



**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.

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CSE Computer Science and Engineering

JSON JavaScript Object Notation

JSONP JSON with Padding

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	21
2	Material UI Logo	22
3	Vite Logo	22
4	NodeJS Logo	23
5	Expressjs Logo	23
6	SocketIO Logo	24
7	JWT Logo	24
8	Detailed explanation of JWT-based authentication mechanism.	24
9	Redis Logo	25
10	PostgreSQL RDBMS Logo	25
11	Student Use Case Diagram	34
12	Staff (Dormitory staff, Student affairs) Use Case Diagram	35
13	Admin Use Case Diagram	36
14	Ticket-Raising Process Workflow	37
15	ER Diagram	38
16	Three-tier Architecture ^[7]	48
17	Get Messages API - HTTP Response with error status code 401	51
18	Get Messages API - HTTP Response with success status code 200	51
19	Profile UI in Desktop device	69
20	Profile UI in Mobile device	70
21	Web UI Live Customization	71
22	Web UI Main Components	73
23	JWT Authentication/Authorization Workflow	83
24	reCAPTCHA in Change Password form	86
25	Landing Page	87
26	Sign in Form	88
27	Failed Sign in attempt	88
28	Reset Password Form	89
29	Reset password successfully	89
30	Reset password failed	89
31	Reset password email instructions	90

32	Student's Home Page	91
33	Student's Profile Page	91
34	Student's Tickets List Page	92
35	Student's Create Tickets Page	93
36	Student's Rate Tickets Page	94
37	Submit a ticket rating	94
38	Student's Message Page	95
39	Student's Newsfeed	96
40	Student's Notification Page	97
41	Student's Announcement Page	98
42	Student's Edit Profile Page	99
43	Student's Change Password Page	100
44	Student's Feedback Page	101
45	Staff's Available Tickets Page	102
46	Handle a ticket	103
47	Successfully add a ticket to ticket handling list	103
48	Staff's Tickets Handling List	104
49	Mark a ticket as done	104
50	Cancel a ticket	105
51	Staff's Tickets History Page	106
52	Staff's Message Page	107
53	Create a Notification Modal	108
54	Create an Announcement Modal	109
55	Admin's Ticket Management Page	110
56	Admin's User Management Page	111
57	Admin's Dormitory Management Page	112
58	Admin's Logs Management Page	113
59	Admin's Feedback Management Page	114
60	Admin's Report Page	114

List of Tables

1	System key features	16
2	System key components	17
3	Existing University Ticketing Systems	20
4	Functional Requirements	30
5	Functional Requirements by User Roles	32
6	Non-Functional Requirements	34
7	User Entity	39
8	Ticket Entity	40
9	User_Ticket Relationship	40
10	Ticket_Type Entity	41
11	Ticket_Status Entity	41
12	Audience_Type Entity	42
13	Attachment Entity	42
14	Rating Entity	43
15	Feedback Entity	44
16	Message Entity	44
17	Dorm Entity	45
18	Announcement Entity	45
19	Notification Entity	46
20	Role Entity	46
21	Notification_Audience Entity	47
22	Log Entity	47
23	Event_Type Entity	48
24	3-Tier Architecture Implementation	49
25	Authentication/Authorization API	52
26	User API	55
27	Ticket API	58
28	Dormitory API	60
29	Attachment API	61
30	Rating API	61
31	Message API	62

32	Role API	63
33	Notification API	64
34	Announcement API	65
35	Feedback API	66
36	Log API	67
37	Report API	68
38	Frontend source code structure	72
39	Backend source code structure	78

List of Code Snippets

1	Example of a React component	21
2	Request body of Login API	53
3	Response of Login API	53
4	Response of Refresh Token API	53
5	Response of Verify Refresh Token API	53
6	Response of Reset Password API	54
7	User Schema	55
8	Ticket Schema	59
9	Dormitory Schema	60
10	Rating Schema	62
11	Message Schema	63
12	Role Schema	63
13	Notification Schema	64
14	Notification Schema	65
15	Feedback Schema	66
16	Create a Feedback	67
17	Log Schema	67
18	Frontend Header Component	73
19	Frontend Sidebar Component	75
20	Example of Lazy Loading Frontend Components	76
21	MUI Grid System	77
22	MUI Media Queries	77
23	ExpressJS Server Setup	78
24	Server connects to PostgreSQL Database	79
25	Server connects to Redis	80
26	Logger middleware - write log to database	80
27	Create Socket.io HTTP server	81
28	Socket.io join_conversation event	81
29	Socket.io send_message event	82
30	Socket.io disconnect event	82
31	Store refresh token in secure cookie	84

32	Password Hashing	85
33	Password Verification	85

Contents

1	Introduction	15
1.1	Project Background	15
1.2	Problem Statement	16
1.3	Objectives of the Project	16
1.4	Scope of the Project	17
1.5	Thesis Structure	18
2	Literature Review	19
2.1	Existing solutions	19
2.1.1	Group Chat-Based Systems (Current Solution at VGU)	19
2.1.2	Existing University and Open-source Ticketing Systems	19
2.1.3	Limitations of Existing Solutions in the University Context	20
2.2	Technology Review	21
2.2.1	Frontend: ReactJS, Material UI, Vite	21
2.2.2	Backend: NodeJS, ExpressJS, SocketIO	23
2.2.3	Authentication: JWT, Redis	24
2.2.4	Database: PostgreSQL	25
2.2.5	Responsive Web Design: Techniques and Tools	26
2.3	Theoretical Background	26
2.3.1	Ticket Management Systems	26
2.3.2	Real-Time Communication Tools	27
2.3.3	Web Application Development Best Practices	27
2.4	Gap Analysis	27
2.4.1	What is Missing from Existing Solutions	27
2.4.2	How the Student Life Support Service Fills These Gaps	28
3	System Design	29
3.1	Functional Requirements	29
3.2	Non-Functional Requirements	32
3.3	Use Case Diagrams	34
3.4	Process Workflow Diagrams	37
3.5	Database Design	38

3.5.1	ER Diagram	38
3.5.2	User Entity	38
3.5.3	Ticket Entity	39
3.5.4	User_Ticket Relationship	40
3.5.5	Ticket_Type Entity	41
3.5.6	Ticket_Status Entity	41
3.5.7	Audience_Type Entity	42
3.5.8	Attachment Entity	42
3.5.9	Rating Entity	43
3.5.10	Feedback Entity	43
3.5.11	Message Entity	44
3.5.12	Dorm Entity	44
3.5.13	Announcement Entity	45
3.5.14	Notification Entity	46
3.5.15	Role Entity	46
3.5.16	Notification_Audience Relationship	47
3.5.17	Log Entity	47
3.5.18	Event_Type Entity	48
3.6	System Architecture	48
3.6.1	Overview	48
3.6.2	3-Tier Architecture Implementation	49
3.7	API Design	50
3.7.1	Authentication/Authorization API	52
3.7.2	User API	54
3.7.3	Ticket API	56
3.7.4	Dormitory API	59
3.7.5	Attachment API	61
3.7.6	Rating API	61
3.7.7	Message API	62
3.7.8	Role API	63
3.7.9	Notification API	63
3.7.10	Announcement API	64
3.7.11	Feedback API	65

3.7.12 Log API	67
3.7.13 Report API	68
3.8 UI/UX Design	68
3.8.1 Responsive Design	69
3.8.2 Live Customization	70
4 System Implementation	72
4.1 Frontend Implementation	72
4.1.1 Source code structure	72
4.1.2 UI Main Components	73
4.1.3 Responsive Design Implementation	76
4.2 Backend Implementation	78
4.2.1 Source code structure	78
4.2.2 Setting up ExpressJs Server	78
4.2.3 Database Integration	79
4.2.4 Middleware	80
4.2.5 Real-time messages	81
4.3 Security Measures	83
4.3.1 JWT Authentication/Authorization	83
4.3.2 Password Hashing	85
4.3.3 Google reCAPTCHA	86
5 User Manual	87
5.1 Sign in	87
5.2 Forgot password	88
5.3 Student's functions	90
5.3.1 View Profile	91
5.3.2 View Tickets	92
5.3.3 Create Tickets	92
5.3.4 Rate Tickets	93
5.3.5 Message	95
5.3.6 Newsfeed	96
5.3.7 Notification	97

5.3.8	Announcement	98
5.3.9	Settings	99
5.3.10	Feedback	100
5.4	Staff's functions	101
5.4.1	Available Tickets	101
5.4.2	Tickets Handling	103
5.4.3	Tickets History	105
5.4.4	Message	106
5.4.5	Notification	107
5.4.6	Announcement	108
5.5	Admin's functions	109
5.5.1	Tickets Management	109
5.5.2	Users Management	110
5.5.3	Dormitory Management	111
5.5.4	Logs Management	112
5.5.5	Feedback Management	113
5.5.6	Report	114
6	Conclusion and Future Work	116
6.1	Conclusion	116
6.2	Future Work	117

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 integrated.
- **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.

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

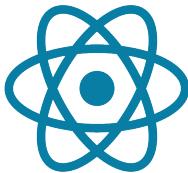


Figure 1: ReactJS Logo

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^[10]. 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.

```
1  const Profile = () => {
2
3      return (
4          <MainCard title="Personal Information">
5              <Grid container spacing={gridSpacing}>
6
7                  <Grid item xs={12} sm={6}>
8                      <ProfileCard />
9                  </Grid>
10
11                  <Grid item xs={12} sm={6}>
12                      <SchoolDetailsCard/>
13                  </Grid>
14
15          </Grid>
16      </MainCard>
17  );
18
19
20  export default Profile;
21
```

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.^[12] Material UI is a great option for creating responsive web applications with a sleek and contemporary design. It offers a vast selection of components that allow developers to efficiently build intricate layouts and user interfaces. Additionally, its well-organized and comprehensive API documentation makes it easy to explore each component and customize it to meet specific requirements. Whether you're developing a basic website or a more complex web application, Material UI helps you meet both design and functionality objectives while providing a seamless and responsive user experience.^[15]



Figure 2: Material UI Logo



Figure 3: Vite Logo

Vite is an innovative frontend build tool designed to enhance development speed and efficiency compared to traditional tools like Webpack. One of Vite's standout features is its instant hot module replacement (HMR), which significantly improves the developer experience by allowing changes to be reflected in the browser almost instantaneously without a full page reload. This feature is particularly beneficial during iterative development cycles, as it enables developers to see the effects of their code modifications in real-time, thereby accelerating the debugging process and fostering a more dynamic workflow.^[11] Furthermore, Vite leverages native ES modules in the browser, allowing for a more optimized development environment. Unlike older bundlers that require extensive preprocessing, Vite serves source files directly to the browser during development, resulting in quicker start-up times and faster builds. This approach not only enhances productivity but also simplifies the development setup, making it accessible for developers of all skill levels.^[5]

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.^[14] Node.js is also lightweight and efficient, allowing developers to use JavaScript on both the frontend and backend, which enhances flexibility and cross-functionality within teams, ultimately reducing development costs. This code reusability accelerates the development process, and since JavaScript is the most popular programming language, the single-thread event loop it employs makes asynchronous programming more manageable. Consequently, new engineers will find it easier to understand the application's codebase.^[9]

ExpressJS is a minimalist web framework for NodeJS. Express simplifies routing, middleware management, and API handling. Over the years, ExpressJS has demonstrated significant scalability, evidenced by its widespread use among major companies operating it on their servers daily. It efficiently manages user requests and responses, needing minimal configuration for large-scale web application development. With its robust modules, packages, and resources, it supports developers in building reliable and scalable web applications.^[6] In this application, ExpressJS 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.^[13] In addition, Socket.IO offers various fallback options, such as long polling, JSONP polling, and iframe-based transport. This allows for continued communication in environments where WebSocket is unavailable.^[1] SocketIO is used to implement features such as real-time messaging between students and staff, making the system more interactive and responsive.

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.^[3]

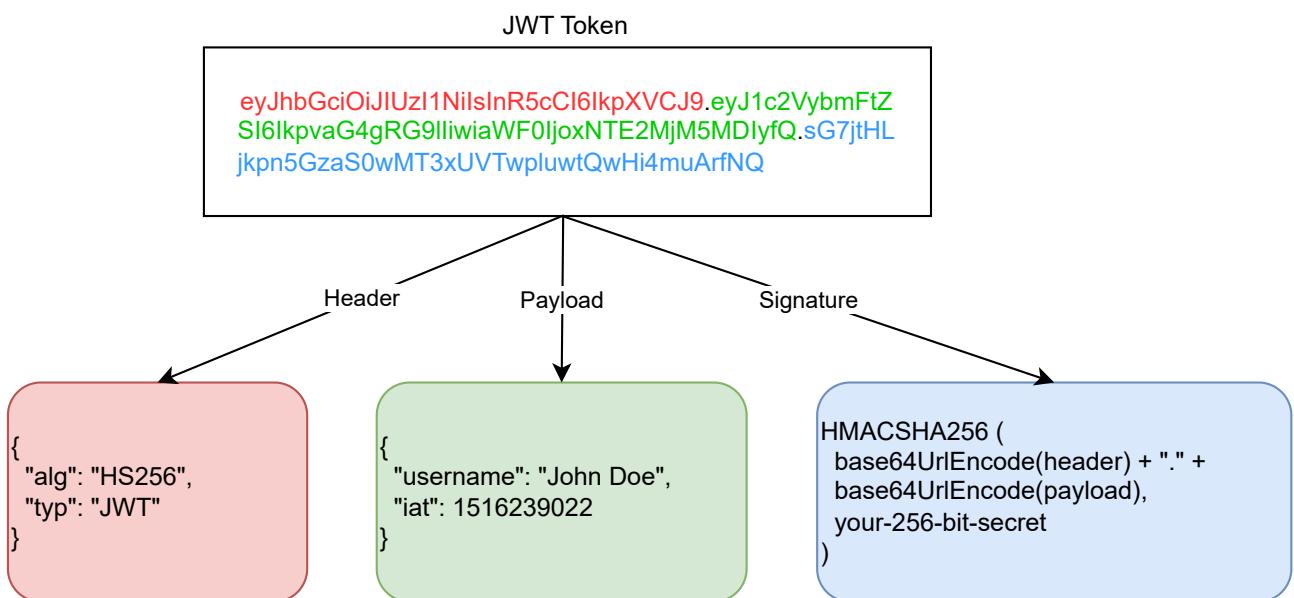


Figure 8: Detailed explanation of JWT-based authentication mechanism.



Figure 9: Redis Logo

Redis is an in-memory data structure store, commonly used as a database, cache, and message broker. It is part of the NoSQL database category called key/value stores. Keys serve as unique identifiers, and their corresponding values can be any of the data types supported by Redis. These data types include basic Strings, Linked Lists, Sets, and Streams, each with its own specific behaviors and commands.^[8] In this system, Redis is specifically utilized for session management, particularly in storing refresh tokens. By caching these tokens, Redis helps to reduce the load on the primary database, which improves both scalability and performance. Since Redis operates in memory, it allows for faster retrieval of session data, ensuring that the authentication process remains responsive and efficient. Additionally, Redis offers built-in features like automatic expiration, which helps in managing token lifetimes and ensuring secure token invalidation, further enhancing the security and reliability of the system.

2.2.4 Database: PostgreSQL

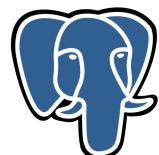


Figure 10: PostgreSQL RDBMS Logo

PostgreSQL is a powerful, open-source relational database that provides strong ACID (Atomicity, Consistency, Isolation, Durability) compliance, ensuring reliability and data integrity, which is critical for managing sensitive information such as user accounts, ticketing systems, and communication logs. Its robust support for advanced querying and indexing mechanisms ensures that the system can handle complex searches and queries with high efficiency,

even as data grows over time.^[4]

PostgreSQL also supports features like foreign key constraints, triggers, and stored procedures, which help in maintaining data consistency across multiple tables, ensuring that relationships between different data entities (such as users and tickets) are enforced and managed correctly. Additionally, its ability to support JSON and JSONB data types allows the system to store and query semi-structured data, offering flexibility in handling modern web applications that require both structured and unstructured data.

The database also scales well, supporting a large number of concurrent users and high-volume transactions, making it an ideal choice for applications with growing user bases. PostgreSQL's support for full-text search and geospatial data (via PostGIS) can be useful for implementing advanced search functionalities and geographical features in the system. With its extensibility (allowing the addition of custom functions, data types, and more), PostgreSQL provides a solid and scalable foundation for the back-end data management of the system.

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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)	<ul style="list-style-type: none">• 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

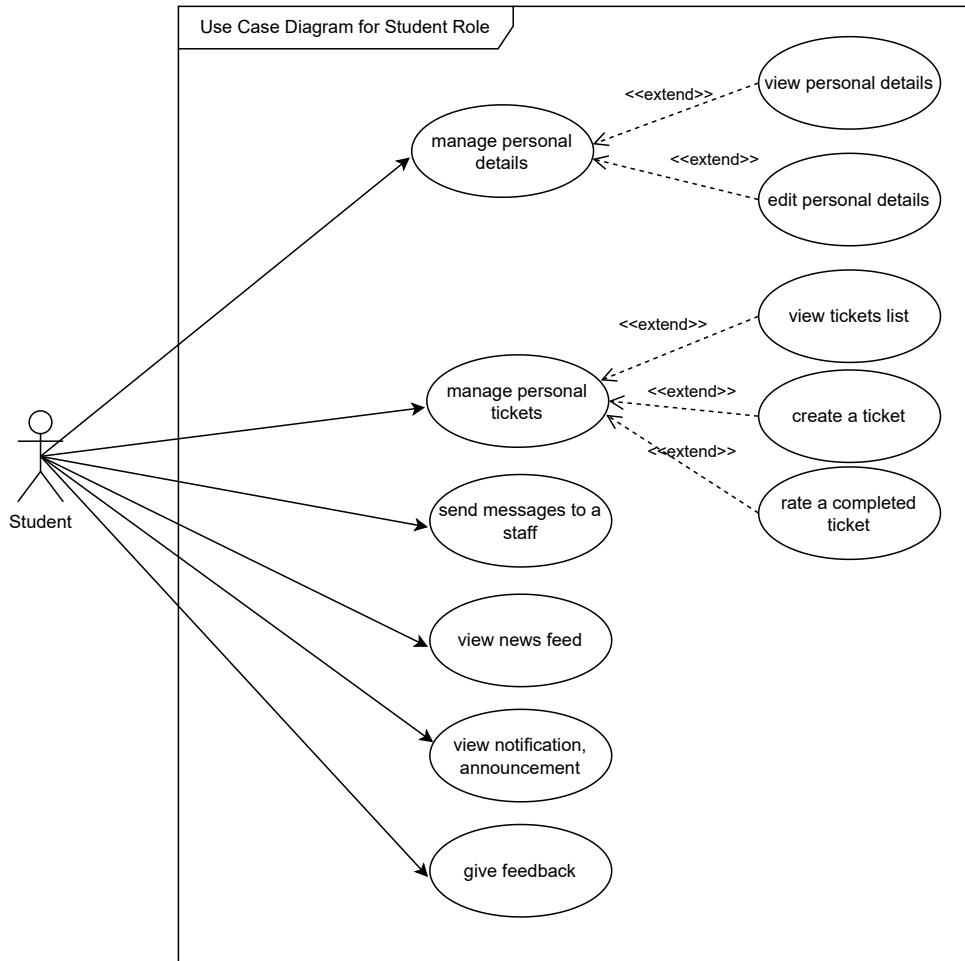


Figure 11: Student Use Case Diagram

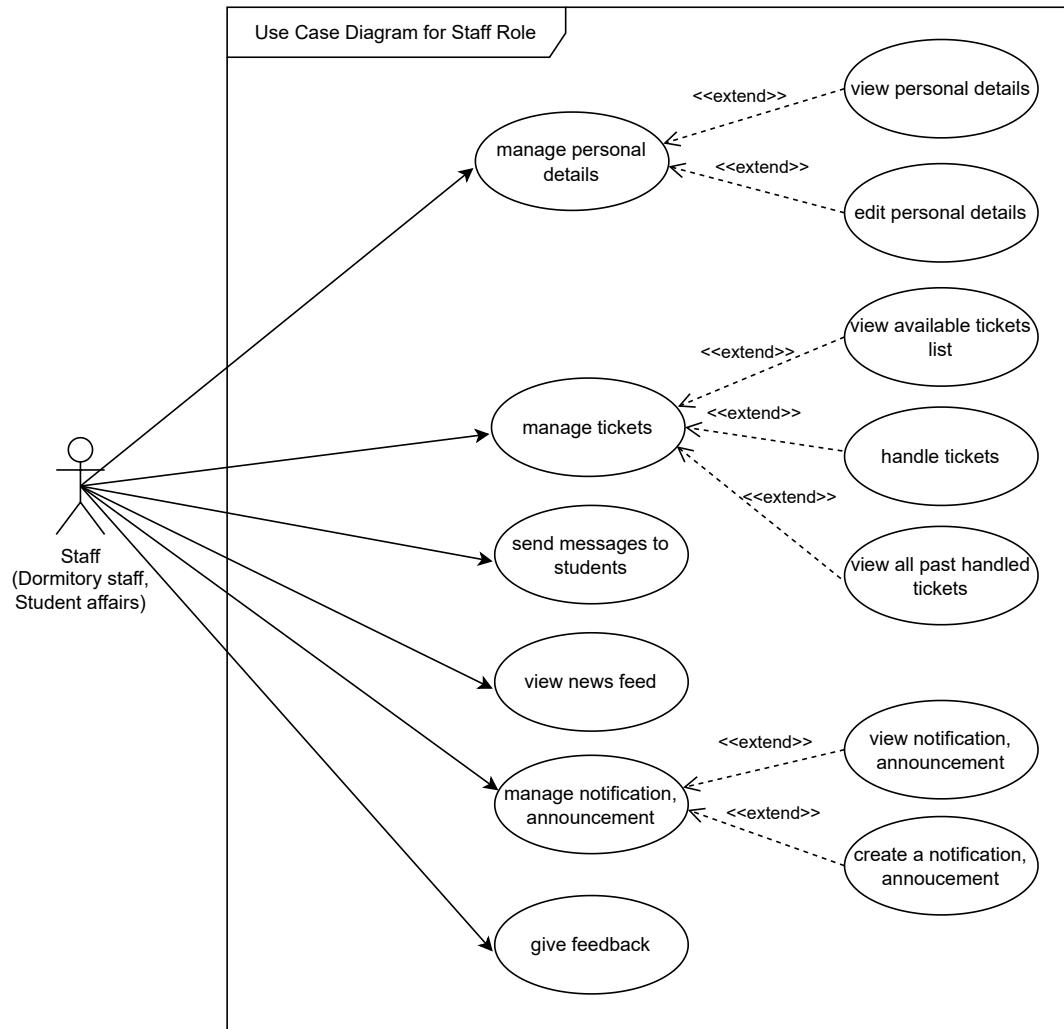


Figure 12: Staff (Dormitory staff, Student affairs) Use Case Diagram



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.

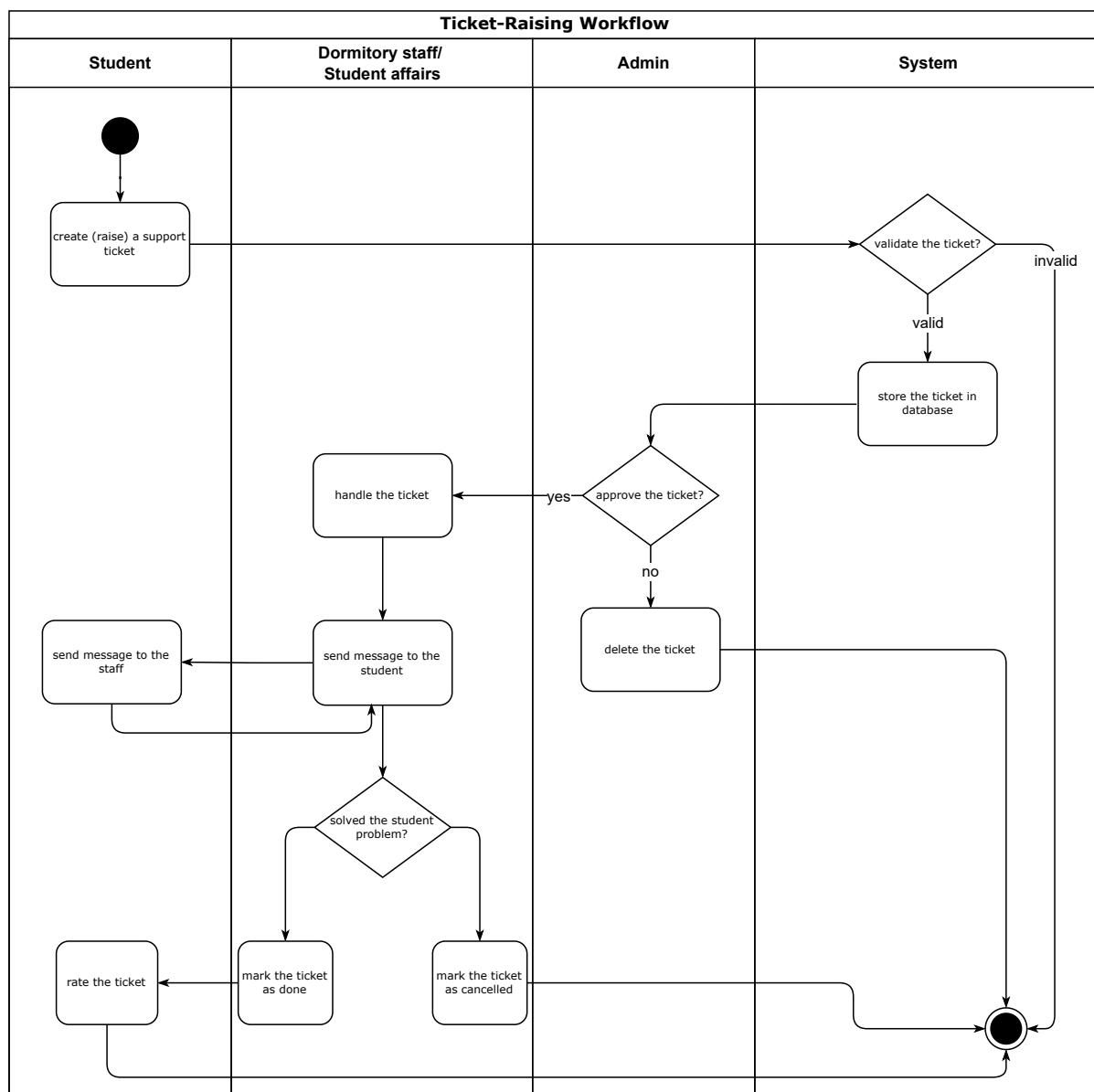


Figure 14: Ticket-Raising Process Workflow

3.5 Database Design

3.5.1 ER Diagram

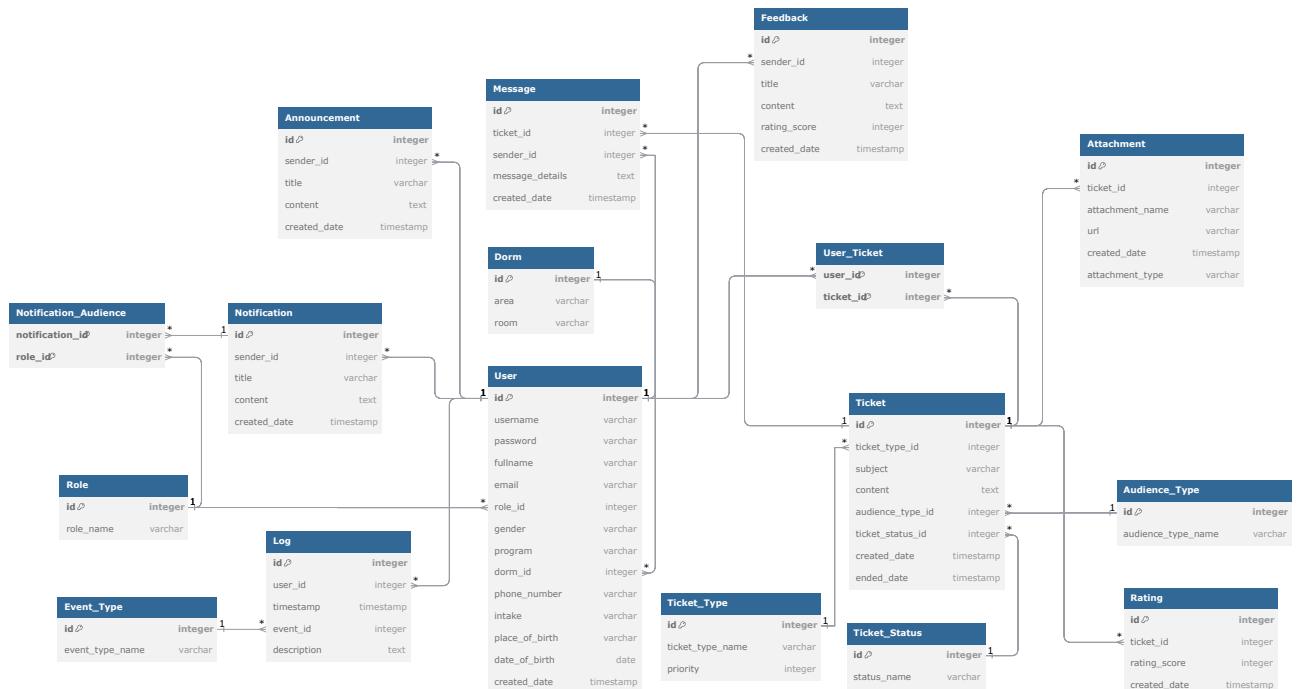


Figure 15: ER Diagram

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:

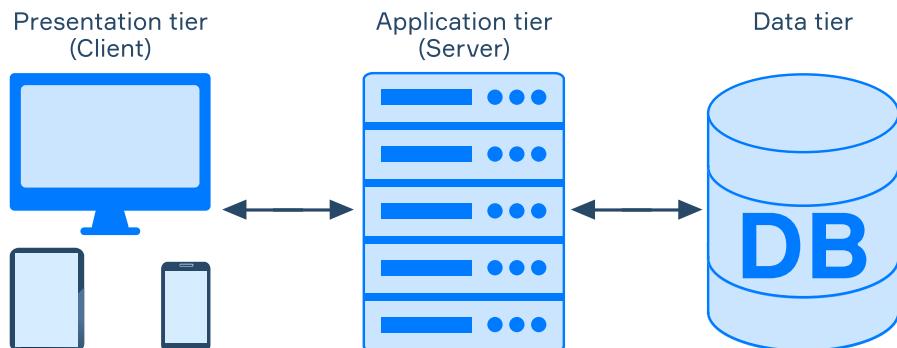


Figure 16: Three-tier Architecture^[7]

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, 401 for unauthorized, 403 for forbidden, 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>).

(see Figures 17, 18)

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** <http://localhost:3000/api/v1/messages/3>
- Auth Type:** Bearer Token
- Body:** (Empty)
- Headers:** (8 items)
- Response Status:** 401 Unauthorized
- Response Body:**

```

1  {
2    "message": "Cannot authorize the user"
3  }

```

Figure 17: Get Messages API - HTTP Response with error status code 401

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** <http://localhost:3000/api/v1/messages/71>
- Auth Type:** Bearer Token
- Body:** (Empty)
- Headers:** (10 items)
- Response Status:** 200 OK
- Response Body:**

```

1  [
2    {
3      "id": 404,
4      "ticket_id": 71,
5      "sender_id": 4,
6      "sender_fullName": "Vũ Hoàng Tuấn Anh",
7      "message_details": "I have just created this ticket",
8      "created_date": "2024-09-30T04:35:50.580Z"
9    },
10   {
11     "id": 405,
12     "ticket_id": 71,
13     "sender_id": 13,
14     "sender_fullName": "Nguyễn Nguyễn Vũ",
15     "message_details": "I have just been assigned to this ticket",
16     "created_date": "2024-09-30T05:15:58.825Z"
17   }
18 ]

```

Figure 18: Get Messages API - HTTP Response with success status code 200

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, re- fresh token and general user in- formation
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 pass- word 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

```
1 {
2   "username": "string",
3   "password": "string"
4 }
```

Code snippet 2: Request body of Login API

```
1 {
2   "user_id": number,
3   "username": "string",
4   "email": "string",
5   "fullname": "string",
6   "role_name": "string",
7   "accessToken": "string"
8 }
```

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

```
1 {
2   "accessToken": "string"          // JWT access token for authentication
3 }
```

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

```
1 {
2   "valid": boolean
3 }
```

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

```
1 {
2   "message": "string"
3 }
```

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.

```
1  {
2    "username": "string",
3    "fullname": "string",
4    "email": "string",
5    "role_name": "string",
6    "gender": "string",
7    "program": "string",
8    "area": "string",
9    "room": "string",
10   "phone_number": "string",
11   "intake": "string",
12   "place_of_birth": "string",
13   "date_of_birth": "string",
14   "created_date": "string"
15 }
```

Code snippet 7: User Schema

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 sup- plied 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.

```
1 {
2   "ticket_id": number,
3   "username": "string",
4   "fullname": "string",
5   "ticket_type_name": "string",
6   "subject": "string",
7   "details": "string",
8   "audience_type": "string",
9   "status": "string",
10  "dorm_area": "string",
11  "dorm_room": "string",
12  "created_date": "string",
13  "ended_date": "string",
14  "attachments": [
15    {
16      "id": number,
17      "type": "string",
18      "name": "string",
19      "url": "string"
20    },
21  ]
22 }
```

Code snippet 8: Ticket Schema

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

```

1 [ 
2 { 
3   "dorm_area": "string",
4   "rooms": [ { "dorm_room": "string"}, ... ]
5 }, ...
6 ]

```

Code snippet 9: Dormitory Schema

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

The Rating API allows users to retrieve and submit ratings for tickets. A GET request fetches the rating score for a specific ticket by ticket ID, while a POST request allows users to submit a rating for a ticket by providing the ticket ID and rating score. Both operations require user authentication via a Bearer token.

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

```
1  {
2    "ticket_id": number,
3    "rating_score": number,
4    "created_date": "string"
5 }
```

Code snippet 10: Rating Schema

3.7.7 Message API

The Message API allows users to manage and send messages related to tickets. Additionally, real-time messaging is supported via a WebSocket event, where users can send messages along with ticket and sender details. All actions require authentication using a Bearer token.

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
WebSocket Event	send_message	Authorization: Bearer <token>	JSON object which contains ticket id, sender details and message	JOSN object which contains ticket id, sender details, message and created date

Table 31: Message API

```
1  {
2    "ticket_id": number,
3    "sender_id": number,
4    "sender.FullName": "string",
5    "message_details": "string",
6    "created_date": "string"
7 }
```

Code snippet 11: Message Schema

3.7.8 Role API

The Role API provides a method for retrieving all user roles within the system.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/roles	Authorization: Bearer <token>	None	JSON object which contains all roles name

Table 32: Role API

```
1  {
2    "role_id": number,
3    "role_name": "string"
4 }
```

Code snippet 12: Role Schema

3.7.9 Notification API

The Notification API allows users to retrieve and create notifications. The GET request returns a list of all notifications, requiring an Authorization Bearer token. The POST request, also requiring a token, allows users to create a new notification by supplying necessary details in the request body, such as the sender ID, recipient roles, title, and content of the notification.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/notification	Authorization: Bearer <token>	None	List of JSON objects which contain all notification
POST	/api/v1/notification	Authorization: Bearer <token>	JSON object contains needed information to create a notification	Create a new notification with supplied data

Table 33: Notification API

```
1 {
2   "sender_id": number,
3   "recipients": [ { "role_id": number } ],
4   "title": "string",
5   "content": "string",
6   "created_date": "string"
7 }
```

Code snippet 13: Notification Schema

3.7.10 Announcement API

The Announcement API provides functionality to retrieve and create announcements. Users can fetch a list of all announcements, requiring authorization. Additionally, authorized users can create new announcements by submitting relevant details, including sender ID, title, content, and the creation date.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/announcement	Authorization: Bearer <token>	None	List of JSON objects which contain all announcements
POST	/api/v1/announcement	Authorization: Bearer <token>	JSON object contains needed information to create an announcement	Create a new announcement with supplied data

Table 34: Announcement API

```
1  {
2      "sender_id": number,
3      "title": "string",
4      "content": "string",
5      "created_date": "string"
6  }
```

Code snippet 14: Notification Schema

3.7.11 Feedback API

The Feedback API allows users to manage feedback within the system. Users can retrieve a list of all feedback or details of specific feedback items by their ID. Authorized users can also create new feedback by submitting relevant information, including the sender's ID, title, content, and rating score. Additionally, the API supports the deletion of specific feedback based on its ID.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/feedback	Authorization: Bearer <token>	None	List of JSON objects which contain all feedback
GET	/api/v1/feedback/{feedback_id}	Authorization: Bearer <token>	None	JSON object which contains feedback details with supplied feedback id
POST	/api/v1/feedback/	Authorization: Bearer <token>	JSON object which contains needed information to create a feedback	Create a new feedback with supplied data
DELETE	/api/v1/feedback/{feedback_id}	Authorization: Bearer <token>	None	Delete a feedback

Table 35: Feedback API

GET /api/v1/feedback{feedback_id}

```

1  {
2    "feedback_id": number,
3    "title": "string",
4    "content": "string",
5    "rating_score": number,
6    "created_date": "string"
7 }
```

Code snippet 15: Feedback Schema

POST /api/v1/feedback

```
1 {
2     "sender_id": number,
3     "title": "string",
4     "content": "string",
5     "rating_score": number,
6     "created_date": "string"
7 }
```

Code snippet 16: Create a Feedback

3.7.12 Log API

The Log API allows authorized users to manage system logs. It provides an endpoint to retrieve a list of all logs, each containing details such as log ID, event type, timestamp, user information, and a description of the event. The API also supports deleting a specific log by its ID or clearing all logs from the system.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/logs	Authorization: Bearer <token>	None	List of JSON objects which contain all system logs
DELETE	/api/v1/logs/{log_id}	Authorization: Bearer <token>	None	Delete a specific log
DELETE	/api/v1/logs	Authorization: Bearer <token>	None	Delete all system logs

Table 36: Log API

```
1 {
2     "log_id": number,
3     "event_type_name": "string",
4     "timestamp": "string",
5     "description": "string",
```

```
6  "user_id": number,  
7  "username": string,  
8  "fullname": "string",  
9  "role_name": "string"  
10 }
```

Code snippet 17: Log Schema

3.7.13 Report API

The Report API provides a method to retrieve report data on tickets within a specified date range. Authorized users can get a list of JSON objects containing ticket report details by providing the start and end dates in the query parameters.

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/report/reports/ tickets?start_date &end_date	Authorization: Bearer <token>	None	List of JSON objects which contain ticket report data

Table 37: Report API

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 19, 20)

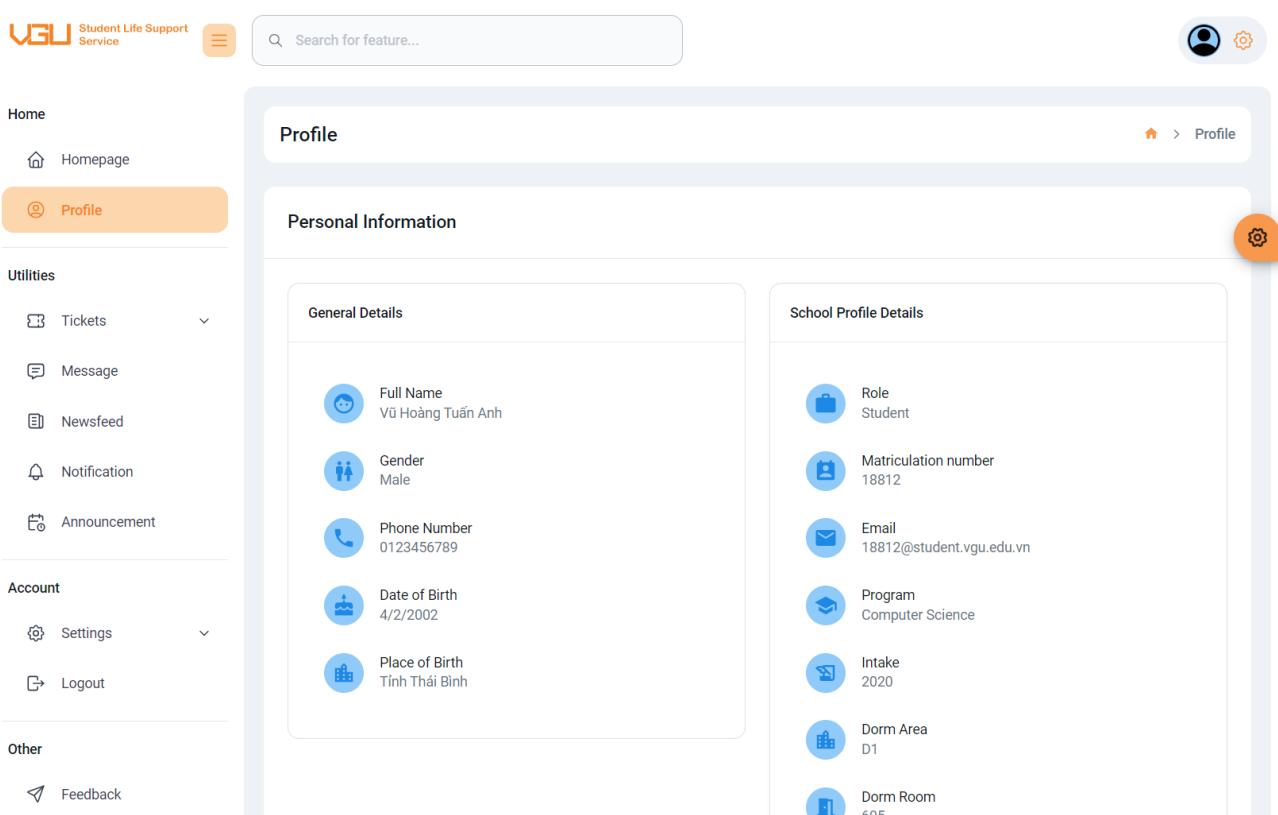


Figure 19: Profile UI in Desktop device

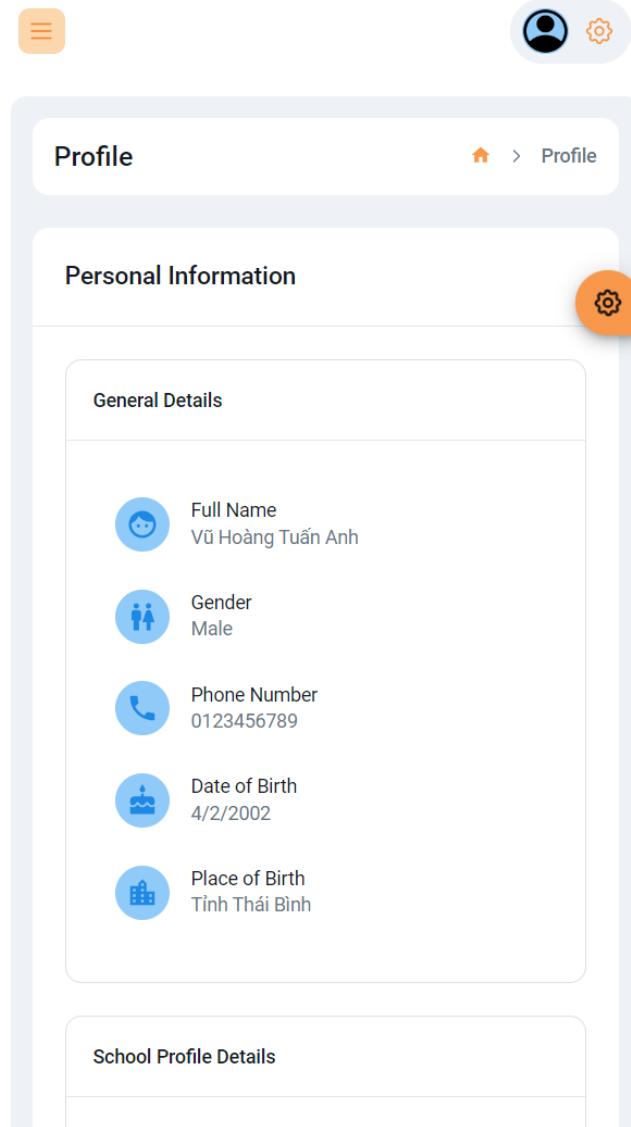


Figure 20: Profile UI in Mobile device

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.

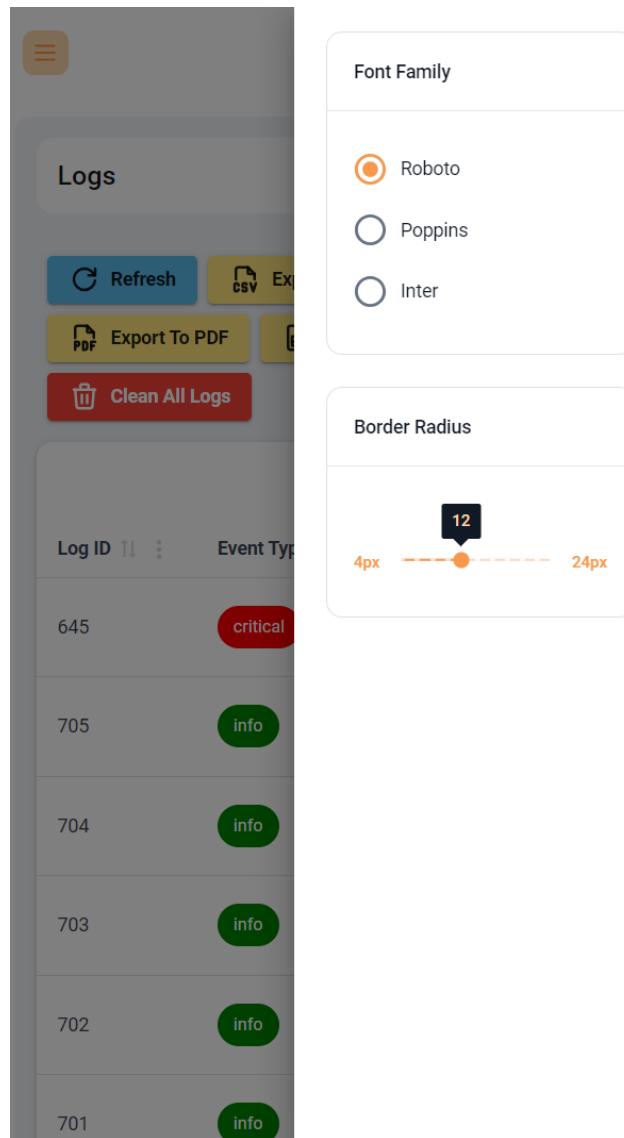


Figure 21: Web UI Live Customization

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 38: Frontend source code structure

4.1.2 UI Main Components



Figure 22: Web UI Main Components

There are 3 main components in the Web UI (marked on Figure 22):

1. **Header** (marked as Red color on Figure 22) contains Logo image, Menu toggle button, Search bar, and Profile section across all pages

```
1 const Header = ({ handleLeftDrawerToggle }) => {
2   const theme = useTheme();
3
4   return (
5     <>
6     {/* logo & toggler button */}
7     <Box
8       sx={{
9         width: 228,
10        display: 'flex',
11        [theme.breakpoints.down('md')]: {
12          width: 'auto'
```

```
13     }
14 }
15 >
16 <Box component="span" sx={{ display: { xs: 'none', md: 'block' },
17   flexGrow: 1 }}>
18 <LogoSection />
19 </Box>
20 <ButtonBase sx={{ borderRadius: '8px', overflow: 'hidden' }}>
21 <Avatar
22   variant="rounded"
23   sx={{
24     ...theme.typography.commonAvatar,
25     ...theme.typography.mediumAvatar,
26     transition: 'all .2s ease-in-out',
27     background: theme.palette.secondary.light,
28     color: theme.palette.secondary.dark,
29     '&:hover': {
30       background: theme.palette.secondary.dark,
31       color: theme.palette.secondary.light
32     }
33   }}
34   onClick={handleLeftDrawerToggle}
35   color="inherit"
36 >
37 <IconMenu2 stroke={1.5} size="1.3rem" />
38 </Avatar>
39 </ButtonBase>
40 </Box>
41 /* header search */
42 <SearchSection />
43 <Box sx={{ flexGrow: 1 }} />
44 <Box sx={{ flexGrow: 1 }} />
45
46
47 /* <ProfileSection /> */
48 <ProfileSection />
49 </>
50 );
```

```
51 };
```

```
52
```

```
53 Header.propTypes = {
```

```
54   handleLeftDrawerToggle: PropTypes.func
```

```
55 };
```

```
56
```

```
57 export default Header;
```

Code snippet 18: Frontend Header Component

2. **Sidebar** (marked as Blue color on Figure 22): A dynamic navigation panel which renders all menu items of a specific roles.

```
1 const Sidebar = ({ drawerOpen, drawerToggle, window }) => {
```

```
2   const theme = useTheme();
```

```
3   const matchUpMd = useMediaQuery(theme.breakpoints.up('md'));
```

```
4
```

```
5   const drawer = (
```

```
6     <>
```

```
7     <Box sx={{ display: { xs: 'block', md: 'none' } }}>
```

```
8     <Box sx={{ display: 'flex', p: 2, mx: 'auto' }}>
```

```
9       <LogoSection />
```

```
10      </Box>
```

```
11    </Box>
```

```
12
```

```
13    <BrowserView>
```

```
14    <PerfectScrollbar
```

```
15      component="div"
```

```
16      style={{
```

```
17        // height: !matchUpMd ? 'calc(100vh - 56px)' : 'calc(100vh - 88px)
```

```
18        ,
```

```
19        height: !matchUpMd ? 'calc(100vh - 56px)' : 'calc(100vh - 88px)',
```

```
20        paddingLeft: '16px',
```

```
21        paddingRight: '16px'
```

```
22      }}
```

```
23    >
```

```
24    <MenuList />
```

```
25    {/* <MenuCard /> */}
```

```
26    <Stack direction="row" justifyContent="center" sx={{ mb: 2 }}>
```

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

Code snippet 19: Frontend Sidebar Component

3. **Content Components** (marked as Green color on Figure 22): 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 20: Example of Lazy Loading 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.

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

2

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

7
8   <Grid item xs={12} sm={6} md={4}>
9     <Card>Content 2</Card>
10    </Grid>

11
12   <Grid item xs={12} md={4}>
13     <Card>Content 3</Card>
14   </Grid>
15 </Grid>
```

Code snippet 21: 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.

```
1 <Box sx={{
2   display: { xs: 'block', sm: 'flex' },
3   flexDirection: { xs: 'column', sm: 'row' }
4 }}>
5 /* Our content */
6 </Box>
```

Code snippet 22: 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 39: 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.

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

```
5
6
7 // ======| | Routes | | ======
8 // 
9 app.use("/auth", authRoutes);
10 app.use("/api/v1/users", userRoutes);
11 app.use("/api/v1/dorms", dormRoutes)
12 app.use("/api/v1/tickets", ticketRoutes);
13 app.use("/api/v1/attachments", attachmentRoutes);
14 app.use("/api/v1/rating", ratingRoutes);
15 app.use("/api/v1/messages", messageRoutes);
16 app.use("/api/v1/roles", roleRoutes);
17 app.use("/api/v1/notification", notificationRoutes);
18 app.use("/api/v1/announcement", announcementRoutes);
19 app.use("/api/v1/feedback", feedbackRoutes);
20 app.use("/api/v1/logs", logsRoutes);
21 app.use("/api/v1/reports", reportRoutes);
```

Code snippet 23: ExpressJS Server Setup

4.2.3 Database Integration

```
1 import pkg from 'pg';
2 import dotenv from 'dotenv';
3
4 dotenv.config();
5
6 const { Pool } = pkg;
7
8 const pool = new Pool({
9   user: String(process.env.PG_DB_USER),
10  host: String(process.env.PG_DB_HOST),
11  password: String(process.env.PG_DB_PASSWORD),
12  port: Number(process.env.PG_DB_PORT),
13  database: String(process.env.PG_DB_DATABASE),
14 });
15
16 export default pool;
```

Code snippet 24: Server connects to PostgreSQL Database

```
1 import redis from 'redis';
2
3 const redisClient = redis.createClient({
4   socket: {
5     host: process.env.REDIS_HOST,
6     port: process.env.REDIS_PORT,
7   },
8 });
9
10
11 const connectRedis = async () => {
12   await redisClient.connect();
13   redisClient.on('error', (err) => {
14     console.error('Redis error:', err);
15   });
16   console.log('Redis connected successfully');
17 };
```

Code snippet 25: Server connects to Redis

4.2.4 Middleware

In the system, the `authenticateToken` middleware is designed to protect API endpoints by verifying JWT (JSON Web Tokens) based on user roles. The `authenticateToken` function accepts an array of `allowedRoles`, which specifies which user roles are permitted to access the route that this middleware protects. The middleware checks for a valid access token in the request headers and verifies it against predefined secrets corresponding to different user roles (e.g., student, staff, admin).

Additionally, `logger` - an asynchronous middleware designed to log events to a PostgreSQL database. This function serves as a logging mechanism, capturing important user actions or system events and storing them in a dedicated "Log" table.

```
1 const writeLogToDB = async (user_id, event_id, description, timestamp) => {
2   try {
3     await pool.query('BEGIN');
4     await pool.query(
5       'INSERT INTO "Log" (user_id, event_id, description, timestamp) VALUES (
6         $1, $2, $3, $4)',
```

```
6     [user_id, event_id, description, timestamp]
7 );
8     await pool.query('COMMIT');
9     logger.info('Log written to database');
10 } catch (error) {
11     await pool.query('ROLLBACK');
12     logger.error(error);
13 }
14 };
```

Code snippet 26: Logger middleware - write log to database

4.2.5 Real-time messages

By creating an HTTP server that integrates with the existing Express application. The Socket.IO server is attached to this HTTP server, allowing for real-time communication over web sockets.

```
1 const server = http.createServer(app);
2 const io = new Server(server, {
3   cors: {
4     origin: WebConfig.corsOptions.origin,
5     methods: ['GET', 'POST'],
6   },
7 });
```

Code snippet 27: Create Socket.io HTTP server

The backend sets up an event listener for incoming connections using the `io.on('connection')` method. This function is triggered whenever a new user connects to the server. When a user joins a conversation, they emit a `join_conversation` event along with the `ticket_id` of the conversation.

```
1 socket.on('join_conversation', (ticket_id) => {
2   socket.join(ticket_id);
3   console.log(`User joined conversation ${ticket_id}`);
4 });
```

Code snippet 28: Socket.io join_conversation event

When a user sends a message, they emit a `send_message` event with the message details. This event is processed to store the message in the database and then broadcast it to other users in the same conversation.

```
1 socket.on('send_message', async (data) => {
2   const { ticket_id, sender_id, message_details } = data;
3   const created_date = new Date();
4
5   try {
6     await pool.query('BEGIN'); // Start a transaction
7
8     // Insert messages data into database
9
10    await pool.query('COMMIT'); // Commit the transaction
11
12    io.to(ticket_id).emit('receive_message', newMessage); // Emit the new
13    message to the room
14  } catch (err) {
15    await pool.query('ROLLBACK'); // Rollback in case of error
16    console.error(err);
17  }
18});
```

Code snippet 29: Socket.io send_message event

Lastly, the disconnect event is implemented to clean up and log when a user disconnects from the conversation.

```
1 socket.on('disconnect', () => {
2   console.log('A user disconnected from the conversation');
3   // Handle clean up
4});
```

Code snippet 30: Socket.io disconnect event

4.3 Security Measures

4.3.1 JWT Authentication/Authorization

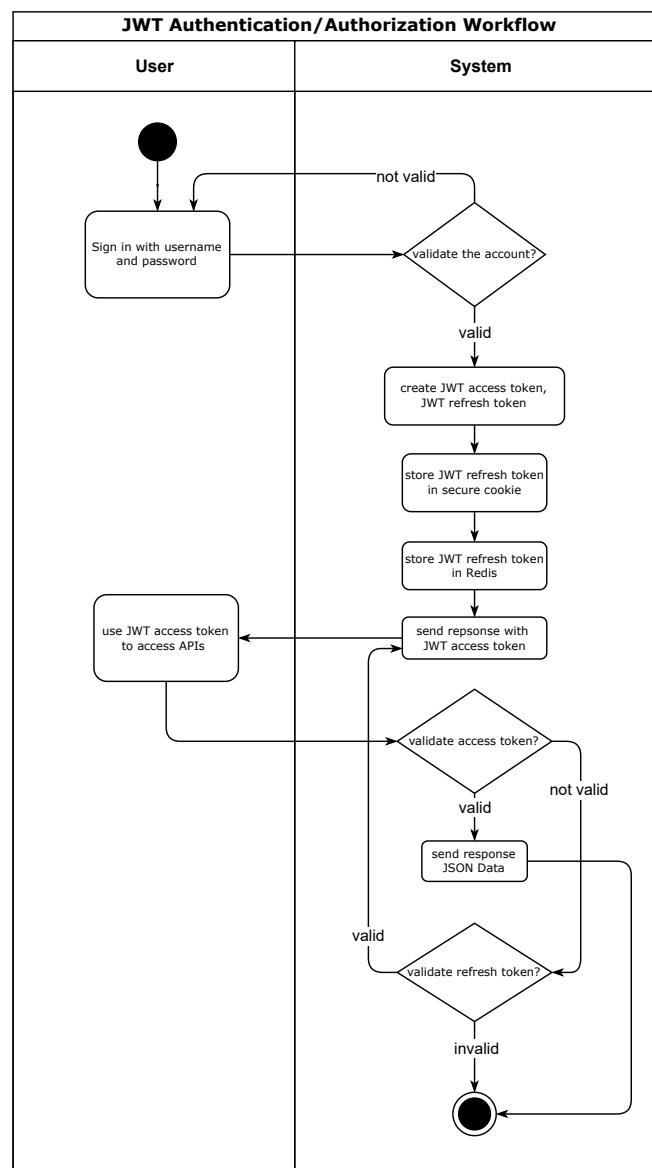


Figure 23: JWT Authentication/Authorization Workflow

The JWT (JSON Web Token) authentication and authorization workflow in the Student Life Support Service is designed to ensure secure access to the system's resources while providing a seamless user experience. The process begins when a user signs in by submitting their credentials, such as a username and password. Upon successful validation of these credentials, the system generates two tokens: an access token and a refresh token. The access token is used for short-term authorization and is typically included in the Authorization header of API requests to grant the user access to protected resources.

To ensure a high level of security, the refresh token is stored in two secure locations: a cookie on the client-side and in Redis, which serves as an in-memory database for managing sessions. The refresh token allows the system to issue a new access token when the original one expires, ensuring that the user does not need to log in repeatedly.

When a user makes an API request, the system first checks the validity of the provided access token. If the token is valid, the request is processed, and the appropriate data is returned. However, if the access token is expired or invalid, the system turns to the refresh token. If the refresh token is verified as valid, a new access token is generated, and the user can continue to access the API without re-authenticating.

In cases where both the access and refresh tokens are invalid, the user is required to log in again, ensuring that expired or tampered tokens do not grant unauthorized access to the system. By utilizing this access-refresh token structure, with refresh tokens stored securely in Redis and cookies, the system balances both security and usability, providing users with continuous access while protecting against unauthorized usage or token expiration.

This approach ensures that the system is scalable and can handle multiple user sessions efficiently, while also offering robust security through token validation and management.

```
1 // Send Refresh Token as an HTTP-only, Secure cookie
2 res.cookie('refreshToken', refreshToken, {
3   httpOnly: true,                      // Prevent JavaScript access to the cookie
4   secure: true,                        // Send only over HTTPS
5   sameSite: 'Strict',                  // Protect against CSRF
6   maxAge: REFRESH_TOKEN_EXPIRED_IN * 1000,
7 }) ;
```

Code snippet 31: Store refresh token in secure cookie

4.3.2 Password Hashing

All user passwords within the system are securely protected through the use of a one-way hashing algorithm. This is accomplished by employing the `bcryptjs` library, which ensures that passwords are irreversibly encrypted, thereby enhancing the security and integrity of user credentials.

The `hashPassword` function below takes a user's plaintext password as input and securely hashes it. First, the function generates a salt using `bcrypt.genSalt`, which adds an extra layer of randomness to the hashing process. This salt is combined with the user's password using the `bcrypt.hash` function to create a unique hashed version of the password. The hashed password is then returned, ensuring that even if the same password is used by multiple users, the stored hash will be different due to the unique salt for each password. In case of any errors during this process, the function throws a detailed error message for debugging purposes.

```
1 async function hashPassword(userInputPassword) {  
2   try {  
3     const salt = await bcrypt.genSalt(saltRounds);  
4     const hash = await bcrypt.hash(userInputPassword, salt);  
5     return hash; // Return the hashed password  
6   } catch (error) {  
7     throw new Error('Error hashing password: ' + error.message);  
8   }  
9 }
```

Code snippet 32: Password Hashing

The `verifyPassword` function is used to check if a user's inputted password matches the stored hashed password in the system. It does this by using `bcrypt.compare`, which compares the plaintext password with the hashed password stored in the database. If the passwords match, the function returns true; otherwise, it returns false. Similar to `hashPassword`, if an error occurs during the verification process, an error message is thrown for further investigation.

```
1 async function verifyPassword(userInputPassword, encryptedPassword) {  
2   try {  
3     const isMatch = await bcrypt.compare(userInputPassword,  
4       encryptedPassword);  
5     return isMatch;  
6   } catch (error) {  
7     throw new Error('Error verifying password: ' + error.message);  
8   }  
9 }
```

```
5 } catch (error) {  
6   throw new Error('Error verifying password: ' + error.message);  
7 }  
8 }
```

Code snippet 33: Password Verification

4.3.3 Google reCAPTCHA

Google reCAPTCHA is a security service designed to protect websites from bots and automated abuse. It uses a combination of behavioral analysis, machine learning, and user challenges to distinguish between human users and malicious bots. The main goal of reCAPTCHA is to prevent attacks such as credential stuffing, brute-force attempts, and other forms of automated exploits while allowing legitimate users to interact with the site without interruption. [2]

In this system, integrating reCAPTCHA in the "Edit Profile" and "Change Password" features enhances security by preventing automated attacks and brute-force attempts. It ensures that only legitimate users can make critical changes, reducing the risk of unauthorized access and data breaches. reCAPTCHA also helps protect sensitive user information, adds an extra layer of verification for critical actions, and prevents abuse or spam, maintaining system integrity and keeping accounts secure.

The screenshot shows a "Change Password" form. It has three input fields: "Enter your current password", "Enter new password", and "Enter new password again", each with an eye icon for password visibility. Below these is a reCAPTCHA challenge: "Verify that you are not a robot:" followed by a checkbox labeled "I'm not a robot" and the reCAPTCHA logo. At the bottom is a large grey "Change Password" button.

Figure 24: reCAPTCHA in Change Password form

5 User Manual

5.1 Sign in

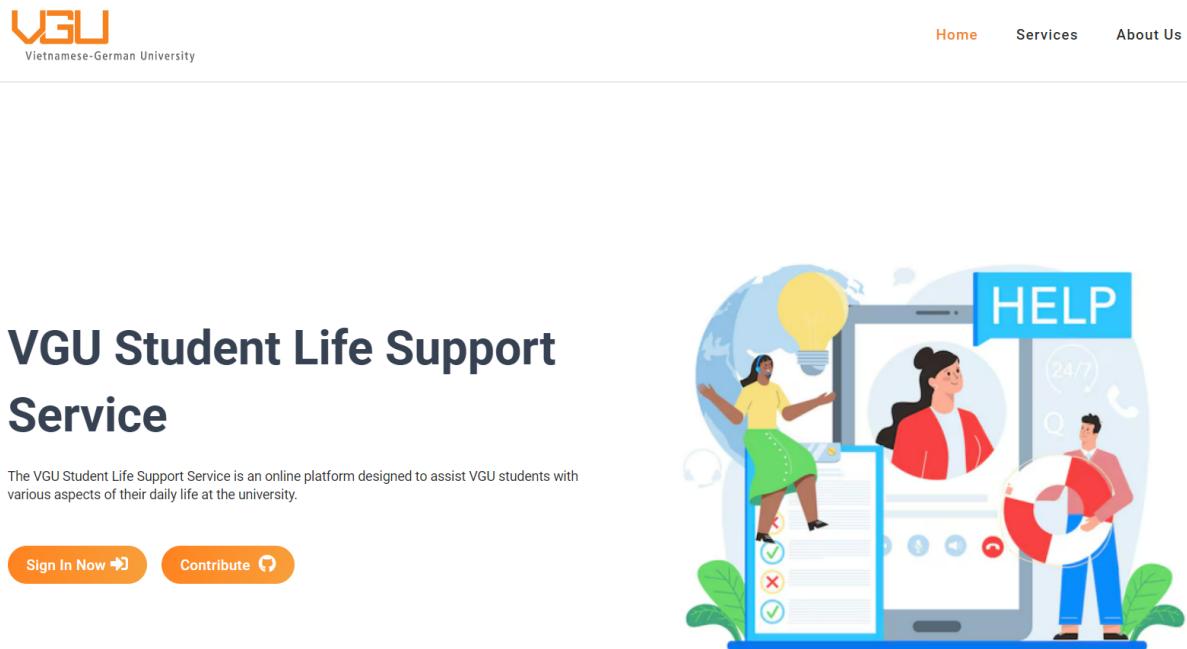
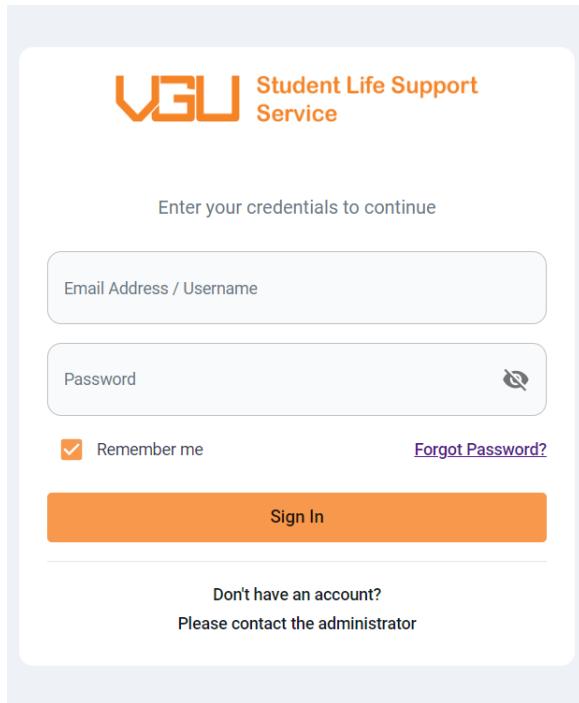


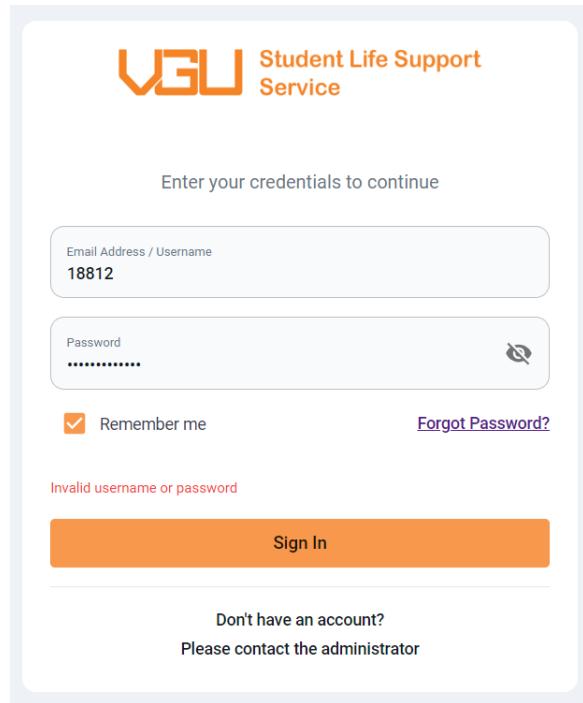
Figure 25: Landing Page

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



The sign-in form features the VGU logo and the text "Student Life Support Service". It includes fields for "Email Address / Username" (containing "18812") and "Password" (containing "*****"). There is a "Remember me" checkbox checked, a "Forgot Password?" link, and an orange "Sign In" button. Below the form are links for "Don't have an account?" and "Please contact the administrator".

Figure 26: Sign in Form



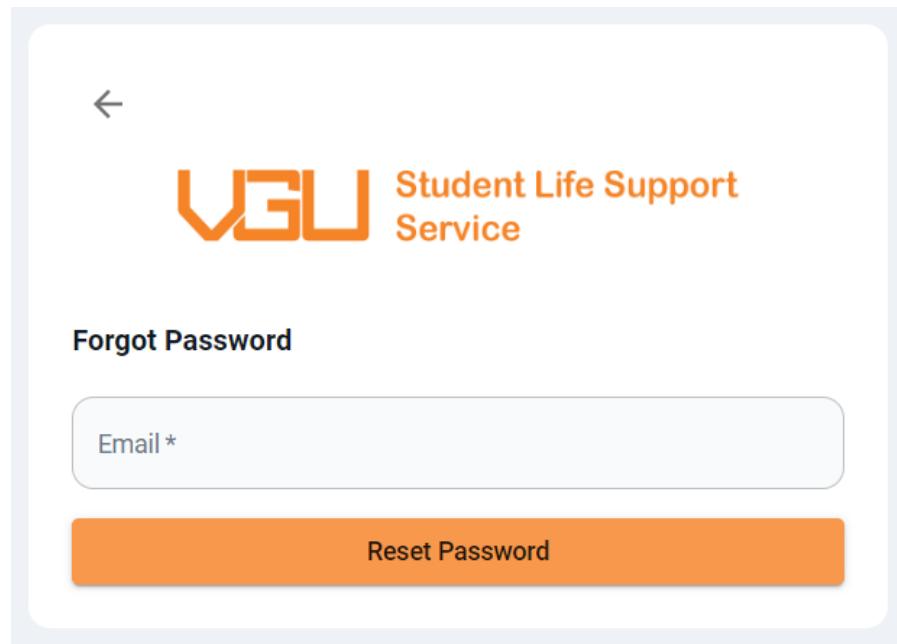
The failed sign-in attempt form shows the same header and field layout as Figure 26. The "Email Address / Username" field contains "18812". The "Password" field contains "*****". A red error message "Invalid username or password" is displayed above the "Sign In" button. The "Sign In" button is orange.

Figure 27: 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 26). If you provide incorrect credentials, a warning message will appear, indicating that you need to correct either your username or password (Figure 27).

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 26). Subsequently, provide your email address in the 'Forgot Password' form and click on 'Reset Password' (see Figure 28).



The image shows a mobile-style reset password form. At the top is the VGU logo and the text "Student Life Support Service". Below this is the heading "Forgot Password". A large input field is labeled "Email *". At the bottom is a large orange button labeled "Reset Password".

Figure 28: 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 29, 31). Conversely, if the email is not found in the system, a failure notification will indicate that the email does not exist (Figure 30).

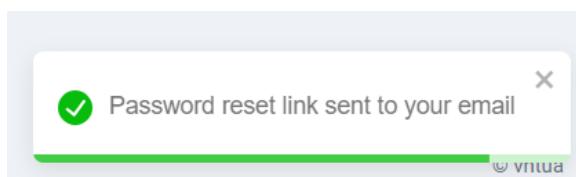


Figure 29: Reset password successfully

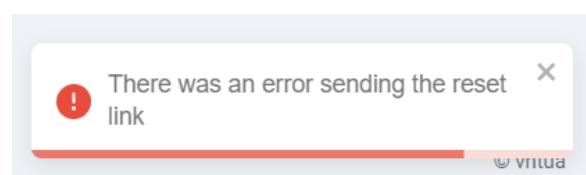


Figure 30: Reset password failed

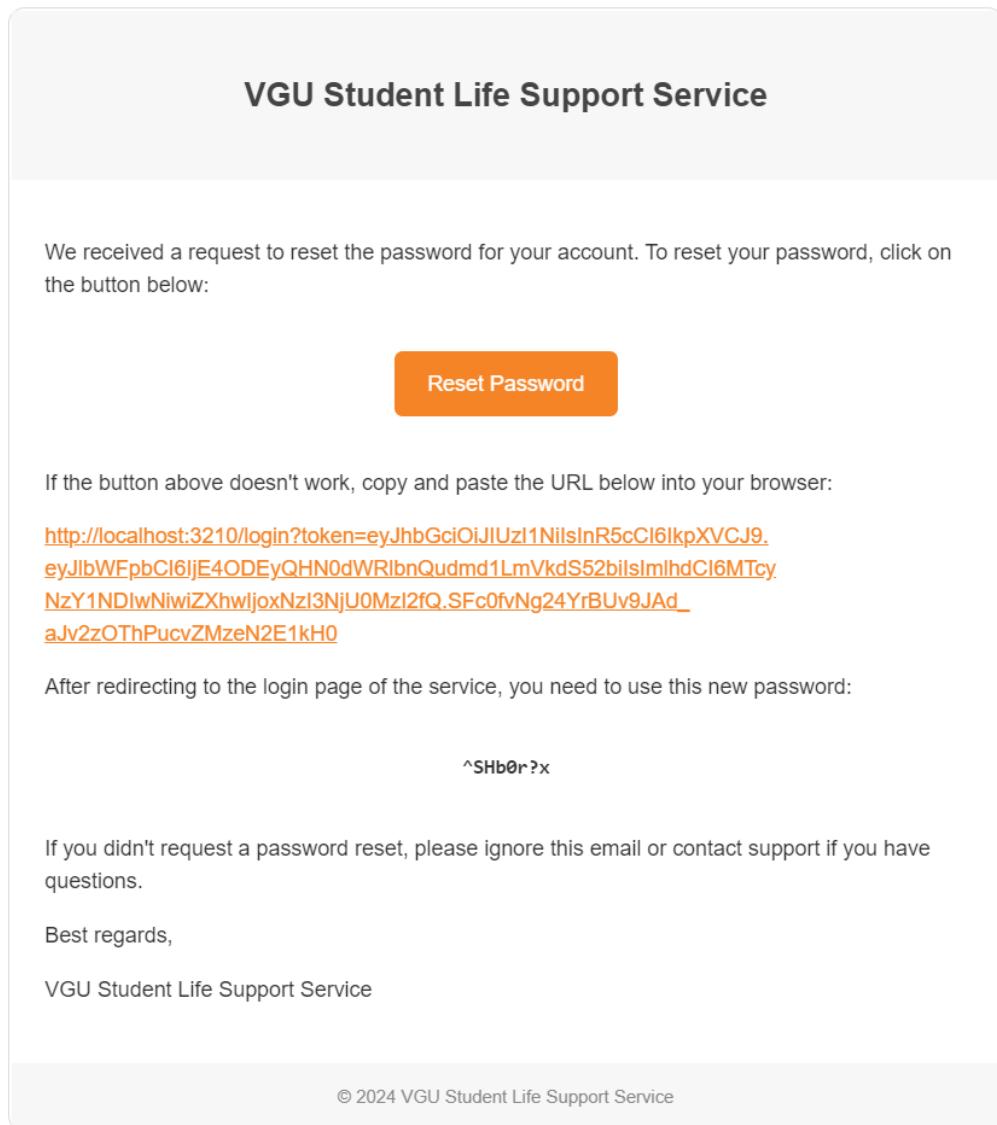


Figure 31: Reset password email instructions

5.3 Student's functions

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

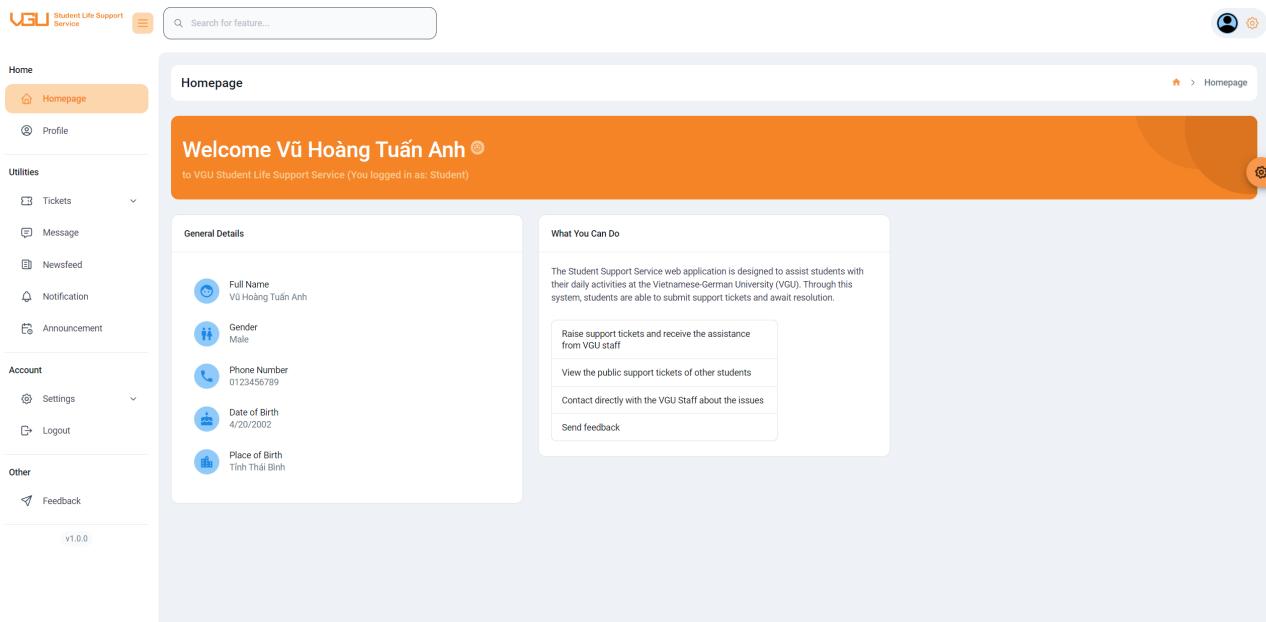


Figure 32: Student's Home Page

5.3.1 View Profile

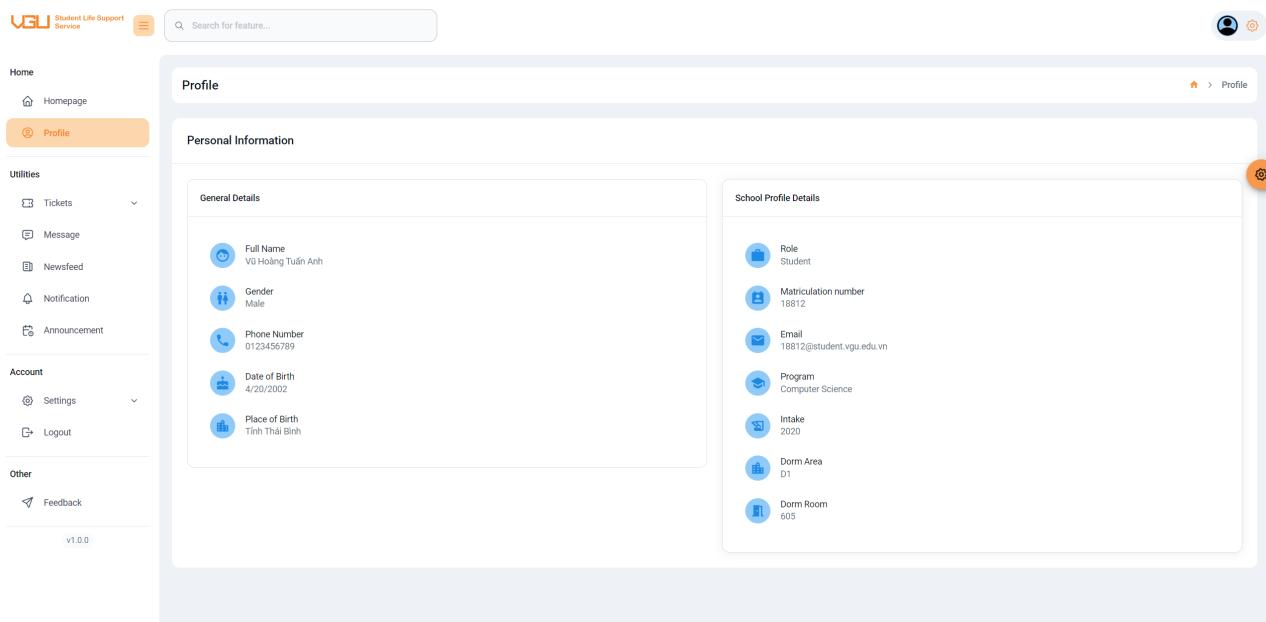


Figure 33: Student's Profile Page

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 34)

The screenshot shows the 'My tickets' page. On the left is a sidebar with links for Home, Utilities (Tickets, Create a ticket, Rate tickets, Message, Newsfeed, Notification, Announcement), Account (Settings, Logout), and Other (Feedback). The main area has a search bar and a table titled 'My tickets'. The table columns are: Ticket ID, Ticket Type, Subject, Audience Type, Status, Created Date, Ended Date, and Actions. There are four rows of data:

Ticket ID	Ticket Type	Subject	Audience Type	Status	Created Date	Ended Date	Actions
71	Lost items	I lost my phone	public	pending	9/30/2024, 11:35:50 AM	N/A	...
62	Violence	Students fight at dorm	public	done	9/26/2024, 5:08:42 PM	9/26/2024, 5:14:54 PM	...
61	Lost items	I just lost my identity card	public	done	9/25/2024, 11:17:37 PM	9/28/2024, 3:07:57 AM	...
39	Dormitory issues	Broken Door in Dormitory Room	private	done	9/25/2024, 9:33:04 PM	9/26/2024, 6:02:53 PM	...

To the right of the table is a detailed view of ticket #71. The ticket subject is 'I lost my phone'. The details pane shows:

- Student: Vũ Hoàng Tuấn Anh, ID: 18812
- Created Date: 9/30/2024, 11:35:50 AM
- Ended Date: N/A
- Dorm: D1 - 605
- Ticket Type: Lost Items
- Details: At 10 AM this morning, I lost my phone while studying at the library. The phone model is Samsung S24 Ultra Black Phantom. If anyone sees my phone, please contact me. Thank you very much.
- Audience Type: public
- Message: #71
- Status: pending
- Attachments: A small image of a black smartphone.

Figure 34: 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 35)

The screenshot shows the 'Create a support ticket' page. On the left, there's a sidebar with links for Home, Profile, Utilities (Tickets selected), Tickets (My tickets, Create a ticket, Rate tickets), Message, Newsfeed, Notification, Announcement, Account (Settings, Logout), and Other (Feedback). The main area has a search bar at the top right. Below it, there's a 'Create a support ticket' form with fields for Ticket Type, Subject, Details, Audience Type (with a file upload area), and Uploaded Files. At the bottom is a large orange 'Create This Ticket' button.

Figure 35: 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 36, 37)

The screenshot shows the 'Rate tickets' page of the VGU Student Life Support Service. The left sidebar includes links for Home, Profile, Utilities (Tickets, Create a ticket, Rate tickets, Message, Newsfeed, Notification, Announcement), Account (Settings, Logout), and Other (Feedback). The main content area has a search bar and a table listing three tickets. The table columns are: Ticket ID, Ticket Type, Subject, Audience Type, Status, Created Date, Ended Date, Rating Score, and Actions. The first ticket (Ticket ID 62) is for 'Violence' (Subject: Students fight at dorm, Audience Type: public, Status: done, Rating Score: 5/5). The second ticket (Ticket ID 61) is for 'Lost items' (Subject: I just lost my identity card, Audience Type: public, Status: done, Rating Score: N/A). The third ticket (Ticket ID 39) is for 'Dormitory issues' (Subject: Broken Door in Dormitory Room, Audience Type: private, Status: done, Rating Score: 4/5).

Figure 36: Student's Rate Tickets Page

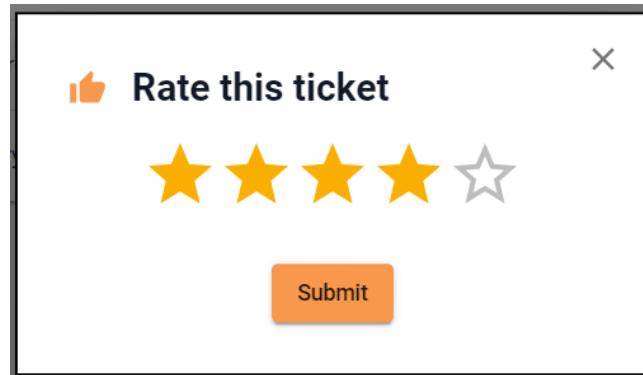


Figure 37: Submit a ticket rating

5.3.5 Message

The screenshot shows the 'Message' page of the VGU Student Life Support Service. On the left, there is a sidebar with the following menu items:

- Home
 - Homepage
 - Profile
- Utilities
 - Tickets
 - Message** (highlighted in orange)
 - Newsfeed
 - Notification
 - Announcement
- Account
 - Settings
 - Logout
- Other
 - Feedback

At the bottom of the sidebar, it says 'v1.0.0'.

The main content area is titled 'Message'. It shows a conversation between a student and a staff member. The student messages are in blue bubbles, and the staff messages are in grey bubbles. The conversation is as follows:

- I am too scared, but now everything is fine
Vũ Hoàng Tuấn Anh
9/26/2024, 5:10:42 PM
- OK, if there exists some incidents like this in the future, please let me know
Nguyễn Nguyễn Vũ
9/26/2024, 5:11:05 PM
- Thank you very much
Vũ Hoàng Tuấn Anh
9/26/2024, 5:11:11 PM
- You're welcome
Nguyễn Nguyễn Vũ
9/26/2024, 5:11:17 PM

At the bottom, there is a text input field with 'Type a message' placeholder and an orange 'Send' button.

Figure 38: 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 38)

5.3.6 Newsfeed

The screenshot shows the 'Newsfeed' section of the VGU Student Life Support Service. On the left is a sidebar with links for Home, Utilities (Tickets, Message, Newsfeed - highlighted in orange), Account (Settings, Logout), and Other (Feedback). The main area has a search bar and two buttons: 'Refresh' and 'Create A Ticket'. Three ticket cards are displayed:

- Ticket #71**: **I lost my phone**. Created by Vu Hoang Tuan Anh (ID: 18812) on 9/30/2024 at 11:35:50 AM. Ended N/A. At Dorm D1 - 605. Ticket Type: Lost Items. Audience Type: public. Status: pending. Attachments: A photo of a Samsung S24 Ultra Black Phantom phone.
- Ticket #68**: **2 students had a fight at dorm**. Created by Bá Nguyễn Quốc Anh (ID: 17965) on 9/27/2024 at 6:47:13 AM. Ended N/A. At Dorm D1 - 123. Ticket Type: Health problems. Audience Type: public. Status: pending. Attachments: A photo of two male students in a conflict.
- Ticket #67**: **I just caught a cold**. Created by Bá Nguyễn Quốc Anh (ID: 17965) on 9/27/2024 at 6:46:04 AM. Ended N/A. At Dorm D1 - 123. Ticket Type: Health problems. Audience Type: public. Status: in progress. Attachments: A photo of a person blowing their nose.

Figure 39: 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 39)

5.3.7 Notification

The screenshot shows the 'Notification' section of the VGU Student Life Support Service. The left sidebar includes links for Home, Utilities (Tickets, Message, Newsfeed), a selected 'Notification' tab, and Account (Settings, Logout). The main content area displays two notifications:

- Call for application for tuition fee reduction for domestic students in academic year 2024/25**
9/24/2024, 4:13:31 AM
Trần Văn Sinh

Dear Students,
We are glad to announce Call for application for tuition fee reduction for domestic students in academic year 2024/25 (attached file) published at <https://vgu.edu.vn/vi/chinh-sach-giam-hoc-phi>
This policy applies for Vietnamese students only.
If you have any question, please don't hesitate to contact and submit your application documents to Ms Le Thi Hanh, Deputy head of Academic and Student Affairs Department (ASA) by 31/10/2024 (Tel: +84-274 222 0990, Ext. 70132) or meet her in person at room 218, ASA, Academic Building, 2nd floor, Vanh Dai 4 Street, Quarter 4, Thoi Hoa Ward, Ben Cat City, Binh Duong Province.
For the students in the campus in Ho Chi Minh City, feel free to have your Faculty Assistant to pass your applications to Ms Hanh.
Only hard copies of application documents are accepted.
Best Regards,
Student Affairs Team
- New Campus Cafeteria Opening on October 1st**
9/21/2024, 2:15:13 PM
Trần Văn Sinh

Figure 40: 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 40)

5.3.8 Announcement

The screenshot shows the 'Announcement' section of the VGU Student Life Support Service website. The left sidebar includes links for Home (Homepage, Profile), Utilities (Tickets, Message, Newsfeed, Notification, Announcement - highlighted in orange), Account (Settings, Logout), and Other (Feedback). The main content area displays an announcement titled 'Scheduled Maintenance for the Service' by Phan Thị Hà on 9/21/2024 at 3:30:08 AM. The announcement text states:

Scheduled Maintenance for the Service
9/21/2024, 3:30:08 AM
Phan Thị Hà

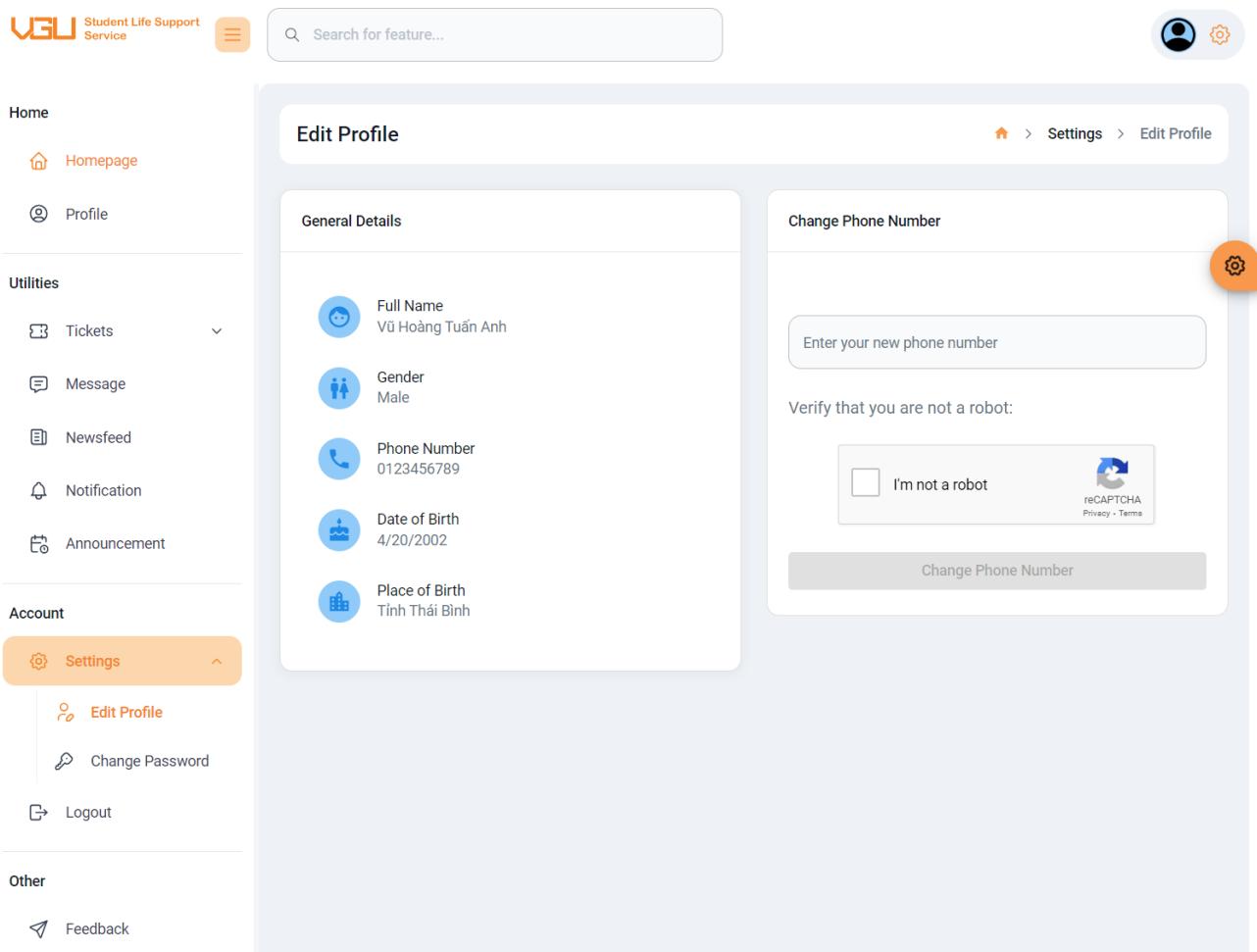
Announcement: Scheduled Maintenance for A Service
Dear Students,
Please be informed that A Service will undergo scheduled maintenance on:
Date: 30.09.2024
Time: from 10:00 to 14:00
During this time, the service will be temporarily unavailable as we perform essential updates and improvements. We apologize for any inconvenience this may cause and appreciate your understanding.
Should you have any questions or concerns, please feel free to contact the admin team.
Thank you for your cooperation.
Best regards,
VGU SLS ADMIN TEAM

Figure 41: 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 41)

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.



The screenshot shows the 'Edit Profile' page of a web application. The left sidebar has a 'Settings' section highlighted with an orange background. The main content area is titled 'Edit Profile' and contains two tabs: 'General Details' and 'Change Phone Number'. The 'General Details' tab displays the following information:

Icon	Label	Value
User icon	Full Name	Vũ Hoàng Tuấn Anh
Gender icon	Gender	Male
Phone icon	Phone Number	0123456789
Birthday icon	Date of Birth	4/20/2002
Place of Birth icon	Place of Birth	Tỉnh Thái Bình

The 'Change Phone Number' tab contains a form with a placeholder 'Enter your new phone number' and a reCAPTCHA verification box labeled 'I'm not a robot'. A 'Change Phone Number' button is at the bottom of this tab. The top right corner of the page shows a user profile icon and a gear icon.

Figure 42: Student's Edit Profile Page

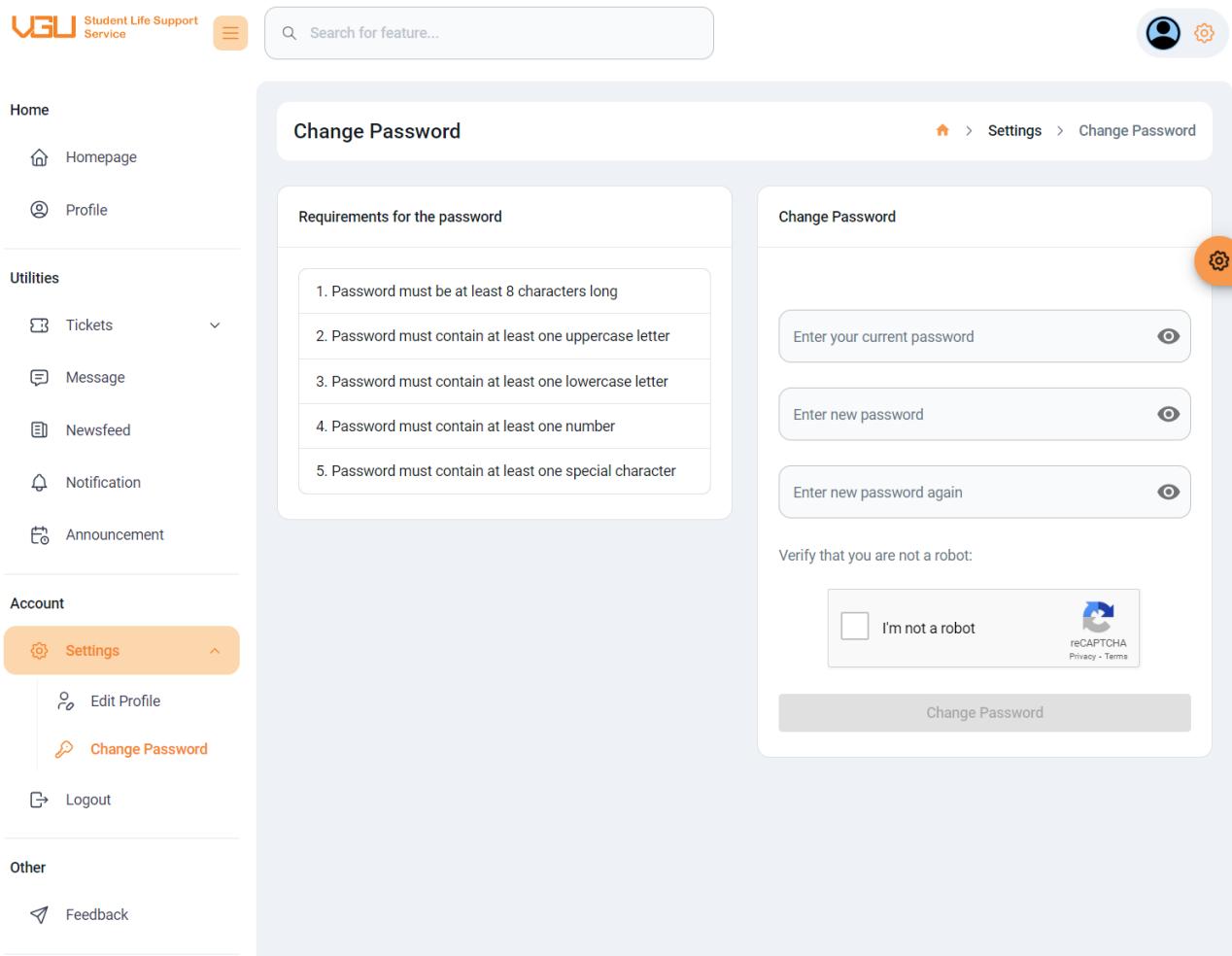


Figure 43: Student's Change Password Page

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 44)

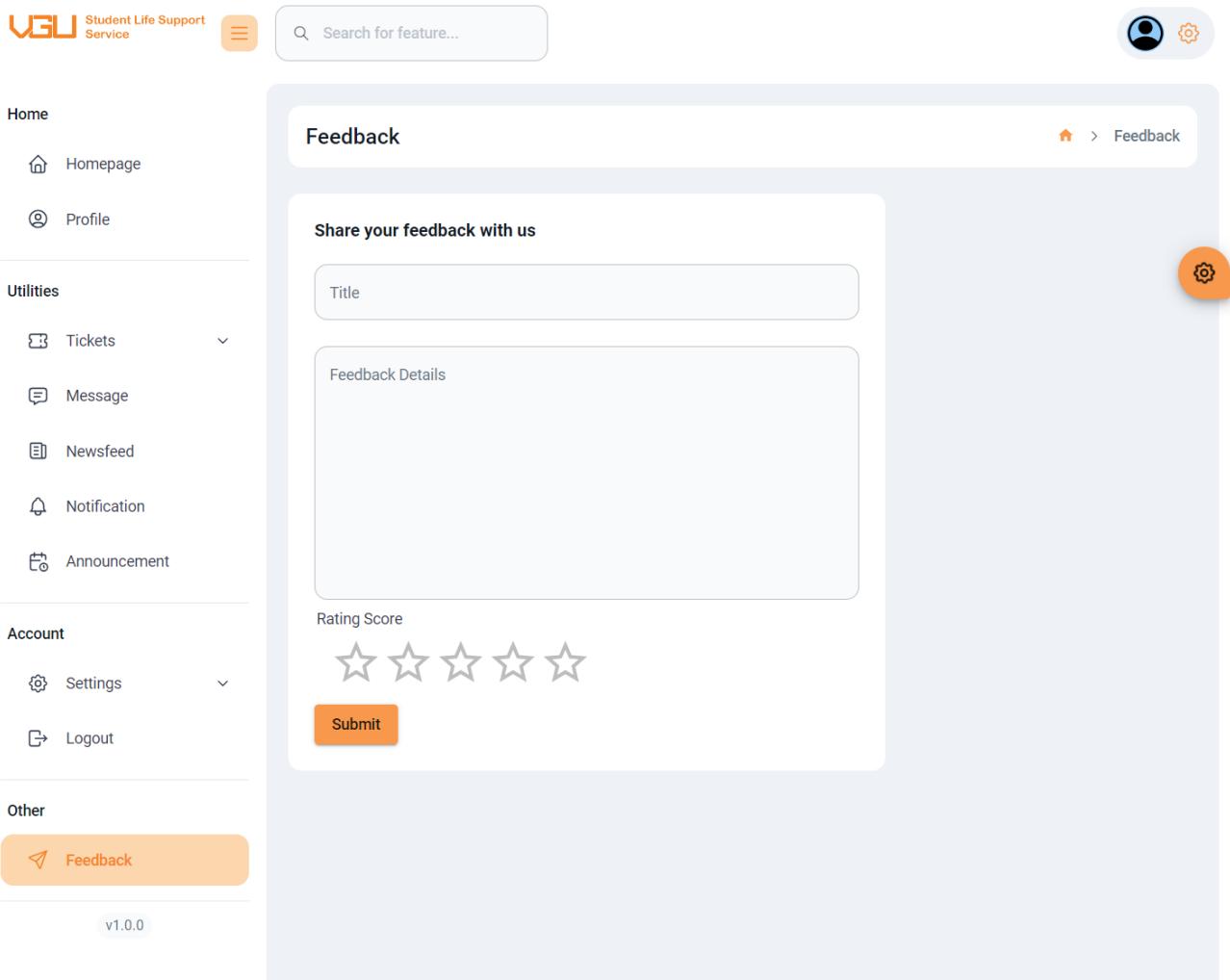


Figure 44: Student's Feedback Page

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.

The screenshot shows the VGU Student Life Support Service web application interface. On the left is a sidebar with links for Home, Utilities (Tickets, Available tickets, Tickets Handling, History, Message, Newsfeed, Notification, Announcement), Account (Settings, Logout), and Other (Feedback). The main area has a search bar and a link to 'Available tickets'. The 'Available tickets' section lists two items:

Ticket ID	Ticket Type	Subject	Audience Type	Status	Created Date	Ended Date	Actions
71	Lost items	I lost my phone	public	pending	9/30/2024, 11:35:50 AM	N/A	eye, magnifying glass
68	Health problems	2 students had a fight at dorm	public	pending	9/27/2024, 6:47:13 AM	N/A	eye, magnifying glass

On the right, a detailed view of Ticket #71 is shown. The subject is "I lost my phone". The ticket type is "Lost items", audience type is "public", and status is "pending". The message states: "At 10 AM this morning, I lost my phone while studying at the library. The phone model is Samsung S24 Ultra Black Phantom. If anyone sees my phone, please contact me. Thank you very much." It includes details about the audience type (public), message status (not available), and attachments (a small image of a smartphone).

Figure 45: Staff's Available Tickets Page

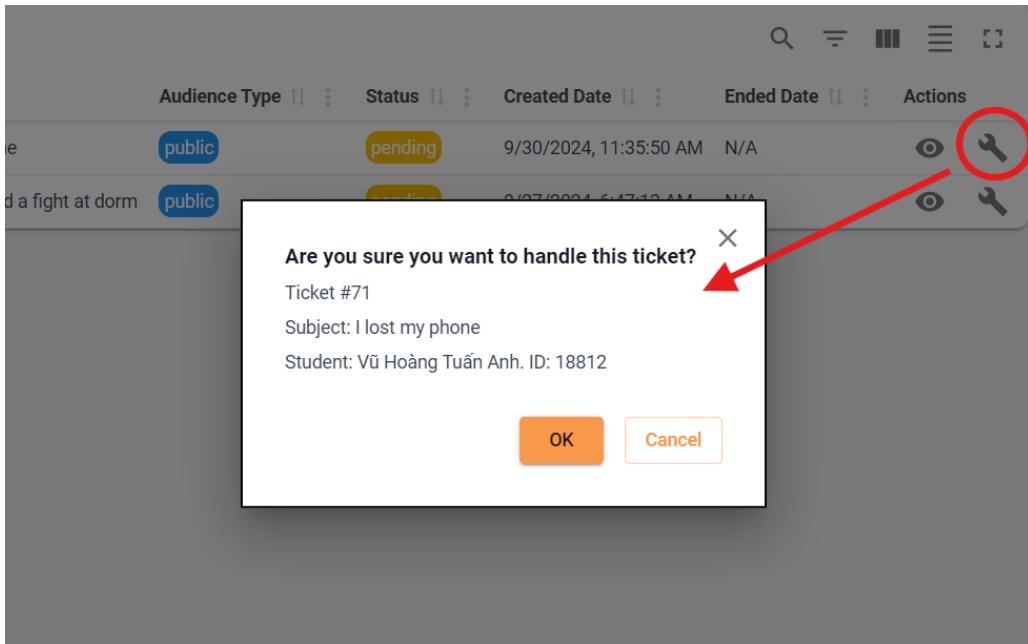


Figure 46: Handle a ticket

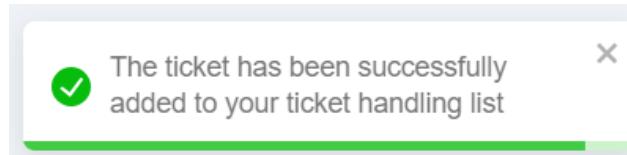


Figure 47: Successfully add a ticket to ticket handling list

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.

The screenshot shows the 'Tickets Handling' section of the VGU Student Life Support Service. On the left, there's a sidebar with links for Home, Utilities (Tickets, Available tickets, Tickets Handling, History, Message, Newsfeed, Notification, Announcement), Account (Settings, Logout), and Other (Feedback). The main area has a search bar and a 'Refresh' button. Below it is a table titled 'Tickets Handling' with columns: Ticket ID, Ticket Type, Subject, Audience Type, Status, Created Date, Ended Date, and Actions. One row is visible: Ticket ID 71, Lost Items, I lost my phone, public, in progress, 9/30/2024, 11:35:50 AM, N/A. To the right of the table is a detailed view of Ticket #71. It shows the subject 'I lost my phone', student information (VU Hoang Tuan Anh, ID: 18812), creation date (9/30/2024, 11:35:50 AM), end date (N/A), and dorm (D1 - 605). The ticket type is 'Lost Items'. A message box contains the text: 'At 10 AM this morning, I lost my phone while studying at the library. The phone model is Samsung S24 Ultra Black Phantom. If anyone sees my phone, please contact me. Thank you very much.' Below the message are sections for Audience Type (public), Message (#71), Status (in progress), and Attachments, which includes a small image of a Samsung S24 Ultra phone.

Figure 48: Staff's Tickets Handling List

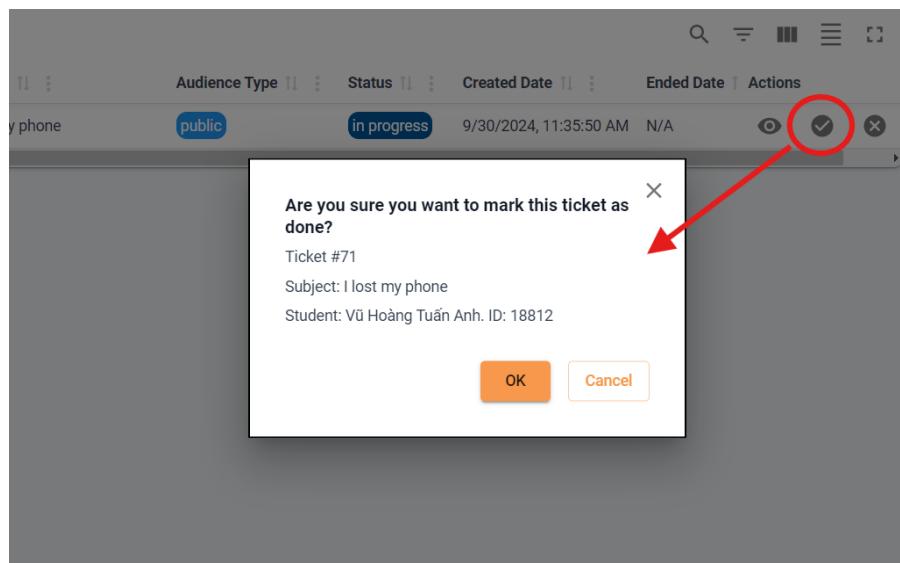


Figure 49: Mark a ticket as done

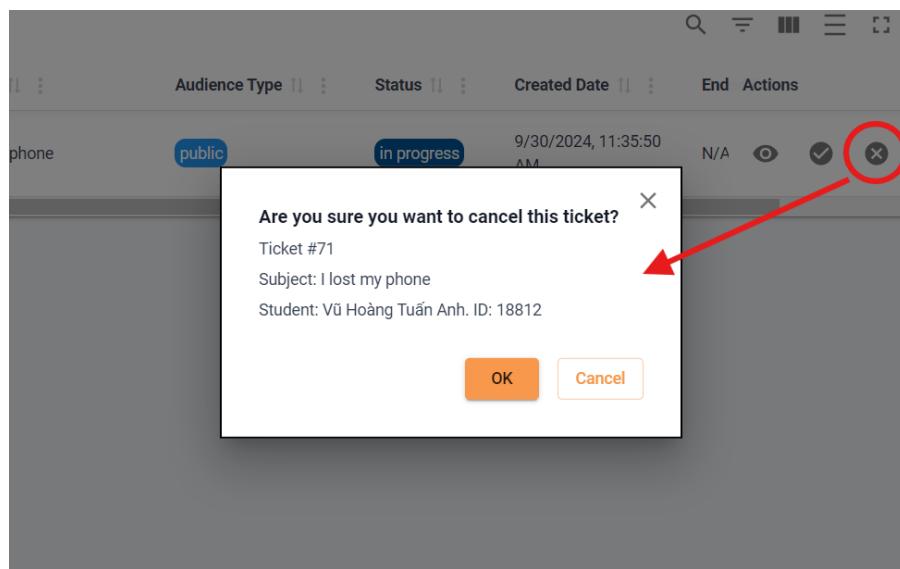


Figure 50: Cancel a ticket

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 51)

Ticket ID	Ticket Type	Subject	Audience Type	Status	Created Date	Ended Date	Actions
70	Harassment	Please help me	public	done	9/27/2024, 10:40:32 PM	9/27/2024, 10:48:05 PM	
62	Violence	Students fight at dorm	public	done	9/26/2024, 5:08:42 PM	9/26/2024, 5:14:54 PM	
38	Scam	A guy scams my laptop	public	done	9/25/2024, 12:24:02 AM	9/26/2024, 5:59:31 PM	
35	Violence	Some one has fought the lecturer	public	done	9/24/2024, 2:54:40 AM	9/26/2024, 6:00:31 PM	
34	Lost Items	I lost my room key	public	done	9/23/2024, 10:46:57 PM	9/26/2024, 5:58:44 PM	
3	Dormitory issues	Broken Faucet	private	done	9/22/2024, 7:20:32 PM	9/23/2024, 10:30:18 AM	

Figure 51: 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 52)

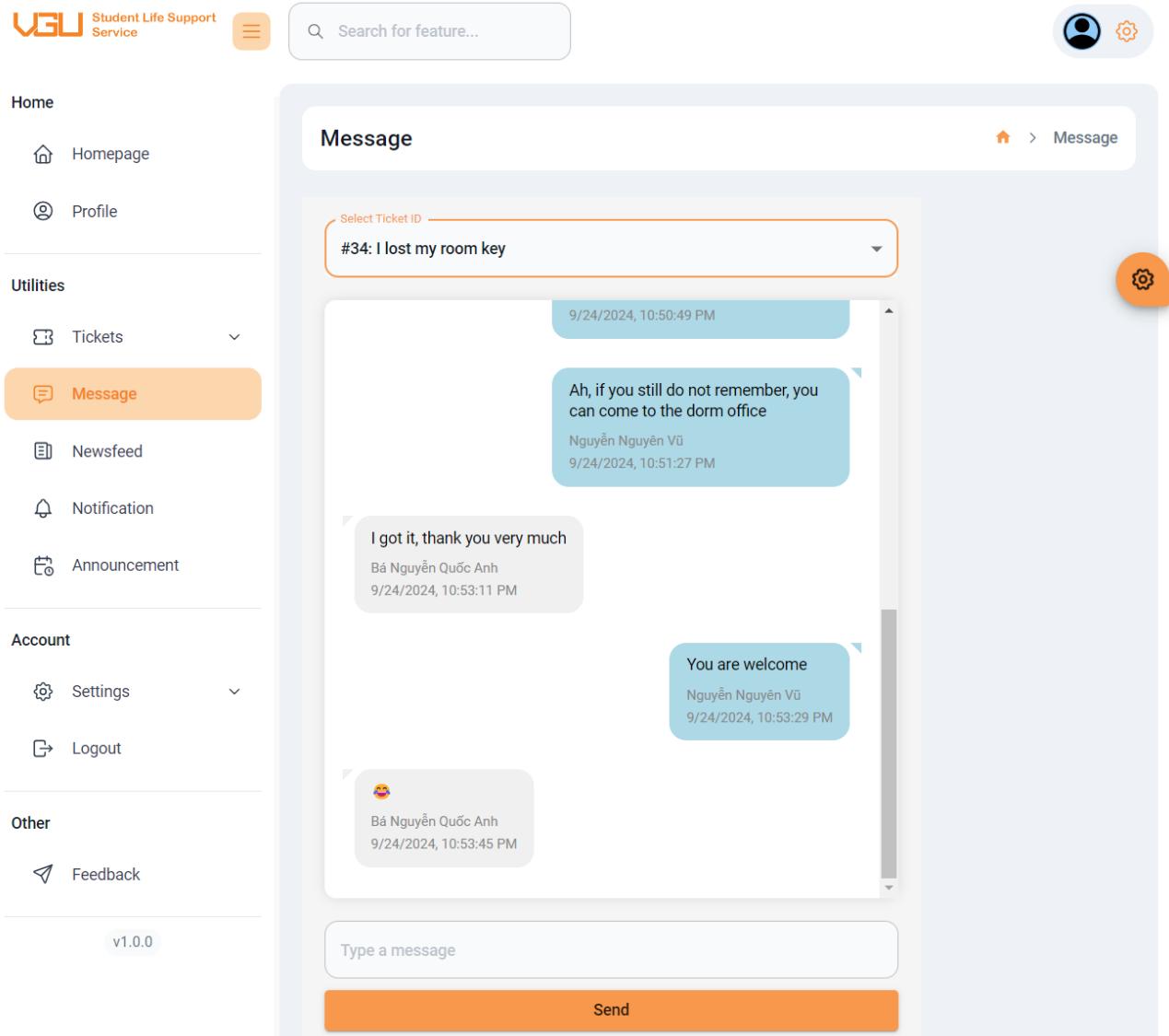


Figure 52: Staff's Message Page

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.

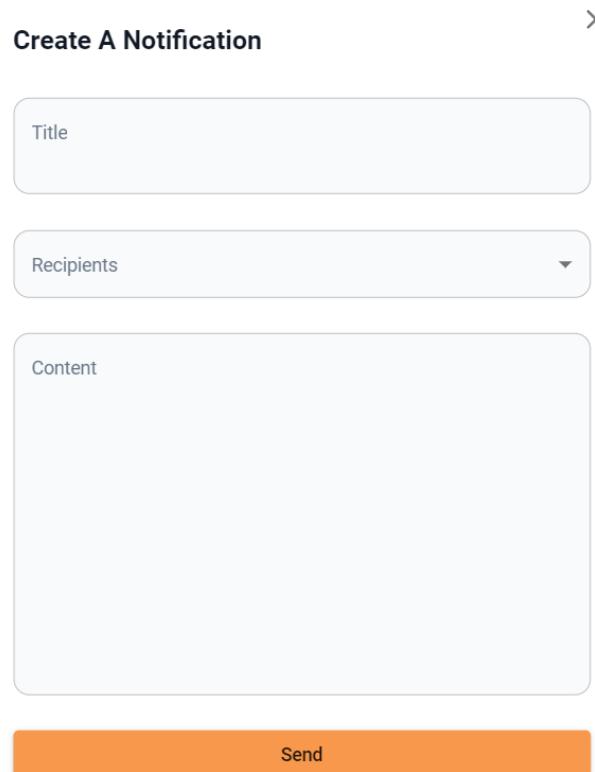


Figure 53: Create a Notification Modal

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.

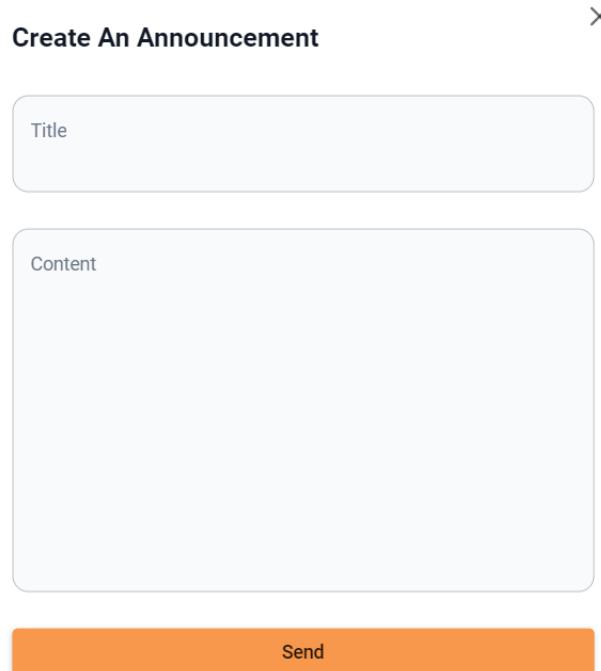


Figure 54: Create an Announcement Modal

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.

The screenshot shows the 'Tickets' section of the Admin's Ticket Management Page. On the left, there is a sidebar with navigation links for Home, Management (with Tickets selected), Utilities, and Account. The main area displays a table of tickets with columns for Ticket ID, Ticket Type, Subject, Audience Type, Status, Created Date, Ended Date, and Actions. A search bar at the top right allows for searching features. On the right, a detailed view of ticket #38 is shown, titled 'A guy scams my laptop'. It includes information about the student (Bá Nguyễn Quốc Anh, ID: 17965), creation date (9/25/2024, 12:24:02 AM), end date (9/26/2024, 5:59:31 PM), dorm (D1-123), and ticket type (Scam). The ticket details state: 'I was in my dorm one evening, working on an assignment when this guy knocked on my door. He looked like any other student—wearing a hoodie, carrying a backpack—so I didn't think much of it. He introduced himself as someone who lived down the hall, said he had a class... [Read more](#)'. Below the details, there are sections for Audience Type (public), Message (not available), Status (done), and Attachments, which includes a small image of a person in a mask.

Figure 55: 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.

User ID	Username	Full Name	Email	Role	Gender	Created Date	Program	Area	Room	Phone Number	Actions			
4	18812	Vũ Hoàng Tuấn Anh	18812@student.vgu.edu.vn	Student	Male	9/18/2024, 4:40:50 PM	Computer Science	D1	605	0123456789				
16	10002	Trần Văn Sinh	10002@staff.vgu.edu.vn	Student Affairs	Male	9/18/2024, 11:44:28 PM		0	0	0981782621				
8	10000	Trần Thị Hương	10000@student.vgu.edu.vn	Student	Female	9/18/2024, 5:43:25 PM	Business Administration	D2	555	0123912300				
1	17965	Bá Nguyễn Quốc Anh	17965@student.vgu.edu.vn	Student	Male	9/18/2024, 4:40:49 PM	Computer Science	D1	123	0987654123				
13	10001	Nguyễn Nguyễn Vũ	10001@staff.vgu.edu.vn	Dorm Staff	Male	9/18/2024, 5:47:59 PM		0	0	0991231441				
19	10003	Phan Thị Hà	10003@admin.vgu.edu.vn	Admin	Female	9/18/2024, 11:46:31 PM		0	0	0933310103				

Figure 56: 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.

The screenshot shows the 'Dormitory' management page. At the top right, there is a search bar with the placeholder 'Search for feature...'. To the right of the search bar are two icons: a user profile and a gear for settings. Below the search bar, the title 'Dormitory' is displayed, along with a 'Refresh' button and a green 'Create A New Dorm' button. The main content area is a table listing dormitory rooms. The columns are 'Dorm Area' and 'Dorm Room'. The rows show the following data:

Dorm Area	Dorm Room	Actions
0	0	
D1	001	
D1	002	
D1	003	
D1	004	
D1	005	
D1	006	
D1	007	
D1	008	
D1	009	
D1	010	
D1	011	
D1	012	
D1	013	

At the bottom of the table, there is a pagination indicator showing 'Page 1 of 150' and '150 of 1500'.

Figure 57: 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.

The screenshot shows the 'Logs' management page. On the left, there is a sidebar with navigation links: Home, Profile, Management (Tickets, Users, Dormitory, Logs - highlighted in orange), Utilities (Report, Newsfeed), Account (Settings, Logout). The main area has a search bar and tabs for Refresh, Export To CSV, Export To PDF, Export To Excel, and Clean All Logs. A table lists log entries with columns: Log ID, Event Type, Timestamp, Description, Username, Full Name, Role, and Actions. The table contains 13 rows of log entries, mostly 'info' level events, with one 'critical' event at Log ID 645. The 'Logs' link in the sidebar is also highlighted in orange.

Log ID	Event Type	Timestamp	Description	Username	Full Name	Role	Actions
645	critical	1970-01-01 08:00:00	User has created a ticket	18812	Vũ Hoàng Tuấn Anh	Student	
671	info	2024-09-30 13:43:07	User successfully get all roles	10003	Phan Thị Hà	Admin	
669	critical	2024-09-30 13:43:07	All users have been retrieved	10003	Phan Thị Hà	Admin	
670	info	2024-09-30 13:43:07	User is getting all roles	10003	Phan Thị Hà	Admin	
668	info	2024-09-30 13:43:07	All users are being retrieved	10003	Phan Thị Hà	Admin	
667	info	2024-09-30 13:42:48	User successfully get all roles	10003	Phan Thị Hà	Admin	
665	critical	2024-09-30 13:42:48	All users have been retrieved	10003	Phan Thị Hà	Admin	
666	info	2024-09-30 13:42:48	User is getting all roles	10003	Phan Thị Hà	Admin	
664	info	2024-09-30 13:42:48	All users are being retrieved	10003	Phan Thị Hà	Admin	
663	info	2024-09-30 13:39:32	Username 10003 has been retrieved	10003	Phan Thị Hà	Admin	
662	info	2024-09-30 13:39:32	Username 10003 is being retrieved	10003	Phan Thị Hà	Admin	
661	security	2024-09-30 13:39:32	Username 10003 locked in successfully	10003	Phan Thị Hà	Admin	

Figure 58: 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.

The screenshot shows the 'Feedback' management page. On the left is a sidebar with navigation links for Home, Management (Tickets, Users, Dormitory, Logs, Feedback), Utilities (Report, Newsfeed), and Account (Settings, Logout). The main area has a search bar and a table titled 'Feedback'. The table columns are: Feedback ID, Title, Rating Score, Created Date, and Actions. There are four rows of data:

Feedback ID	Title	Rating Score	Created Date	Actions
7	student feed	4	2024-09-26 13:06:26	
6	test	2	2024-09-26 13:04:59	
4	Working good as Student Role	5	2024-09-24 21:24:36	
2	Good service	3	2024-09-21 03:49:37	

At the bottom right of the table are buttons for 'Rows per page' (50), '1-4 of 4', and navigation arrows.

Figure 59: Admin's Feedback Management Page

5.5.6 Report

The screenshot shows the 'Report' page. The sidebar includes links for Home, Management (Tickets, Users, Dormitory, Logs, Feedback, Announcement, Notification), Utilities (Report, Newsfeed), and Account (Settings, Logout). The main content area has a search bar and two charts under 'Ticket Reports':

- Monthly Ticket Status:** A bar chart showing ticket counts for pending, in progress, done, and cancelled statuses across months. The x-axis is labeled '2024-09' and the y-axis ranges from 0 to 9. The legend indicates: pending (orange), in progress (green), done (blue), and cancelled (red).
- Ticket Status Distribution:** A pie chart showing the percentage distribution of ticket statuses. The legend indicates: pending (orange), in progress (green), done (blue), and cancelled (red).

At the bottom are 'Export To PNG' and 'Export To PDF' buttons.

Figure 60: 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

6.1 Conclusion

In conclusion, the Student Life Support Service web application represents a significant advancement in the way students can access and manage essential resources. By leveraging modern technologies such as ReactJS, Material UI, NodeJS, and PostgreSQL, the application provides a robust and responsive platform that meets the dynamic needs of its users. The use of ReactJS enables the creation of a seamless user interface, fostering an intuitive navigation experience. Material UI enhances this by offering a comprehensive set of pre-designed components that streamline the development process and maintain a consistent aesthetic across the application.

The backend, built with NodeJS and ExpressJS, ensures efficient data handling and real-time interactions through the implementation of SocketIO. This combination allows for swift processing of requests and timely updates, critical for a platform designed to support communication among students and staff. PostgreSQL's strong ACID compliance guarantees the integrity and reliability of data, which is crucial for managing sensitive information such as user accounts and communication logs.

Throughout the development process, extensive attention has been paid to user experience. The application is designed to be responsive, catering to various devices and screen sizes, which is essential in today's mobile-first world. Furthermore, thorough testing and validation have been conducted to ensure the application functions smoothly, providing users with a dependable resource for support and information.

Looking ahead, the project has laid a strong foundation for future enhancements. As the needs of students evolve and grow, so too must the capabilities of the application. By focusing on areas such as dynamic role management, scalability, and improved handling of concurrent user requests, the Student Life Support Service can adapt to changing user demands while ensuring high performance and security.

In essence, the development of this web application is not merely an endpoint but the be-

ginning of an ongoing journey. The commitment to continuous improvement will ensure that the platform remains relevant and valuable to students, supporting their academic and social endeavors in an increasingly digital world. With a clear vision for future enhancements and a solid technical foundation, the Student Life Support Service is poised to significantly impact student life on campus, promoting engagement and accessibility like never before.

6.2 Future Work

There are several key areas for improvement and expansion have been identified in the future:

1. **Dynamic Role Management:** Implementing a dynamic role management system will enhance the application's flexibility. This feature will allow administrators to create and assign specific permissions to different user roles, tailoring access and functionality based on user needs. By enabling fine-grained control over resource access, the system can better accommodate various user requirements and enhance security.
2. **Scalability:** To ensure the application can handle a growing number of users and increased data load, scalability must be a primary focus. This involves optimizing the architecture to support horizontal scaling, which can be achieved by implementing load balancers and distributing requests across multiple server instances. Additionally, strategies such as microservices architecture can be explored to further enhance scalability and maintainability.
3. **Handling High Concurrent User Requests:** Improving the application's ability to handle high volumes of simultaneous user requests is crucial for maintaining performance and user satisfaction. Techniques such as caching strategies (using Redis or similar technologies) can significantly reduce database load and improve response times. Additionally, optimizing database queries and utilizing connection pooling can further enhance the system's efficiency under heavy load.

By addressing these future work areas, the Student Life Support Service web application can continue to evolve and better serve its users in reality, ensuring a reliable and efficient platform for student engagement and support.

References

- [1] Apidog. Socket. IO vs. WebSocket: Keys Differences, Accessed on 01.10.2024. URL: <https://apidog.com/articles/socket-io-vs-websocket/>.
- [2] Google Documentation. What is reCAPTCHA?, Accessed on 01.10.2024. URL: <https://www.google.com/recaptcha/about/>.
- [3] JWT Documentation. Introduction to JSON Web Tokens, Accessed on 01.10.2024. URL: <https://jwt.io/introduction>.
- [4] PostgreSQL Documentation. About, Accessed on 01.10.2024. URL: <https://www.postgresql.org/about/>.
- [5] Vitejs Documentation. Why Vite, 2023. URL: <https://vitejs.dev/guide/why>.
- [6] Kinsta. What Is Express.js? Everything You Should Know, September 2023. URL: <https://kinsta.com/knowledgebase/what-is-express-js/>.
- [7] Aryan Patil. Three-tier architecture, 2023. URL: <https://hyperskill.org/learn/step/25083>.
- [8] Simon Prickett. What is Redis?: An Overview, February 2024. URL: <https://redis.io/learn/develop/node/nodecrashcourse/whatisredis>.
- [9] Justyna Rachowicz. When, How, And Why Use Node.js as Your Backend?, July 2024. URL: <https://www.netguru.com/blog/node-js-backend>.
- [10] Sanity. React.js overview, August 2023. URL: <https://www.sanity.io/glossary/react-js>.
- [11] Eric Simons. What is Vite (and why is it so popular?), September 2024. URL: <https://blog.stackblitz.com/posts/what-is-vite-introduction>.
- [12] Alesia Sirotka. What is Material UI?, March 2022. URL: <https://flatlogic.com/blog/what-is-material-ui/>.
- [13] Socket.IO. Introduction, Accessed on 01.10.2024. URL: <https://socket.io/docs/v4/>.

-
- [14] Taha Sufiyan. What Is Node.js? A Complete Guide for Developers, July 2024. URL: <https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs>.
 - [15] Goodluck Woha. Why should you use Material UI?, May 2023. URL: <https://blog.openreplay.com/why-should-you-use-material-ui/>.