FRANKFURT UNIVERSITY OF APPLIED SCIENCES
VIETNAMESE-GERMAN UNIVERSITY

Frankfurt University of Applied Sciences
Faculty 2: Computer Science and Engineering

STUDYING WEB FULL-STACK TECHNOLOGIES AND APPLYING IN STUDENT LIFE SUPPORT SERVICE WEB APPLICATION DEVELOPMENT

Full name: Vu Hoang Tuan Anh
Matriculation number: 1403143
First supervisor: Dr. Tran Hong Ngoc
Second supervisor: Dr. Truong Dinh Huy

BACHELOR THESIS

Submitted in partial fulfillment of the requirements for the degree of Bachelor Engineering in study program Computer Science, Vietnamese - German University, 2024

Binh Duong, Viet Nam

Declaration

I hereby declare that the research presented in this thesis, carried out at both the Vietnamese-German University and the Frankfurt University of Applied Sciences, is my own original work. The thesis was completed under the guidance and supervision of Dr. Tran Hong Ngoc and Dr. Truong Dinh Huy. I further affirm that no part of this thesis has been included in any previous submission for a degree and that it does not violate any intellectual property rights.

Vu Hoang Tuan Anh
Program: Computer Science and Engineering
FraUAS Student ID: 1403143
VGU Student ID: 18812
Intake: 2020 - 2024
Frankfurt University of Applied Sciences
Vietnamese-German University

Date

Acknowledgments

First and foremost, I would like to extend my heartfelt gratitude to my two supervisors, Dr. Tran Hong Ngoc and Mr. Truong Dinh Huy, for dedicating their valuable time and effort to review and provide feedback on my thesis. Working closely with Dr. Tran Hong Ngoc over the years has made me appreciate her constant enthusiasm and approachable nature, which significantly boosted my confidence and comfort in completing this work. She was always available to offer guidance and constructive feedback whenever I needed assistance.

Moreover, I am also deeply thankful to the dedicated Computer Science and Engineering (CSE) assistants, whose thorough guidance throughout the thesis process and patience in addressing my questions were invaluable to my research.

Lastly, I want to express my sincere appreciation to all the lecturers at Vietnamese-German University (VGU) and Frankfurt University of Applied Sciences, whose teachings and guidance have been instrumental in shaping my academic journey. I am also incredibly grateful to my friends and family, whose unwavering support and encouragement have been a constant source of strength throughout my four years of study.

Abstract

The Student Life Support Service is a web application developed to streamline student support processes at the Vietnamese-German University (VGU). The system addresses the needs of students, dormitory staff, and administrators by facilitating efficient communication and ticket management for daily student life issues.

The key objectives of this project are to enhance student-staff interaction, simplify ticket resolution, and improve the overall support experience. Students can create, view, and manage support tickets, while staff members handle ticket processing and communication with students. Administrators oversee the entire system, managing users, roles, and system reports.

The application is built using a modern technology stack. The frontend, developed with ReactJS, Material UI, and Vite, incorporates a responsive design that ensures compatibility with various devices, including desktops, laptops, tablets, and smartphones. This ensures that users have a seamless experience regardless of the device they are using. The backend is powered by NodeJS, ExpressJS, and SocketIO for real-time communication, with JWT-based authentication (utilizing access and refresh tokens stored in a Redis in-memory database). The system's data is managed using PostgreSQL for robust and scalable database management.

The project adopts a modular and RESTful API-driven architecture to facilitate scalability and maintainability. The methodology involves iterative development with thorough testing at each stage to ensure the system meets functional and performance requirements.

Preliminary results indicate that the Student Life Support Service significantly improves the efficiency of support ticket management and fosters better communication between students and university staff. The system's modular design and responsiveness enable future enhancements, making it adaptable to evolving requirements at VGU.

Contents

1 Introduction **14**
1.1 Project Background . . . 14
1.2 Problem Statement . . . 15
1.3 Objectives of the Project . . . 15
1.4 Scope of the Project . . . 16
1.5 Thesis Structure . . . 17

2 Literature Review **18**
2.1 Existing solutions . . . 18
2.1.1 Group Chat-Based Systems (Current Solution at VGU) . . . 18
2.1.2 Existing University and Open-source Ticketing Systems . . . 18
2.1.3 Limitations of Existing Solutions in the University Context . . . 19
2.2 Technology Review . . . 20
2.2.1 Frontend: ReactJS, Material UI, Vite . . . 20
2.2.2 Backend: NodeJS, ExpressJS, SocketIO . . . 21
2.2.3 Authentication: JWT, Redis . . . 22
2.2.4 Database: PostgreSQL . . . 23
2.2.5 Responsive Web Design: Techniques and Tools . . . 23
2.3 Theoretical Background . . . 23
2.3.1 Ticket Management Systems . . . 23
2.3.2 Real-Time Communication Tools . . . 24
2.3.3 Web Application Development Best Practices . . . 24
2.4 Gap Analysis . . . 24
2.4.1 What is Missing from Existing Solutions . . . 24
2.4.2 How the Student Life Support Service Fills These Gaps . . . 25

3 System Design **26**
3.1 Functional Requirements . . . 26
3.2 Non-Functional Requirements . . . 29
3.3 Use Case Diagrams . . . 31
3.4 Process Workflow Diagrams . . . 34
3.5 Database Design . . . 35

3.5.1 ER Diagram . . . 35
3.5.2 User Entity . . . 35
3.5.3 Ticket Entity . . . 36
3.5.4 User_Ticket Relationship . . . 37
3.5.5 Ticket_Type Entity . . . 38
3.5.6 Ticket_Status Entity . . . 38
3.5.7 Audience_Type Entity . . . 39
3.5.8 Attachment Entity . . . 39
3.5.9 Rating Entity . . . 40
3.5.10 Feedback Entity . . . 40
3.5.11 Message Entity . . . 41
3.5.12 Dorm Entity . . . 41
3.5.13 Announcement Entity . . . 42
3.5.14 Notification Entity . . . 43
3.5.15 Role Entity . . . 43
3.5.16 Notification_Audience Relationship . . . 44
3.5.17 Log Entity . . . 44
3.5.18 Event_Type Entity . . . 45
3.6 System Architecture . . . 45
3.6.1 Overview . . . 45
3.6.2 3-Tier Architecture Implementation . . . 46
3.7 API Design . . . 47
3.7.1 Authentication/Authorization API . . . 48
3.7.2 User API . . . 50
3.7.3 Ticket API . . . 52
3.7.4 Dormitory API . . . 55
3.7.5 Attachment API . . . 57
3.7.6 Rating API . . . 57
3.7.7 Message API . . . 57
3.7.8 Message Web Socket Event . . . 58
3.8 UI/UX Design . . . 58
3.8.1 Responsive Design . . . 59
3.8.2 Live Customization . . . 60

4 System Implementation **62**
4.1 Frontend Implementation . . . 62
4.1.1 Source code structure . . . 62
4.1.2 UI Main Components . . . 63
4.1.3 Responsive Design Implementation . . . 66
4.2 Backend Implementation . . . 67
4.2.1 Source code structure . . . 67
4.2.2 Setting up ExpressJs Server . . . 68
4.2.3 Database Integration . . . 69
4.2.4 Security Measures . . . 70

5 User Manual **71**
5.1 Sign in . . . 71
5.2 Forgot password . . . 72
5.3 Student's functions . . . 74
5.3.1 View Profile . . . 75
5.3.2 View Tickets . . . 76
5.3.3 Create Tickets . . . 76
5.3.4 Rate Tickets . . . 77
5.3.5 Message . . . 79
5.3.6 Newsfeed . . . 80
5.3.7 Notification . . . 81
5.3.8 Announcement . . . 82
5.3.9 Settings . . . 83
5.3.10 Feedback . . . 84
5.4 Staff's functions . . . 85
5.4.1 Available Tickets . . . 85
5.4.2 Tickets Handling . . . 87
5.4.3 Tickets History . . . 89
5.4.4 Message . . . 90
5.4.5 Notification . . . 91
5.4.6 Announcement . . . 92
5.5 Admin's functions . . . 93

5.5.1 Tickets Management 93
5.5.2 Users Management 94
5.5.3 Dormitory Management 95
5.5.4 Logs Management 96
5.5.5 Feedback Management 97
5.5.6 Report 98

6 Conclusion and Future Work **100**

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CSE Computer Science and Engineering

JWT JSON Web Token

REST Representational State Transfer

SQL Structured Query Language

UI User Interface

URL Uniform Resource Locator

UX User Experience

VGU Vietnamese-German University

List of Figures

1 ReactJS Logo . . . 20
2 Material UI Logo . . . 21
3 Vite Logo . . . 21
4 NodeJS Logo . . . 21
5 Expressjs Logo . . . 21
6 SocketIO Logo . . . 21
7 JWT Logo . . . 22
8 Detailed explanation of JWT-based authentication mechanism. . . . 22
9 Redis Logo . . . 22
10 PostgreSQL RDBMS Logo . . . 23
11 Student Use Case Diagram . . . 31
12 Staff (Dormitory staff, Student affairs) Use Case Diagram . . . 32
13 Admin Use Case Diagram . . . 33
14 Ticket-Raising Process Workflow . . . 34
15 ER Diagram . . . 35
16 Three-tier Architecture[1] . . . 45
17 Profile UI in Desktop device . . . 59
18 Profile UI in Mobile device . . . 60
19 Web UI Live Customization . . . 61
20 Web UI Main Components . . . 63
21 Landing Page . . . 71
22 Sign in Form . . . 72
23 Failed Sign in attempt . . . 72
24 Reset Password Form . . . 73
25 Reset password successfully . . . 73
26 Reset password failed . . . 73
27 Reset password email instructions . . . 74
28 Student's Home Page . . . 75
29 Student's Profile Page . . . 75
30 Student's Tickets List Page . . . 76
31 Student's Create Tickets Page . . . 77

32 Student's Rate Tickets Page . . . 78
33 Submit a ticket rating . . . 78
34 Student's Message Page . . . 79
35 Student's Newsfeed . . . 80
36 Student's Notification Page . . . 81
37 Student's Announcement Page . . . 82
38 Student's Edit Profile Page . . . 83
39 Student's Change Password Page . . . 84
40 Student's Feedback Page . . . 85
41 Staff's Available Tickets Page . . . 86
42 Handle a ticket . . . 87
43 Successfully add a ticket to ticket handling list . . . 87
44 Staff's Tickets Handling List . . . 88
45 Mark a ticket as done . . . 88
46 Cancel a ticket . . . 89
47 Staff's Tickets History Page . . . 90
48 Staff's Message Page . . . 91
49 Create a Notification Modal . . . 92
50 Create an Announcement Modal . . . 93
51 Admin's Ticket Management Page . . . 94
52 Admin's User Management Page . . . 95
53 Admin's Dormitory Management Page . . . 96
54 Admin's Logs Management Page . . . 97
55 Admin's Feedback Management Page . . . 98
56 Admin's Report Page . . . 98

List of Tables

1 System key features . . . 15
2 System key components . . . 16
3 Existing University Ticketing Systems . . . 19
4 Functional Requirements . . . 27
5 Functional Requirements by User Roles . . . 29
6 Non-Functional Requirements . . . 31
7 User Entity . . . 36
8 Ticket Entity . . . 37
9 User_Ticket Relationship . . . 37
10 Ticket_Type Entity . . . 38
11 Ticket_Status Entity . . . 38
12 Audience_Type Entity . . . 39
13 Attachment Entity . . . 39
14 Rating Entity . . . 40
15 Feedback Entity . . . 41
16 Message Entity . . . 41
17 Dorm Entity . . . 42
18 Announcement Entity . . . 42
19 Notification Entity . . . 43
20 Role Entity . . . 43
21 Notification_Audience Entity . . . 44
22 Log Entity . . . 44
23 Event_Type Entity . . . 45
24 3-Tier Architecture Implementation . . . 46
25 Authentication/Authorization API . . . 48
26 User API . . . 51
27 Ticket API . . . 54
28 Dormitory API . . . 56
29 Attachment API . . . 57
30 Rating API . . . 57
31 Rating API . . . 58

32 Frontend source code structure 62
33 Backend source code structure 68

List of Code Snippets

1 Example of a React component . . . 20
2 Request body of Login API . . . 49
3 Response of Login API . . . 49
4 Response of Refresh Token API . . . 49
5 Response of Verify Refresh Token API . . . 49
6 Response of Reset Password API . . . 50
7 User Scheme . . . 51
8 Ticket Scheme . . . 55
9 Dormitory Scheme . . . 56
10 Frontend Header Component . . . 63
11 Frontend Sidebar Component . . . 65
12 Example of Lazy Load Frontend Components . . . 66
13 MUI Grid System . . . 67
14 MUI Media Queries . . . 67
15 ExpressJS Server Setup . . . 68
16 Server connects to PostgreSQL Database . . . 69
17 Server connects to Redis . . . 70

1 Introduction

1.1 Project Background

The Student Life Support Service is a web-based platform designed to enhance the efficiency and accessibility of student support services at the Vietnamese-German University (VGU). Universities typically handle a large volume of student inquiries and requests, ranging from dormitory issues to general student affairs, but the traditional systems in place often fall short of meeting modern student expectations. The current support mechanisms at many educational institutions are not streamlined, leading to delays in issue resolution, inefficient communication between students and staff, and lack of transparency in the handling of support tickets. Students frequently experience difficulty in tracking the progress of their requests, and support staff often lack the tools needed to manage tickets effectively.

This project aims to address these challenges by introducing an integrated system that automates the submission, handling, and resolution of student support tickets. In addition to providing students with a clear communication channel with the relevant university staff, the system also includes features such as real-time messaging, ticket status updates, and feedback mechanisms. The system will allow administrators to manage user roles, view comprehensive reports on ticket status, and optimize resource allocation.

Additionally, at VGU, students living in dormitories or dealing with other administrative issues often face challenges in receiving timely support. Current methods of submitting issues through email or in-person communication are prone to delays and mismanagement, leading to student dissatisfaction. This is exacerbated by the lack of real-time updates and the absence of a centralized platform where students can view the status of their requests. Similarly, staff members experience difficulty in managing the volume of requests, tracking the status of tickets, and effectively communicating with students.

The proposed Student Life Support Service will streamline these processes by creating a user-friendly, centralized system that not only tracks and manages support tickets but also fosters better communication between students and staff.

1.2 Problem Statement

The lack of a streamlined, accessible system for managing student support services at VGU has led to inefficiencies in communication and delayed resolution of student requests. Students often face prolonged waiting times, uncertainty about the status of their tickets, and difficulty in communicating with the responsible staff. On the other hand, staff members face challenges in managing multiple requests efficiently, tracking their progress, and prioritizing tasks.
The specific problem addressed by this project is the absence of an integrated platform that facilitates smooth communication, real-time ticket management, and timely issue resolution between students and university staff. The current system is fragmented, lacking automation, and fails to provide transparency in the support process.

1.3 Objectives of the Project

The primary objective of this project is to develop a web-based Student Life Support Service that enables students to submit, track, and manage their support requests efficiently. The system will provide several key features, including:

Key features	**Description**
Ticket Management	Allow students to submit support tickets related to dormitory issues or other university services. Students can track the progress of their tickets in real time.
Real-time Communication	Enable direct communication between students and staff handling the tickets using a real-time messaging system.
Role Management	Provide administrators with tools to manage user roles, such as students, dormitory staff, and student affairs personnel.
Feedback Mechanism	Allow students to give feedback on the support provided and rate the resolution of their tickets.
Notifications and Announcements	Provide students and staff with timely notifications and announcements related to their tickets or university activities.
Responsive Design	Ensure the system is fully compatible with devices of all sizes, including desktops, laptops, tablets, and smartphones.

Table 1: System key features

The focus of the system is to create an efficient, user-friendly, and responsive platform that can be accessed by students and staff across various devices, ensuring convenience and accessibility.

1.4 Scope of the Project

The Student Life Support Service project includes the development of a full-stack web application with several key components:

Key components	**Description**
Frontend	Built with ReactJS, Material UI, and Vite, the frontend will focus on providing a responsive, interactive interface that can be accessed from any device. Users will be able to submit support tickets, communicate with staff, and view ticket updates.
Backend	Using NodeJS, ExpressJS, and SocketIO, the backend will handle ticket processing, real-time communication, and manage user roles. JWT-based authentication will be used to secure the platform, with refresh tokens stored in Redis for session management.
Database	A PostgreSQL database will store user data, tickets, and related information. This will allow efficient querying and management of all system data.

Table 2: System key components

The system does not cover advanced analytics or AI-driven decision-making, as it is focused on the core functionality of ticket management and communication. Additionally, the scope does not include integration with third-party tools for external service management, though future expansions could allow for such features.

1.5 Thesis Structure

The thesis is organized into several sections, each addressing different aspects of the project:

- **Section 1: Introduction** – Provides an overview of the project background, objectives, problem statement, scope, and thesis structure.
- **Section 2: Literature Review** – Reviews existing solutions and technologies related to student support services, analyzing gaps in current systems that the Student Life Support Service aims to address.
- **Section 3: System Design** – Discusses the system's functional and non-functional requirements, architecture, database design, and API structure. It also covers the UI/UX design approach and how the responsive feature is implemented.
- **Section 4: System Implementation** – Details the step-by-step implementation of the frontend, backend, database, and security mechanisms. It includes code snippets, system flows, and real-time messaging features.
- **Section 5: Results and Discussion** – Analyzes the results of the project, discussing whether the initial objectives were met.
- **Section 6: Conclusion and Future Work** – Concludes the thesis by summarizing the project outcomes and discussing possible future enhancements, such as extending the system to other universities or integrating advanced analytics features.

2 Literature Review

2.1 Existing solutions

2.1.1 Group Chat-Based Systems (Current Solution at VGU)

Currently, many educational institutions, including VGU, rely on informal systems like social media group chats (e.g., Facebook or WhatsApp groups) for raising support tickets and contacting staff. While these systems are easy to set up and require minimal resources, they suffer from significant limitations:

- **Lack of Structure**: The conversation threads are disorganized, making it hard to track specific issues or prioritize them.
- **Absence of Accountability**: There's no formal ticketing system, leading to delays in responses and no mechanism to track whether an issue has been resolved.
- **Inadequate Historical Data**: It's difficult to retrieve past conversations or analyze data to improve service.
- **Lack of Privacy**: Group chats often expose personal information to all participants, which may raise privacy concerns.

2.1.2 Existing University and Open-source Ticketing Systems

Several universities have adopted formal ticket management systems for handling student support services. These systems are often integrated into larger university management platforms or custom-built web applications. Common examples include:

Systems	Features	Limitations
JIRA Service Management	Offers customizable workflows, automated prioritization, and detailed issue tracking.	Too complex for university needs, expensive, and difficult to adapt without major customization.

Systems	Features	Limitations
Freshdesk	Supports ticket management, multi-channel communication, and agent collaboration.	Feature-heavy and expensive for universities; lacks educational-specific tools.
Zendesk	Provides email, live chat, and ticketing, with automation and analytics.	Geared towards businesses; lacks flexibility for diverse student needs and real-time communication.
OSTicket	Open-source, customizable, with email-based ticketing and status tracking.	Requires customization for universities, not intuitive for non-technical users, lacks real-time communication.

Table 3: Existing University Ticketing Systems

2.1.3 Limitations of Existing Solutions in the University Context

- **Complexity**: Many existing solutions are designed for enterprise environments and are not tailored to the unique requirements of universities.
- **Lack of Customization**: Solutions like JIRA and Zendesk require extensive customization to meet university-specific needs, such as handling dormitory issues or academic support tickets.
- **Cost**: Proprietary solutions can be expensive, making them less viable for universities with limited IT budgets.
- **Lack of Real-Time Communication**: Most solutions offer asynchronous communication through email or message boards but do not provide real-time chat, which is essential for time-sensitive student support.

2.2 Technology Review

2.2.1 Frontend: ReactJS, Material UI, Vite

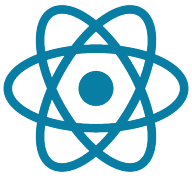

Figure 1: ReactJS Logo

ReactJS is a popular JavaScript library for building user interfaces, which provides a fast, scalable, and modular way to develop the frontend of web applications[2]. Its component-based architecture allows for reusability and efficient state management using hooks like `useState()` and `useEffect()`. This enables a responsive and dynamic user experience, ideal for handling real-time ticket updates.

```
const Profile = () => {

  return (
    <MainCard title="Personal Information">
      <Grid container spacing={gridSpacing}>

        <Grid item xs={12} sm={6}>
          <ProfileCard />
        </Grid>

        <Grid item xs={12} sm={6}>
          <SchoolDetailsCard/>
        </Grid>

      </Grid>
    </MainCard>
  );
}

export default Profile;

```

Code snippet 1: Example of a React component

With a vast array of libraries and tools available, React offers flexibility for adding extra functionality like routing, form handling, or animations. This helps build a rich, dynamic user experience.

Material UI is a React-based UI component library that implements Google's Material Design principles. Material UI ensures that the frontend is both visually appealing and functionally intuitive. Pre-built components like buttons, forms, and dialogs accelerate development while maintaining consistency in design.[4]

Figure 2: Material UI Logo

Figure 3: Vite Logo

Vite, a modern frontend build tool that offers faster development speed compared to older tools like Webpack. Vite optimizes the build process for React applications by providing instant hot module replacement (HMR), which is useful for a smooth developer experience during iterative development cycles.[3]

2.2.2 Backend: NodeJS, ExpressJS, SocketIO

Figure 4: NodeJS Logo

NodeJS is a runtime that enables JavaScript to be used for server-side scripting, making it possible to use a single language (JavaScript) throughout the stack. NodeJS is non-blocking and event-driven, making it ideal for handling I/O-heavy tasks like managing support ticket requests in real time.

ExpressJS is a minimalist web framework for NodeJS, Express simplifies routing, middleware management, and API handling. It serves as the backbone of the server, processing requests from the frontend, interacting with the database, and managing the business logic.

Figure 5: Expressjs Logo

Figure 6: SocketIO Logo

SocketIO is a JavaScript library that enables real-time, bidirectional communication between clients and servers. SocketIO is used to implement features such as real-time messaging between students and staff, making the system more interactive and responsive.[5]

2.2.3 Authentication: JWT, Redis

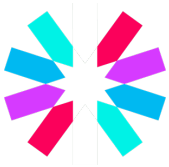

Figure 7: JWT Logo

JWT (JSON Web Tokens) is a token-based authentication system that provides secure stateless authentication for users. JWT is ideal for modern web applications because tokens can be stored on the client-side (in local storage or cookies) and are transmitted with each request, allowing for scalability.

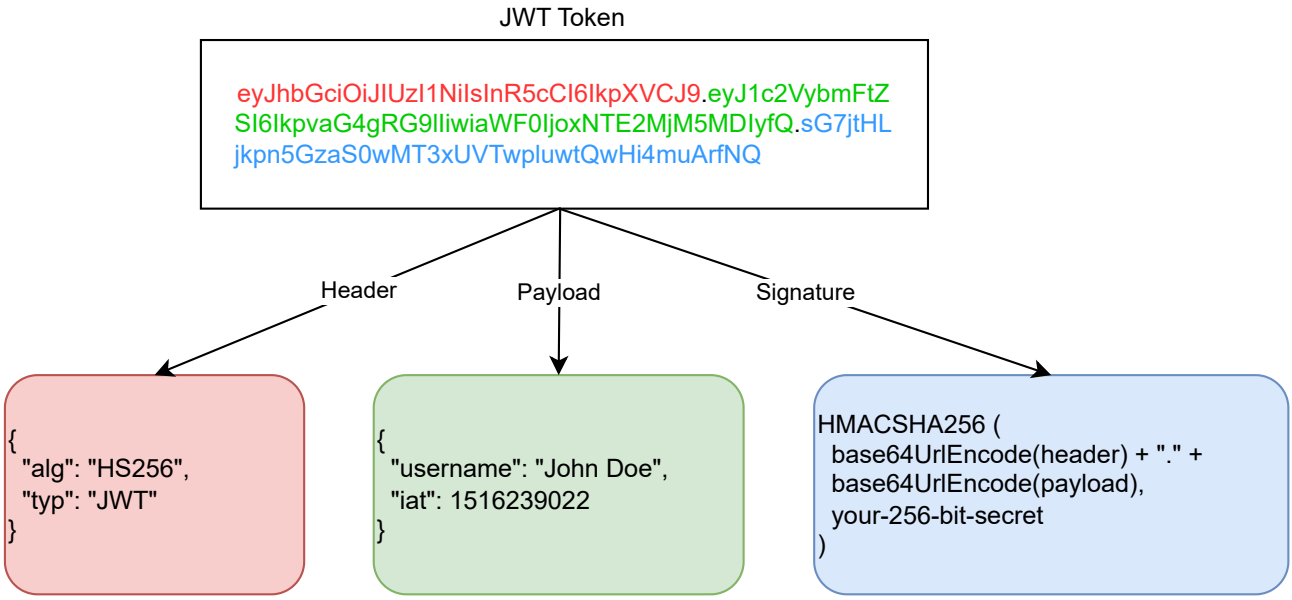

Figure 8: Detailed explanation of JWT-based authentication mechanism.

Figure 9: Redis Logo

Redis is an in-memory data structure store, Redis is used for session management, particularly in storing refresh tokens. By caching these tokens, Redis reduces the load on the database and enhances the system's performance.

2.2.4 Database: PostgreSQL

Figure 10: PostgreSQL RDBMS Logo

PostgreSQL is a powerful, open-source relational database that offers strong ACID compliance, making it suitable for managing critical data like user accounts, ticket information, and communication logs. Its support for advanced querying and indexing ensures the system can handle complex searches efficiently.

2.2.5 Responsive Web Design: Techniques and Tools

- **Media Queries**: CSS media queries are used to apply different styles based on device characteristics (screen size, resolution). This allows the frontend to automatically adapt to different devices, ensuring that the system is usable on desktops, laptops, tablets, and smartphones.
- **CSS Flexbox/Grid**: These CSS layout models allow for flexible, responsive layouts that adjust to different screen sizes. Flexbox is ideal for managing component positioning in small screens, while Grid is useful for creating complex layouts in larger screens.

2.3 Theoretical Background

2.3.1 Ticket Management Systems

A ticket management system is a tool designed to manage and track the progress of support requests, from the time they are submitted until they are resolved. The system typically assigns a unique identifier (ticket) to each request, enabling staff to monitor progress, prioritize issues, and provide timely responses. In a university context, ticket management systems are particularly useful for handling student issues, such as dormitory problems, academic inquiries,

and administrative requests. By assigning specific staff members to tickets, the system ensures accountability and reduces response time.

2.3.2 Real-Time Communication Tools

Real-time communication tools like SocketIO or WebSockets are essential in modern web applications. These tools allow for instantaneous data transmission between the server and client, enabling real-time messaging and live updates. For instance, in the Student Life Support Service, students and staff can exchange messages directly without having to refresh the page, ensuring efficient communication.

2.3.3 Web Application Development Best Practices

- **Modular Design**: Applications should be developed in a modular fashion, separating concerns into distinct components (frontend, backend, database). This allows for easier maintenance and scalability.
- **Security First**: With the increasing number of security breaches in web applications, implementing security best practices like JWT for authentication, HTTPS for communication, and proper data validation is essential.
- **Responsive Design**: Ensuring that the application works across different devices and screen sizes is a fundamental best practice, especially for a university setting where students and staff might use a wide variety of devices.

2.4 Gap Analysis

2.4.1 What is Missing from Existing Solutions

Existing solutions for university support systems face several shortcomings. Privacy concerns arise in social media-based group chats, where sensitive student information may be exposed, and even proprietary systems lack a strong focus on educational privacy needs. Role-specific functionalities are often missing, with few systems offering specialized tools for students, dormitory staff, or administrators, or including student-centric features like feedback collection, ticket rating, and public status views. Limited analytics is another issue; while general analytics are provided, they don't cater to the specific needs of student services, such as tracking recurring

issues or ticket performance. Additionally, many systems, like JIRA, are not user-friendly for students, requiring training and posing barriers in environments where simplicity is essential.

2.4.2 How the Student Life Support Service Fills These Gaps

The Student Life Support Service addresses the gaps in existing systems by offering a solution tailored specifically to university needs. Its customizable structure supports role-specific functionalities for students, dormitory staff, and administrators, making it ideal for managing university-specific scenarios like dormitory issues and academic inquiries. Real-time communication is enabled through SocketIO, allowing fast, interactive responses between students and staff. The user-friendly interface, built with ReactJS and Material UI, ensures easy navigation for non-technical users. As an open-source, cost-effective platform using NodeJS, PostgreSQL, and ReactJS, it avoids the high costs of proprietary software. The system also provides role-specific features, such as ticket creation, tracking, and rating for students, efficient ticket handling for staff, and detailed reporting tools for administrators. Enhanced privacy and security are ensured through JWT-based authentication and role-based access, preventing unauthorized access to sensitive information. Additionally, built-in data analytics offers administrators insights into ticket trends and areas for improvement in student support services.

3 System Design

3.1 Functional Requirements

The Student Life Support Service is designed to fulfill the specific functional requirements of three key user roles: Students, Dormitory Staff (or Student Affairs), and Administrators. Each role has its own set of features tailored to its needs within the system.
User Type: S-Student, DS-Dormitory Staff/Student Affairs, A-Admin (Operator)
Categorized: F-Functional, NF-Nonfunctional

No	Requirement	Description	Priority	User Type	Category
1	Manage personal info	Users can view and update their personal information.	Medium	S, DS, A	F
2	Support tickets	Users can create (raise), view support tickets.	High	S, DS, A	F
3	Contact through messages	Users can contact the staff or students handling the support ticket through text messages.	High	S, DS	F
4	Ticket rating	Students can rate their tickets which are marked as done.	Medium	S	F
5	View newsfeed	Users can view a newsfeed of public pending/in-process tickets.	Low	S, DS, A	F
6	View notifications	Users can view notifications and announcements.	Medium	S, DS, A	F
7	Feedback and suggestions	Users can give feedback and suggestions for the system.	Medium	S, DS, A	F
8	Handle support tickets	Dormitory staff can view and handle (mark as done, cancel) support tickets.	High	DS	F

No	Requirement	Description	Priority	User Type	Category
9	View past tickets	Dormitory staff can view all previously handled support tickets.	Medium	DS	F
10	Manage notifications	Dormitory staff and admins can create and manage notifications and announcements.	High	DS, A	F
11	Manage users	Admins can manage all users/roles (create, view, update, delete).	High	A	F
12	Manage tickets	Admins can manage all support tickets (view, delete).	High	A	F
13	Manage dormitories	Admins can manage all dormitories (create, view, delete).	Medium	A	F
14	Manage system logs	Admins can manage system logs (view, delete).	Medium	A	F
15	Manage feedback	Admins can manage system feedback (view, delete).	Low	A	F
16	View system report	Admins can generate and view system reports.	High	A	F

Table 4: Functional Requirements

For clearer comprehension, the table presented below provides a detailed visualization of the functional requirements, organized according to the different user roles within the system. This structure allows for a more precise understanding of how each role interacts with the system's features and capabilities.

User roles	**Functional Requirements**
Student	• can view, update his/her personal information. • can create (raise), view his/her support tickets. • can contact the staff who handles the support ticket through text messages. • can rate his/her tickets which are marked as done. • can view newsfeed (public pending/in process tickets). • can view notifications, announcement. • can give feedback and suggestions for the system.
Dormitory staff/ Student Affairs	• can view, update his/her personal information. • can view all available support tickets. • can handle support tickets. (mark as done, cancelled) • can view all past handled tickets. • can contact students who owns the ticket through text messages. • can view newsfeed (public pending/in process tickets). • can create, view notifications, announcement. • can give feedback and suggestions for the system.

User roles	Functional Requirements
Admin (Operator)	• can manage his/her personal information (view, update). • can manage all users/roles (create, view, update, delete). • can manage all support tickets (view, delete). • can manage all dormitories (create, view, delete). • can manage system logs (view, delete). • can manage system feedback (view, delete). • can view newsfeed (public pending/in process tickets). • can manage notifications, announcement (create, view). • can view the system report.

Table 5: Functional Requirements by User Roles

3.2 Non-Functional Requirements

Categorized: NF-Nonfunctional

No	Requirement	Description	Priority	Category	Functioning
1	Fast Response Time	The system should provide fast responses for user interactions such as submitting tickets, viewing statuses, and real-time messaging.	High	NF	Performance
2	Real-Time Communication	Messages between students and staff should be transmitted with minimal latency (under 100 milliseconds).	High	NF	Performance

No	Requirement	Description	Priority	Category	Functioning
3	Concurrent Users	The system must support up to 500 concurrent users without significant performance degradation.	High	NF	Performance
4	Database Query Optimization	PostgreSQL database should be optimized to handle high read/write volume efficiently even during peak load.	High	NF	Performance
5	JWT-Based Authentication	Secure authentication using JSON Web Tokens (JWT), with short-lived tokens and securely stored refresh tokens in Redis.	High	NF	Security
6	Role-Based Access Control	Enforce strict role-based access to ensure users only have access to the functionality appropriate for their role.	High	NF	Security
7	Encryption	All communications between the client and server must be encrypted using HTTPS to ensure data security.	High	NF	Security
8	Data Validation	Input from users must be validated and sanitized to protect against common vulnerabilities like SQL Injection and Cross-Site Scripting.	High	NF	Security
9	Audit Logs	Admins must have access to immutable and secure audit logs to track user actions such as login attempts and system modifications.	Medium	NF	Security
10	Database Scalability	The PostgreSQL database should scale efficiently as the number of tickets, messages, and users grows.	High	NF	Scalability
11	User-Friendly Interface	The interface should be intuitive and easy to navigate for users of varying technical abilities.	High	NF	Usability

No	Requirement	Description	Priority	Category	Functioning
12	Cross-Device Compatibility	The system should be responsive and function well on desktops, laptops, tablets, and smartphones.	High	NF	Usability

Table 6: Non-Functional Requirements

3.3 Use Case Diagrams

Figure 11: Student Use Case Diagram

Figure 12: Staff (Dormitory staff, Student affairs) Use Case Diagram

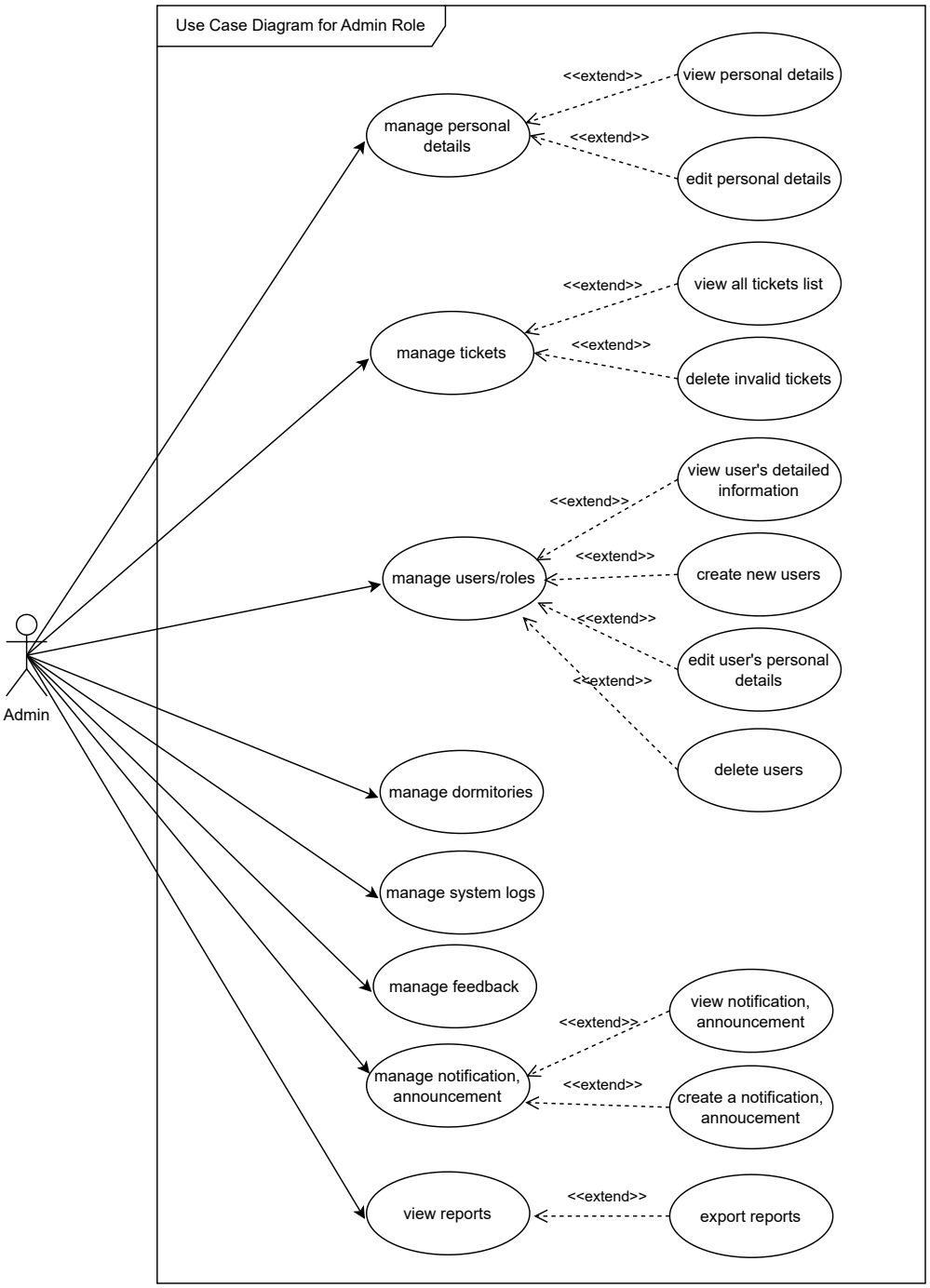

Figure 13: Admin Use Case Diagram

3.4 Process Workflow Diagrams

The core functionality of the Student Life Support Service is its ticket-raising process. The following diagram provides a detailed step-by-step illustration of this process.

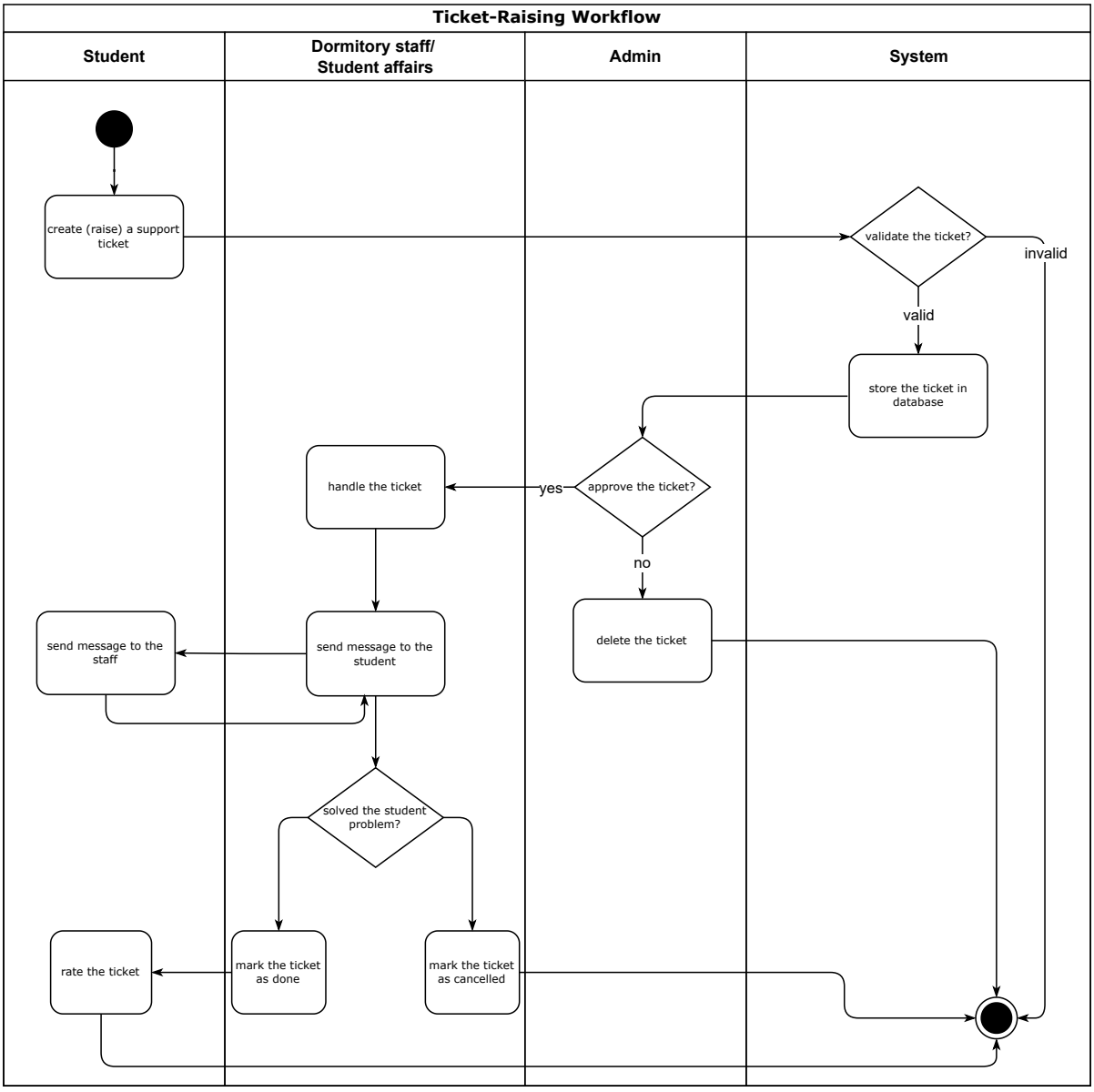

Figure 14: Ticket-Raising Process Workflow

3.5 Database Design

3.5.1 ER Diagram

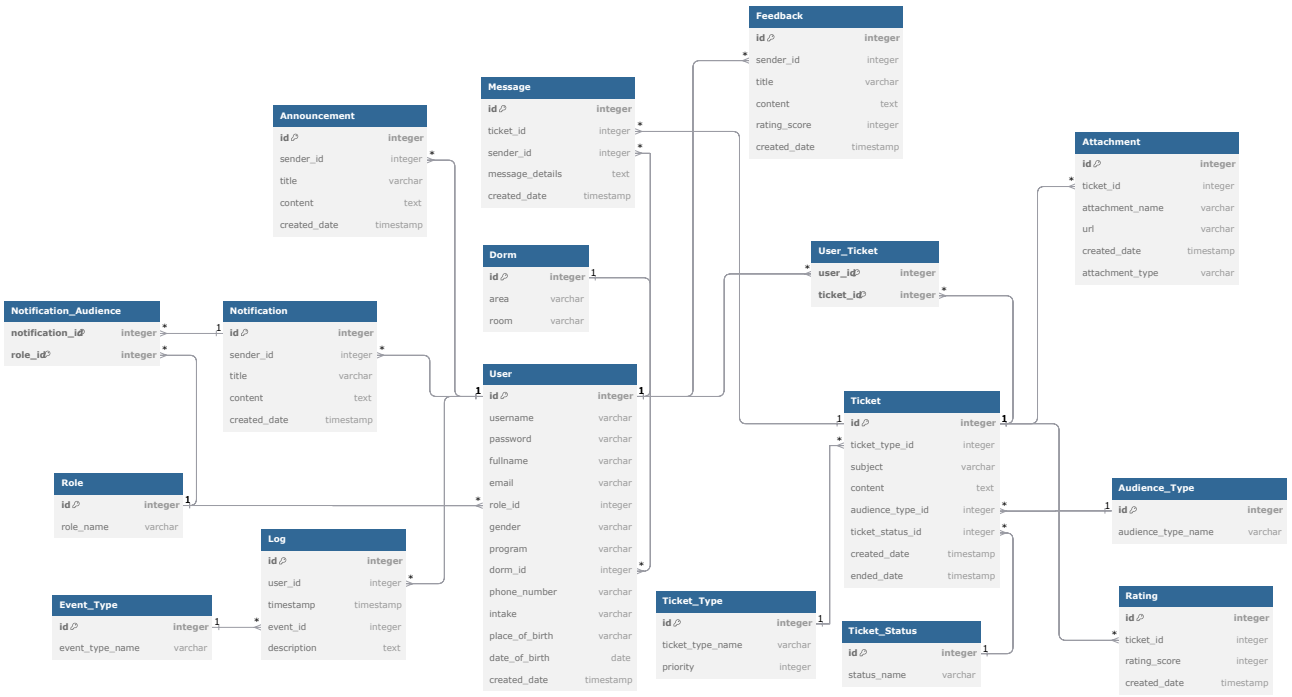

Figure 15: ER Diagram

3.5.2 User Entity

The User entity is fundamental to the system's user management, encompassing essential information that defines each user's profile and access rights. This entity includes several key attributes that contribute to its operational integrity (see Table 7)

Key Type	Field Name	Data Type	Constraints	Description
Primary	id	serial (int)	NOT NULL	id of a user
	username	varchar(255)	NOT NULL UNIQUE	the user name of a user, it could be matriculation number of a student
	email	varchar(255)	NOT NULL UNIQUE	the email of a user

Key Type	Field Name	Data Type	Constraints	Description
	fullname	varchar(255)	NOT NULL	the full name of a user
	gender	varchar(255)	NOT NULL	the gender of a user
Foreign	role_id	int	NOT NULL	the role id of a user
Foreign	dorm_id	int	NOT NULL	the dorm id where user lives (if the user does not live in a dormitory, dorm_id value equals to 1)
	program	varchar(255)		the program that user registered at university (E.g: Computer Science, Architecture, etc.)
	intake	varchar(255)		the time when a user registered a specific program at university (E.g: 2020, 2021, etc.)
	phone_number	varchar(255)	NOT NULL	the phone number of a user
	place_of_birth	varchar(255)	NOT NULL	the birth place of a user
	date_of_birth	date	NOT NULL	the birth date of a user
	password	varchar(255)	NOT NULL	the password of a user (in hashed string)
	created_date	timestamp with time zone	NOT NULL	the date time when a user account is created in the system

Table 7: User Entity

3.5.3 Ticket Entity

This table outlines the key attributes necessary for managing ticket entities within the system. It includes various fields such as the ticket ID, type, subject, content, and associated status. Additionally, it specifies data types, constraints, and a detailed description of each field to ensure proper handling of ticket information (refer to Table 8).

Key Type	Field Name	Data Type	Constraints	Description
Primary	id	serial (int)	NOT NULL	id of a ticket
Foreign	ticket_type_id	int	NOT NULL	the id of the ticket type
	subject	varchar(255)	NOT NULL	the subject (title) the ticket
	content	text	NOT NULL	the detailed description of the problem declared in the ticket
Foreign	audience_type_id	int	NOT NULL	the id of audience type assigned to the ticket
Foreign	ticket_status_id	int	NOT NULL	the id of current status assigned to the ticket
	created_date	timestamp with time zone	NOT NULL	the date time when a ticket is created in the system
	ended_date	timestamp with time zone	NOT NULL	the date time when a ticket is marked as done or cancelled in the system

Table 8: Ticket Entity

3.5.4 User_Ticket Relationship

Key Type	Field Name	Data Type	Constraints	Description
Primary	user_id	int	NOT NULL	id of a ticket
Primary	ticket_id	int	NOT NULL	id of a user

Table 9: User_Ticket Relationship

This table describes the relationship between users and tickets within the system. It contains two primary key fields: `user_id` and `ticket_id`, each identified by a unique integer. The `user_id` represents the unique identifier for a user, while the `ticket_id` corresponds to a unique ticket within the system. Both fields are non-nullable, ensuring that each user and

ticket association is properly recorded and maintained (refer to Table 9). This relationship is essential for tracking which users have raised and handled specific tickets in the system .

3.5.5 Ticket_Type Entity

The Ticket_Type entity serves to classify different categories of tickets within the system, each defined by a unique identifier and a descriptive name. It also includes a priority level that indicates the urgency of each ticket type, where a higher value corresponds to a lower priority. This structure enables efficient ticket management and prioritization (see Table 10).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a ticket type
	ticket_type_name	varchar(255)	NOT NULL UNIQUE	the type name of a ticket
	priority	int	NOT NULL	the priority of the ticket type (higher indicates lower priority)

Table 10: Ticket_Type Entity

3.5.6 Ticket_Status Entity

The Ticket_Status entity is designed to define various states that a ticket can be in within the system. Each status is uniquely identified by an ID and has a descriptive name, allowing for clear tracking and management of tickets throughout their lifecycle. This structured approach enhances the ability to monitor ticket progress and provides clarity to users regarding the current status of their tickets (see Table 11).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a ticket status
	status_name	varchar(255)	NOT NULL UNIQUE	the status name of a ticket

Table 11: Ticket_Status Entity

3.5.7 Audience_Type Entity

The Audience_Type entity categorizes target audiences for tickets in the system. Each type is uniquely identified by an ID and has a descriptive name, enhancing communication and ticket management for specific user groups (see Table 12).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a ticket audience type
	audience_type_name	varchar(255)	NOT NULL UNIQUE	the target audience type name of a ticket

Table 12: Audience_Type Entity

3.5.8 Attachment Entity

The Attachment entity manages files associated with tickets in the system. Each attachment is uniquely identified by an ID and linked to a specific ticket through a foreign key. The entity includes attributes for the original file name, its server URL, and the timestamp of its upload, ensuring organized storage and retrieval of related documents for user reference (see Table 13).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of an attachment
Foreign	ticket_id	int	NOT NULL	the reference ticket id which has an attachment
	attachment_name	varchar(255)	NOT NULL	the original name of the attachment
	url	varchar(255)	NOT NULL UNIQUE	the address of an attachment on the server
	created_date	timestamp with time zone	NOT NULL	the date time when an attachment is firstly uploaded to the system

Table 13: Attachment Entity

3.5.9 Rating Entity

The Rating entity captures user ratings for tickets in the system. Each rating is uniquely identified by an ID and linked to a specific ticket through a foreign key. It includes a score reflecting the user's assessment and a timestamp indicating when the rating was given, enabling effective tracking of user feedback (see Table 14).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a rating
Foreign	ticket_id	int	NOT NULL	the reference ticket id which has the rating
	rating_score	int	NOT NULL	the rating score of a ticket
	created_date	timestamp with time zone	NOT NULL	the date time when user rates a ticket.

Table 14: Rating Entity

3.5.10 Feedback Entity

The Feedback entity collects user feedback on various aspects of the system. Each feedback entry is uniquely identified by an ID and linked to the user providing it through a foreign key. It includes a title for the feedback, detailed content, a rating score reflecting the user's evaluation, and a timestamp indicating when the feedback was submitted. This structure facilitates the collection and analysis of user opinions to enhance system performance (see Table 15).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a feedback
Foreign	sender_id	int	NOT NULL	the id of the user who gives the feedback
	title	varchar(255)	NOT NULL	the title (subject) of the feedback
	content	text	NOT NULL	the details of the feedback

Key Type	Field Name	Data Type	Constraints	Description
	rating_score	int	NOT NULL	the rating score of the feedback
	created_date	timestamp with time zone	NOT NULL	the date time when user gives the feedback.

Table 15: Feedback Entity

3.5.11 Message Entity

The Message entity stores individual messages associated with tickets within the system. Each message is identified by a unique ID and linked to a specific ticket through a foreign key, effectively acting as a conversation identifier. It records the sender's ID, the content of the message, and a timestamp indicating when the message was sent. This structure supports organized communication related to tickets, enabling users to track conversations effectively (see Table 16).

Key Type	Field Name	Data Type	Constraints	Description
Primary	id	serial (int)	NOT NULL	id of a message
Foreign	ticket_id	int	NOT NULL	the id of a ticket which has the message (acts as a conversation id)
	sender_id	int	NOT NULL	id of a user who has the message
	message_details	text	NOT NULL	the details of a message
	created_date	timestamp with time zone	NOT NULL	the date time when user send a message.

Table 16: Message Entity

3.5.12 Dorm Entity

The Dorm entity represents the various dormitories within the system. Each dormitory is uniquely identified by an ID and includes details about its area and specific room designation.

This structure facilitates the organization and management of dormitory accommodations, ensuring that each location is easily identifiable (see Table 17).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a dorm
	area	varchar(255)	NOT NULL UNIQUE	the dorm area name
	room	varchar(255)	NOT NULL UNIQUE	the dorm room of an area

Table 17: Dorm Entity

3.5.13 Announcement Entity

The Announcement entity captures information about announcements made within the system. Each announcement is uniquely identified by an ID and includes details such as the sender's ID, title, content, and the timestamp of when it was created. This structure enables effective communication and dissemination of important information among users (see Table 18).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of an announcement
Foreign	sender_id	int	NOT NULL	the id of user who sends the announcement
	title	varchar(255)	NOT NULL	the title of an announcement
	content	text	NOT NULL	the details of an announcement
	created_date	timestamp with time zone	NOT NULL	the date time when user sends an announcement.

Table 18: Announcement Entity

3.5.14 Notification Entity

The Notification entity records details about notifications sent within the system. Each notification is uniquely identified by an ID and includes information such as the sender's ID, title, content, and the timestamp of when it was created. This structure supports timely communication and updates for users (see Table 19).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a notification
Foreign	sender_id	int	NOT NULL	the id of user who sends the notification
	title	varchar(255)	NOT NULL	the title of a notification
	content	text	NOT NULL	the details of a notification
	created_date	timestamp with time zone	NOT NULL	the date time when user sends a notification.

Table 19: Notification Entity

3.5.15 Role Entity

The Role entity is crucial for defining user permissions and access levels within the system. This entity facilitates the management of user roles, ensuring that access rights are clearly delineated and easily maintained (see Table 20).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a role
	role_name	varchar(255)	NOT NULL	the name of a role

Table 20: Role Entity

3.5.16 Notification_Audience Relationship

The Notification_Audience relationship establishes a many-to-many association between notifications and user roles, defining which roles are eligible to receive specific notifications. This structure enhances targeted communication within the system, allowing for tailored information delivery based on user roles (see Table 21).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	notification_id	int	NOT NULL	id of a notification
Primary	role_id	int	NOT NULL	the id of a role

Table 21: Notification_Audience Entity

3.5.17 Log Entity

The Log entity serves as a critical component for tracking user actions and system events within the application. It records essential information, including the user involved, the specific event associated with the action, a detailed description, and the timestamp of when the log entry was created. This structured approach enables effective auditing and monitoring of system activities (see Table 22).

Key Type	**Field Name**	**Data Type**	**Constraints**	**Description**
Primary	id	serial (int)	NOT NULL	id of a log
Foreign	user_id	int	NOT NULL	the reference user who has actions on log
Foreign	event_id	int	NOT NULL	the id of an event
	description	text	NOT NULL	the details of a log
	timestamp	timestamp with time zone	NOT NULL	the date time when a log is written to the database.

Table 22: Log Entity

3.5.18 Event_Type Entity

The Event_Type entity is designed to categorize various system events, providing a structured framework for event classification. It includes unique identifiers for each event type, along with descriptive names that facilitate easy reference and management within the system. This organization enhances the overall functionality and tracking of system activities (see Table 23).

Key Type	Field Name	Data Type	Constraints	Description
Primary	id	serial (int)	NOT NULL	id of a system event type
	event_type_name	varchar(255)	NOT NULL UNIQUE	the name of a system event type

Table 23: Event_Type Entity

3.6 System Architecture

3.6.1 Overview

The system follows a three-tier architecture, consisting of the following layers:

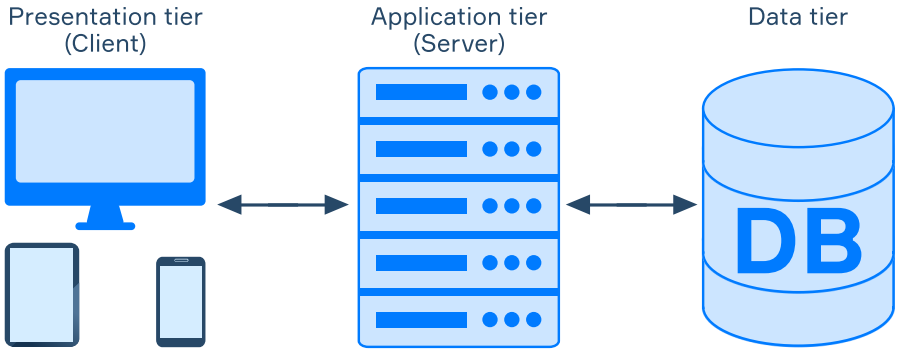

Figure 16: Three-tier Architecture[1]

1. **Presentation Layer (Client):**

   Handles all interactions with the user. Implements the user interface using ReactJS and Material UI. Communicates with the server through RESTful API calls and SocketIO for real-time features. Responsible for rendering components, collecting user input, and displaying data received from the backend.

2. **Business Logic Layer (Server)**:

   NodeJS and ExpressJS handle the core business logic, such as processing support ticket requests, authenticating users, managing roles, and communicating with the database. SocketIO is used to manage real-time messaging between students and staff. Implements security features like JWT-based authentication and session management using Redis.

3. **Data Layer (Database)**:

   PostgreSQL stores all persistent data, including user profiles, support tickets, messages, and system logs. The server communicates with the database using SQL queries to retrieve, create, update, and delete records. Ensures data consistency and integrity by enforcing constraints, foreign keys, and relationships.

3.6.2 3-Tier Architecture Implementation

Presentation Layer	The frontend is built using ReactJS, Material UI, and Vite. These technologies allow for dynamic rendering, responsive design, and a user-friendly interface. The presentation layer is responsible for capturing user input, displaying data, and providing real-time updates through WebSockets (using Socket.IO).
Business Logic Layer	The backend is implemented using NodeJS and ExpressJS. This layer handles all business logic, processes user requests, applies business rules, and interacts with the data layer. The business logic layer leverages Socket.IO for real-time communication, allowing instant messaging between students and staff.
Data Access Layer	The data access layer utilizes PostgreSQL to store user data, support ticket information, and system logs. The data layer interacts with the business logic layer to retrieve and store information as needed. Redis is employed for session management by storing the user JWT refresh token.

Table 24: 3-Tier Architecture Implementation

3.7 API Design

The API (Application Programming Interface) design is crucial for enabling communication between the frontend and backend of the Student Life Support Service application. A well-structured API facilitates seamless data exchange and supports the application's functionalities.

API Design Principles:

- **RESTful Architecture**: The API follows RESTful principles, utilizing standard HTTP methods (GET, POST, PATCH, DELETE) for interaction. This allows for clear and intuitive endpoints.
- **Versioning**: The API is versioned (/api/v1/) to manage changes and ensure backward compatibility for existing clients.
- **Error Handling**: Consistent error responses are defined, returning meaningful HTTP status codes (200 for success, 404 for not found, 500 for server errors) along with descriptive messages.

Data format:

- **Request Format**: All requests to the API are in JSON format, with the appropriate headers set (Content-Type: application/json).
- **Response Format**: API responses are standardized to return JSON objects. Successful responses include a status field, data field (for returned data), and an optional message field for additional context.

Security Measures:

- JWT is used for user authentication, with tokens sent in the Authorization header of each request (Authorization: Bearer <token>).

3.7.1 Authentication/Authorization API

Method	Endpoint	Header	Request Body	Response / Description
POST	/auth/login	Content-Type: application/json	JSON object with username and password	JSON object which contains access token, refresh token and general user information
POST	/auth/logout	Authorization: Bearer <token> Cookie: <refreshToken>	None	Clear tokens and cookie
POST	/auth/refresh-token	Cookie: <refreshToken>	None	JSON object which contains new access token
POST	/auth/verify-refreshToken	Cookie: <refreshToken>	None	JSON object which contains validation status and message
POST	/auth/reset-password	Content-Type: application/json	JSON object with email	JSON object which contains message
PATCH	/auth/reset-password	Content-Type: application/json	JSON object with reset password token	JSON object which contains message

Table 25: Authentication/Authorization API

The Login API enables the client to authenticate with the server and obtain permissions based on the client's role. Upon a successful request, the server returns a token string, which the client can use to access the server's protected routes.

`POST /auth/login`

```
{
  "username": "string",
  "password": "string"
}
```

Code snippet 2: Request body of Login API

```
{
  "user_id": "string",          // Unique identifier for the user
  "username": "string",         // User's login name
  "email": "string",            // User's email address
  "fullname": "string",         // User's full name
  "role_name": "string",        // User's role in the system (e.g., Admin)
  "accessToken": "string"       // JWT access token for authorization
}
```

Code snippet 3: Response of Login API

The Refresh Token API issues a new access token for the user, using the refresh token stored in the secure cookie for authorization.

`POST /auth/refresh-token`

```
{
  "accessToken": "string"       // JWT access token for authentication
}
```

Code snippet 4: Response of Refresh Token API

The Verify Refresh Token API is used to verify the validity of a refresh token. Upon receiving a request, the server checks if the provided refresh token is valid. If the token is valid, the response will return a JSON object indicating the token's validity status.

`POST /auth/verify-refreshToken`

```
{
  "valid": boolean
}
```

Code snippet 5: Response of Verify Refresh Token API

The Reset Password API enables users to reset their passwords by sending a password reset link to their email.

`POST /auth/reset-password`

```
{
  "message": "string"
}
```

Code snippet 6: Response of Reset Password API

3.7.2 User API

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/users	Authorization: Bearer <token>	None	JSON object which contains current user details
GET	/api/v1/users/all	Authorization: Bearer <token>	None	List of JSON objects which contain all user details
POST	/api/v1/users	Authorization: Bearer <token>	JSON object with needed information to create a new user	create a new user with supplied information
PATCH	/api/v1/users/password	Authorization: Bearer <token>	JSON object with old password and new password	Update new password of the current user

Method	Endpoint	Header	Request Body	Response / Description
PATCH	/api/v1/users/phone-number	Authorization: Bearer <token>	JSON object with new phone number	Update new phone number of the current user
PATCH	/api/v1/users/dorm/{user_id}	Authorization: Bearer <token>	JSON object with new dorm details	Update new dorm details for the given user id
PATCH	/api/v1/users/role/{user_id}	Authorization: Bearer <token>	JSON object with new role detail	Update new role for the given user id
PATCH	/api/v1/users/{user_id}	Authorization: Bearer <token>	JSON object with new user details	Update personal details for the given user id
DELETE	/api/v1/users/{user_id}	Authorization: Bearer <token>	None	Delete a user

Table 26: User API

The User API provides various endpoints to manage user-related functionalities within the Student Life Support Service application. It allows users to retrieve their own details, view all users, and perform actions like creating new users or updating existing user information. Each request requires an authentication token in the header to ensure secure access.

Users can retrieve their personal information or a list of all users through GET requests. For user creation, a POST request is utilized, where necessary details are supplied in the request body. The API also facilitates user updates through PATCH requests, allowing users to change their passwords, phone numbers, dormitory assignments, roles, or any personal details. Finally, a DELETE request enables the removal of a user from the system.

This API structure ensures comprehensive user management while maintaining security through token-based authentication.

```
{
  "username": "string",
  "fullname": "string",
  "email": "string",
  "role_name": "string",
  "gender": "string",
  "created_date": "string",
  "program": "string",
  "area": "string",
  "room": "string",
  "phone_number": "string",
  "intake": "string",
  "place_of_birth": "string",
  "date_of_birth": "string"
}
```

Code snippet 7: User Scheme

3.7.3 Ticket API

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/tickets	Authorization: Bearer <token>	None	List of JSON objects which contain current user tickets
GET	/api/v1/tickets/{ticket_id}	Authorization: Bearer <token>	None	JSON object which contains a ticket detail of current user
GET	/api/v1/tickets/all	Authorization: Bearer <token>	None	List of JSON objects which contain all tickets

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/tickets/all/{ticket_id}	Authorization: Bearer <token>	None	JSON object which contains a ticket detail
GET	/api/v1/tickets/types	Authorization: Bearer <token>	None	List of JSON objects which contain all ticket types
GET	/api/v1/tickets/types	Authorization: Bearer <token>	None	List of JSON objects which contain all ticket types
GET	/api/v1/tickets/public	Authorization: Bearer <token>	None	List of JSON objects which contain all public tickets
GET	/api/v1/tickets/pending	Authorization: Bearer <token>	None	List of JSON objects which contain all pending tickets
GET	/api/v1/tickets/in-progress	Authorization: Bearer <token>	None	List of JSON objects which contain all closed tickets
GET	/api/v1/tickets/audience-type	Authorization: Bearer <token>	None	List of JSON objects which contain all ticket audience types

Method	**Endpoint**	**Header**	**Request Body**	**Response / Description**
POST	/api/v1/tickets/	Authorization: Bearer <token> Content-Type: 'multipart/form-data'	JSON object with needed information to create a ticket	Create a new ticket with supplied information
PATCH	/api/v1/tickets/status	Authorization: Bearer <token>	JSON object with ticket id and status id	Update a ticket status with given information
DELETE	/api/v1/tickets/{ticket_id}	Authorization: Bearer <token>	None	Delete a specific ticket

Table 27: Ticket API

The Ticket API provides a set of endpoints to manage support tickets within the Student Life Support Service application. It enables users to create, view, update, and delete tickets, ensuring comprehensive ticket management while enforcing secure access through token-based authentication.

Users can retrieve a list of their own tickets or all tickets using GET requests, and detailed information about specific tickets is accessible through dedicated endpoints. The API also provides functionality to list all ticket types and audience types, as well as to filter tickets based on their current status, such as pending or in-progress.

For ticket creation, a POST request is utilized, where necessary details are submitted in the request body. Updates to ticket status can be performed with PATCH requests. Finally, users can delete tickets using DELETE requests, which ensures that the system can manage and maintain a clean ticket database effectively.

this API structure enhances user interaction with the ticketing system, providing the necessary tools for both users and administrators to manage support tickets efficiently.

```
{
  "ticket_id": number,
  "username": "string",
  "fullname": "string",
  "created_date": "string",
  "ended_date": "string",
  "ticket_type_name": "string",
  "subject": "string",
  "details": "string",
  "audience_type": "string",
  "status": "string",
  "dorm_area": "string",
  "dorm_room": "string",
  "attachments": [
    {
      "id": number,
      "type": "string",
      "name": "string",
      "url": "string"
    },
  ]
}
```

Code snippet 8: Ticket Scheme

3.7.4 Dormitory API

The Dormitory API provides functionality for managing dormitories, areas, and rooms. It allows users to retrieve lists of all dormitories, specific dormitory areas, or rooms within an area. Additionally, it supports creating new dormitories by specifying an area and room and allows for the deletion of dormitory rooms based on the area and room identifiers. These operations require user authentication via a Bearer token for access control.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/dorms	Authorization: Bearer <token>	None	List of JSON objects which contain all dormitory details
GET	/api/v1/dorms/area	Authorization: Bearer <token>	None	List of JSON objects which contain all dormitory areas
GET	/api/v1/dorms/rooms/{area}	Authorization: Bearer <token>	None	List of JSON objects which contain all dormitory rooms in a specific area
POST	/api/v1/dorms/	Authorization: Bearer <token>	JSON object which contains dormitory area and room	Create a new dormitory with supplied data
DELETE	/api/v1/dorms/{area}/{room}	Authorization: Bearer <token>	None	Delete a dormitory room

Table 28: Dormitory API

```
[
  {
    "dorm_area": "string",
    "rooms": [ { "dorm_room": "string"}, ... ]
  }, ...
]
```

Code snippet 9: Dormitory Scheme

3.7.5 Attachment API

The Attachment API allows users to retrieve an image or video file by specifying the attachment name in the endpoint. It does not require authentication or a request body, and the response includes the requested media file (either image or video).

Method	**Endpoint**	**Header**	**Request Body**	**Response / Description**
GET	/api/v1/attachment/ {attachment_name}	None	None	image or video file

Table 29: Attachment API

3.7.6 Rating API

Method	**Endpoint**	**Header**	**Request Body**	**Response / Description**
GET	/api/v1/rating/{ticket_id}	Authorization: Bearer <token>	None	JOSN object which contains rating score of a ticket
POST	/api/v1/rating	Authorization: Bearer <token>	JSON object which contains ticket id and the rating score	Create a rating score for the ticket with given id

Table 30: Rating API

3.7.7 Message API

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/message/conversation	Authorization: Bearer <token>	None	JOSN object which contains all conversation id of the user
GET	/api/v1/message/{ticket_id}	Authorization: Bearer <token>	None	List of JOSN objects which contain all messages of the given ticket id

Table 31: Rating API

3.7.8 Message Web Socket Event

- Method: WebSocket Event
- Headers: Authorization: Bearer <token>
- Request Body:

3.8 UI/UX Design

In accordance with the functional requirements, the system must incorporate a user interface (UI) that is not only intuitive and user-friendly but also fully functional and straightforward. The design should prioritize simplicity, ensuring that users can navigate and utilize the features without encountering unnecessary complexity. Additionally, the UI must be responsive, seamlessly adapting to various devices, including desktops, tablets, and smartphones. This adaptability will enhance the overall user experience, allowing individuals to access and engage with the system effortlessly, regardless of the device they are using. Ultimately, the goal is to create a cohesive and effective interface that meets users' needs while providing a smooth and enjoyable interaction with the system.

3.8.1 Responsive Design

The web user interface (UI) is designed to be fully responsive, ensuring an optimal user experience (UX) across a diverse range of devices, including mobile phones, tablets, and desktop computers. This adaptability allows the interface to seamlessly adjust its layout and functionality according to the screen size and resolution of the device being used. By prioritizing responsiveness, the design enhances accessibility and usability, providing users with a consistent and intuitive experience regardless of whether they are accessing the application on a small mobile screen, a medium-sized tablet, or a larger desktop monitor. This commitment to a responsive design ultimately fosters greater user engagement and satisfaction. (see Figures 18, 17)

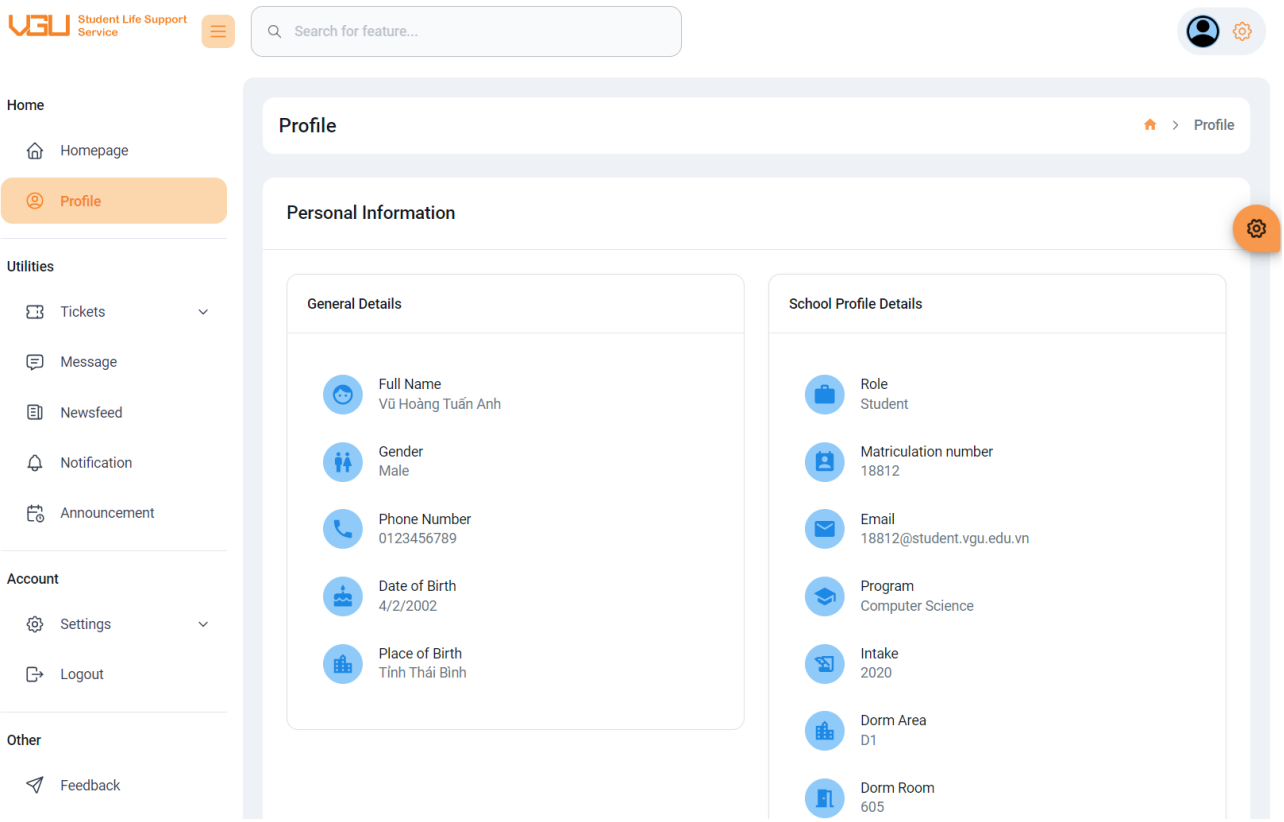

Figure 17: Profile UI in Desktop device

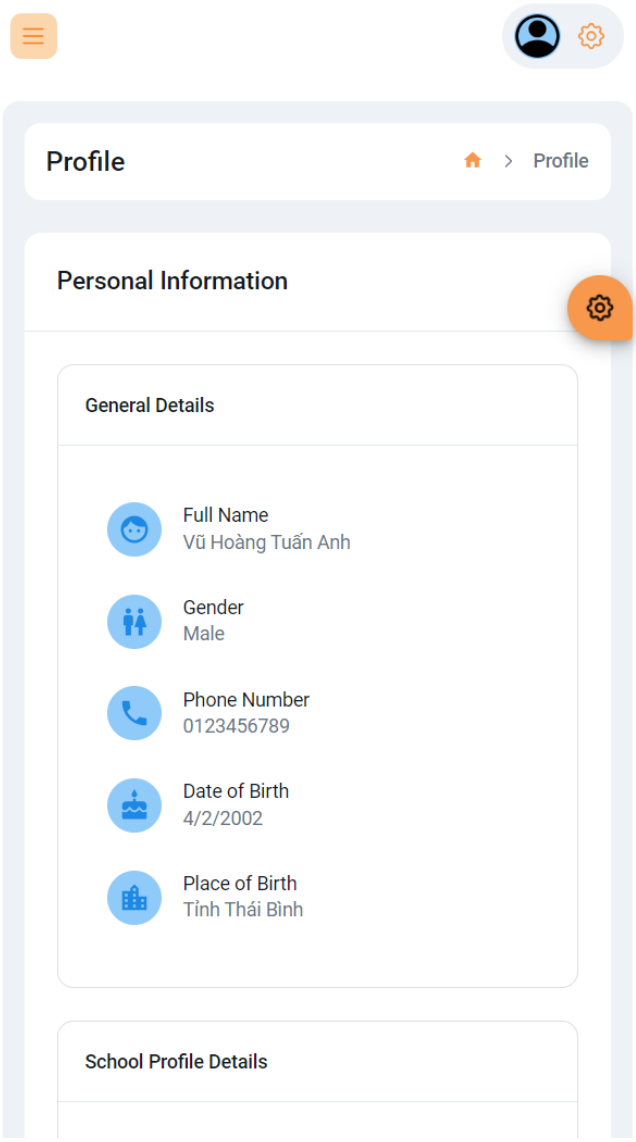

Figure 18: Profile UI in Mobile device

3.8.2 Live Customization

Users have the ability to personalize the web application in real-time by selecting the "Live Customize" option. Within this feature, they can choose from a variety of font families and adjust the border radius to suit their preferences. This functionality not only allows users to create a more tailored and visually appealing experience but also significantly enhances overall

user experience (UX). By empowering users to modify these visual elements according to their individual tastes, the application becomes more engaging and user-friendly, ultimately leading to greater satisfaction and a more enjoyable interaction with the platform.

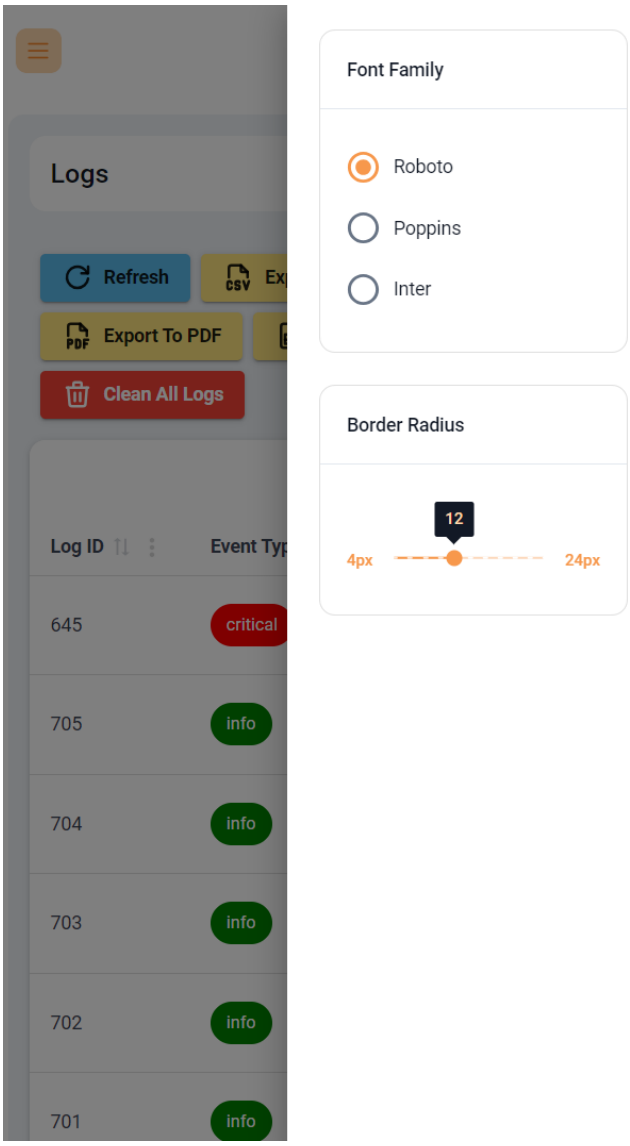

Figure 19: Web UI Live Customization

4 System Implementation

4.1 Frontend Implementation

4.1.1 Source code structure

Module/component name	**Description**
api/	defines custom instance of api communication instance with the backend server.
assets/	contains all images and cascading style sheets.
context/	contains the constants which have to be shared across these components.
hooks/	contains custom React hooks
public/	contains public assets (mostly for landing page).
routes/	defines all routes of the frontend
store/	holds specific states of the frontend
themes/	defines the color scheme, palette for the frontend
utils/	contains shared utilities across components
views/	contains main UI components for each roles
index.jsx	root component of the frontend
config.js	contains initial configuration of the front end
package.json	contains custom scripts and frontend package information
vite.config.mjs	contains configuration for Vite

Table 32: Frontend source code structure

4.1.2 UI Main Components

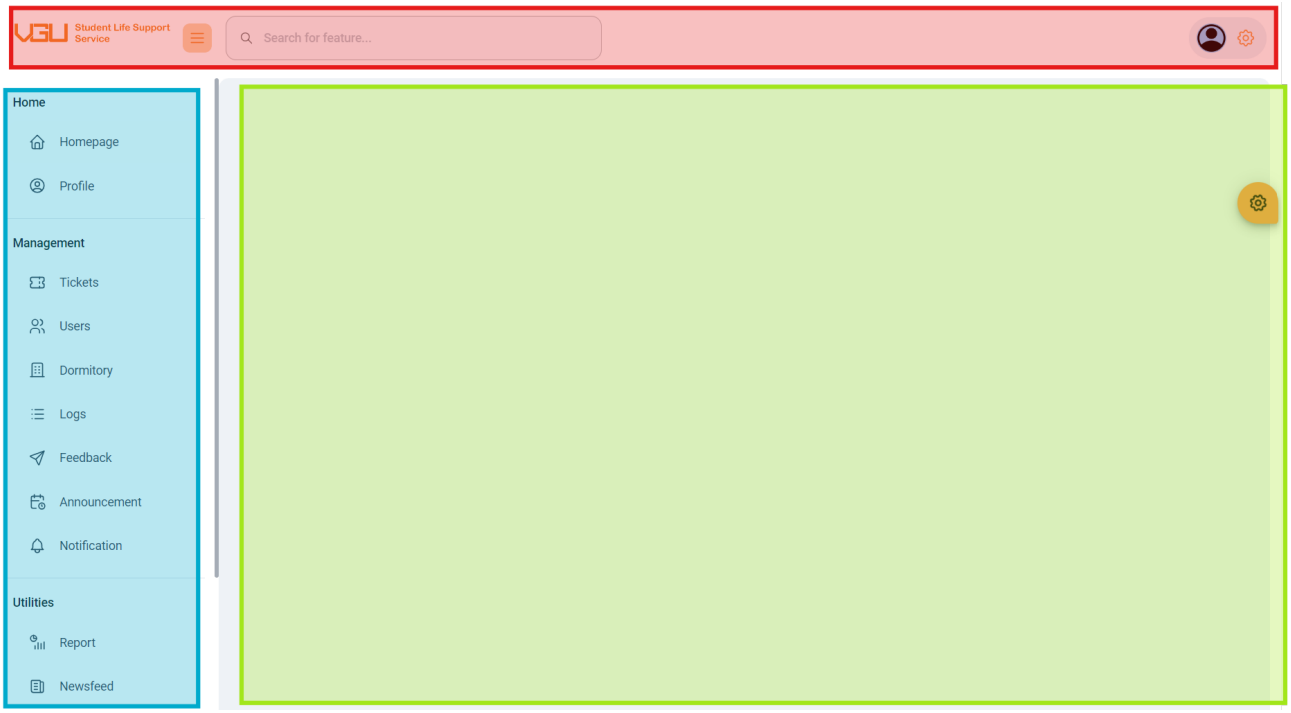

Figure 20: Web UI Main Components

There are 3 main components in the Web UI (from Figure 20):

1. Header: contains Logo image, Menu toggle button, Search bar, and Profile section across all pages

```
const Header = ({ handleLeftDrawerToggle }) => {
  const theme = useTheme();

  return (
  <>
  {/* logo & toggler button */}
  <Box
  sx={{
      width: 228,
      display: 'flex',
      [theme.breakpoints.down('md')]: {
        width: 'auto'
```

```
      }
  }}
  >
  <Box component="span" sx={{ display: { xs: 'none', md: 'block' },
   flexGrow: 1 }}>
  <LogoSection />
  </Box>
  <ButtonBase sx={{ borderRadius: '8px', overflow: 'hidden' }}>
  <Avatar
  variant="rounded"
  sx={{
      ...theme.typography.commonAvatar,
      ...theme.typography.mediumAvatar,
      transition: 'all .2s ease-in-out',
      background: theme.palette.secondary.light,
      color: theme.palette.secondary.dark,
      '&:hover': {
        background: theme.palette.secondary.dark,
        color: theme.palette.secondary.light
      }
  }}
  onClick={handleLeftDrawerToggle}
  color="inherit"
  >
  <IconMenu2 stroke={1.5} size="1.3rem" />
  </Avatar>
  </ButtonBase>
  </Box>

  {/* header search */}
  <SearchSection />
  <Box sx={{ flexGrow: 1 }} />
  <Box sx={{ flexGrow: 1 }} />


  {/* <ProfileSection /> */}
  <ProfileSection />
  </>
  );
```

```
};

Header.propTypes = {
  handleLeftDrawerToggle: PropTypes.func
};

export default Header;
```

Code snippet 10: Frontend Header Component

2. Sidebar: A dynamic navigation panel which renders all menu items of a specific roles.

```
const Sidebar = ({ drawerOpen, drawerToggle, window }) => {
  const theme = useTheme();
  const matchUpMd = useMediaQuery(theme.breakpoints.up('md'));

  const drawer = (
  <>
  <Box sx={{ display: { xs: 'block', md: 'none' } }}>
  <Box sx={{ display: 'flex', p: 2, mx: 'auto' }}>
  <LogoSection />
  </Box>
  </Box>

  <BrowserView>
  <PerfectScrollbar
  component="div"
  style={{
      // height: !matchUpMd ? 'calc(100vh - 56px)' : 'calc(100vh - 88px)
  ',
      height: !matchUpMd ? 'calc(100vh - 56px)' : 'calc(100vh - 88px)',
      paddingLeft: '16px',
      paddingRight: '16px'
  }}
  >
  <MenuList />
  {/* <MenuCard /> */}
  <Stack direction="row" justifyContent="center" sx={{ mb: 2 }}>
  <Chip label="v1.0.0" disabled chipcolor="secondary" size="small" sx={{
    cursor: 'pointer' }} />
```

```
27    </Stack>
28    </PerfectScrollbar>
29    </BrowserView>
30
31    <MobileView>
32    <Box sx={{ px: 2 }}>
33    <MenuList />
34    {/* <MenuCard /> */}
35    <Stack direction="row" justifyContent="center" sx={{ mb: 2 }}>
36    <Chip label="v1.0.0" disabled chipcolor="secondary" size="small" sx={{
        cursor: 'pointer' }} />
37    </Stack>
38    </Box>
39    </MobileView>
40
41    </>
42    );
```

Code snippet 11: Frontend Sidebar Component

3. Content Components: The content component will be displayed based on the matched predefined route. Once the route is determined, the content component is then lazy-loaded into the page. This means that the component will only be fetched and rendered when it is needed, which optimizes performance and improves loading times by minimizing the amount of initial content loaded.

```
1 const DashboardDefault = Loadable(lazy(() => import('views/homepage')));
2 const EditProfile = Loadable(lazy(() => import('views/EditProfile')));
3 const MyTickets = Loadable(lazy(() => import('views/MyTickets')));
```

Code snippet 12: Example of Lazy Load Frontend Components

4.1.3 Responsive Design Implementation

Using Material-UI's grid system, responsive components, and utility features enables us to create a flexible and adaptive user interface in React applications. This ensures that the Web UI looks good and functions well on devices of all sizes, enhancing the overall user experience.

1. Grid System:

MUI provides a powerful Grid component that allows us to create responsive layouts easily. It uses a 12-column layout, and we can specify how many columns a component should span at different screen sizes.

```
import { Grid } from '@mui/material';

  <Grid container spacing={2}>
    <Grid item xs={12} sm={6} md={4}>
      <Card>Content 1</Card>
    </Grid>

    <Grid item xs={12} sm={6} md={4}>
      <Card>Content 2</Card>
    </Grid>

    <Grid item xs={12} md={4}>
      <Card>Content 3</Card>
    </Grid>
  </Grid>
```

Code snippet 13: MUI Grid System

2. Media Queries:

   MUI allows us to utilize CSS media queries through the sx prop or styles to create responsive designs that adapt to different screen sizes without much hassle.

```
<Box sx={{
    display: { xs: 'block', sm: 'flex' },
    flexDirection: { xs: 'column', sm: 'row' }
}}>
{/* Our content */}
</Box>
```

Code snippet 14: MUI Media Queries

4.2 Backend Implementation

4.2.1 Source code structure

Module/component name	Description
config/	contains server configurations, database connection class, constants.
controllers/	contains API controllers.
middleware/	contains server middleware (token authentication, logger, mailer).
routes/	defines explicit API routes
sql/	contains all SQL queries.
uploads/	saves the user's uploaded attachments
utils/	contains shared utilities across modules
index.js	root (start module) module of the backend server
package.json	contains custom scripts and backend packages information
.env	contains sensitive backend configurations

Table 33: Backend source code structure

4.2.2 Setting up ExpressJs Server

The implementation below sets up a web server with essential middleware and route handling for a robust application. It initializes the app, enabling JSON parsing, cookie parsing, and Cross-Origin Resource Sharing (CORS) with restricted options. The server then defines various API endpoints for authentication and resource management, including user, dorm, ticket, attachment, rating, message, role, notification, announcement, feedback, logs, and reports. Each route corresponds to a specific functionality, organized under the /auth and /api/v1/ prefixes, ensuring a structured and scalable approach to managing the application's various services.

```
const app = express();
app.use(express.json());
app.use(cookieParser()); // Enable cookie parsing
app.use(cors(WebConfig.corsOptions));


```

```
// ==============================|| Routes ||==============================
   //
app.use("/auth", authRoutes);
app.use("/api/v1/users", userRoutes);
app.use("/api/v1/dorms", dormRoutes)
app.use("/api/v1/tickets", ticketRoutes);
app.use("/api/v1/attachments", attachmentRoutes);
app.use("/api/v1/rating", ratingRoutes);
app.use("/api/v1/messages", messageRoutes);
app.use("/api/v1/roles", roleRoutes);
app.use("/api/v1/notification", notificationRoutes);
app.use("/api/v1/announcement", announcementRoutes);
app.use("/api/v1/feedback", feedbackRoutes);
app.use("/api/v1/logs", logsRoutes);
app.use("/api/v1/reports", reportRoutes);
```

Code snippet 15: ExpressJS Server Setup

4.2.3 Database Integration

```
import pkg from 'pg';
import dotenv from 'dotenv';

dotenv.config();

const { Pool } = pkg;

const pool = new Pool({
  user: String(process.env.PG_DB_USER),
  host: String(process.env.PG_DB_HOST),
  password: String(process.env.PG_DB_PASSWORD),
  port: Number(process.env.PG_DB_PORT),
  database: String(process.env.PG_DB_DATABASE),
});

export default pool;
```

Code snippet 16: Server connects to PostgreSQL Database

```
import redis from 'redis';

const redisClient = redis.createClient({
  socket: {
    host: process.env.REDIS_HOST,
    port: process.env.REDIS_PORT,
  },
});


const connectRedis = async () => {
  await redisClient.connect();
  redisClient.on('error', (err) => {
    console.error('Redis error:', err);
  });
  console.log('Redis connected successfully');
};
```

Code snippet 17: Server connects to Redis

4.2.4 Security Measures

5 User Manual

5.1 Sign in

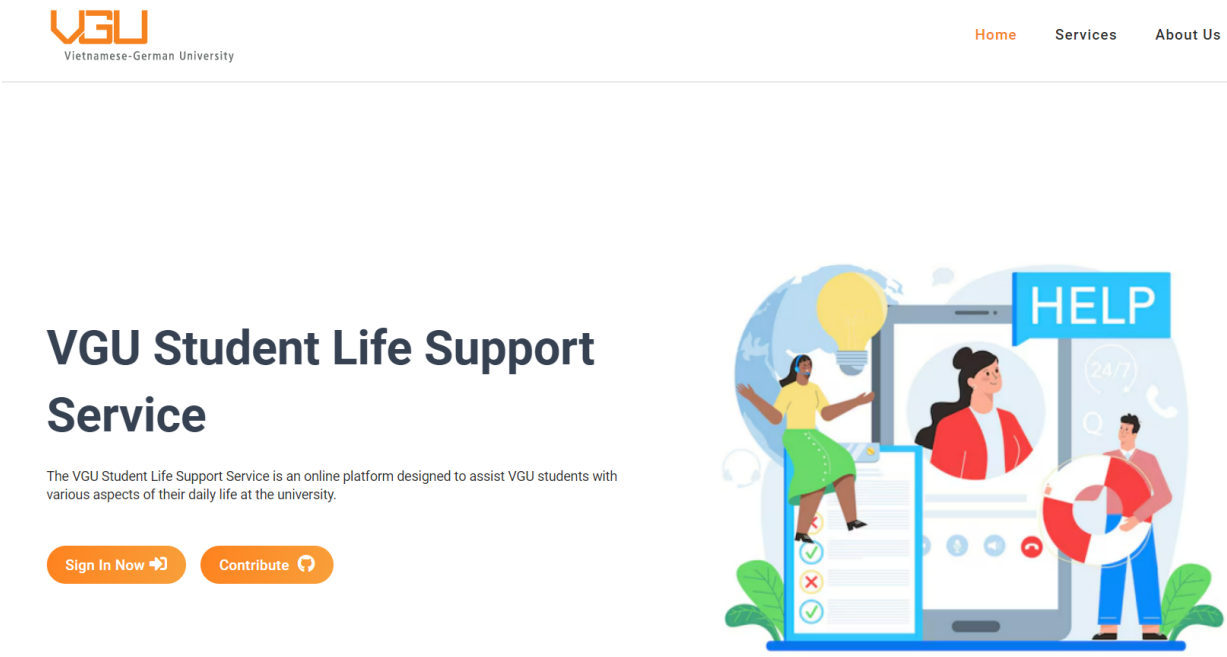

Figure 21: Landing Page

In the landing page of the service, navigate to Login page by clicking "Sign In Now"

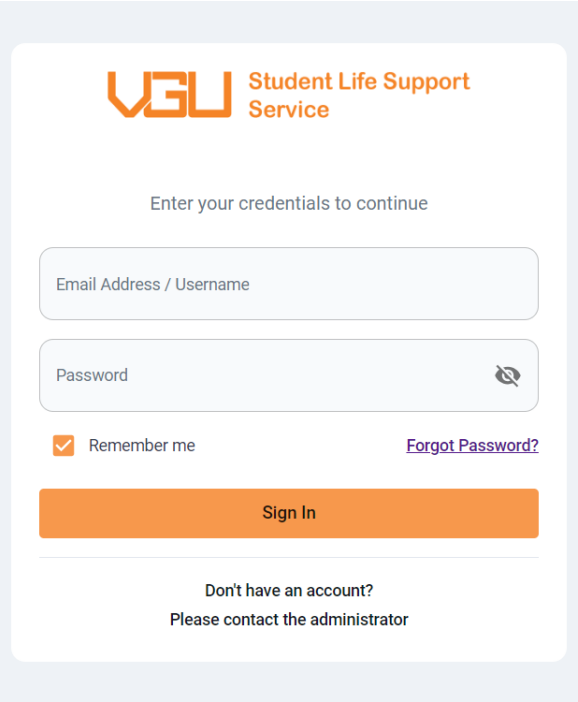

Figure 22: Sign in Form

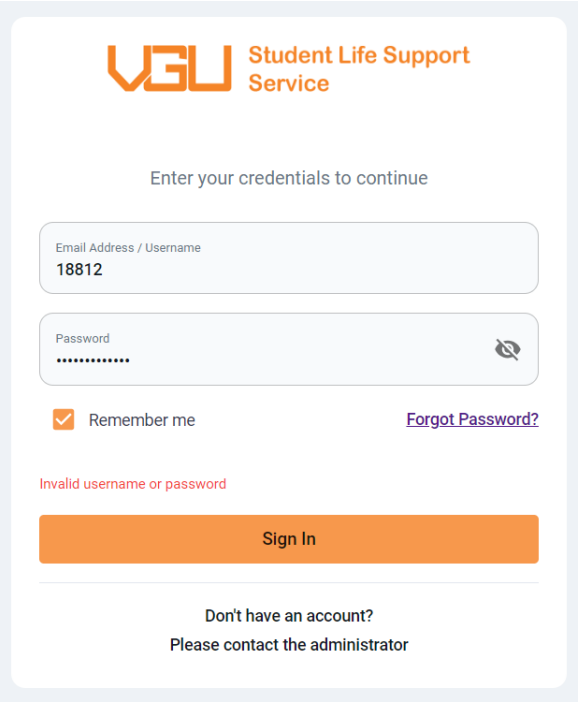

Figure 23: Failed Sign in attempt

To access the service, input your username (or email) and password, then click on the sign-in button (refer to Figure 22). If you provide incorrect credentials, a warning message will appear, indicating that you need to correct either your username or password (Figure 23).

5.2 Forgot password

If you forget your password, you can initiate a reset by selecting the 'Forgot Password?' option on the 'Sign in' form (refer to Figure 22). Subsequently, provide your email address in the 'Forgot Password' form and click on 'Reset Password' (see Figure 24).

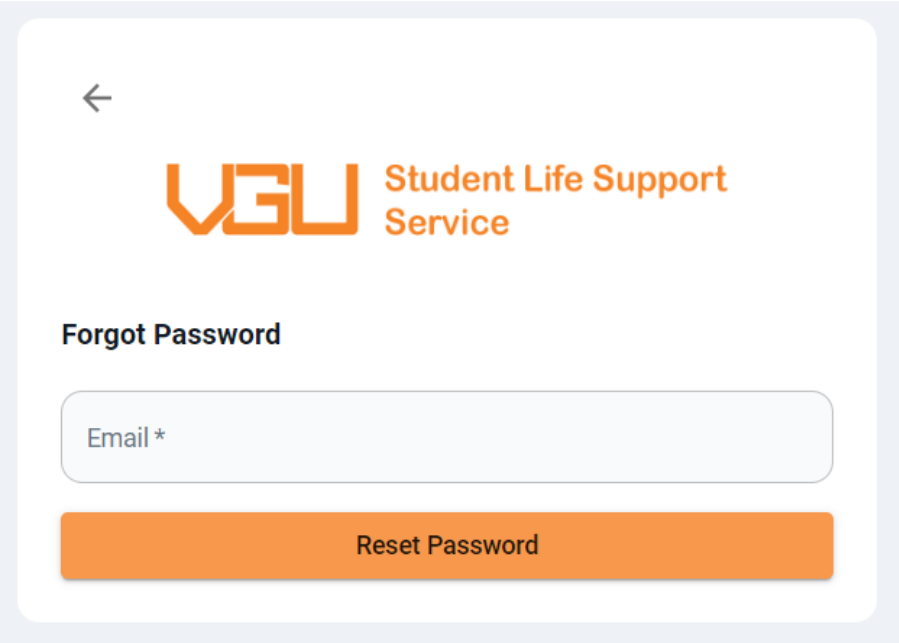

Figure 24: Reset Password Form

If your email is associated with an account in the system, a success notification will appear, and password reset instructions will be sent to your email (Figure 25, 27). Conversely, if the email is not found in the system, a failure notification will indicate that the email does not exist (Figure 26).

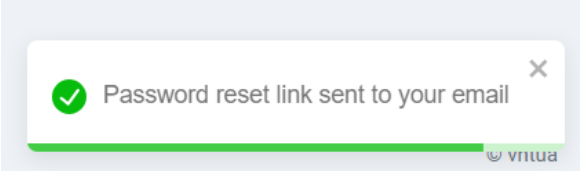

Figure 25: Reset password successfully

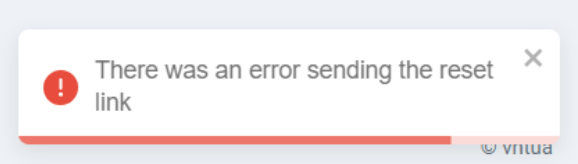

Figure 26: Reset password failed

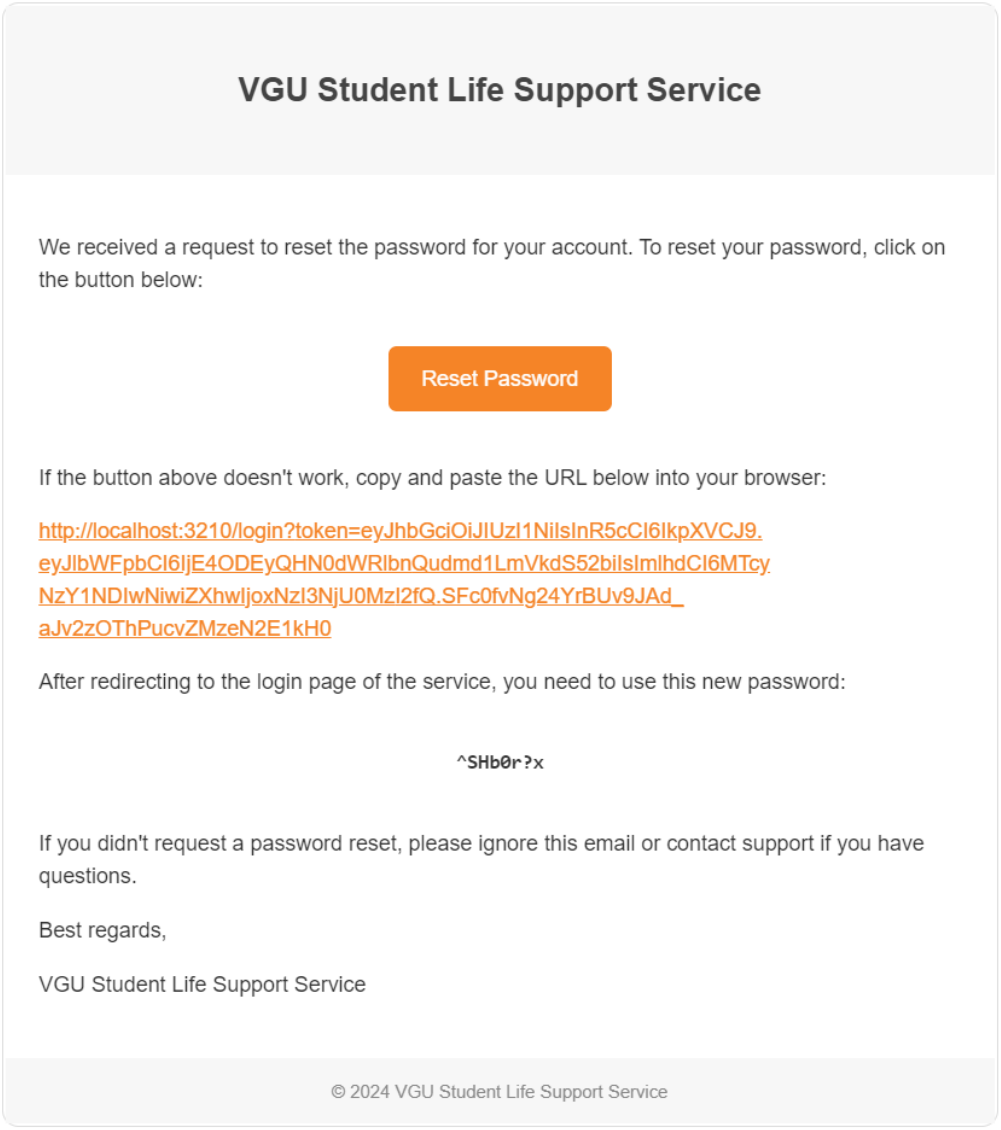

Figure 27: Reset password email instructions

5.3 Student's functions

After logging in successfully, students will be navigated to the homepage

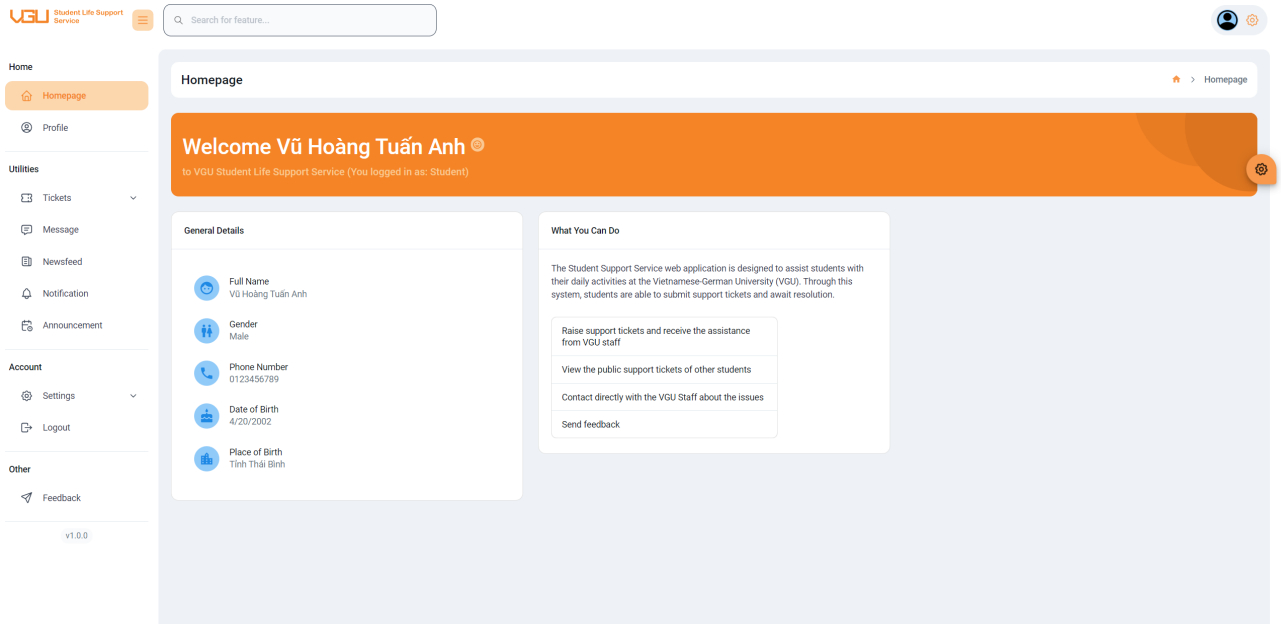

Figure 28: Student's Home Page

5.3.1 View Profile

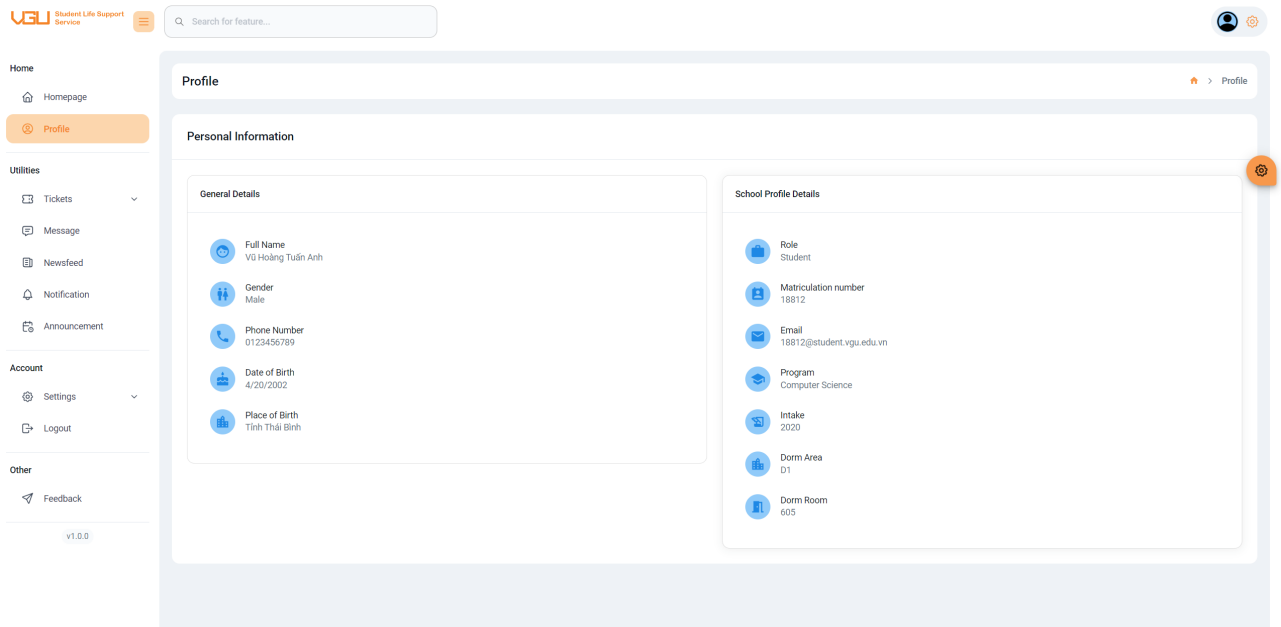

Figure 29: Student's Profile Page

Students can access their personal information through the "Profile" menu. This section provides an overview of their personal details, allowing students to view and manage their account-related information.

5.3.2 View Tickets

Students can access their tickets by navigating to `Tickets > My Tickets`. This page displays a table listing all of the students' tickets along with a ticket details panel. To view the details of a specific ticket, students can click the visibility icon corresponding to the selected ticket in the table. (see Figure 30)

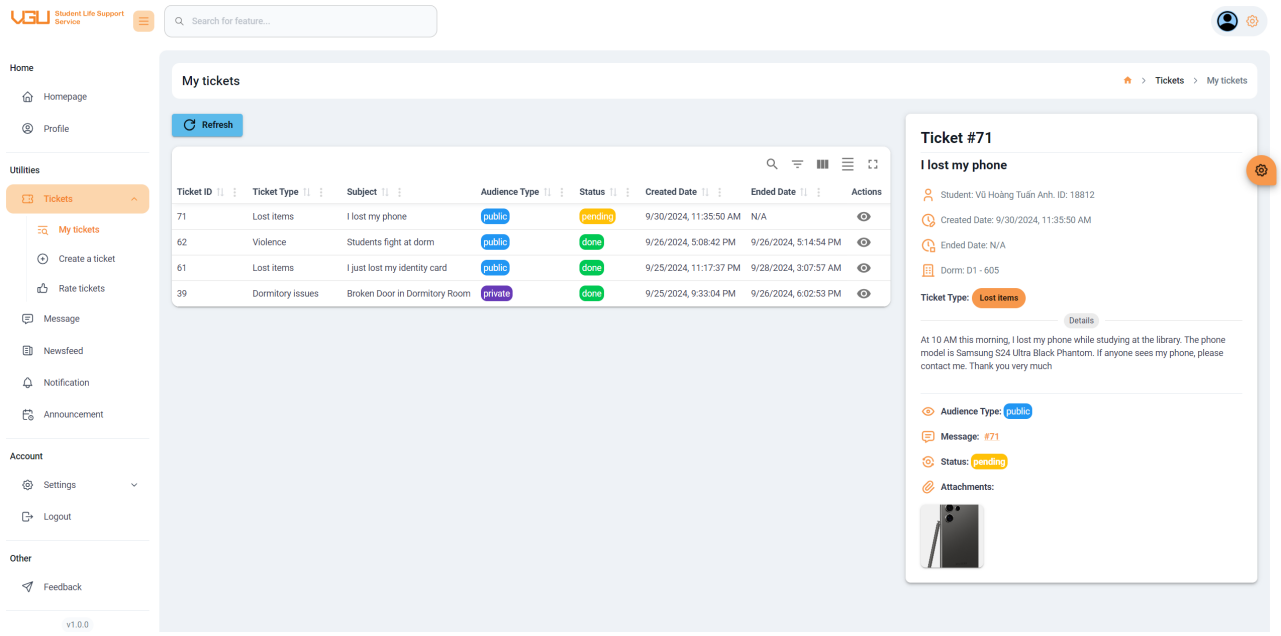

Figure 30: Student's Tickets List Page

5.3.3 Create Tickets

Students can submit a new support ticket by selecting `Tickets > Create a ticket`. After completing the form with all required information and clicking 'Create This Ticket', the ticket will be marked as pending, awaiting approval from an admin and resolution by the staff. (Figure 31)

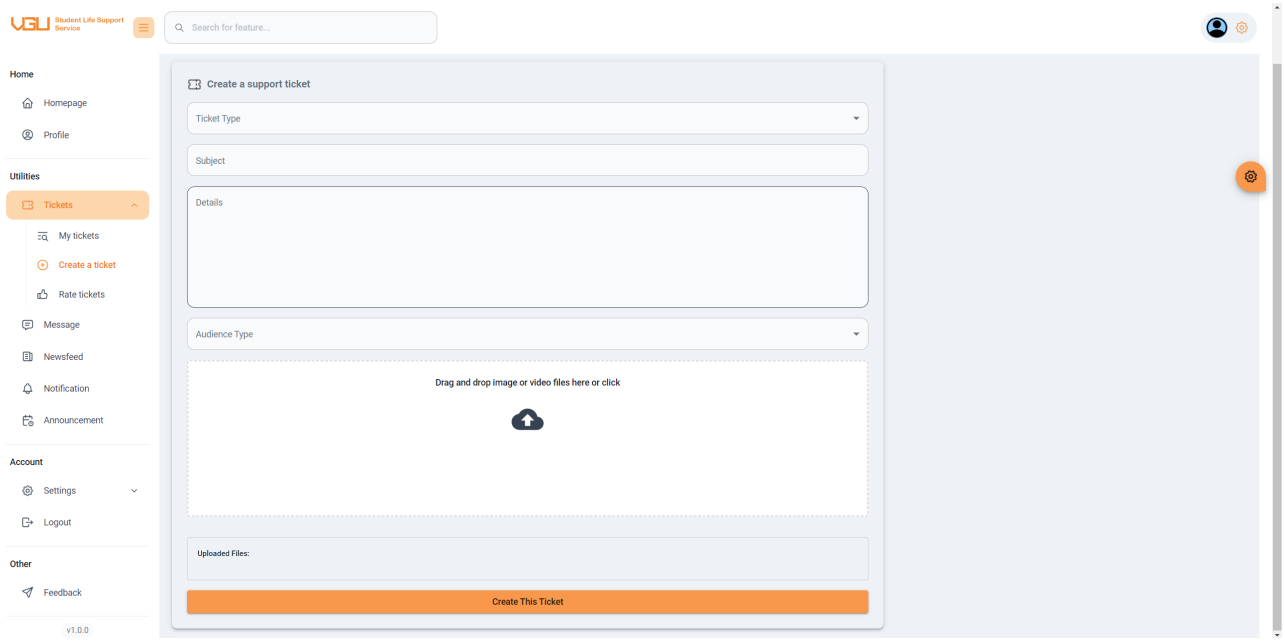

Figure 31: Student's Create Tickets Page

5.3.4 Rate Tickets

Once a ticket has been marked as completed, students have the option to provide feedback by rating the ticket through the "Rate Tickets" menu. To submit a rating, students can simply click the "Rate" action corresponding to each completed ticket and input their rating score before submitting it. (see Figures 32, 33)

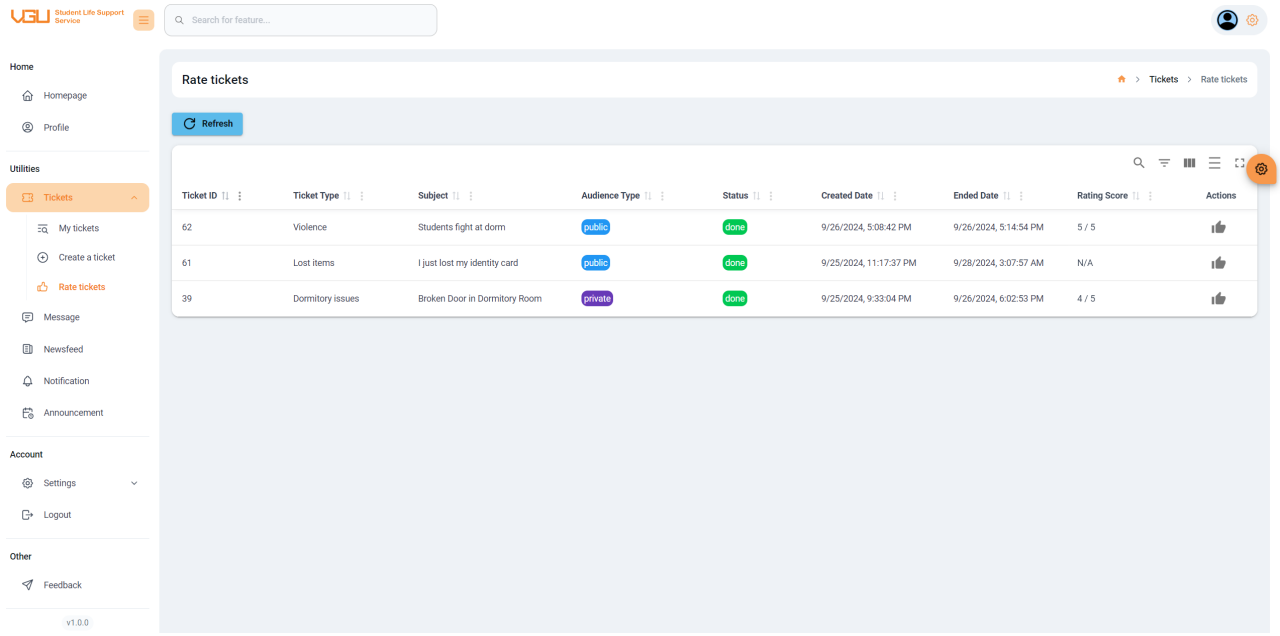

Figure 32: Student's Rate Tickets Page

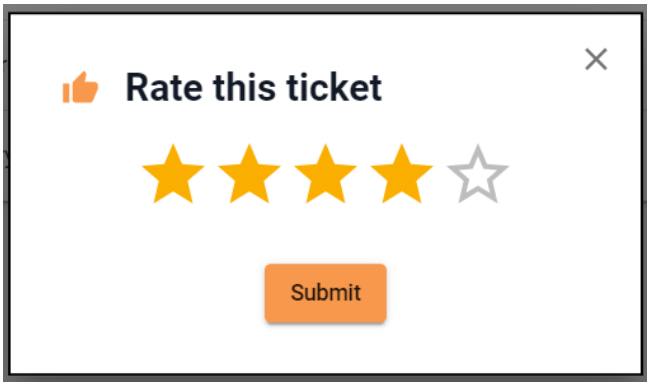

Figure 33: Submit a ticket rating

5.3.5 Message

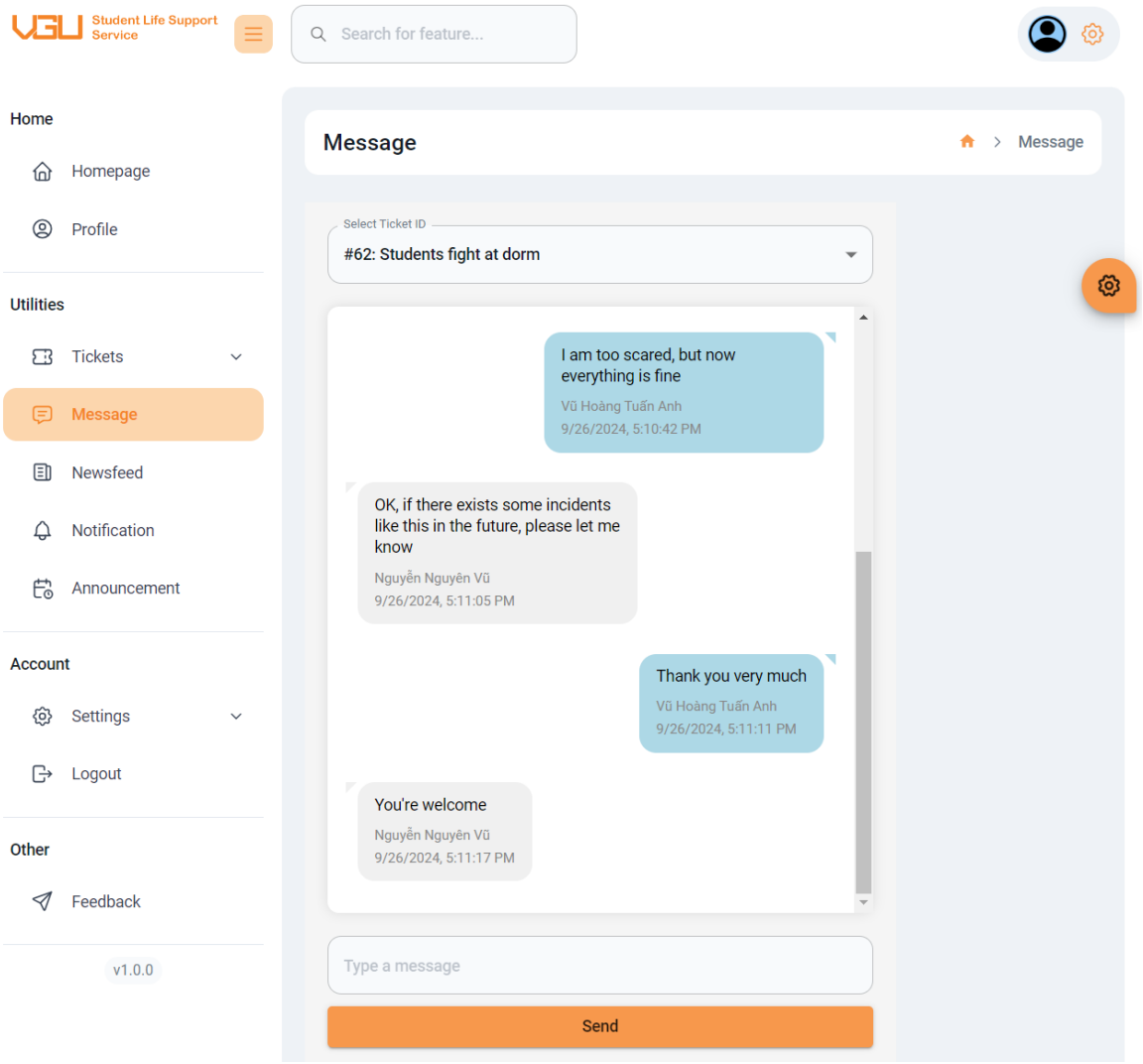

Figure 34: Student's Message Page

Students can communicate with the staff assigned to their tickets by going to the "Message" menu. In this section, they can select the conversation associated with the ticket ID and send messages directly to the staff member handling their case. (Figure 34)

5.3.6 Newsfeed

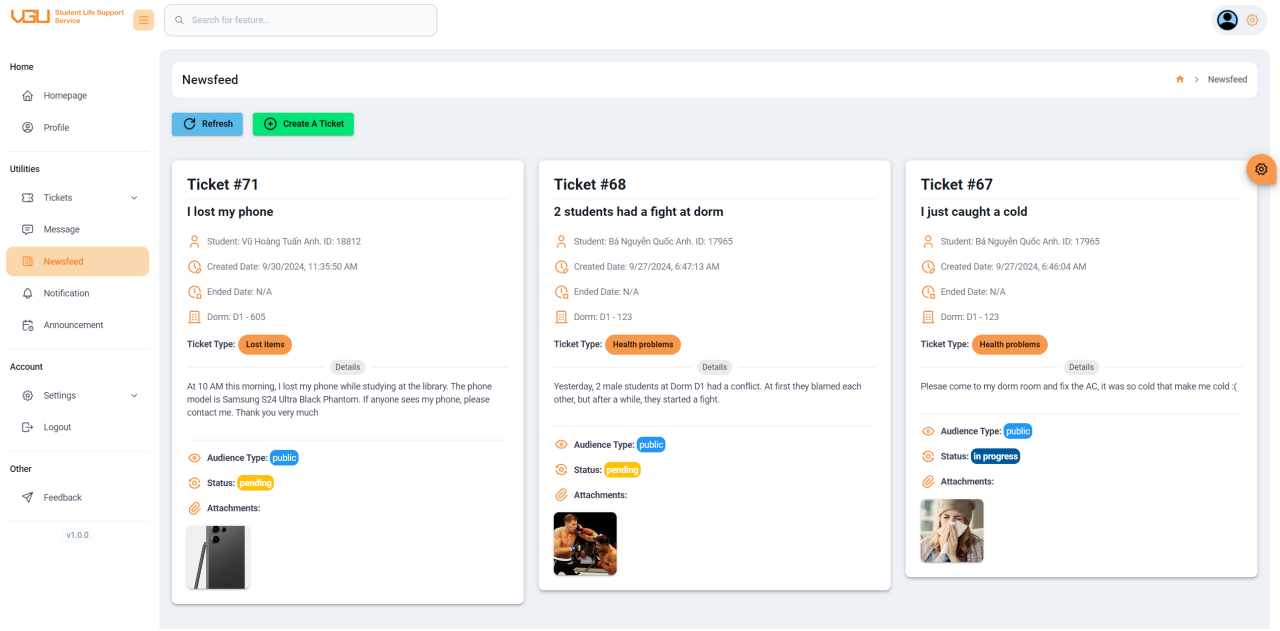

Figure 35: Student's Newsfeed

Students have the ability to view public support tickets submitted by other students through the "News Feed" menu. This section provides access to a list of publicly shared tickets, allowing students to stay informed about common issues, solutions, or inquiries raised by their peers. (Figure 35)

5.3.7 Notification

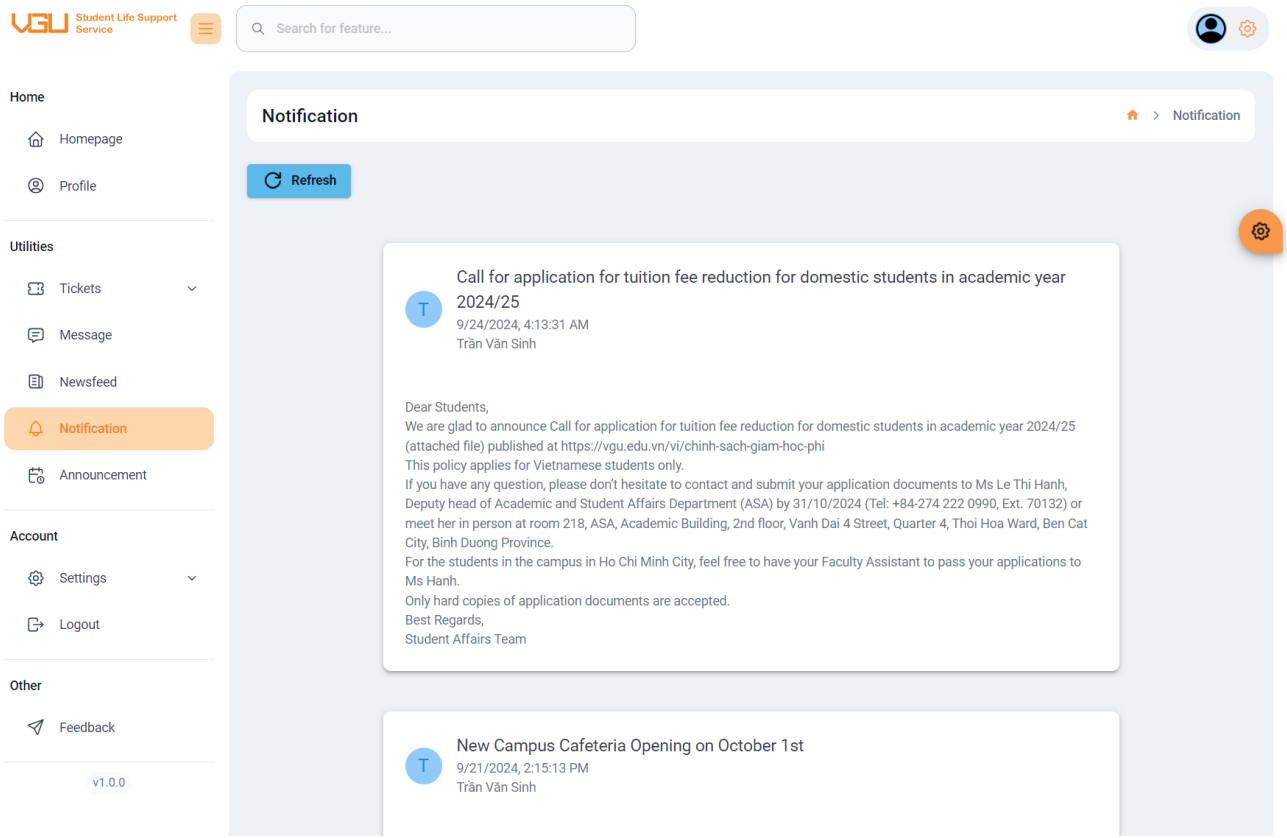

Figure 36: Student's Notification Page

Students can receive important notifications from staff or administrators in the "Notification" menu. This section is dedicated to keeping students updated on various announcements, ticket status changes, or other critical communications from the support team. By regularly checking this menu, students ensure they stay informed about any new developments or actions that require their attention. Notifications may include updates on submitted tickets, administrative notices, or responses to queries. (Figure 36)

5.3.8 Announcement

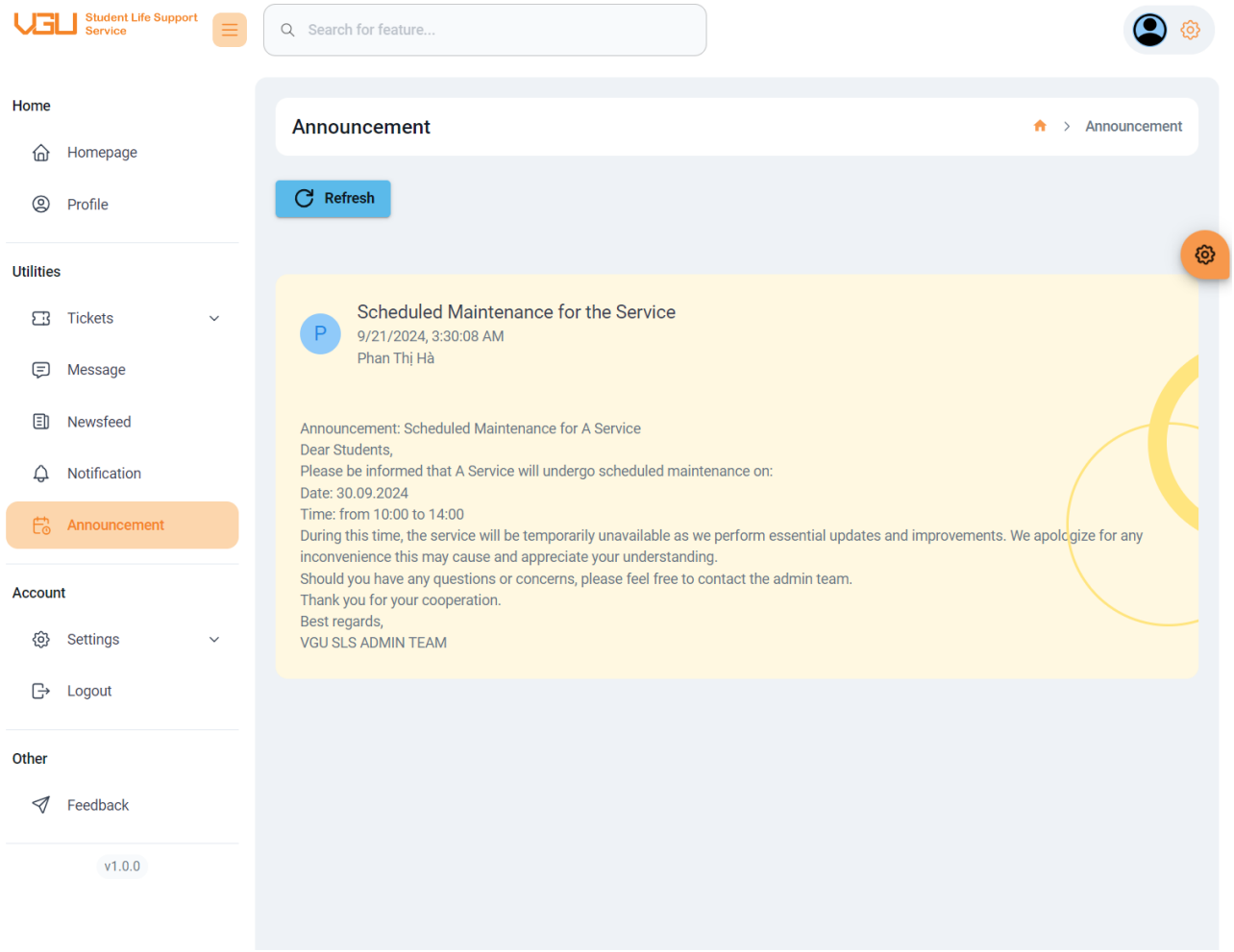

Figure 37: Student's Announcement Page

Students can access important announcements from staff or administrators in the "Announcement" menu. This dedicated section provides students with updates and information relevant to their academic and campus life. By regularly checking the "Announcement" menu, students can stay informed about events, policy changes, or other significant news that may affect them. (Figure 37)

5.3.9 Settings

Students have the ability to modify their personal information and change their account password through the "Settings" menu. This feature allows students to ensure that their details are up to date, which is crucial for effective communication and account security. By navigating to the "Settings" menu, students can easily access fields to edit their name, contact information, and other relevant details. Additionally, they can update their password to enhance their account's security, ensuring that their personal data remains protected. This functionality empowers students to take control of their profiles and maintain accurate and secure information.

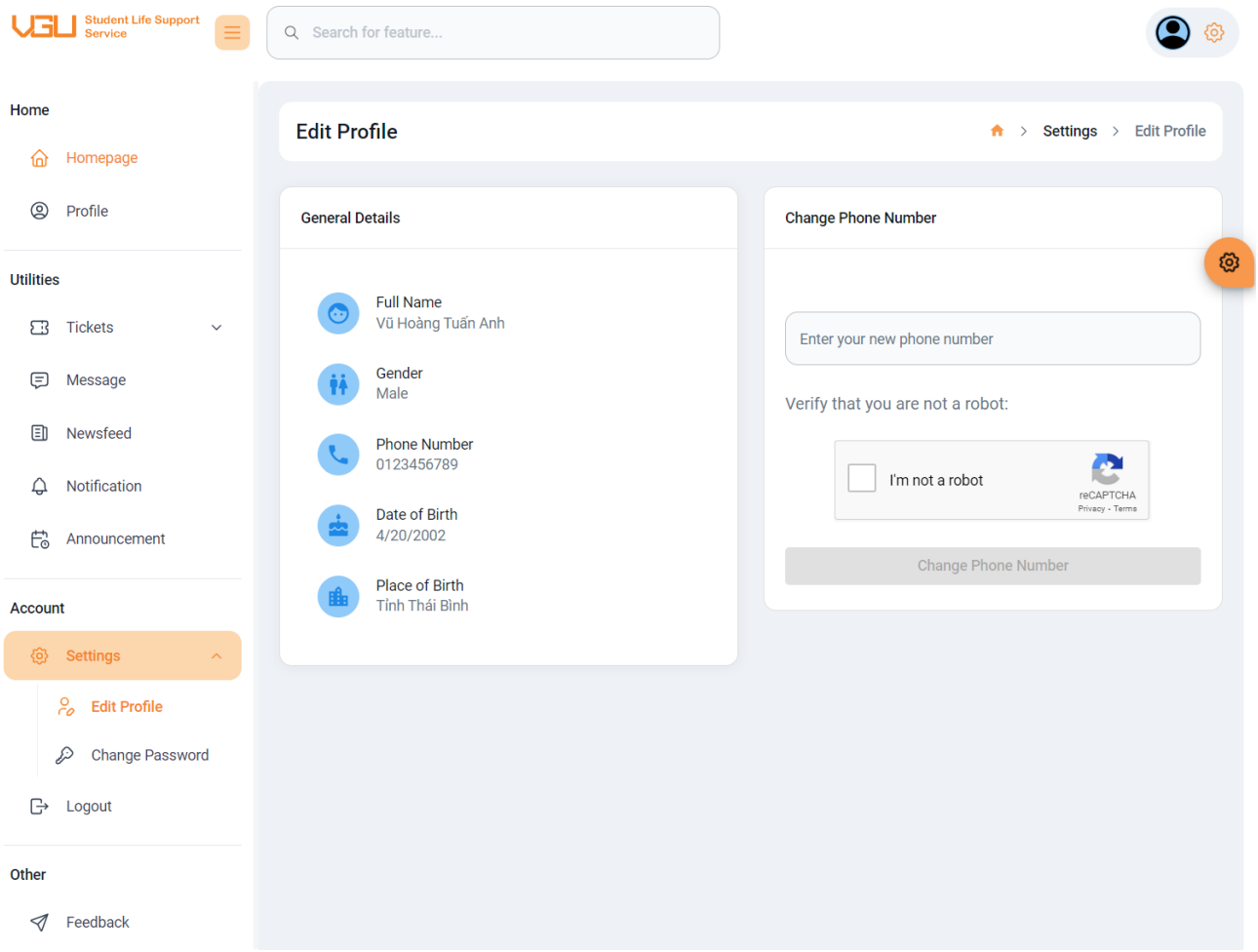

Figure 38: Student's Edit Profile Page

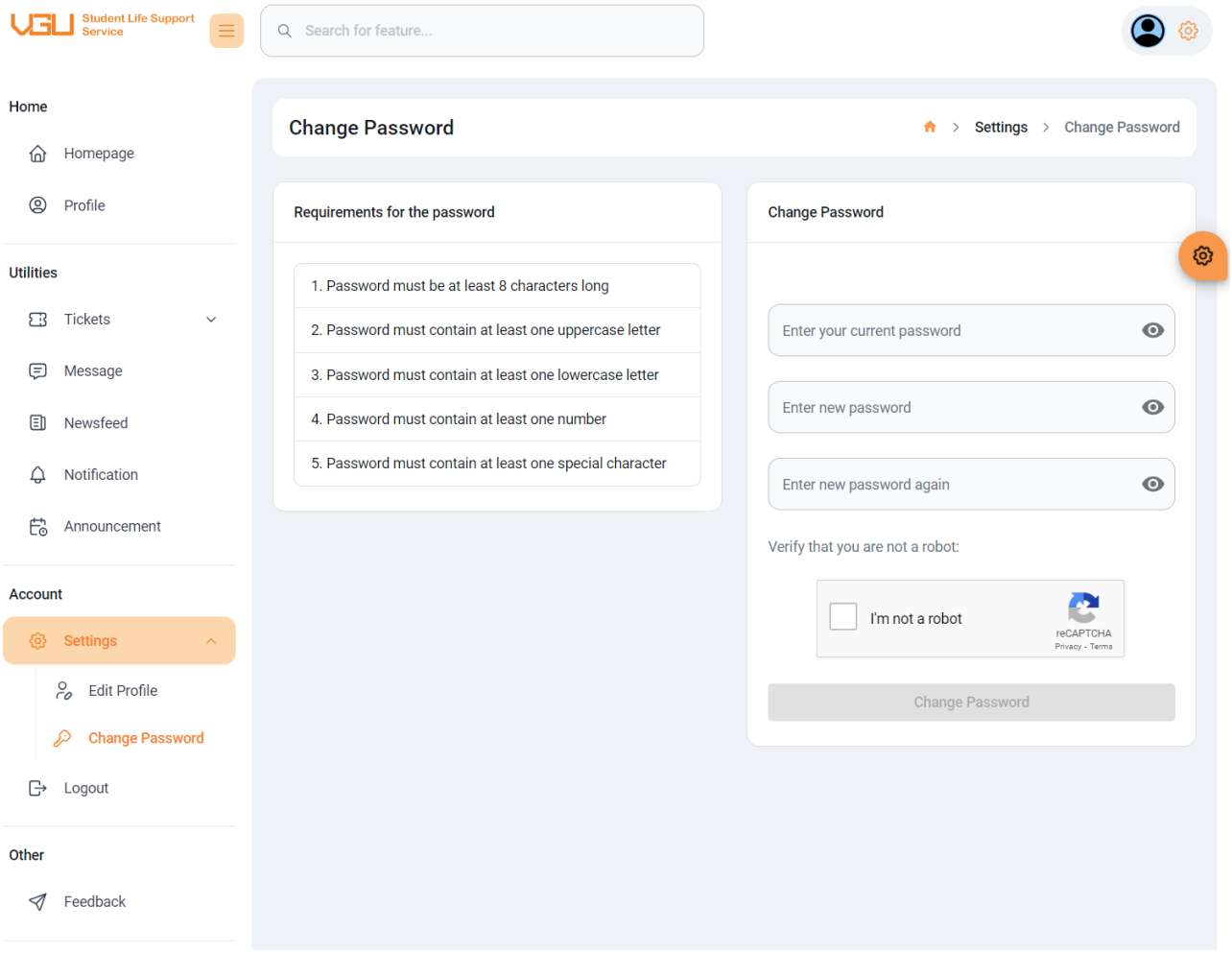

Figure 39: Student's Change Password Page

5.3.10 Feedback

Students have the opportunity to provide feedback on the system by accessing the "Feedback" menu. Within this section, they can express their thoughts and experiences regarding the platform, which is essential for continuous improvement and user satisfaction. Additionally, students are encouraged to rate their experience by selecting a rating score before submitting their feedback. This structured approach allows them to convey not only qualitative insights but also quantitative assessments, enabling the administrators to better understand user satisfaction and identify areas that may require enhancements. By engaging in this feedback process, students contribute to the overall development and effectiveness of the system. (see Figure 40)

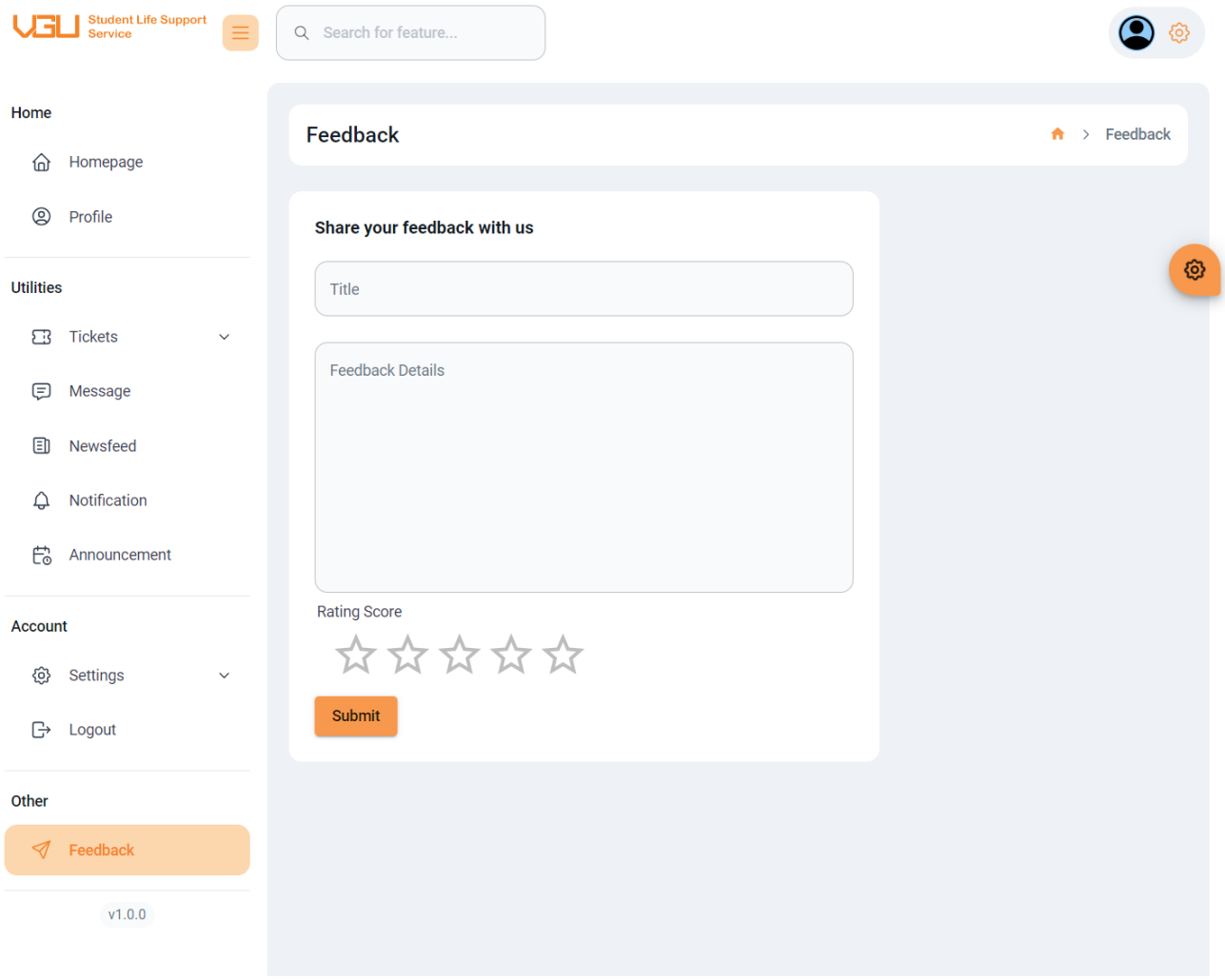

Figure 40: Student's Feedback Page

5.4 Staff's functions

5.4.1 Available Tickets

Staff members have access to the "Available Tickets" menu, where they can view a comprehensive list of all pending tickets. This menu serves as a centralized hub for managing incoming requests from students, allowing staff to efficiently monitor and address each ticket's status. Once they navigate to this section, staff can review the details of each pending ticket, including the nature of the issue raised by the student. From there, they can take appropriate actions

to resolve the issues, whether that involves communicating with the student for further clarification, assigning the ticket to the relevant department, or providing direct assistance. This streamlined process ensures that all student inquiries are handled promptly and effectively, contributing to a more responsive support system.

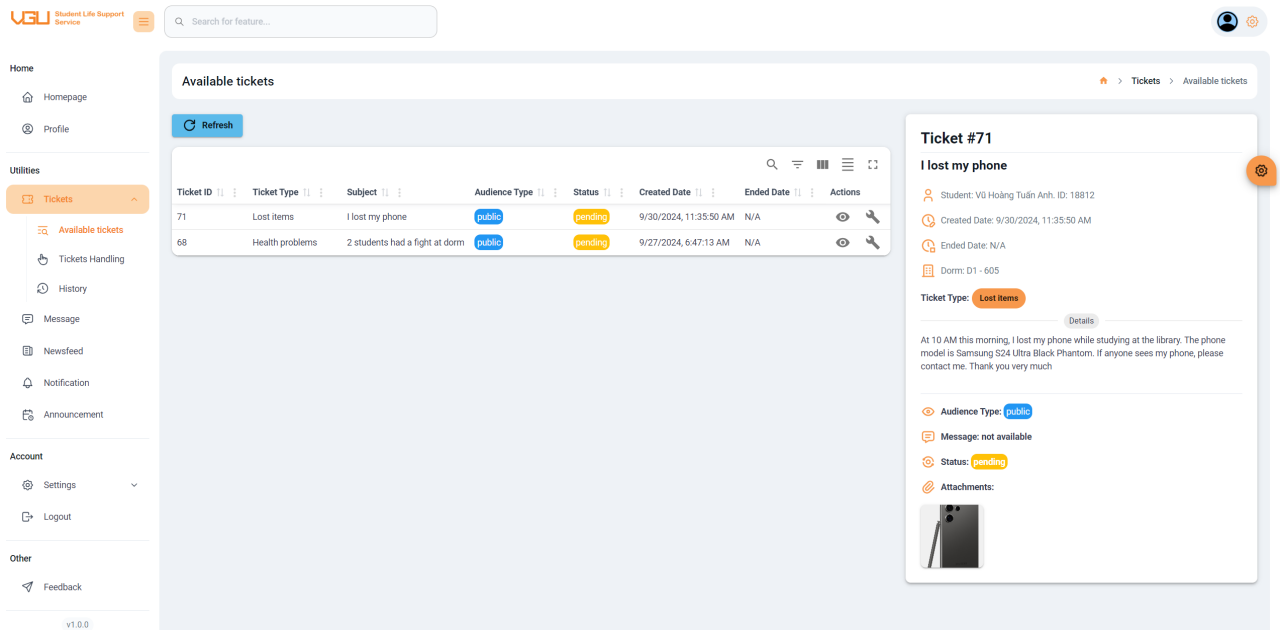

Figure 41: Staff's Available Tickets Page

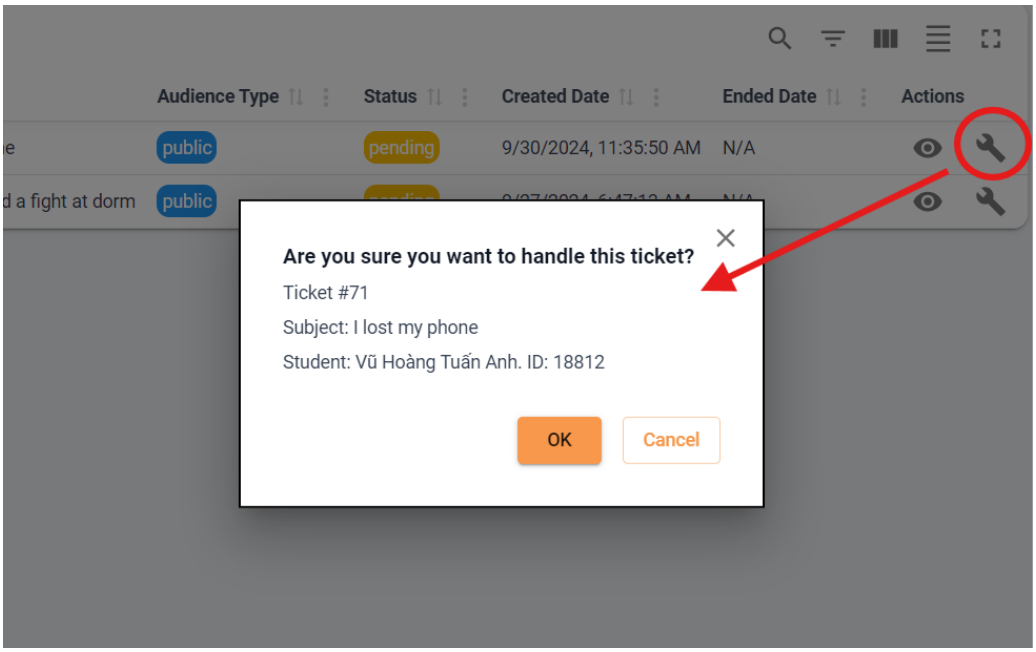

Figure 42: Handle a ticket

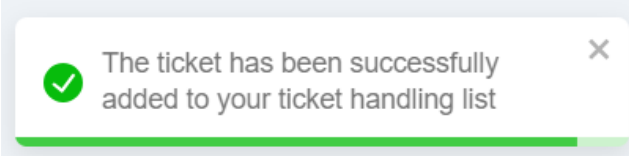

Figure 43: Successfully add a ticket to ticket handling list

5.4.2 Tickets Handling

Within the "Ticket Handling" menu, staff members have the ability to manage tickets efficiently by marking them as completed or canceling them as necessary. This menu provides a dedicated interface for staff to oversee the progress of each ticket. When a ticket is resolved to the satisfaction of the student or when the issue has been adequately addressed, staff can easily mark it as "done." This action not only updates the ticket's status in the system but also informs the student that their request has been successfully fulfilled. Alternatively, if a ticket needs to be canceled—perhaps due to a change in the student's needs or if the issue is no longer relevant—staff can take this action as well. By allowing for these functionalities, the Ticket Handling menu plays a crucial role in maintaining clear communication between staff

and students while ensuring that all ticket statuses are accurately reflected in the system.

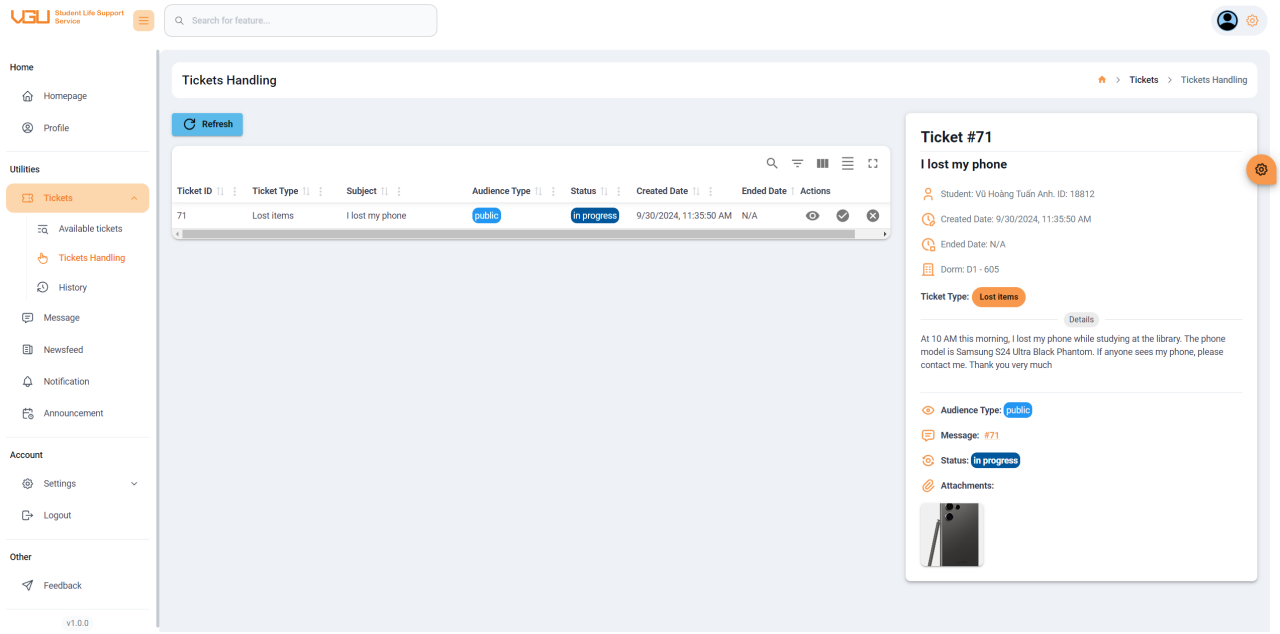

Figure 44: Staff's Tickets Handling List

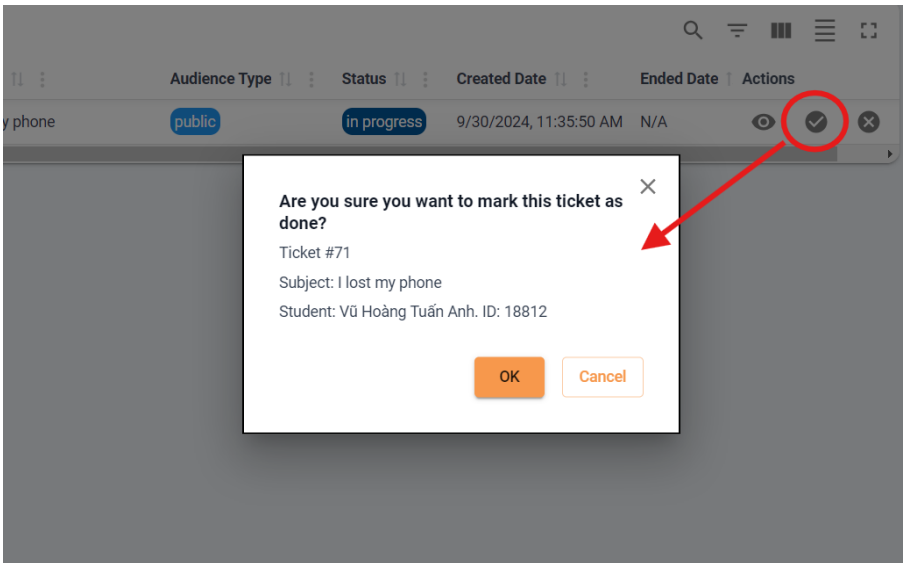

Figure 45: Mark a ticket as done

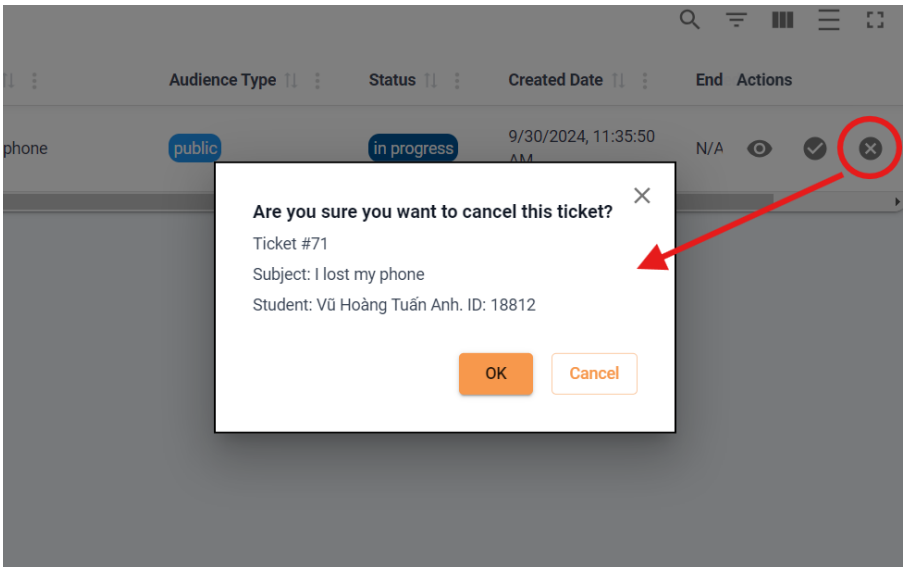

Figure 46: Cancel a ticket

5.4.3 Tickets History

Staff members have the capability to access a comprehensive record of all past tickets through the "Tickets History" section. This feature allows them to review and analyze previously handled tickets, providing valuable insights into the nature of the inquiries and issues raised by students. In the Tickets History, staff can examine the details of each ticket, including the submission date, resolution status, and any notes or comments that were recorded during the handling process. This historical data is essential for assessing the effectiveness of past responses, identifying recurring issues, and improving the overall support process. By utilizing the information available in the Tickets History, staff can enhance their understanding of student concerns and refine their approach to future ticket management, ultimately leading to a more efficient and responsive support system. (see Figure 47)

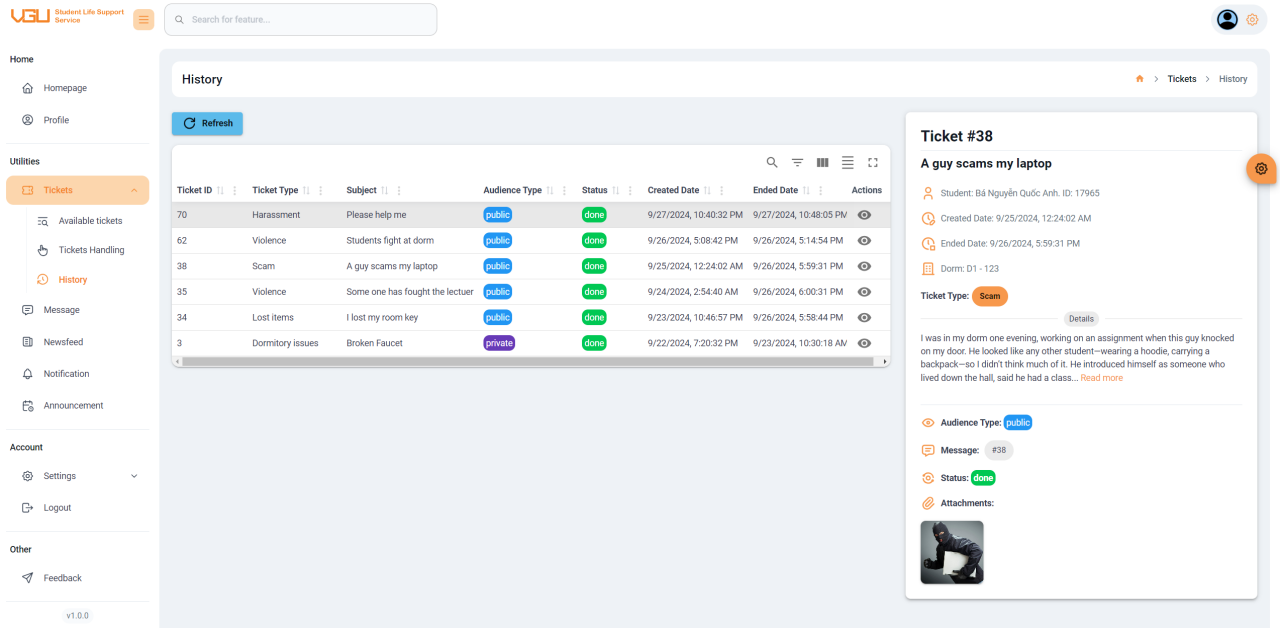

Figure 47: Staff's Tickets History Page

5.4.4 Message

Staff members have the ability to communicate directly with students regarding specific tickets through the "Message" menu. This feature enables staff to send personalized messages to students associated with a particular ticket, facilitating clearer communication and providing necessary updates or clarifications. By selecting the relevant ticket, staff can compose messages that address the student's concerns, offer guidance, or request additional information if needed. This direct messaging capability not only enhances the support experience for students but also fosters a collaborative environment where staff can ensure that students are well-informed throughout the ticket resolution process. (Figure 48)

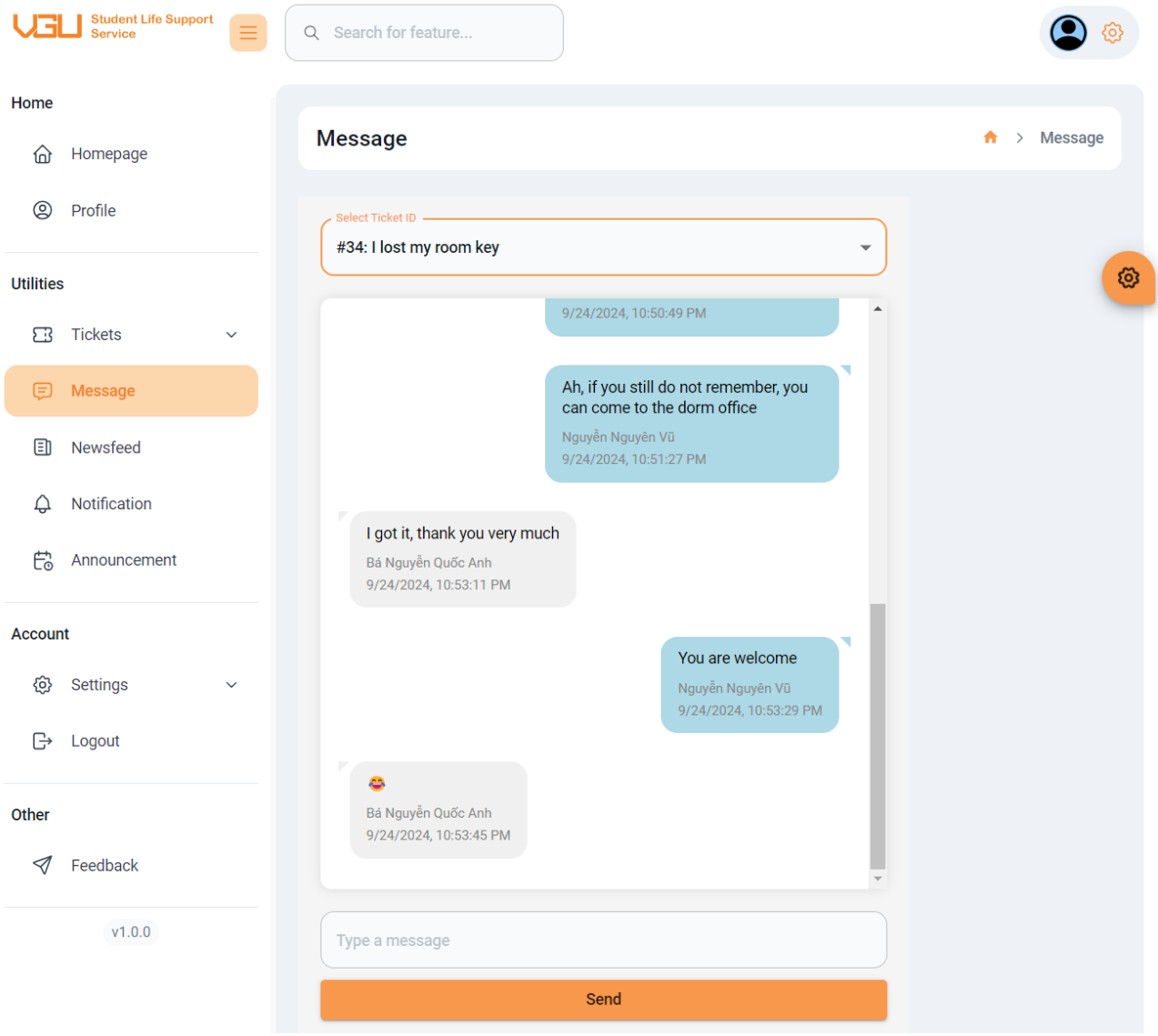

Figure 48: Staff's Message Page

5.4.5 Notification

In addition to viewing existing notifications, staff members have the capability to create new notifications for various purposes. This functionality allows staff to disseminate important information, updates, or alerts to students and other relevant parties effectively. By utilizing the notification creation feature, staff can ensure that critical announcements reach their intended audience in a timely manner. Whether it's communicating changes in policies, upcom-

ing events, or any other significant news, the ability to create notifications enhances the overall communication strategy within the system.

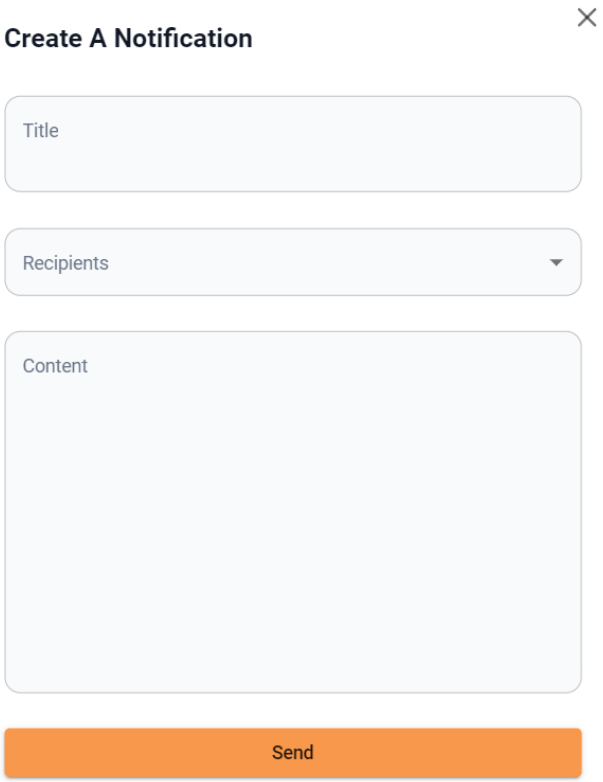

Figure 49: Create a Notification Modal

Staff can tailor the content of the notification to suit specific needs, ensuring that the information is clear and relevant to the recipients. This proactive approach to communication helps maintain an informed community and encourages engagement among students and staff alike.

5.4.6 Announcement

In addition to being able to view existing announcements, staff members also have the option to create new announcements as needed. This feature empowers staff to communicate important updates, news, or information to students and other stakeholders effectively. By utilizing the announcement creation functionality, staff can share relevant details about events, policy changes, or any critical developments within the institution.

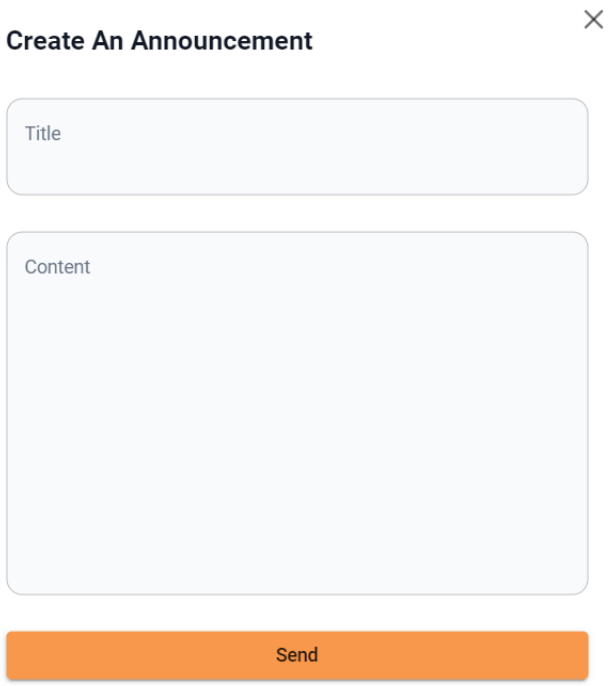

Figure 50: Create an Announcement Modal

The ability to craft announcements allows staff to ensure that their messages are clear, concise, and tailored to the audience they wish to reach. This not only fosters transparency within the organization but also keeps students informed and engaged with the latest happenings. Through the creation of announcements, staff can play a vital role in enhancing communication and ensuring that important information is readily accessible to all members of the community. Overall, this capability supports an open and informative environment, contributing to a more connected and responsive academic community.

5.5 Admin's functions

5.5.1 Tickets Management

Administrators possess the authority to oversee and manage all tickets within the Tickets Management section of the system. This functionality enables them to access a comprehensive list of all existing tickets, providing a clear overview of the current support requests submitted by users.

In addition to simply viewing these tickets, administrators have the capability to delete any tickets that have not received approval or are deemed invalid. This ensures that the ticketing system remains organized and efficient, as it helps to eliminate clutter caused by unnecessary or unapproved tickets. By actively managing the ticketing process, administrators can maintain the integrity of the system, ensuring that only legitimate and relevant tickets are processed.

This management feature not only enhances the overall user experience but also streamlines the workflow for support staff, allowing them to focus on valid tickets that require attention. Ultimately, this proactive approach to ticket management fosters a more efficient and effective support environment, benefiting both administrators and users alike.

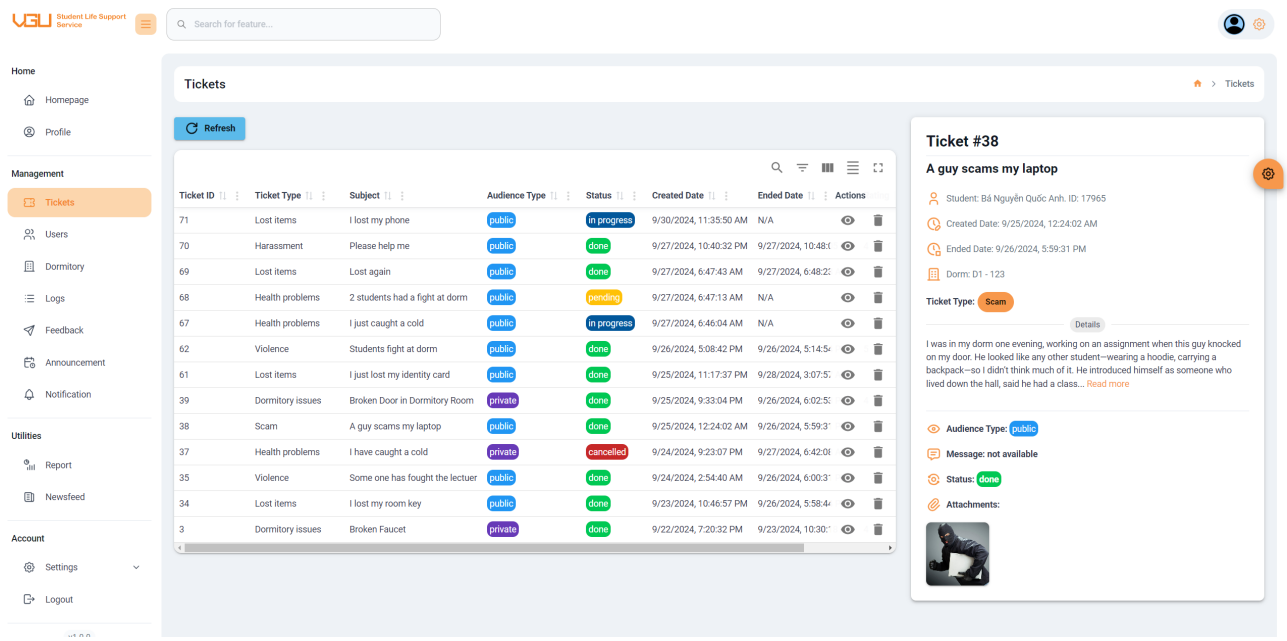

Figure 51: Admin's Ticket Management Page

5.5.2 Users Management

Administrators have comprehensive access to the user management functionalities within the system. They are able to view all registered users, which allows them to monitor user activity and maintain an overview of the community within the platform. This capability ensures that administrators can identify any potential issues or areas that require attention among the

user base.

In addition to merely viewing users, administrators also have the authority to create new user accounts as needed. This feature is particularly useful for onboarding new members, whether they are students, staff, or other stakeholders who require access to the system. Furthermore, administrators can edit the information of existing users, including updating personal details and modifying user roles as necessary. This flexibility is crucial for adapting to changes in user responsibilities or correcting any inaccuracies in user profiles.

Moreover, administrators possess the ability to delete user accounts if required, particularly in cases where users are no longer active or if there are concerns regarding compliance with system policies. This feature aids in maintaining the security and integrity of the system by ensuring that only authorized users have access.

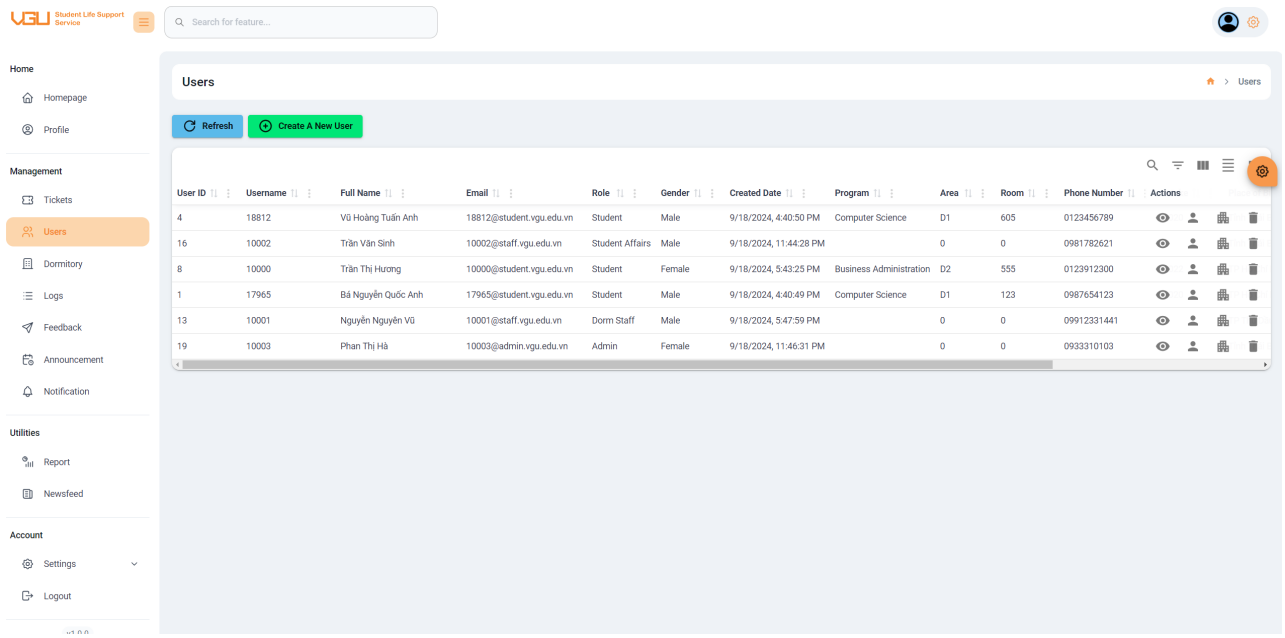

Figure 52: Admin's User Management Page

5.5.3 Dormitory Management

Administrators have the ability to access and manage all dormitory information within the system, including viewing details such as occupancy rates and amenities. They can create new

dorm entries when new facilities are established and delete outdated or erroneous entries to maintain data accuracy. This functionality ensures that the dormitory offerings are accurately represented and helps administrators efficiently oversee housing resources for users.

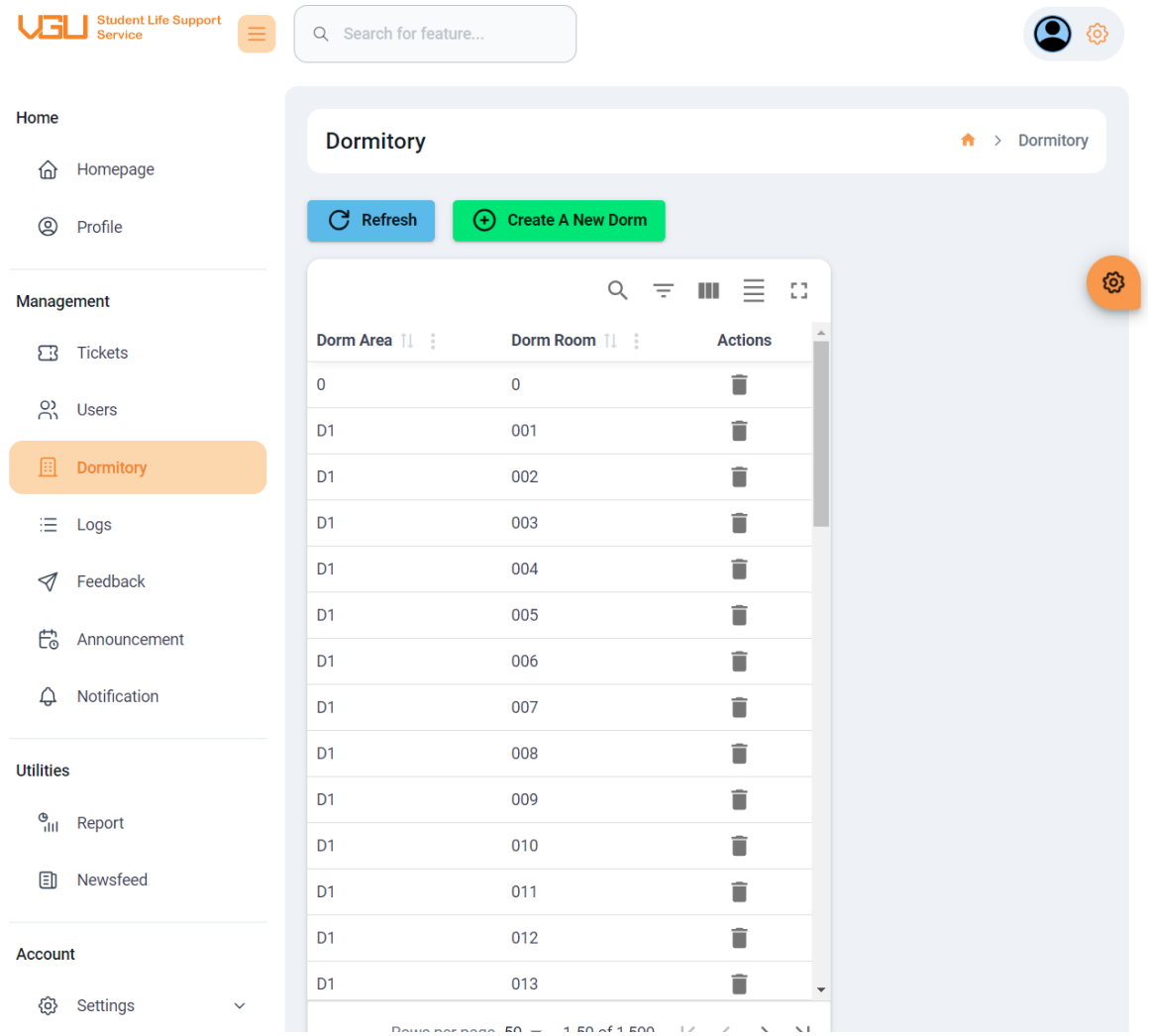

Figure 53: Admin's Dormitory Management Page

5.5.4 Logs Management

Administrators have the capability to access the complete system logs, which provide a detailed record of all activities and events occurring within the system. They can selectively delete individual logs if they find them unnecessary or opt to clear all logs at once to maintain

a cleaner database. Additionally, administrators can export these logs in various formats, including CSV, PDF, or Excel, enabling easier analysis and reporting. This functionality ensures that administrators can efficiently manage log data while also retaining the flexibility to utilize the information in formats that best suit their needs.

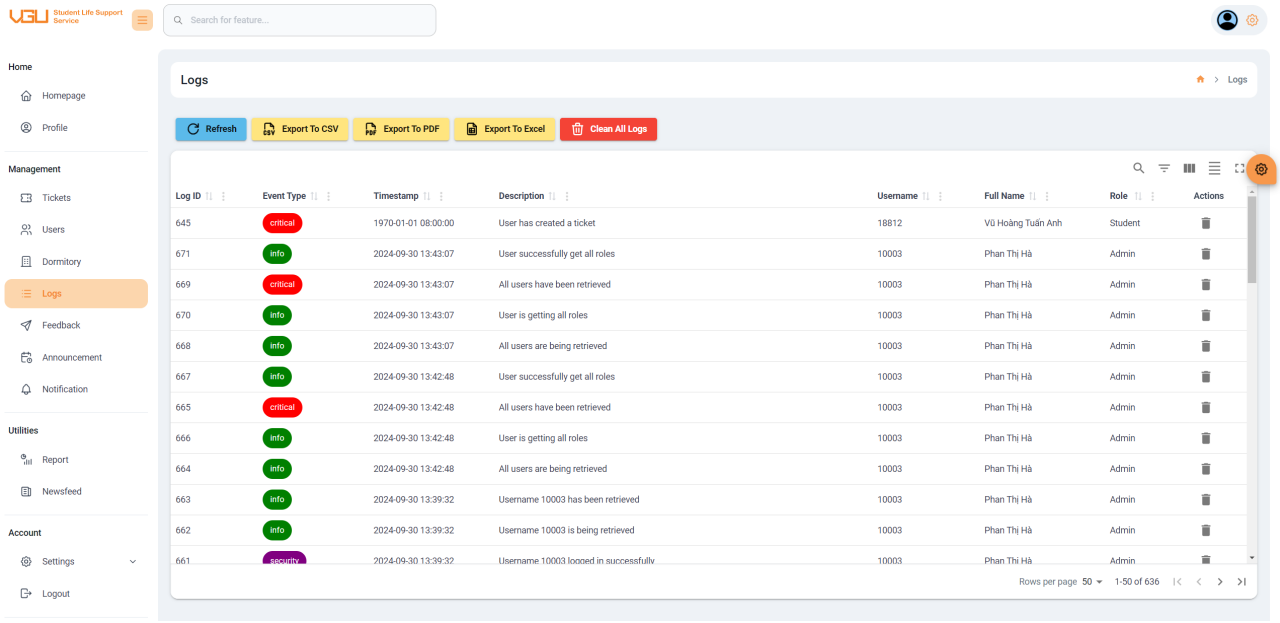

Figure 54: Admin's Logs Management Page

5.5.5 Feedback Management

Administrators have the ability to oversee and manage all feedback submitted by both students and staff members within the system. This includes viewing detailed comments, suggestions, and evaluations provided by users, as well as the corresponding feedback scores. By analyzing this information, administrators can gain valuable insights into the experiences and perspectives of users, helping them identify areas for improvement and enhance the overall functionality and user satisfaction of the system. This comprehensive management of feedback ensures that administrators remain informed about the sentiments and concerns of the user community.

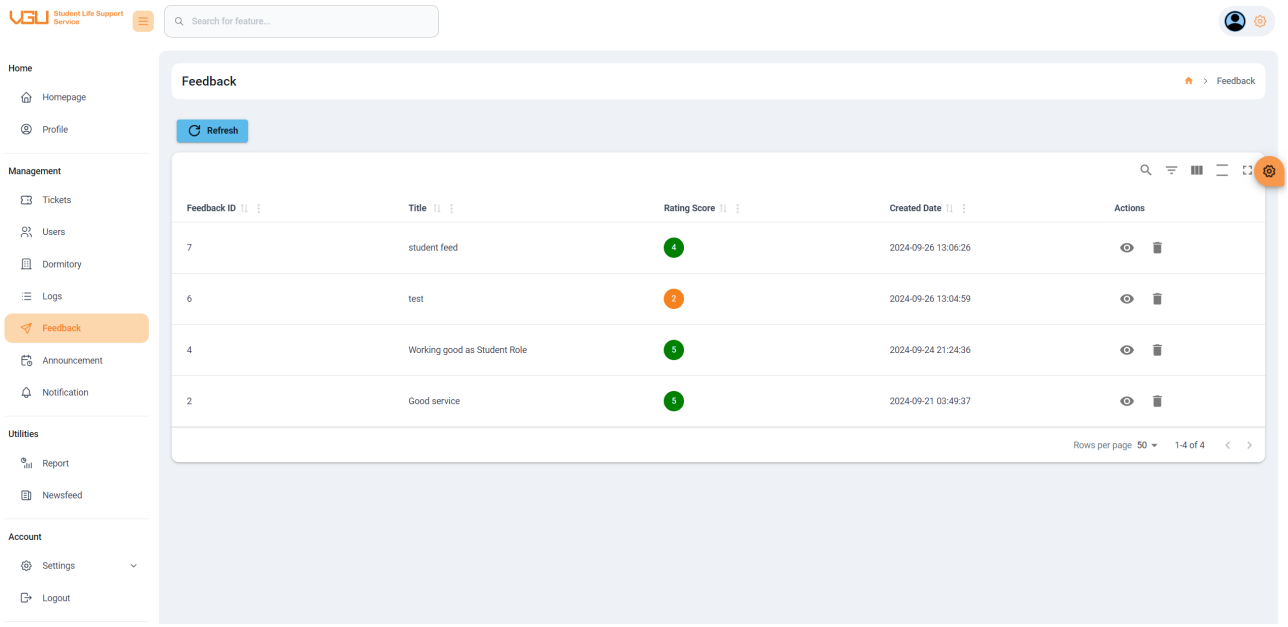

Figure 55: Admin's Feedback Management Page

5.5.6 Report

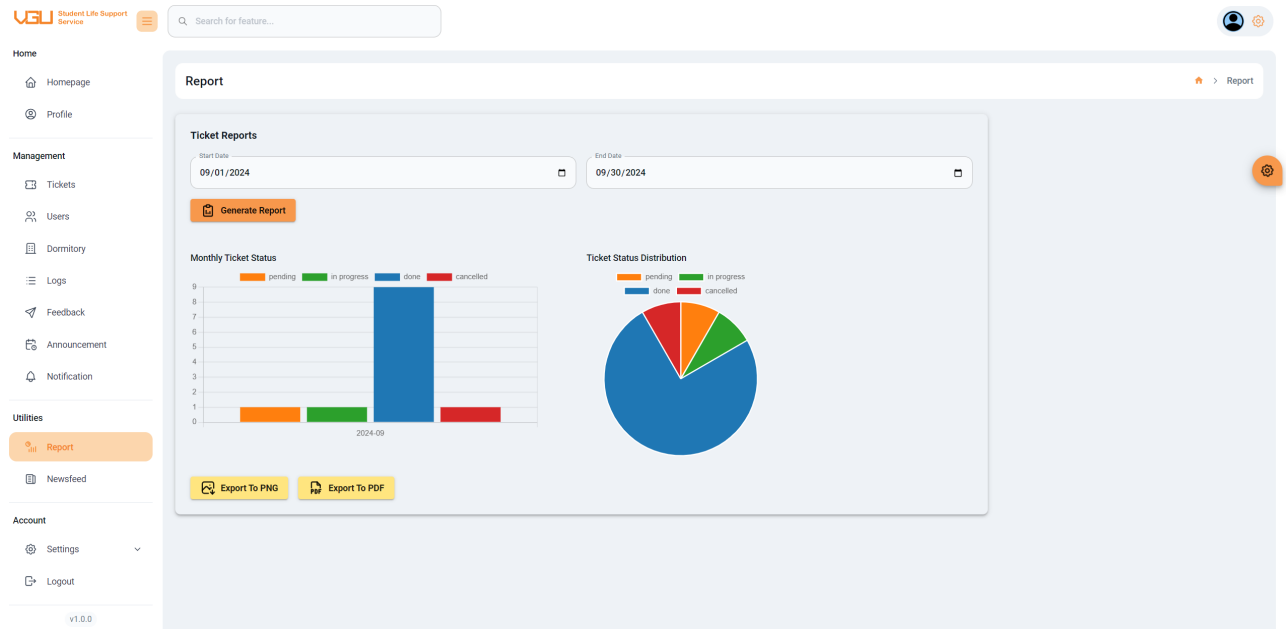

Figure 56: Admin's Report Page

Administrators have the capability to generate detailed ticket reports for a specified time period, allowing them to analyze ticket activity and performance metrics during that interval. Once the report is created, administrators can easily export it in various formats, such as PNG or PDF. This functionality not only facilitates efficient record-keeping but also enables administrators to share insights and data with stakeholders, ensuring that they are well-informed about ticket trends, resolutions, and user interactions within the system. By having access to these reports, administrators can make data-driven decisions to improve service delivery and address any issues effectively.

6 Conclusion and Future Work

References

[1] Aryan Patil. Three-tier architecture, 2023. URL: https://hyperskill.org/learn/step/25083.

[2] Sanity. React.js overview, August 2023. URL: https://www.sanity.io/glossary/react-js.

[3] Eric Simons. What is Vite (and why is it so popular)?, September 2024. URL: https://blog.stackblitz.com/posts/what-is-vite-introduction/.

[4] Alesia Sirotka. What is Material UI?, March 2022. URL: https://flatlogic.com/blog/what-is-material-ui/.

[5] Socket.IO. Introduction, July 2024. URL: https://socket.io/docs/v4/.