additionally, it will streamline the checkout process, making it easy and efficient for patrons to borrow books.

The success criteria for the project may include:

- **Increased Book Circulation:** The project's success can be measured by an increase in the number of books borrowed from the library. This indicates that the system has facilitated a smoother borrowing process and improved access to library resources.
- **Reduced Administrative Workload:** The system should significantly reduce the manual administrative tasks involved in book cataloging, tracking, and managing borrowing activities. Success can be measured by a decrease in the time and effort required for these tasks.
- **Improved User Satisfaction:** User satisfaction is a crucial measure of success. The system should enhance the overall user experience by offering easy book search options, efficient checkout processes, and convenient access to library resources. A positive user feedback and increased patron engagement can indicate improved user satisfaction.

II. Project Management

II.1. Members and Scope

TEAM 1			
No.	**Name**	**Role**	**Email**
1	Đinh Thái Duy	**Leader**	10421014@student.vgu.edu.vn
2	Nguyễn Minh Thuận	**Full-stack**	10421057@student.vgu.edu.vn
3	Đoàn Quang Tiến	**Front-End; UI/UX Design**	10421059@student.vgu.edu.vn
4	Mai Nguyễn Việt Phú	**Database Administrator; Full-stack**	10421047@student.vgu.edu.vn
5	Trần Duy Khang	**DevOps; Security; Back-end**	10421025@student.vgu.edu.vn
6	Nguyễn Minh Đức	**Back-end**	10421010@student.vgu.edu.vn
7	Lê Trọng Nhân	**Front-end; UI/UX Design**	10421044@student.vgu.edu.vn
8	Đỗ Đức Toàn	**Database Administrator + Back-end**	10421060@student.vgu.edu.vn

II.2. Project Manager Method

Our team will have a weekly meeting with an approach to Agile Methodology and Sprint Planning. Each week we will have a clear action plan, what we need to do in this week, specific for each member in our team. The detailed process for each week can be found here:

- Review all tasks of the previous week, which is done and which is pending. Rearrange the role (if any), small reflection between members and their feedback to the overall plan. Leader will take a look with the timeline of each stage and ensure every work is under control..
- Define the goal and what we need to achieve in the following week
- Leader will deliver the task for member (the task will be discussed between members and the leader).
- During the weeks, members will do their own task, send updates to the leader and their teammate.

Our team will manage our project using Git environment and Google Workspace. The meeting will be fowed with the following concept:

- Review the tasks.
- Members represent their work during the week.
- Leader with team will define the goal of the following week.
- Leader/Members will represent new concept, knowledge (if any related to the whole project).
- Everyone gives their own thought, unclear task,... and team will discuss.

II.3. Project Timeline

GRIEFFINGDOOR GRANTT CHART TIMELINE

Stage	Name	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14	Week 15
1	Project Idea Finalizing - Analyze Requirement and Project Proposal															
2	Software Analysis and Task Distribution															
3	Front-end Design															
4	Database Design															
5	Back-end Implementation & Testing															
6	Front-end Implementation & Testing															
7	Security and Privacy															
8	Upgrading Requirement version 2															
9	Upgrading Analysis and Design version 2															
10	Upgrading Code version 2															
11	Testing and Finalize Everything															
12	Report															
13	Presentation															

III. Software Design and Analysis

III.1. Software Architecture and Technology Base

The project follows the Model-View-Controller (MVC) architectural pattern, which is a widely adopted design pattern for building web applications. The MVC pattern separates the application logic into three interconnected components: the Model, the View, and the Controller.

- Model: The Model component represents the data and the business logic of the application. It is responsible for managing the data, performing data-related operations, and implementing the application's core functionality. In the context of your project, the Model component handles tasks such as interacting with the database, performing CRUD (Create, Read, Update, Delete) operations, and enforcing business rules.

- View: The View component is responsible for the user interface and the presentation layer of the application. It receives data from the Model and presents it to the user in a visually appealing and intuitive manner. The View component also handles user interactions, such as capturing input and responding to user actions. In your project, the View component is responsible for rendering the user interface, handling user inputs, and updating the display based on the data provided by the Model.

- Controller: The Controller component acts as an intermediary between the Model and the View. It receives user inputs from the View, processes the data, and communicates with the Model to perform the necessary operations. The Controller is responsible for interpreting the user's actions, validating the input, and coordinating the flow of data between the Model and the View. In your project, the Controller component handles tasks such as processing user requests, validating input, and orchestrating the interactions between the Model and the View.

III.2.Table of Requirements

ID	Requirement	Type	Actor	Priority	Dependencies	Acceptance Criteria
REG001	Users can create account to use the system	Functional		High		- Users register with username, password (complex), email and optional name. - Username must be unique.
LOG001	Users use the account to login	Functional		High	REG001	- Users log in with username and password. - System validates credentials and directs to the dashboard on success, or displays an error message.

ID	Requirement	Type	Actor	Priority	Dependencies	Acceptance Criteria
PSU001	Users must setup their public profile.	Functional		High	LOG001	- Users edit profile information (name, email, phone, notification). - Changes are saved and reflected in their profile. - Links to other social platforms such as Facebook, Instagram, TikTok... - Display read and/or recently read books. - Show user's reviews. - Show which book clubs the user joined.
BRW001	Users can browse books from the library	Functional	User, Admin	High	LOG001	- Users browse a list of available books with details: (Title, author, category, publication date, availability). - Clicking a book might lead to a detailed page.
OLM002	Users can filter books based on category, author, publication, etc	Functional	User	High	OLM001	Filter for book searching: - Search by title: Provide a text input field where users can enter the title of a book. - Author: Users can enter the name of an author in a text input field. - Genre: Filter a book by choosing genres: Fantasy, Historical fiction, Mystery, Science fiction, Thriller, Horror, Romance, Young adult, Adventure, Short story, Dystopian, Graphic novel, Contemporary, Memoir, Women's fiction, Biography, LGBTQ, Magical Realism, Western fiction, Fairy tale, Paranormal. - Publication year: Include two input fields where users can enter the start and end years, and the search results will display books published within that range. - Availability: Add an option to filter books based on their availability. Users can choose to display only available books or include books that are currently checked out or on hold. - Sorting: Users can choose to sort by relevance, title, author, publication year, or any other relevant criteria. Include ascending and descending order options for each sorting criterion. - Format: hardcover, paperback, e-book, ... - Language
BID001	Users can preview book detailed	Functional				
PAR001	Users can rent books from the library for a certain period of time	Functional		High	BRO001	Users should check their accounts before paying/renting books. After selecting a book, users can specify the desired rental period. Clear information about the rental duration, including the start and end dates, should be displayed for user confirmation. Accepted payment methods should be communicated to users. The payment process should confirm successful payment and issue a receipt. After the payment is completed, users receive a confirmation with a summary of the transaction details and get the book. It should remind the user the deadline to return the book via mail or SMS. If the book is overdue, the user gets a fine.
LAR001	Users can rate books (not compulsory)	Functional		Low	BRO001	- User rate book from 1 star to 5 stars. - User can write comments on the book. - Other users can see their reviews.

ID	Requirement	Type	Actor	Priority	Dependencies	Acceptance Criteria
LAM001	Admins can view books' details	Functional		High	REG001, LOG001	List of Books: - The admin interface should display a list of all books currently in the system. - Each book entry should include relevant information such as title, author(s), genre, ISBN, publication date, quantity available, etc. Permissions and Security: - Only users with administrative privileges should have access to this functionality. - Access should be secured with proper authentication and authorization mechanisms.
LAM001	Admins can edit books' details	Functional		High	REG001, LOG001	Edit Book Information: - Admins should be able to edit book details such as title, author, ISBN, summary, etc. - Changes made should be reflected accurately in the system and should be saved persistently. Permissions and Security: - Only users with
REG001	Users can create account to use the system	Functional		High		- Users register with username, password (complex), email and optional name. - Username must be unique.
LOG001	Users use the account to login	Functional		High	REG001	- Users log in with username and password. - System validates credentials and directs to the dashboard on success, or displays an error message.
PSU001	Users must setup their public profile.	Functional		High	LOG001	- Users edit profile information (name, email, phone, notification). - Changes are saved and reflected in their profile. - Links to other social platforms such as Facebook, Instagram, TikTok... - Display read and/or recently read books. - Show user's reviews. - Show which book clubs the user joined.
BRW001	Users can browse books from the library	Functional	User, Admin	High	LOG001	- Users browse a list of available books with details: (Title, author, category, publication date, availability). - Clicking a book might lead to a detailed page.
OLM002	Users can filter books based on category, author, publication, etc	Functional	User	High	OLM001	Filter for book searching: - Search by title: Provide a text input field where users can enter the title of a book. - Author: Users can enter the name of an author in a text input field. - Genre: Filter a book by choosing genres: Fantasy, Historical fiction, Mystery, Science fiction, Thriller, Horror, Romance, Young adult, Adventure, Short story, Dystopian, Graphic novel, Contemporary, Memoir, Women's fiction, Biography, LGBTQ, Magical Realism, Western fiction, Fairy tale, Paranormal. - Publication year: Include two input fields where users can enter the start and end years, and the search results will display books published within that range. - Availability: Add an option to filter books based on their availability. Users can choose to display only available books or include books that are currently checked out or on hold. - Sorting: Users can choose to sort by relevance, title, author, publication year, or any other relevant criteria. Include ascending and descending order options for each sorting criterion. - Format: hardcover, paperback, e-book, ... - Language
BID001	Users can preview book detailed	Functional				

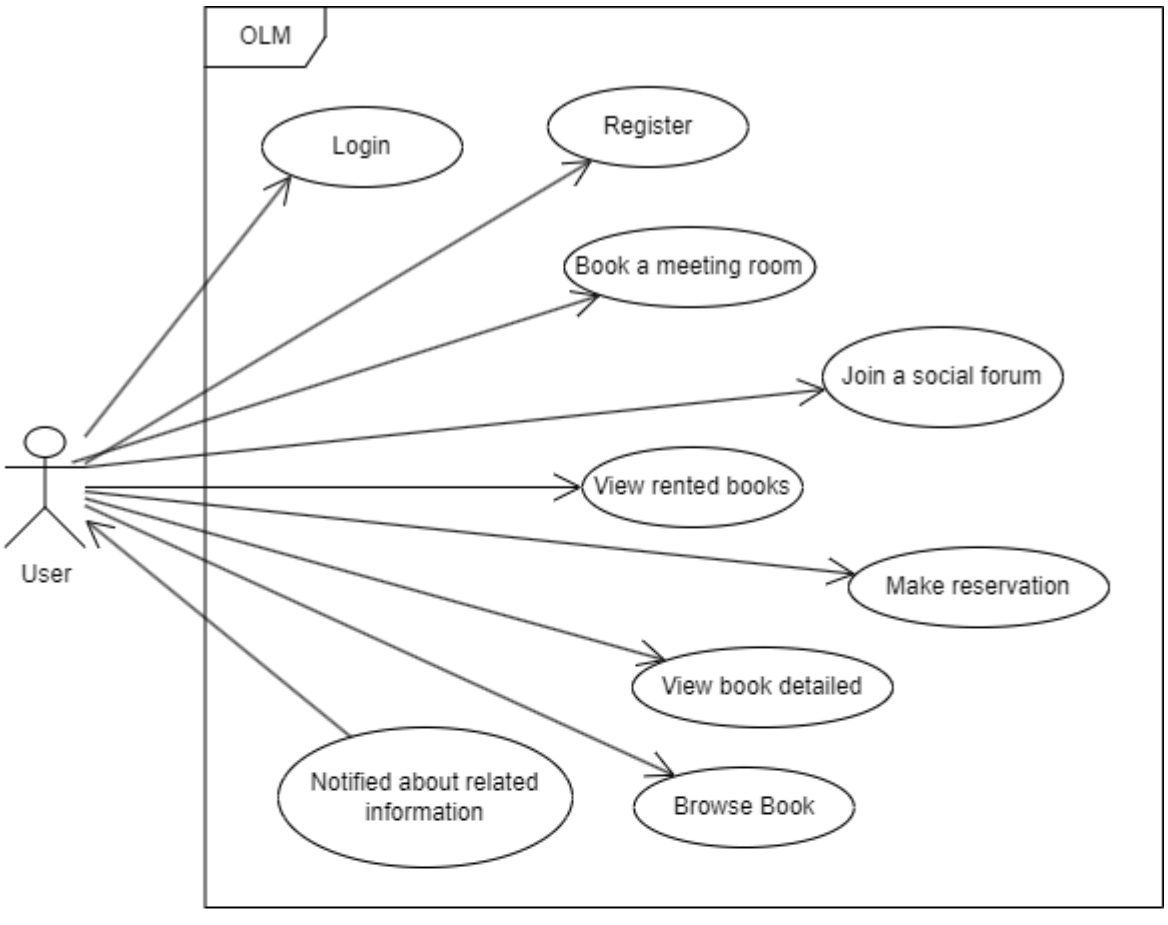
ID	Requirement	Type	Actor	Priority	Dependencies	Acceptance Criteria
PAR001	Users can rent books from the library for a certain period of time	Functional		High	BRO001	Users should check their accounts before paying/renting books. After selecting a book, users can specify the desired rental period. Clear information about the rental duration, including the start and end dates, should be displayed for user confirmation. Accepted payment methods should be communicated to users. The payment process should confirm successful payment and issue a receipt. After the payment is completed, users receive a confirmation with a summary of the transaction details and get the book. It should remind the user the deadline to return the book via mail or SMS. If the book is overdue, the user gets a fine.
LAR001	Users can rate books (not compulsory)	Functional		Low	BRO001	- User rate book from 1 star to 5 stars. - User can write comments on the book. - Other users can see their reviews.
LAM001	Admins can view books' details	Functional		High	REG001, LOG001	List of Books: - The admin interface should display a list of all books currently in the system. - Each book entry should include relevant information such as title, author(s), genre, ISBN, publication date, quantity available, etc. Permissions and Security: - Only users with administrative privileges should have access to this functionality. - Access should be secured with proper authentication and authorization mechanisms.
LAM001	Admins can edit books' details	Functional		High	REG001, LOG001	Edit Book Information: - Admins should be able to edit book details such as title, author, ISBN, summary, etc. - Changes made should be reflected accurately in the system and should be saved persistently. Permissions and Security: - Only users with administrative
LAM001	Admins can add new books	Functional	Admin	High	REG001, LOG001	**Add New Books: **- Admins should be able to add new books to the system, providing all necessary information such as title, author, ISBN, genre, quantity, etc.** **- Newly added books should appear in the list immediately and be searchable.** Permissions and Security:** - Only users with administrative privileges should have access to this functionality. - Access should be secured with proper authentication and authorization mechanisms.
LAM002	Admins can delete books	Functional	Admin	High	REG001, LOG001	**Delete Books: **- Admins should be able to remove books from the system.** **- Deleting a book should prompt for confirmation to prevent accidental deletions.** Permissions and Security:** - Only users with administrative privileges should have access to this functionality. - Access should be secured with proper authentication and authorization mechanisms.

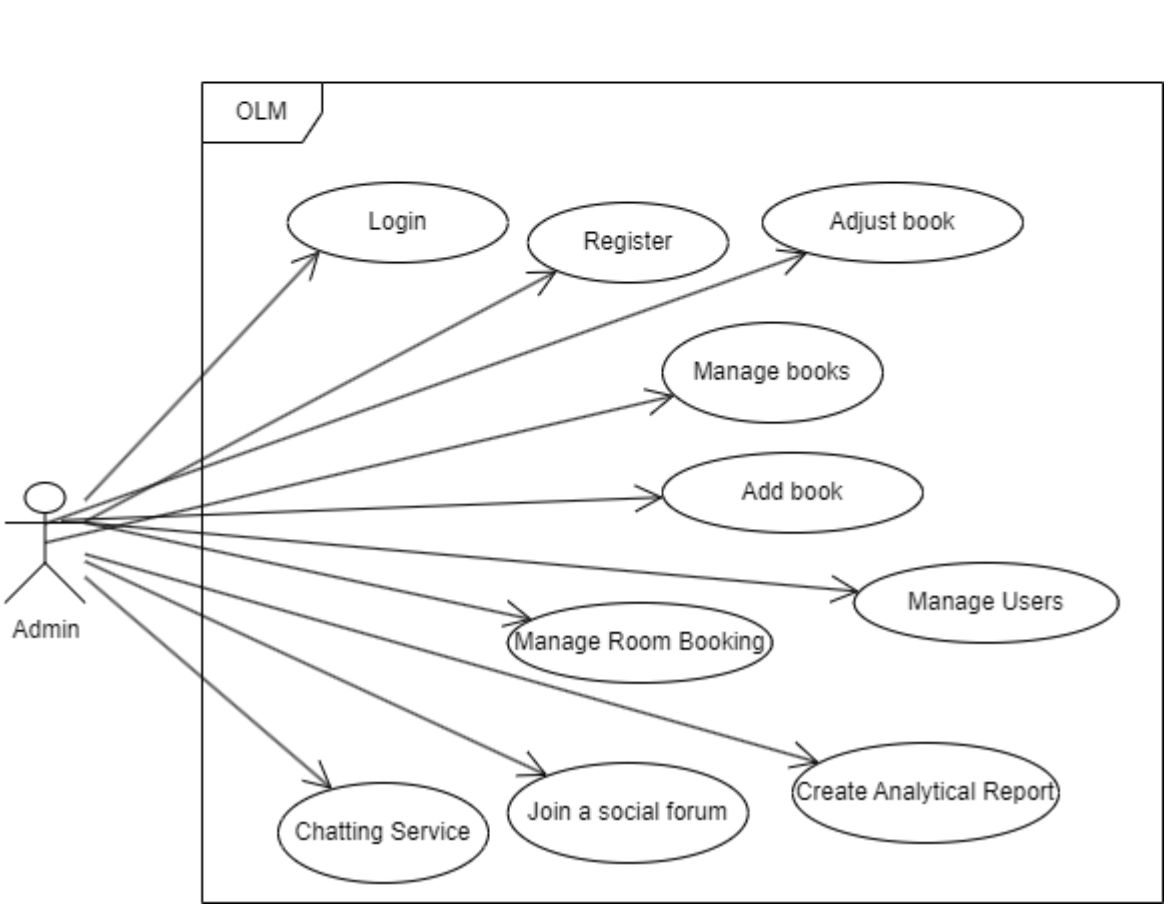
ID	Requirement	Type	Actor	Priority	Dependencies	Acceptance Criteria
LAM001	Admins can search books based on their details	Functional	Admin	High	REG001, LOG001	**Search Functionality: **- Admins should be able to search for books by title, author, genre, ISBN, or any other relevant criteria.** **- Search results should be displayed promptly and accurately.** Permissions and Security:** - Only users with administrative privileges should have access to this functionality. - Access should be secured with proper authentication and authorization mechanisms.
SEP001	The system must provide admins with the capability to separate books based on categories, authors, publications, and other relevant criteria.	Functional	Admin	High	REG001, LOG001	**Category Separation: **- Admins can view, add, edit, and delete book categories.** **- Books can be assigned to multiple categories.** **- Filtering books by category is supported.** Author Separation: **- Admins can view, add, edit, and delete author information.** **- Books can be associated with multiple authors.** **- Filtering books by author is supported.** Publication Separation:** - Admins can view, add, edit, and delete publication details. - Books can be linked to multiple publications. - Filtering books by publication is supported.
TRA001	Admin can view a list of all rented books	Functional	Admin	High		A list of all rented books is displayed with details such as book ID, book title, borrower, and due date
TRA002	Admin can check the availability status of a book	Functional	Admin	High	TRA001	When a book title is entered, the system shows whether the book is available or rented
TRA003	Admin can update the status of a book when it is returned	Functional	Admin	High	TRA001	When a book is returned, the admin can update its status to available
TRA004	Admin receives a notification when a book is overdue	Functional	Admin	Medium	TRA001, TRA002	The system sends a notification to the admin when a book is not returned by the due date
SEN001	System sends notifications automatically via email to users once their lease expires	Functional		High	TRA004	When a book lease expires, the system automatically sends an email notification to the user. The admin can also manually trigger these notifications.
AAR001	Admins can analyze overall renting/like/rating/best choice to write a report	Functional	Admin	High	TRA001, TRA002, LAM001, LAR001	Admin can manage and view all data about book rentals, book reviews, and most selected books. All of the data are organized in an easy-to-access and understand way. Then analyze and evaluate to complete weekly/monthly/yearly reports.
SF002	The application has a section to display user recommendations for the week	Functional		Medium	SF001	Display top-rated books of the previous week on the page. Each recommendation must have a score and number of reviews. Users can see that book by clicking into it.

ID	Requirement	Type	Actor	Priority	Dependencies	Acceptance Criteria
SF003	Users can talk to librarian	Functional		Low		Users can ask for availability of books or problems related to the library by contacting the corresponding staff.
BF001	Users can share their reading, thoughts, and comments publicly on a forum for friends and other users to see and discuss (mimicking Reddit upvote and downvote)	Functional		Medium	SF001	Users can publish their thoughts on a segment of the application. Viewers can comment or if they think something contributes to the conversation, upvote it. Users can easily access the forum.
BF002	Create book clubs with other members	Functional		Low		Users can create, invite, and ask to join book clubs on the website. The book club should have their community chat for members.
IEP001	E-book Platform Integration	Functional		High		The application can successfully integrate with e-book platforms.
IEP002	E-book Catalog Management	Functional		High		Library administrators can add, edit, and remove e-books from the catalog.
IEP003	E-book Search and Discovery	Functional		High		Users can search for e-books based on title, author, genre, or other metadata.
IEP004	E-book Reading Interface	Functional		High		Users can read e-books within the application and utilize features like bookmarking, highlighting, and note-taking.

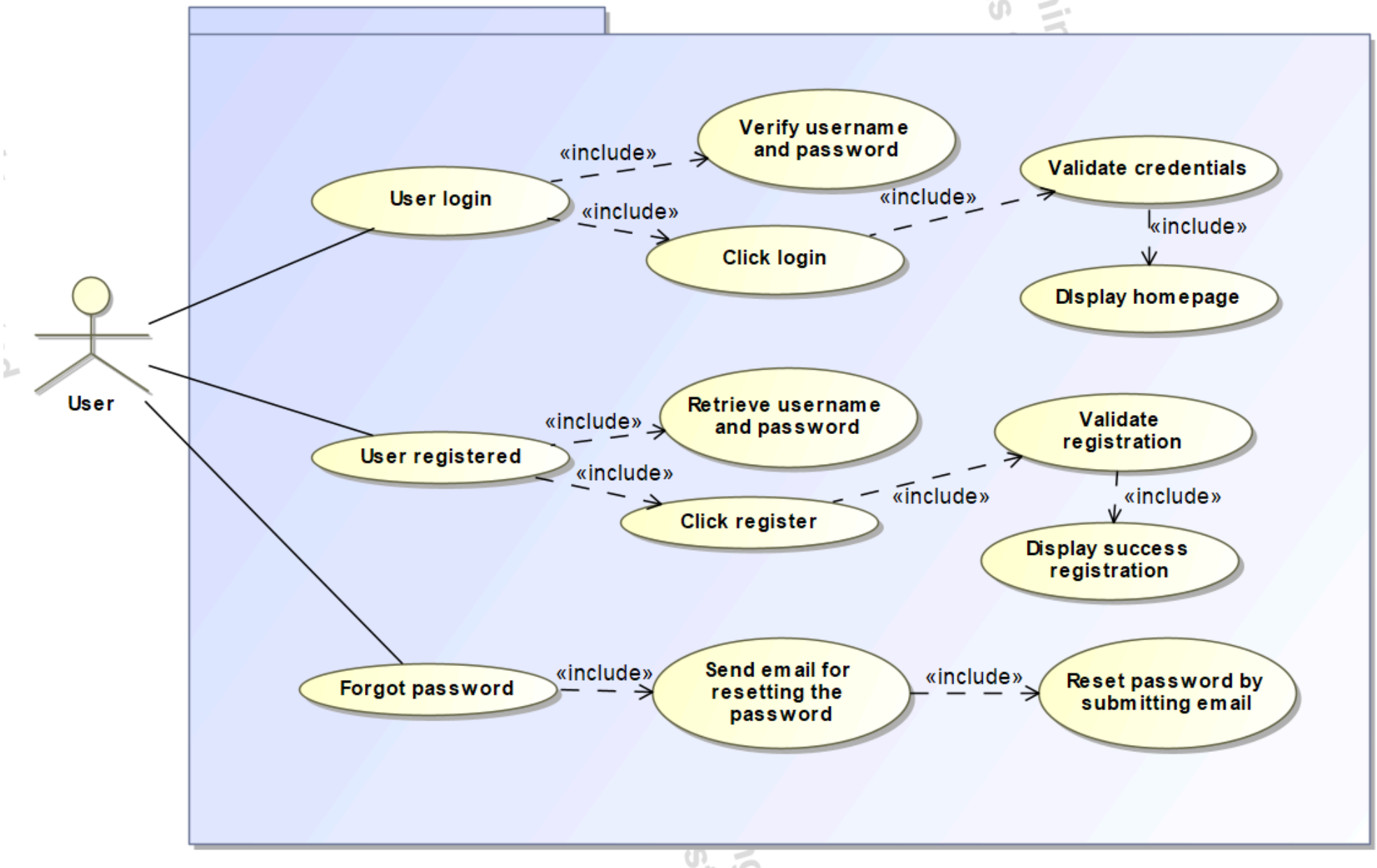
III.3. Requirement Analysis - Use Case

Use Case Diagram 1: Overall System - Use and Admin





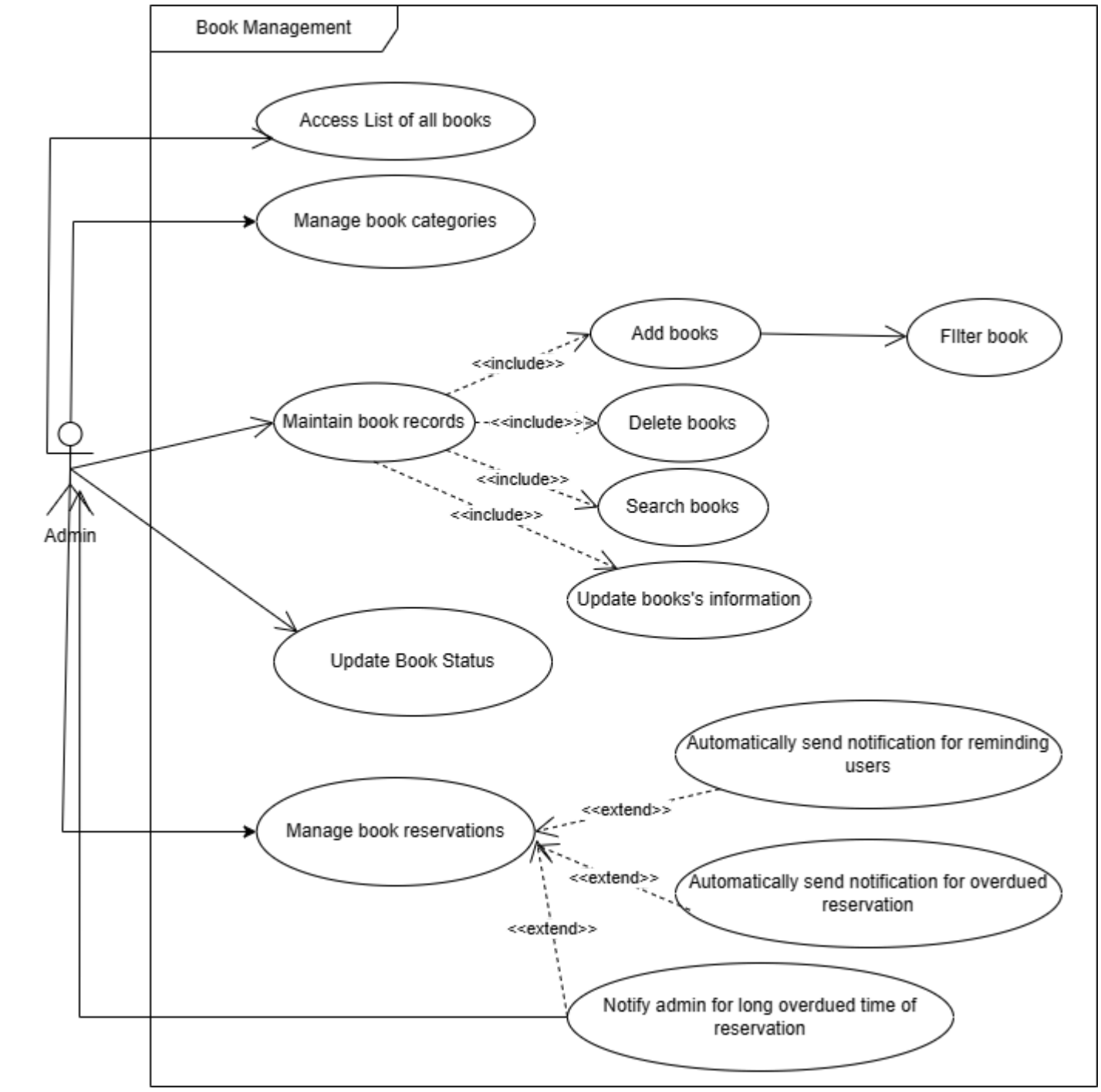
Use Case Diagram 2: Login/Register and Authentication



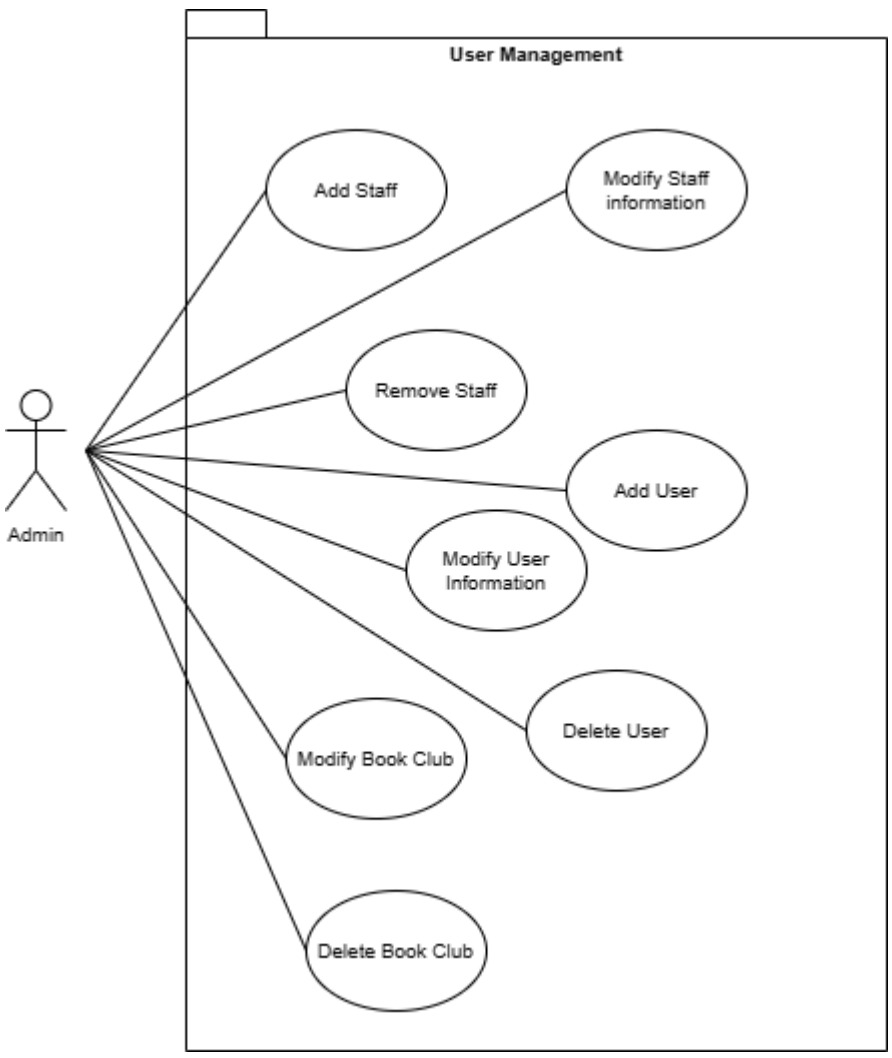
Use Case Diagram 3: Browse a book

Use Case Diagram 4: Borrow a book

Use Case Diagram 5: Book Management - Admin



Use Case Diagram 6: User Management - Admin



III.4. Requirement Analysis - UML Diagram

III.4.1 Login and Register

This sequence diagram illustrates the procedure of login and registration for both users and administrators. The process commences with either the user or the administrator initiating the sign-in process by providing their email and password. Upon verification of the provided details, access to the library website is granted. In the event of incorrect credentials, the system displays an error message to thwart potential hacking attempts, prompting the user to re-enter their email or password. Should the user forget their password, they can conveniently reset it via their registered email address. For users who do not possess an account yet, registration is mandatory, necessitating the provision of their email and a preferred password. The system dispatches a

confirmation link to the provided email address, which the user must click for authentication. Subsequently, upon successful registration, they are redirected to the login page.

Initially, upon clicking the 'sign in' button, users are directed to the login screen. Subsequently, they are prompted to select between signing in as a Student/Guest or as an Admin. If the user chooses the Student/Guest option, they are presented with two choices: Registration or Sign In, as outlined in the aforementioned sequence. In the Registration option, if the user inputs a registered email, the system will prompt the message "Email has already been registered" and subsequently guide the user to the login screen to input their email and password. Last but not least, should the user forget their password, they can conveniently change it via email.

The process begins with the Login Page open state. Users can either Login as Student/Guest or Login as Admin. If a student/guest tries to log in and the email entered is invalid or already registered, it prompts for a new email entry. For new users, they proceed to Registration. After registration, they receive a Verification link via email. Upon clicking the verification link, the account gets Verified leading to successful Account Registration. If users forget their password, they can enter their registered email to receive a reset link. They can then enter a new password, resulting in a successful password change.

III.4.2.[USER] Browse and Search Book Sequence, Activity and State Diagram

User initiates browsing by requesting all available books. Library System queries the Database for all books. Database sends a list of all books back to the System. System presents the list of available books to the User. Searching Scenario: User initiates searching by providing specific criteria (e.g., title, author). Library System forwards the search criteria to the Database. Database searches for books matching the criteria and sends results back. System handles two possibilities: Matching Books Found: System presents the list of matching books to the User. No Matching Books Found: System informs the User that no books match the criteria.

The user begins browsing books, from which they can search for a specific book or view details of a selected book; if searching, they can transition to viewing details or return to browsing, and from viewing details, they can either return to browsing or go back to the previous state

Users begin in the "Browsing" activity, where they can either request all available books or initiate a search. If users choose to browse, all available books are displayed. From the list of displayed books, users can either select a book to view its details or initiate a search. If users initiate a search, they enter search criteria. Once search criteria are entered and submitted, the system searches for books matching the criteria. If matching books are found, the system displays the results, and users can view the details of any of the matching books. If no matching books are found, the system informs the user, and they can continue browsing. The process loops back to the browsing activity after viewing details or handling search results.

III.4.3.[ADMIN] Add/Edit books' details

This sequence diagram outlines the secure process for an administrator to manage book details in an online library system. After logging in, the administrator can retrieve a list of all books. The core functionalities involve adding new books by providing details and creating database entries. Alternatively, the administrator can select existing books, edit their information, and update the corresponding database records. The system confirms successful actions and provides error messages for failed login attempts.

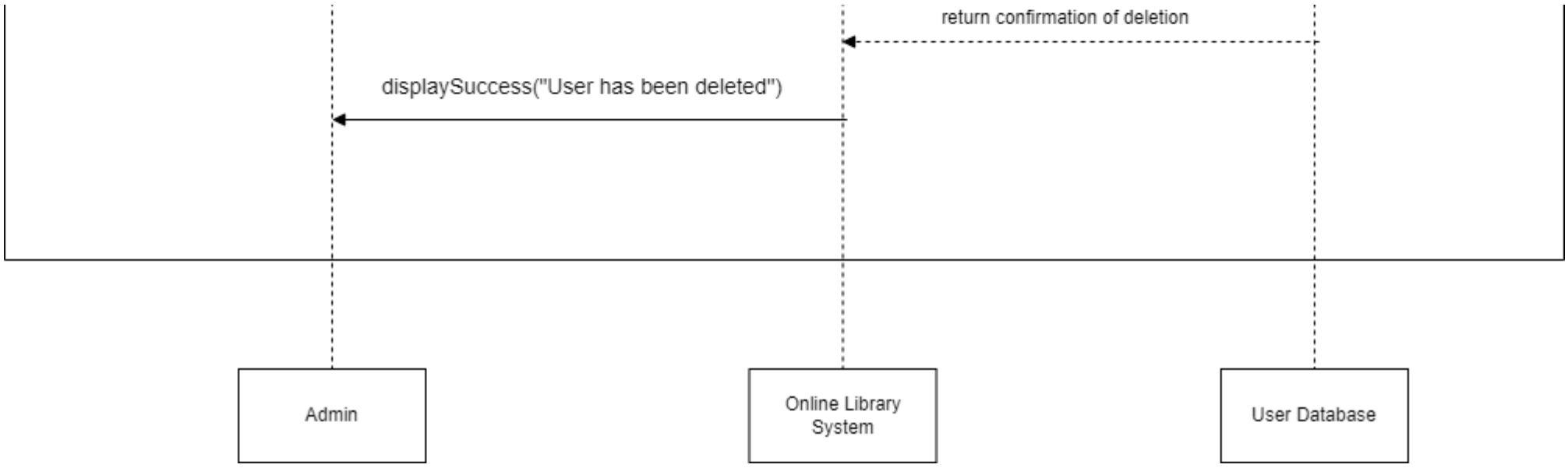
III.4.4.Rating/Review Books

The sequence diagram shows the interaction between a user and a system that allows the user to log in, select a book, and provide a rating/review. The system retrieves book details from a database and stores the rating and review in the database. The diagram also includes confirmation messages indicating the successful completion of each step.

The activity diagram that shows the process of a user logging in, initiating a booking, selecting a specific booking to rate and review, entering the rating and review, and verifying the rating and review. The system then saves the rating and review, sends a notification to the administrator, and the administrator reviews the rating and review. After the administrator's review, the system sends a notification to the user and the provider. The user then views the rating and review. The activity diagram shows the flow of activities from one step to another, with a decision node for the verification of the rating and review. This diagram shows the process of a user rating and reviewing a booking, with the involvement of the administrator and the provider. The activity diagram provides a clear overview of the steps involved in the process and the flow of the process.

III.4.5.User Management

Admin
Online Library System
User Database
Login(email, password)
Validate Login
alt
[Login successfully]
requestUserList()
getUserList()
return UserList
displayUserList(UserList)
[else]
displayLoginError()
alt
[Add user]
selectUserRegistration(RegID)
getUserRegistration()
return UserRegistration
acceptRegistration()
addUser()
saveUser()
displaySuccess("User added successfully")
[else]
selectUserRegistration(RegID)
getUserRegistration()
return UserRegistration
denyRegistration()
removeRegistration()
displaySuccess("Registration has been denied")
alt
[Delete user]
deleteUser(UserID)
removeUser()



Firstly, the administrator initiates a login process within the Library System and awaits authentication. Upon successful login, the administrator requests the User List. Subsequently, the system transmits this request to the database, which returns the list of users to the administrator. In the event of a login failure, the system will prompt an error message labeled "Login Error".

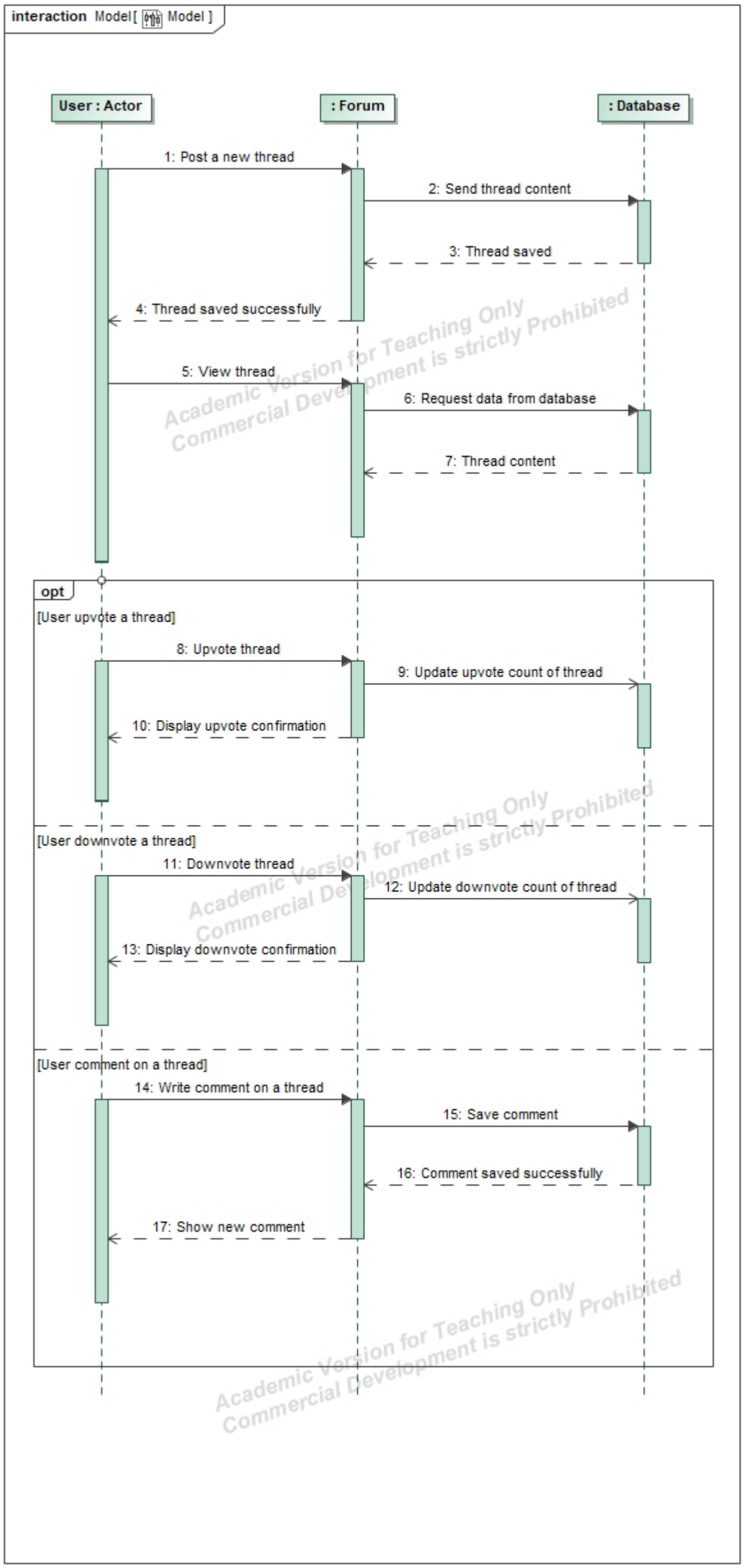
Should the administrator wish to add a user, they must select "User Registration" and proceed to confirm the action. Subsequently, the system will store the new user in the User database and present a confirmation message, "User added successfully", on the interface for the administrator. Conversely, if the administrator chooses to reject the registration, the User Database will discard the registration, and the system will notify the administrator with the message "Registration has been denied".

Finally, if the administrator intends to delete a user, they can execute the "deleteUser(UserID)" command, prompting the database to remove the specified user. The system will then display the message "User has been deleted".

III.4.6. Book Forum Sequence, Activity and State Diagram

This feature serves as the social aspect of our project. Users can share their thoughts and opinions on a book or an personal interpretation of meaning of a book. This feature would create a sense of community for our take on the library app.

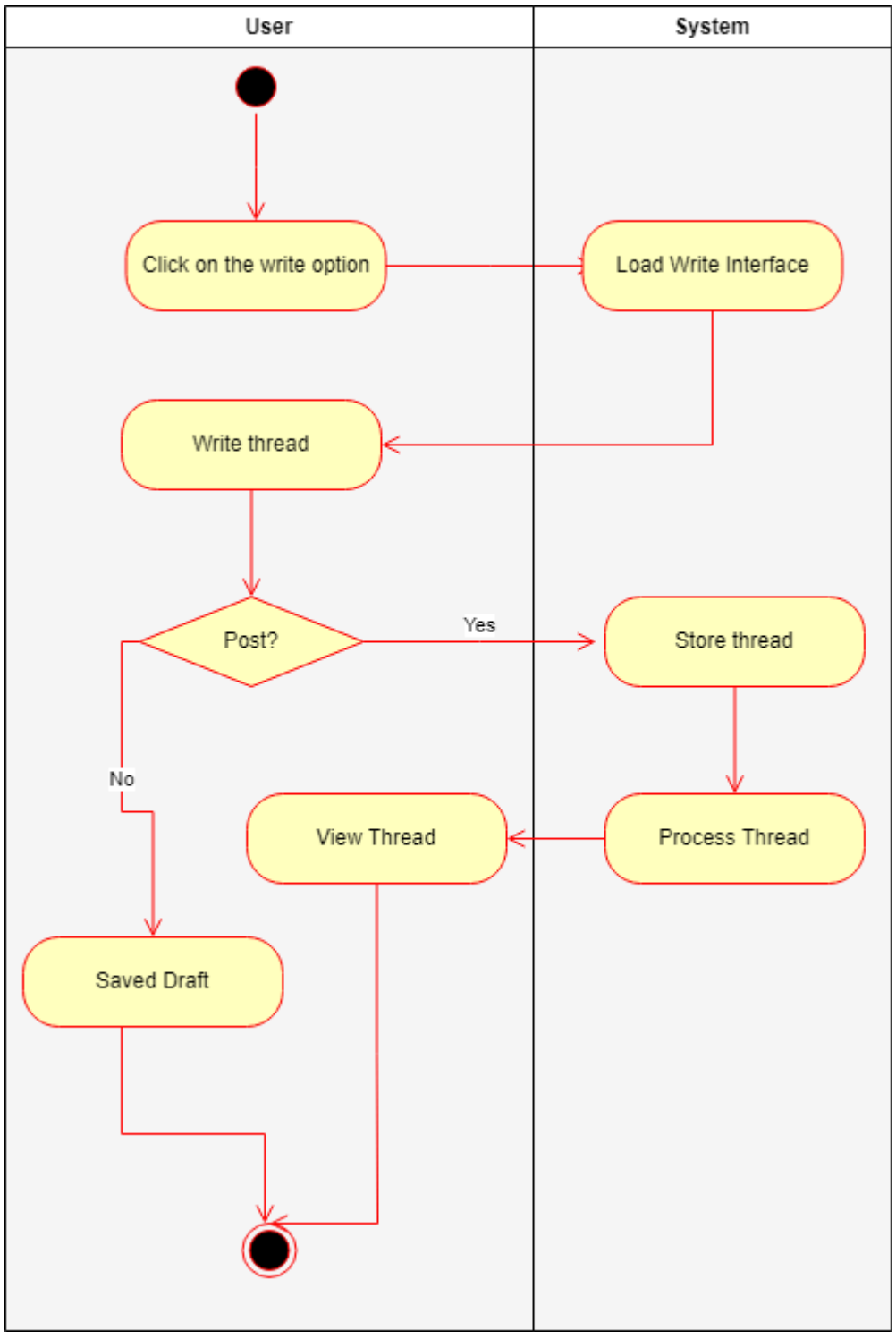
Sequence Diagram



The user can post a new thread, which is then saved in the database. They can also view existing threads, which requires the system to request data from the database. Additionally, users can upvote or downvote threads, and can write comments on threads, all of which are saved in the database and reflected back to the user. With every step of the way, the app must interact with the database to retrieve information and update it accordingly.

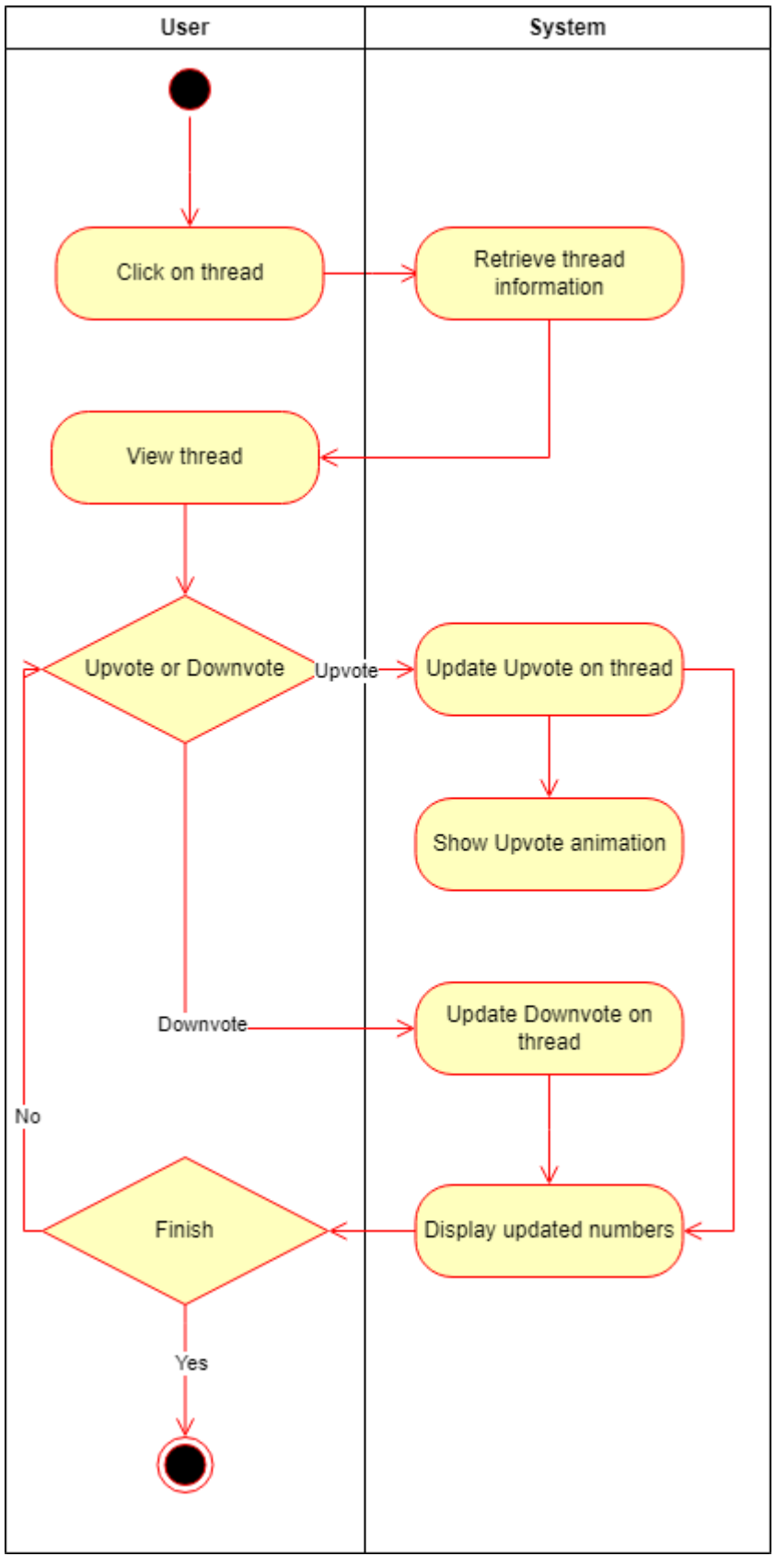
Activity Diagrams

Posting



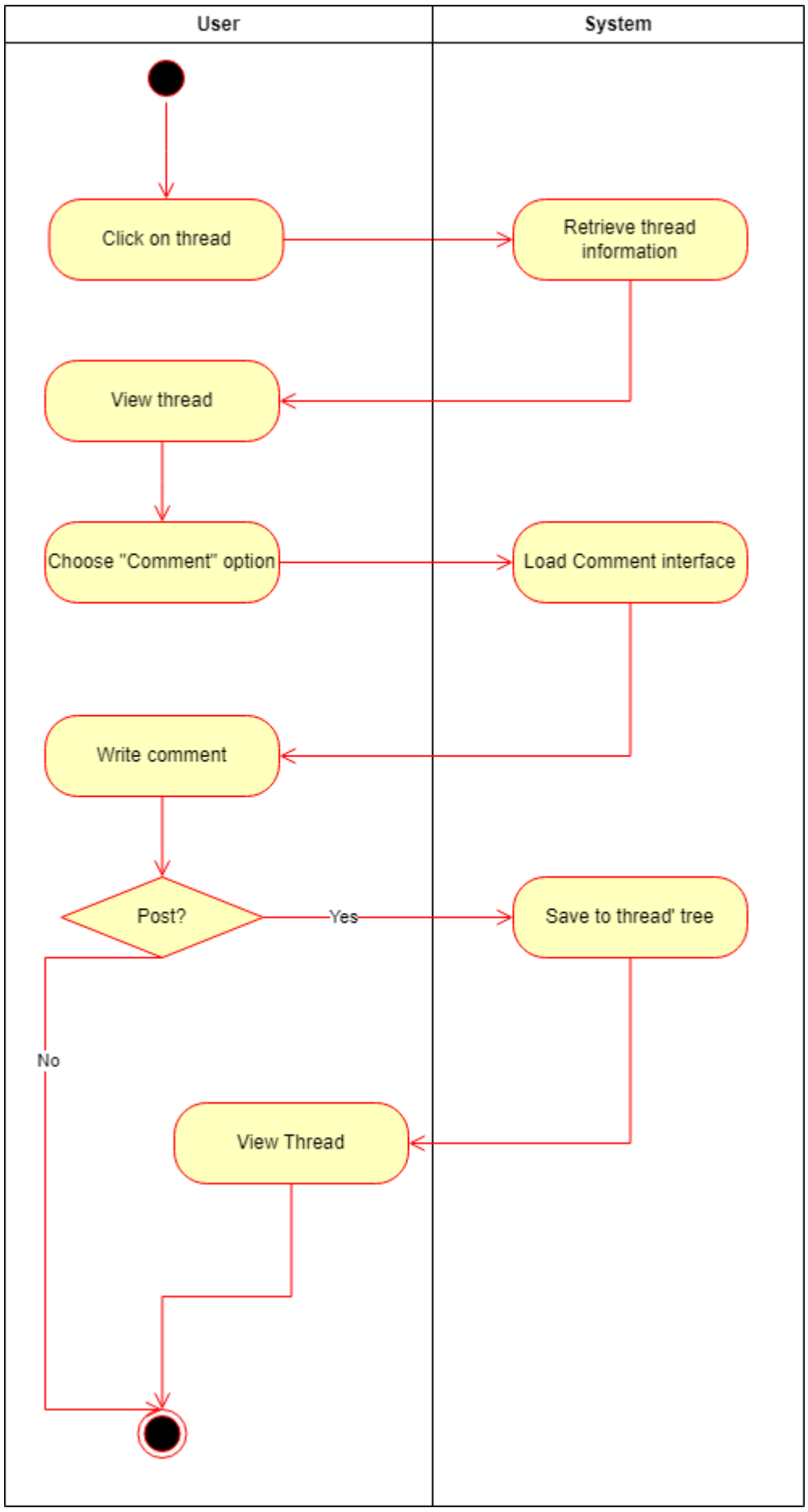
This diagram depicts a flowchart of a system for creating a thread. One noticeably component is that the user can choose to post it publicly or save it as a draft for later.

Upvote and Downvote



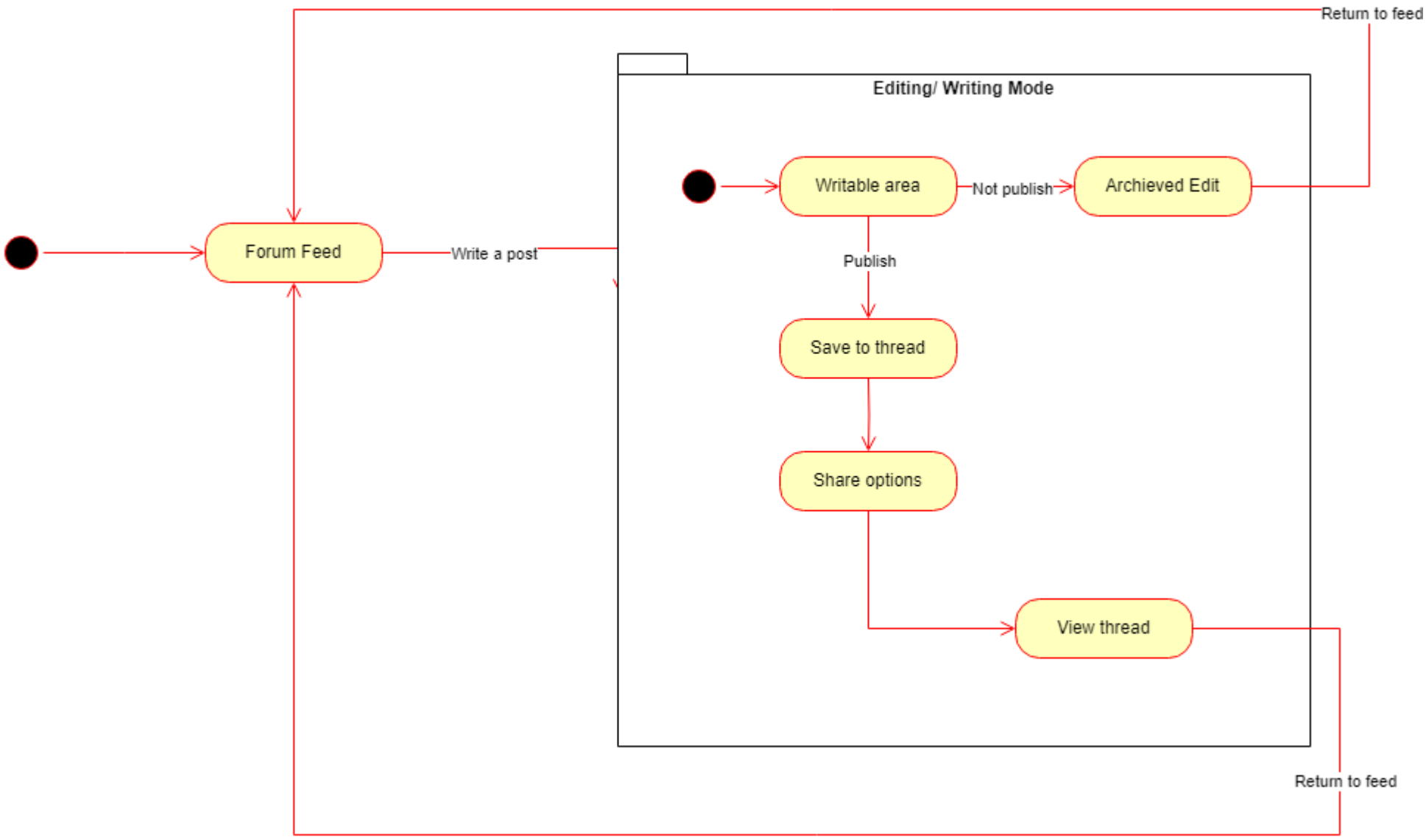
Upvoting is one of the mechanisms for users to positively rate or express approval for a thread in online forums, while downvoting allows users to negatively rate or express disapproval. Both actions contribute to the overall rating and visibility of the thread, serving as a means for users to express their opinions and influence the community's perception of the content.

Comment



Comments are essential in online forums. They help people talk, share, and connect. Users can express thoughts, ask questions, and discuss topics. Comments make content richer and improve the user experience. They build a sense of community and help people form relationships. Comments also give feedback to content creators, helping them understand their audience better. Overall, comments are vital for keeping forums lively and engaging.

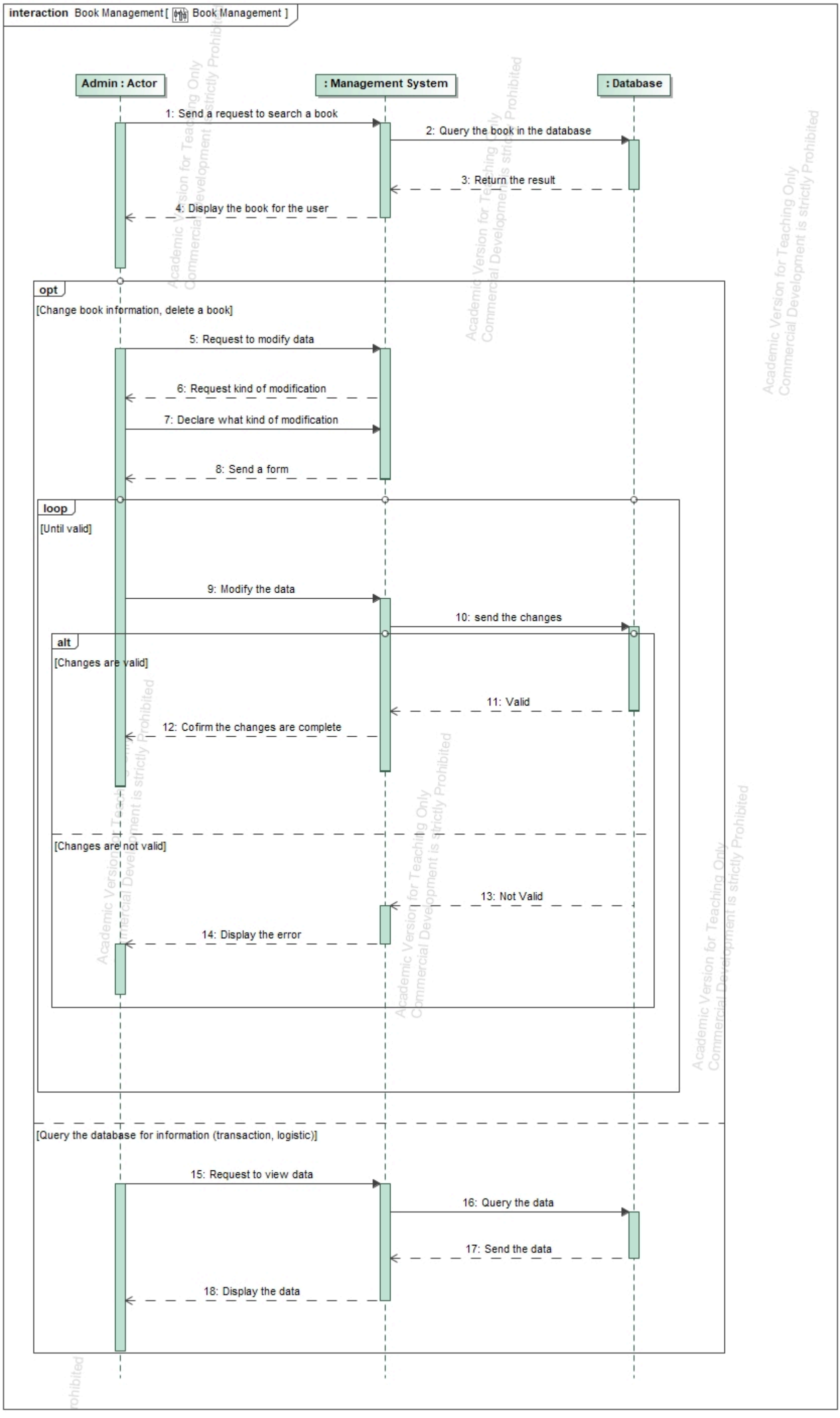
State machine Diagram



It starts with the writer creating the text. After this initial draft, they enter an "Edit" stage, likely for revising and proofreading. Once satisfied, they can choose to "Publish" the work. This could involve submitting it to a platform or making it public. Finally, the published text is archived for safekeeping, ensuring a saved copy exists. \

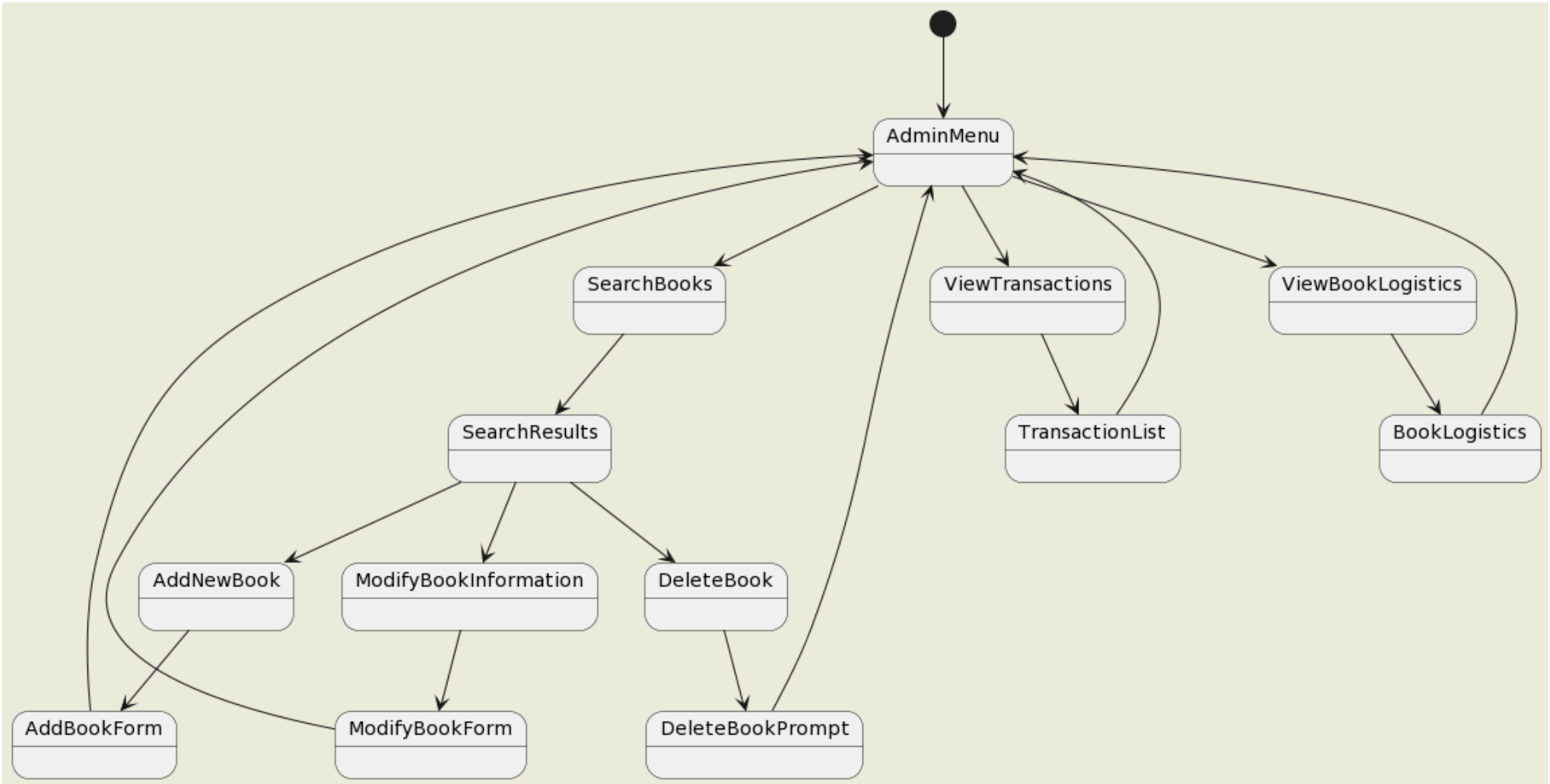
III.4.7. Book Management Sequence and State Machine Diagram

Sequence Diagram



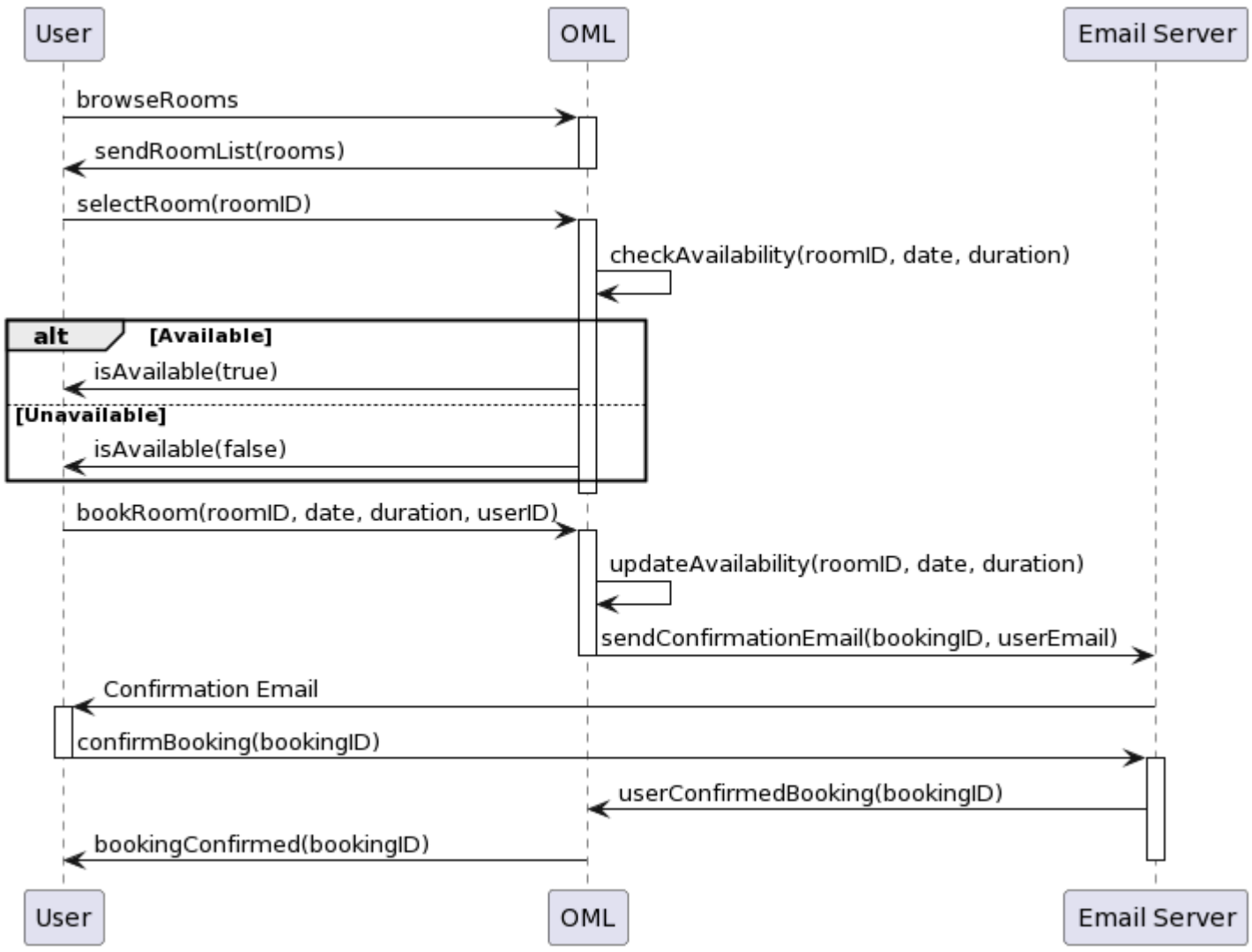
The diagram showcases the interactions between the admin user and the system. The admin can initiate a search for books, add new books, delete existing books, modify book information, view transactions, and access book logistics. The diagram demonstrates the sequential order of these actions, showcasing how the admin interacts with the system at each step. It provides a visual representation of the process and highlights the various states and functions involved in managing books on the library website.

State Machine Diagram



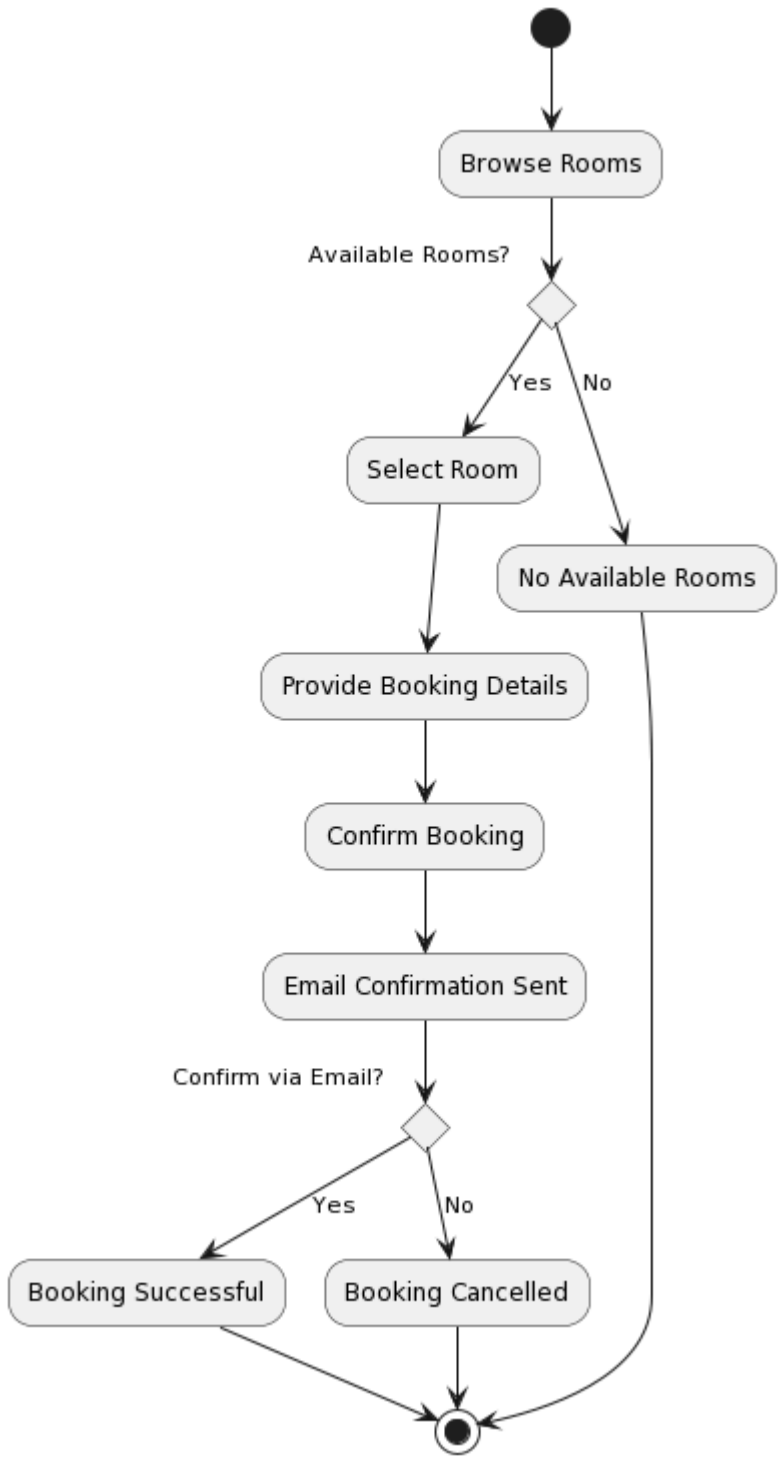
This state machine offers a structured approach for administrators to interact with the system. It guides the user through distinct states, such as searching the catalog, reviewing past transactions, or managing book inventory. Within the inventory management state, sub-states allow for adding new books, modifying existing entries, or deleting titles. This state-based approach ensures clear and efficient interaction with the library's book collection.

III.4.8. Room Booking Sequence, Activity and State Diagram

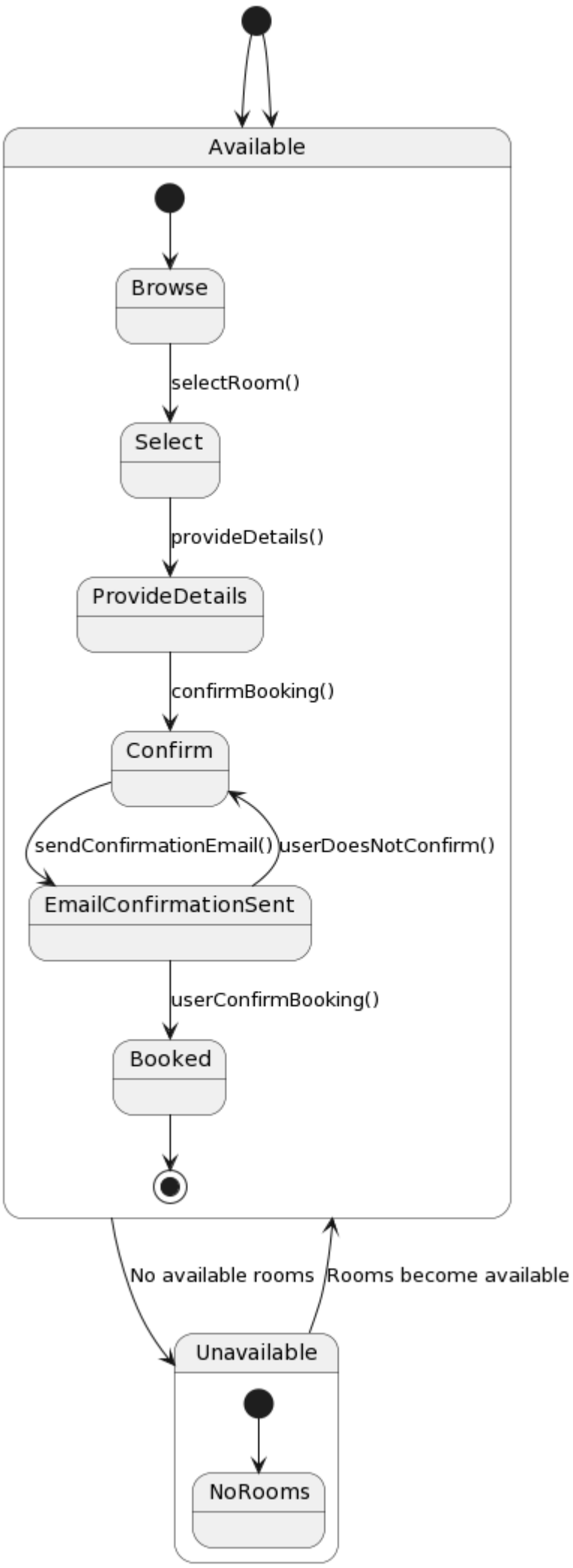


This sequence diagram depicts the process of booking a room with email confirmation. It begins with the user browsing available rooms, selecting a room, and checking its availability. If the room is available, the user proceeds to book it. Upon booking, the Order Management System (OML) updates the room's availability status and sends a confirmation email to the user via the Email Service. The user confirms the booking via email, and upon

confirmation, the OML acknowledges the booking.

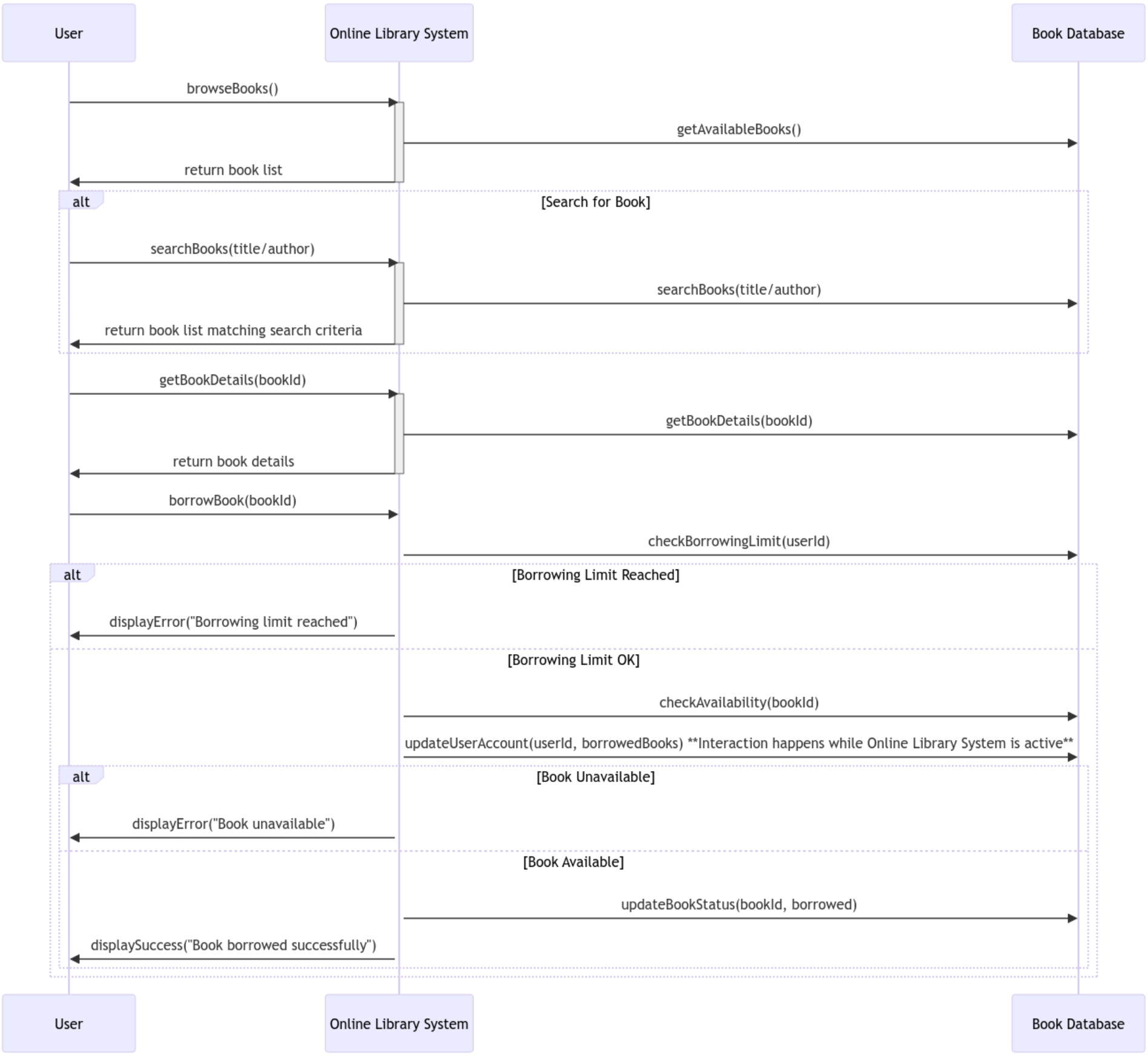


This activity diagram outlines the process of booking a room with email confirmation. It starts with the user browsing available rooms. If rooms are available, the user selects a room, provides booking details, and confirms the booking. An email confirmation is then sent. If the user confirms the booking via email, it is marked as successful; otherwise, the booking is canceled. If no rooms are available, the process ends.



This state machine diagram illustrates the process of room booking. It starts in the "Available" state where users can browse rooms. If rooms are available, they can select a room, provide details, and confirm the booking. After confirmation, an email confirmation is sent. If the user confirms via email, the booking is marked as successful. If rooms are not available, the system transitions to the "Unavailable" state, and users are informed. When rooms become available again, the system transitions back to the "Available" state.

III.4.9.[USER] Room Booking Sequence Diagram



This sequence diagram depicts the functionalities of an online library system for book borrowing. Logged-in users enjoy full access, allowing them to browse, search, view book details, initiate borrowing requests, and receive confirmation upon successful borrows.

III.5. UI Prototype

III.5.1 Theme and Reference

Main theme: Mythic/Mystic

Name and logo of the library:

III.5.2. Lunacy Design

Lunacy is a software that allows you to create designs for mobile applications and websites. It is an ergonomic software with all the features to achieve powerful designs. It is a UX and UI design tool that you can use to design applications or websites.

Login

Browse Book

MYTHOS MYSTERIUM
A MYTHIC LIBRARY

Duy Dinh Thai

Home Search Book Borrow Book Social Forum Room Booking FAQ

Retain Filters | Clear Filters

Search Results Display
- Show all (2,900)
- Hide Duplicates
- Group editions and formats

Expand This Search With
- Related Terms

Format
- Article, Chapter (2.6K)
  - Article
  - Download Article (252)
- Book (328)
  - eBook (323)
  - Printed Book (5)
- Journal, Magazine (1)
  - eJournal, e Magazine (1)

Content Type
- Full Text
- Open Acess
- Biography (20)
- Fiction (2)
- Non Ficiton (2.9K)
- Peer Reviewed

Publication Year
- All
- Last 5 Years
- Last 10 Years
- Last 25 Years

Author/Creator
- Albert A (23)
- Abbott B P (13)
- Fabre Laetitia (10)
- Adriean Martinez S (9)
- Cushman Mary (9)

Show More

Subject
- All
- Computer Science (136)
  - Computer Software (39)
  - Computer Programming & Programming Languages
  - Computer Networks (20)

Show More

About 2400 results at Mythos Mysterium Library

Sort: Best Match

View Ebook

Harry Potter: Half Blood Prince

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library
Home delivery possible

View Ebook

The Last Four Things

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library
Home delivery possible

View Ebook

Awake

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library
Home delivery possible

View Ebook

Princess and The Cat

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library
Home delivery possible

The Last Four Things

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library

- [ ] Language & Literature (44)
- [ ] American Literature (18)
- [ ] English Literature (5)
- [ ] Philology, Linguistics (3)

Show More

Show More

Language

- [ ] English
- [ ] Vietnamese
- [ ] Italian
- [ ] German
- [ ] Korean

Show More

View Ebook

Home delivery possible

View Ebook

Awake

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library

Home delivery possible

View Ebook

Princess and The Cat

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library

Home delivery possible

View Ebook

Princess and The Cat

Authors: JK Rowling

Summary: "Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005....
Show more

Available at our library

Home delivery possible

1 2 3 4 ...

HELP US TO IMPROVE OUR LIBRARY

We welcome every comment about our wide range of Library services as your views significantly help us ensure the best possible services for today and develop even better services in the future.

SURVEY | BOOK DONATION | FEEDBACK | LOST AND FOUND

VIETNAMESE-GERMAN UNIVERSITY LIBRARY
Ring road 4, Quarter 4, Thoi Hoa Ward, Ben Cat Town,
Binh Duong Province
Tel. (0274) 222 0990 - 70206

MYTHOS MYSTERIUM
A MYTHIC LIBRARY

Passion for knowledge, passion for lifelong learning

Book Details

MYTHOS MYSTERIUM
A MYTHIC LIBRARY

Duy Dinh Thai

Home Search Book Borrrow Book Social Forum Room Booking FAQ

Back to Search Results

Harry Potter: Half Blood Prince

JK Rowling

Get ready to uncover the dark secrets and betrayals in the book. A thrilling adventure awaits you.

View Ebook Borrow

Available at our library

Home delivery possible

ISBN: 987 3 32564 455B

Authors: JK Rowling

Publisher: Bloomsbury Publishing (in the United Kingdom), Scholastic (in the United States)

Year of Publication: 2005

Main Category: Fiction

Gerne: Fantasy, Adventure

Shelf Location: S5

Languages: English

Paper Back: paper texture, full color, 345 pages

Tags: fiction children english english magic fantasy art english magic fantasy art english magic fantasy art fiction children english

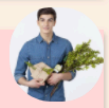
Description:
"Harry Potter and the Half-Blood Prince" is the sixth book in the Harry Potter series written by J.K. Rowling. It was first published by Bloomsbury Publishing in the United Kingdom and Scholastic in the United States in 2005. The book continues the story of Harry Potter, a young wizard attending Hogwarts School of Witchcraft and Wizardry.

In this installment, Harry Potter returns to Hogwarts for his sixth year and discovers a mysterious textbook that once belonged to the "Half-Blood Prince." As he delves into the book, Harry uncovers secrets about Lord Voldemort's past and begins to understand the importance of his own role in the battle against the dark wizard. The book delves deeper into the darker aspects of the wizarding world and sets the stage for the final confrontation between Harry and Voldemort.

Nguyen Minh Thuan Homo

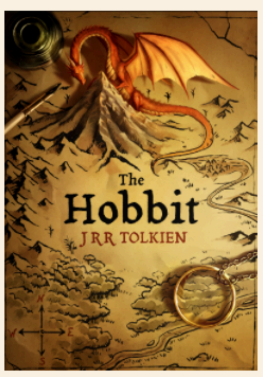
This book is amazing! I love it, it brings me a special time. Getting a journey with Harry Potter always makes me happy.

I love it a lot! It is really interesting. **Doan Quang Tien**



Read more at our Social Forum

EXPLORE MORE BOOKS



HELP US TO IMPROVE OUR LIBRARY

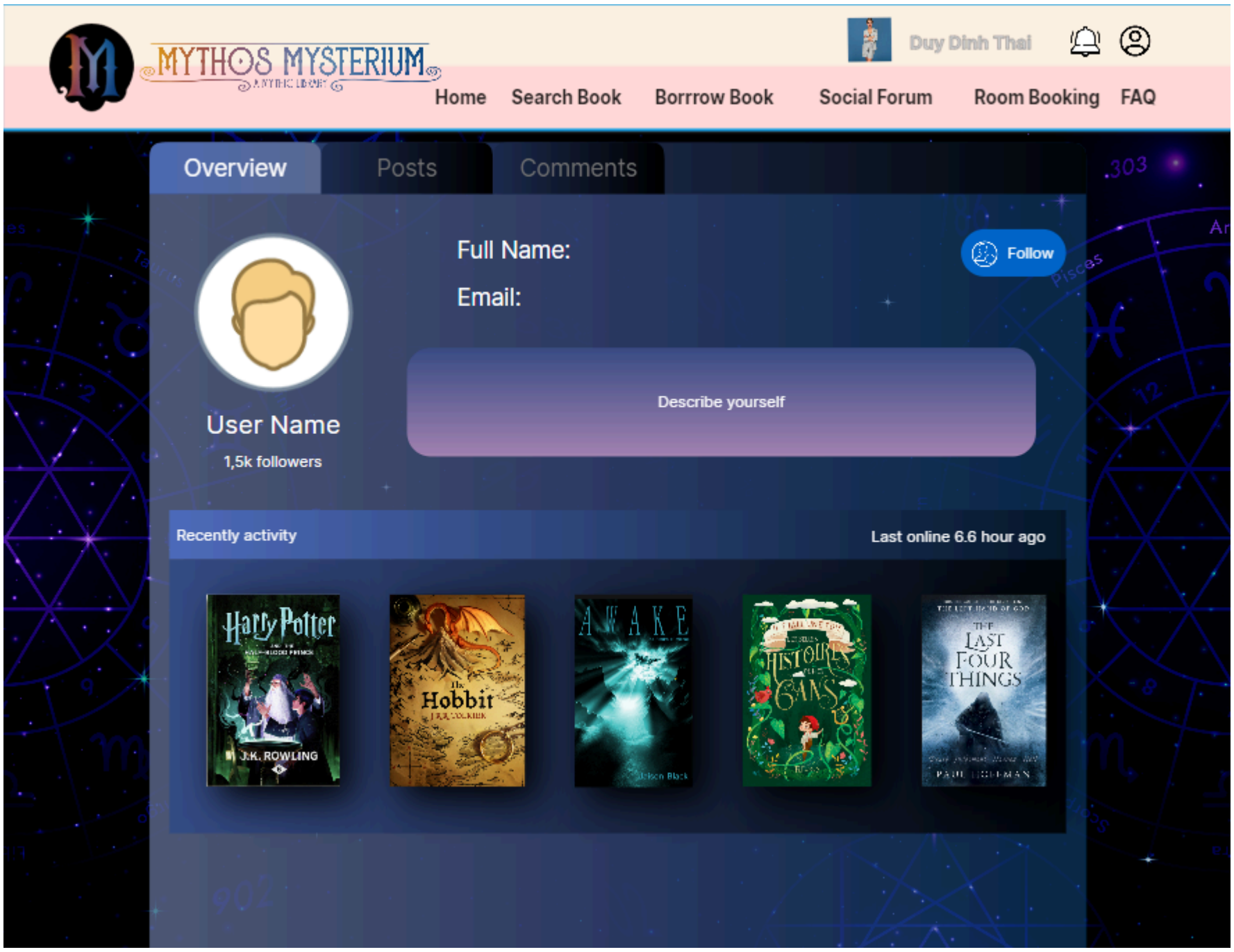
We welcome every comment about our wide range of Library services as your views significantly help us ensure the best possible services for today and develop even better services in the future.

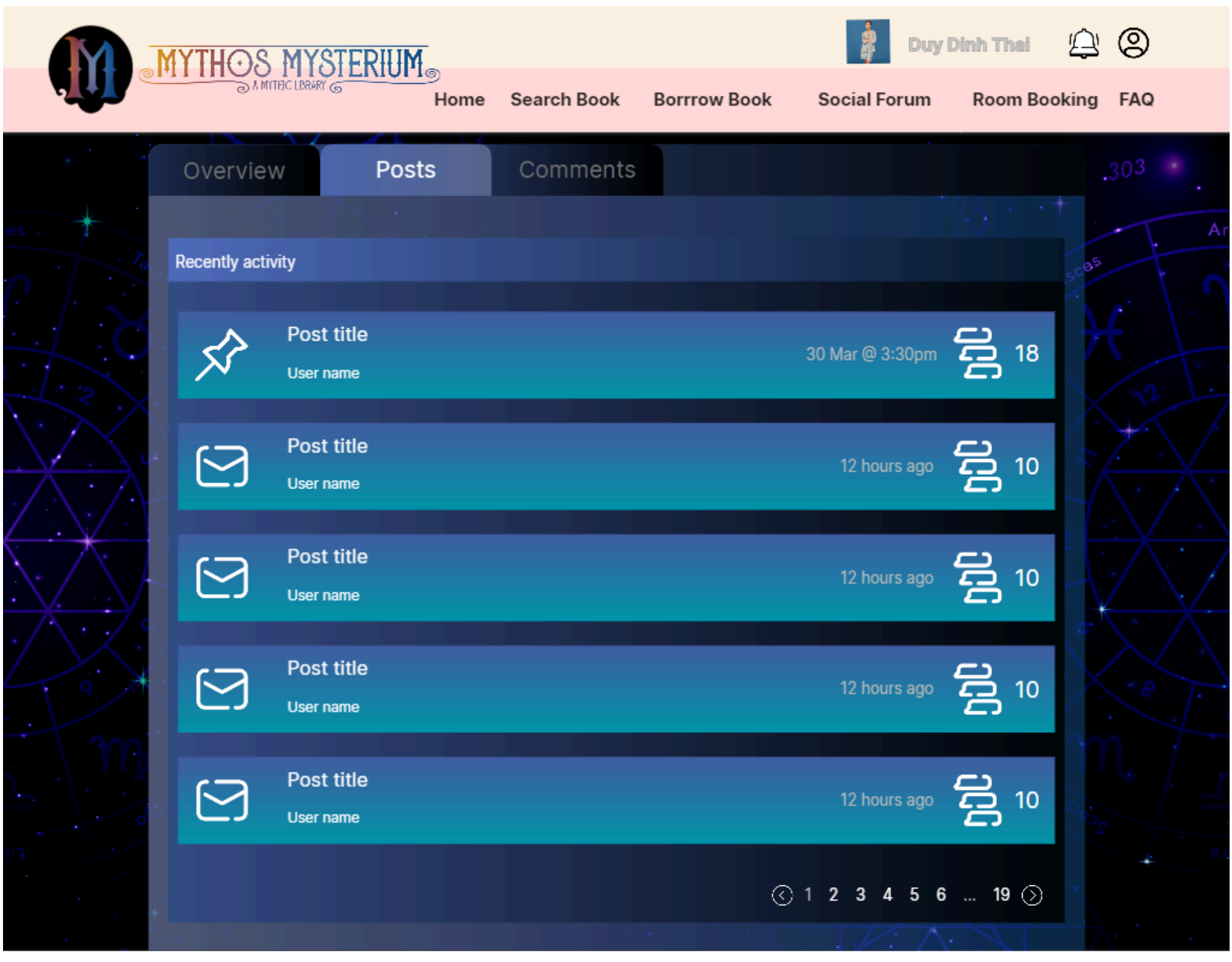
SURVEY BOOK DONATION FEEDBACK LOST AND FOUND

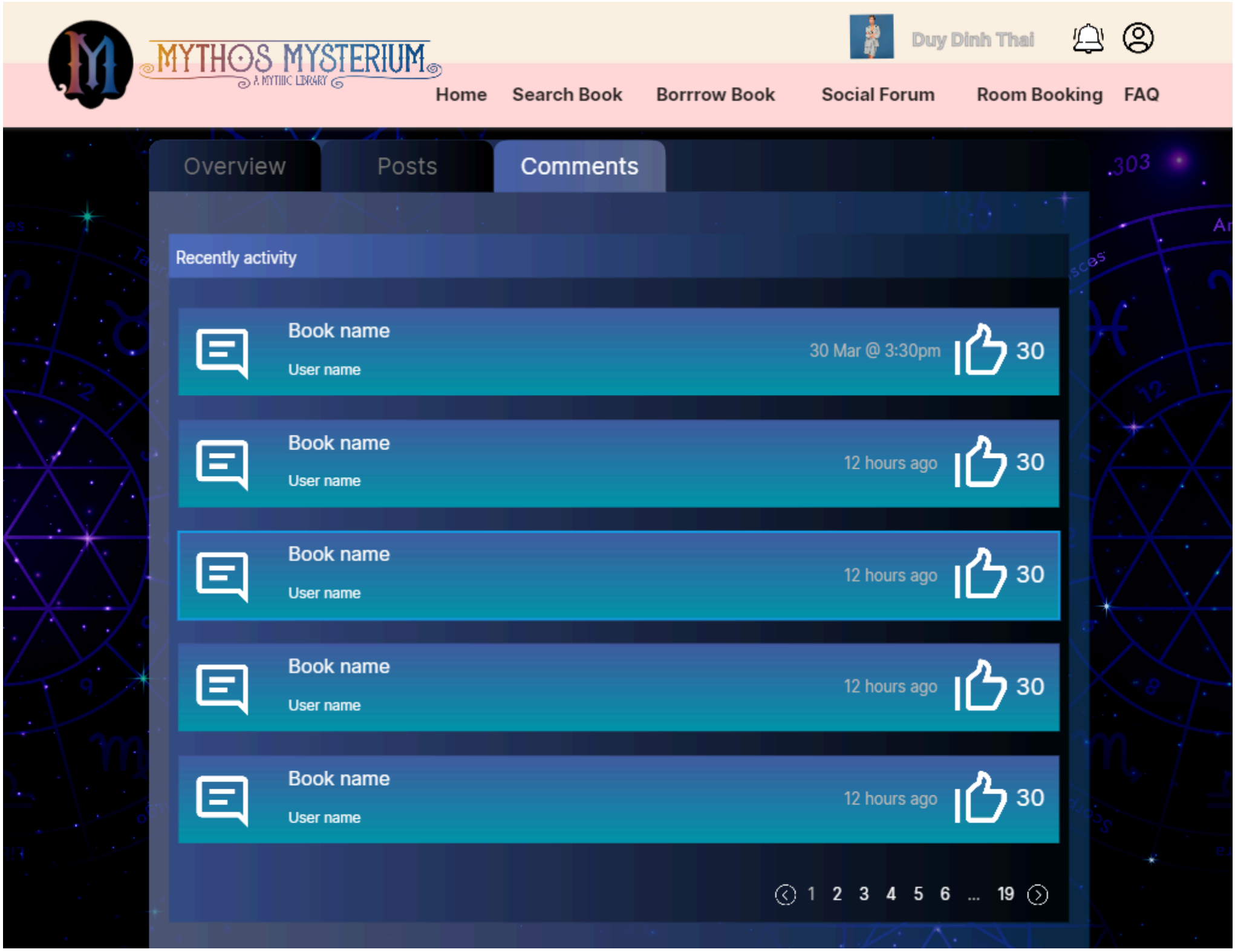
VIETNAMESE-GERMAN UNIVERSITY LIBRARY
Ring road 4, Quarter 4, Thoi Hoa Ward, Ben Cat Town,
Binh Duong Province
Tel. (0274) 222 0990 - 70206

MYTHOS MYSTERIUM
A MYTHIC LIBRARY

Passion for knowledge, passion for lifelong learning

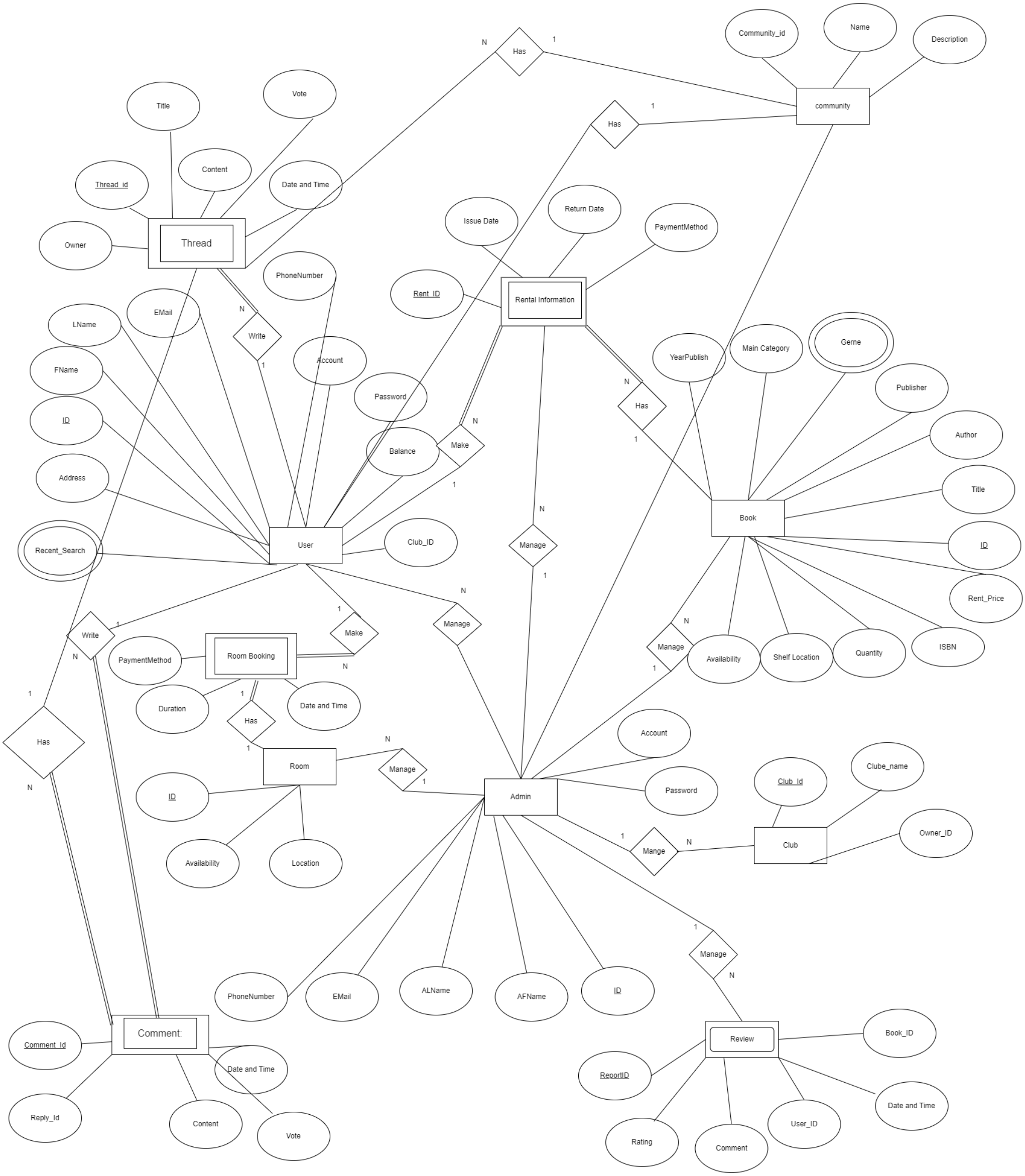

MYTHOS MYSTERIUM
A MYTHIC LIBRARY
Duy Dinh Thai
Home
Search Book
Borrrow Book
Social Forum
Room Booking
FAQ
Overview
Posts
Comments
Full Name:
Email:
Follow
User Name
1,5k followers
Describe yourself
Recently activity
Last online 6.6 hour ago
Harry Potter
J.K. ROWLING
The Hobbit
AWAKE
HISTOIRES DE GANS
THE LAST FOUR THINGS


MYTHOS MYSTERIUM
A MYTHIC LIBRARY
Duy Dinh Thai
Home
Search Book
Borrrow Book
Social Forum
Room Booking
FAQ
Overview
Posts
Comments
Recently activity
Post title
User name
30 Mar @ 3:30pm
18
Post title
User name
12 hours ago
10
Post title
User name
12 hours ago
10
Post title
User name
12 hours ago
10
Post title
User name
12 hours ago
10
1 2 3 4 5 6 ... 19


MYTHOS MYSTERIUM
A MYTHIC LIBRARY
Duy Dinh Thai
Home
Search Book
Borrrow Book
Social Forum
Room Booking
FAQ
Overview
Posts
Comments
Recently activity
Book name
User name
30 Mar @ 3:30pm
30
Book name
User name
12 hours ago
30
Book name
User name
12 hours ago
30
Book name
User name
12 hours ago
30
Book name
User name
12 hours ago
30
1 2 3 4 5 6 ... 19

III.6. Database Design

Entity-Relationship Diagram for Database



Entities

User:

- ID: Unique identifier for a user account.
- FName: User's first name.
- LName: User's last name.
- Email: User's email address.
- PhoneNumber: User's phone number.
- Account: User's account for login.
- Password: User's account password (encrypted text for security).

- Balance: User's account balance.
- Club_Id: User's Club ID to know which club the user belongs to.
- Address: User's address used in book delivery.
- RecentSearch: User's search history used in AI recommendations.

Admin:

- ID: Unique identifier for an administrator account.
- AFName: Administrator's first name.
- ALName: Administrator's last name.
- Email: Administrator's email address.
- PhoneNumber: Administrator's phone number.
- Account: Administrator's account for login.
- Password: Administrator's account password (encrypted text for security).

Book:

- ID: Unique identifier for a book.
- IBSN: Unique numeric identifier used globally to catalog and manage books.
- Title: Title of the book.
- Author: Author(s) of the book.
- Publisher: Publisher of the book.
- YearPublish: Year the book was published.
- MainCategory: Main category the book belongs to (e.g., Fiction, Non-Fiction).
- Genre: Subcategory the book belongs to (e.g., Mystery, Romance) (multivalued).
- Quantity: Number of copies of a book with this tag.
- Availability: Boolean to check if that book is available to be rented.
- Shelf location: Location used in delivery to user.
- Rent_Price: Rental price for a book.

Rental Info:

- Rent_ID: Unique identifier for a rental transaction.
- IssueDate: Date the book was rented out.
- ReturnDate: Date the book is due to be returned.
- PaymentMethod: User payment method (cash, Visa card, etc.).

Review:

- Report_ID: Unique identifier for a review/report.
- Rating: Rating given to the book (1-5 stars).
- Comment: User's comment about the book (optional).
- Date_Time: Date that user rating the book

Club:

- Club_Id: Unique Identifier for a club.
- Club_name: Name of the club.
- Club_Owner: The user who created the club (User ID).

Thread:

- Thread_Id:Unique identifier for a post in forum(number)
- Content: Any status from user(text)
- Owner: To know who post this thread
- Vote: UpVote/ Down Vote for the comment like the thread
- Title: Title of the Thread
- Date: Date and time user post(date)

Comment:

- Comment_Id: Unique identifier for a comment in the forum.
- Thread_id: Post ID to know which post the comment belongs to.
- Content: Any text from the user.
- Vote: UpVote/ Down Vote for the post
- Reply_id: To know if the comment belongs the other comment
- Date and Time: Date and time the user commented.

Community

- Community_id: Unquie identifier for the community
- Description: Introduction to community

- Name: Name of the community

Room:

- Room_id: A unique identifier for each room within the system.
- Availability: Indicates whether the room is currently available or booked.
- Location: Provides information about the physical location of the room.

Roombooking:

- Date and Time: Records the date and time of a room booking.
- Duration: Represents the duration for which the room is booked.
- PaymentMethod: Specifies the payment method chosen by the user for the booking.

Database schema (Relational model)

Relationships

- User can create many Thread (One-to-Many)
- A Thread can have many Comment (One-to-Many) by different Users (Many-to-Many)
- A User can be a member of many Club (Many-to-Many) (optional relationship depending on system requirements)
- Admin manages the Library Information System
- A User can have many Rent (One-to-Many)
- A Book can be rented in many Rent (One-to-Many)
- A Rent can have one corresponding Payment (One-to-One) (optional relationship depending on system requirements)
- A User can make many RoomBooking (One-to-Many)
- 1 Community may have N users - the users is the followers of the Community(one-to-Many)
- 1 Cominity may have N Thread(One-to-Many)

IV. Implementation

IV.1. Folder Structure and Scope

The library-api folder contains the server-side code for your application. It is divided into several subfolders:

- config: This folder is used to store configuration files for your application, such as database connections, environment variables, logging, and other application-specific configurations.

- controllers: The controller folder contains the logic for handling incoming requests and generating appropriate responses. Controllers typically handle the business logic of your application, retrieve data from models, perform any necessary validations, and orchestrate the flow of data.

- models: The models folder is responsible for defining the data schema and interacting with the database or any other persistence layer. It contains the code for creating, querying, updating, and deleting data from the underlying storage.

- middlewares: Middlewares are functions that intercept incoming requests and perform certain operations before passing the request to the appropriate controller. They can be used for authentication, authorization, input validation, logging, error handling, and other cross-cutting concerns.

- routes: The routes folder holds the definitions of various routes in your application. Routes determine the URL patterns and HTTP methods that your application will respond to. They map incoming requests to the corresponding controller methods or functions, allowing you to define the API endpoints and their associated behavior. The library-ui folder contains the client-side (front-end) code for your application. It is divided into two subfolders:

- components: This folder contains the React components that make up the user interface of your application.

- services: The services folder holds the code for interacting with the API endpoints provided by the server-side application. It typically contains data services that handle the communication with the backend. This folder structure follows a common convention for organizing the files in a full-stack JavaScript application, separating the server-side (API) and client-side (UI) concerns into their respective folders.

IV.2. Front-end Implementation

1. Login and SignUp

Overview

This component is designed to handle user login functionality within a React application. It utilizes several key features and libraries of the React ecosystem, including React Hooks, asynchronous JavaScript with async/await, and React Router for navigation.Libraries and Technologies Used

- React

- React Hooks **Key Components of the Code**

- *State Management*: The component manages its state using the useState hook. It tracks the user's email and password inputs, as well as any error messages that need to be displayed.

```
const [email, setEmail] = useState('');
const [password, setPassword] = useState('');
const [errorMessage, setErrorMessage] = useState('');
```

- *Form Submission Handling*: The handleLogin function is an asynchronous event handler for form submission. It attempts to log the user in by sending their credentials to a server and handling the response.

```
const handleLogin = async (e) => {
    e.preventDefault();
    }
  };
```

- Navigation: Upon a successful login (indicated by a 200 HTTP status code), the user is navigated to the dashboard page using the navigate function from useNavigate.
- Error Handling: In case of an error during the login process (e.g., incorrect credentials), an error message is captured from the server's response and stored in the component's state to be displayed to the user.

2. Main page

Overview

The `Header.jsx` file defines a React functional component named Header, which serves as the navigation bar for a web application. This component incorporates styling, image assets, smooth scrolling functionality, and navigation capabilities, making it a crucial part of the user interface for guiding users through different sections of the application.

Key Features and Implementation

```
<div className="container">
      <nav className='xdd'>
      <img src={logoM} onClick={()=>{navigate('/')}} alt="" className='logo'/>
      <ul className={mobileMenu?'':'hide-mobile-menu'}>

        <li><Link to="container" offset={0} onClick={()=>{navigate('/')}} smooth ={true} duration = {500}>Home</Link></
        <li onClick={()=>{navigate('/regulation')}}>Regulations</li>
        <li onClick={()=>{navigate('/browsebook')}}>Search Book</li>
        <li onClick={()=>{navigate('/roombooking')}}>Room Booking</li>
        <li><Link to="Delivery" offset={-50} smooth ={true} duration = {500} onClick={handleBookDeliveryClick}>Social F
        <li onClick={handleChatClick}>Chat&Chit</li>
          {userName ? (
            <li>

            <button onClick={handleShowSidebar} className='btn1'>{userName}</button>
            <Sidebar visible={visible} position="right" onHide={handleHideSidebar}>
                <div className="username">
                    <p>{userName}</p>
                </div>
                <div className="function">
                  <Button onClick={() => { navigate('/profile') }} label="Profile" className="p-button-text" />
                  {role=='admin' && (
                <Button onClick={handleDashboard} label="Dashboard" className="p-button-danger p-button-text" />
                )}
                  <Button onClick={handleLogout} label="Log out" className="p-button-danger p-button-text" />
                </div>
            </Sidebar>
          </li>
        ) : (
          <li>
            <button onClick={() => { navigate('/login') }} className='btn1'>Sign in</button>
          </li>
        )}
    </ul>
    <img src={menu_icon} className="menuicon" onClick={toggleMenu} alt="" />
```

```
</nav>
</div>
```

- Styling and Assets: The component imports its styling from ./Header.css and an image asset used as a logo from ../../Assets/logo for header.png. This approach separates concerns, keeping styling and assets organized and easily maintainable.
- Smooth Scrolling: Utilizes the react-scroll library's Link component to enable smooth scrolling to different sections of the application. This feature enhances user experience by providing a smooth transition between sections identified by to props such as "container", "categories", "trending", and "Delivery".
- Navigation: Employs react-router-dom's `useNavigate` hook for programmatic navigation. This is evident in the navigation to the home page (/),
- Responsive UI Elements

b. Main page swiper

This code snippet defines a React functional component named Swipe, which is designed to render a slider or carousel feature within a web application. Here's a breakdown of its key parts:

- Slider is a custom component imported from './Slider.jsx', presumably designed to render the actual slider using the Swiper components or another method.

Slider code explanation:

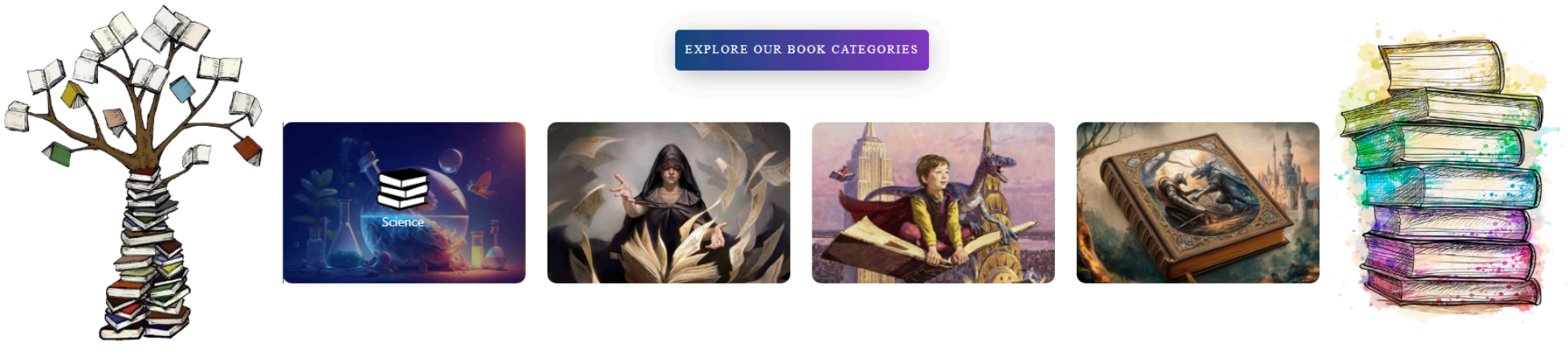
```
const Slider = ({slides}) => {
  return (
}
```

- slides are imported from './mock.json', which contains data for the slides to be displayed in the slider, such as images, titles, and descriptions.

Swipe Component:

Overview

The `Category.jsx` component is a React functional component designed to display a list of book categories in a visually appealing manner. It is part of a larger web application, presumably related to books or a library system. The component makes extensive use of images to represent different book genres, including science, fantasy, adventure, and mystic, enhancing the user interface and user experience.



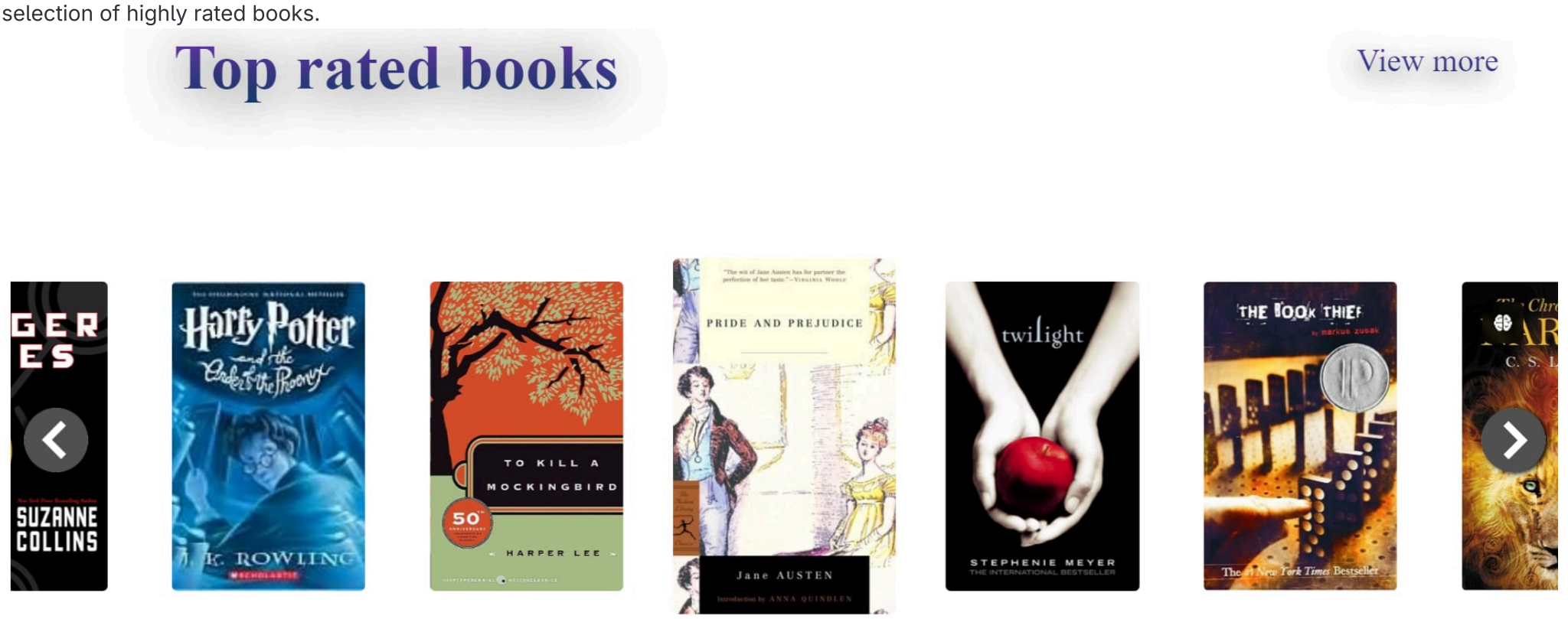
Implementation Details

- Imports and Assets: The component imports React and its CSS stylesheet (Category.css) for styling. It also imports eight image assets, four for the category backgrounds (science, fantasy, mystic, adventure) and four icons (bookicon to bookicon4) to accompany the category names.
- Structure: The component returns a div element with a class name of 'categories', which serves as a container for individual category items. Each category is represented by a div with a class name of "category", containing an image for the category background and a caption. The caption itself is another div that includes an icon and a paragraph tag <p> for the category name.
- Styling and Animation: The accompanying CSS file, `Category.css`, is crucial for the visual presentation and animation of the categories. It includes styles and animations that enhance the appearance of the category sections.

d. Top-Rated

Overview

The `Topbook.jsx` file defines a React component named `Topbook` that showcases a carousel of the top 10 rated books using the Swiper component from the "swiper/react" package. This component is designed to provide an interactive and visually appealing way for users to browse through a selection of highly rated books.



Imports and Dependencies

- Functional Component: Topbook is defined as a functional component that renders a carousel of book images, titles, and authors.

Fetch Data

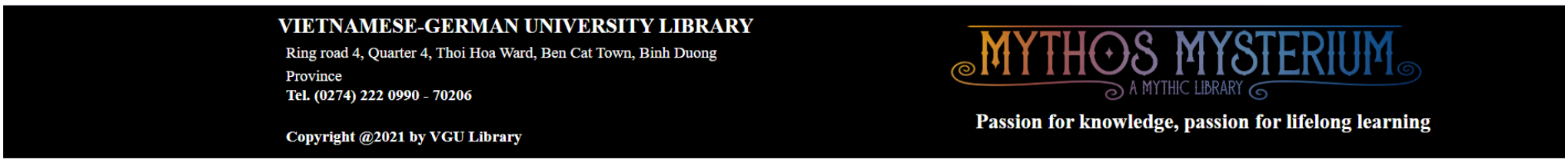
```
useEffect(() => {
    const fetchTopBooks = async () => {
    fetchTopBooks();
  }, []);
```

The `useEffect` hook in React to fetch the top books from an API and update the state with the fetched data.

- `fetchTopBooks`: An asynchronous function that fetches data from the API, parses the response as JSON, and sets the state with the fetched data.

Carousel Configuration

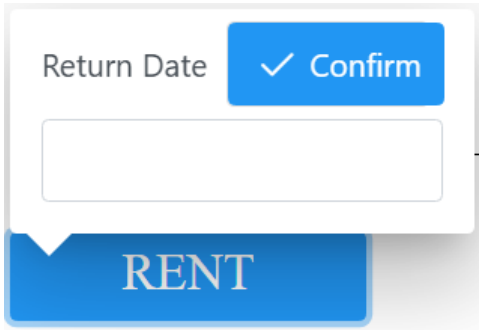
```
        <img src={logoimage} alt="" />
    </div>
```

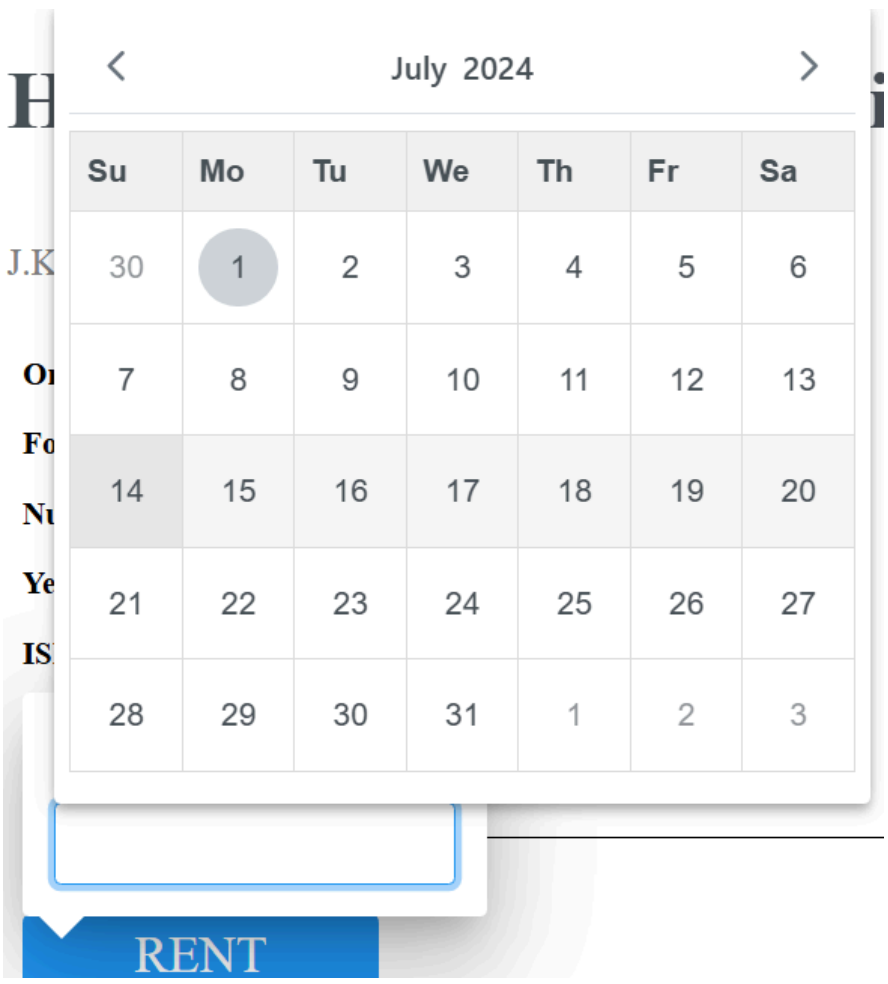


This `Contact.jsx` file defines a React functional component named Contact that renders a contact information section for the Vietnamese-German University Library. It displays the library's name, address, telephone number, copyright notice, and a slogan about passion for knowledge and lifelong learning. Additionally, it includes an image, likely the library's logo, sourced from the ../../Assets/Library name.png path. The component uses a CSS file named Contact.css for styling.

3. Rent Page

The `Rentbook.jsx` component is a React functional component designed for a book rental application. It incorporates various libraries and components to create a user-friendly interface for renting books. The component is structured to include book details, rental options, and a calendar for selecting a return date.





Libraries and Frameworks Used

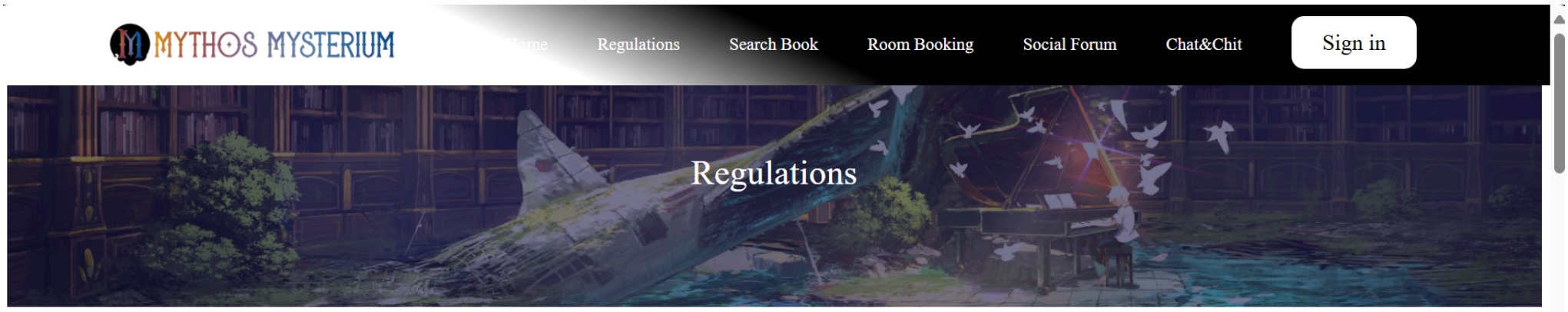
- *React*: Utilized for building the component with hooks such as `useState` and `useRef`.
- *Swiper*: Used for creating carousels, though its direct usage within this component isn't visible, implying it might be used in child components like `ImgCarousel`.
- *PrimeReact*: Provides UI components like `Calendar`, `OverlayPanel`, `Button`, `Toast`, and `ConfirmPopup` for rich interactions and UI elements.
- *React Router Dom*: Uses `useParams` to retrieve the `bookId` from the URL, indicating a dynamic page based on the book selected.

Component Structure

- *Header and Carousel*: The component starts with rendering an image carousel (`ImgCarousel`) and a header (`Header`), likely for navigation and showcasing book images.
- *Book Details*: Displays the title, author, and various attributes of the book such as format, number of pages, year of issue, ISBN, and circulation. It uses a mix of static and dynamic content, with placeholders for dynamic data.
- _Rental Section: Includes a button to initiate the rental process, which upon clicking, reveals an overlay panel with a calendar for selecting a return date and a confirm button to proceed with the rental.
- *Toast Notifications*: Utilizes `Toast` for feedback on successful rental, login requirement, or errors during the rental process.
- *Confirm Popup*: A confirmation dialog is used to confirm the rental action before proceeding.

4. Regulation page

The `regulation.jsx` file is a React component that serves as a page for displaying library regulations and policies. It integrates several components and utilizes CSS for styling. The component is structured to provide a comprehensive overview of the library's rules, including general regulations, circulation service details, and penalty and compensation policies.



Mythos Mysterium Regulations

In order to effectively manage library operations, and to enable open knowledge sharing and fair access to library materials, the Presidential Board has issued “Library Regulations” and “Policies of Anti-Plagiarism” for your reference. Please abide by these regulations.

Library Regulation

I. GENERAL REGULATIONS

1. Opening hours:
    - Mondays – Fridays: 8:30 am – 10:00 pm
    - Saturdays: 9:00 am – 6:00 pm.
2. Always bring your ID cards with you when using the library. Do not use other user's ones.
3. Keep your belongings in lockers and take your valuable things with you. Return the locker key in its original position after use.
4. Place in-house used books, newspapers and journals on the sorting shelves after reading.
5. Circulation service:
    - Borrowing:
        - Students and Administrative staff: 3 books/ week
        - PhD students and Researchers: 3 books/ 2 weeks
        - Lecturers and RTA: 3 books/ 3 weeks
        - Interns, exchange students and probationers: reading in the library and borrowing up to two books/ week
        - External users: reading in the library and borrowing up to two books/ week with the required bond
    - Renewing: 1 week for all entitled users if the books are available
    - Returning & renewing: use Selfcheck machines in the library. For renewing, please renew your borrowed books online via “Sign in” button on the website.
6. Use the library materials and facilities with care.
7. Keep quiet, clean and order.
8. Smoking and foods with smell should be outside the library. Soft drinks are not encouraged.
9. Please acknowledge the copyright and intellectual property laws. You are responsible for legal issues if you make any copyright infringements.

II. PENALTY AND COMPENSATION POLICIES

No.	VIOLATED ACTIONS	PENALTY
1	Overdue loan	◦ 1-14 days: a fine of VND 5,000/book/day ◦ 15 – 30 days: a fine of VND 10,000/book/day ◦ Over 1 month: a fine of VND 10,000/book/day and deactivate the borrowing

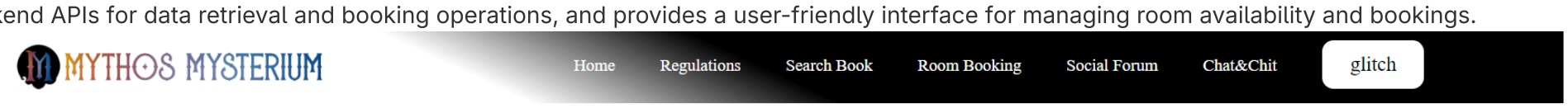
Component Composition

- *Imports and Dependencies*: The component imports React, a custom template component (`Templatexdd`), a `Header` component, a `Final` component, a `Contact` component, and a CSS file for styling.
- *Structure and Layout*: The component returns a structured layout that includes a header, a template component with an image and overview text, detailed library regulations, a penalty table, and ends with a final section and contact information.
- *Content and Data Presentation*: The main content is divided into sections detailing the library's regulations, including opening hours, ID card policies, borrowing rules, and penalties for various infringements. This information is presented in an ordered list and a table for clarity.

5. `RoomBooking` Component

Overview

The `RoomBooking` component is a React-based module designed to facilitate room bookings within an application. It manages various states, interacts with backend APIs for data retrieval and booking operations, and provides a user-friendly interface for managing room availability and bookings.



Time	101	201	202	203	301	302
07:00 - 09:00	Available	Available	Available	Available	Available	Available
09:00 - 11:00	Available	Available	Available	Available	Available	Available
11:00 - 13:00	Available	Available	Available	Available	Available	Available
13:00 - 15:00	Available	Available	Available	Available	Available	Available
15:00 - 17:00	Booked by Nhan Le Trong	Available	Available	Available	Available	Available
17:00 - 19:00	Available	Available	Available	Available	Available	Available
19:00 - 21:00	Available	Available	Available	Available	Available	Available
21:00 - 23:00	Available	Available	Available	Available	Available	Available

---

Component Overview

1. **State Management:**
   - `useState` hooks are used to manage various states such as `rooms`, `bookings`, `showModal`, `selectedTime`, `selectedRoom`, `name`, and `UserID`.
   - `rooms`: Stores the list of rooms available for booking.
   - `bookings`: Initially set with predefined time slots for booking.
   - `showModal`, `selectedTime`, `selectedRoom`: Manage the modal state for booking.
   - `name`: Holds the user's name input.
   - `UserID`: Stores the ID of the current user fetched from the backend.

```
import React, { useState, useEffect } from "react";

const RoomBooking = () => {
  const [rooms, setRooms] = useState([]);
  const [bookings, setBookings] = useState([
    { time: "07:00 - 09:00" },
    { time: "09:00 - 11:00" },
    { time: "11:00 - 13:00" },
    { time: "13:00 - 15:00" },
    { time: "15:00 - 17:00" },
    { time: "17:00 - 19:00" },
    { time: "19:00 - 21:00" },
    { time: "21:00 - 23:00"}
  ]);

  const [showModal, setShowModal] = useState(false);
  const [selectedTime, setSelectedTime] = useState("");
  const [selectedRoom, setSelectedRoom] = useState("");
  const [name, setName] = useState("");
  const [UserID, setUser] = useState(null);

  const accessToken = localStorage.getItem("accessToken");
  const now = new Date();
  const date = now.getFullYear() + "-" + (now.getMonth() + 1) + "-" + now.getDate();

  // Fetch user info and set user ID
  useEffect(() => {
    fetch("http://localhost:6868/api/user/info", {
      headers: {
        'x-access-token': accessToken
```

```
      }
    })
    .then(response => response.json())
    .then(data => setUser(data.id))
    .catch(error => console.error('Error fetching user:', error));
  }, []);

  // Fetch rooms
  useEffect(() => {
    fetch("http://localhost:6868/api/rooms")
      .then((response) => response.json())
      .then((data) => setRooms(data))
      .catch((error) => console.error("Error fetching rooms:", error));
  }, []);

  // Fetch bookings for the current date
  useEffect(() => {
    fetch(`http://localhost:6868/api/roombookings/date/${date}`)
      .then((response) => response.json())
      .then(async (data) => {
        console.log("Bookings:", data);
        let updatedBookings = [];
        for (const booking of bookings) {
          for (const roombooking of data) {
            const startTime = new Date(roombooking.StartTime);
            const endTime = new Date(roombooking.EndTime);
            const startHour = startTime.getHours();
            const endHour = endTime.getHours();
            const time = `${startHour}:00 - ${endHour}:00`;
            console.log("Time:", time);
            if (time === booking.time) {
              let userName = await displayUserName(roombooking.ID_User);
              booking[roombooking.ID_Room] = `Booked by ${userName}`;
              booking.booked = true;
            }
          }
          updatedBookings.push(booking);
        }
        setBookings(updatedBookings);
      })
      .catch((error) => console.error("Error fetching bookings:", error));
  }, []);

  return (
  );
};


export default RoomBooking;
```

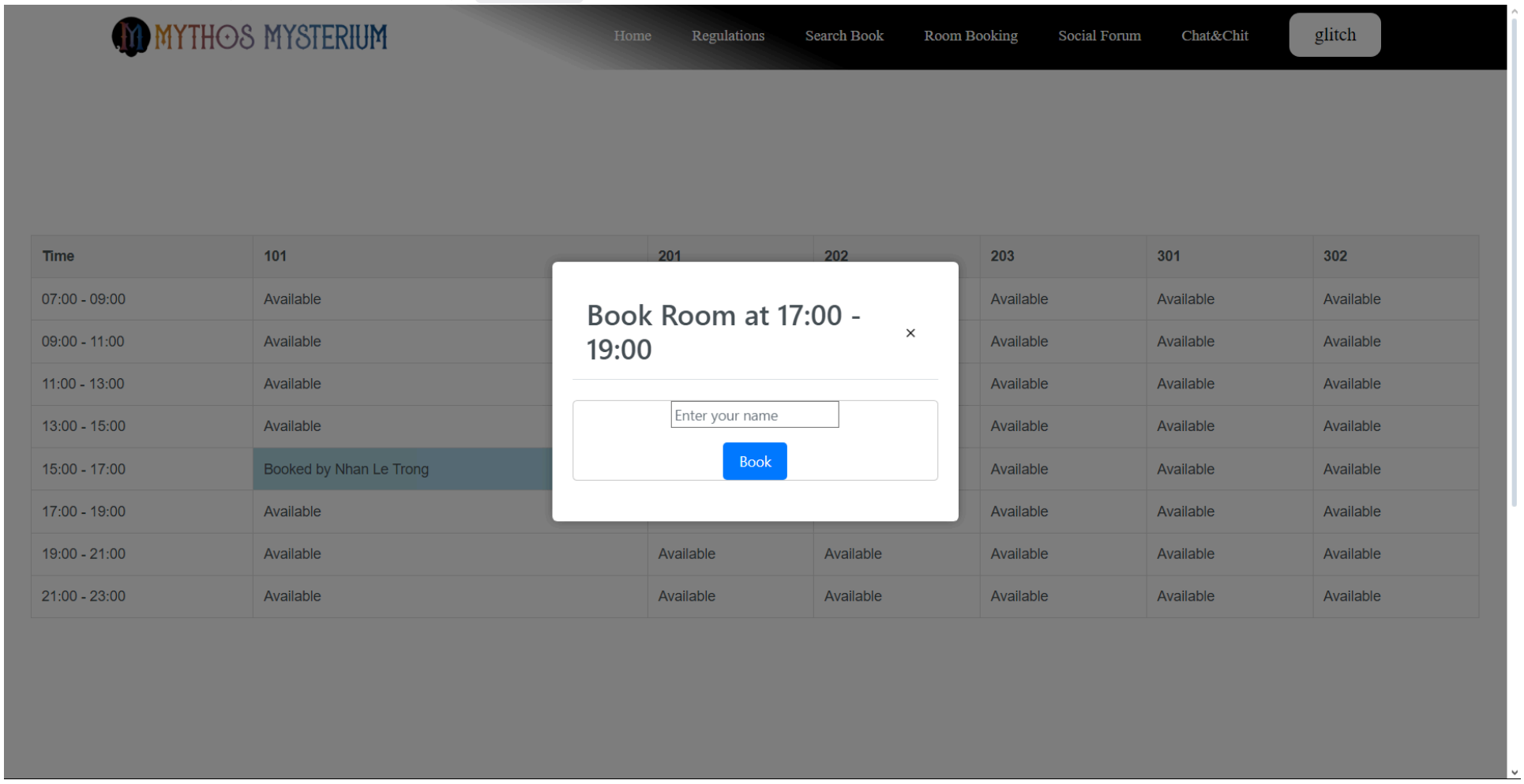
2. **Data Fetching:**
   - Uses `useEffect` hooks to fetch data from APIs (`/api/rooms`, `/api/roombookings/date/${date}`, `/api/user/info`).
   - Room data is fetched initially to populate `rooms`.
   - Bookings data for the current date (`date`) is fetched to check for existing room bookings and update `bookings` accordingly.
   - User information is fetched to get the current user's ID (`UserID`) for booking purposes.

```
import axios from "axios";

const RoomBooking = () => {
  const accessToken = localStorage.getItem("accessToken");

  const fetchUserName = async (token) => {
    try {
      const response = await axios.get("http://localhost:6868/api/user/list", {
        headers: {
          "x-access-token": token,
        },
      });
```

```
    return response.data;
  } catch (error) {
    console.error("Error fetching user:", error);
    return [];
  }
};

const findUserName = async (ID) => {
  const users = await fetchUserName(accessToken);
  const user = users.find((user) => user.id === ID);
  return `${user.fname} ${user.lname}`;
};

async function displayUserName(userID) {
  try {
    const userName = await findUserName(userID);
    return userName;
  } catch (error) {
    console.error("Failed to fetch user name:", error);
  }
}

// Fetch user info and set user ID
useEffect(() => {
  fetch("http://localhost:6868/api/user/info", {
    headers: {
      'x-access-token': accessToken
    }
  })
  .then(response => response.json())
  .then(data => setUser(data.id))
  .catch(error => console.error('Error fetching user:', error));
}, []);

// Fetch rooms
useEffect(() => {
  fetch("http://localhost:6868/api/rooms")
    .then((response) => response.json())
    .then((data) => setRooms(data))
    .catch((error) => console.error("Error fetching rooms:", error));
}, []);

// Fetch bookings for the current date
useEffect(() => {
  fetch(`http://localhost:6868/api/roombookings/date/${date}`)
    .then((response) => response.json())
    .then(async (data) => {
      console.log("Bookings:", data);
      let updatedBookings = [];
      for (const booking of bookings) {
        for (const roombooking of data) {
          const startTime = new Date(roombooking.StartTime);
          const endTime = new Date(roombooking.EndTime);
          const startHour = startTime.getHours();
          const endHour = endTime.getHours();
          const time = `${startHour}:00 - ${endHour}:00`;
          console.log("Time:", time);
          if (time === booking.time) {
            let userName = await displayUserName(roombooking.ID_User);
            booking[roombooking.ID_Room] = `Booked by ${userName}`;
            booking.booked = true;
          }
        }
        updatedBookings.push(booking);
      }
      setBookings(updatedBookings);
    })
```

```
      .catch((error) => console.error("Error fetching bookings:", error));
  }, []);

  return (
  );
};

export default RoomBooking;
```

3. **User Authentication and Display Name:**
   - Uses an access token (`accessToken`) stored in localStorage for API authentication.
   - Fetches user information (`api/user/list`) to display the user's full name (`fname` + `lname`) when a room is booked.
4. **Booking Logic:**
   - Clicking on an available time slot (`BookingCell`) triggers a modal (`showModal`) to enter the user's name.
   - Checks if the room and time slot are available (`handleBookingSubmit`) and updates the UI accordingly.
   - Posts the booking details to the backend (`/api/roombookings/`) and reloads the page (`window.location.reload()`) to reflect changes.
5. **UI Rendering:**
   - Renders a table where each row represents a time slot (`bookings`) and each column represents a room (`rooms`).
   - Colors cells differently based on availability (`isBooked` state).



Areas of Improvement

1. **Error Handling:**
   - Improve error handling in data fetching (`fetchUserName`, API calls) to provide better user feedback or fallback mechanisms.
2. **Performance:**
   - Consider optimizing data fetching and rendering processes, especially if dealing with large datasets or frequent updates.
3. **State Management:**
   - Ensure state updates (`setBookings`, `setRooms`) are handled correctly to avoid unnecessary re-renders or inconsistencies.
4. **Security:**
   - Validate and sanitize user inputs, especially when interacting with APIs (`handleBookingSubmit`).
5. **User Experience:**
   - Enhance UI/UX, such as providing loading indicators during data fetching or feedback for successful bookings.

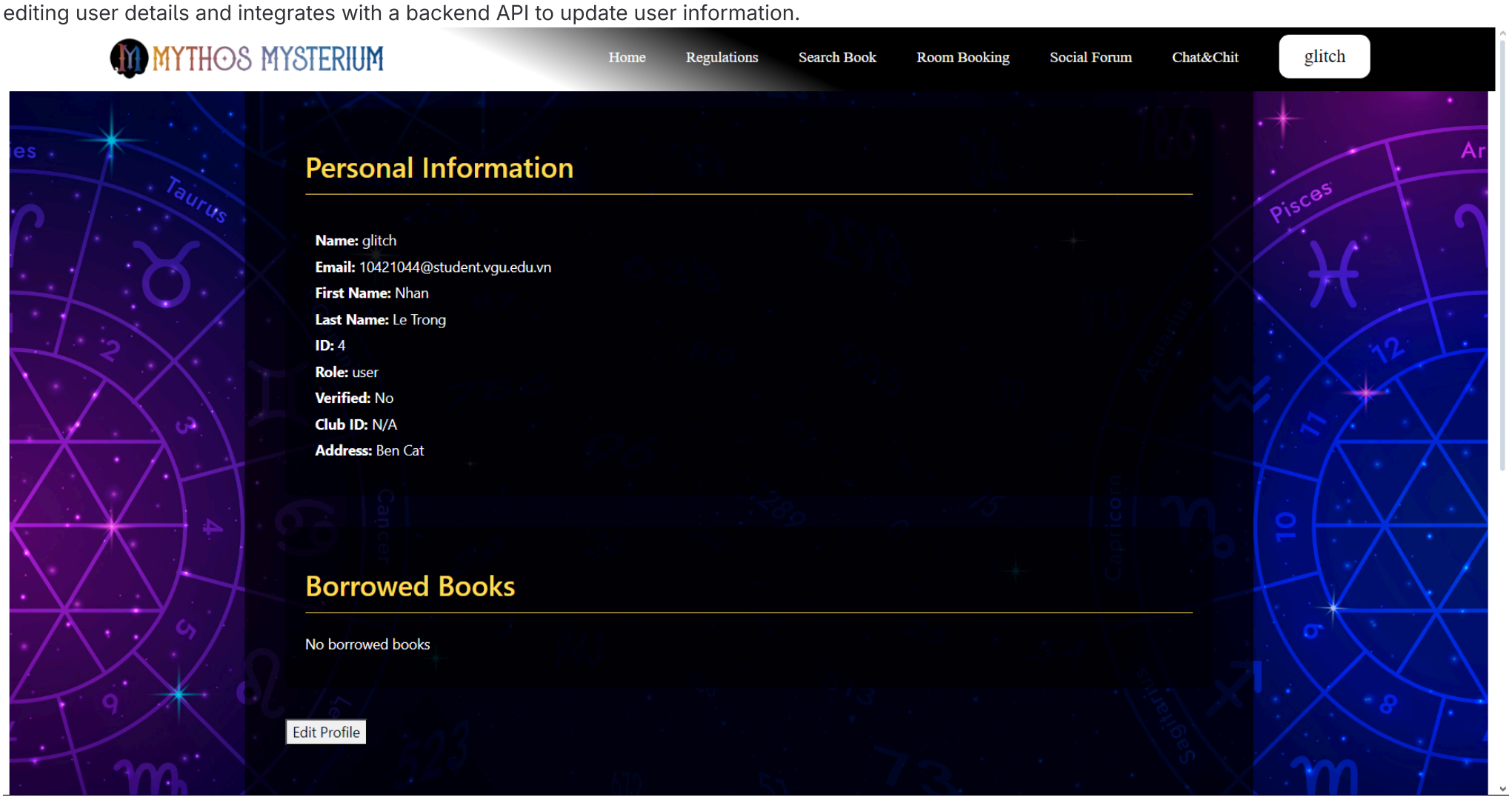
Conclusion

The `RoomBooking` component effectively manages room bookings through integration with backend APIs and provides a responsive user interface for interacting with room availability. By addressing areas for improvement, such as error handling, performance optimization, security enhancements, and code modularity, the component can be further refined to deliver a robust and user-friendly booking experience within the application.

6. `UserProfile` Component

Overview

The `UserProfile` component is a React functional component that displays and allows the editing of user profile information. It includes a modal for editing user details and integrates with a backend API to update user information.



Dependencies

- **React**: For building the user interface.
- **useState**: React hook for managing state within the component.
- **Modal**: From `react-modal`, used to display the edit profile form in a modal dialog.
- **Axios**: For making HTTP requests to the backend API.
- **CSS**: The component imports a CSS file `UserProfile.css` for styling and includes child components for the header and footer sections.

Component Structure

1. **State Management**:
   - `showEdit`: A boolean state to control the visibility of the edit profile modal.
   - `editedUser`: An object state to store the edited user details.

```
import React, { useState } from "react";
import Modal from "react-modal";

const UserProfile = ({ user }) => {
  const [showEdit, setShowEdit] = useState(false);
  const [editedUser, setEditedUser] = useState({ ...user });

  // Other parts of the component
};
```

2. **Event Handlers**:
   - `handleSaveProfile`: Makes an API call to update the user profile. It sends a PUT request with the edited user data and an authorization token from `localStorage`.
   - `handleInputChange`: Updates the `editedUser` state as the user types in the input fields in the modal.

```
const handleSaveProfile = () => {
  const accessToken = localStorage.getItem("accessToken");
  axios
    .put(`http://localhost:6868/api/user/update/${editedUser.ID}`, editedUser, {
      headers: {
```

```
        "x-access-token": accessToken,
      },
    })
    .then((response) => {
      setEditedUser(response.data);
      setShowEdit(false);
      window.location.reload();
    })
    .catch((error) => {
      console.error("Error editing user:", error);
    });
};

const handleInputChange = (e) => {
  const { name, value } = e.target;
  setEditedUser({
    ...editedUser,
    [name]: value,
  });
};
```

3. **JSX Elements**:
   - **Header**: A child component for the page header.
   - **Profile Information**: Displays the user's personal information.
   - **Borrowed Books**: Lists the books the user has borrowed.
   - **Edit Profile Button**: Opens the modal to edit user profile.
   - **Footer**: Includes the `Final` and `Contact` components for the page footer.
   - **Modal**: Contains a form for editing the user profile.

Key Functionalities

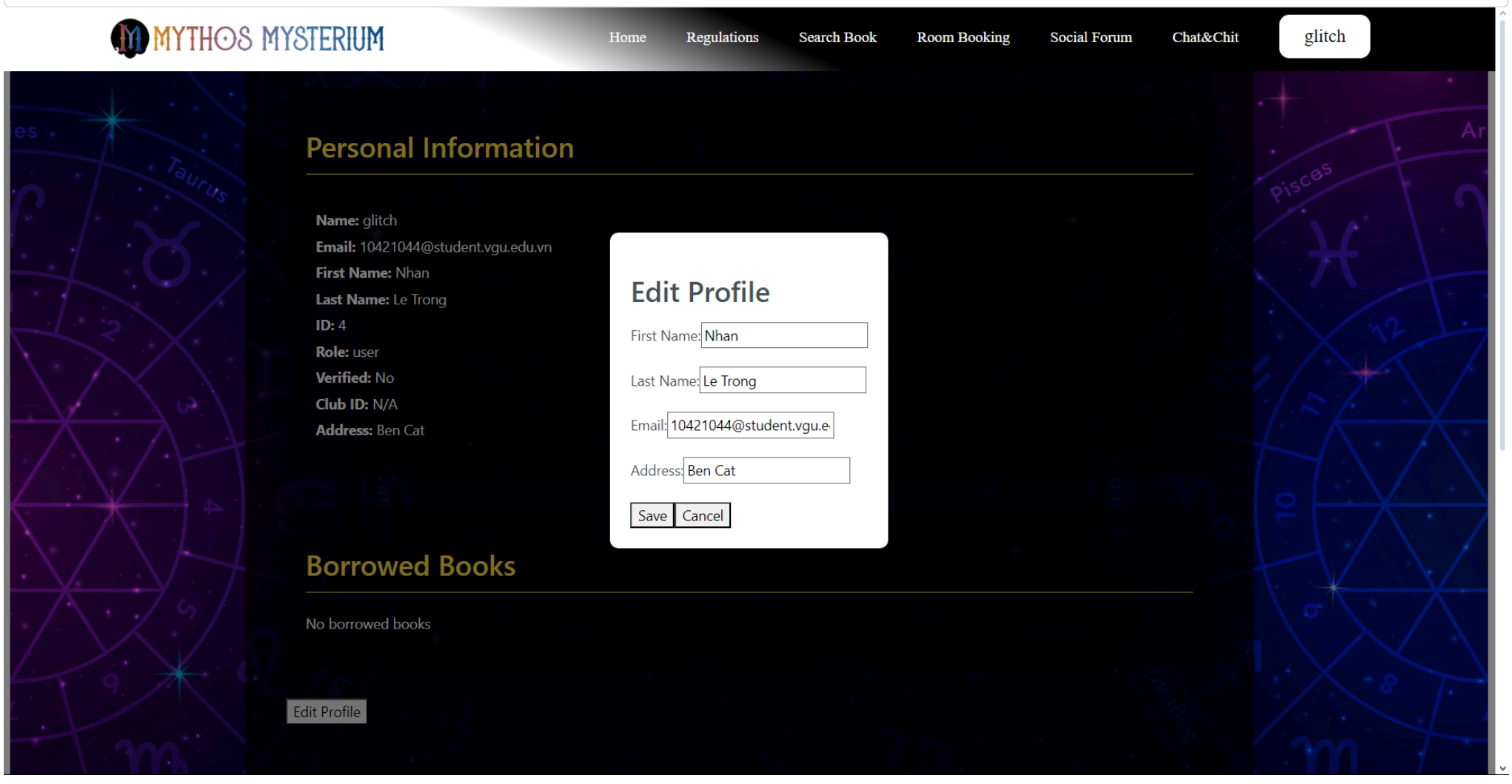
1. **Displaying User Information**:
   - The component displays various user attributes such as name, email, first name, last name, ID, role, verification status, club ID, and address.
   - It conditionally renders information if available, otherwise displays "N/A".

```
<div className="profile-section">
  <h2>Personal Information</h2>
  <div className="personal-info">
    <div className="info-text">
      <p><strong>Name:</strong> {user.name || "N/A"}</p>
      <p><strong>Email:</strong> {editedUser.email || "N/A"}</p>
      <p><strong>First Name:</strong> {editedUser.fname || "N/A"}</p>
      <p><strong>Last Name:</strong> {editedUser.lname || "N/A"}</p>
      <p><strong>ID:</strong> {user.ID || "N/A"}</p>
      <p><strong>Role:</strong> {user.role || "N/A"}</p>
      <p><strong>Verified:</strong> {user.isVerified ? "Yes" : "No"}</p>
      <p><strong>Club ID:</strong> {user.club_id || "N/A"}</p>
      <p><strong>Address:</strong> {editedUser.address || "N/A"}</p>
    </div>
  </div>
</div>
```

2. **Editing User Profile**:
   - When the user clicks the "Edit Profile" button, the modal opens, showing a form pre-filled with the current user information.
   - The modal contains input fields for first name, last name, email, and address.
   - The `handleInputChange` function updates the state with the values entered by the user.
   - The `handleSaveProfile` function sends a PUT request to update the user profile and refreshes the page upon success.

```
<Modal
  isOpen={showEdit}
  onRequestClose={() => setShowEdit(false)}
  contentLabel="Edit Profile"
  className="modalUP"
  overlayClassName="overlayUP"
>
  <h2>Edit Profile</h2>
  <form>
```

```
      <div className="form-group">
        <label>First Name:</label>
        <input
          type="text"
          name="fname"
          value={editedUser.fname || ""}
          onChange={handleInputChange}
        />
      </div>
      <div className="form-group">
        <label>Last Name:</label>
        <input
          type="text"
          name="lname"
          value={editedUser.lname || ""}
          onChange={handleInputChange}
        />
      </div>
      <div className="form-group">
        <label>Email:</label>
        <input
          type="email"
          name="email"
          value={editedUser.email || ""}
          onChange={handleInputChange}
        />
      </div>
      <div className="form-group">
        <label>Address:</label>
        <input
          type="text"
          name="address"
          value={editedUser.address || ""}
          onChange={handleInputChange}
        />
      </div>
      <div className="modal-buttons">
        <button type="button" onClick={handleSaveProfile}>Save</button>
        <button type="button" onClick={() => setShowEdit(false)}>Cancel</button>
      </div>
    </form>
  </Modal>
```



3. **Handling Borrowed Books**:
   - The component displays a list of books borrowed by the user, including the book's image, title, author, and due date.
   - If no books are borrowed, it displays a message indicating this.

```
<div className="profile-section">
  <h2>Borrowed Books</h2>
  <div className="book-list">
    {user.borrowedBooks && user.borrowedBooks.length > 0 ? (
      user.borrowedBooks.map((book, index) => (
        <div key={index} className="book-item">
          <img src={book.image || "https://via.placeholder.com/50"} alt={book.title} className="book-image" />
          <div className="book-details">
            <p><strong>Title:</strong> {book.title || "N/A"}</p>
            <p><strong>Author:</strong> {book.author || "N/A"}</p>
            <p><strong>Due Date:</strong> {book.dueDate || "N/A"}</p>
          </div>
        </div>
      ))
    ) : (
      <p>No borrowed books</p>
    )}
  </div>
</div>
```

CSS and Styling

- The component applies styles from `UserProfile.css` to various elements, ensuring a consistent and visually appealing layout.
- The modal is styled with `modalUP` and `overlayUP` classes for its content and overlay.

API Integration

- The component uses Axios to interact with the backend API.
- It includes an authorization header with a token retrieved from `localStorage` for authentication purposes.

Potential Enhancements

1. **Error Handling**: Add more robust error handling to provide user feedback in case of API call failures.
2. **Loading State**: Implement a loading indicator while the API call is in progress to enhance the user experience.
3. **Form Validation**: Add client-side validation to ensure the input fields are correctly filled out before making the API call.

Conclusion

The `UserProfile` component in React is a well-structured and functional piece of code that efficiently manages user profile data and interactions. By leveraging React hooks for state management, Axios for API communication, and `react-modal` for the modal dialog, the component provides a seamless user experience for viewing and editing profile information.

Key Strengths:

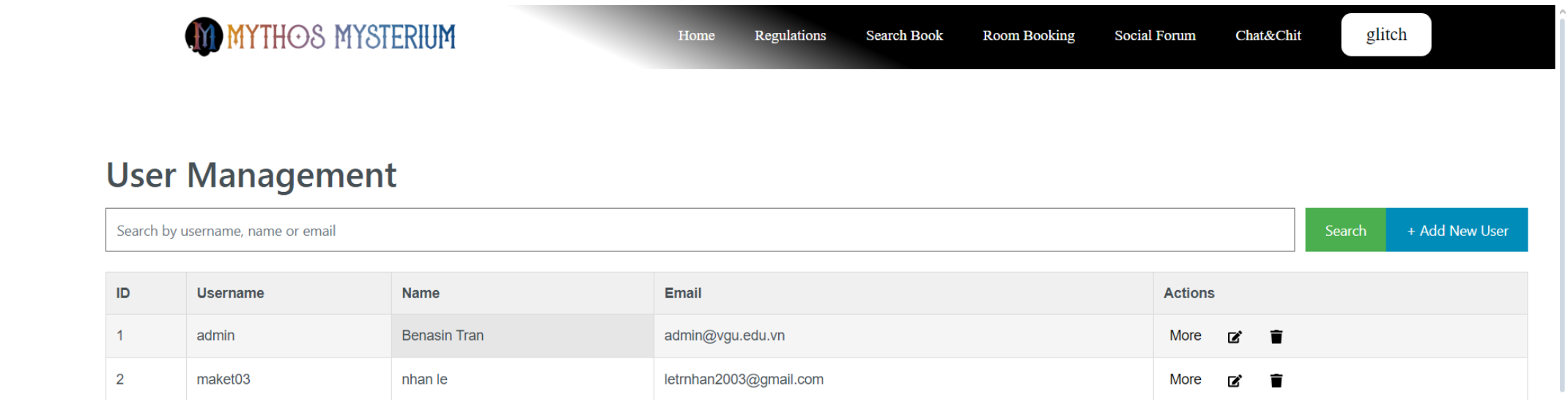
1. **Clear Structure**: The component is logically organized, separating user information display, borrowed books listing, and the edit profile functionality.
2. **State Management**: Effective use of `useState` hooks to manage component state, making the component responsive to user actions.
3. **API Integration**: Utilizes Axios for secure and efficient communication with the backend API, ensuring that user profile updates are reflected in real-time.
4. **User Interaction**: The edit modal is user-friendly, with pre-filled input fields and clear buttons for saving changes or cancelling the operation.

Overall, the `UserProfile` component demonstrates good practices in React development, with a focus on user experience and efficient data management. By addressing the areas for improvement, the component can become even more robust and user-friendly.

7. Report on User Management Component

Overview

The `UserManagement` component is a React functional component designed to manage user-related operations such as searching, adding, editing, and deleting users. It utilizes several React hooks (`useState` and `useEffect`), Axios for API requests, and Modal for managing modal dialogs. The component also includes various UI elements for interacting with user data, including search functionality, a user table, and pagination.



Key Functionalities

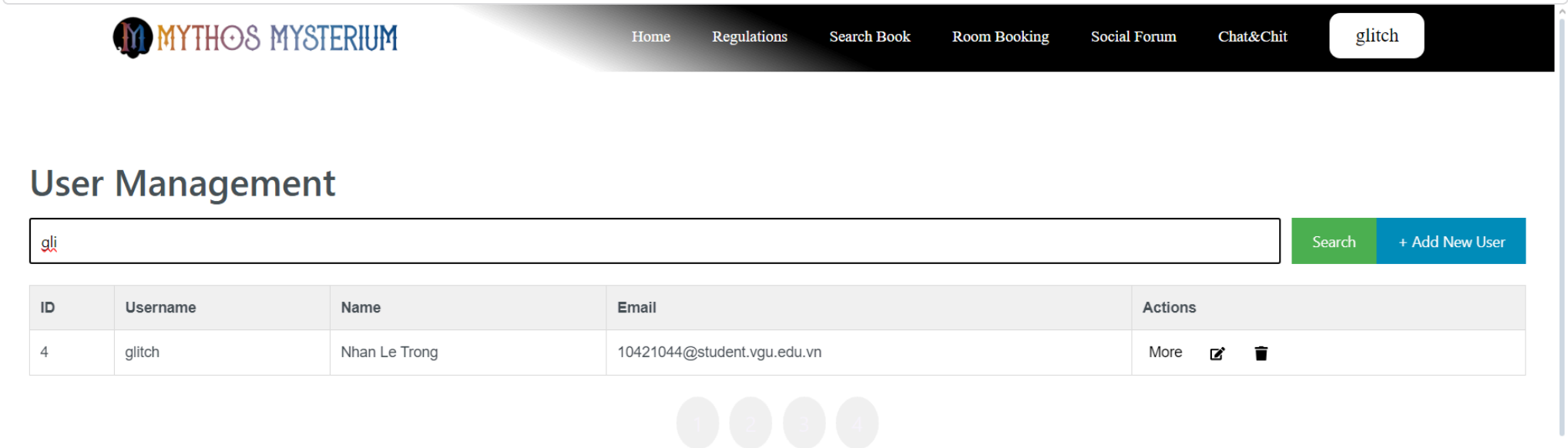
1. **Fetching Users**:
   - The `useEffect` hook is used to fetch the user list from the API when the component mounts.
   - Axios is used to send a GET request to the server with an access token retrieved from localStorage.

```
useEffect(() => {
  const fetchUsers = async () => {
    try {
      const accessToken = localStorage.getItem("accessToken");
      if (!accessToken) {
        console.error("Access token not found in localStorage");
        return;
      }
      const response = await axios.get("http://localhost:6868/api/user/list", {
        headers: {
          "x-access-token": accessToken,
        },
      });

      setUsers(response.data);
    } catch (error) {
      console.error("Error fetching user data:", error);
    }
  };

  fetchUsers();
}, []);
```

2. **Search Functionality**:
   - The search input allows users to filter the user list by username, first name, last name, or email.
   - The `filteredUsers` array is derived from the `users` state and updated based on the search query.

```
const handleSearchChange = (event) => {
  setSearch(event.target.value);
};

const filteredUsers = users.filter(
  (user) =>
    user.username.toLowerCase().includes(search.toLowerCase()) ||
    user.fname.toLowerCase().includes(search.toLowerCase()) ||
    user.lname.toLowerCase().includes(search.toLowerCase()) ||
```

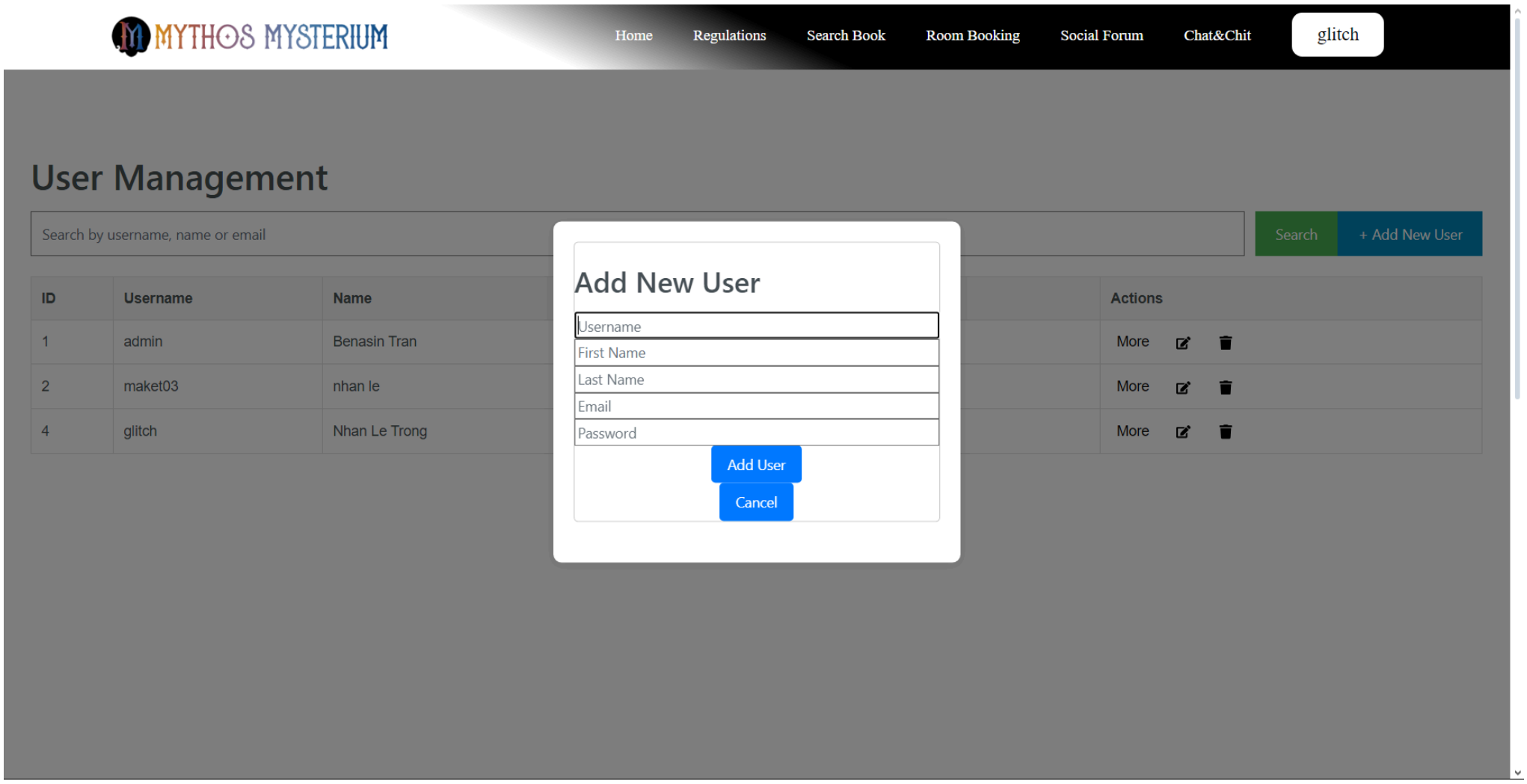
```
    user.email.toLowerCase().includes(search.toLowerCase())
);
```



3. **Adding a New User**:

- A modal dialog is used for inputting new user details.
- The `handleAddUserSubmit` function sends a POST request to the API to create a new user, and the user list is updated upon successful creation.

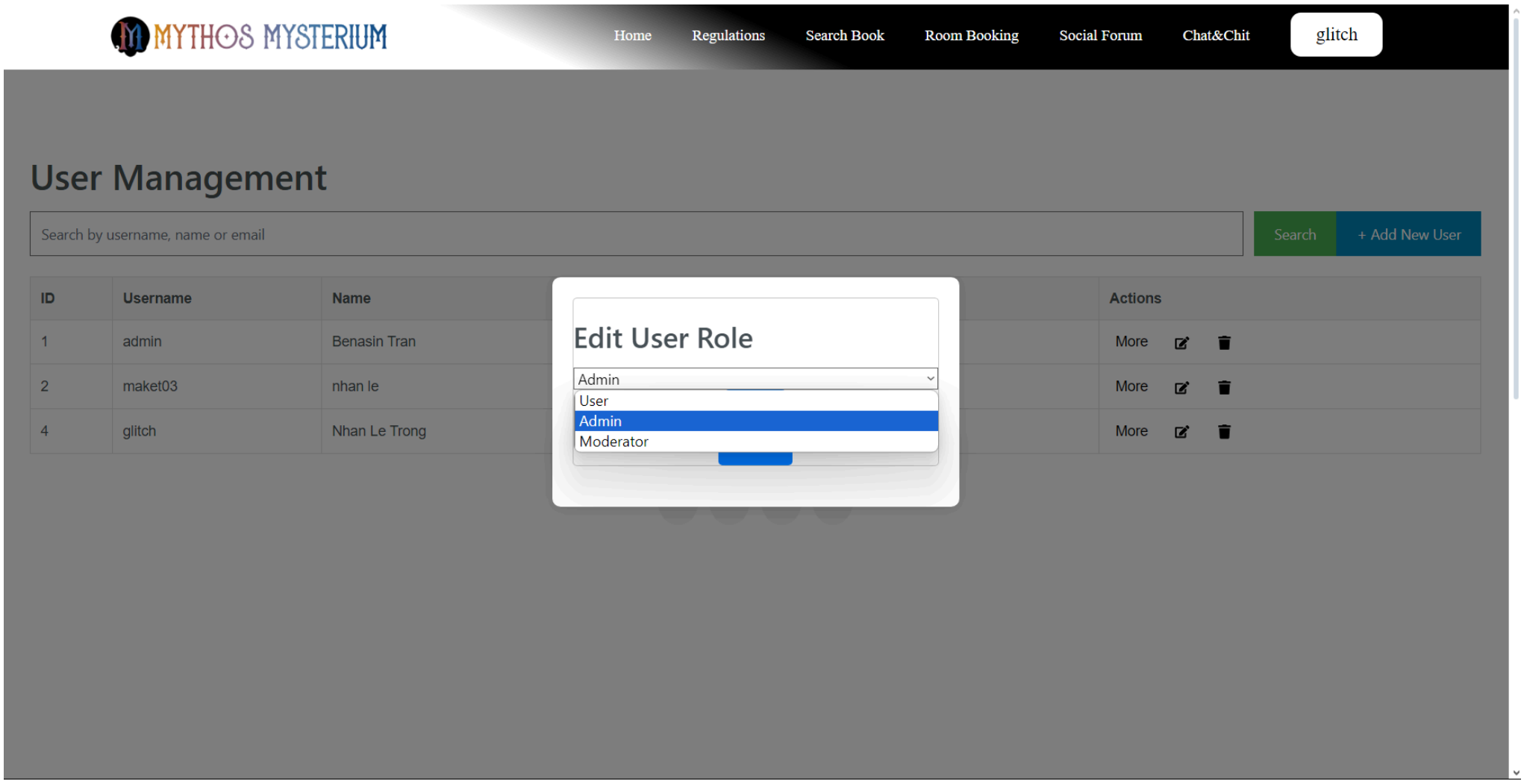
```
const handleAddUserChange = (event) => {
  const { name, value } = event.target;
  setNewUser({ ...newUser, [name]: value });
};

const handleAddUserSubmit = () => {
  const accessToken = localStorage.getItem("accessToken");
  axios
    .post("http://localhost:6868/api/auth/register", newUser, {
      headers: {
        "x-access-token": accessToken,
      },
    })
    .then((response) => {
      setUsers([...users, response.data]);
      setShowAddUserModal(false);
      setNewUser({
        username: "",
        fname: "",
        lname: "",
        email: "",
        password: "",
        role: "user",
      });
    })
    .catch((error) => {
      console.error("Error adding user:", error);
    });
};
```



4. **Editing User Role**:

- Another modal dialog is used for editing a user's role.
- The `handleEditUserRole` function sends a PUT request to update the selected user's role, and the user list is updated upon successful update.

```
const handleEditUserRole = () => {
  const accessToken = localStorage.getItem("accessToken");
  axios
    .put(`http://localhost:6868/api/user/update/${selectedUser.id}`, { role: editUserRole }, {
      headers: {
        "x-access-token": accessToken,
      },
    })
    .then((response) => {
      setUsers(users.map((user) => (user.id === selectedUser.id ? response.data : user)));
      setShowEditUserModal(false);
      setShowUserDetailsModal(false);
      setSelectedUser(null);
    })
    .catch((error) => {
      console.error("Error updating user role:", error);
    });
};
```



5. **Deleting a User**:

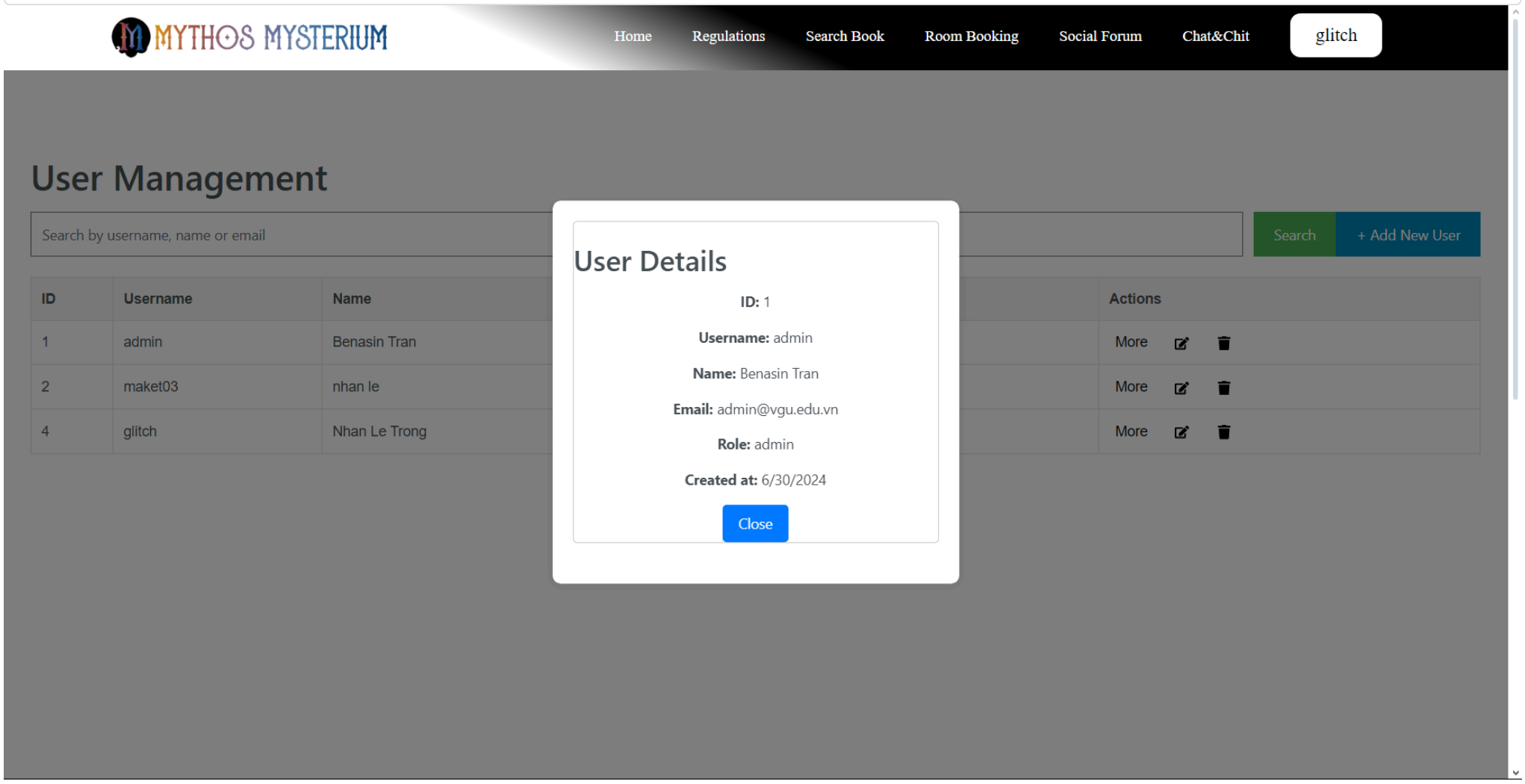
- The `handleDeleteUser` function sends a DELETE request to the API to remove a user by ID.
- Upon successful deletion, the user is removed from the user list.

```
const handleDeleteUser = (userId) => {
  const accessToken = localStorage.getItem("accessToken");
  axios
    .delete(`http://localhost:6868/api/user/delete/${userId}`, {
      headers: {
        "x-access-token": accessToken,
      },
    })
    .then(() => {
      setUsers(users.filter((user) => user.id !== userId));
    })
    .catch((error) => {
      console.error("Error deleting user:", error);
    });
};
```

6. **User Details Modal**:
    - A modal dialog displays detailed information about a selected user.
    - The details include user ID, username, name, email, role, and creation date.

```
<Modal
  isOpen={showUserDetailsModal}
  onRequestClose={() => { setShowUserDetailsModal(false); setSelectedUser(null); }}
  contentLabel="User Details Modal"
  className="modalUM"
  overlayClassName="overlayUM"
>
  <div className="modal-content">
    <h2>User Details</h2>
    {selectedUser && (
      <div className="details">
        <p>
          <strong>ID:</strong> {selectedUser.id}
        </p>
        <p>
          <strong>Username:</strong> {selectedUser.username}
        </p>
        <p>
```

```
          <strong>Name:</strong> {`${selectedUser.fname} ${selectedUser.lname}`}
        </p>
        <p>
          <strong>Email:</strong> {selectedUser.email}
        </p>
        <p>
          <strong>Role:</strong> {selectedUser.role}
        </p>
        <p>
          <strong>Created at:</strong> {new Date(selectedUser.createdAt).toLocaleDateString()}
        </p>
      </div>
    )}
    <button onClick={() => { setShowUserDetailsModal(false); setSelectedUser(null); }}>Close</button>
  </div>
</Modal>
```



State Management

The component maintains several state variables to handle user data and UI states:

- `users`: an array of user objects.
- `search`: a string for the search query.
- `showAddUserModal`, `showEditUserModal`, `showUserDetailsModal`: booleans to control the visibility of the respective modals.
- `selectedUser`: an object representing the currently selected user.
- `newUser`: an object for storing new user details during the addition process.
- `editUserRole`: a string for the role of the user being edited.

Handling User Interactions

- **Search Input**: Updates the `search` state and filters the `users` array based on the input value.
- **Add User Modal**: Collects new user data through controlled inputs and submits it to the API.
- **Edit User Modal**: Allows selection of a new role for the selected user and submits the updated role to the API.
- **Delete User**: Confirms deletion and removes the user from the list upon successful API request.

Error Handling

- Basic error handling is implemented using `console.error` to log errors from API requests.
- There is potential to improve error handling by displaying user-friendly error messages within the UI.

Accessibility

- The `Modal.setAppElement("#root")` function call ensures that the modals are accessible by setting the root element for the modal dialogs.

UI Components

- **Header**: A reusable component likely providing the main page header.
- **Final and Contact**: Reusable components providing page end content and contact information.

Conclusion

The `UserManagement` component effectively manages user data with functionalities for searching, adding, editing, and deleting users. It uses modals for user interactions and maintains state efficiently using React hooks. While it provides a solid foundation, there are opportunities for enhancing error handling, user feedback, and overall user experience.

IV.3. Back-end Implementation

db.config.js:

The `db.config.js` file is used to configure the database connection settings for your application. This configuration file typically includes details such as the database host, user credentials, database name, and other connection options. Below is a detailed summary of the file:

Configuration Properties:

1. **HOST**:
   - Description: The hostname or IP address of the database server.
   - Value: Retrieved from the environment variable `DB_HOST`.
2. **USER**:
   - Description: The username used to authenticate with the database.
   - Value: Retrieved from the environment variable `DB_USER`.
3. **PASSWORD**:
   - Description: The password used to authenticate with the database.
   - Value: Retrieved from the environment variable `DB_PASSWORD`.
4. **DB**:
   - Description: The name of the database to connect to.
   - Value: Retrieved from the environment variable `DB_NAME`.
5. **port**:
   - Description: The port number on which the database server is listening.
   - Value: Retrieved from the environment variable `DB_PORT`.
6. **dialect**:
   - Description: The type of database being used (e.g., MySQL).
   - Value: Set to `"mysql"`.
7. **dialectOptions**:
   - **charset**:
     - Description: The character set to use for the database connection.
     - Value: Set to `"utf8mb4"` to support a wide range of characters including emojis.
8. **define**:
   - **charset**:
     - Description: The character set to use for defining tables.
     - Value: Set to `"utf8mb4"`.
   - **collate**:
     - Description: The collation to use for defining tables, which specifies the rules for comparing and sorting strings.
     - Value: Set to `"utf8mb4_unicode_ci"`.
9. **pool**:
   - **max**:
     - Description: The maximum number of connections in the pool.
     - Value: Set to `5`.
   - **min**:
     - Description: The minimum number of connections in the pool.
     - Value: Set to `0`.
   - **acquire**:
     - Description: The maximum time, in milliseconds, that a connection can be idle before being released.
     - Value: Set to `30000`.
   - **idle**:
     - Description: The maximum time, in milliseconds, that a connection can be idle before being released.
     - Value: Set to `10000`.

Example Usage:

This configuration file is used by Sequelize (a Node.js ORM) to establish and manage database connections. The environment variables used in this file should be set in your environment or in a `.env` file.

```
module.exports = {
  HOST: process.env.DB_HOST,
```

```
  USER: process.env.DB_USER,
  PASSWORD: process.env.DB_PASSWORD,
  DB: process.env.DB_NAME,
  port: process.env.DB_PORT,
  dialect: "mysql",
  dialectOptions: {
      charset: "utf8mb4",
  },
  define: {
      charset: "utf8mb4",
      collate: "utf8mb4_unicode_ci",
  },
  pool: {
      max: 5,
      min: 0,
      acquire: 30000,
      idle: 10000
  }
};
```

This configuration ensures that the application can connect to a MySQL database using the specified settings and can handle database operations efficiently with a connection pool.

IV.3.1. Model Files

1. Book Model (book.model.js):

- **Fields**:
  - `id`: Primary key, auto-incremented integer.
  - `title`: String, not null.
  - `authors`: String, not null.
  - `description`: Text.
  - `edition`: String, default empty.
  - `format`: String.
  - `num_pages`: Integer.
  - `rating`: Float.
  - `rating_count`: Integer.
  - `review_count`: Integer.
  - `genres`: String.
  - `genre_list`: String.
  - `image_url`: String.
  - `Quote1`, `Quote2`, `Quote3`: Text fields for quotes.
  - `visited_times`: Integer, default 0.
  - `location`: String, default "Main Hall".
  - `amount`: Integer, default 100.
- **Configuration**:
  - Table name is explicitly set to 'Books'.
  - Charset and collation set to support UTF-8.

2. Room Model (room.model.js):

- **Fields**:
  - `ID_Room`: Primary key, integer.
  - `Location`: String.
  - `Availability`: String, to indicate if the room is available or not.
- **Configuration**:
  - Table name is explicitly set to 'Rooms'.

3. RoomBooking Model (roombooking.model.js):

- **Fields**:
  - `ID`: Primary key, auto-incremented integer.
  - `ID_Room`: Integer, references 'Rooms' table.
  - `ID_User`: Integer.
  - `StartTime`: Date.
  - `EndTime`: Date.
  - `Payment_method`: String.
- **Associations**:
  - Belongs to `Room` model with `ID_Room` as foreign key.

4. Token Model (token.model.js):

- **Fields**:
  - `id`: Primary key, auto-incremented integer.
  - `userId`: Integer, not null, references `users` table, with cascade on update and delete.
  - `token`: String.
- **Configuration**:
  - Timestamps enabled.

5. User Model (user.model.js):

- **Fields**:
  - `username`: String, unique, not null.
  - `fname`: String, not null.
  - `lname`: String, not null.
  - `email`: String, unique, not null.
  - `password`: String, not null.
  - `role`: String, default 'user', validated to be either 'user' or 'admin'.
  - `isVerified`: Boolean, default false.
  - `balance`: Integer.
  - `club_id`: Integer.
  - `address`: String.
  - `recent_search`: String.
  - `phone_number`: Integer.
- **Configuration**:
  - Defines validations and defaults for various fields.

6. Rent Model (rent.model.js):

- **Fields**:
  - `ID_Rent`: Primary key String, no autoIncrement and no allowNull.
  - `ID_Book`: Integer, references 'id' from Book Model.
  - `Issue_Date`: DATE.
  - `Return_Date`: DATE.
  - `ID_User`: Integer, references 'id' from 'users' table.
  - `late_fee`: Float, default 0.
  - `status`: String, default "returned".

7. Database Configuration (index.js):

- **Setup**:
  - Configures Sequelize with database connection details (DB name, user, password, host, dialect).
  - Pooling options for database connections.
- **Models**:
  - Imports and initializes all models (Book, Room, RoomBooking, Token, User, Rent).
  - Defines associations:
    - `User` has one `Token`.
    - `Token` belongs to `User`.
    - `Book` has one `Rent`.
    - `Rent` belongs to `Book`.
    - `User` has many `Rent`.
    - `Rent` belongs to `User`.
- **Synchronization**:
  - Syncs models to create or update database schema with `alter: true`.

IV.3.2. Controller Files (MVC Model)

1. admin.controller.js:

- **test**: Sends a simple response message to test admin functionality.

```
exports.test = async (req, res) => {
  res.status(200).send({ message: "Hello admin!" });
};
```

2. auth.controller.js:

- **register**: Validates email, hashes password, creates a user, generates a verification token, and sends a verification email.
- **login**: Validates email, checks if the user exists, compares passwords, generates JWT token if valid.

- **verifyEmail**: Verifies the user's email using a token, updates the user's verification status.

```
exports.register = (req, res) => { ... };
exports.login = async (req, res) => { ... };
exports.verifyEmail = async (req, res) => { ... };
```

3. book.controller.js:

- **createBooksFromJSON**: Reads JSON file, parses it, maps data to Book model, bulk creates books.
- **create**: Validates request, creates and saves a book.
- **createMultiple**: Bulk creates books from request body.
- **findAll/findOne**: Retrieves all books or a book by ID.
- **update/delete**: Updates or deletes a book by ID.
- **deleteAll**: Deletes all books.
- **findByAuthor/MinRating/Title/Genre**: Searches books based on author, rating, title, and genre.
- **findAllGenres/Authors/Titles**: Retrieves all unique genres, authors, and titles.
- **sortByRating/VisitedTimes**: Retrieves books sorted by rating and visited times.
- **count**: Retrieves the total number of books.
- **findBy**: Searches books based on multiple criteria.

```
exports.createBooksFromJSON = async (filePath) => { ... };
exports.create = (req, res) => { ... };
exports.createMultiple = (req, res) => { ... };
exports.findAll = (req, res) => { ... };
exports.findOne = (req, res) => { ... };
exports.update = (req, res) => { ... };
exports.delete = (req, res) => { ... };
exports.deleteAll = (req, res) => { ... };
exports.findByAuthor = (req, res) => { ... };
exports.findByMinRating = (req, res) => { ... };
exports.findByTitle = (req, res) => { ... };
exports.findByGenre = (req, res) => { ... };
exports.findAllGenres = (req, res) => { ... };
exports.findAllAuthors = (req, res) => { ... };
exports.findAllTitles = (req, res) => { ... };
exports.sortByRating = (req, res) => { ... };
exports.sortByVisitedTimes = (req, res) => { ... };
exports.count = (req, res) => { ... };
exports.findBy = (req, res) => { ... };
```

4. mailing.controller.js:

- **sendingMail**: Configures nodemailer, sends an email with specified details.

```
module.exports.sendingMail = async ({ from, to, subject, text }) => { ... };
```

5. response.js:

- **response**: Constructs and sends a JSON response with status, message, and values.

```
exports.response = (rs, message, status, values) => {
  let messages = {
    status: status || 200,
    message: message || null,
    values: values || null
  };
  return rs.json(messages);
};
```

6. room.controller.js:

- **createRoomsFromJSON**: Reads JSON file, parses it, maps data to Room model, bulk creates rooms.
- **create**: Validates request, creates and saves a room.
- **findAll/findOne**: Retrieves all rooms or a room by ID.
- **update/delete**: Updates or deletes a room by ID, handles future bookings if unavailable.
- **deleteAll**: Deletes all rooms.

- **findAllByAvailable/Unavailable**: Retrieves rooms by availability.
- **findByLocation**: Retrieves rooms by location.
- **findAllRoomNumbers/Locations**: Retrieves room numbers, unique locations.
- **count**: Retrieves total number of rooms.

```
exports.createRoomsFromJSON = async (filePath) => { ... };
exports.create = (req, res) => { ... };
exports.findAll = (req, res) => { ... };
exports.findOne = (req, res) => { ... };
exports.update = (req, res) => { ... };
exports.delete = (req, res) => { ... };
exports.deleteAll = (req, res) => { ... };
exports.findAllByAvailable = (req, res) => { ... };
exports.findAllByUnavailable = (req, res) => { ... };
exports.findByLocation = (req, res) => { ... };
exports.findAllRoomNumbers = (req, res) => { ... };
exports.findAllRoomLocations = (req, res) => { ... };
exports.count = async (req, res) => { ... };
```

7. roombooking.controller.js:

- **create**: Validates request, checks room availability, creates booking, sends confirmation email.
- **findAll/findOne**: Retrieves all bookings or a booking by ID.
- **update/delete**: Validates access, updates or deletes booking by ID, sends notification email.
- **deleteAll**: Deletes all bookings.
- **findByUser/Room/Date/StartEndRoom**: Searches bookings by user ID, room ID, date, start/end times.

```
exports.create = async (req, res) => { ... };
exports.findAll = (req, res) => { ... };
exports.findOne = (req, res) => { ... };
exports.update = async (req, res) => { ... };
exports.delete = async (req, res) => { ... };
exports.deleteAll = (req, res) => { ... };
exports.findByUser = (req, res) => { ... };
exports.findByRoom = (req, res) => { ... };
exports.findByDate = (req, res) => { ... };
exports.findByStartEndRoom = (req, res) => { ... };
```

8. user.controller.js:

- **info**: Retrieves current authenticated user's information.
- **list/count**: Retrieves list or count of all users.
- **delete/deleteAll**: Deletes a user by ID or all users.
- **deleteByUsername/Email**: Deletes users by username or email.
- **findByClubId**: Retrieves users by club ID.
- **sendMail/ToOne**: Sends email to all users or a specific user.
- **update**: Updates user information, sends notification email.

```
exports.info = async (req, res) => { ... };
exports.list = async (req, res) => { ... };
exports.count = async (req, res) => { ... };
exports.delete = async (req, res) => { ... };
exports.deleteAll = async (req, res) => { ... };
exports.deleteByUsername = async (req, res) => { ... };
exports.deleteByEmail = async (req, res) => { ... };
exports.findByClubId = async (req, res) => { ... };
exports.sendMail = async (req, res) => { ... };
exports.sendMailToOne = async (req, res) => { ... };
exports.update = async (req, res) => { ... };
```

9. rent.controller.js:

- **getDataBorrow**: Retrieves all rental records for a specified user.
- **getBorrowbyId**: Retrieves a specific rental record by `ID_Book`.
- **insertBorrow**: Handles borrowing books with given `ID_Book` or `title` by a user.
- **returnBook**: Handles returning books by updating rental records and book availability.

- **deleteRental**: Allows an admin to delete rental records of the user for returned books.

```
exports.getDataBorrow = (rq, rs) => { ... };
exports.getBorrowbyId = (rq, rs) => { ... };
exports.insertBorrow = (rq, rs) => { ... };
exports.returnBook = (rq, rs) => { ... };
exports.deleteRental = (rq, rs) => { ... };
```

Detailed Summary of authorize.js:

The `authorize.js` file is a middleware for handling authentication and authorization in your application. It uses JSON Web Tokens (JWT) to verify the identity of users and checks their roles to ensure they have permission to access certain resources.

Key Components:

1. **Importing Dependencies**:
   - The `jsonwebtoken` library is imported to handle JWT operations.

```
const jwt = require('jsonwebtoken');
```

2. **authorize Function**:
   - The main function `authorize` takes an optional parameter `roles`, which can be a single role string or an array of roles that are allowed to access a particular resource.
   - If `roles` is a string, it converts it into an array.

```
const authorize = (roles = []) => {
  if (typeof roles === 'string') {
    roles = [roles];
  }
```

3. **Middleware Functions**:
   - The `authorize` function returns an array of two middleware functions: one for authenticating the JWT token and another for authorizing based on user roles.

4. **JWT Authentication Middleware**:
   - This middleware function extracts the token from the `x-access-token` header.
   - If no token is provided, it responds with a `403 Forbidden` status.
   - It verifies the token using the secret key stored in `process.env.JWT_SECRET`.
   - If the token is invalid or expired, it responds with a `401 Unauthorized` status.
   - If the token is valid, it attaches the decoded token (which contains user information) to `req.user` and calls `next()` to pass control to the next middleware function.

```
(req, res, next) => {
  const token = req.headers['x-access-token'];
  if (!token) {
    return res.status(403).send({ message: "No token provided!" });
  }

  jwt.verify(token, process.env.JWT_SECRET, (err, decoded) => {
    if (err) {
      return res.status(401).send({ message: "Unauthorized!" });
    }
    req.user = decoded;
    next();
  });
},
```

5. **Role Authorization Middleware**:
   - This middleware function checks if the user's role (attached to `req.user` by the previous middleware) is included in the `roles` array.
   - If the user's role is not authorized, it responds with a `401 Unauthorized` status.

- If the user's role is authorized, it calls `next()` to pass control to the next middleware or the actual route handler.

```
(req, res, next) => {
  if (roles.length && !roles.includes(req.user.role)) {
    return res.status(401).json({ message: 'Unauthorized' });
  }
  next();
}
```

6. **Exporting the Middleware**:
   - The `authorize` function is exported as a module so that it can be used in other parts of the application, such as route definitions.

```
module.exports = authorize;
```

Example Usage:

In a route file, you might use the `authorize` middleware as follows:

```
const authorize = require('./authorize');

app.get('/admin', authorize('Admin'), (req, res) => {
  res.send('Hello, Admin!');
});

app.get('/user', authorize(['User', 'Admin']), (req, res) => {
  res.send('Hello, User or Admin!');
});
```

This configuration ensures that only users with the specified roles can access the protected routes, enhancing the security of your application.

Route Files

1. admin.routes.js:

- **Purpose**: Defines routes for admin-specific functionalities.
- **Routes**:
  - `GET /test`: Tests admin access, only accessible by users with the 'admin' role.
- **Middleware**:
  - Uses `authorize` middleware to check for 'admin' role before allowing access to the route.

```
module.exports = app => {
  const admin = require("../controllers/admin.controller.js");
  const authorize = require("../middlewares/authorize.js");
  var router = require("express").Router();

  router.get("/test", authorize(["admin"]), admin.test);

  app.use('/api/admin', router);
};
```

2. auth.routes.js:

- **Purpose**: Handles user authentication routes such as registration and login.
- **Routes**:
  - `POST /register`: Registers a new user.
  - `POST /login`: Authenticates a user and returns a token.
  - `GET /verify-email/:id/:token`: Verifies user's email.

```
module.exports = app => {
  const user = require("../controllers/auth.controller.js");
  var router = require("express").Router();

  router.post("/register", user.register);
  router.post("/login", user.login);
  router.get('/verify-email/:id/:token', user.verifyEmail);
```

```
  app.use('/api/auth', router);
};
```

3. book.routes.js:

- **Purpose**: Manages routes related to book operations.
- **Routes**:
  - `POST /`: Creates a new book (admin only).
  - `POST /multiple`: Creates multiple books (admin only).
  - `GET /`: Retrieves all books.
  - `DELETE /delete/:id`: Deletes a book by ID (admin only).
  - `DELETE /delete_all`: Deletes all books (admin only).
  - `PUT /update/:id`: Updates a book by ID (admin only).
  - `GET /find/:id`: Retrieves a single book by ID.
  - `POST /author`: Retrieves books by author.
  - `POST /rating`: Retrieves books with a specific rating or higher.
  - `POST /title`: Retrieves books by title.
  - `POST /genre`: Retrieves books by genre.
  - `GET /genres`: Retrieves all unique genres.
  - `GET /authors`: Retrieves all unique authors.
  - `GET /titles`: Retrieves all book titles.
  - `GET /ratings`: Retrieves books sorted by rating.
  - `GET /visited_times`: Retrieves books sorted by visited times.
  - `GET /count`: Retrieves the total number of books.
  - `POST /findBy`: Searches books based on multiple criteria.
- **Middleware**:
  - Uses `authorize` middleware to restrict certain routes to admin users.

```
module.exports = app => {
  const books = require("../controllers/book.controller.js");
  const router = require("express").Router();

  router.post("/", authorize("admin"), books.create);
  router.post("/multiple", authorize("admin"), books.createMultiple);
  router.get("/", books.findAll);
  router.delete("/delete/:id", authorize("admin"), books.delete);
  router.delete("/delete_all", authorize("admin"), books.deleteAll);
  router.put("/update/:id", authorize("admin"), books.update);
  router.get("/find/:id", books.findOne);
  router.post("/author", books.findByAuthor);
  router.post("/rating", books.findByMinRating);
  router.post("/title", books.findByTitle);
  router.post("/genre", books.findByGenre);
  router.get("/genres", books.findAllGenres);
  router.get("/authors", books.findAllAuthors);
  router.get("/titles", books.findAllTitles);
  router.get("/ratings", books.sortByRating);
  router.get("/visited_times", books.sortByVisitedTimes);
  router.get("/count", books.count);
  router.post("/findBy", books.findBy);

  app.use('/api/books', router);
};
```

4. room.routes.js:

- **Purpose**: Manages routes related to room operations.
- **Routes**:
  - `POST /`: Creates a new room (admin only).
  - `DELETE /delete/:id`: Deletes a room by ID (admin only).
  - `DELETE /delete_all`: Deletes all rooms (admin only).
  - `PUT /update/:id`: Updates a room by ID (admin only).
  - `GET /`: Retrieves all rooms.
  - `GET /find/:id`: Retrieves a single room by ID.
  - `GET /location`: Retrieves rooms by location.
  - `GET /available`: Retrieves all available rooms.
  - `GET /unavailable`: Retrieves all unavailable rooms.

- `GET /roomnumbers`: Retrieves all room numbers.
- `GET /roomlocations`: Retrieves all unique room locations.
- `GET /count`: Retrieves the total number of rooms.

- **Middleware**:
  - Uses `authorize` middleware to restrict certain routes to admin and authorized users.

```
module.exports = app => {
  const rooms = require("../controllers/room.controller.js");
  const user = require("../controllers/user.controller.js");
  const authorize = require("../middlewares/authorize.js");
  var router = require("express").Router();

  router.post("/", authorize("admin"), rooms.create);
  router.delete("/delete/:id", authorize("admin"), rooms.delete);
  router.delete("/delete_all/", authorize("admin"), rooms.deleteAll);
  router.put("/update/:id", authorize("admin"), rooms.update);
  router.get("/", rooms.findAll);
  router.get("/find/:id", authorize(["admin", "user"]), rooms.findOne);
  router.get("/location", authorize(["admin", "user"]), rooms.findByLocation);
  router.get("/available", authorize(["admin", "user"]), rooms.findAllByAvailable);
  router.get("/unavailable", authorize(["admin", "user"]), rooms.findAllByUnavailable);
  router.get("/roomnumbers", authorize(["admin", "user"]), rooms.findAllRoomNumbers);
  router.get("/roomlocations", authorize(["admin", "user"]), rooms.findAllRoomLocations);
  router.get("/count", rooms.count);

  app.use('/api/rooms', router);
};
```

5. roombooking.routes.js:

- **Purpose**: Manages routes related to room booking operations.
- **Routes**:
  - `POST /`: Creates a new room booking (admin and user).
  - `PUT /update/:id`: Updates a room booking by ID (admin and user).
  - `DELETE /delete/:id`: Deletes a room booking by ID (admin and user).
  - `DELETE /delete_all`: Deletes all room bookings (admin only).
  - `GET /`: Retrieves all room bookings.
  - `GET /find/:id`: Retrieves a single room booking by ID.
  - `GET /user/:id`: Searches bookings by user ID.
  - `GET /room/:id`: Searches bookings by room ID.
  - `GET /date/:date`: Searches bookings by date.
  - `GET /startendroom`: Searches bookings by start time, end time, and room ID.
- **Middleware**:
  - Uses `authorize` middleware to restrict certain routes to admin and authorized users.

```
module.exports = app => {
  const roomBookings = require("../controllers/roombooking.controller.js");
  const user = require("../controllers/user.controller.js");
  const authorize = require("../middlewares/authorize.js");
  var router = require("express").Router();

  router.post("/", authorize(["admin", "user"]), roomBookings.create);
  router.put("/update/:id", authorize(["admin", "user"]), roomBookings.update);
  router.delete("/delete/:id", authorize(["admin", "user"]), roomBookings.delete);
  router.delete("/delete_all/", authorize("admin"), roomBookings.deleteAll);
  router.get("/", authorize(["admin", "user"]), roomBookings.findAll);
  router.get("/find/:id", authorize(["admin", "user"]), roomBookings.findOne);
  router.get("/user/:id", authorize(["admin"]), roomBookings.findByUser);
  router.get("/room/:id", authorize(["admin", "user"]), roomBookings.findByRoom);
  router.get("/date/:date", authorize(["admin", "user"]), roomBookings.findByDate);
  router.get("/startendroom", authorize(["admin", "user"]), roomBookings.findByStartEndRoom);

  app.use('/api/roombookings', router);
};
```

6. user.routes.js:

- **Purpose**: Manages routes related to user operations.
- **Routes**:
  - `GET /info`: Retrieves information about the current authenticated user.
  - `GET /list`: Retrieves a list of all users.
  - `GET /count`: Retrieves the total number of users.
  - `GET /findByClubId/:club_id`: Retrieves users by club ID.
  - `POST /sendMailToAll`: Sends an email to all users (admin only).
  - `POST /sendMailToOne/:id`: Sends an email to a specific user (admin only).
  - `DELETE /delete/:id`: Deletes a user by ID (admin only).
  - `DELETE /deleteAll`: Deletes all users (admin only).
  - `DELETE /deleteByUsername/:username`: Deletes users by username (admin only).
  - `DELETE /deleteByEmail/:email`: Deletes users by email (admin only).
  - `PUT /update/:id`: Updates user information by ID.
- **Middleware**:
  - Uses `authorize` middleware to restrict certain routes to admin and authorized users.

```
module.exports = app => {
  const user = require("../controllers/user.controller.js");
  const authorize = require("../middlewares/authorize.js");
  var router = require("express").Router();

  router.get("/info", authorize(["admin", "user"]), user.info);
  router.get("/list", authorize(["admin", "user"]), user.list);
  router.get("/count", authorize(["admin", "user"]), user.count);
  router.get("/findByClubId/:club_id", authorize(["admin", "user"]), user.findByClubId);
  router.post("/sendMailToAll", authorize(["admin"]), user.sendMail);
  router.post("/sendMailToOne/:id", authorize(["admin"]), user.sendMailToOne);
  router.delete("/delete/:id", authorize(["admin"]), user.delete);
  router.delete("/deleteAll", authorize(["admin"]), user.deleteAll);
  router.delete("/deleteByUsername/:username", authorize(["admin"]), user.deleteByUsername);
  router.delete("/deleteByEmail/:email", authorize(["admin"]), user.deleteByEmail);
  router.put("/update/:id", authorize(["admin", "user"]), user.update);

  app.use('/api/user', router);
};
```

7. rent.routes.js:

- **Purpose**: Manages routes related to rental operations.
- **Routes**:
  - `GET /borrows`: Retrieves rental data for the current user (admin and user).
  - `GET /borrows/:ID_Book`: Retrieves rental information by book ID (user).
  - `POST /borrows`: Borrows books (user).
  - `PATCH /return`: Returns borrowed books, either specific ones or all (user).
  - `DELETE /borrows`: Deletes rental information, either specific ones or all (admin).
- **Middleware**:
  - Uses `authorize` middleware to restrict certain routes to admin and authorized users.

```
module.exports = app => {
  const rent = require("../controllers/rent.controller");
  const router = require("express").Router()

  router.get("/borrows", authorize(["admin", "user"]), rent.getDataBorrow);
  router.get("/borrows/:ID_Book", authorize(["user"]), rent.getBorrowbyId);
  router.post("/borrows", authorize(["user"]), rent.insertBorrow);
  router.patch("/return", authorize(["user"]), rent.returnBook);
  router.delete("/borrows", authorize(["admin"]), rent.deleteRental); // New route for admin to delete rental info

  // Register the router with the app
  app.use('/api', router);
};
```

IV.4. Testing Phase

The testing phase for the online library management project was conducted between April 15th and June 30th, 2023. The primary objectives of the testing were to validate the core functionality of the system, identify and resolve any defects, and ensure a smooth user experience for both librarians and patrons.

The testing was performed using a combination of manual and automated testing techniques, including:

1. Unit testing of individual components and modules, such as:

- Verifying the book search functionality returns accurate results based on title, author, and ISBN inputs
- Validating the patron registration process, including form validations and email confirmation

2. Integration testing to validate end-to-end user workflows, including:

- Checking the book borrowing and return process, ensuring the availability status and patron account are updated correctly
- Verifying the overdue notification system triggers emails to patrons with outstanding loans

3. UI/UX testing to assess the usability and visual design, such as:

- Evaluating the intuitive nature of the library catalog browsing experience
- Identifying any inconsistencies or usability issues in the patron and librarian dashboards

4. Load and performance testing to ensure the system can handle expected user volumes, including:

- Simulating 500 concurrent users performing common actions like searching, borrowing, and returning books
- Monitoring system response times and resource utilization under high load

5. Security testing to identify and mitigate potential vulnerabilities, including:

- Conducting penetration testing to attempt to gain unauthorized access to patron data and library resources
- Reviewing the implementation of access controls, input validation, and encryption mechanisms.

The testing phase was successful in validating the core functionality of the online library management system and ensuring a reliable and secure user experience. A few minor issues were identified and resolved, and the system is now ready for deployment to the production environment.

To further enhance the system, the following recommendations are suggested:

- Implement a more robust user feedback mechanism to continuously gather insights and improve the user experience
- Explore the integration of additional features, such as e-book lending, user reviews, and personalized recommendations
- Develop a comprehensive training plan to ensure smooth onboarding of librarians and effective utilization of the system.

Overall, the testing phase has provided confidence in the system's readiness for launch, and the project team is excited to move forward with the deployment and rollout to the end users.

IV.5. Docker and Final Deployment

Detailed Description of `docker-compose.yml` of the Main Library Application

The `docker-compose.yml` file defines the multi-container Docker application setup for a library system. It includes three services: a MySQL database (`mysqldb`), a backend API (`library-api`), and a frontend UI (`library-ui`). Below is a detailed breakdown of the file:

Version

```yaml
version: '3.8'
```

- Specifies the version of the Docker Compose file format being used.

Services

1. mysqldb

```yaml
mysqldb:
  image: mysql:5.7
  restart: unless-stopped
  env_file: ./.env
  environment:
    - MYSQL_ROOT_PASSWORD=$MYSQLDB_ROOT_PASSWORD
    - MYSQL_DATABASE=$MYSQLDB_DATABASE
  ports:
    - $MYSQLDB_LOCAL_PORT:$MYSQLDB_DOCKER_PORT
  volumes:
    - db:/var/lib/mysql
  networks:
    - backend
```

- **image**: Uses the `mysql:5.7` Docker image.
- **restart**: Restarts the container unless it is explicitly stopped.

- **env_file**: Loads environment variables from a `.env` file located in the same directory.
- **environment**: Sets environment variables for the MySQL container:
  - `MYSQL_ROOT_PASSWORD`: Sets the root password for MySQL.
  - `MYSQL_DATABASE`: Creates a database with the specified name.
- **ports**: Maps the local machine's port to the container's port:
  - `$MYSQLDB_LOCAL_PORT` to `$MYSQLDB_DOCKER_PORT` (values are taken from the `.env` file).
- **volumes**: Mounts a Docker volume to persist the database data:
  - `db` volume is mapped to `/var/lib/mysql` in the container.
- **networks**: Connects the container to the `backend` network.

2. library-api

```
library-api:
  depends_on:
    - mysqldb
  build: ./library-api
  restart: unless-stopped
  env_file: ./.env
  ports:
    - $NODE_LOCAL_PORT:$NODE_DOCKER_PORT
  environment:
    - DB_HOST=mysqldb
    - DB_USER=$MYSQLDB_USER
    - DB_PASSWORD=$MYSQLDB_ROOT_PASSWORD
    - DB_NAME=$MYSQLDB_DATABASE
    - DB_PORT=$MYSQLDB_DOCKER_PORT
    - CLIENT_ORIGIN=$CLIENT_ORIGIN
  networks:
    - backend
    - frontend
  volumes:
    - ./library-api/app:/library-api/app
```

- **depends_on**: Ensures that the `mysqldb` service is started before the `library-api` service.
- **build**: Builds the Docker image from the `./library-api` directory.
- **restart**: Restarts the container unless it is explicitly stopped.
- **env_file**: Loads environment variables from the `.env` file.
- **ports**: Maps the local machine's port to the container's port:
  - `$NODE_LOCAL_PORT` to `$NODE_DOCKER_PORT` (values are taken from the `.env` file).
- **environment**: Sets environment variables for the backend API container:
  - `DB_HOST`: Sets the database host to `mysqldb`.
  - `DB_USER`: Sets the database user.
  - `DB_PASSWORD`: Sets the database password.
  - `DB_NAME`: Sets the database name.
  - `DB_PORT`: Sets the database port.
  - `CLIENT_ORIGIN`: Sets the client origin URL.
- **networks**: Connects the container to both `backend` and `frontend` networks.
- **volumes**: Mounts a local directory to the container:
  - `./library-api/app` is mapped to `/library-api/app` in the container.

3. library-ui

```
library-ui:
  depends_on:
    - library-api
  build:
    context: ./library-ui
    args:
      - REACT_APP_API_BASE_URL=$CLIENT_API_BASE_URL
  ports:
    - $REACT_LOCAL_PORT:$REACT_DOCKER_PORT
  networks:
    - frontend
  volumes:
    - ./library-ui/src:/library-ui/src
```

- **depends_on**: Ensures that the `library-api` service is started before the `library-ui` service.

- **build**: Builds the Docker image from the `./library-ui` directory with build arguments:
  - `REACT_APP_API_BASE_URL`: Sets the API base URL for the React application.
- **ports**: Maps the local machine's port to the container's port:
  - `$REACT_LOCAL_PORT` to `$REACT_DOCKER_PORT` (values are taken from the `.env` file).
- **networks**: Connects the container to the `frontend` network.
- **volumes**: Mounts a local directory to the container:
  - `./library-ui/src` is mapped to `/library-ui/src` in the container.

Volumes

```
volumes:
  db:
```

- Defines a named volume `db` to persist the MySQL database data.

Networks

```
networks:
  backend:
  frontend:
```

- Defines two networks `backend` and `frontend` to isolate and manage the communication between the services.

Detailed Description of docker-compose.yml for the Forum and Chat Application

The `docker-compose.yml` file defines the multi-container Docker application setup for a forum and chat system. It includes two separate services: a chat service and a forum service. Below is a detailed breakdown of the files for each service:

Chat Service

Dockerfile

The Dockerfile builds and serves a React application using Nginx.

```
# Step 1: Build the React application
FROM node:20 AS build
WORKDIR /app
COPY package.json ./
RUN npm install
COPY . ./
RUN npm run build

# Step 2: Serve the application with Nginx
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

- **Base Image**: Uses `node:20` for building the React application.
- **Build Stage**:
  - Sets the working directory to `/app`.
  - Copies `package.json` and installs dependencies.
  - Copies the rest of the application files and builds the React application.
- **Serve Stage**:
  - Uses `nginx:alpine` to serve the built files.
  - Copies the build output from the previous stage to the Nginx HTML directory.
  - Exposes port 80.
  - Starts Nginx in the foreground.

docker-compose.yml

Defines the service for the chat application.

```
version: '3.8'
services:
  web:
    build: .
    ports:
```

```
      - "3001:80"
    volumes:
      - ./src:/app/src
      - ./public:/app/public
    environment:
      - NODE_ENV=production
```

- **build**: Builds the Docker image from the current directory.
- **ports**: Maps local port 3001 to container port 80.
- **volumes**: Mounts local directories to the container:
  - `./src` to `/app/src`
  - `./public` to `/app/public`
- **environment**: Sets the environment variable `NODE_ENV` to `production`.

Forum Service

Dockerfile

The Dockerfile sets up a Node.js application with PostgreSQL and Redis dependencies.

```
# Step 1: Use an official Node.js runtime as a parent image
FROM node:20

# Step 2: Set the working directory in the container
WORKDIR /usr/src/app

# Step 3: Copy package.json, yarn.lock
COPY package.json yarn.lock ./

# Step 4: Install dependencies using Yarn
RUN yarn

# Step 5: Copy the rest of your application
COPY . .

# Step 6: Add wait-for-it script
COPY wait-for-it.sh /usr/src/app/wait-for-it.sh
RUN chmod +x /usr/src/app/wait-for-it.sh

# Step 7: Add entrypoint script
COPY entrypoint.sh /usr/src/app/entrypoint.sh
RUN chmod +x /usr/src/app/entrypoint.sh

# Step 8: Build your Next.js application
RUN yarn prisma generate
RUN yarn build

# Step 9: Define the entrypoint to run your app
ENTRYPOINT ["/usr/src/app/entrypoint.sh"]
```

- **Base Image**: Uses `node:20`.
- **Setup**:
  - Sets the working directory to `/usr/src/app`.
  - Copies `package.json` and `yarn.lock` and installs dependencies.
  - Copies the rest of the application files.
  - Copies `wait-for-it.sh` and `entrypoint.sh`, and makes them executable.
  - Generates Prisma client and builds the Next.js application.
- **ENTRYPOINT**: Sets the entrypoint to `entrypoint.sh`.

docker-compose.yml

Defines the services for the forum application.

```
version: '3.8'

services:
  db:
    image: postgres:13
    volumes:
```

```
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: breadit
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
  app:
    build: .
    volumes:
      - .:/app
    ports:
      - "3000:3000"
    depends_on:
      - db
      - redis
  redis:
    image: redis:6
    volumes:
      - redis_data:/data

volumes:
  postgres_data:
  redis_data:
```

db Service

- **image**: Uses `postgres:13`.
- **volumes**: Mounts a volume to persist PostgreSQL data:
  - `postgres_data` to `/var/lib/postgresql/data`.
- **environment**: Sets environment variables for PostgreSQL:
  - `POSTGRES_DB`: Database name.
  - `POSTGRES_USER`: Database user.
  - `POSTGRES_PASSWORD`: Database password.

app Service

- **build**: Builds the Docker image from the current directory.
- **volumes**: Mounts the current directory to the container:
  - `.` to `/app`.
- **ports**: Maps local port 3000 to container port 3000.
- **depends_on**: Ensures that the `db` and `redis` services are started before the `app` service.

redis Service

- **image**: Uses `redis:6`.
- **volumes**: Mounts a volume to persist Redis data:
  - `redis_data` to `/data`.

volumes

Defines named volumes to persist data for PostgreSQL and Redis.

```
volumes:
  postgres_data:
  redis_data:
```

This Docker Compose setup ensures that both the chat and forum services are configured correctly, with appropriate dependencies, environment variables, and data persistence. The services are connected through defined networks, and data is persisted using named volumes.

V. Social Forum

Table of Contents

Community

This section of the report will detail the implementation of the function "community".

Create a community

Validator

`subreddit.ts` is a validator which defines a set of validation rules for creating and subscribing to a subreddit using Zod, a TypeScript and JavaScript validation library.

```
import { z } from "zod";

export const SubredditValidator = z.object({
  name: z.string().min(3).max(30),
});
```

`SubredditValidator` is defined as a Zod schema to validate subreddit objects.

- The subreddit object is expected to have a name property.
- The name should be a string with a minimum length of 3 characters and a maximum length of 30 characters.

```
export const SubredditSubscriptionValidator = z.object({
  subredditId: z.string(),
});
```

`SubredditSubscriptionValidator` is defined as another Zod schema to validate subreddit subscription objects.

- The subscription object is expected to have a `subredditId` property.
- The `subredditId` should be a string.

```
export type CreateSubredditPayload = z.infer<typeof SubredditValidator>;
export type SubscribeToSubredditPayload = z.infer<typeof SubredditSubscriptionValidator>;
```

Two TypeScript types, `CreateSubredditPayload` and `SubscribeToSubredditPayload`, are inferred from the `SubredditValidator` and `SubredditSubscriptionValidator` schemas respectively using Zod's infer keyword.

- These types match the structure and type of the objects described by the respective schemas.
- They can be used to ensure that the data in the TypeScript code is valid according to the defined schemas. Any discrepancies will result in a compile-time error.

Required Components

Provider.tsx

The code snippet is as follows:

```
'use client'

interface LayoutProps {
  children: ReactNode
}

const queryClient = new QueryClient()

const Providers: FC<LayoutProps> = ({ children }) => {
  return (
    <QueryClientProvider client={queryClient}>
      <SessionProvider>{children}</SessionProvider>
    </QueryClientProvider>
  )
}

export default Providers
```

- `'use client'` is a directive that indicates the code will be executed in a client-side context.
- The `QueryClientProvider` and `QueryClient` are imported from '@tanstack/react-query'. These are used to provide a React Query client to the component tree.
- The `SessionProvider` is imported from 'next-auth/react'. This is used to provide session state to the component tree.
- The `FC` (Function Component) and `ReactNode` are imported from 'react'. `ReactNode` is a type that accepts any kind of React child elements (like a component, a JSX element, or a string), and `FC` is a type for function components.
- `LayoutProps` is an interface that expects a `children` prop of type `ReactNode`.
- `queryClient` is an instance of `QueryClient` which will be used to configure the `QueryClientProvider`.
- `Providers` is a function component that uses `QueryClientProvider` to provide the `queryClient` to its child components, and `SessionProvider` to provide session state to its child components.
- Finally, `Providers` is exported as the default export of the module.

Client-side rendering

```
const router = useRouter()
const [input, setInput] = useState<string>('')
const { loginToast } = useCustomToasts()
```

- Initialization: The component initializes the router from `next/navigation`, a state for the subreddit input, and a custom login toast from `useCustomToasts`.

```
const { mutate: createCommunity, isLoading } = useMutation({
  mutationFn: async () => {
    const payload: CreateSubredditPayload = {
      name: input,
    }

    const { data } = await axios.post('/api/subreddit', payload)
    return data as string
  },
```

- Mutation definition: The component defines a mutation for creating a new subreddit using `useMutation`. The mutation function sends a POST request to the `/api/subreddit` endpoint with the subreddit name as payload.

```
onError: (err) => {
  if (err instanceof AxiosError) {
    if (err.response?.status === 409) {
      return toast({
        title: 'Subreddit already exists.',
        description: 'Please choose a different name.',
        variant: 'destructive',
      })
    }

    if (err.response?.status === 422) {
```

```
      return toast({
        title: 'Invalid subreddit name.',
        description: 'Please choose a name between 3 and 21 letters.',
        variant: 'destructive',
      })
    }

    if (err.response?.status === 401) {
      return loginToast()
    }
  }

  toast({
    title: 'There was an error.',
    description: 'Could not create subreddit.',
    variant: 'destructive',
  })
},
```

- Error handling: If an error occurs during the mutation, the component checks the type and status of the error and displays an appropriate toast message.

```
onSuccess: (data) => {
  router.push(`/r/${data}`)
},
```

5. **Success handling**: If the mutation is successful, the component redirects the user to the newly created subreddit page.

```
return (
  <div className='container flex items-center h-full max-w-3xl mx-auto'>
    <div className='relative bg-white w-full h-fit p-4 rounded-lg space-y-6'>
      // ... omitted for brevity
    </div>
  </div>
)
```

- Rendering: The component renders a form for creating a new subreddit. The form includes an input for the subreddit name and two buttons for cancelling and submitting the form. The submission button triggers the `createCommunity` mutation.

API

Create an API route to handle HTTP request of creating a new community from the client. In next.js, this API route will be created in project's directory structure: `src/app/api/subreddit/route.ts`

subreddit/route.ts

```
const session = await getAuthSession()

if (!session?.user) {
  return new Response('Unauthorized', { status: 401 })
}
```

- Authentication: The `getAuthSession` function is called to get the current session. If there is no authenticated user in the session, the function returns an HTTP 401 Unauthorized status.

```
const body = await req.json()
const { name } = SubredditValidator.parse(body)
```

- Request Parsing: The function parses the incoming request to extract the request body as JSON. It then validates the parsed body to extract the subreddit name using `SubredditValidator`.

```
const subredditExists = await db.subreddit.findFirst({
  where: {
    name,
  },
})
```

```
if (subredditExists) {
  return new Response('Subreddit already exists', { status: 409 })
}
```

- Subreddit Existence Check: The function checks if a subreddit with the extracted name already exists in the database. If it does, the function returns an HTTP 409 Conflict status.

```
const subreddit = await db.subreddit.create({
  data: {
    name,
    creatorId: session.user.id,
  },
})
```

- Subreddit Creation: If the subreddit does not already exist, the function creates a new subreddit in the database, associating it with the authenticated user.

```
await db.subscription.create({
  data: {
    userId: session.user.id,
    subredditId: subreddit.id,
  },
})
```

- Subscription Creation: The function also creates a subscription for the authenticated user to the newly created subreddit.

```
} catch (error) {
  if (error instanceof z.ZodError) {
    return new Response(error.message, { status: 422 })
  }

  return new Response('Could not create subreddit', { status: 500 })
}
```

- Error Handling: If an error occurs during this process, the function checks if the error is an instance of `z.ZodError` (a validation error). If it is, the function returns an HTTP 422 Unprocessable Entity status with the error message. For any other error, it returns an HTTP 500 Internal Server Error status.

Community detail page

Required Components

MiniPostCreate.tsx

```
interface MiniCreatePostProps {
  session: Session | null
}

const MiniCreatePost: FC<MiniCreatePostProps> = ({ session }) => {
  const router = useRouter()
  const pathname = usePathname()
```

- Defines the `MiniCreatePostProps` interface to specify that the component accepts a `session` prop, which can be either a `Session` object or `null`.
- Declares the `MiniCreatePost` functional component (FC) which takes `MiniCreatePostProps` as its props.
- Uses the `useRouter` and `usePathname` hooks from Next.js to get the router instance and the current pathname, respectively.

```
return (
  <li className='overflow-hidden rounded-md bg-white shadow'>
    <div className='h-full px-6 py-4 flex justify-between gap-6'>
```

- The component returns a list item (`<li>`) with some Tailwind CSS classes for styling.

- Inside the list item, there is a `div` that uses flexbox for layout and applies padding and gap styles.

```
<div className='relative'>
  <UserAvatar
    user={{
      name: session?.user.name || null,
      image: session?.user.image || null,
    }}
  />
  <span className='absolute bottom-0 right-0 rounded-full w-3 h-3 bg-green-500 outline outline-2 outline-white' />
</div>
```

- A `div` with a `relative` class is used to position elements inside it.
- The `UserAvatar` component is used to display the user's avatar, with the user's name and image passed as props.
- A `span` element is used to display a green dot at the bottom-right corner, indicating the user's online status.

```
<Input
  onClick={() => router.push(pathname + '/submit')}
  readOnly
  placeholder='Create post'
/>
```

- An `Input` component is rendered with a `readOnly` attribute and a placeholder text "Create post".
- The `onClick` event handler navigates the user to the `/submit` page when the input is clicked.

```
<Button
  onClick={() => router.push(pathname + '/submit')}
  variant='ghost'>
  <ImageIcon className='text-zinc-600' />
</Button>
<Button
  onClick={() => router.push(pathname + '/submit')}
  variant='ghost'>
  <Link2 className='text-zinc-600' />
</Button>
```

- Two `Button` components are rendered, each with an `onClick` event handler that navigates the user to the `/submit` page.
- The buttons use the `variant='ghost'` prop for styling.
- Each button contains an icon (`ImageIcon` and `Link2`, respectively) with a text color applied using Tailwind CSS classes.

```
export default MiniCreatePost
```

- Exports the `MiniCreatePost` component as the default export of the module.

PostFeed.tsx

```
interface PostFeedProps {
  initialPosts: ExtendedPost[];
  subredditName?: string;
}

const PostFeed: FC<PostFeedProps> = ({ initialPosts, subredditName }) => {
```

- Defines the `PostFeedProps` interface to specify that the component accepts `initialPosts` (an array of `ExtendedPost` objects) and an optional `subredditName` prop.
- Declares the `PostFeed` functional component (FC) which takes `PostFeedProps` as its props.

```
const lastPostRef = useRef<HTMLElement>(null);
const { ref, entry } = useIntersection({
  root: lastPostRef.current,
  threshold: 1,
});
const { data: session } = useSession();
```

- Uses the `useRef` hook to create a reference `lastPostRef` for the last post element.
- Uses the `useIntersection` hook from `@mantine/hooks` to detect when the last post comes into view.
- Uses the `useSession` hook from `next-auth/react` to get the current session data.

```
const { data, fetchNextPage, isFetchingNextPage } = useInfiniteQuery(
  ["infinite-query"],
  async ({ pageParam = 1 }) => {
    const query =
      `/api/posts?limit=${INFINITE_SCROLL_PAGINATION_RESULTS}&page=${pageParam}` +
      (!!subredditName ? `&subredditName=${subredditName}` : "");

    const { data } = await axios.get(query);
    return data as ExtendedPost[];
  },
  {
    getNextPageParam: (_, pages) => {
      return pages.length + 1;
    },
    initialData: { pages: [initialPosts], pageParams: [1] },
  }
);
```

- Uses the `useInfiniteQuery` hook from `@tanstack/react-query` to fetch posts in an infinite scroll manner.
- The query fetches posts from the `/api/posts` endpoint, with pagination and optional subreddit filtering.
- Sets up the `getNextPageParam` function to determine the next page parameter.
- Provides initial data with `initialPosts`.

```
useEffect(() => {
  if (entry?.isIntersecting) {
    fetchNextPage(); // Load more posts when the last post comes into view
  }
}, [entry, fetchNextPage]);
```

- Uses the `useEffect` hook to trigger `fetchNextPage` when the last post comes into view.

```
const posts = data?.pages.flatMap((page) => page) ?? initialPosts;
```

- Flattens the pages of posts into a single array, falling back to `initialPosts` if no data is available.

```
return (
  <ul className="flex flex-col col-span-2 space-y-6">
    {posts.map((post, index) => {
      const votesAmt = post.votes.reduce((acc, vote) => {
        if (vote.type === "UP") return acc + 1;
        if (vote.type === "DOWN") return acc - 1;
        return acc;
      }, 0);

      const currentVote = post.votes.find(
        (vote) => vote.userId === session?.user.id
      );

      if (index === posts.length - 1) {
        // Add a ref to the last post in the list
        return (
          <li key={post.id} ref={ref}>
            <Post
              post={post}
              commentAmt={post.comments.length}
              subredditName={post.subreddit.name}
              votesAmt={votesAmt}
              currentVote={currentVote}
            />
          </li>
        );
```

```
      } else {
        return (
          <Post
            key={post.id}
            post={post}
            commentAmt={post.comments.length}
            subredditName={post.subreddit.name}
            votesAmt={votesAmt}
            currentVote={currentVote}
          />
        );
      }
    })}

    {isFetchingNextPage && (
      <li className="flex justify-center">
        <Loader2 className="w-6 h-6 text-zinc-500 animate-spin" />
      </li>
    )}
  </ul>
);
```

- Maps over the `posts` array to render each post.
- Calculates the total votes for each post.
- Finds the current user's vote on each post.
- Adds a ref to the last post in the list to trigger the intersection observer.
- Displays a loading spinner if more posts are being fetched.

Client-side rendering

```
interface PageProps {
  params: {
    slug: string
  }
}
```

- Defines the `PageProps` interface to specify that the component accepts `params` with a `slug` property.

```
const page = async ({ params }: PageProps) => {
  const { slug } = params
```

- Declares the `page` asynchronous function component which takes `PageProps` as its props.
- Destructures the `slug` from `params`.
- **Session Retrieval**

```
const session = await getAuthSession()
```

  - Calls `getAuthSession` to retrieve the current user session.

```
const subreddit = await db.subreddit.findFirst({
  where: { name: slug },
  include: {
    posts: {
      include: {
        author: true,
        votes: true,
        comments: true,
        subreddit: true,
      },
      orderBy: {
        createdAt: 'desc'
      },
      take: INFINITE_SCROLL_PAGINATION_RESULTS,
    },
```

```
  },
})
```

- Uses the `db.subreddit.findFirst` method to fetch the subreddit data based on the `slug`.
- Includes related `posts`, `author`, `votes`, `comments`, and `subreddit` data.
- Orders the posts by `createdAt` in descending order.
- Limits the number of posts fetched to `INFINITE_SCROLL_PAGINATION_RESULTS`.

```
if (!subreddit) return notFound()
```

- Checks if the subreddit exists. If not, returns a `notFound` response.

```
return (
  <>
    <h1 className='font-bold text-3xl md:text-4xl h-14'>
      r/{subreddit.name}
    </h1>
    <MiniCreatePost session={session} />
    <PostFeed initialPosts={subreddit.posts} subredditName={subreddit.name} />
  </>
)
```

- Returns a fragment containing:
  - A heading displaying the subreddit name.
  - The `MiniCreatePost` component, passing the `session` as a prop.
  - The `PostFeed` component, passing the initial posts and subreddit name as props.

```
export default page
```

- Exports the `page` component as the default export of the module.

Join and leave community

Required Components

SubscribeLeaveToggle.tsx

```
interface SubscribeLeaveTogglePropsProps {
  isSubscribed: boolean
  subredditId: string
  subredditName: string
}
```

- Defines the `SubscribeLeaveTogglePropsProps` interface to specify that the component accepts `isSubscribed`, `subredditId`, and `subredditName` props.

```
const SubscribeLeaveToggle = ({
  isSubscribed,
  subredditId,
  subredditName,
}: SubscribeLeaveTogglePropsProps) => {
```

- Declares the `SubscribeLeaveToggle` functional component which takes `SubscribeLeaveTogglePropsProps` as its props.

```
const { toast } = useToast()
const { loginToast } = useCustomToasts()
const router = useRouter()
```

- Initializes the `useToast` and `useCustomToasts` hooks to display toast notifications.
- Initializes the `useRouter` hook from Next.js to handle navigation.

```
const { mutate: subscribe, isLoading: isSubLoading } = useMutation({
  mutationFn: async () => {
    const payload: SubscribeToSubredditPayload = {
```

```
      subredditId,
    }

    const { data } = await axios.post('/api/subreddit/subscribe', payload)
    return data as string
  },
  onError: (err) => {
    if (err instanceof AxiosError) {
      if (err.response?.status === 401) {
        return loginToast()
      }
    }

    return toast({
      title: 'There was a problem.',
      description: 'Something went wrong. Please try again.',
      variant: 'destructive',
    })
  },
  onSuccess: () => {
    startTransition(() => {
      router.refresh()
    })
    toast({
      title: 'Subscribed!',
      description: `You are now subscribed to r/${subredditName}`,
    })
  },
})
```

- Uses the `useMutation` hook from `@tanstack/react-query` to handle the subscription process.
- Defines the `mutationFn` to send a POST request to subscribe to the subreddit.
- Handles errors and displays appropriate toast notifications.
- On success, refreshes the current route and displays a success toast.

```
const { mutate: unsubscribe, isLoading: isUnsubLoading } = useMutation({
  mutationFn: async () => {
    const payload: SubscribeToSubredditPayload = {
      subredditId,
    }

    const { data } = await axios.post('/api/subreddit/unsubscribe', payload)
    return data as string
  },
  onError: (err: AxiosError) => {
    toast({
      title: 'Error',
      description: err.response?.data as string,
      variant: 'destructive',
    })
  },
  onSuccess: () => {
    startTransition(() => {
      router.refresh()
    })
    toast({
      title: 'Unsubscribed!',
      description: `You are now unsubscribed from/${subredditName}`,
    })
  },
})
```

- Uses the `useMutation` hook from `@tanstack/react-query` to handle the unsubscription process.
- Defines the `mutationFn` to send a POST request to unsubscribe from the subreddit.
- Handles errors and displays appropriate toast notifications.

- On success, refreshes the current route and displays a success toast.

```
return isSubscribed ? (
  <Button
    className='w-full mt-1 mb-4'
    isLoading={isUnsubLoading}
    onClick={() => unsubscribe()}>
    Leave community
  </Button>
) : (
  <Button
    className='w-full mt-1 mb-4'
    isLoading={isSubLoading}
    onClick={() => subscribe()}>
    Join to post
  </Button>
)
```

- Renders a `Button` component based on the `isSubscribed` prop.
- If `isSubscribed` is true, renders a button to leave the community, with an `onClick` handler to trigger the `unsubscribe` mutation.
- If `isSubscribed` is false, renders a button to join the community, with an `onClick` handler to trigger the `subscribe` mutation.

ToFeedButton.tsx

```
const ToFeedButton = () => {
  const pathname = usePathname()
```

- Declares the `ToFeedButton` functional component.
- Uses the `usePathname` hook from Next.js to get the current pathname.

```
// if path is /r/mycom, turn into /
// if path is /r/mycom/post/cligad6jf0003uhest4qqkeco, turn into /r/mycom

const subredditPath = getSubredditPath(pathname)
```

- Calls the `getSubredditPath` function to determine the appropriate path based on the current pathname.

```
return (
  <a href={subredditPath} className={buttonVariants({ variant: 'ghost' })}>
    <ChevronLeft className='h-4 w-4 mr-1' />
    {subredditPath === '/' ? 'Back home' : 'Back to community'}
  </a>
)
```

- Returns a link (`<a>`) with the `href` set to `subredditPath` .
- Uses the `buttonVariants` function to apply the `ghost` variant styling to the link.
- Displays a `ChevronLeft` icon and conditional text based on `subredditPath` .

```
const getSubredditPath = (pathname: string) => {
  const splitPath = pathname.split('/')

  if (splitPath.length === 3) return '/'
  else if (splitPath.length > 3) return `/${splitPath[1]}/${splitPath[2]}`
  // default path, in case pathname does not match expected format
  else return '/'
}
```

- Defines the `getSubredditPath` function which takes `pathname` as an argument.
- Splits the `pathname` into parts using the `/` delimiter.
- Returns `/` if the path has three segments.
- Returns the path up to the subreddit name if the path has more than three segments.
- Returns `/` as the default path if the pathname does not match the expected format.

Client-side Rendering

```
export const metadata: Metadata = {
  title: "Mythos Forum",
};
```

- Defines metadata for the page, setting the title to "Mythos Forum".

```
const Layout = async ({
  children,
  params: { slug },
}: {
  children: ReactNode;
  params: { slug: string };
}) => {
  const session = await getAuthSession();
```

- Declares the `Layout` asynchronous function component which takes `children` and `params` with `slug` as its props.
- Retrieves the current user session using `getAuthSession`.

```
const subreddit = await db.subreddit.findFirst({
  where: { name: slug },
  include: {
    posts: {
      include: {
        author: true,
        votes: true,
      },
    },
  },
});
```

- Fetches the subreddit data based on the `slug`.
- Includes related `posts`, `author`, and `votes` data.

```
const subscription = !session?.user
  ? undefined
  : await db.subscription.findFirst({
      where: {
        subreddit: {
          name: slug,
        },
        user: {
          id: session.user.id,
        },
      },
    });

const isSubscribed = !!subscription;
```

- Checks if the user is subscribed to the subreddit.
- Sets `subscription` to `undefined` if there is no user session.
- Sets `isSubscribed` to a boolean indicating if a subscription exists.

```
if (!subreddit) return notFound();
```

- Returns a `notFound` response if the subreddit does not exist.

```
const memberCount = await db.subscription.count({
  where: {
    subreddit: {
      name: slug,
    },
```

```
  },
});
```

- Counts the number of members in the subreddit.

```
return (
  <div className="sm:container max-w-7xl mx-auto h-full pt-12">
    <div>
      <ToFeedButton />

      <div className="grid grid-cols-1 md:grid-cols-3 gap-y-4 md:gap-x-4 py-6">
        <ul className="flex flex-col col-span-2 space-y-6">{children}</ul>
```

- Returns the main layout structure.
- Uses a container with responsive classes for styling.
- Includes a `ToFeedButton` component.
- Creates a grid layout for the main content and sidebar, with `children` rendered in a list.

```
<div className="overflow-hidden h-fit rounded-lg border border-gray-200 order-first md:order-last">
  <div className="px-6 py-4">
    <p className="font-semibold py-3">About r/{subreddit.name}</p>
  </div>
```

- Creates an info sidebar with a heading displaying the subreddit name.

```
<dl className="divide-y divide-gray-100 px-6 py-4 text-sm leading-6 bg-white">
  <div className="flex justify-between gap-x-4 py-3">
    <dt className="text-gray-500">Created</dt>
    <dd className="text-gray-700">
      <time dateTime={subreddit.createdAt.toDateString()}>
        {format(subreddit.createdAt, "MMMM d, yyyy")}
      </time>
    </dd>
  </div>
  <div className="flex justify-between gap-x-4 py-3">
    <dt className="text-gray-500">Members</dt>
    <dd className="flex items-start gap-x-2">
      <div className="text-gray-900">{memberCount}</div>
    </dd>
  </div>
```

- Displays the creation date and member count of the subreddit.

```
{subreddit.creatorId === session?.user?.id ? (
  <div className="flex justify-between gap-x-4 py-3">
    <dt className="text-gray-500">You created this community</dt>
  </div>
) : null}
```

- Shows a message if the current user is the creator of the subreddit.

```
{subreddit.creatorId !== session?.user?.id ? (
  <SubscribeLeaveToggle
    isSubscribed={isSubscribed}
    subredditId={subreddit.id}
    subredditName={subreddit.name}
  />
) : null}
```

- Renders the `SubscribeLeaveToggle` component if the current user is not the creator of the subreddit.

```
<Link
  className={buttonVariants({
    variant: "outline",
```

```
    className: "w-full mb-6",
  })}
  href={`r/${slug}/submit`}
>
  Create Post
</Link>
```

- Renders a link button to create a new post in the subreddit.

API

subscribe.ts

```
import { getAuthSession } from '@/lib/auth'
import { db } from '@/lib/db'
import { SubredditSubscriptionValidator } from '@/lib/validators/subreddit'
import { z } from 'zod'
```

- Imports necessary functions and libraries.

```
export async function POST(req: Request) {
  try {
    const session = await getAuthSession()

    if (!session?.user) {
      return new Response('Unauthorized', { status: 401 })
    }
```

- Declares an asynchronous `POST` function to handle the subscription request.
- Retrieves the current user session using `getAuthSession`.
- Checks if the user is authenticated. If not, returns an `Unauthorized` response.

```
const body = await req.json()
const { subredditId } = SubredditSubscriptionValidator.parse(body)
```

- Parses the request body to extract the `subredditId` using `SubredditSubscriptionValidator`.

```
// check if user has already subscribed to subreddit
const subscriptionExists = await db.subscription.findFirst({
  where: {
    subredditId,
    userId: session.user.id,
  },
})

if (subscriptionExists) {
  return new Response("You've already subscribed to this subreddit", {
    status: 400,
  })
}
```

- Checks if the user has already subscribed to the subreddit.
- If a subscription exists, returns a response indicating that the user has already subscribed.

```
// create subreddit and associate it with the user
await db.subscription.create({
  data: {
    subredditId,
    userId: session.user.id,
  },
})

return new Response(subredditId)
```

- Creates a new subscription and associates it with the user.

- Returns a response with the `subredditId` on successful subscription.

```
} catch (error) {
  if (error instanceof z.ZodError) {
    return new Response(error.message, { status: 400 })
  }

  return new Response(
    'Could not subscribe to subreddit at this time. Please try later',
    { status: 500 }
  )
}
```

- Catches any errors that occur during the process.
- If the error is a `ZodError`, returns a response with the error message and a `400` status.
- For other errors, returns a response indicating that the subscription could not be processed at this time, with a `500` status.

unsubscribe.ts For the most, part this API route is similar to the subscribe route with some differences:

````
```typescript
if (!subscriptionExists) {
  return new Response(
    "You've not been subscribed to this subreddit, yet.",
    {
      status: 400,
    }
  )
}
```
- Checks if the user is not subscribed to the subreddit and returns an error if true.

```typescript
await db.subscription.delete({
  where: {
    userId_subredditId: {
      subredditId,
      userId: session.user.id,
    },
  },
})
```
- Deletes the existing subscription that associates the user with the subreddit.

```typescript
return new Response(
  'Could not unsubscribe from subreddit at this time. Please try later',
  { status: 500 }
)
```
- Returns an error message indicating that the unsubscription could not be processed.
````

Post

Editor

Validator

```
import { z } from "zod";

export const PostValidator = z.object({
  title: z
    .string()
    .min(3, {
      message: "Title must be at least 3 characters long",
    })
```

```
    .max(128, {
      message: "Title must be less than 128 characters long",
    }),
  subredditId: z.string(),
  content: z.any(),
  subredditName: z.string(),
});
```

- **PostValidator**: This is a schema defined using the `zod` library to validate the structure of a post creation request.
  - **title**: A string that must be between 3 and 128 characters long. Custom error messages are provided for both the minimum and maximum length constraints.
  - **subredditId**: A string representing the ID of the subreddit.
  - **content**: A field that can be of any type. This is used to store the content of the post.
  - **subredditName**: A string representing the name of the subreddit.

Usage in the Editor Component

`PostValidator` and `PostCreationRequest` are used to manage and validate the form data for creating a new post.

- **Form Initialization**:

```
const {
  register,
  handleSubmit,
  formState: { errors },
} = useForm<FormData>({
  resolver: zodResolver(PostValidator),
  defaultValues: {
    subredditId,
    title: "",
    content: null,
    subredditName,
  },
});
```

  - The `useForm` hook is initialized with the `PostValidator` schema using `zodResolver` to handle form validation.
  - Default values for the form fields are set using the `defaultValues` option.

- **Form Submission Handler**:

```
async function onSubmit(data: FormData) {
  const blocks = await ref.current?.save();

  const payload: PostCreationRequest = {
    title: data.title,
    content: blocks,
    subredditId,
    subredditName,
  };

  createPost(payload);
}
```

  - The `onSubmit` function is defined to handle form submission.
  - The form data is validated against the `PostValidator` schema.
  - The validated data is then used to create a payload of type `PostCreationRequest` for the post creation API call.

Form Initialization

```
const {
  register,
  handleSubmit,
  formState: { errors },
} = useForm<FormData>({
  resolver: zodResolver(PostValidator),
  defaultValues: {
    subredditId,
```

```
    title: "",
    content: null,
    subredditName,
  },
});
```

- Uses the `useForm` hook from `react-hook-form` to manage form state and validation.
- **register**: Registers the form fields.
- **handleSubmit**: Handles form submission.
- **errors**: Contains validation errors.
- **resolver**: Uses `zodResolver` to integrate `zod` validation schema.
- **defaultValues**: Sets default values for the form fields.

Refs and State Initialization

```
const ref = useRef<EditorJS>();
const _titleRef = useRef<HTMLTextAreaElement>(null);
const router = useRouter();
const [isMounted, setIsMounted] = useState<boolean>(false);
const pathname = usePathname();
```

- **ref**: A ref to hold the EditorJS instance.
- **_titleRef**: A ref for the title input element.
- **router**: A Next.js hook for navigation.
- **isMounted**: A state to track if the component is mounted.
- **pathname**: A Next.js hook to get the current path.

Mutation for Post Creation

```
const { mutate: createPost } = useMutation({
  mutationFn: async ({
    title,
    content,
    subredditId,
    subredditName,
  }: PostCreationRequest) => {
    const payload: PostCreationRequest = {
      title,
      content,
      subredditId,
      subredditName,
    };
    const { data } = await axios.post("/api/subreddit/post/create", payload);
    return data;
  },
  onError: () => {
    return toast({
      title: "Something went wrong.",
      description: "Your post was not published. Please try again.",
      variant: "destructive",
    });
  },
  onSuccess: () => {
    const newPathname = pathname.split("/").slice(0, -1).join("/");
    router.push(newPathname);

    router.refresh();

    return toast({
      description: "Your post has been published.",
    });
  },
});
```

- **useMutation**: A hook from `@tanstack/react-query` to handle the post creation process.
  - **mutationFn**: The function to execute when the mutation is triggered. It sends a POST request to create a new post.
  - **onError**: Handles errors by showing a toast notification.

- **onSuccess**: Handles success by navigating to the subreddit page, refreshing the router, and showing a success toast.

Editor Initialization

```
const initializeEditor = useCallback(async () => {
  const EditorJS = (await import("@editorjs/editorjs")).default;
  const Header = (await import("@editorjs/header")).default;
  const Embed = (await import("@editorjs/embed")).default;
  const Table = (await import("@editorjs/table")).default;
  const List = (await import("@editorjs/list")).default;
  const Code = (await import("@editorjs/code")).default;
  const LinkTool = (await import("@editorjs/link")).default;
  const InlineCode = (await import("@editorjs/inline-code")).default;
  const ImageTool = (await import("@editorjs/image")).default;

  if (!ref.current) {
    const editor = new EditorJS({
      holder: "editor",
      onReady() {
        ref.current = editor;
      },
      placeholder: "Type here to write your post...",
      inlineToolbar: true,
      data: { blocks: [] },
      tools: {
        header: Header,
        linkTool: {
          class: LinkTool,
          config: {
            endpoint: "/api/link",
          },
        },
        image: {
          class: ImageTool,
          config: {
            uploader: {
              async uploadByFile(file: File) {
                const [res] = await uploadFiles([file], "imageUploader");

                return {
                  success: 1,
                  file: {
                    url: res.fileUrl,
                  },
                };
              },
            },
          },
        },
        list: List,
        code: Code,
        inlineCode: InlineCode,
        table: Table,
        embed: Embed,
      },
    });
  }
}, []);
```

- **initializeEditor**: A function to dynamically import and initialize EditorJS and its tools.
- Imports various tools like Header, Embed, Table, List, Code, LinkTool, InlineCode, and ImageTool.
- Configures the EditorJS instance with these tools and sets up the editor.

Error Handling Effect

```
useEffect(() => {
  if (Object.keys(errors).length) {
```

```
    for (const [_key, value] of Object.entries(errors)) {
      toast({
        title: "Something went wrong.",
        description: (value as { message: string }).message,
        variant: "destructive",
      });
    }
  }
}, [errors]);
```

- Uses `useEffect` to watch for validation errors.
- If there are errors, displays a toast notification for each error.

Component Mounting Effect

```
useEffect(() => {
  if (typeof window !== "undefined") {
    setIsMounted(true);
  }
}, []);
```

- Uses `useEffect` to set `isMounted` to true when the component is mounted.
- Ensures the code runs only on the client-side.

Editor Initialization Effect

```
useEffect(() => {
  const init = async () => {
    await initializeEditor();

    setTimeout(() => {
      _titleRef?.current?.focus();
    }, 0);
  };

  if (isMounted) {
    init();

    return () => {
      ref.current?.destroy();
      ref.current = undefined;
    };
  }
}, [isMounted, initializeEditor]);
```

- Uses `useEffect` to initialize the editor when the component is mounted.
- Calls `initializeEditor` and sets focus on the title input.
- Cleans up the editor instance when the component is unmounted.

Form Submission Handler

```
async function onSubmit(data: FormData) {
  const blocks = await ref.current?.save();

  const payload: PostCreationRequest = {
    title: data.title,
    content: blocks,
    subredditId,
    subredditName,
  };

  createPost(payload);
}
```

- Defines the `onSubmit` function to handle form submission.
- Saves the editor content and creates a payload for the post.
- Triggers the `createPost` mutation with the payload.

Component Return Structure

```
if (!isMounted) {
  return null;
}

const { ref: titleRef, ...rest } = register("title");

return (
  <div className="w-full p-4 bg-zinc-50 rounded-lg border border-zinc-200">
    <form
      id="subreddit-post-form"
      className="w-fit"
      onSubmit={handleSubmit(onSubmit)}
    >
      <div className="prose prose-stone dark:prose-invert">
        <TextareaAutosize
          ref={(e) => {
            titleRef(e);
            _titleRef.current = e;
          }}
          {...rest}
          placeholder="Title"
          className="w-full resize-none appearance-none overflow-hidden bg-transparent text-5xl font-bold focus:outline
        />
        <div id="editor" className="min-h-[500px]" />
        <p className="text-sm text-gray-500">
          Use{" "}
          <kbd className="rounded-md border bg-muted px-1 text-xs uppercase">
            Tab
          </kbd>{" "}
          to open the command menu.
        </p>
      </div>
    </form>
  </div>
);
```

- Renders the form and editor only if the component is mounted.
- Destructures `register` to get the `titleRef` and other props for the title input.
- Uses `TextareaAutosize` for the title input with styling and placeholder.
- Renders the EditorJS instance inside a `div` with an ID of "editor".
- Displays a message about using the Tab key to open the command menu.

API

```
export async function POST(req: Request) {
  try {
    const body = await req.json();

    const { title, content, subredditId, subredditName } =
      PostValidator.parse(body);

    const session = await getAuthSession();

    const knockClient = new Knock(process.env.KNOCK_SECRET_API_KEY);

    if (!session?.user) {
      return new Response("Unauthorized", { status: 401 });
    }
```

- **POST**: Declares an asynchronous function to handle POST requests.
- **body**: Parses the JSON body of the request.
- **PostValidator.parse**: Validates the request body against the `PostValidator` schema.
- **session**: Retrieves the current user session.

- **knockClient**: Initializes a Knock client with the secret API key.
- Checks if the user is authenticated. If not, returns an `Unauthorized` response.

```
  // verify user is subscribed to passed subreddit id
  const subscription = await db.subscription.findFirst({
    where: {
      subredditId,
      userId: session.user.id,
    },
  });

  if (!subscription) {
    return new Response("Subscribe to post", { status: 403 });
  }
```

- **subscription**: Queries the database to check if the user is subscribed to the subreddit.
- If the user is not subscribed, returns a `403 Forbidden` response.

```
  await db.post.create({
    data: {
      title,
      content,
      authorId: session.user.id,
      subredditId,
    },
  });
```

- **db.post.create**: Creates a new post in the database with the provided data.

Notification Trigger

```
  const otherUsers = await db.user.findMany({
    where: {
      id: {
        not: session.user.id,
      },
      subscriptions: {
        some: {
          subredditId: subredditId,
        },
      },
    },
    select: {
      id: true,
    },
  });

  await knockClient.workflows.trigger("new-post", {
    actor: session.user.id,
    recipients: otherUsers.map((user) => user.id),
    data: {
      data: {
        subredditName,
        title,
      },
    },
  });
```

- **otherUsers**: Queries the database to find other users who are subscribed to the subreddit.
- **knockClient.workflows.trigger**: Triggers the "new-post" workflow in Knock, notifying the other users about the new post.
-

```
    return new Response("OK");
  } catch (error) {
    if (error instanceof z.ZodError) {
```

```
      return new Response(error.message, { status: 400 });
    }

    return new Response(
      "Could not post to subreddit at this time. Please try later",
      { status: 500 }
    );
  }
}
```

- **return new Response("OK")**: Returns an "OK" response if the post creation and notification were successful.
- **catch**: Catches any errors that occur during the process.
  - If the error is a `ZodError`, returns a `400 Bad Request` response with the error message.
  - For other errors, returns a `500 Internal Server Error` response with a generic error message.

Display Post

In this section, we will use the `PostFeed.tsx` component to display the post. This component is covered in section Community Detail Page

Post

Import Statements

```
'use client'

import { FC, useRef } from 'react'
import EditorOutput from './EditorOutput'
import PostVoteClient from './post-vote/PostVoteClient'
```

- **useRef**: Hook from React to create a reference.
- **EditorOutput**: Component to render the post content.
- **PostVoteClient**: Component to handle post voting.

```
type PartialVote = Pick<Vote, 'type'>

interface PostProps {
  post: Post & {
    author: User
    votes: Vote[]
  }
  votesAmt: number
  subredditName: string
  currentVote?: PartialVote
  commentAmt: number
}
```

- **PartialVote**: Defines a type that includes only the `type` property from the `Vote` type.
- **PostProps**: Interface for the props that the `Post` component expects:
  - **post**: A post object that includes the author and votes.
  - **votesAmt**: The total number of votes.
  - **subredditName**: The name of the subreddit.
  - **currentVote**: The current user's vote (optional).
  - **commentAmt**: The number of comments on the post.

```
const Post: FC<PostProps> = ({
  post,
  votesAmt: _votesAmt,
  currentVote: _currentVote,
  subredditName,
  commentAmt,
}) => {
  const pRef = useRef<HTMLParagraphElement>(null)
```

- Declares the `Post` functional component using `React.FC` with `PostProps`.

- Destructures the props and initializes a ref for the paragraph element.

```
return (
  <div className='rounded-md bg-white shadow'>
    <div className='px-6 py-4 flex justify-between'>
      <PostVoteClient
        postId={post.id}
        initialVotesAmt={_votesAmt}
        initialVote={_currentVote?.type}
      />

      <div className='w-0 flex-1'>
        <div className='max-h-40 mt-1 text-xs text-gray-500'>
          {subredditName ? (
            <>
              <a
                className='underline text-zinc-900 text-sm underline-offset-2'
                href={`/r/${subredditName}`}>
                r/{subredditName}
              </a>
              <span className='px-1'>•</span>
            </>
          ) : null}
          <span>Posted by u/{post.author.username}</span>{' '}
          {formatTimeToNow(new Date(post.createdAt))}
        </div>
        <a href={`/r/${subredditName}/post/${post.id}`}>
          <h1 className='text-lg font-semibold py-2 leading-6 text-gray-900'>
            {post.title}
          </h1>
        </a>

        <div
          className='relative text-sm max-h-40 w-full overflow-clip'
          ref={pRef}>
          <EditorOutput content={post.content} />
          {pRef.current?.clientHeight === 160 ? (
            // blur bottom if content is too long
            <div className='absolute bottom-0 left-0 h-24 w-full bg-gradient-to-t from-white to-transparent'></div>
          ) : null}
        </div>
      </div>
    </div>

    <div className='bg-gray-50 z-20 text-sm px-4 py-4 sm:px-6'>
      <Link
        href={`/r/${subredditName}/post/${post.id}`}
        className='w-fit flex items-center gap-2'>
        <MessageSquare className='h-4 w-4' /> {commentAmt} comments
      </Link>
    </div>
  </div>
)
}
export default Post
```

- **Outer Container**: A `div` with rounded corners, a white background, and a shadow.
- **Header Section**: Contains the `PostVoteClient` component and post metadata.
  - **PostVoteClient**: Handles voting functionality.
  - **Metadata**: Displays the subreddit name, author, and post creation time.
- **Post Title and Content**:
  - **Title**: A clickable title that navigates to the post detail page.
  - **Content**: Renders the post content using the `EditorOutput` component.
  - **Blur Effect**: Adds a blur effect if the content is too long.
- **Footer Section**: Contains a link to the comments section of the post.
  - **Link**: Navigates to the post detail page with comments.

- **MessageSquare**: Icon indicating the number of comments.

Voting System

This functionality has been briefly cover in the Post as the mean to render upvote and downvote. We split the `PostVotes` into a client and a server component to allow for dynamic data fetching inside of this component, allowing for faster page loads via suspense streaming. We also have to option to fetch this info on a page-level and pass it in. It also supports pre-fetched data passed in as props for flexibility in different use cases.

Voting on client side

```
### Detailed Explanation of the Code

- **Component Definition**:
  - The `PostVoteClient` component is a functional component that handles user voting (upvote/downvote) on a post.

- **Props Interface**:
  - `PostVoteClientProps` defines the props for the `PostVoteClient` component:
    - `postId`: The ID of the post being voted on.
    - `initialVotesAmt`: The initial number of votes the post has.
    - `initialVote`: The user's initial vote on the post, if any.

- **State Management**:
  - `votesAmt`: State to manage the current number of votes, initialized with `initialVotesAmt`.
  - `currentVote`: State to manage the current vote of the user, initialized with `initialVote`.
  - `prevVote`: A reference to the previous vote using `usePrevious` hook to manage state changes.

- **useEffect Hook**:
  - Ensures the `currentVote` state is synchronized with the `initialVote` prop whenever it changes.

```typescript
useEffect(() => {
  setCurrentVote(initialVote)
}, [initialVote])
```
```

- **useMutation Hook**:
  - Utilizes `useMutation` from `@tanstack/react-query` to handle the voting logic with server communication.
  - `mutationFn`: Defines the function to execute when a vote is cast, sending a PATCH request to the server.
  - `onError`: Handles errors by reverting the vote and displaying appropriate messages or actions, like a login prompt if the user is unauthorized.
  - `onMutate`: Optimistically updates the vote state before the server response, handling both voting and unvoting scenarios.

```
const { mutate: vote } = useMutation({
  mutationFn: async (type: VoteType) => {
    const payload: PostVoteRequest = {
      voteType: type,
      postId: postId,
    }
    await axios.patch('/api/subreddit/post/vote', payload)
  },
  onError: (err, voteType) => {
    if (voteType === 'UP') setVotesAmt((prev) => prev - 1)
    else setVotesAmt((prev) => prev + 1)
    setCurrentVote(prevVote)
    if (err instanceof AxiosError) {
      if (err.response?.status === 401) {
        return loginToast()
      }
    }
    return toast({
      title: 'Something went wrong.',
      description: 'Your vote was not registered. Please try again.',
      variant: 'destructive',
    })
  },
  onMutate: (type: VoteType) => {
    if (currentVote === type) {
      setCurrentVote(undefined)
```

```
      if (type === 'UP') setVotesAmt((prev) => prev - 1)
      else if (type === 'DOWN') setVotesAmt((prev) => prev + 1)
    } else {
      setCurrentVote(type)
      if (type === 'UP') setVotesAmt((prev) => prev + (currentVote ? 2 : 1))
      else if (type === 'DOWN')
        setVotesAmt((prev) => prev - (currentVote ? 2 : 1))
    }
  },
})
```

- **Return JSX**
  - **Container**:
    - A `div` with flex properties to arrange the upvote button, score display, and downvote button vertically.
  - **Upvote Button**:
    - A `Button` component that triggers the `vote` function with 'UP' when clicked.
    - The `ArrowBigUp` icon changes color if the current vote is 'UP'.
  - **Score Display**:
    - A `p` element displaying the current number of votes.
  - **Downvote Button**:
    - A `Button` component that triggers the `vote` function with 'DOWN' when clicked.
    - The `ArrowBigDown` icon changes color if the current vote is 'DOWN'.

```
return (
  <div className='flex flex-col gap-4 sm:gap-0 pr-6 sm:w-20 pb-4 sm:pb-0'>
    <Button
      onClick={() => vote('UP')}
      size='sm'
      variant='ghost'
      aria-label='upvote'>
      <ArrowBigUp
        className={cn('h-5 w-5 text-zinc-700', {
          'text-emerald-500 fill-emerald-500': currentVote === 'UP',
        })}
      />
    </Button>
    <p className='text-center py-2 font-medium text-sm text-zinc-900'>
      {votesAmt}
    </p>
    <Button
      onClick={() => vote('DOWN')}
      size='sm'
      className={cn({
        'text-emerald-500': current
        'text-emerald-500': currentVote === 'DOWN',
      })}
      variant='ghost'
      aria-label='downvote'>
      <ArrowBigDown
        className={cn('h-5 w-5 text-zinc-700', {
          'text-red-500 fill-red-500': currentVote === 'DOWN',
        })}
      />
    </Button>
  </div>
)
```

- **Conditional Classes**:
  - `cn` utility function is used to conditionally apply classes based on the `currentVote` state, changing the icon's color when a vote is cast.

Voting on Server Side

- **Component Definition**:

- `PostVoteServer` is an asynchronous functional component that handles server-side logic for post voting. It fetches or receives vote data and passes it to the `PostVoteClient` component.

```
const session = await getAuthSession()
```

- **Local Variables**:
  - `_votesAmt`: A local variable to store the computed number of votes.
  - `_currentVote`: A local variable to store the current user's vote type.
- **Data Fetching and Processing**:
  - If `getData` is provided, it fetches the post data, calculates the total votes, and determines the current user's vote.
  - If `getData` is not provided, it uses the initial values passed via props.

```
if (getData) {
  const post = await getData()
  if (!post) return notFound()

  _votesAmt = post.votes.reduce((acc, vote) => {
    if (vote.type === 'UP') return acc + 1
    if (vote.type === 'DOWN') return acc - 1
    return acc
  }, 0)

  _currentVote = post.votes.find(
    (vote) => vote.userId === session?.user?.id
  )?.type
} else {
  _votesAmt = initialVotesAmt!
  _currentVote = initialVote
}
```

- **Return JSX**:
  - The component returns the `PostVoteClient` component, passing the post ID, computed votes amount, and current vote as props.

```
return (
  <PostVoteClient
    postId={postId}
    initialVotesAmt={_votesAmt}
    initialVote={_currentVote}
  />
)
```

API

This function used Redis - a third party data platform - to cache user data for better performance

- **Imports**:
  - Various utility functions and libraries are imported for authentication, database interaction, Redis caching, and input validation.

```
import { getAuthSession } from '@/lib/auth'
import { db } from '@/lib/db'
import { redis } from '@/lib/redis'
import { PostVoteValidator } from '@/lib/validators/vote'
import { CachedPost } from '@/types/redis'
import { z } from 'zod'
```

- **Constants**:
  - `CACHE_AFTER_UPVOTES`: A threshold for the number of upvotes required before caching the post in Redis.

```
const CACHE_AFTER_UPVOTES = 1
```

- **PATCH Handler**:

- An asynchronous function `PATCH` handles PATCH requests for voting on a post.

```
export async function PATCH(req: Request) {
  try {
    const body = await req.json()

    const { postId, voteType } = PostVoteValidator.parse(body)

    const session = await getAuthSession()

    if (!session?.user) {
      return new Response('Unauthorized', { status: 401 })
    }
```

- **Fetch Existing Vote**:
  - Checks if the user has already voted on the post.

```
    const existingVote = await db.vote.findFirst({
      where: {
        userId: session.user.id,
        postId,
      },
    })
```

- **Fetch Post Data**:
  - Retrieves the post data along with its author and votes.

```
    const post = await db.post.findUnique({
      where: {
        id: postId,
      },
      include: {
        author: true,
        votes: true,
      },
    })

    if (!post) {
      return new Response('Post not found', { status: 404 })
    }
```

- **Handle Existing Vote**:
  - If the user's vote matches the existing vote, it deletes the vote.
  - If the vote type is different, it updates the vote.

```
    if (existingVote) {
      if (existingVote.type === voteType) {
        await db.vote.delete({
          where: {
            userId_postId: {
              postId,
              userId: session.user.id,
            },
          },
        })
      } else {
        await db.vote.update({
          where: {
            userId_postId: {
              postId,
              userId: session.user.id,
            },
          },
          data: {
            type: voteType,
```

```
        },
      })
    }
  } else {
    await db.vote.create({
      data: {
        type: voteType,
        userId: session.user.id,
        postId,
      },
    })
  }
```

- **Recount and Cache Votes**:
  - Recounts the votes and caches the post if the upvote count exceeds the threshold.

```
  const votesAmt = post.votes.reduce((acc, vote) => {
    if (vote.type === 'UP') return acc + 1
    if (vote.type === 'DOWN') return acc - 1
    return acc
  }, 0)

  if (votesAmt >= CACHE_AFTER_UPVOTES) {
    const cachePayload: CachedPost = {
      authorUsername: post.author.username ?? '',
      content: JSON.stringify(post.content),
      id: post.id,
      title: post.title,
      currentVote: voteType,
      createdAt: post.createdAt,
    }

    await redis.hset(`post:${postId}`, cachePayload)
  }

  return new Response('OK')
} catch (error) {
  (error)
  if (error instanceof z.ZodError) {
    return new Response(error.message, { status: 400 })
  }

  return new Response(
    'Could not post to subreddit at this time. Please try later',
    { status: 500 }
  )
  }
}
```

#Comment

Comment Section

Validator

This code defines a schema for validating comments using `zod`, a TypeScript-first schema declaration and validation library.

- **Schema Definition**:
  - `CommentValidator` is a schema that validates the structure of a comment object.
  - `postId`: A required string representing the ID of the post to which the comment belongs.
  - `text`: A required string representing the content of the comment.
  - `replyToId`: An optional string representing the ID of the comment to which this comment is a reply.

```
export const CommentValidator = z.object({
  postId: z.string(),
  text: z.string(),
```

```
  replyToId: z.string().optional(),
});
```

- `CommentRequest`:
  - This type is inferred from the `CommentValidator` schema. It represents the shape of the data that matches the schema.
  - `CommentRequest` type can be used throughout the application to ensure that objects conform to the validated schema.

Required Components

CommentSections.tsx

This code defines a React component called `CommentsSection` that handles the display and functionality of comments on a post.

- **Type Definitions**
- `ExtendedComment`: A type that extends `Comment` to include `votes`, `author`, and `replies`.
- `ReplyComment`: A type that extends `Comment` to include `votes` and `author`.
- `CommentsSectionProps`: An interface for the properties of the `CommentsSection` component.

```
type ExtendedComment = Comment & {
  votes: CommentVote[];
  author: User;
  replies: ReplyComment[];
};

type ReplyComment = Comment & {
  votes: CommentVote[];
  author: User;
};

interface CommentsSectionProps {
  postId: string;
  comments: ExtendedComment[];
}
```

- **Function Definition**:
  - An asynchronous function component that takes `CommentsSectionProps` as its parameter.

```
const CommentsSection = async ({ postId }: CommentsSectionProps) => {
  // Fetch the current session
  const session = await getAuthSession();

  // Fetch comments from the database
  const comments = await db.comment.findMany({
    where: {
      postId: postId,
      replyToId: null, // only fetch top-level comments
    },
    include: {
      author: true,
      votes: true,
      replies: {
        // first level replies
        include: {
          author: true,
          votes: true,
        },
      },
    },
  });
```

- **Rendering**:
  - Returns a `div` containing the comments section layout.
  - Renders the `CreateComment` component for adding new comments.

- Iterates over the top-level comments and their replies, rendering each using the `PostComment` component.

```
return (
  <div className="flex flex-col gap-y-4 mt-4">
    <hr className="w-full h-px my-6" />

    {/* Render the CreateComment component */}
    <CreateComment postId={postId} />

    <div className="flex flex-col gap-y-6 mt-4">
      {comments
        .filter((comment) => !comment.replyToId)
        .map((topLevelComment) => {
          // Calculate the total votes for the top-level comment
          const topLevelCommentVotesAmt = topLevelComment.votes.reduce(
            (acc, vote) => {
              if (vote.type === "UP") return acc + 1;
              if (vote.type === "DOWN") return acc - 1;
              return acc;
            },
            0
          );

          // Find the current user's vote on the top-level comment
          const topLevelCommentVote = topLevelComment.votes.find(
            (vote) => vote.userId === session?.user.id
          );

          return (
            <div key={topLevelComment.id} className="flex flex-col">
              <div className="mb-2">
                {/* Render the PostComment component for the top-level comment */}
                <PostComment
                  comment={topLevelComment}
                  currentVote={topLevelCommentVote}
                  votesAmt={topLevelCommentVotesAmt}
                  postId={postId}
                />
              </div>

              {/* Render replies */}
              {topLevelComment.replies
                .sort((a, b) => b.votes.length - a.votes.length) // Sort replies by most liked
                .map((reply) => {
                  // Calculate the total votes for the reply
                  const replyVotesAmt = reply.votes.reduce((acc, vote) => {
                    if (vote.type === "UP") return acc + 1;
                    if (vote.type === "DOWN") return acc - 1;
                    return acc;
                  }, 0);

                  // Find the current user's vote on the reply
                  const replyVote = reply.votes.find(
                    (vote) => vote.userId === session?.user.id
                  );

                  return (
                    <div
                      key={reply.id}
                      className="ml-2 py-2 pl-4 border-l-2 border-zinc-200"
                    >
                      {/* Render the PostComment component for the reply */}
                      <PostComment
                        comment={reply}
                        currentVote={replyVote}
                        votesAmt={replyVotesAmt}
                        postId={postId}
```

```
                        />
                      </div>
                    );
                  })}
                </div>
              );
            })}
          </div>
        </div>
      );
};

export default CommentsSection;
```

- Comments are fetched from the database, including their authors, votes, and replies.
- Only top-level comments (comments without a `replyToId`) are fetched initially.
- **Rendering Comments**:
  - The `CreateComment` component is rendered to allow users to add new comments.
  - Comments are displayed with their respective votes and replies.
  - Replies are sorted by the number of votes and displayed nested within their parent comments.
- **Vote Calculation**:
  - Total votes for each comment are calculated.
  - The user's vote for each comment is identified to manage vote state in the UI.

This component structure ensures a clear and organized display of comments

CreateComment.tsx

```
const CreateComment: FC<CreateCommentProps> = ({ postId, replyToId }) => {
  const [input, setInput] = useState<string>("");
  const router = useRouter();
  const { loginToast } = useCustomToasts();
```

- The `CreateComment` component is defined as a functional component (FC) that takes `postId` and optional `replyToId` as props.
- The component maintains an `input` state to manage the text of the comment.
- `router` is used for navigation.
- `loginToast` is a custom toast notification for prompting login.

```
const { mutate: comment, isLoading } = useMutation({
  mutationFn: async ({ postId, text, replyToId }: CommentRequest) => {
    const payload: CommentRequest = {
      postId,
      text,
      replyToId,
    };

    const { data } = await axios.patch(
      `/api/subreddit/post/comment/`,
      payload
    );
    return data;
  },

  onError: (err) => {
    if (err instanceof AxiosError) {
      if (err.response?.status === 401) {
        return loginToast();
      }
    }

    return toast({
      title: "Something went wrong.",
      description: "Comment wasn't created successfully. Please try again.",
      variant: "destructive",
```

```
    });
  },
  onSuccess: () => {
    router.refresh();
    setInput("");
  },
});
```

- `useMutation` is used to handle the comment creation process:
  - `mutationFn` defines the asynchronous function to send a PATCH request to create a comment.
  - `onError` handles errors, displaying a login toast if unauthorized or a generic error toast otherwise.
  - `onSuccess` refreshes the page and clears the input field.

```
return (
  <div className="grid w-full gap-1.5">
    <Label htmlFor="comment">Your comment</Label>
    <div className="mt-2">
      <Textarea
        id="comment"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        rows={1}
        placeholder="What are your thoughts?"
      />

      <div className="mt-2 flex justify-end">
        <Button
          isLoading={isLoading}
          disabled={input.length === 0}
          onClick={() => comment({ postId, text: input, replyToId })}
        >
          Post
        </Button>
      </div>
    </div>
  </div>
);
```

- The component renders a form for creating a comment:
  - A `Label` for the textarea.
  - A `Textarea` for entering the comment, with an `onChange` handler to update the input state.
  - A `Button` to submit the comment, which is disabled if the input is empty and shows a loading state while the mutation is in progress.
  - When the button is clicked, the `comment` mutation function is called with the current `postId`, `input` text, and `replyToId`.

API

```
import { getAuthSession } from "@/lib/auth";
import { db } from "@/lib/db";
import { CommentValidator } from "@/lib/validators/comment";
import { Knock } from "@knocklabs/node";
import { z } from "zod";
```

- These statements import various modules and libraries used in the code:
  - `getAuthSession`: Function to get the current authentication session.
  - `db`: Instance of a database client.
  - `CommentValidator`: Schema validator for the comment input.
  - `Knock`: Library for sending notifications.
  - `z`: Library for schema validation.
- **Function Definition**:
  - `PATCH` is an asynchronous function that handles HTTP PATCH requests.

- **Parsing Request Body**:

```
const body = await req.json();
const { postId, text, replyToId } = CommentValidator.parse(body);
```

  - The request body is parsed as JSON.
  - `postId`, `text`, and `replyToId` are extracted and validated using `CommentValidator`.

- **Getting User Session and Knock Client**:

```
const session = await getAuthSession();
const knockClient = new Knock(process.env.KNOCK_SECRET_API_KEY);
```

  - The current user session is retrieved.
  - A Knock client is instantiated using an API key from environment variables.

- **Authorization Check**:

```
if (!session?.user) {
  return new Response("Unauthorized", { status: 401 });
}
```

  - If the user is not authenticated, return a 401 Unauthorized response.

- **Creating a New Comment**:

```
await db.comment.create({
  data: {
    text,
    postId,
    authorId: session.user.id,
    replyToId,
  },
});
```

  - A new comment is created in the database with the provided data.

- **Fetching Post and Author Details**:

```
const post = await db.post.findUnique({
  where: {
    id: postId,
  },
});

const postAuthor = await db.post.findUnique({
  where: {
    id: postId,
  },
  select: {
    authorId: true,
  },
});

const subreddit = await db.subreddit.findUnique({
  where: {
    id: post?.subredditId,
  },
});
```

  - The post and its author's details are fetched from the database.
  - The subreddit details are also fetched.

- **Triggering Notification**:

```
if (
  postAuthor?.authorId !== session.user.id &&
  postAuthor?.authorId !== undefined
) {
  await knockClient.workflows.trigger("comment", {
    actor: session.user.id,
    recipients: [postAuthor.authorId],
    data: {
      title: post?.title,
      preview: text.slice(0, 20),
      subredditName: subreddit?.name,
    },
  });
}
```

  - If the comment author is not the post author, a notification is triggered using Knock.

- **Response Handling**:

```
return new Response("OK");
} catch (error) {
if (error instanceof z.ZodError) {
  return new Response(error.message, { status: 400 });
}

return new Response(
  "Could not post to subreddit at this time. Please try later",
  { status: 500 }
);
}
```

  - If no errors occur, a response with "OK" is returned.
  - If a `ZodError` occurs, a 400 response with the error message is returned.
  - For other errors, a 500 response is returned with a generic error message.

Voting for Comment

Required Components

CommentVotes.tsx

```
interface CommentVotesProps {
  commentId: string
  votesAmt: number
  currentVote?: PartialVote
}

type PartialVote = Pick<CommentVote, 'type'>
```

- Defines the TypeScript interface for the component props.
- `PartialVote` is a type that only includes the `type` property from `CommentVote`.

```
const CommentVotes: FC<CommentVotesProps> = ({
  commentId,
  votesAmt: _votesAmt,
  currentVote: _currentVote,
}) => {
  const { loginToast } = useCustomToasts()
  const [votesAmt, setVotesAmt] = useState<number>(_votesAmt)
  const [currentVote, setCurrentVote] = useState<PartialVote | undefined>(
    _currentVote
  )
  const prevVote = usePrevious(currentVote)
```

- The `CommentVotes` component is defined as a functional component (FC) that takes `commentId`, `votesAmt`, and an optional `currentVote` as props.
- The component maintains `votesAmt` and `currentVote` state variables.
- `prevVote` stores the previous state of `currentVote`.

```
const { mutate: vote } = useMutation({
  mutationFn: async (type: VoteType) => {
    const payload: CommentVoteRequest = {
      voteType: type,
      commentId,
    }

    await axios.patch('/api/subreddit/post/comment/vote', payload)
  },
  onError: (err, voteType) => {
    if (voteType === 'UP') setVotesAmt((prev) => prev - 1)
    else setVotesAmt((prev) => prev + 1)

    // reset current vote
    setCurrentVote(prevVote)

    if (err instanceof AxiosError) {
      if (err.response?.status === 401) {
        return loginToast()
      }
    }

    return toast({
      title: 'Something went wrong.',
      description: 'Your vote was not registered. Please try again.',
      variant: 'destructive',
    })
  },
  onMutate: (type: VoteType) => {
    if (currentVote?.type === type) {
      // User is voting the same way again, so remove their vote
      setCurrentVote(undefined)
      if (type === 'UP') setVotesAmt((prev) => prev - 1)
      else if (type === 'DOWN') setVotesAmt((prev) => prev + 1)
    } else {
      // User is voting in the opposite direction, so subtract 2
      setCurrentVote({ type })
      if (type === 'UP') setVotesAmt((prev) => prev + (currentVote ? 2 : 1))
      else if (type === 'DOWN')
        setVotesAmt((prev) => prev - (currentVote ? 2 : 1))
    }
  },
})
```

- `useMutation` is used to handle the vote process:
  - `mutationFn` defines the asynchronous function to send a PATCH request to register the vote.
  - `onError` handles errors, reverting the vote and displaying a login toast if unauthorized or a generic error toast otherwise.
  - `onMutate` updates the vote state optimistically, handling cases where the user votes the same way or the opposite way.

```
return (
  <div className='flex gap-1'>
    {/* upvote */}
    <Button
      onClick={() => vote('UP')}
      size='xs'
      variant='ghost'
      aria-label='upvote'>
      <ArrowBigUp
        className={cn('h-5 w-5 text-zinc-700', {
          'text-emerald-500 fill-emerald-500': currentVote?.type === 'UP',
        })}
      />
```

```
      </Button>

      {/* score */}
      <p className='text-center py-2 px-1 font-medium text-xs text-zinc-900'>
        {votesAmt}
      </p>

      {/* downvote */}
      <Button
        onClick={() => vote('DOWN')}
        size='xs'
        className={cn({
          'text-emerald-500': currentVote?.type === 'DOWN',
        })}
        variant='ghost'
        aria-label='downvote'>
        <ArrowBigDown
          className={cn('h-5 w-5 text-zinc-700', {
            'text-red-500 fill-red-500': currentVote?.type === 'DOWN',
          })}
        />
      </Button>
    </div>
  )
```

- The component renders a voting interface:
  - An upvote button that triggers the `vote` mutation with 'UP' when clicked. The button icon changes color if the current vote is 'UP'.
  - A paragraph displaying the current vote amount.
  - A downvote button that triggers the `vote` mutation with 'DOWN' when clicked. The button icon changes color if the current vote is 'DOWN'.

VI. Chat-Fun

1)Configuration for Firebase in a Chat Application

The first part of building a chat application with React.js and Firebase involves configuring Firebase services. This is done in the config.js file, where we import necessary functions from Firebase SDKs and initialize the app with our Firebase project's configuration.

// Import the functions you need from the SDKs you need

import { initializeApp } *from* 'firebase/app';

import { getAuth } *from* 'firebase/auth';

import { getFirestore } *from* 'firebase/firestore';

// Your web app's Firebase

const firebaseConfig = {

apiKey: "AIzaSyA7Opqr5hrW_jWuB7HCRj9FO7E9gUVM_zI",

authDomain: "chat-library-f1aa1.firebaseapp.com",

projectId: "chat-library-f1aa1",

storageBucket: "chat-library-f1aa1.appspot.com",

messagingSenderId: "111225498935",

appId: "1:111225498935:web:165e3b94fc5106d02bbeb2",

measurementId: "G-HNQ0R8M850"

};

// Initialize Firebase

const app = initializeApp(firebaseConfig);

const auth = getAuth(app);

const db = getFirestore(app);

// Export the initialized services for use in other parts of your app

export { db, auth };

export default app;

By setting up Firebase with this configuration, the chat application can leverage Firebase's powerful features for authentication and real-time data storage, providing a solid foundation for building a robust and scalable chat platform.

2)Google Login Implementation

implementing the login functionality using Firebase Authentication with Google as the provider. This process includes creating a login component in React that allows users to sign in with their Google accounts. The relevant data is stored in Firebase Firestore, and upon successful login, the user is redirected to the chat page.
Google Provider Setup: A GoogleAuthProvider instance is created to handle Google sign-ins.
try {

const { _tokenResponse, user } = await signInWithPopup(auth, googleProvider);

if (_tokenResponse?.isNewUser) {

await addDoc(collection(db, 'users'), {

displayName: user.displayName,

email: user.email,

uid: user.uid,

photoURL: user.photoURL,

providerId: _tokenResponse.providerId,

keywords: generateKeywords(user.displayName)

});

}

} catch (error) {

console.error("Error during sign-in with Google: ", error);

}

handleGoogleLogin Function: This async function handles the sign-in process using a popup.

- signInWithPopup(auth, googleProvider): Opens a popup for Google sign-in.
- If the user is new (_tokenResponse?.isNewUser), their details (displayName, email, uid, photoURL, providerId, and generated keywords) are added to the 'users' collection in Firestore using addDoc.

Creating Context: AuthContext is created to manage and provide user authentication state across the application.

1. **Managing State**: The component uses React hooks to manage the user state and a loading state (isLoading).
2. **Handling Auth State Changes**:
   - **Timeout for Login**: A timeout is set to handle cases where the login process takes too long due to network issues or server unresponsiveness.
   - **onAuthStateChanged**: This listener handles authentication state changes. If a user is logged in, their details are stored in the user state, and they are navigated to the home (chat) page. If not, an error message is logged, and the user is redirected to the login page.
3. **Providing Context**: The AuthContext.Provider wraps the children components and provides the user state and a loading spinner (Spin) while the authentication state is being determined.

By integrating Google login using Firebase and setting up the authentication provider, the chat application ensures a seamless and secure user authentication process. Upon successful login, users are redirected to the chat page, while any issues during the process are handled gracefully with appropriate error messages and redirects.

3)Service Functions

utility functions for interacting with Firebase Firestore and generating keywords for user search. These functions are essential for managing documents in the Firestore database and enhancing the search functionality within the chat application.
The service.js file contains two main functions: addDocument for adding documents to Firestore, and generateKeywords for creating search keywords based on user display names.

1. **addDocument Function**:
   - **Parameters**: Takes col (collection name) and data (document data) as parameters.
   - **Firestore Operation**: Uses addDoc to add a new document to the specified collection in Firestore.
   - **Timestamp**: Adds a createAt field with a server-generated timestamp.
   - **Error Handling**: Catches and logs any errors that occur during the document addition process.
2. **generateKeywords Function**:

- **Parameters**: Takes displayName (user's display name) as a parameter.
- **Splitting Display Name**: Splits the display name into individual words, filtering out any empty strings.
- **Keyword Generation**:
  - Uses a permutation algorithm to generate all possible orderings of the words in the display name.
  - createKeywords function creates a list of keyword substrings for each permutation.
- **Keyword Combination**: Combines all generated keywords into a single array.

By implementing these utility functions, the chat application can efficiently manage user data in Firestore and provide enhanced search functionality based on user display names. The addDocument function ensures that new user data is correctly stored in the database, while the generateKeywords function facilitates quick and accurate user searches. (we use generateKeywords function for invite members in to group chat, because it does not support when we type a character, it will display full name you we should have attributes keywords to retrieve to the username for invite members)

4)Modals and Context for Chat Functionality

context and modal components that support various functionalities in the chat application, such as creating rooms and inviting members. These components leverage the Firebase Firestore for data management and integrate seamlessly with the rest of the application.

AppProvider.js

The AppProvider.js file provides the application context, including the state and logic for managing chat rooms and member invitations. **Context Creation:**

- AppContext is created using createContext(). This will hold the global state and functions related to the chat app.

State Variables:

- isAddRoomVisible and isInviteMemberVisible are booleans controlling the visibility of modals.
- selectedRoomId stores the ID of the currently selected room.

Fetch Rooms:

- roomsCondition is a memoized condition object used by useFirestore to fetch rooms where the current user (uid) is a member.
- rooms contains the fetched room data.

Fetch Selected Room and Members:

- selectedRoom is the room currently selected by the user.
- usersCondition is a memoized condition object used by useFirestore to fetch users who are members of the selected room.
- members contains the fetched user data.

Context Provider:

- AppContext.Provider wraps around the children components, providing the state and functions to manage rooms and member invitations.

AddRoomModal.js

The AddRoomModal.js file contains a modal component for creating new chat rooms. When the user submits the form, a new room is added to Firestore.

Code Breakdown:

1. **Imports:**
   - React, useState, and useContext for state management and context.
   - Modal, Form, Input from antd for the UI components.
   - AppContext and AuthContext to access the global state.
   - addDocument function to add data to Firestore.
2. **Modal Component:**
   - Uses useContext to get isAddRoomVisible and setIsAddRoomVisible from AppContext.
   - Uses Form from antd to create a form for the room name and description.
   - handleOk adds the new room to Firestore and hides the modal.
   - handleCancel hides the modal without making any changes.

InviteMemberModal.js

The InviteMemberModal.js file contains a modal component for inviting new members to a chat room. It includes a debounced search input for finding users.

Code Breakdown:

1. **Imports:**
   - React, useState, and useContext for state management and context.
   - Modal, Form, Select, Spin, Avatar from antd for the UI components.
   - AppContext and AuthContext to access the global state.
   - addDocument, db for Firestore operations.
   - debounce from lodash to create a debounced search function.
   - query, collection, where, limit, getDocs, doc, updateDoc from Firestore.

2. **DebounceSelect Component:**
   - A custom select component that fetches options with a debounce to prevent too many API calls.
   - Uses debounce to create debounceFetcher.
3. **fetchUserList Function:**
   - Fetches a list of users from Firestore based on the search input and excludes current members.
4. **InviteMemberModal Component:**
   - Uses useContext to get isInviteMemberVisible, setIsInviteMemberVisible, selectedRoomId, and selectedRoom from AppContext.
   - Uses useState to manage the selected users.
   - handleOk updates the room's member list in Firestore and hides the modal.
   - handleCancel hides the modal without making any changes.

useFirestore.js

The useFirestore.js file contains a custom hook for querying Firestore based on specific conditions and returns the fetched documents.

Code Breakdown:

1. **Imports:**
   - React, useState, useEffect for state management and lifecycle hooks.
   - Firestore functions (collection, query, where, orderBy, onSnapshot) for querying and listening to Firestore.
2. **useFirestore Hook:**
   - Takes collectionName and condition as arguments.
   - Sets up a Firestore query based on the provided condition.
   - Uses onSnapshot to listen for real-time updates to the collection.
   - Updates the documents state with the fetched data.

6)Chat Page Components

1. ChatRoom Component

- Purpose:
- Acts as the main container for the chat interface, integrating Sidebar and ChatWindow components.
- Components Used:
- Sidebar: Provides navigation and user information.
- ChatWindow: Displays messages and handles message input and sending.
- Features:
- Structured Layout: Utilizes Ant Design's Row and Col components for a responsive layout.
- Integration: Manages state and actions through AppProvider context, ensuring consistent data flow between components.

2. Sidebar Component

- Purpose:
- Displays user information and room list for navigation.
- Features:
- User Information: Shows user avatar and username retrieved from Firebase Authentication.
- Room List: Lists available rooms fetched from Firebase Firestore.
- Styling: Uses styled-components for a customizable, dark-themed interface.

3. ChatWindow Component

- Purpose:
- Displays messages and handles message input and sending functionality.
- Features:
- Real-time Messaging: Messages fetched and updated in real-time from Firebase Firestore.
- Message Display: Messages are sorted and displayed with sender information, timestamp, and content.
- Message Input: Provides an input field for users to type messages and send them to selected rooms.
- Integration: Uses AppProvider for managing room state, member invitations, and message sending.
- Styling:
- Styled with styled-components for consistent UI design and responsiveness.

4. UserInfo Component

- Purpose:
- Displays user profile information and manages logout functionality.
- Features:
- Profile Information: Shows user avatar and username fetched from Firebase Authentication.
- Logout Button: Allows users to sign out using Firebase authentication.
- Integration:
- Integrated with AuthContext from AuthProvider for seamless authentication management.

5. Message Component

- Purpose:
- Displays individual chat messages with sender details and timestamp.
- Features:
- Message Display: Shows sender's avatar, message content, and timestamp formatted for readability.
- Styling: Uses styled-components for structured message layout and visual presentation.
- Integration:
- Utilizes message data fetched from Firebase Firestore, integrated with ChatWindow for rendering.

6. RoomList Component

- Purpose:
- Lists available chat rooms and provides functionality to add new rooms.
- Features:
- Room Listing: Displays rooms fetched from Firebase Firestore using Firestore queries.
- Add Room Button: Allows users to create new rooms, triggering visibility changes in AddRoomModal.
- Integration:
- Uses AppProvider context for managing room state and navigation within the application.
- Styled with styled-components for a cohesive design language and UI consistency.

These components collectively provide a comprehensive chat application experience, leveraging React.js for frontend development, Firebase Firestore for real-time database operations, and Ant Design for UI components. Each component is designed to be modular, facilitating scalability and maintainability of the application

VII. Conclusion and Future Work

This online library management system provides a comprehensive solution for libraries to streamline their operations and enhance the user experience. The system features the following key capabilities:

User Features:

- Book Browse by Genre: Patrons can easily browse the library's collection by filtering books based on various genres, enabling them to quickly discover new titles that match their interests.
- Search Functionality: Patrons can search for specific books or authors using the intuitive search functionality, making it simple to locate the desired materials within the library's catalog.
- Social Forum: The system includes a social forum where patrons can engage with fellow readers, discuss books, and share recommendations, fostering a sense of community and enhancing the overall user experience.
- Chat Support: Patrons can directly communicate with library staff through the built-in chat feature, allowing them to receive immediate assistance with their queries or requests. ** Admin Features:**
- Book Management: Librarians can efficiently manage the library's collection, including adding new books, updating existing titles, and removing obsolete materials.
- User Management: Administrators can monitor and manage user accounts, track borrowing activities, and handle user-related tasks with ease.
- Numeric Analytics and Charts: The system provides detailed analytics and visual representations (such as charts and graphs) to help administrators gain valuable insights into the library's operations, usage patterns, and trends, enabling data-driven decision-making.

The implementation of this web-based library management application has resulted in several key benefits, including reduced paperwork, improved inventory tracking, and enhanced user satisfaction. Librarians now have the ability to manage the library's collection, monitor borrowing activities, and generate detailed reports with ease. Furthermore, patrons can conveniently search for books, place holds, and renew items from the comfort of their own devices, reducing the burden on library staff.

While the current online library management system has proven to be a valuable asset, there are several areas that can be explored for future enhancements and improvements:

- Integration with E-book Platforms: Expand the system's capabilities to include seamless integration with popular e-book platforms, allowing patrons to access digital content directly through the library's web application.
- Recommendation Engine: Incorporate a recommendation engine that suggests books or resources based on the user's borrowing history and preferences, fostering a more personalized user experience.
- Gamification and User Engagement: Explore the implementation of gamification elements, such as achievement badges or reading challenges, to incentivize patron engagement and encourage more active participation in the library's activities.
- Analytical Dashboards: Develop advanced analytical dashboards that provide library administrators with in-depth insights into usage patterns, popular titles, and other key performance indicators, enabling data-driven decision-making.
- Mobile App Development: Create a dedicated mobile application that offers a native user experience, further enhancing the accessibility and convenience of the library's services for patrons on the go.
- Multilingual Support: Implement multilingual functionality to cater to a diverse user base, making the library's resources accessible to individuals from different linguistic backgrounds.

- Collaboration and Resource Sharing: Explore the possibility of integrating the system with other libraries or academic institutions, enabling cross-institutional resource sharing and collaborative initiatives.

VIII. References

1. Abramov, D. (2022). React Documentation. React. https://reactjs.org/docs/getting-started.html
2. Node.js Foundation. (2023). Node.js Documentation. Node.js. https://nodejs.org/en/docs/
3. MySQL. (2023). MySQL Documentation. MySQL. https://dev.mysql.com/doc/
4. Shadab, H. (2021). Building a Full-Stack Web Application with React.js, Node.js, and MySQL. Towards Data Science. https://towardsdatascience.com/building-a-full-stack-web-application-with-react-js-node-js-and-mysql-e2e4c8dda4b3
5. Verma, R. (2022). Full-Stack Web Development with React.js and Node.js. Hackernoon. https://hackernoon.com/full-stack-web-development-with-reactjs-and-nodejs
6. Dahl, D. (2020). Building a Full-Stack Web Application with React, Node.js, and MySQL. Medium. https://medium.com/@dhan.dahl/building-a-full-stack-web-application-with-react-node-js-and-mysql-part-1-3a3c3b6e9c34
7. Vasquez, J. (2021). Building a CRUD App with React, Node.js, and MySQL. freeCodeCamp. https://www.freecodecamp.org/news/how-to-build-a-crud-app-with-react-node-js-and-mysql/