FRANKFURT UNIVERSITY OF APPLIED SCIENCES
VIETNAMESE-GERMAN UNIVERSITY

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

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

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

BACHELOR THESIS

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

Binh Duong, Viet Nam

Declaration

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

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

Date

Acknowledgments

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

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

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

Abstract

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

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

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

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

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

Contents

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CSE Computer Science and Engineering

JWT JSON Web Token

REST Representational State Transfer

SQL Structured Query Language

UI User Interface

URL Uniform Resource Locator

UX User Experience

VGU Vietnamese-German University

List of Figures

List of Tables

List of Code Snippets

1 Introduction

1.1 Project Background

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

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

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

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

1.2 Problem Statement

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

1.3 Objectives of the Project

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

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

Table 1: System key features

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

1.4 Scope of the Project

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

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

Table 2: System key components

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

1.5 Thesis Structure

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

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

2 Literature Review

2.1 Existing solutions

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

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

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

2.1.2 Existing University and Open-source Ticketing Systems

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

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

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

Table 3: Existing University Ticketing Systems

2.1.3 Limitations of Existing Solutions in the University Context

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

2.2 Technology Review

2.2.1 Frontend: ReactJS, Material UI, Vite

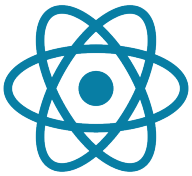


Figure 1: ReactJS Logo

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

```
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

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

Figure 2: Material UI Logo

Figure 3: Vite Logo

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

2.2.2 Backend: NodeJS, ExpressJS, SocketIO

Figure 4: NodeJS Logo

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

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

Figure 5: Expressjs Logo

Figure 6: SocketIO Logo

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

2.2.3 Authentication: JWT, Redis

Figure 7: JWT Logo

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

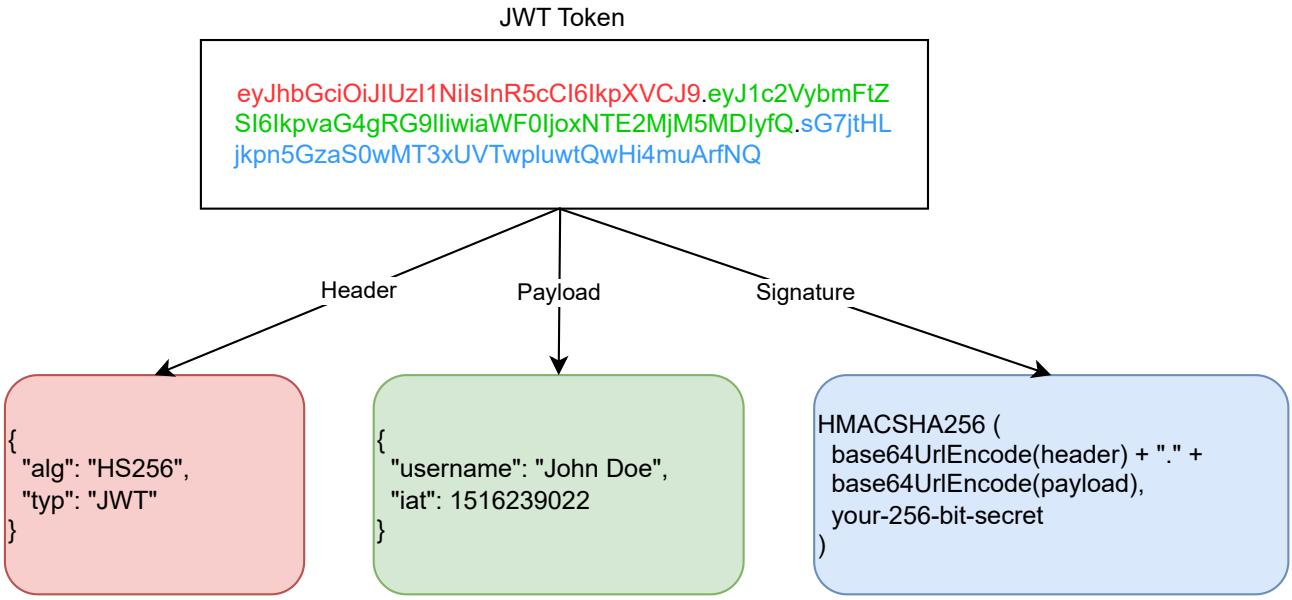


Figure 8: Detailed explanation of JWT-based authentication mechanism.

Figure 9: Redis Logo

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

2.2.4 Database: PostgreSQL

Figure 10: PostgreSQL RDBMS Logo

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

2.2.5 Responsive Web Design: Techniques and Tools

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

2.3 Theoretical Background

2.3.1 Ticket Management Systems

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

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

2.3.2 Real-Time Communication Tools

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

2.3.3 Web Application Development Best Practices

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

2.4 Gap Analysis

2.4.1 What is Missing from Existing Solutions

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

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

2.4.2 How the Student Life Support Service Fills These Gaps

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

3 System Design

3.1 Functional Requirements

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

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

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

Table 4: Functional Requirements

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

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

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

Table 5: Functional Requirements by User Roles

3.2 Non-Functional Requirements

Categorized: NF-Nonfunctional

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

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

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

Table 6: Non-Functional Requirements

3.3 Use Case Diagrams

Figure 11: Student Use Case Diagram

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

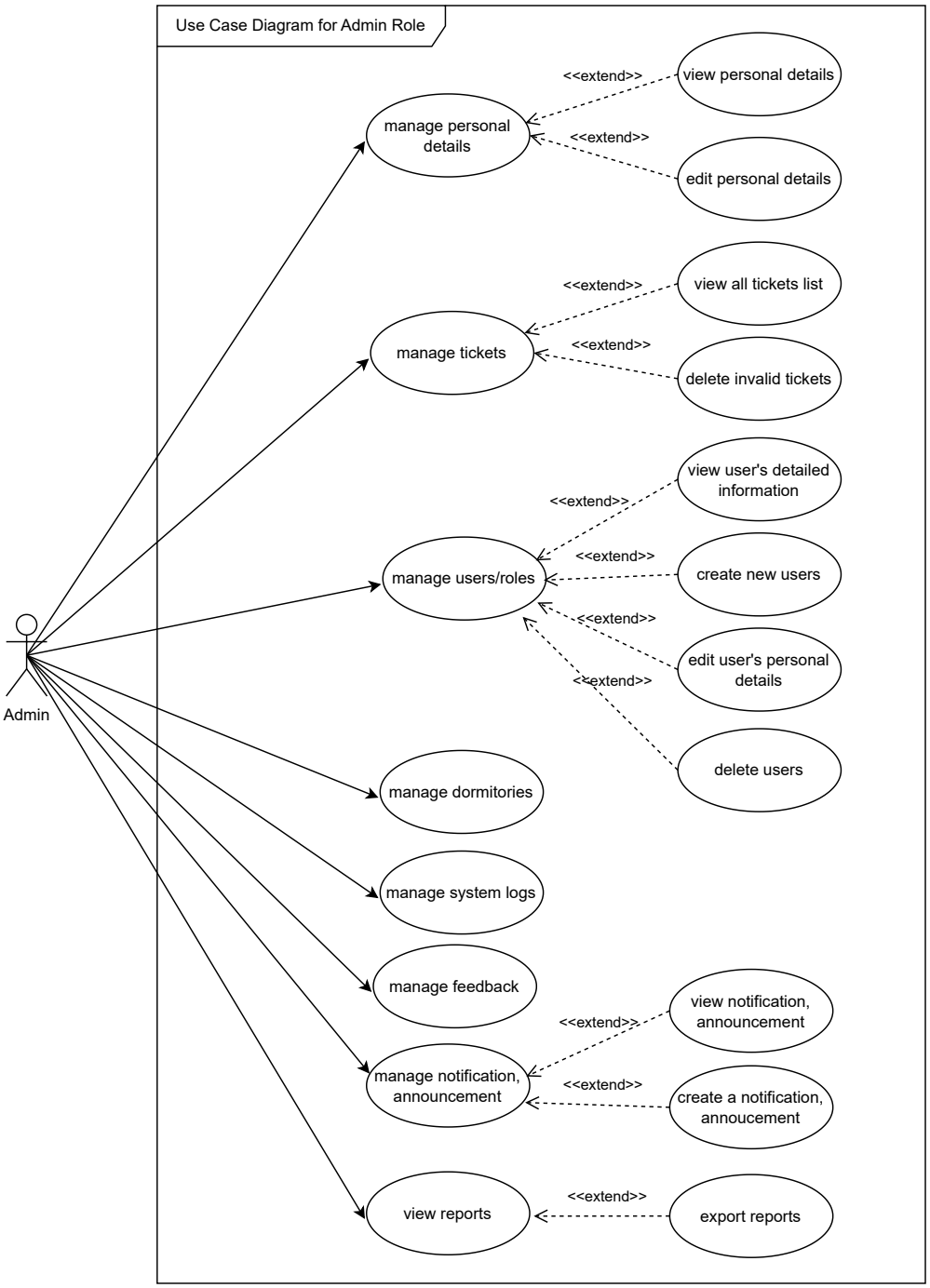


Figure 13: Admin Use Case Diagram

3.4 Process Workflow Diagrams

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

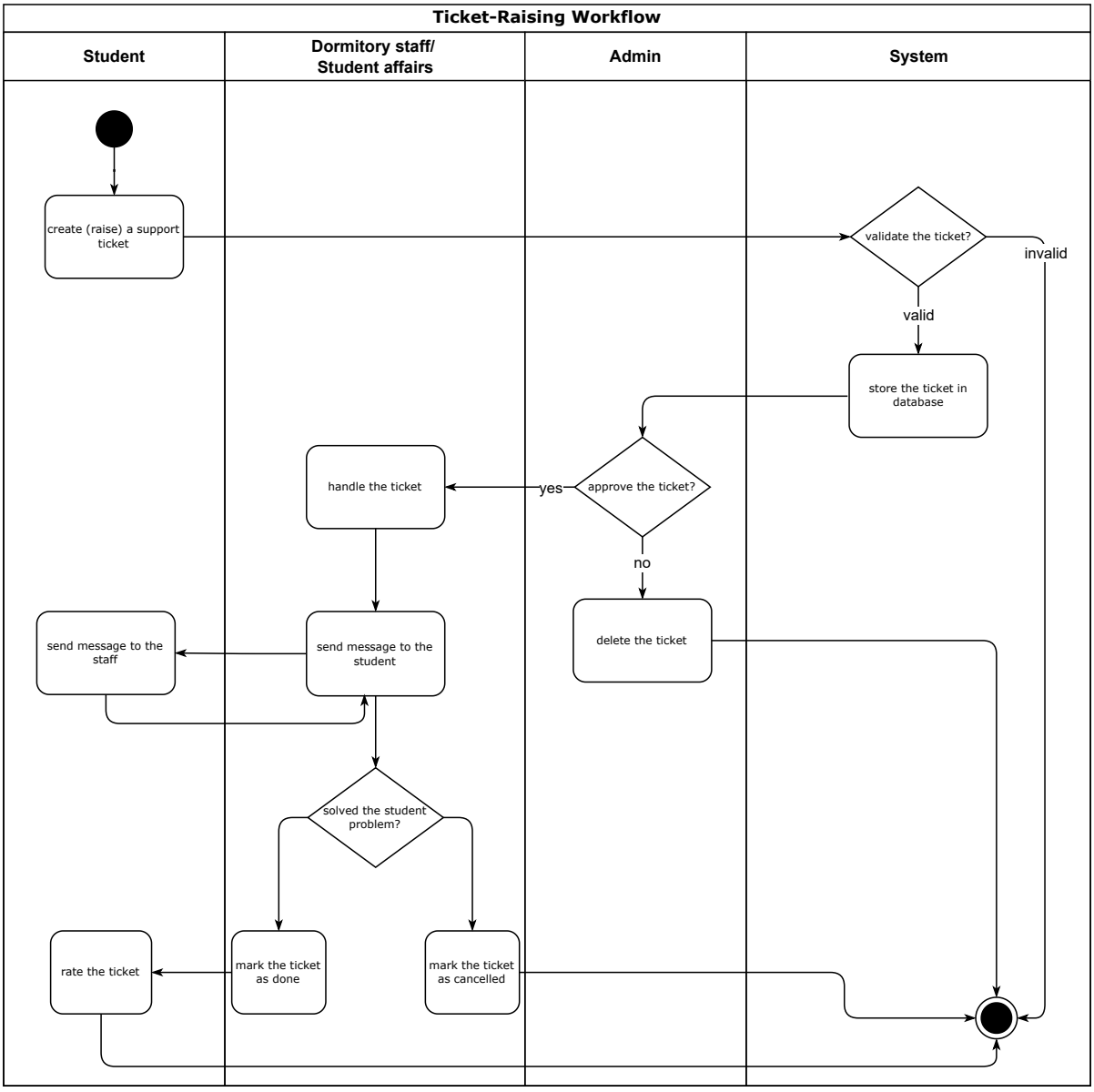


Figure 14: Ticket-Raising Process Workflow

3.5 Database Design

3.5.1 ER Diagram

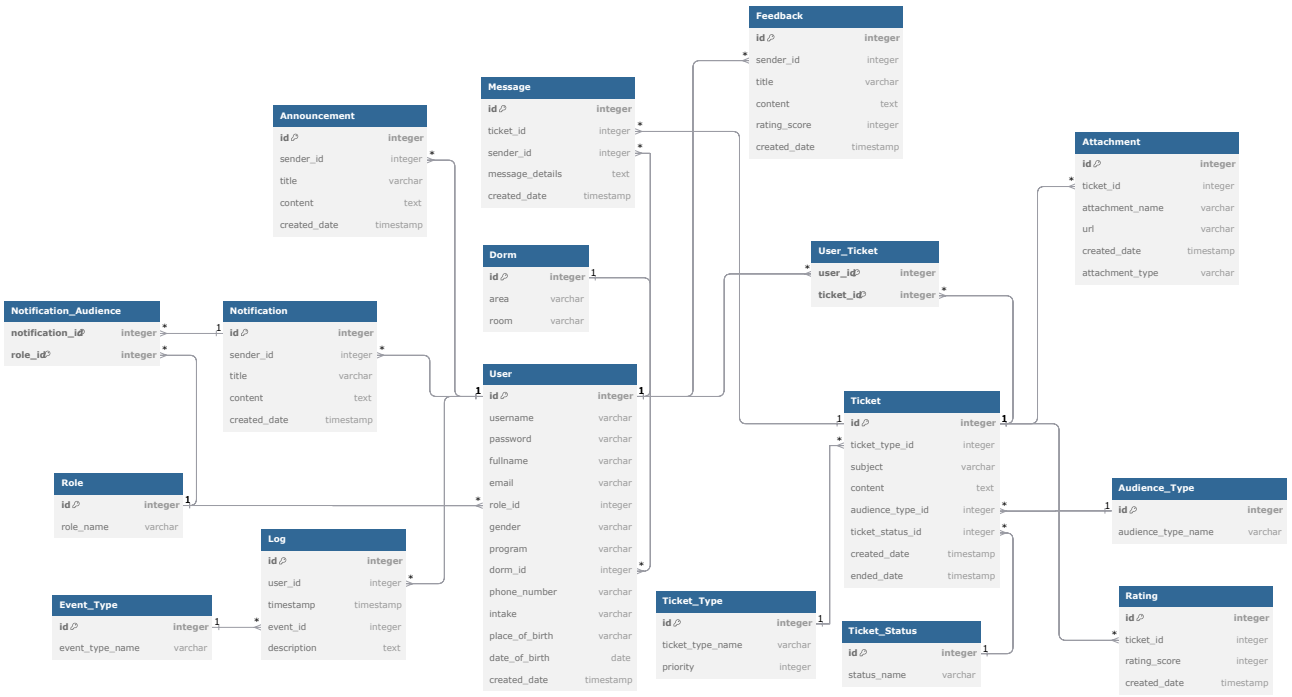


Figure 15: ER Diagram

3.5.2 User Entity

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

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

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

Table 7: User Entity

3.5.3 Ticket Entity

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

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

Table 8: Ticket Entity

3.5.4 User_Ticket Relationship

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

Table 9: User_Ticket Relationship

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

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

3.5.5 Ticket_Type Entity

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

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

Table 10: Ticket_Type Entity

3.5.6 Ticket_Status Entity

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

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

Table 11: Ticket_Status Entity

3.5.7 Audience_Type Entity

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

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

Table 12: Audience_Type Entity

3.5.8 Attachment Entity

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

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

Table 13: Attachment Entity

3.5.9 Rating Entity

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

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

Table 14: Rating Entity

3.5.10 Feedback Entity

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

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

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

Table 15: Feedback Entity

3.5.11 Message Entity

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

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

Table 16: Message Entity

3.5.12 Dorm Entity

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

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

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

Table 17: Dorm Entity

3.5.13 Announcement Entity

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

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

Table 18: Announcement Entity

3.5.14 Notification Entity

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

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

Table 19: Notification Entity

3.5.15 Role Entity

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

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

Table 20: Role Entity

3.5.16 Notification_Audience Relationship

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

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

Table 21: Notification_Audience Entity

3.5.17 Log Entity

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

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

Table 22: Log Entity

3.5.18 Event_Type Entity

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

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

Table 23: Event_Type Entity

3.6 System Architecture

3.6.1 Overview

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

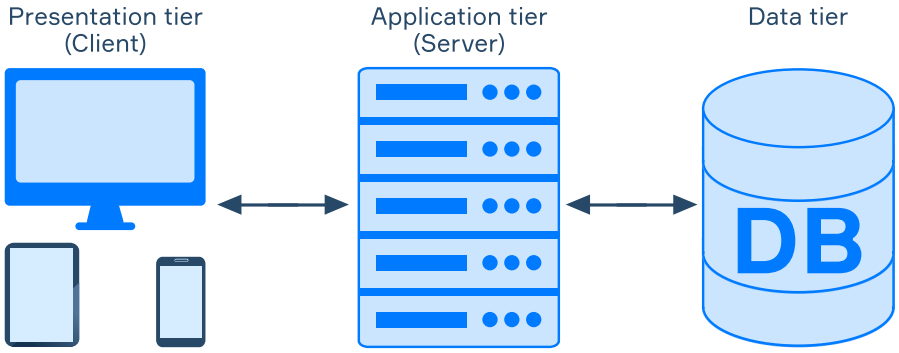


Figure 16: Three-tier Architecture[1]

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

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

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

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

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

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

3.6.2 3-Tier Architecture Implementation

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

Table 24: 3-Tier Architecture Implementation

3.7 API Design

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

API Design Principles:

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

Data format:

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

Security Measures:

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

3.7.1 Authentication/Authorization API

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

Table 25: Authentication/Authorization API

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

POST /auth/login

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

Code snippet 2: Request body of Login API

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

Code snippet 3: Response of Login API

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

POST /auth/refresh-token

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

Code snippet 4: Response of Refresh Token API

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

POST /auth/verify-refreshToken

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

Code snippet 5: Response of Verify Refresh Token API

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

`POST /auth/reset-password`

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

Code snippet 6: Response of Reset Password API

3.7.2 User API

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

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

Table 26: User API

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

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

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

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

Code snippet 7: User Scheme

3.7.3 Ticket API

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

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

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

Table 27: Ticket API

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

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

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

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

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

Code snippet 8: Ticket Scheme

3.7.4 Dormitory API

Method	Endpoint	Header	Request Body	Response / Description
GET	/api/v1/dorms	Authorization: Bearer <token>	None	List of JSON objects which contain all dormitory details

Method	Endpoint	Header	Request Body	Response / Description
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>	JSON object which contains dormitory area and room	Create a new dormitory with supplied data

Table 28: Dormitory API

3.8 UI/UX Design

4 System Implementation

5 User Manual

5.1 Sign in

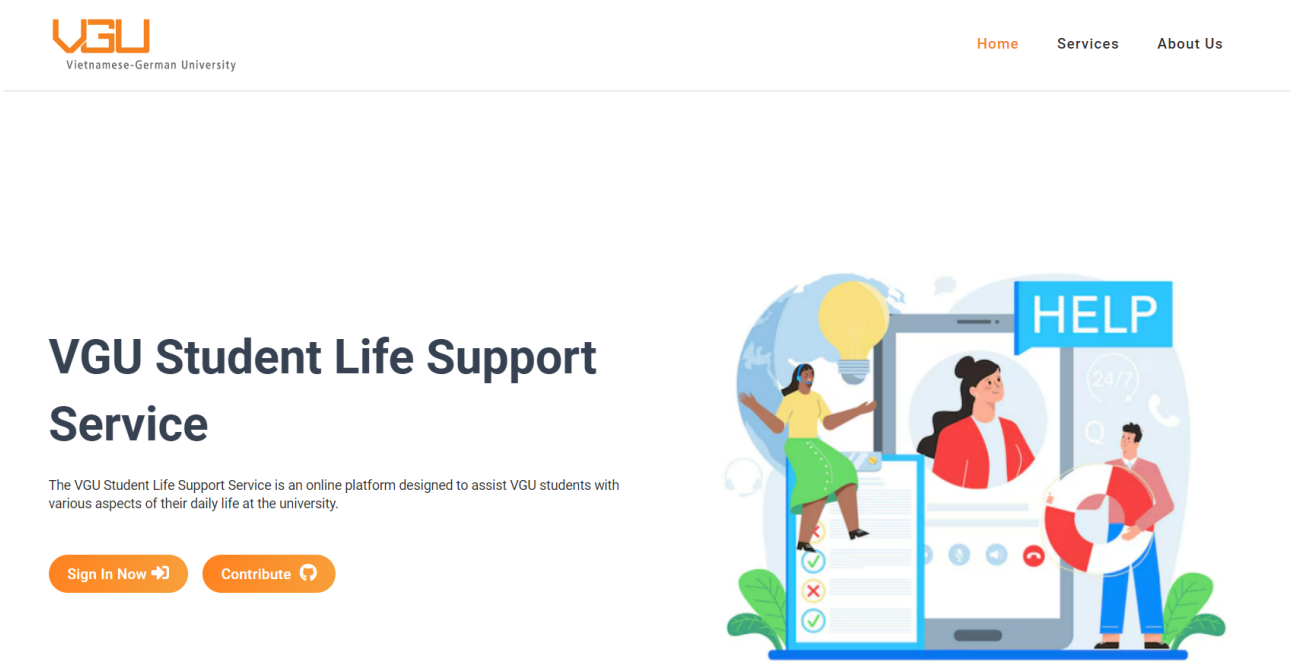


Figure 17: Landing page

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

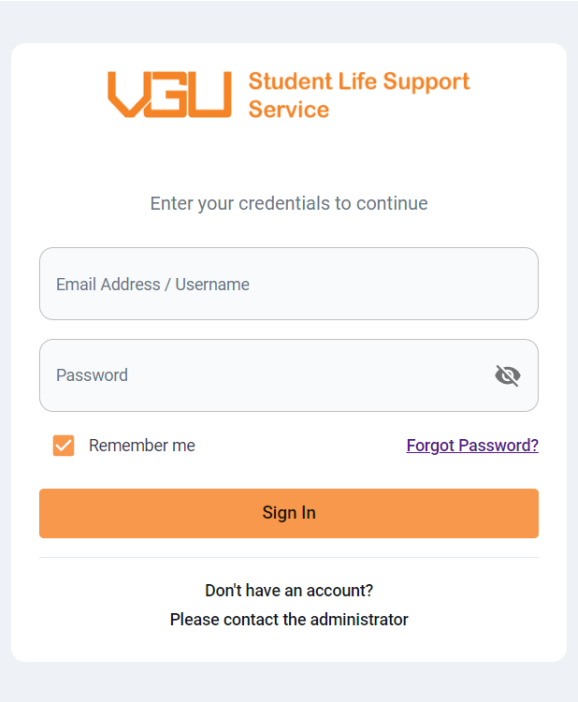


Figure 18: Sign in Form

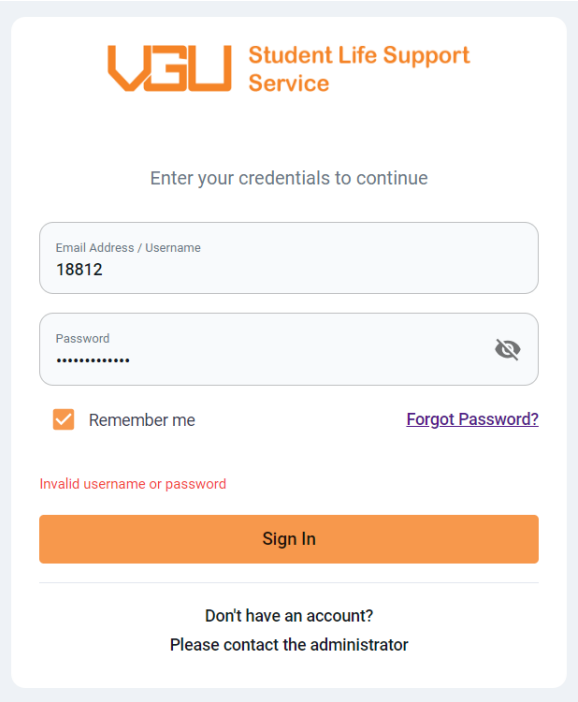


Figure 19: Failed Sign in attempt

To access the service, input your username (or email) and password, then click on the sign-in button (refer to Figure 18). If you provide incorrect credentials, a warning message will appear, indicating that you need to correct either your username or password (Figure 19).

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 18). Subsequently, provide your email address in the 'Forgot Password' form and click on 'Reset Password' (see Figure 20).

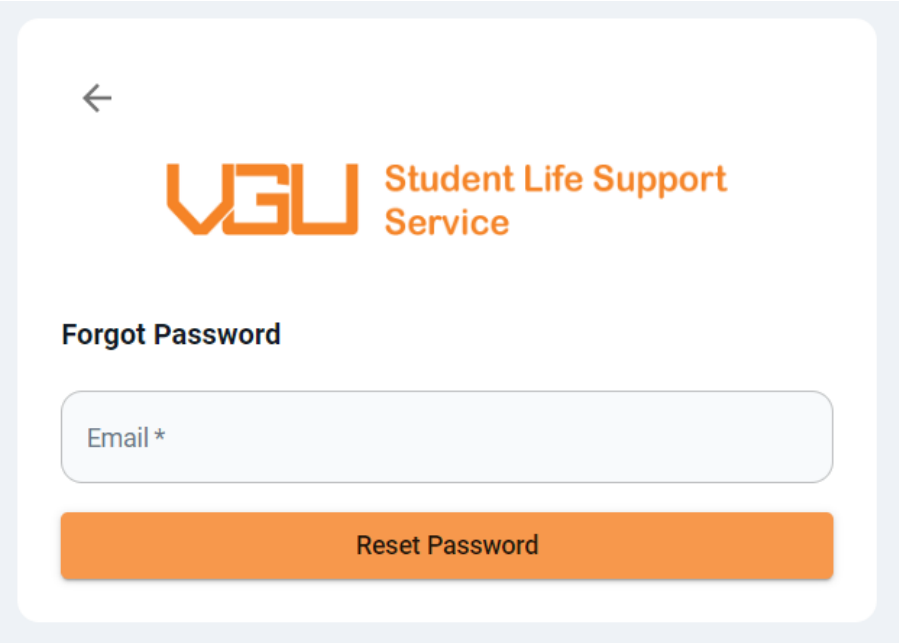


Figure 20: Reset password form

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 21, 23). Conversely, if the email is not found in the system, a failure notification will indicate that the email does not exist (Figure 22).

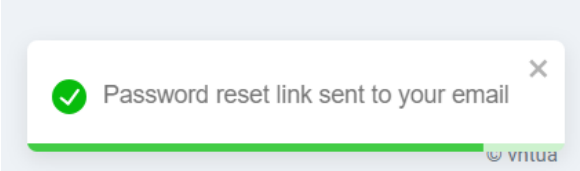


Figure 21: Reset password successfully

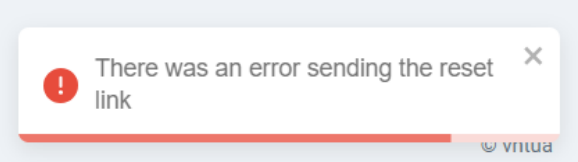
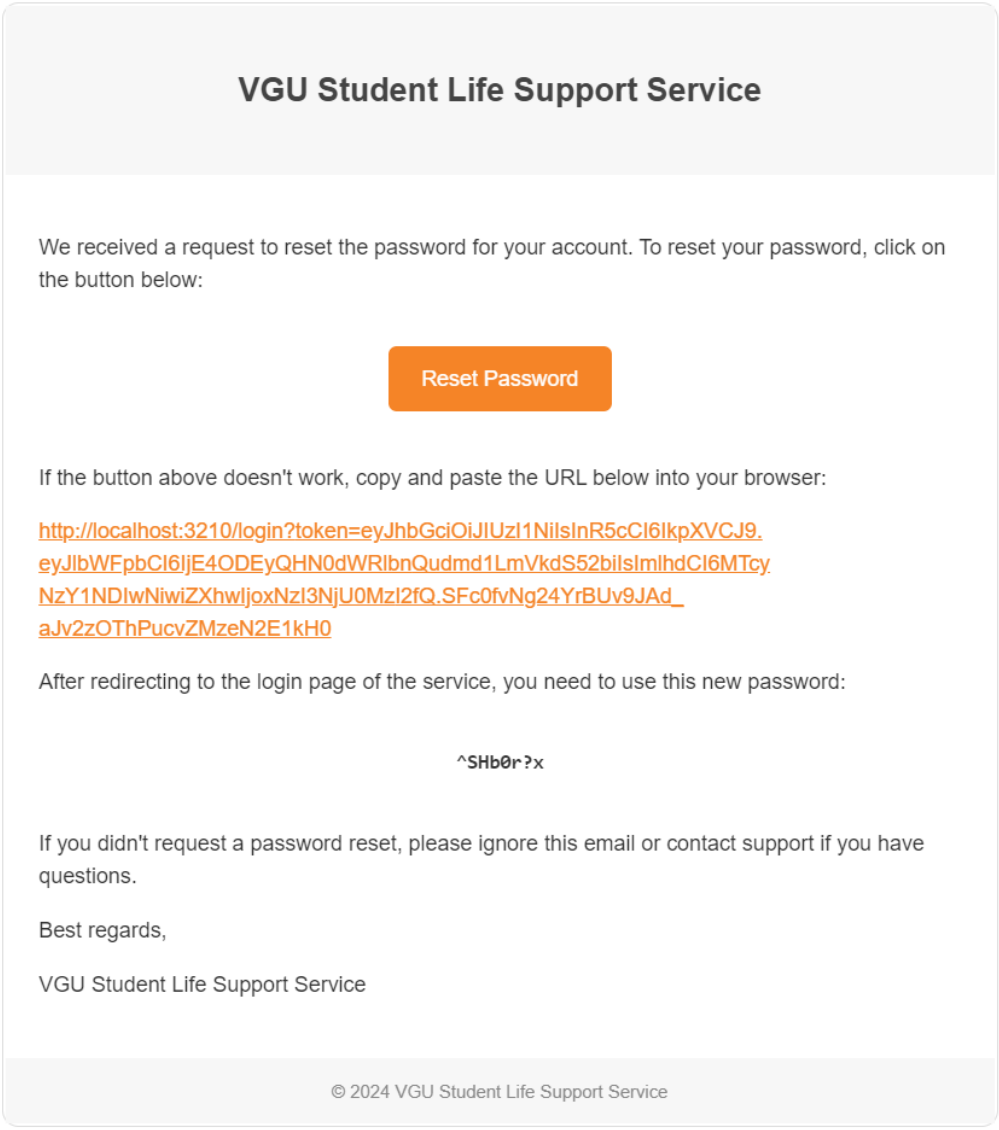


Figure 22: Reset password failed



VGU Student Life Support Service

We received a request to reset the password for your account. To reset your password, click on the button below:

Reset Password

If the button above doesn't work, copy and paste the URL below into your browser:

http://localhost:3210/login?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6IjE4ODEyQHN0dWRlbnQudmd1LmVkdS52biIsImlhdCI6MTcyNzY1NDIwNiwiZXhwIjoxNzI3NjU0MzI2fQ.SFc0fvNg24YrBUv9JAd_aJv2zOThPucvZMzeN2E1kH0

After redirecting to the login page of the service, you need to use this new password:

^SHb0r?x

If you didn't request a password reset, please ignore this email or contact support if you have questions.

Best regards,

VGU Student Life Support Service

© 2024 VGU Student Life Support Service

Figure 23: Reset password email instructions

6 Conclusion and Future Work

References

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

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

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

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

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